

Machine Learning for Mechanical Engineering

Mark Fuge

2025-09-17

Table of contents

Preface	8
Pre-Requisites	9
How This Book is Structured	10
Elements within the Book	11
Runnable and Interactive Code Elements	11
Callouts with Notes and Experiments	11
Contributors	13
 I Foundational Skills	 14
1 Reviewing Supervised Linear Models	16
1.1 What is a Linear Model?	17
1.2 Regularization and Controlling Model Complexity	22
1.3 Examples of Other Commonly used Loss Functions in Linear Models	28
1.4 Loss Functions for Regression	29
1.5 Loss Functions for Linear Classification	30
1.6 Penalty Functions: Example of How L_p Penalty Changes Loss Optima	34
1.7 Summary of How to Select a Loss or Penalty function	42
1.8 Exercises	44
 2 Evaluating Machine Learning Models	 52
2.1 Getting an Unbiased Estimate of Out-of-Sample Performance	53
2.2 Choosing a Cross-Validation Strategy	58
2.3 Finding the Optimal Hyper-parameters	61
2.4 What if we have more than one variable?	62
2.5 Optimization	63
2.6 Randomized Search	87
2.7 Global Bayesian Optimization	111
2.8 The Curse of Dimensionality	115
2.9 Exercises	116

3	Introduction to Gradient Descent	121
3.1	How Large Can the Step Size Be?	126
3.2	Stochastic Gradient Descent	128
3.3	Existing Implementations	131
3.4	Advanced Techniques	131
3.5	Exercises	136
4	Review of Linear Unsupervised Learning	141
4.1	Principal Component Analysis	143
4.2	Sparse PCA	153
4.3	Dictionary Learning	155
4.4	Non-Negative Matrix Factorization (NMF)	157
4.5	Model Comparison	160
4.6	Exercises	162
5	Taking Derivatives with Automatic Differentiation	168
5.1	Automatic Differentiation	169
5.2	PyTorch Autograd Example with Simple Function	174
5.3	PyTorch Autograd Example with Optimization	175
5.4	Demonstration of AD on Verlet Integration	179
6	Measuring Distribution Distances	189
6.1	A Toy Example: Comparing Two Gaussians	189
6.2	Kullback–Leibler Divergence	190
6.3	Jensen–Shannon Divergence	193
6.4	Maximum Mean Discrepancy (MMD)	194
6.5	Optimal Transport (Wasserstein-2 Distance)	195
6.6	Information Geometry vs. Optimal Transport	197
6.7	Wrap-Up	198
7	Introduction to Inference	199
7.1	Motivation and Toy Problem Definition	199
7.2	Linear Bayesian Regression	200
7.3	A Small Neural Network (Nonlinear Model)	203
7.4	MCMC Approximation	206
7.5	Variational Inference	219
7.6	Possible Additional Experiments	238
7.7	Summary and the bridge to VAEs	239
8	Introduction to Probabilistic Programming	242
8.1	Learning Objectives	242
8.2	Problem Setup: Inferring Material Properties from Stress-Strain Data	242
8.3	Bayesian Linear Regression	243

8.4	Bayesian Material Model Identification (Ramberg-Osgood)	246
8.5	Summary and Next Steps	250
II	Foundations of Generative Models	251
9	Review of Neural Networks	253
9.1	Example of PyTorch SGD for Linear Regression and a Simple Feed-Forward Network	253
9.2	Unsupervised Learning using Autoencoders	259
9.3	Least Volume Regularization	268
9.4	How might we select the right order in an Autoencoder? Building the case for Least Volume Analysis (LVA)	274
10	Introduction to Push-Forward Generative Models – Generative Adversarial Networks (GANs)	281
10.1	What are Generative Models really doing?	281
10.2	Basic Deterministic Push-Forward Models: From Latent Space to Data Space	282
10.3	Toy Dataset: Ring of Gaussians	283
10.4	What is a Generative Adversarial Network (GAN)?	283
10.5	A Simple Generative Adversarial Network (GAN)	284
10.6	Basic Training Loop	285
10.7	Conditional Generation Examples	289
10.8	Summary and Next Steps:	297
11	GAN Training Pitfalls	298
11.1	Experiment 1: When the Discriminator Overpowers the Generator	299
11.2	Experiment 2: When the Generator Overpowers the Discriminator	301
11.3	Experiment 3: Effect of Discriminator Spectral Normalization	302
11.4	Summary and Next Steps	304
12	Optimal Transport for Generative Models	306
12.1	What is Optimal Transport?	306
12.2	Building an Entropic OT GAN	310
12.3	Summary and Next Steps	315
13	Variational Autoencoders (VAEs)	317
13.1	Learning Objectives	317
13.2	What is a Variational Autoencoder?	317
13.3	Building a Simple VAE	319
13.4	Exploring the Learned Latent Space	326
13.5	Conditional Generation Examples	328
13.6	Comparing VAEs to GANs and OT Models	336
13.7	Summary and Looking Forward	337

14 Normalizing Flows	339
14.1 Learning Objectives	339
14.2 VAEs vs. Normalizing Flows	339
14.3 Key Idea: The Change-of-Variables Formula	340
14.4 Constructing a Flow with Coupling Layers using RealNVP	347
14.5 Conditional Generation Examples	353
14.6 Advantages and Limitations of Normalizing Flows	363
14.7 Summary and Looking Ahead	364
15 From Discrete Transformations to Continuous Flows	365
15.1 Learning Objectives	365
15.2 From Finite Jacobians to Differential Updates	365
15.3 Example: Divergence-Free Rotation	369
15.4 Example: Contracting Divergence Controls Density Growth	369
15.5 Example 1D Continuous Normalizing Flow with Neural ODEs	370
15.6 Summary and Next Steps	374
16 From Continuous Flows to Score Matching	375
16.1 Learning Objectives	375
16.2 Sampling by “Climbing” the Probability Landscape	376
16.3 The First Hurdle: How to Learn the Score?	376
16.4 Solution 1: Hyvärinen’s Score Matching	377
16.5 Solution 2: Denoising Score Matching (DSM)	377
16.6 Example: Learning a Score Model for a 2D Ring of Gaussians	379
16.7 How do we use the learned Score Function?	382
16.8 The Final Bridge to Diffusion Models	385
16.9 Summary and Next Steps	385
17 From Score Matching to Diffusion Models	386
17.1 Learning Objectives	386
17.2 Why Go Beyond Single-Noise Score Matching?	387
17.3 The Forward Diffusion Process $q(x_t x_0)$	387
17.4 The Reverse Process and Multi-Scale Score Estimation	390
17.5 How Training Works (Denoising Score Matching Perspective)	392
17.6 Generating Samples via Reverse Sampling	396
17.7 Connection to Langevin Dynamics	399
17.8 Conditional Generation Examples	399
17.9 Summary and Next Steps	405
18 Flow Matching	407
18.1 Learning Objectives	407
18.2 The Noise-Free Limit	408
18.3 Interpolating a Path	408

18.4	Training via Velocity Regression	410
18.5	Generating Samples by Solving the ODE	416
18.6	Conditional Generation with Flow Matching	417
18.7	Summary and Next Steps	428
19	Latent Generative Models	430
19.1	Learning Objectives	430
19.2	Setup and Data Generation	431
19.3	Initial Baselines: Generative Models in the Original Data Space	431
19.4	Build a Least Volume Autoencoder to Learn the Latent Space	433
19.5	Retrain the Generative Models on the Latent Space	435
19.6	Sampling and Decoding to Compare to Original Data	436
19.7	Summary	437
19.8	Review of Generative Models Covered So Far and Next Steps	437
20	Introduction to Deep Reinforcement Learning	439
20.1	Learning Objectives	439
20.2	Deep Q-Network (DQN)	440
20.3	Policy Gradients (REINFORCE)	446
20.4	Vanilla Actor-Critic (AC)	451
20.5	Proximal Policy Optimization (PPO)	457
20.6	Summary and Next Steps	464
21	Introduction to Transformers	466
21.1	Learning Objectives	466
21.2	Example: 2D Shapes — Fixed-Size Set Classification	466
21.3	2D Shapes — Variable-Length Sets with Noise and Shuffling	469
21.4	What Is a Transformer?	471
21.5	3D Shapes — Synthetic Point Clouds	487
21.6	Summary	494
III	Engineering-Specific Considerations	495
22	Active and Semi-Supervised Learning	497
22.1	Learning Objectives	497
22.2	Active Learning	497
22.3	Types of Active Learning Strategies	499
22.4	Uncertainty Sampling Example: Gaussian Processes	500
22.5	Uncertainty Sampling of Classification Boundaries	530
22.6	Semi-Supervised Learning	549
22.7	Summary	562
	References	563

Appendices	564
A Helpful Tooling for Working with and Debugging Machine Learning Models	564
A.1 TL;DR Checklist	564
A.2 Reproducible Runs	565
A.3 Code Management	567
A.4 Debugging	570
A.5 A Practical Example	573
B Exercise Solutions	582
C Review of Matrices and the Singular Value Decomposition	583
C.1 The Singular Value Decomposition	586
C.2 What about different dimensions?	589
D Reviewing Mathematical and Computational Foundations for Machine Learning	594
D.1 Random Variables, Monte Carlo, and Concentration Inequalities	594
D.2 Change of Variables & Jacobians	597
D.3 Linear Algebra & Spectral Norms	602
D.4 Differentiation & Curvature	607
D.5 Marginalization of Probability Distributions	611

Preface

This is an open textbook to accompany my course notes for “Machine Learning for Mechanical Engineering” at ETH Zürich in the Department of Mechanical and Process Engineering (D-MAVT).

Pre-Requisites

This course was designed as an introductory course in Machine Learning (ML) focused on applications within Mechanical Engineering. However, it is also designed as follow on course from ETHZ's "Stochastics and Machine Learning" course which is required of all D-MAVT students, and therefore, I assume familiarity with the topics covered in that course.

This book is also designed with several executable code sections that allow you to interactively experiment with the course materials using Python. For this, we have tried to keep the required packages as close as possible to the default Anaconda Package structure in Python 3.12, but will place any additional requirements in a requirements file to enable easy creation of Virtual Environments if you want to run the code in this book.

How This Book is Structured

This book is designed to map to a one semester course at the advanced Undergraduate level (although early Masters students may still find it useful), and essentially features three parts:

1. **Part 1: Foundational Skills** – These chapters cover skills that cut across different ML approaches and that should serve you well in the future regardless of which models are “state-of-the-art” at the time. We will revisit these skills throughout the course, but I wanted to separate them into distinct chapters to (1) highlight/elevate their critical importance as independent skills worthy of your learning, and (2) enable a certain amount of depth and self-learning on these skills beyond what we can draw out during individual course modules or later chapters.
2. **Part 2: Model-Specific Approaches** – These chapters look more like a “traditional” textbook, in the sense that they cover individual techniques or concepts that I thought were maximally relevant to Mechanical Engineers at the time that I wrote the book. These chapters will be most immediately useful in terms of getting up to speed with specific types of models, but they are also the most likely to become out of date quickly as newer/better models are invented.
3. **Part 3: Engineering-Specific Considerations** – These chapters deal with issues that are particularly prevalent in Mechanical Engineering contexts and may cut across specific models mentioned in Part 2. These are likely to persist over time, even as new models or approaches are invented, although they will likely get easier to address as research fields expand.
4. **Appendix: Useful Background Information** – I will reserve the Appendix for useful background knowledge that either should have been covered in prior courses, or which is of an implementation nature beyond what is practical to include the earlier chapters. I anticipate that this part of the book will initially be very empty, but that I will add to it as needed over future course semesters.

Elements within the Book

Runnable and Interactive Code Elements

Many of the elements of this book involve runnable code elements in Python, or interactive experiments that you can conduct or visualize while you are reading. These will appear largely static in the rendered book, since some may have long run-times, but you can download and run the corresponding notebook in a browser-based environment (*e.g.*, CoLab) to be able to run the experiments. Most experiments in the book are designed to be run in class in the span of a few minutes, so browser-based environments should work well for this.

Callouts with Notes and Experiments

Through the book, I will occasionally use callouts like the one demonstrated below for a couple of use cases:

Example Collapsed Note, *e.g.*, for a Derivation

Here is what a note with a collapsable block structure will look like. We will also use this type of structure to hide details of long derivations, either so you can work them out on your own first, or because the details are not immediately central to following what comes next in the chapter or lecture.

Example Experiment

Here is an example where we might pose an experiment or task for you to do on your own or in class. Often wrestling with these experiments is a good way to build intuitive understanding and integrate some of the material into practice. When you see this type of callout, you should pause and do the exercise at this point in the chapter, rather than reading ahead, as it may build intuition which is useful later in the chapter.

Example Exercise (~5 min)

Here is what an end-of-chapter exercise looks like. Each chapter in Part 1 closes with a short set of these. They reuse the chapter's own data and code, carry a rough time

estimate in the title, and range from quick recall checks to small open-ended modeling tasks. Worked solutions are collected in Appendix [B](#), but you will get far more out of the exercises if you attempt them before looking.

Contributors

I (Mark Fuge) originally created many of the foundational notebooks included in this book over around 8 years of teaching, mostly as separate Jupyter notebooks I used in my class, and which I now am compiling and extending here in a more coherent and organized form. At the request of others, I have decided to make this content open and extendable by others under a suitable license ([CC BY-NC-SA 4.0](#)). In addition, below I acknowledge the help and additions of others who have contributed elements to these notes:

- Matthew Keeler: Helped reformat many of my existing notebooks into a format that made this online version possible.
- Dr. Florian Felten: Created Appendix [A](#).

Part I

Foundational Skills

This part of the book covers useful foundational skills that should serve you well regardless of which model is State-of-the-Art at the time. We will use them throughout the book, but it is helpful to have some material and exercises together in a cohesive whole.

1 Reviewing Supervised Linear Models

We will start by reviewing Linear Models, since they are likely already familiar to you from prior coursework, are widely used, and will serve as a useful (if simple) launching off point for later discussions of more advanced techniques. So, while some of what we will explore in this section might seem pretty basic at first glance, do not let it's simplicity fool you, as we will revisit similar concepts throughout the rest of the notes, and these concepts will help you form a strong foundation that will serve us well once things get more complex.

This chapter will do this in three parts:

1. Review the basic setup of a linear model, including notation and how adding features to the linear model can make it behave in ways that would appear “non-linear”.
2. Review the concept of Regularization — which will essentially penalize model complexity to avoid overfitting.
3. Discuss different types of commonly used loss functions for linear models, demonstrating their differences and possible use cases.

With this as a baseline model, the next chapter will review the concept of Cross-Validation and how we evaluate whether an Machine Learning model is “good”, and then the subsequent chapter will review (Stochastic) Gradient Descent, in the Linear Model context.

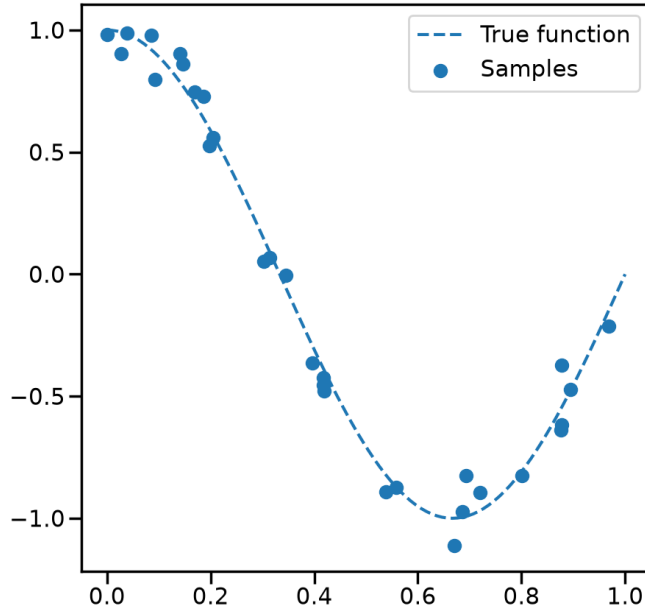
Learning Objectives

By the end of this chapter, you should be able to:

1. Explain what makes a model *linear*, and recognize that polynomial and other feature expansions still give linear models.
2. Describe how a regularization penalty controls model complexity, and connect the L_1 , L_2 and general L_p norms to the shape of that penalty.
3. Choose a regression loss (squared, absolute, Huber, epsilon-insensitive) for a given noise or outlier situation and explain the consequences of that choice.
4. Choose a classification loss (hinge, logistic, and the others shown here) and describe how it shapes the decision boundary and treats points near it.
5. Predict how adding a penalty term moves the optimum of a training objective.

1.1 What is a Linear Model?

Let's start by trying to fit a model to the (admittedly simple) dataset below, where I have just sampled some (noisy) points from a periodic function:



If we wanted to fit a line to this data, we would use what is called a linear model:

$$y = w_0 + w_1 \cdot x$$

where w_0 is the intercept and w_1 is the slope of the line. We can write this more compactly using vector notation as:

$$y = \mathbf{w}^T \mathbf{x}$$

where $\mathbf{w}$ is the weight vector $[w_0, w_1]$ and $\mathbf{x}$ is the feature vector $[1, x]$. We can see here that taking the dot product between $\mathbf{w}$ and $\mathbf{x}$ is equivalent to the equation above. Importantly, even though the above equation represents a straight line with respect to $\mathbf{x}$, we are not limited to using linear models only for this. For example, we could make:

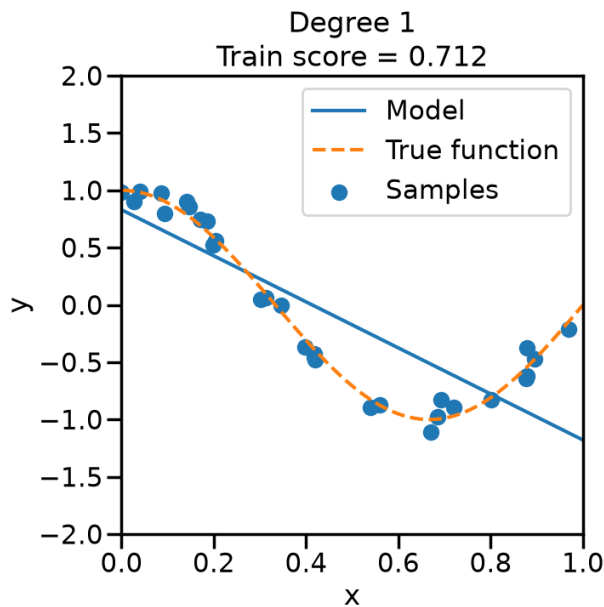
$$\mathbf{w} = [w_0, w_1, w_2, w_3]; \quad \mathbf{x} = [1, x, x^2, x^3]$$

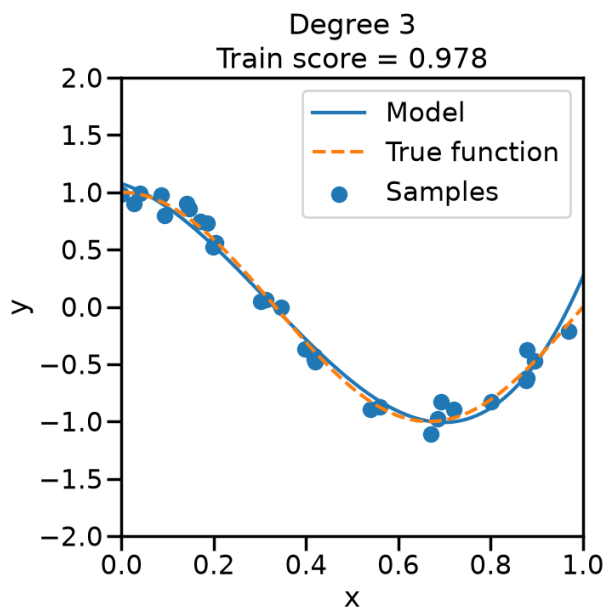
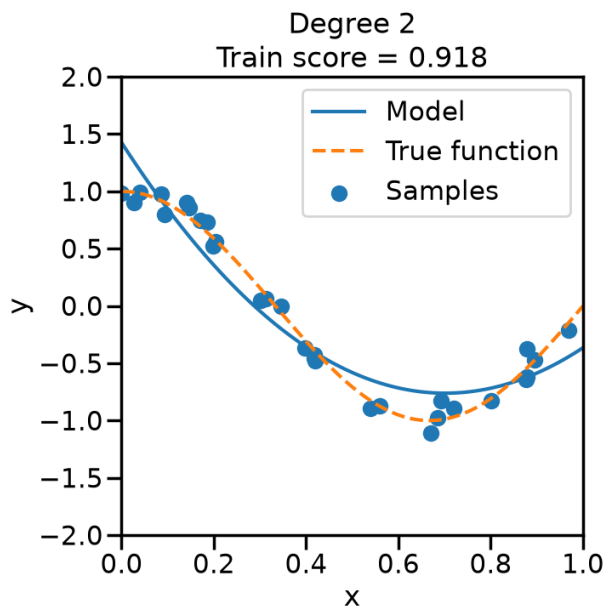
and in this way, we can model y as a cubic function of x , while $y = \mathbf{w}^T \mathbf{x}$ remains a “linear model”, since it is still linear with respect to the weights w . This is quite powerful, since by adding features (i.e., additional concatenated entries) to $\mathbf{x}$, we can fit functions that are apparently non-linear with respect to the original input variable x , but will possess many useful properties of linear models that we will discuss later (e.g., convexity with respect to w).

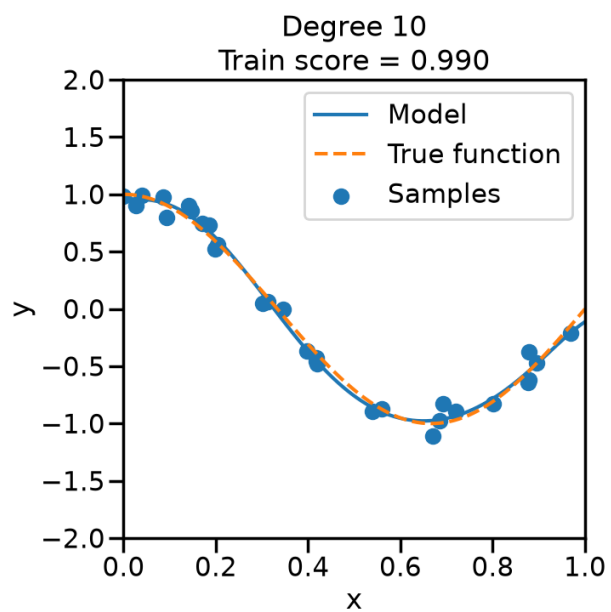
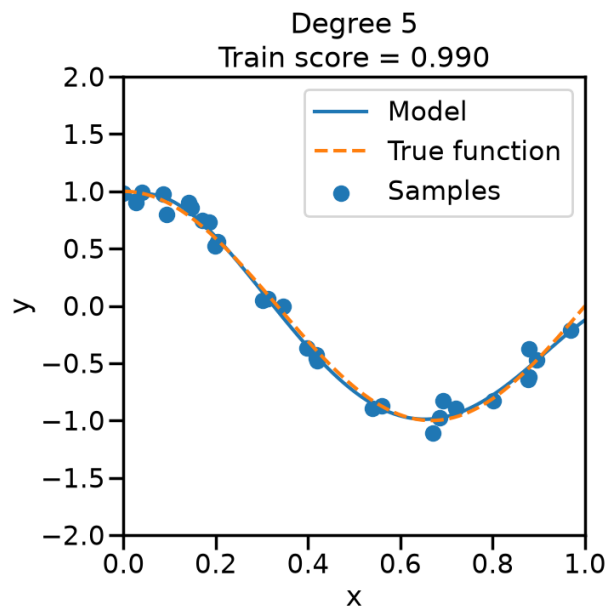
💡 Experiment: Effect of Polynomial Degree on Model Fit

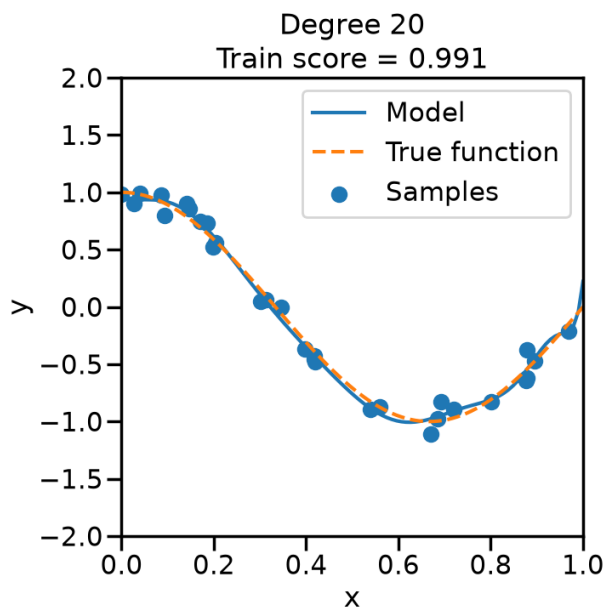
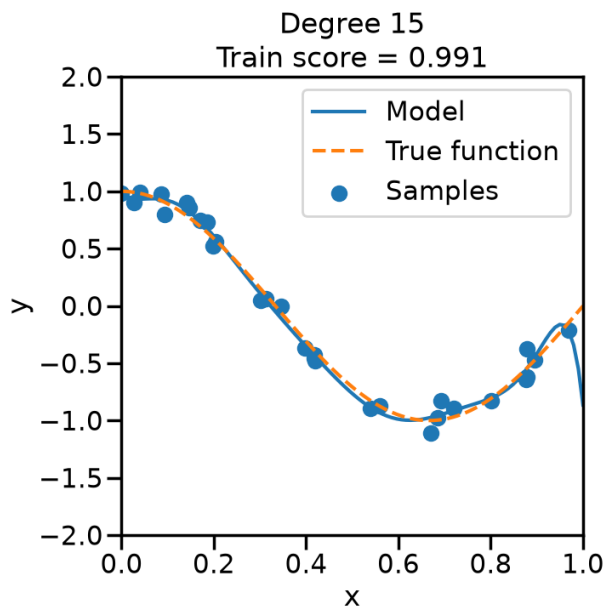
We can see this in the below experiment, where we fit progressively higher order polynomial functions to the above training data. In each case we are fitting a linear model, by virtue of appending additional polynomial features to x . When you do this experiment, ask yourself:

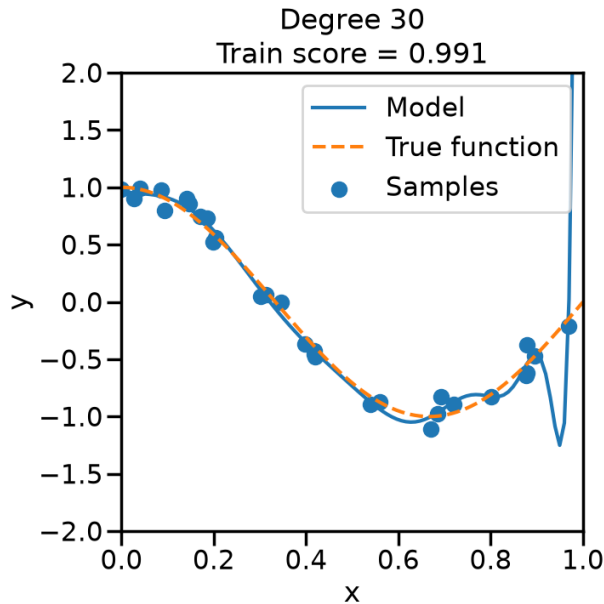
- How does the model fit change as we increase the polynomial degree?
- What happens to the training score (R^2) as we increase the polynomial degree?
- Does this increase in the training score reflect what you intuitively expect? Why or why not?











We can see that even though we are fitting a linear model every time, the behavior with respect to x is markedly non-linear. Moreover, we see some strange behavior as we increase the polynomial degree. What is going on here and why is it behaving in this way? To build some intuition, we can take a look at the learned weight coefficients, by accessing the `coef_` attribute of the fitted model:

```
array([-1.10e+00,  1.82e+01, -1.61e+02,  2.20e+02,  4.17e+02, -7.20e+02,
       -5.59e+02,  3.35e+02,  8.38e+02,  6.39e+02,  3.80e+01, -5.39e+02,
       -8.23e+02, -7.51e+02, -4.13e+02,  3.95e+01,  4.56e+02,  7.30e+02,
        8.09e+02,  6.95e+02,  4.28e+02,  7.07e+01, -3.02e+02, -6.17e+02,
       -8.11e+02, -8.32e+02, -6.46e+02, -2.33e+02,  4.12e+02,  1.28e+03])
```

What is going on here? To understand this, we need to understand something about how the model is optimizing error and how we might control this behavior.

1.2 Regularization and Controlling Model Complexity

In the base case above, all the model is trying to do is minimize the Mean Squared Error (MSE) with respect to the training error, and with a sufficient number of parameters (polynomial features, in this case), it becomes possible to always achieve perfect accuracy (i.e., fit every point) in the training data. Unfortunately, as we can see, fitting the training data perfectly does not necessarily lead to a model that generalizes well to new data. To help us control

this behavior, we can introduce the concept of **Regularization**, which is a way of penalizing overly complex models. Specifically, we can modify our cost function to be something like:

$$Cost = Loss(w, D) + \Omega(w)$$

where $\Omega(w)$ represents what we call a “Regularization” of the function or a “Penalty Term”. The purpose of $\Omega(w)$ is to help us prevent the (otherwise complex) model from being overly complicated, by penalizing this complexity. There are many ways to do this that we will see later on, but one common way to do this for linear models is to penalize the **total weight** that you allow all of the w_i to have. Specifically how one calculates this total weight turns out to matter a lot, and we shall see it return in later chapters and sections. But to get us started in un-packing how to do this, we first need to talk about what a **Norm** is, how it relates to Linear Regression weights, and how it helps us perform Regularization.

1.2.1 Norms and their relationship to Regularization

A Norm is a concept in mathematics that allows us to essentially measure length or size, typically of vectors. Any time you have tried to compute the distance between two points in space (say, by using the Pythagorean Theorem), or the magnitude of an applied Force vector, you have been using a Norm — most likely the Euclidean Norm or Euclidean Distance. For example, for a vector $\mathbf{x}$ with n dimensions, the Euclidean Norm looks like this:

$$||\mathbf{x}||_2 = \sqrt{x_1^2 + x_2^2 + \dots x_n^2}$$

If you have ever had to compute the total Force Magnitude given its x and y components (for example, in Statics class), you have used the Euclidean Norm to do so. In that context, it served to take multiple components of a Force aggregate them in such a way as to tell you something about the **total force** – by analogy, we will do the same thing here with linear regression, where each weight is like a component and we can use the Euclidean Norm to compute the total weight.

While Euclidean Norms may be quite useful or familiar to Engineers, it turns out that they are a special case of a much wider *family* of Norms called *p-norms*, which are defined as:

$$||\mathbf{x}||_p = (|x_1|^p + |x_2|^p + \dots + |x_n|^p)^{1/p} = \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}$$

Specifically, the Euclidean Norm is called the L2-norm, or sometimes just the 2-norm. To see why this is, just set $p = 2$ in the above, and note how it corresponds to the Euclidean Norm that we all know and love. So, by setting p to a number between 0 and ∞ , we can modify what the *total weight* means, and setting $p = 2$ is the setting which we are all most familiar with. To get a visual sense of how norms vary, see below, which visualizes a line of “circle”

of radius 1, but where the length of the line is determined by the p-norm. You will see that when $p=2$ this corresponds to what we are familiar with, but when p goes up or down things change.

[Interactive output: available in the online version of this chapter.]

For today, we will just focus on the L2-Norm, however we will revisit norms again later where we will see how changing the one we are using can have positive or negative effects in certain circumstances.

We will use the L2-Norm to help us penalize having linear regression models with really large weights, by essentially putting a cost on the total weight of the weight vector, where the total is measured by the L2-Norm. That is:

$$Cost = \sum_{n=1}^N ||y - w \cdot x||^2 + \alpha \cdot ||w||^2$$

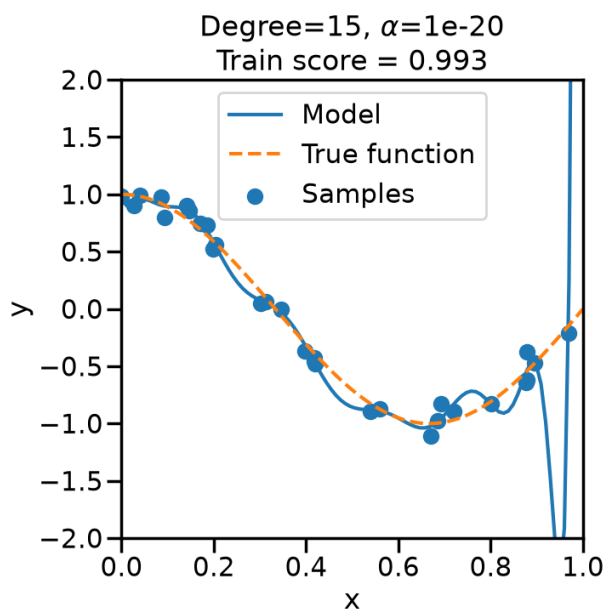
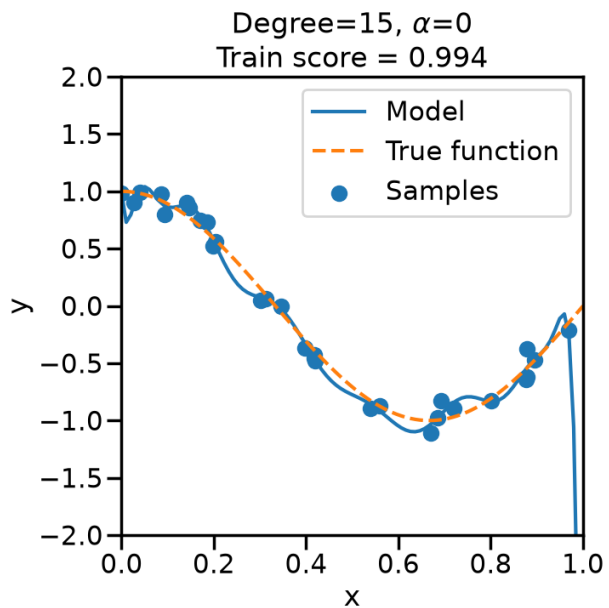
Where α is the price we pay for including more weight in the linear model. If it reduces our error cost enough to offset the cost of the increased weight, then that may be worth it to us. Otherwise, we would err on the side of using less weight overall.

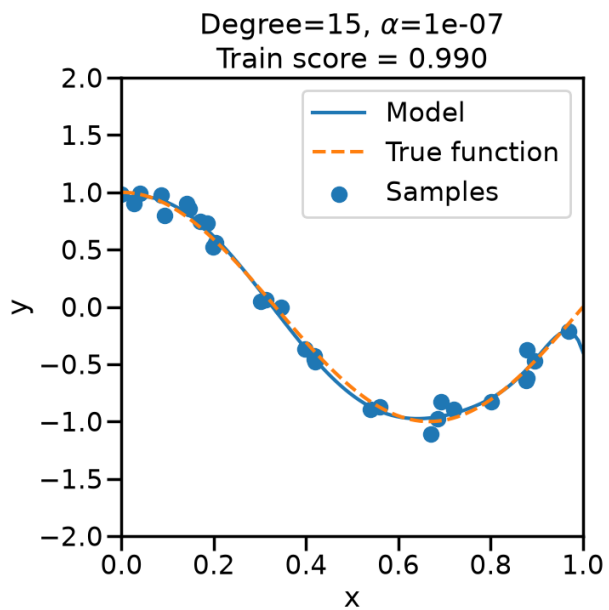
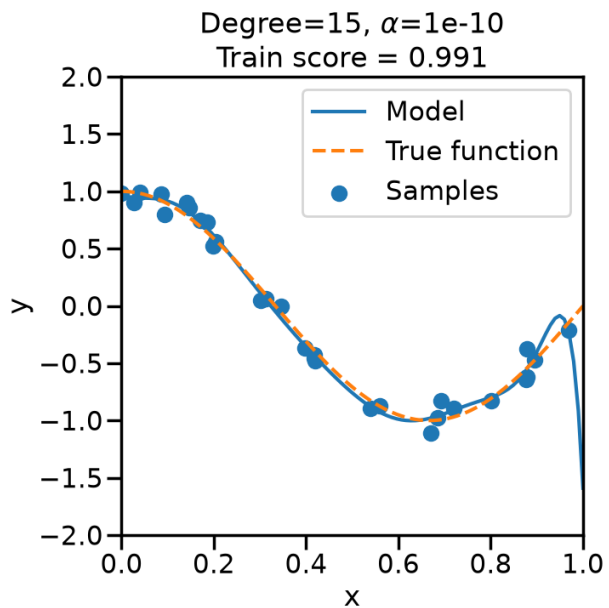
This Regularization (i.e., increasing α) essentially allows you to trade off bias and variance, as we will see below.

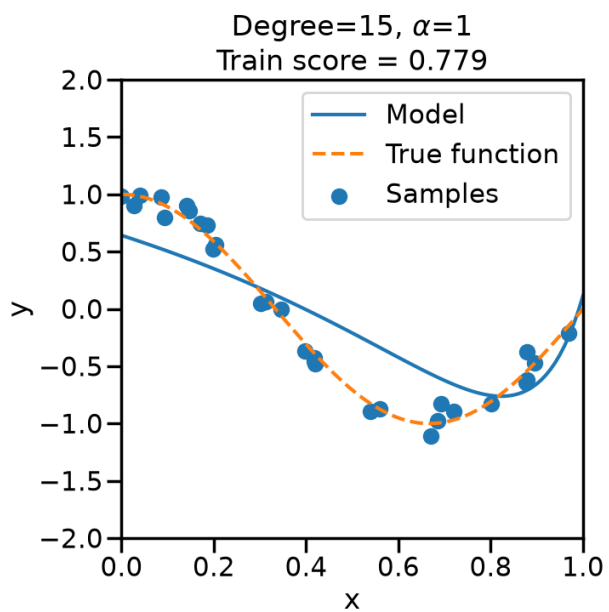
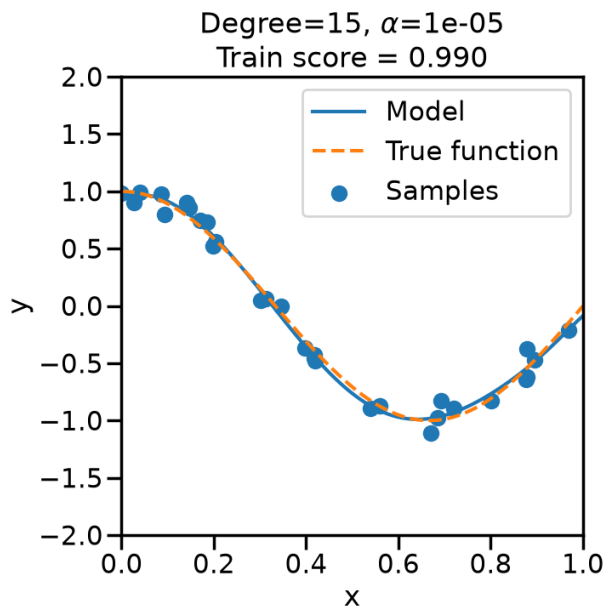
Experiment: Effect of Increasing Regularization Strength on Polynomial Fit

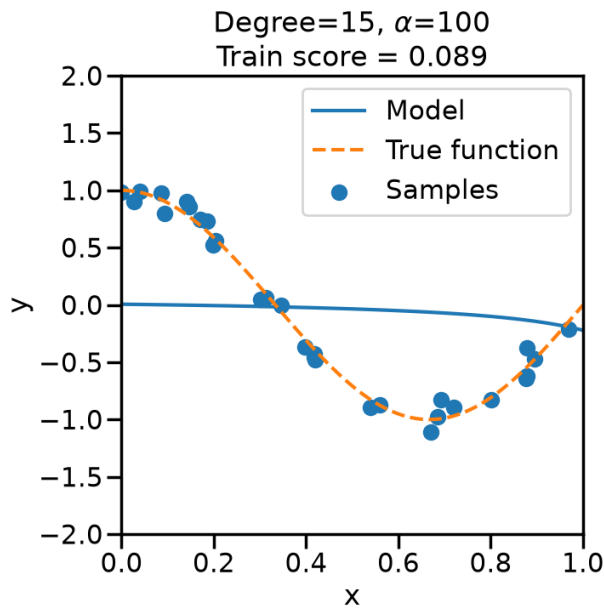
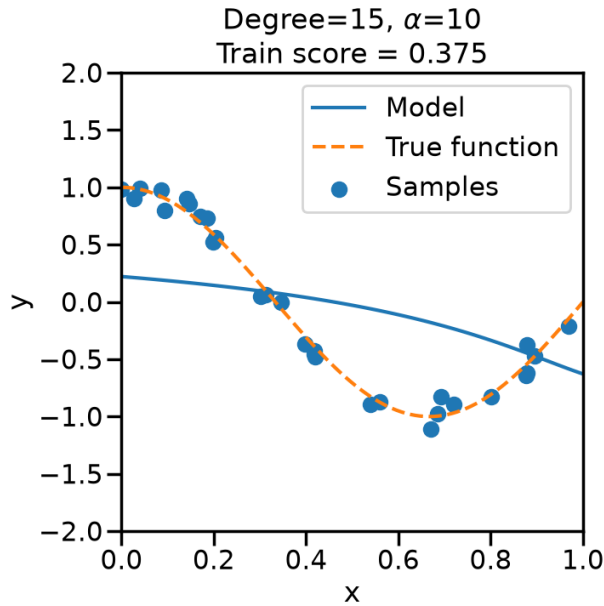
In the below experiment, we will increase the regularization strength (α) for a fixed 15-degree polynomial, and observe its effect on the overfitting problem before. Consider the following questions as you observe the experiment below:

- What happens when we set α to a low (close to zero) value?
- What happens when we set α to a high value?









1.3 Examples of Other Commonly used Loss Functions in Linear Models

Thus far we have been discussing Linear Models in their most familiar context — minimizing the Mean Squared Error (MSE) with respect to the training data, optionally with an L2

regularization on the weight vector:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \alpha \|\mathbf{w}\|_2^2$$

For now, let us ignore the regularization term, and just focus on the Loss term. Why should we minimize the squared error?¹ Why not the absolute error or other possible loss functions? Let's explore a few of those options and then see, in practice, how they affect the learned linear model.

1.4 Loss Functions for Regression

Beyond classical MSE, there are two main types of variants that are commonly used for regression problems:

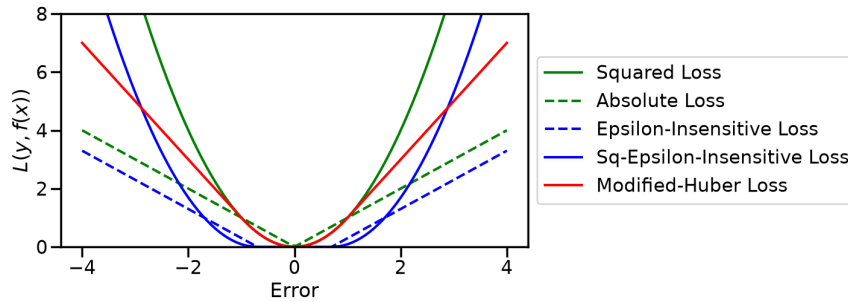
- Robust Loss Functions, that minimize the quadratic effect of the MSE loss for very large errors. These are typically used to make the trained model less sensitive to outliers in the training data.
- Epsilon-Insensitive Loss functions, that ignore errors that are sufficiently small (within an ϵ margin). These are typically used to incur some advantages in terms of sparsity in the learned model (for example, in Support Vector Methods, which we will see later).

In reality, these two variants can be combined in different ways, which we will see reflected below, but as a summary, these are:

- Absolute Loss: $|y - \hat{y}|$
- Huber Loss: A squared loss for small errors, and then transitioning to an absolute loss for large errors.
- Epsilon-Insensitive Loss: A loss that is zero for errors up to ϵ and then uses absolute loss for larger errors.
- Squared Epsilon-Insensitive Loss: Similar to Epsilon-Insensitive Loss, but uses squared loss for errors larger than ϵ .

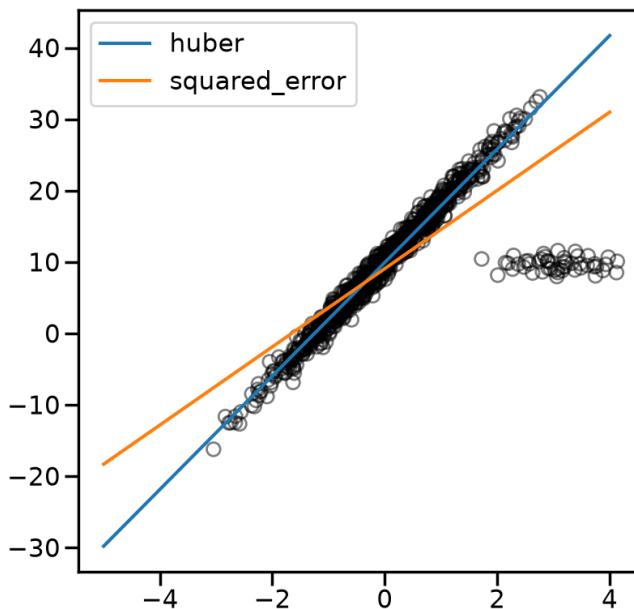
Of course, you could imagine more complex variants and combinations of these properties, but these capture the main properties and benefits that we will see below.

¹It turns out that there are good theoretical reasons for this, for example, that a Linear Model trained via an L2/MSE Loss is the Best Linear Unbiased Estimate (BLUE) of the Linear Model, according to the [Gauss-Markov Theorem](#), but, as we will see, there are other reasons to forgo these advantages.



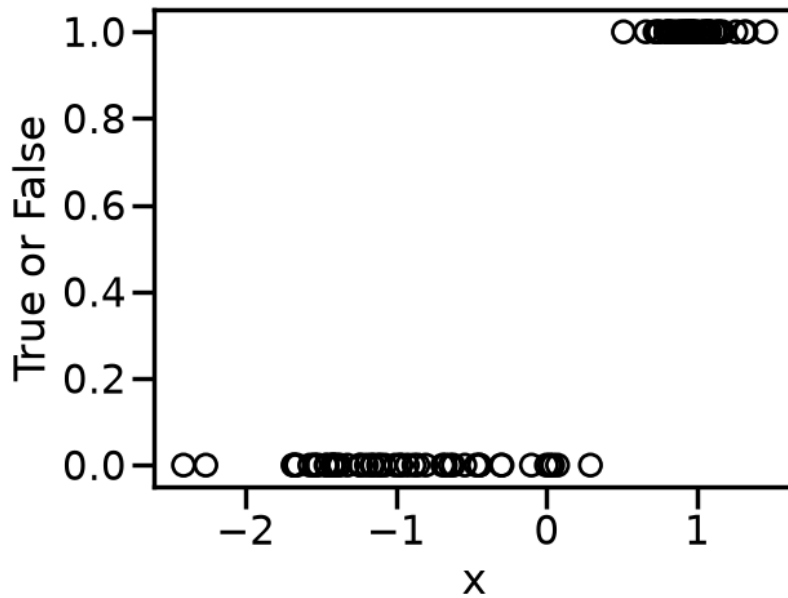
1.4.1 Handling Outliers using Robust Loss Functions

The below example demonstrates how applying a slightly different loss function (a Huber loss) to an otherwise identically formulated linear model can cause drastically different behavior in the trained model:

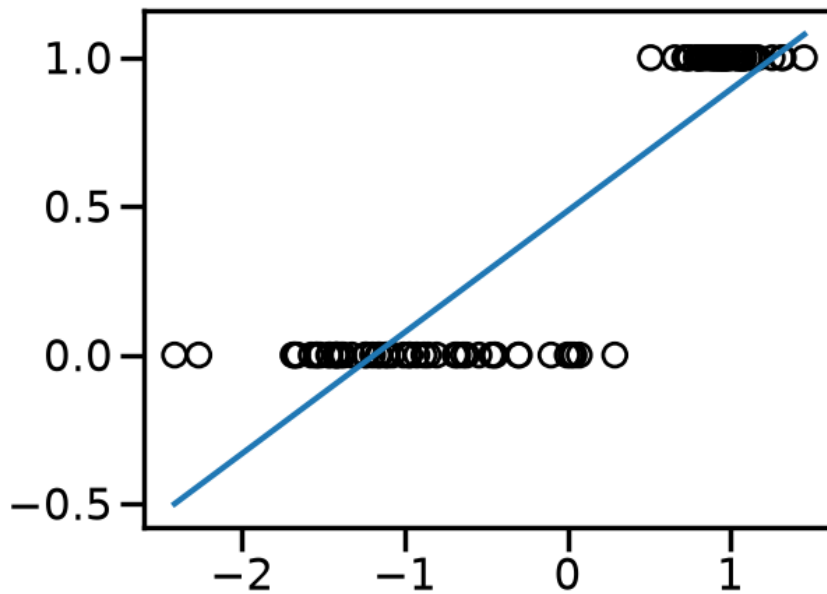


1.5 Loss Functions for Linear Classification

Thus far we have only discussed Regression problems, where we are modeling a continuous output (e.g., $y = w^T \cdot x$). However, Linear Models can also be used for Classification problems, where the output is discrete (e.g., $y \in \{0, 1\}$ or $y \in \{-1, 1\}$). A naive approach to handling this, would be to just train a regression model as per before, but then just threshold the output at some value to derive the class label. For example, we can take the below binary data:



and then fit a regular linear model to this:



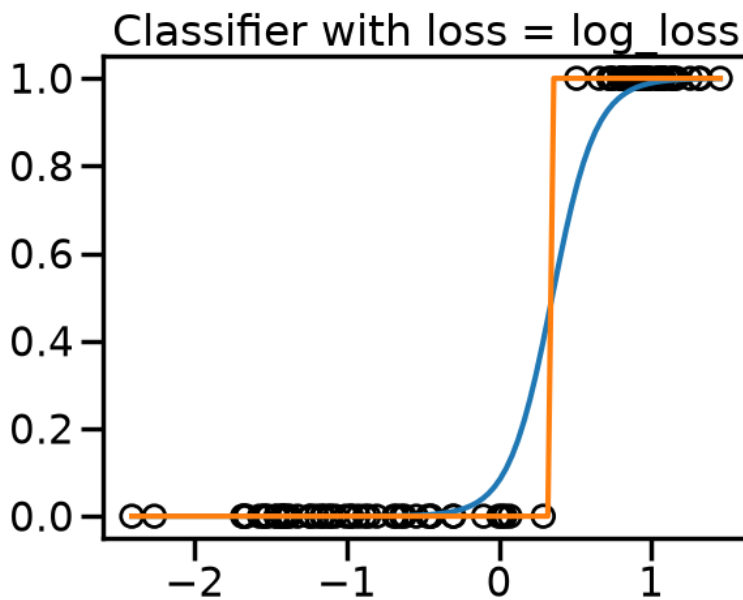
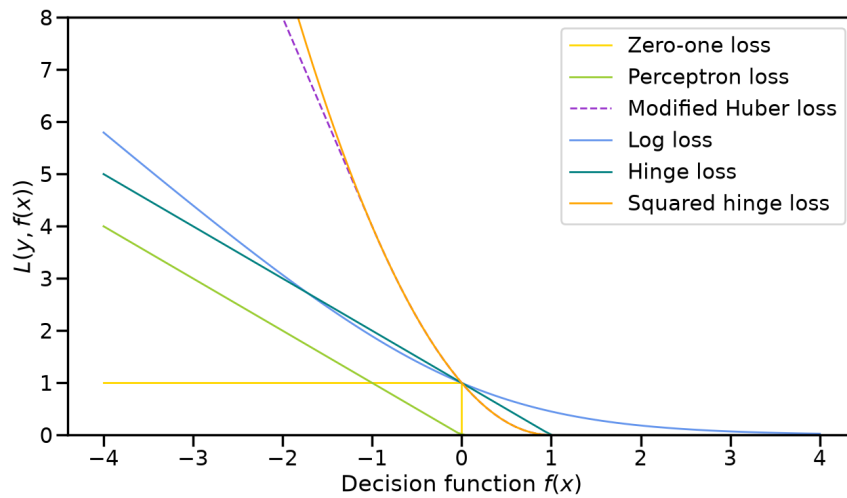
We can see from the above that this is not entirely what we want – for example, just setting the class cutoff at 0.5 would not produce a classification boundary that would be optimal. In contrast, we can modify the loss function to more accurately project the $w^T \cdot x$ linear model into a classification context.

We do this by modifying the loss function from Mean Squared Error to something like:

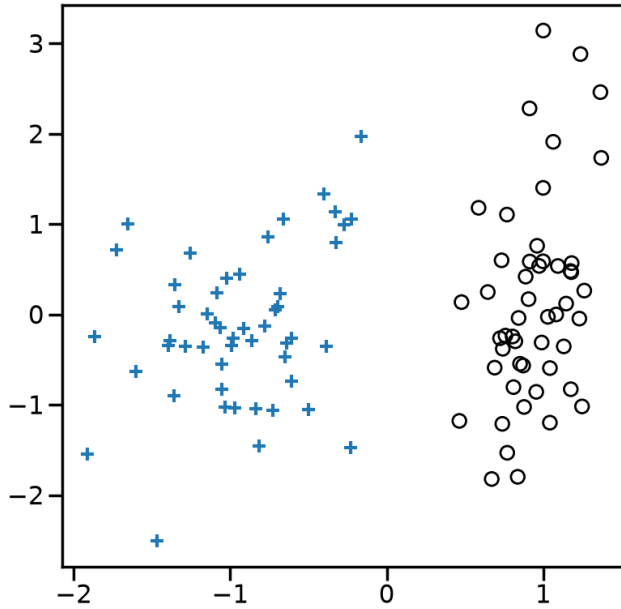
$$y_i \cdot (w \cdot x_i)$$

where $y_i = \pm 1$ such that if y_i and $w \cdot x_i$ point have similar signs, then the decision function is positive, otherwise it is negative.

With this change, we are now interested primarily with how the behavior of the loss function when we are in the negative regime (i.e., misclassified points):



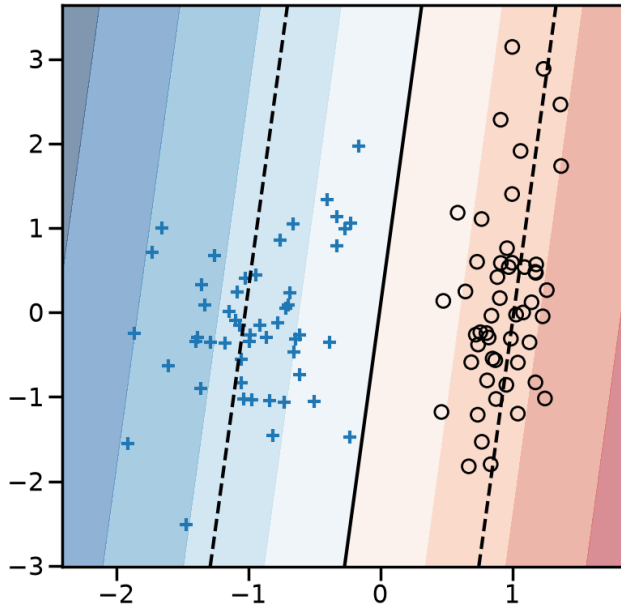
We can also try this with 2D data to get a better sense of how some of the other loss functions behave:



💡 Experiment: Effect of Linear Model Classification Losses

In the below experiment, try modifying the different classification loss functions and re-running the model. You will see a dark solid line representing the decision boundary, and dashed lines representing where the decision boundary is $+1$ or -1 . What do you notice?:

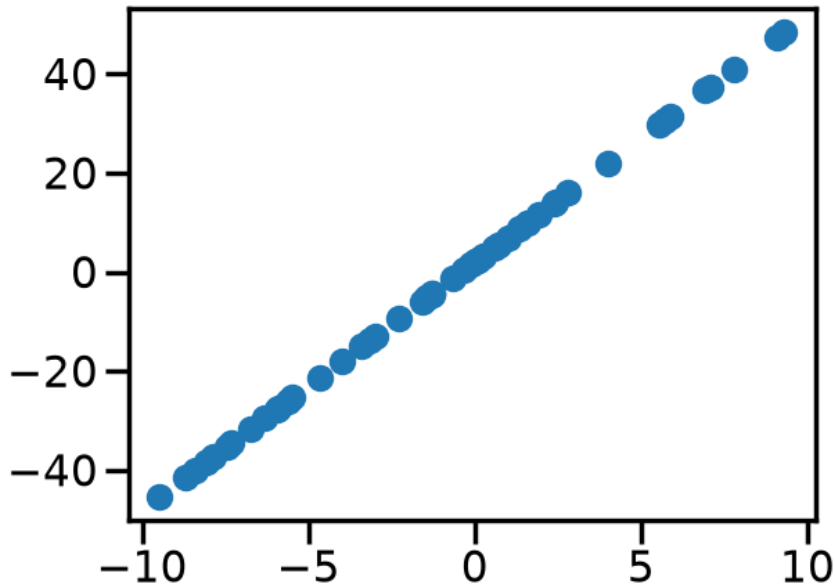
- For the perceptron loss, what behavior do you observe if you re-run this model multiple times? Why do you observe this behavior?
- Comparing the hinge loss to the perceptron loss, the loss functions look remarkably similar, yet they have very different behavior in the model. Why do you think this is?
- Comparing the squared error versus log loss versus hinge, what sorts of differences in behavior do you observe? Thinking about the location of the decision boundary and the shape of the loss functions, why do you think they behave differently?



1.6 Penalty Functions: Example of How L_p Penalty Changes Loss Optima

OK, so we have seen how different loss functions can affect the learned linear model. But what about the penalty function? How does changing the penalty function affect the learned model? Let's explore this with a simple example by again returning to fitting a line. In this case, we will fit an actual "line" to the data, by which I mean:

$$y = w_0 + w_1 \cdot x = \mathbf{w}^T \cdot \mathbf{x}$$



In this case, since we have very little noise (although you can play with this if you would like), you can see that if we put in the true intercept and slope, we get zero (or close to zero) training objective.

```
from numpy.linalg import norm
def loss(a,b,alpha,order=2):
    return np.average((y_line - (a*x_line+b))**2) + alpha*norm([a,b],ord=order)

#example
print(f"Objective: {loss(a=5,b=2,alpha=0)}")
```

Objective: 9.464156818699441e-07

This tells us the objective at the true parameters, but now let's visualize how the total objective (training error + penalty) varies as we change the parameters. To do this, we can compute the total objective over a grid of possible values for w_0 and w_1 , and then plot the contours of this objective function:

```
interact(plot_objective,
        alpha=np.logspace(-2,5,8),
        order=FloatSlider(min=0,max=10,step=0.1,continuous_update=False,value=2.0))
```

[Interactive output: available in the online version of this chapter.]

Here we can see the true optimal parameters as a blue star, the objective function contours as dark gray lines, and the red diamond shows the point where both weights are zero. The orange circle shows the minimum point of the objective function. If you increase the value of alpha in the drop down, this will increase the penalty weight, and you can see how both the objective and the optimal point change. Moreover, you can change the p-norm order in the penalty to see what effect this has on the total objective function.

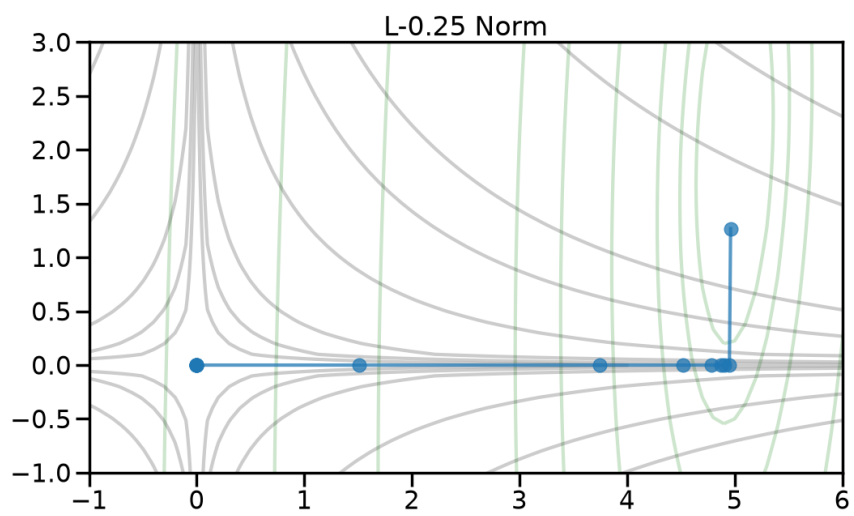
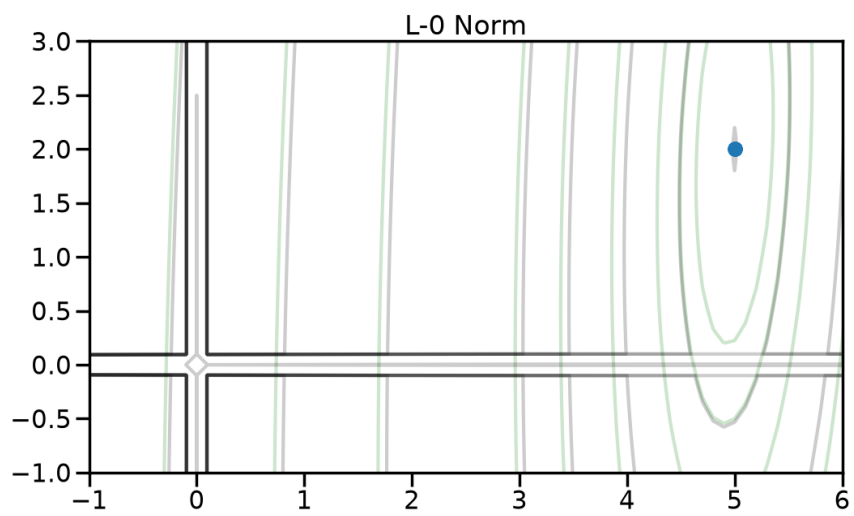
💡 Experiment: How do the different penalty terms affect the Objective Function and the optimal weight?

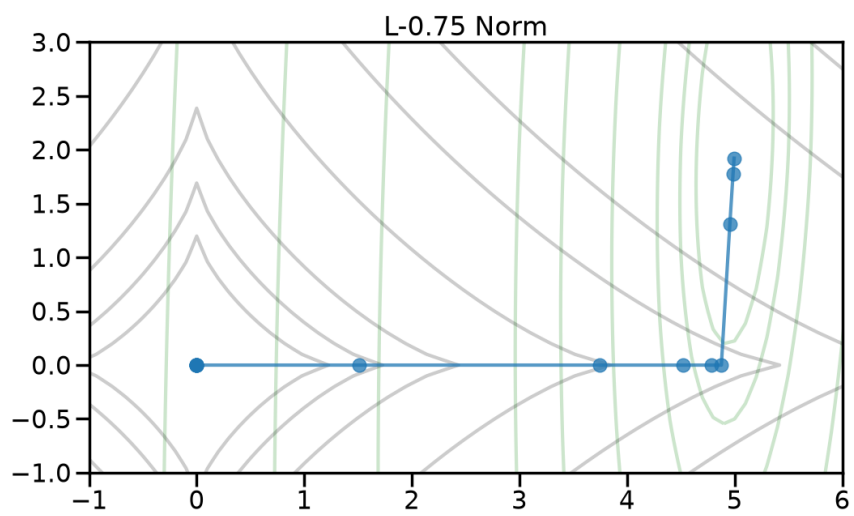
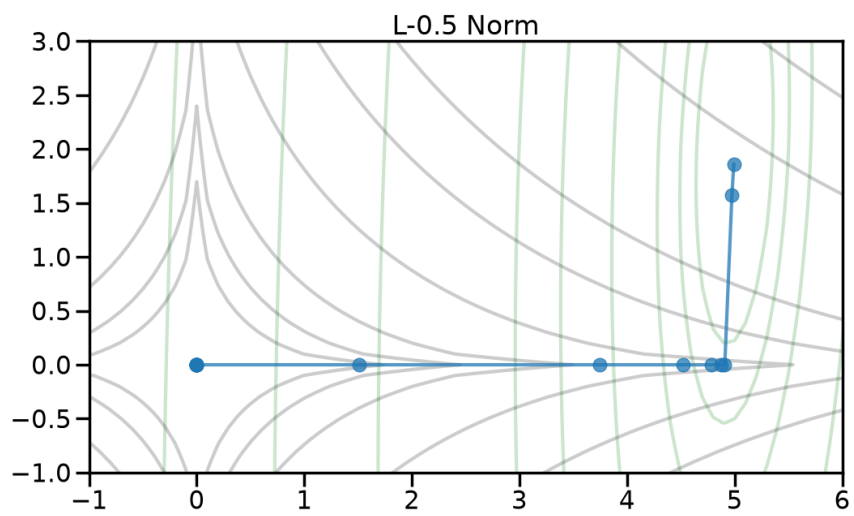
Try modifying the following:

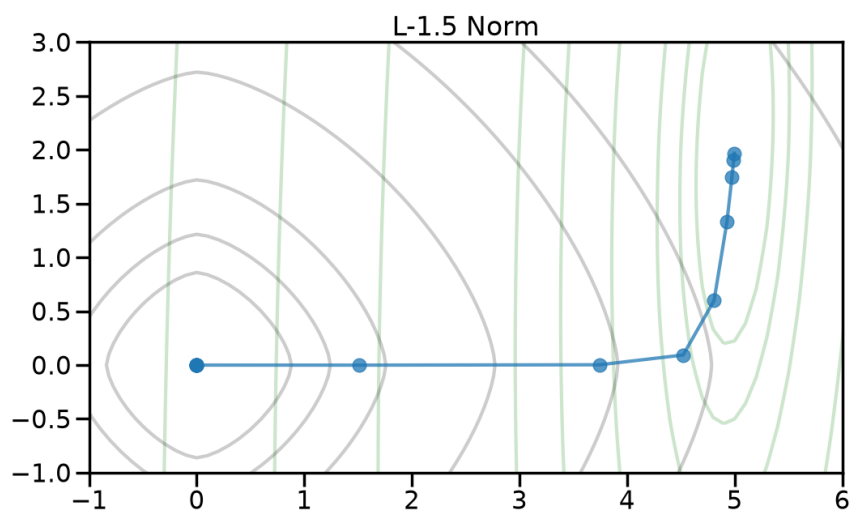
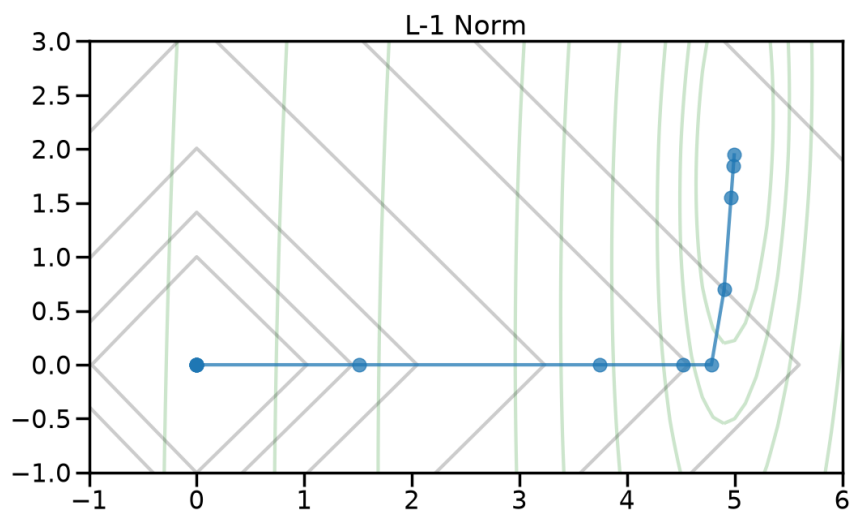
- As you increase the penalty weight under the L2 norm, how does the objective landscape change?
- If you use a p-order that is less than 2, how does this alter the objective landscape?
- The L1 norm is known to induce sparsity in the optimal weights. Do you observe this in the objective landscape? Why or why not?
- Different types of regularization within the p-norm family are often referred to as “Shrinkage” operators. Why do you think this is the case, based on what you observe?
- If my goal is to induce sparsity, it would make sense to consider the L0 norm, which just counts the number of non-zero entries in a vector. Based on the plots below, why won’t this work?

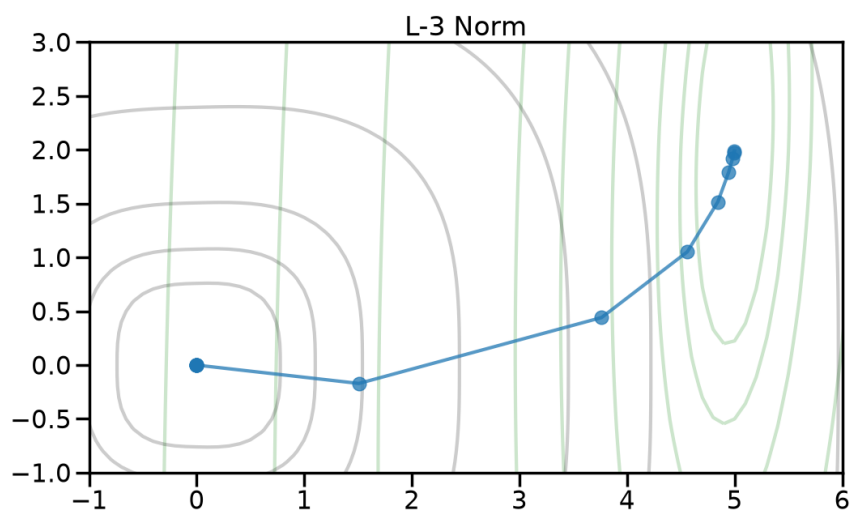
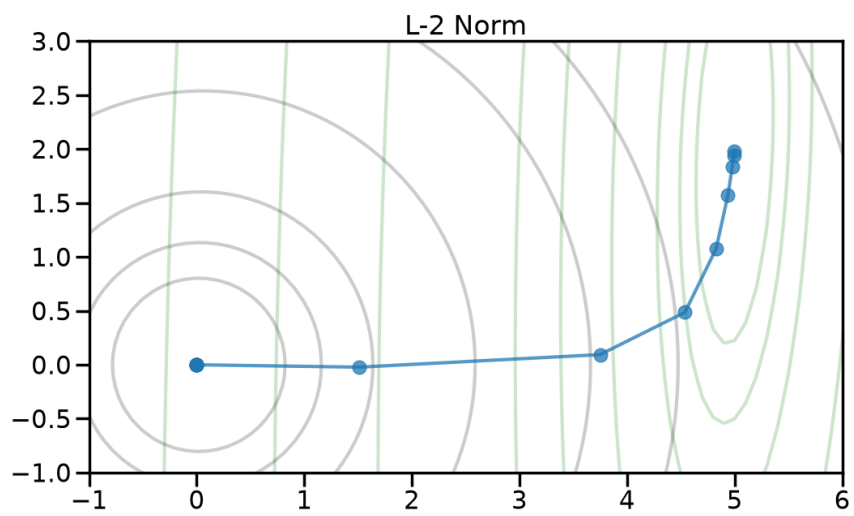
To help visualize the experiment above, let’s try plotting, for different norm orders, the path that the coefficients take as we set $\alpha = 0$ (no regularization) to a large number (essentially fully regularized). Now the light green contour represents just the contribution to the objective from the training error (no regularization), and the light gray contour shows the contribution from the penalty term (no training error). The blue line shows how the optimal weight changes as we increase the penalty weight from 0 to a large number.²

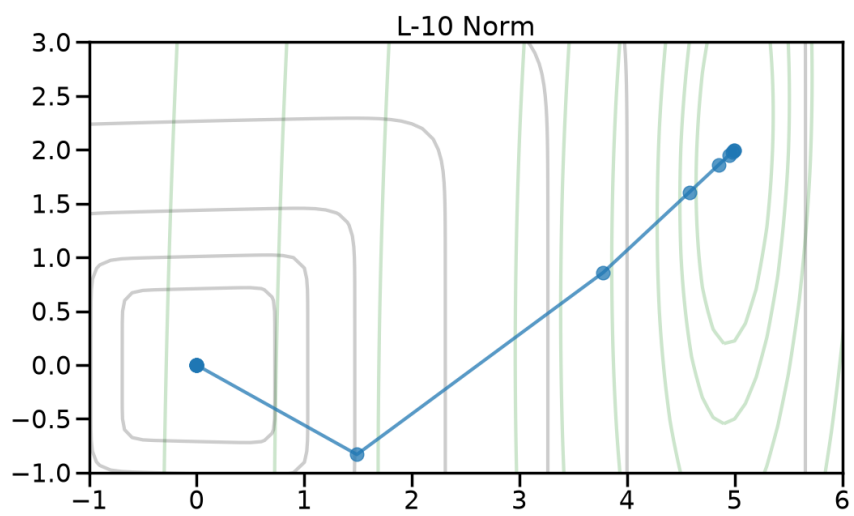
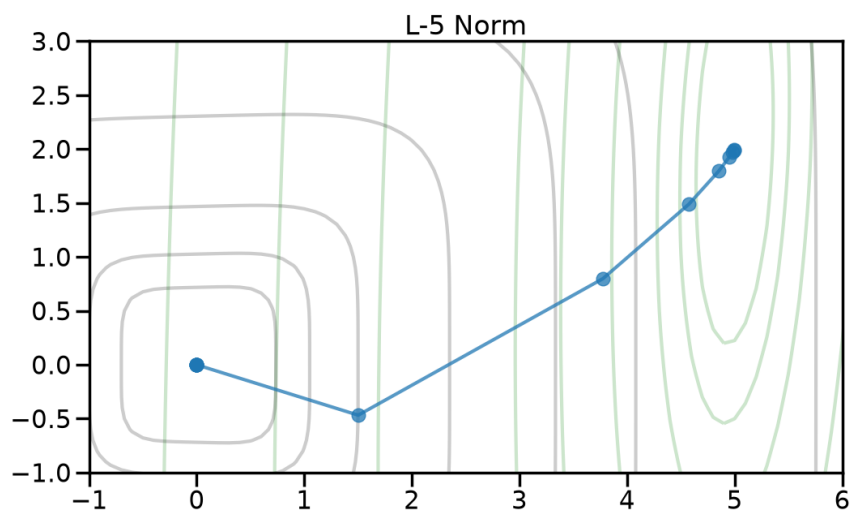
²Note how I can visually find the optimal weight by fixing a given iso-contour of the regularization term (the gray lines) and then finding the point along that iso-contour where the green contour is minimized.

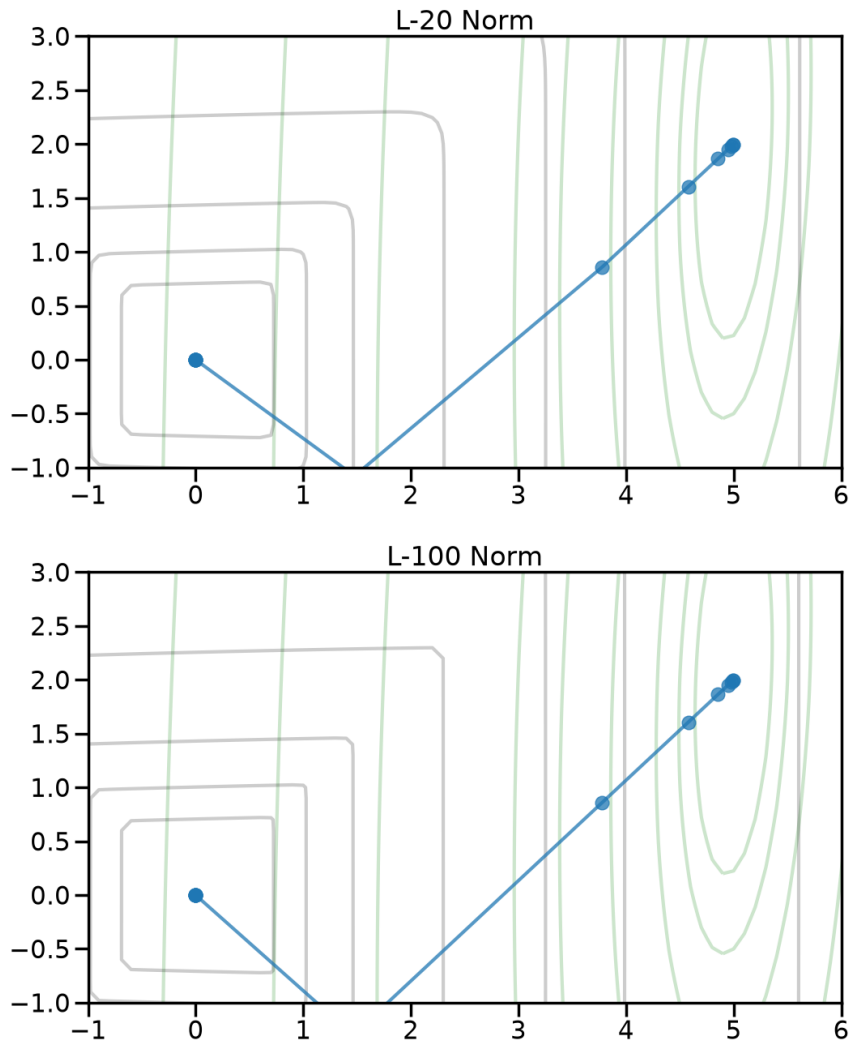












1.7 Summary of How to Select a Loss or Penalty function

In class, we reviewed a couple of different forms of loss functions and penalty functions and talked a bit about the criteria for selecting them. Below is a very short summary of these.

1.7.1 Loss Functions for Regression:

- If you have no prior knowledge of the function or data, then selecting an L2 type loss (like the Mean Squared Error) is reasonable. When data nicely behaves w.r.t. a linear model (e.g., features are uncorrelated, errors in the linear model are uncorrelated, have

equal variances, and expected error of zero around the linear model, etc.) then a Linear Model with an L2 Loss is the Best Linear Unbiased Estimate (BLUE) according to the [Gauss-Markov Theorem](#).

- If you have reason to believe that the data will have outliers or otherwise need to be robust to spurious large samples, then L2 loss will not be robust to this (as we saw in Lecture). For this, moving to an L1 type loss (like an Absolute Loss or Huber loss) will make the model less sensitive to outliers. It is one approach to handling [Robust Regression](#).
- If you need to have the model's loss be dominated only by a handful of points/data as opposed to all of the data, then epsilon-insensitive loss is appropriate since many points well-fit by the model will have “zero” loss. For things like Linear Models, this has limited usefulness right now. However, when we “kernalize” Linear models in “Kernels” week, you will see that this is a big deal, and it is what gives rise to the “Support Vector” part of “Support Vector Machines”. Specifically, decreasing epsilon towards zero increases the number of needed Support Vectors (can be a bad thing), and increasing epsilon can decrease the number of needed Support Vectors (can be a good thing). This will make more sense in a few week's time.

1.7.2 Loss Functions for Classification:

- Zero-One loss sounds nice, but is not useful in practice, since it is not differentiable.
- [Perceptron loss](#), while of historical importance, is not terribly useful in practice, since it does not converge to a unique solution and an SVM (i.e., Hinge Loss below) has all of the same benefits.
- If you need a simple linear model which outputs actual probabilities (like, 95% sure the component has failed), then the log-loss does this, via [Logistic Regression](#) and allows you to calculate classification probabilities in closed form.
- If you want something that [maximizes the margin](#) of the classifier, then the Hinge Loss can get close to this. It is the basis of [Linear Support Vector Machines](#)

1.7.3 Penalty Terms (Lp Norms) for Linear Models:

- L_2 Norm penalties on the weight vector essentially “shrink” the weights towards zero as you increase the penalty weight. Adding this kind of penalty to a linear model has different names, depending on which community of people you are talking with. Some of these other names are: (1) [Tikhonov regularization](#), (2) Ridge Regression, (3) L_2 Shrinkage Estimators, or (4) Gaussian Weight Priors. I find looking at the penalty term itself more helpful at understanding the effects rather than memorizing the different names.

- L_0 Norm penalties, while conceptually interesting since they essentially “count” entries in the weight vector, are not practically useful since they are not differentiable and are thus difficult to optimize.
- L_1 Norm penalties are a compromise between L_2 and L_0 in that they promote sparsity in the weights (some weights will become zero) but are (largely) differentiable, meaning that you can meaningfully optimize them (unlike L_0). Shrinking certain weights to zero in this way can be useful when (1) you are in the $n \ll p$ regime (many more features than data points) where the model is underdetermined and (2) you want some degree of model interpretability (it sets many weights to zero). Some of these other names for this kind of linear regression with this penalty are the [LASSO \(least absolute shrinkage and selection operator\)](#).
- L_∞ (where p is really large) essentially penalize the size of the biggest element of the weight vector (w). While there are some niche instances where this kind of norm is useful for a Loss function (e.g., [Chebyshev Regression](#)), I have rarely seen meaningful use cases in practice where this makes sense as a penalty term for the weight vector.
- Combinations of penalties. For example, a common combination is combining both an L_2 and L_1 penalty on the weights, as in: $\Omega(w) = \alpha\|w\|_2 + \beta\|w\|_1$. This particular combination is called “[Elastic Net](#)” and exhibits some of the good properties of L_2 penalties with the sparsity inducing properties of L_1 regularization.

1.8 Exercises

The exercises below reuse the data and code from this chapter, so run the chapter’s cells first. Rough time estimates are in the titles. Worked solutions are collected in the Exercise Solutions appendix, but try each one before looking.

Exercise 1 (~4 min): Is it linear?

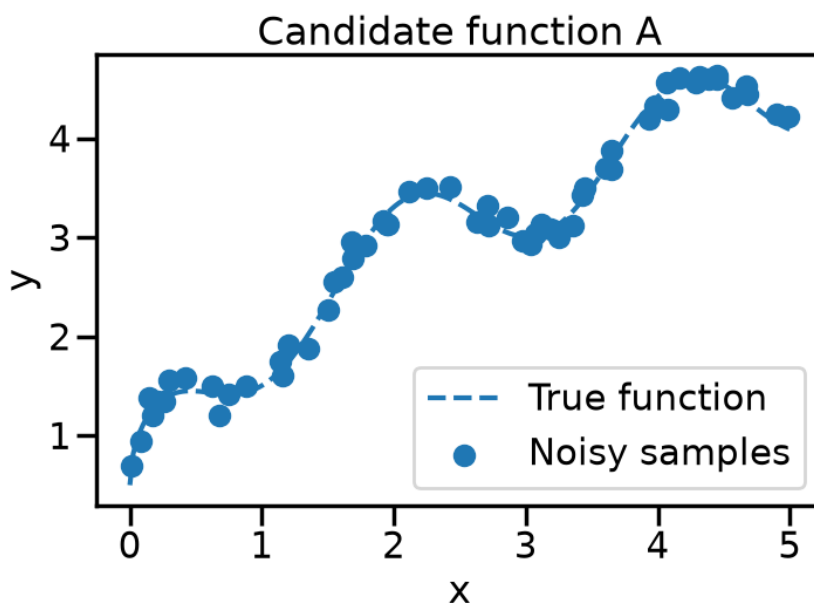
A model is *linear* when it is linear in its parameters, not necessarily in its inputs. For each of the candidate models below (with parameters w_i , a , b and input x), decide whether it is (i) a linear model as written, (ii) not linear as written but can become one after we transform either y or x (and if so, say which one or both and how you would transform it), or (iii) not a linear model in any simple way and cannot be made linear even through simple transformations.

- (a) $y = w_0 + w_1x + w_2x^2$
- (b) $y = ae^{bx}$
- (c) $y = w_0 + w_1 \log x + w_2\sqrt{x}$
- (d) $y = w_1 \sin(w_2x)$
- (e) $y = ax^b$ (for example, the Basquin law relating stress amplitude to cycles to failure)

- (f) $y = w_0 + \frac{w_1}{w_2 + x}$
- (g) $y = \frac{1}{w_0 + w_1 x}$

For the models you transformed: if the measurement noise on y is additive and Gaussian, is it still additive and Gaussian after your transformation? Why does this question matter in the context of this chapter?

The below exercise two gives you a few functions that are each a sum of simple terms. Run the cell below to see the candidate functions and to sample noisy data from one of them.



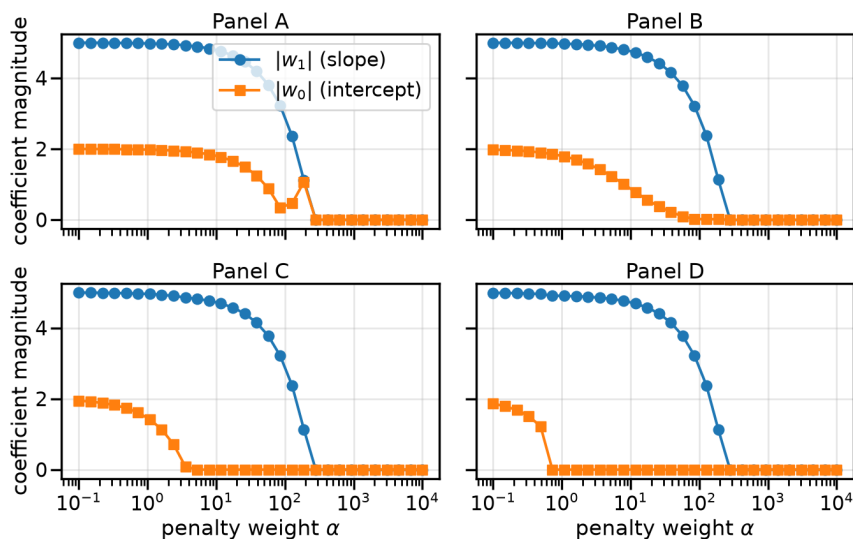
Exercise 2 (~8 min): Build your own feature expansion

Each candidate function above is a sum of a few simple terms, so, just like the polynomial case at the start of the chapter, a linear model can fit it exactly once it has the right features. Pick one candidate (Feel free to modify and re-run the code cell above to generate data from different candidate functions).

- (a) Write a function that turns the input array x into a feature matrix whose columns are the terms you think you need (the polynomial example used `PolynomialFeatures` to do this for you in the earlier notebook, but you can implement this however you like).

- (b) Fit a `LinearRegression` on `x_ex`, `y_ex` using your features and compare the fitted weights with the coefficients in the candidate function.
- (c) Predict y at `x_new` and compare with the true function.
- (d) If you have time: give the model the *wrong* features, for example polynomial features of degree 3, and look at what happens to the fit near the ends of the interval.

The next two exercises return to the line-fitting problem from the penalty section, where the data came from $y = 5x + 2$. Run the cell below to produce the figure for Exercise 3.

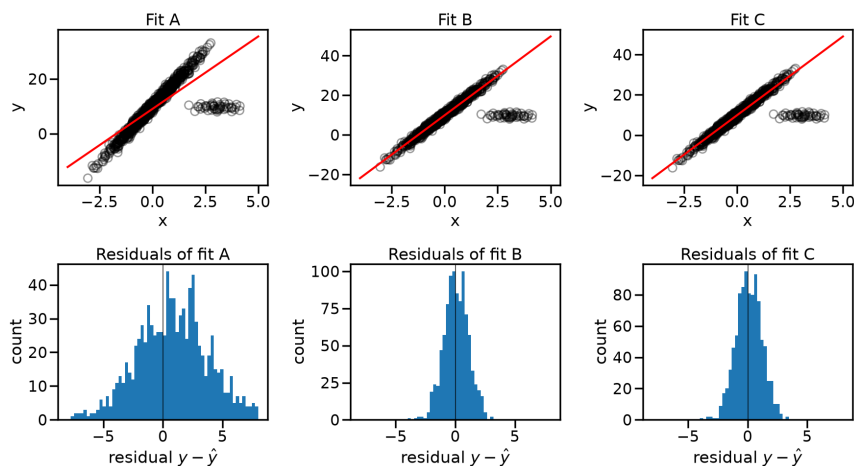


Exercise 3 (~5 min): Match the shrinkage path to the norm

Each panel above shows how the two fitted coefficients of the line $y = w_1x + w_0$ shrink as the penalty weight α grows, for a penalty $\alpha \|\mathbf{w}\|_p$ with one of $p \in \{0.5, 1, 2, 100\}$ (the last one is essentially the L_∞ norm).

- (a) Match each panel to its p . Write one sentence per panel saying which feature of the curves gave it away.
- (b) The true intercept (2) is smaller than the true slope (5). Under which norms does the smaller coefficient reach exactly zero while the other is still clearly non-zero? Why does that happen for those norms and not the others?

Run the cell below to produce the figure for Exercise 4. It refits the outlier data from the robust regression section with three different losses.

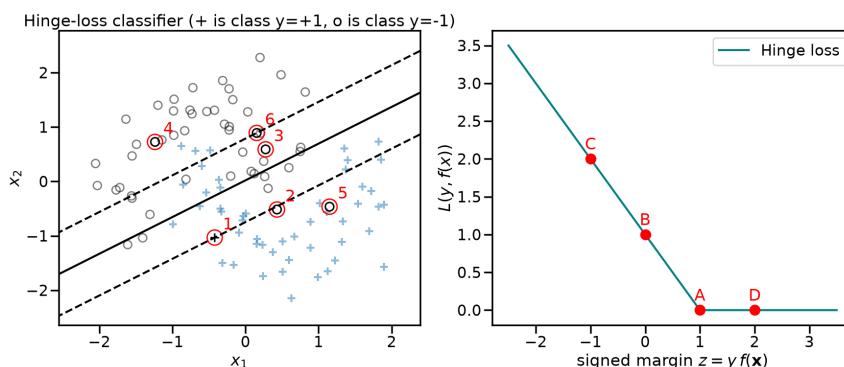


Exercise 4 (~5 min): Which loss made this fit?

The three fits above were trained on the same outlier data with the squared loss, the Huber loss, and the epsilon-insensitive loss (with $\epsilon = 1$), in a shuffled order. The bottom row shows the histogram of residuals $y - \hat{y}$ for each fit.

- (a) One of the fits is clearly different from the other two. Which loss produced it? Point to two features of its residual histogram that give it away.
- (b) Can you tell the remaining two fits apart from these plots? Look back at the loss-shape plot earlier in the chapter and explain why these two losses lead to almost the same line on this data even though their formulas differ.
- (c) Under what kind of data would you expect those two losses to give visibly different fits?

The next two exercises use a hinge-loss classifier trained on the “moons” data from the classification section. Run the cell below to produce the figure. The left panel shows the decision boundary (solid) and the margins where the decision function equals ± 1 (dashed), with six numbered points and a dashed line from each to the boundary. The right panel shows the hinge loss as a function of the signed margin $z = y f(\mathbf{x})$, with four positions marked A to D.

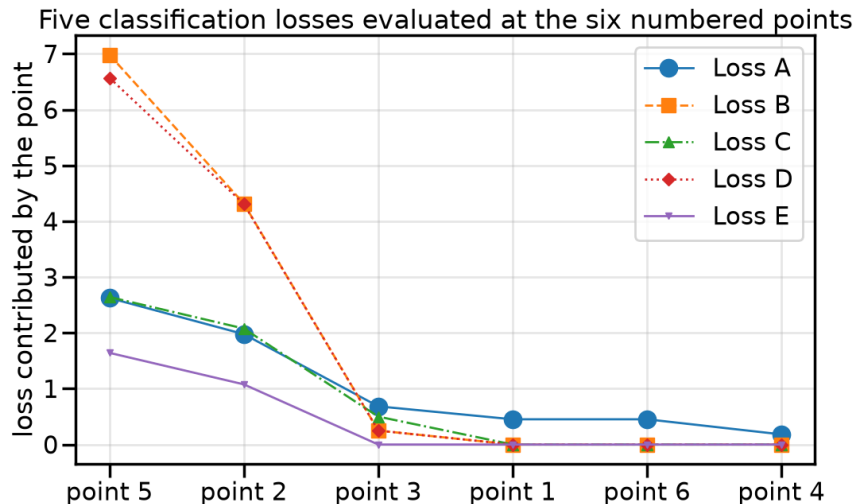


Exercise 5 (~6 min): Where do these points sit on the hinge loss?

The classifier above was trained with the hinge loss, so every point contributes $\max(0, 1 - y f(\mathbf{x}))$ to the training loss, where $y = \pm 1$ is the true label and $f(\mathbf{x})$ is the decision function (zero on the solid line, ± 1 on the dashed lines).

- (a) For each numbered point, decide which marked position on the loss curve (A, B, C or D) it is closest to. Use the point's class and its position relative to the solid and dashed lines; you do not need any numbers.
- (b) Which points contribute exactly zero loss? Among those, which ones sit exactly at the kink in the hinge loss (position A)?
- (c) In a Support Vector Machine (SVM), the points at the kink and those with positive loss are the *support vectors* which give the method its name. In fact, moving any other data points slightly does not change the boundary at all for such models. Why is this? Which of the six points could you delete without changing the classifier?

Run the cell below to produce the figure for Exercise 6. It evaluates five of the classification losses from this chapter on the same six numbered points from Exercise 5, but labels the losses only with letters.



Exercise 6 (~5 min): Name the classification loss

The plot above evaluates five losses from the classification section (hinge, perceptron, log loss, modified Huber, squared hinge) at the six numbered points of Exercise 5, sorted from the most misclassified point on the left to the most confidently correct point on the right. The losses are labelled only A to E.

- (a) Identify each loss. State the one feature of each curve that identifies it (for example: which points get exactly zero, how fast the loss grows on the left, whether it ever reaches zero). Hint: once you have found the hinge loss, you can read off the signed margin z of every point from it, and then check the other formulas by hand.
- (b) Two of the losses give exactly zero for every correctly classified point, however close to the boundary. Which two, and what does that imply about where the decision boundary ends up when you train with them? (Look back at the perceptron question in the classification experiment.)

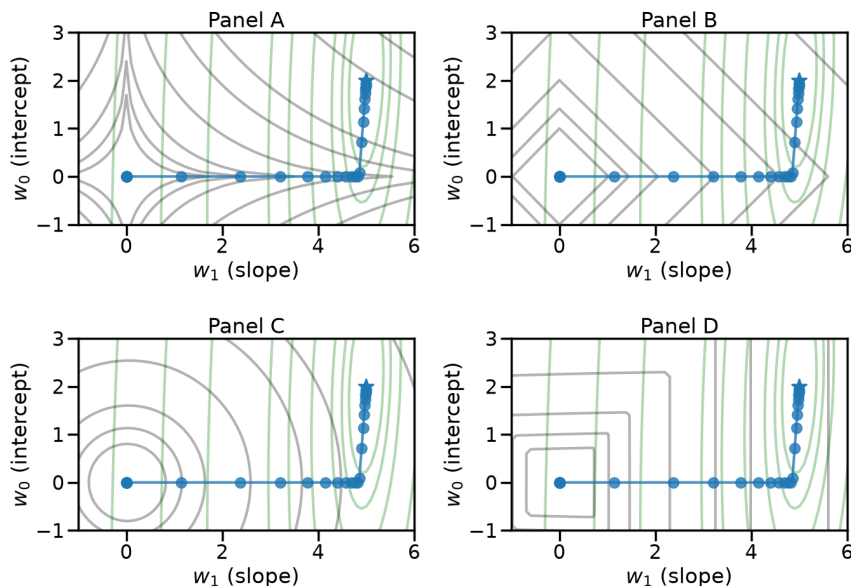
Exercise 7 (~3 min): Choose the penalty

For each situation, argue for an L_1 or an L_2 penalty on the weights (or a combination), in two or three sentences each.

- (a) You are fitting a linear fatigue-life model with 40 candidate features (load amplitudes, temperatures, surface measures, several of them strongly correlated). The report must name the handful of features that actually matter.
- (b) You are fitting a sensor calibration model that is re-fit every month on a fresh

batch of data. Several input channels are nearly redundant, and the plant engineers complain when the coefficients jump around between months.

Run the cell below to produce the figure for Exercise 8. Each panel shows the training-error contours (green) and the penalty contours (grey) for one p -norm, together with *the same* coefficient path (blue), which was computed for a single value of p as α increases.

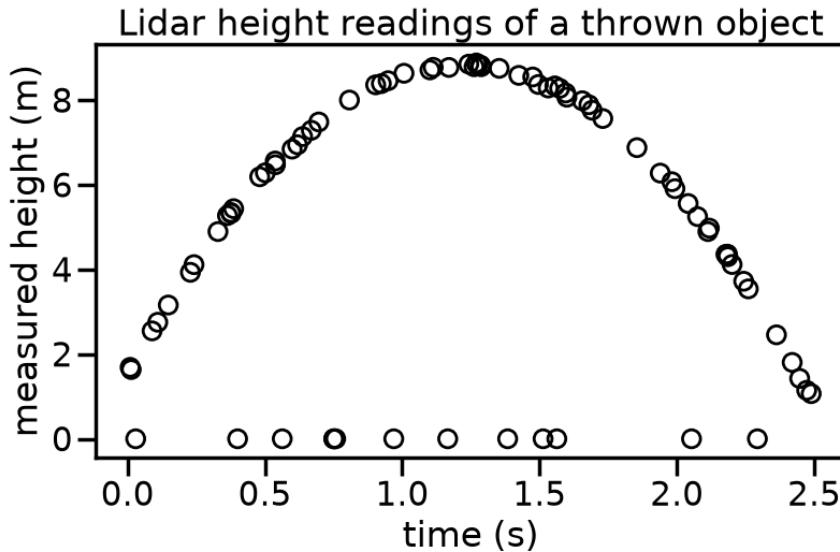


Exercise 8 (~3 min): Which penalty produced this path?

The blue path is the same in all four panels: it is the sequence of optimal (w_1, w_0) as the penalty weight α grows from small to large for **one** choice of p . The grey contours differ between panels because each panel uses a different $p \in \{0.5, 1, 2, 100\}$.

- (a) Identify the p of each panel from the shape of the grey penalty contours.
- (b) Only one panel shows the penalty that actually produced the blue path. Which one is it? Explain, using the way the path meets the axis $w_0 = 0$, why the other three penalties could not have produced it.
- (c) Sketch (or describe) how the path would look in each of the other three panels.

The last exercise is more open-ended. Run the cell below to load the measurements.



Exercise 9 (~12 min): Tracking a thrown object with a flaky lidar

A lidar tracks a ball thrown upwards and records its height over time in the arrays `t_lidar` and `height_lidar`. Whenever the lidar misses the ball, it reports a height of exactly zero, and it does so fairly often. Your job is to recover the launch height, the launch speed and the gravitational acceleration from these readings using the tools of this chapter.

- (a) Decide what model to fit. Which features of t do you need so that the model is still linear in its parameters? What does each parameter mean physically?
- (b) Decide how to treat the dropped readings. Rather than deleting them by hand, choose a loss function from this chapter that will not be thrown off by them, and say why.
- (c) Fit the model and report your estimates of the three physical quantities. The ball was thrown near sea level, so you can check one of them.
- (d) Fit the same model with the squared loss and compare. Which estimate suffers most from the dropped readings, and can you explain why from the shape of the data?

If you use `SGDRegressor`, remember that it is sensitive to the scale of the features; `HuberRegressor` from `sklearn.linear_model` is a convenient alternative that fits the Huber loss directly.

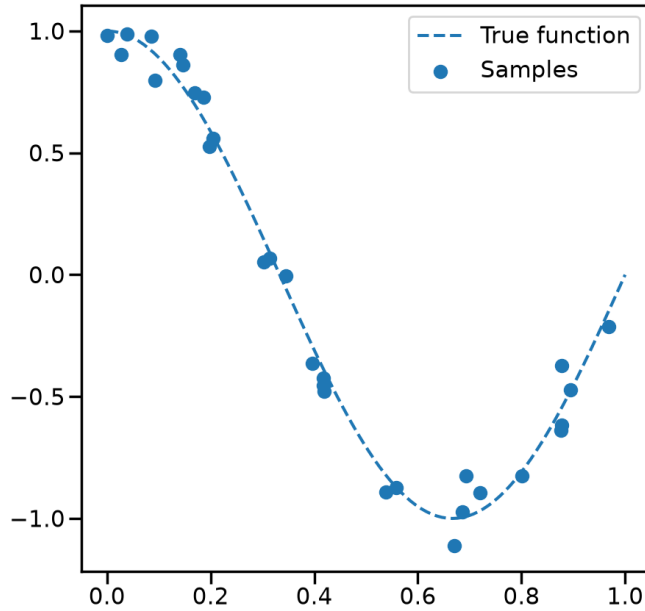
2 Evaluating Machine Learning Models

In the prior chapter, we covered how different loss functions and regularization terms affected a linear model, in terms of the model's qualitative performance and its effect on the training score. However, as we saw, the training score can be a misleading performance indicator. How might we judge the model's performance more rigorously? This chapter addresses this by reviewing the why and how of performing Cross Validation, and also what this means regarding optimizing the hyperparameters of a model.

Learning Objectives

By the end of this chapter, you should be able to:

1. Explain why training error is an optimistic estimate of performance, and use hold-out and k -fold cross-validation to estimate out-of-sample error instead.
2. Choose a cross-validation strategy (shuffled, stratified, grouped, time-ordered) that matches how the model will be used, and recognize the leakage each one prevents.
3. Select hyperparameters such as the regularization strength and the polynomial degree from validation curves and cross-validation grids.
4. Use grid search and randomized search over a pipeline, and state when each is preferable.
5. Describe Bayesian optimization as model-based search with an acquisition function, and say when it pays off.
6. Explain the curse of dimensionality and its consequences for search and for distance-based methods.



2.1 Getting an Unbiased Estimate of Out-of-Sample Performance

When we train a machine learning model, we are typically interested in how well the model will perform on data it has not seen before. This is often referred to as the model's generalization performance. However, if we evaluate the model's performance on the same data it was trained on, we may get an overly optimistic estimate of its true performance. This is because the model may have simply memorized the training data, rather than learning the underlying patterns. Put another way, assessing the model on data it was trained on does not give us an accurate picture of performance because the training loss is not an *unbiased* estimate of the model's out-of-sample performance, where *out-of-sample* here refers to new data I haven't seen yet. So how do we properly obtain (*i.e.*, in an unbiased way) an estimate of the model's out-of-sample performance? One popular way to assess this is through [Cross Validation](#), where we attempt to split the given data into two sets – a training set and a testing set – where the testing set should come from a distribution similar to what we would expect to see in the real-world when we deploy the model. A simple way to do this is just to select points at random:

```
# Now let's split the data into training and test data:
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X_cos, y_cos, test_size=0.2, random_state=0)
```

Let's take a look at what the above has actually done.

```

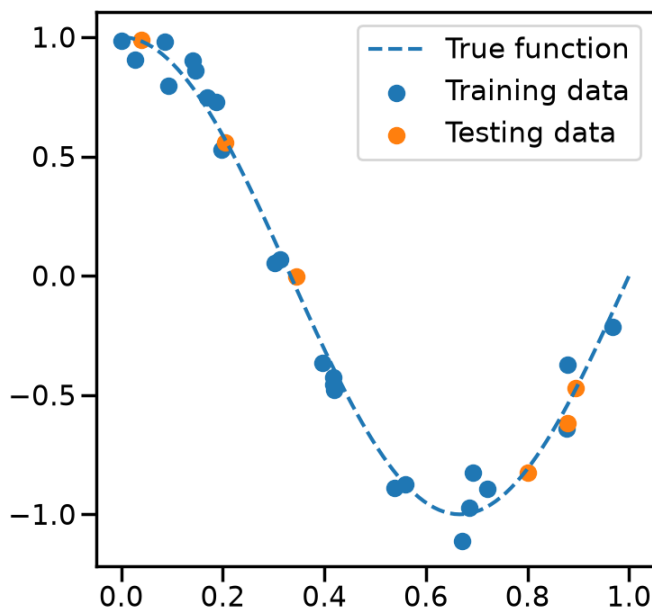
X_train
[8.78142503e-01 3.02332573e-01 4.19194514e-01 6.92322616e-01
 1.40386939e-01 4.17304802e-01 1.86260211e-01 3.96767474e-01
 7.20324493e-01 6.70467510e-01 2.73875932e-02 9.68261576e-01
 1.46755891e-01 9.23385948e-02 5.38816734e-01 5.58689828e-01
 1.98101489e-01 1.69830420e-01 8.76389152e-01 8.50442114e-02
 1.14374817e-04 6.85219500e-01 4.17022005e-01 3.13424178e-01]

X_test
[0.03905478 0.89460666 0.34556073 0.20445225 0.87811744 0.80074457]

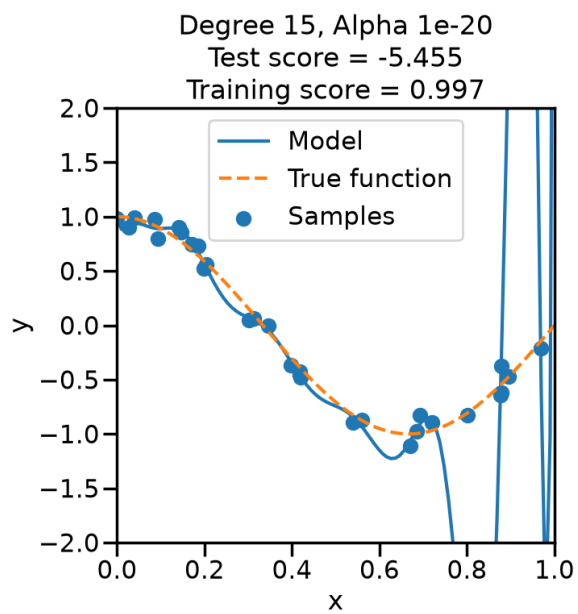
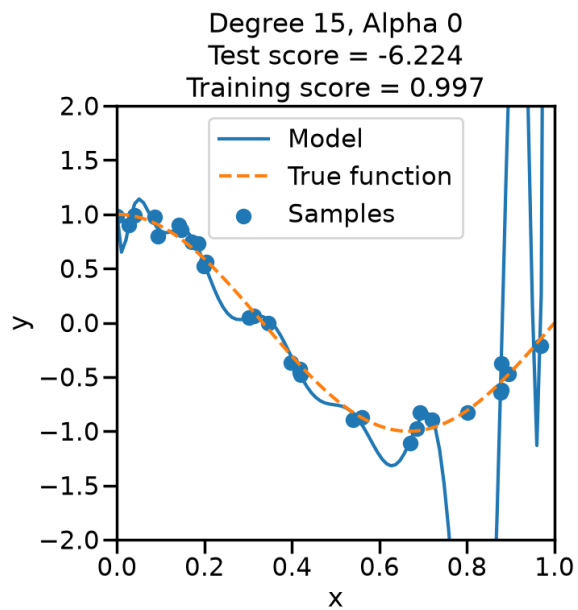
y_train
[-0.37395132 0.05199163 -0.47818202 -0.82672018 0.90350849 -0.45417721
 0.72898424 -0.36366041 -0.89399733 -1.11157064 0.90389734 -0.21270638
 0.86040424 0.79675107 -0.89105816 -0.87458209 0.52662674 0.74673588
 -0.63887828 0.97904574 0.98275703 -0.97273902 -0.42390528 0.0668933 ]

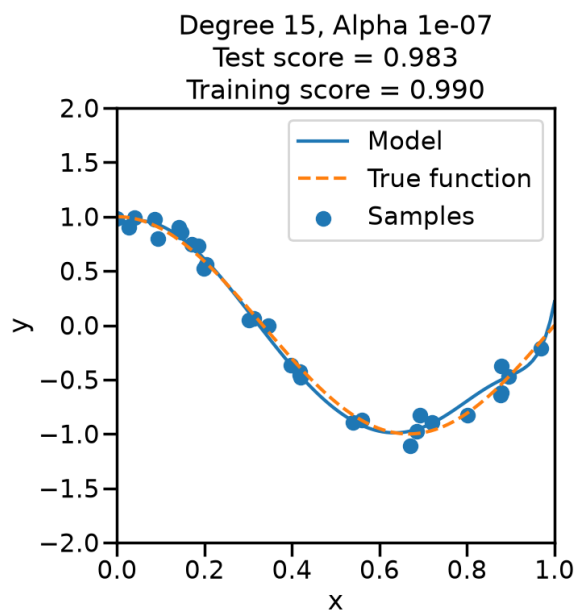
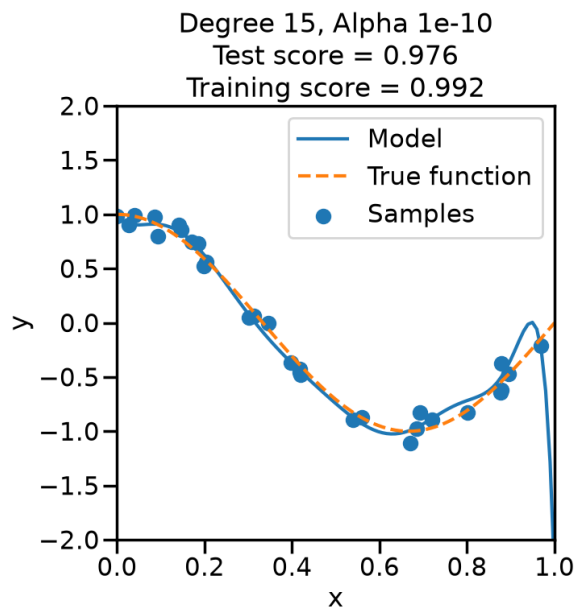
y_test
[0.98733352 -0.47140604 -0.00455281 0.55839434 -0.61801179 -0.82613321]

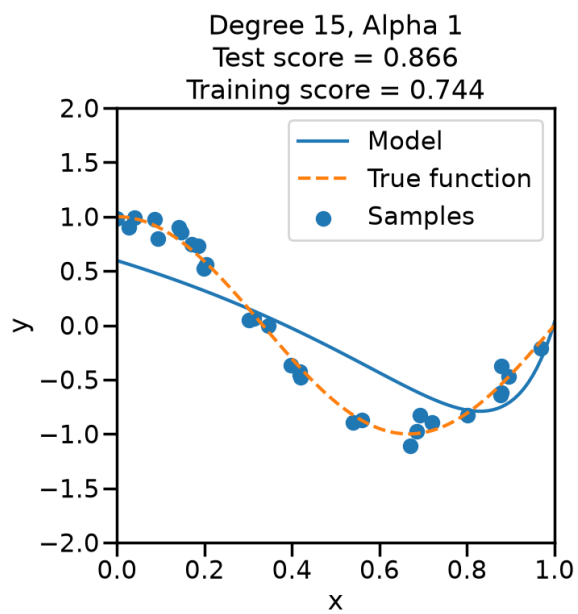
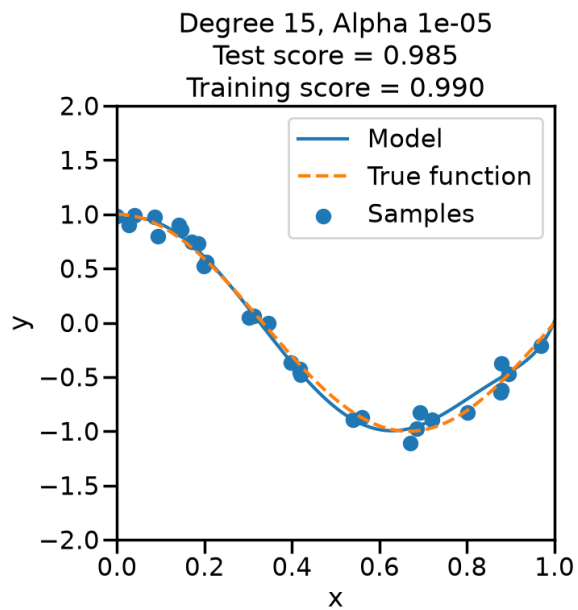
```

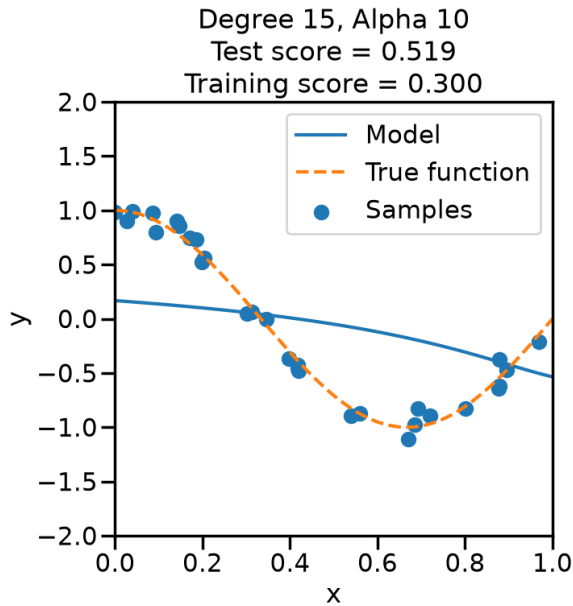


We can now plot or track the error across both the training and testing data to see how the different models compare:









We can see from the above behavior that while the training score prefers the increasingly more complex models, the test score tells us more accurately how the model would perform under new data that it hasn't seen yet, and that we can use this information to actually assess which model to choose.

2.2 Choosing a Cross-Validation Strategy

The split above is the simplest kind of cross-validation: shuffle the rows, select some that we will hold out for test data, and then fit the model on the remaining training data. In principle, this sounds reasonable, but is this always the correct approach? Should we be selecting our test data a different way? The key to answering this question is to ask yourself whether you are selecting test data that is truly an *unbiased* estimate of the *out-of-sample* error. Let's unpack what this means. For the data to represent something which is out of sample, this means the data that I'm selecting for testing needs to represent new samples of data that I am likely to see in reality. To make this a little less abstract we can look at three types of situations that commonly occur in cross validation where just randomly splitting the data as we did above would cause problems:

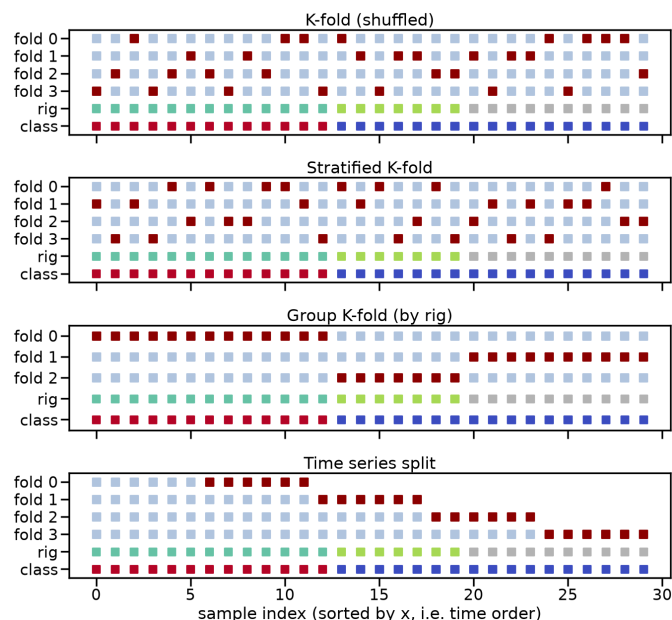
1. **Stratification:** Consider cases where the data suffers from severe class imbalance – for example let's say I have 1000 samples of engine performance but only five of those samples contain engine faults. Faulty data will not be heavily represented in the data overall. What is likely to happen if I just sampled this data randomly is that by chance I may get all of the faulty data in my training set and none of it in my test set. If this happens my test set will not look like my actual out of sample error in the sense that

it will only be testing the model's ability to predict non-faulty or normal engine data and I will not get an accurate out of sample estimate for its true performance in real world conditions. The solution here would be to maintain the same ratio of normal to faulty data points across both the training and testing sets. This is referred to in cross validation as *stratification*. In libraries that implement cross validation, or if you were doing this manually yourself, you would first need to separate out the different classes or bands or regimes in which you expect your data to vary significantly and then sample the data in proportion to these bands in a similar way across both the training and testing splits. We will see examples below in `sklearn` of `StratifiedKFold` that do this, but the idea is simple enough you could implement it yourself.

2. **Grouping:** Often we may have data that is specific to individual parts or patients or systems where we would like to train a model on them, but then in deployment in the real world we would have to test on new parts or patients that we have never seen previously. For example, you can imagine a case where I have ten turbine blades, and for each I have time series data on their fatigue and vibration performance. I might wish to build a model that can predict their remaining useful life so that I could predict a servicing envelope. It would be tempting to just take the entire set of data for all blades and randomly sample different time points and build a predictive model that tests on other time points for all blades. However, this is not a realistic situation that I would encounter in the real world, since this would imply that I have the ability to look at random time histories of every part that I would be required to test and likely this will not be the case because I will be asked to predict new turbine blades that I never trained on. Rather, I would have, say, only eight of the turbine blades as a group that I could train on, and then my test data would look like two completely new turbine blades that I never trained on; turbine blades for whom I have zero history or knowledge. In cross validation this is referred to as *grouping* since I am essentially taking a subset of my time series data and purposefully grouping them by a part number in this case and testing on a completely new set of groups, that is new part numbers. In `sklearn` the `GroupKFold` is an example of this.
3. **Time Series Splitting:** Lastly, I might be in a situation where I need to make predictions about the future, say for our turbine blade remaining useful life estimate above or perhaps for some dynamical system. In these cases it does not make sense to take a single time series and split different points in time into training and testing since then I would be training on points that could occur in the future and using them to predict test points that might occur in the past. You can imagine that it would be far easier to predict the future if I already got an opportunity to see another point in the further future! As a result for anything in which we need to do forecasting in either space or time where we expect that we will not be able to rely upon measurements of those future quantities at deployment time, we need to train and test our model under similar circumstances. What this looks like is taking a set of time series data and then picking randomly some split point in time wherein I can train on any data in the past but only test on data in the future. This then mirrors my actual out-of-sample error that my model is likely to incur.

Notice that these are not mutually exclusive: a dataset can be grouped *and* ordered in time, and you may have to build the splitter yourself (see below exercises). The same logic also applies to any preprocessing you fit: a scaler or a PCA fit on all rows before splitting has already looked at the test data. Putting preprocessing inside a `Pipeline`, as this chapter does, keeps the preprocessing fit only on the training fold. For further details on common Cross Validation strategies, you can read the [scikit-learn User Guide page on cross-validation](#).

To see what each splitter actually does, let us give our thirty points three plausible attributes: they were measured in time order (so the sorted x doubles as a clock), each third of the x range came from a different test rig, and each point has a class label (positive or negative y). The figure shows which rows each strategy puts into the test fold (dark) and the training fold (light) for four folds.

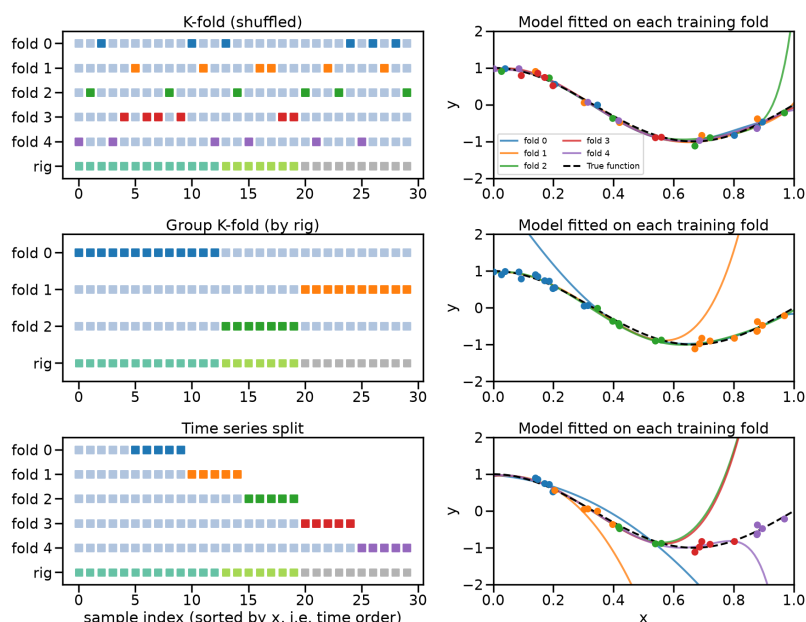


Dark squares are test rows and the two extra rows show each point's rig and class. Shuffled K-fold sprinkles test points everywhere. Stratified K-fold looks similar but guarantees that both classes appear in every test fold in similar proportion. Group K-fold holds out one entire rig at a time, so every test fold asks the model to predict a region of x it has never seen. The time series split only ever tests on points that come after all the training points.

While this example was a little bit contrived, we can see that the different cross validation strategies give very different final MSE numbers. Here is the 15-degree ridge polynomial from above for different cross validation strategies. In the figure, the left column shows which rows each fold holds out (coloured squares) and the right column shows the model fitted on each fold's training rows, in that fold's colour, together with the test points it is scored on:

strategy mean test MSE

K-fold (shuffled)	0.062
Group K-fold (by rig)	7.352
Time series split	1.192



Part of the reason the scores are so different is that in the grouped and time series case, the model has to extrapolate in ways that are more difficult than what naive shuffle splitting is asking the model to do. But which of these is the “right” cross validation approach? This is where you need to use your intuition and logic for how you plan to deploy the trained model in practice. You will have to understand what it means to have a correct out of sample estimate of your error and then combine procedures to construct the cross validation appropriately. This is something where leveraging your domain knowledge and understanding of how and where data will be available in practice compared to training will come in handy.

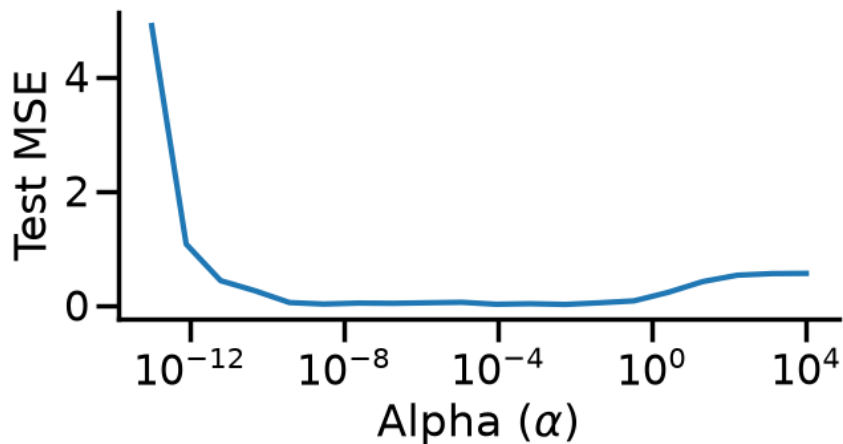
2.3 Finding the Optimal Hyper-parameters

Now that we have introduced the usage of hyper-parameters and cross-validation, a natural question arises: How do we choose the hyper-parameters? There are many ways to do this, and this section will describe the most common and basic ones, while leaving more advanced techniques (like Implicit Differentiation) for later. Specifically, this section will: 1. Define the concepts of Grid and Random Hyper-parameter search. 2. Use Grid and Random search to optimize hyper-parameters of a model. 2. Distinguish when Randomized Search is much better than grid search. 3. Describe how Global Optimization procedures such as Bayesian

Optimization work. 4. Recognize why none of those at all work in High Dimensions and describe the “Curse of Dimensionality”

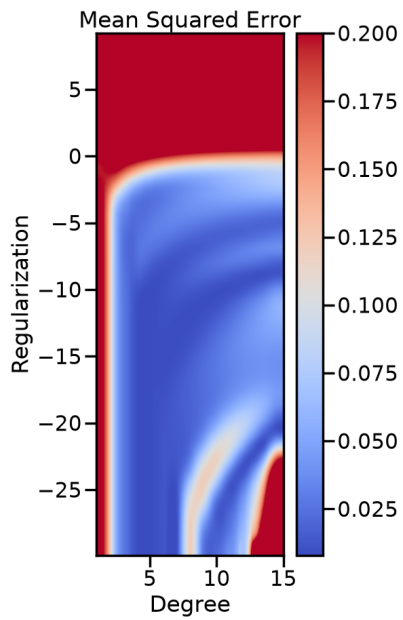
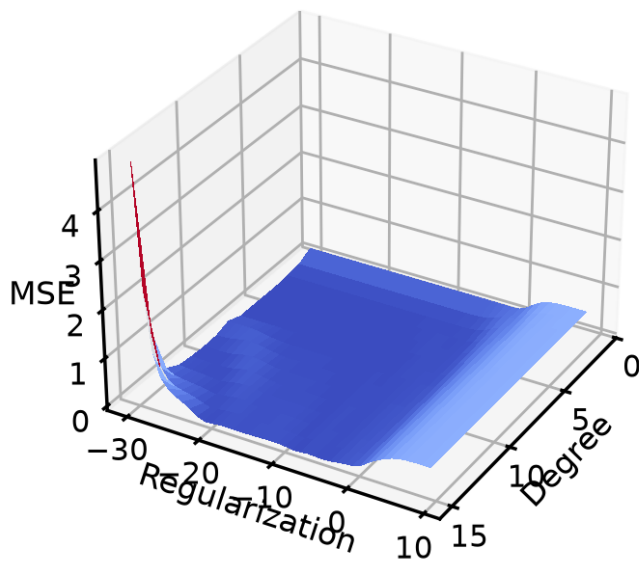
In future chapters once we cover more advanced derivative methods, we can discuss how to use tools like Implicit Differentiation to directly compute the gradient of the cross-validation score with respect to hyper-parameters, and then use this gradient to optimize the hyper-parameters using standard gradient-based optimization methods. However, for now, let’s focus on more basic derivative-free methods, since they are more widely used and easier to understand.

Let’s start by returning to our Polynomial example, and this time focus on finding the best combination of degree and penalty weight for a linear model.



2.4 What if we have more than one variable?

Let’s look at both polynomial degree and regularization weight



2.5 Optimization

At the end of the day, all this is doing is optimization/search over different parameters. How should we go about automating this? The most common and easy to implement method is

Grid Search, which will work OK for us when the number of possible hyper-parameters are small and have equal effect on the model performance:

parameters we could change:

```
memory
steps
transform_input
verbose
polynomial_features
linear_regression
polynomial_features__degree
polynomial_features__include_bias
polynomial_features__interaction_only
polynomial_features__order
linear_regression__alpha
linear_regression__copy_X
linear_regression__fit_intercept
linear_regression__max_iter
linear_regression__positive
linear_regression__random_state
linear_regression__solver
linear_regression__tol
```

estimator estimator:
estimator object

Pipeline(step...4(10000.0))))]

This is assumed to
implement the scikit-learn
estimator interface.
Either estimator needs to
provide a “score” function,
or “scoring” must be passed.

<code>param_grid</code> <code>param_grid</code>: dict or list of dictionaries Dictionary with parameters names ('str') as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings.	<pre>{'linear__regression__alpha': array([1.0000...000000000e+04]), 'polyno- mial__features__degree': [1, 2, ...], 'polyno- mial__features__include_bias': [True, False]}</pre>
---	---

scoring scoring: str, callable, 'neg_mean_squared_error'
list, tuple or dict,
default=None

Strategy to evaluate the
performance of the
cross-validated model on
the test set.

If 'scoring' represents a
single score, one can use:

- a single string (see
:ref:'scoring_string_names');
- a callable (see
:ref:'scoring_callable') that
returns a single value;
- 'None', the 'estimator's
:ref:'default evaluation
criterion
<scoring_api_overview>' is
used.

If 'scoring' represents
multiple scores, one can use:

- a list or tuple of unique
strings;
- a callable returning a
dictionary where the keys are
the metric
names and the values are the
metric scores;
- a dictionary with metric
names as keys and callables
as values.

See
:ref:'multimetric_grid_search'
for an example.

`cv` `cv`: int, cross-validation generator or an iterable, default=None

`ShuffleSplit(...ain_size=None)`

Determines the cross-validation splitting strategy.
Possible inputs for `cv` are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a ‘(Stratified)KFold’,
- :term:‘CV splitter’,
- an iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and “y” is either binary or multiclass, :class:‘StratifiedKFold’ is used. In all other cases, :class:‘KFold’ is used. These splitters are instantiated with ‘shuffle=False’ so the splits will be the same across calls.

Refer :ref:‘User Guide <cross_validation>’ for the various cross-validation strategies that can be used here.

.. versionchanged:: 0.22
“cv” default value if None changed from 3-fold to 5-fold.

<code>n_jobs</code>	<code>n_jobs: int,</code> <code>default=None</code>	None
---------------------	--	------

Number of jobs to run in parallel.
“None“ means 1 unless in a `:obj:'joblib.parallel_backend'` context.
“-1“ means using all processors. See `:term:'Glossary <n_jobs>'` for more details.

.. versionchanged:: v0.20
'n_jobs' default changed from 1 to None

refit refit: bool, str, or
callable, default=True

True

Refit an estimator using the
best found parameters on the
whole
dataset.

For multiple metric
evaluation, this needs to be a
'str' denoting the
scorer that would be used to
find the best parameters for
refitting
the estimator at the end.

Where there are
considerations other than
maximum score in
choosing a best estimator,
"refit" can be set to a
function which
returns the selected
"best_index_" given
"cv_results_". In that
case, the "best_estimator_"
and "best_params_" will be
set
according to the returned
"best_index_" while the
"best_score_"
attribute will not be
available.

The refitted estimator is
made available at the
"best_estimator_"
attribute and permits using
"predict" directly on this
"GridSearchCV" instance.

Also for multiple metric
evaluation, the attributes
"best_index_",
"best_score_" and
"best_params_"⁶⁹ will only be
available if
"refit" is set and all of them
will be determined w.r.t this
specific
scorer.

⁶⁹ Since "best_index_" is not a class attribute, it is not included in the list of attributes.

verbose verbose: int, 0
default=0

Controls the verbosity of information printed during fitting, with higher values yielding more detailed logging.

- 0 : no messages are printed;
- >=1 : summary of the total number of fits;
- >=2 : computation time for each fold and parameter candidate;
- >=3 : fold indices and scores;
- >=10 : parameter candidate indices and START messages before each fit.

`pre_dispatch pre_dispatch: '2*n_jobs'`
`int, or str,`
`default='2*n_jobs'`

Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A str, giving an expression as a function of `n_jobs`, as in `'2*n_jobs'`

error_score	error_score:	nan
'raise' or numeric,		
default=np.nan		

Value to assign to the score if an error occurs in estimator fitting.

If set to 'raise', the error is raised. If a numeric value is given,

FitFailedWarning is raised.

This parameter does not affect the refit step, which will always raise the error.

	<pre> return_train_score return_train_score: bool, default=False </pre> If “False“, the “cv_results_“ attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance. .. versionadded:: 0.19 .. versionchanged:: 0.21 Default value was changed from “True“ to “False“	False
Name	Type	Value

best_estimator_ best_estimator_: estimator	Pipeline	Pipeline(step...46257e-06)))]))
---	----------	---------------------------------

Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if “refit=False”.

See “refit” parameter for more information on allowed values.

best_index_ best_index_: int	int64	np.int64(127)
---------------------------------	-------	---------------

The index (of the “cv_results_” arrays) which corresponds to the best candidate parameter setting.

The dict at “search.cv_results_[‘params’][search.best_index_]” gives the parameter setting for the best model, that gives the highest mean score (“search.best_score”).

For multi-metric evaluation, this is present only if “refit” is specified.

best_params_ best_params_: dict	dict	{'li...ha': np.float64(3....638046257e- 06), 'po...ee': 4, 'po...as': False}
Parameter setting that gave the best results on the hold out data.		
For multi-metric evaluation, this is present only if “refit“ is specified.		
best_score_ best_score_: float	float64	-0.00845
Mean cross-validated score of the best_estimator		
For multi-metric evaluation, this is present only if “refit“ is specified.		
This attribute is not available if “refit“ is a function.		

```
cv_results_ cv_results_:      dict      {'me...me': array([0.0020...,
dict of numpy (masked)      0.0018765 ]), 'me...me':
ndarrays      array([0.0008..., 0.00082632]),
      'me...re': array([-0.181...
      -0.59315893]), 'pa...ha':
      masked_array(...1_value=1e+20),
      ...}
```

A dict with keys as column headers and values as columns, that can be imported into a pandas “DataFrame”.

For instance the below given table

```
+-----+-----+-----+
-- +-----+-- +-----+
+
|param_kernel|param_gamma|param_degree|split0_test_score|...|rank_t...|
+=====+=====+=====+=====+=====+=====+
| 'poly' | -- | 2 | 0.80 |...| 2 |
+-----+-----+-----+
-- +-----+-- +-----+
+
| 'poly' | -- | 3 | 0.70 |...| 4 |
+-----+-----+-----+
-- +-----+-- +-----+
+
| 'rbf' | 0.1 | -- | 0.80 |...| 3 |
+-----+-----+-----+
-- +-----+-- +-----+
+
| 'rbf' | 0.2 | -- | 0.93 |...| 1 |
+-----+-----+-----+
-- +-----+-- +-----+
+
```

will be represented by a “cv_results_” dict of::

```
{
'param_kernel':
masked_array(data = ['poly',
'poly', 'rbf', 'rbf'],
mask = [False False False
False]...)
'param_gamma':
masked_array(data = [-- --
0.1 0.2],
mask = [ True True False
False]...),
'param_degree':
masked_array(data = [2.0
3.0 -- --],
mask = [False False True
False False True])
}
```

multimetric__ multimetric__: bool	bool	False
Whether or not the scorers compute several metrics. n_features_in__ n_features_in_: int	int	1
Number of features seen during :term:‘fit’. Only defined if ‘best_estimator_‘ is defined (see the documentation for the ‘refit’ parameter for more details) and that ‘best_estimator_‘ exposes ‘n_features_in_‘ when fit.		
.. versionadded:: 0.24 n_splits__ n_splits_: int	int	20
The number of cross-validation splits (folds/iterations). refit_time__ refit_time_: float	float	0.001999
Seconds used for refitting the best model on the whole dataset.		
This is present only if “refit” is not False.		
.. versionadded:: 0.20		

scorer_	scorer_:	function or a dict	__Scorer	make_scorer(m...hod='predict')
---------	----------	--------------------	----------	--------------------------------

Scorer function used on the held out data to choose the best parameters for the model.

For multi-metric evaluation, this attribute holds the validated “scoring“ dict which maps the scorer key to the scorer callable.

degree	degree:	int or tuple (min_degree, max_degree), default=2	4
--------	---------	--	---

If a single int is given, it specifies the maximal degree of the polynomial features. If a tuple ‘(min_degree, max_degree)’ is passed, then ‘min_degree’ is the minimum and ‘max_degree’ is the maximum polynomial degree of the generated features. Note that ‘min_degree=0’ and ‘min_degree=1’ are equivalent as outputting the degree zero term is determined by ‘include_bias’.

include_bias	include_bias:	False
bool, default=True		
If 'True' (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).		
interaction_only	interaction_only:	False
bool, default=False		
If 'True', only interaction features are produced: features that are products of at most 'degree' *distinct* input features, i.e. terms with power of 2 or higher of the same input feature are excluded:		
- included: 'x[0]', 'x[1]', 'x[0] * x[1]', etc.		
- excluded: 'x[0] ** 2', 'x[0] ** 2 * x[1]', etc.		
order	order: {'C', 'F'},	'C'
default='C'		
Order of output array in the dense case. 'F' order is faster to compute, but may slow down subsequent estimators.		
.. versionadded:: 0.21		

Name	Type	Value
<code>n_features_in_</code> <code>n_features_in_</code> : int	int	1
Number of features seen during :term:‘fit‘.		
<code>.. versionadded:: 0.24</code> <code>n_output_features_</code> <code>n_output_features_</code> : int	int	4
The total number of polynomial output features. The number of output features is computed by iterating over all suitably sized combinations of input features.		
<code>powers_</code> <code>powers_</code> : ndarray of shape (‘ <code>n_output_features_</code> ’, ‘ <code>n_features_in_</code> ’)	ndarray[int64](4, 1)	[[1], [2], [3], [4]]
‘ <code>powers_[i, j]</code> ’ is the exponent of the jth input in the ith output.		

x_0
 x_0^2
 x_0^3
 x_0^4

alpha	alpha: float or array-like of shape (n_targets,), default=1.0	np.float64(3....638046257e-06)
-------	---	--------------------------------

Constant that multiplies the L2 term, controlling regularization strength. ‘alpha’ must be a non-negative float i.e. in ‘[0, inf)’.

When ‘alpha = 0’, the objective is equivalent to ordinary least squares, solved by the :class:‘LinearRegression’ object. For numerical reasons, using ‘alpha = 0’ with the ‘Ridge’ object is not advised.

Instead, you should use the :class:‘LinearRegression’ object.

If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

See

:ref:‘sphinx_glr_auto_examples_linear_model_plot_ridge_coeffs.py’ for an illustration of the effect of alpha on the model coefficients.

<code>fit_intercept</code> <code>fit_intercept</code> : bool, default=True	True
Whether to fit the intercept for this model. If set to false, no intercept will be used in calculations (i.e. “X” and “y” are expected to be centered). <code>copy_X</code> <code>copy_X</code> : bool, default=True	True
If True, X will be copied; else, it may be overwritten. <code>max_iter</code> <code>max_iter</code> : int, default=None	None
Maximum number of iterations for conjugate gradient solver. For 'sparse_cg' and 'lsqr' solvers, the default value is determined by <code>scipy.sparse.linalg</code>. For 'sag' solver, the default value is 1000. For 'lbfgs' solver, the default value is 15000.	

tol tol: float, default=1e-4 0.0001

The precision of the solution ('coef_') is determined by 'tol' which specifies a different convergence criterion for each solver:

- 'svd': 'tol' has no impact.
 - 'cholesky': 'tol' has no impact.
 - 'sparse_cg': norm of residuals smaller than 'tol'.
 - 'lsqr': 'tol' is set as atol and btol of `scipy.sparse.linalg.lsqr`, which control the norm of the residual vector in terms of the norms of matrix and coefficients.
 - 'sag' and 'saga': relative change of coef smaller than 'tol'.
 - 'lbfgs': maximum of the absolute (projected) gradient= $\max|\text{residuals}|$ smaller than 'tol'.
- .. versionchanged:: 1.2
Default value changed from 1e-3 to 1e-4 for consistency with other linear models.

```
solver solver: {'auto', 'svd', 'auto',  
'cholesky', 'lsqr', 'sparse_cg',  
'sag', 'saga', 'lbfgs'},  
default='auto'
```

Solver to use in the
computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. It is the most stable solver, in particular more stable for singular matrices than 'cholesky' at the cost of being slower.
- 'cholesky' uses the standard :func:`scipy.linalg.solve` function to obtain a closed-form solution.
- 'sparse_cg' uses the conjugate gradient solver as found in :func:`scipy.sparse.linalg.cg`. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set 'tol' and 'max_iter').
- 'lsqr' uses the dedicated regularized least-squares routine :func:`scipy.sparse.linalg.lsqr`. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its improved, unbiased version named SAGA. Both

	positive: bool, default=False	False
	When set to “True“, forces the coefficients to be positive. Only 'lbfgs' solver is supported in this case.	
	random_state random_state: int, RandomState instance, default=None	None
	Used when “solver“ == 'sag' or 'saga' to shuffle the data. See :term:‘Glossary <random_state>’ for details.	
	.. versionadded:: 0.17 ‘random_state’ to support Stochastic Average Gradient.	

Name	Type	Value
coef_ coef_: ndarray of shape (n_features,) or (n_targets, n_features)	ndarray[float64](4,)	[2.38,-27.44, 40. , -15.95]
Weight vector(s). intercept_ intercept_: float or ndarray of shape (n_targets,)	float64	0.9257
Independent term in decision function. Set to 0.0 if “fit_intercept = False“.		
n_features_in_ n_features_in_: int	int	4
Number of features seen during :term:‘fit‘.		
.. versionadded:: 0.24		

<code>n_iter_</code>	<code>n_iter_</code> : None or ndarray of shape (<code>n_targets</code> ,)	NoneType	None
----------------------	---	----------	------

Actual number of iterations
for each target. Available
only for
'sag' and 'lsqr' solvers. Other
solvers will return None.

<code>.. versionadded:: 0.17</code>	
<code>solver_</code>	<code>solver_</code> : str
	'ch...ky'

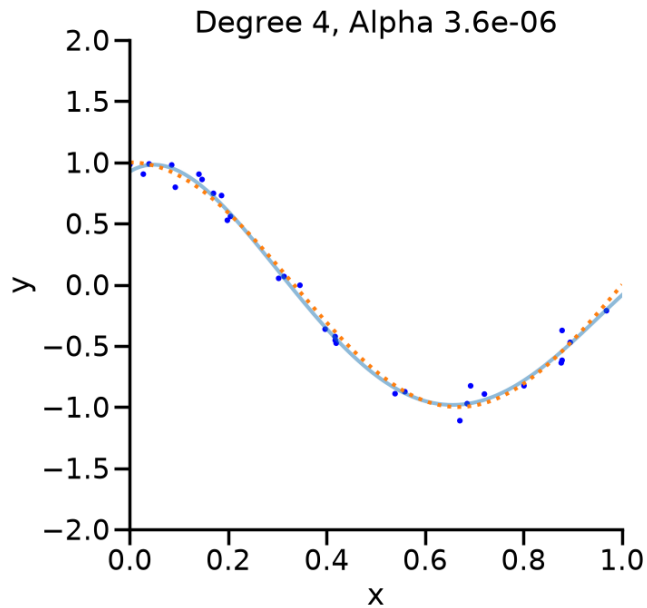
The solver that was used at
fit time by the computational
routines.

`.. versionadded:: 1.5`

```
{'linear_regression__alpha': np.float64(3.5938136638046257e-06),  
 'polynomial_features__degree': 4,  
 'polynomial_features__include_bias': False}
```

Best MSE for Grid Search: 8.45e-03

```
array([ 0.92597995,  0.97112185,  0.97915236,  0.95343228,  0.94185444,  
        0.8235348 ,  0.80306679,  0.72117767,  0.65636685,  0.60679754,  
        0.57933728,  0.10937232,  0.05389104, -0.10522154, -0.34633312,  
       -0.43501599, -0.43622117, -0.44424987, -0.84518448, -0.88774347,  
       -0.98056731, -0.97399224, -0.96929348, -0.94126767, -0.78277808,  
       -0.54827761, -0.54221084, -0.54212267, -0.48316009, -0.20435414])
```

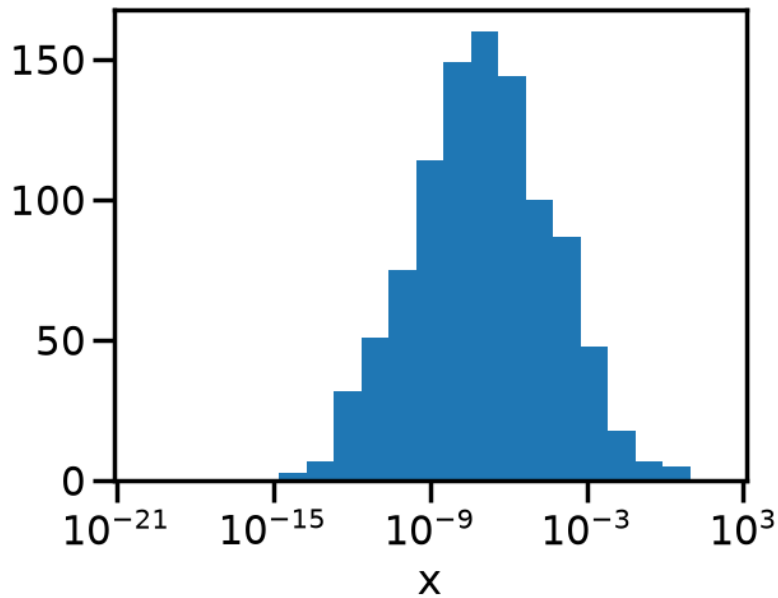


2.6 Randomized Search

In reality, grid search is wasteful and not easy to control. A better (and still easy way) is to randomize the search. To do this, we will need to tell the model how to sample randomly in the hyper-parameter space:

Need something whose logarithmic distribution we can control – *e.g.*, something on the positive reals? How about a lognormal?

$$\mathcal{N}(\ln x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp \left[-\frac{(\ln x - \mu)^2}{2\sigma^2} \right].$$



estimator estimator:
estimator object

Pipeline(step...4(10000.0))))]

An object of that type is
instantiated for each grid
point.
This is assumed to
implement the scikit-learn
estimator interface.
Either estimator needs to
provide a “score“ function,
or “scoring“ must be passed.

<code>param_distributions</code> <code>param_distributions</code>: dict or list of dicts Dictionary with parameters names ('str') as keys and distributions or lists of parameters to try. Distributions must provide a "rvs" method for sampling (such as those from <code>scipy.stats.distributions</code>). If a list is given, it is sampled uniformly. If a list of dicts is given, first a dict is sampled uniformly, and then a parameter is sampled using that dict as above. <code>n_iter</code> <code>n_iter</code>: int, default=10 Number of parameter settings that are sampled. <code>n_iter</code> trades off runtime vs quality of the solution.	<pre>{'linear__regression__alpha': <scipy.stats....001DA1FBB69D0>, 'polynomial_features__degree': <scipy.stats....001DA1FAF0350>, 'polynomial_features__include_bias': [True, False]}</pre> 300
---	--

scoring scoring: str, callable, 'neg_mean_squared_error'
list, tuple or dict,
default=None

Strategy to evaluate the
performance of the
cross-validated model on
the test set.

If 'scoring' represents a
single score, one can use:

- a single string (see
:ref:'scoring_string_names');
- a callable (see
:ref:'scoring_callable') that
returns a single value;
- 'None', the 'estimator's
:ref:'default evaluation
criterion
<scoring_api_overview>' is
used.

If 'scoring' represents
multiple scores, one can use:

- a list or tuple of unique
strings;
- a callable returning a
dictionary where the keys are
the metric
names and the values are the
metric scores;
- a dictionary with metric
names as keys and callables
as values.

See
:ref:'multimetric_grid_search'
for an example.

If None, the estimator's score
method is used.

`cv` `cv`: int, cross-validation generator or an iterable, default=None

`ShuffleSplit(...ain_size=None)`

Determines the cross-validation splitting strategy.
Possible inputs for `cv` are:

- None, to use the default 5-fold cross validation,
- integer, to specify the number of folds in a ‘(Stratified)KFold’,
- :term:‘CV splitter’,
- an iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if the estimator is a classifier and “y” is either binary or multiclass, :class:‘StratifiedKFold’ is used. In all other cases, :class:‘KFold’ is used. These splitters are instantiated with ‘shuffle=False’ so the splits will be the same across calls.

Refer :ref:‘User Guide <cross_validation>’ for the various cross-validation strategies that can be used here.

.. versionchanged:: 0.22
“cv” default value if None changed from 3-fold to 5-fold.

<code>n_jobs</code>	<code>n_jobs: int,</code> <code>default=None</code>	None
---------------------	--	------

Number of jobs to run in parallel.
“None“ means 1 unless in a `:obj:'joblib.parallel_backend'` context.
“-1“ means using all processors. See `:term:'Glossary <n_jobs>'` for more details.

.. versionchanged:: v0.20
'n_jobs' default changed from 1 to None

refit refit: bool, str, or
callable, default=True

True

Refit an estimator using the
best found parameters on the
whole
dataset.

For multiple metric
evaluation, this needs to be a
'str' denoting the
scorer that would be used to
find the best parameters for
refitting
the estimator at the end.

Where there are
considerations other than
maximum score in
choosing a best estimator,
"refit" can be set to a
function which
returns the selected
"best_index_" given the
"cv_results_". In that
case, the "best_estimator_"
and "best_params_" will be
set
according to the returned
"best_index_" while the
"best_score_"
attribute will not be
available.

The refitted estimator is
made available at the
"best_estimator_"
attribute and permits using
"predict" directly on this
"RandomizedSearchCV"
instance.

Also for multiple metric
evaluation, the attributes
"best_index_",
"best_score_" and
"best_params_" will only be
available if
"refit" is set and all of them
will be determined w.r.t this
specific
scorer.

`verbose` `verbose: int, default 0`
`= 0`

Controls the verbosity of information printed during fitting, with higher values yielding more detailed logging.

- 0 : no messages are printed;
- >=1 : summary of the total number of fits;
- >=2 : computation time for each fold and parameter candidate;
- >=3 : fold indices and scores;
- >=10 : parameter candidate indices and START messages before each fit.

`pre_dispatch pre_dispatch: '2*n_jobs'`
`int, or str,`
`default='2*n_jobs'`

Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A str, giving an expression as a function of `n_jobs`, as in `'2*n_jobs'`

<code>random_state</code> <code>random_state: int, RandomState instance or None, default=None</code> Pseudo random number generator state used for random uniform sampling from lists of possible values instead of <code>scipy.stats</code> distributions. Pass an int for reproducible output across multiple function calls. See :term:‘Glossary <random_state>’.	None
<code>error_score error_score: 'raise' or numeric, default=np.nan</code> Value to assign to the score if an error occurs in estimator fitting. If set to 'raise', the error is raised. If a numeric value is given, <code>FitFailedWarning</code> is raised. This parameter does not affect the <code>refit</code> step, which will always raise the error.	nan

	<pre> return_train_score return_train_score: bool, default=False </pre> If “False“, the “cv_results_“ attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance. .. versionadded:: 0.19 .. versionchanged:: 0.21 Default value was changed from “True“ to “False“	False
Name	Type	Value

best_estimator_ best_estimator_: estimator	Pipeline	Pipeline(step...78128e-06)))]))
Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if “refit=False”. For multi-metric evaluation, this attribute is present only if “refit” is specified. See “refit” parameter for more information on allowed values.		
best_index_ best_index_: int	int64	np.int64(280)
The index (of the “cv_results_” arrays) which corresponds to the best candidate parameter setting. The dict at “search.cv_results_[‘params’][search.best_index_]” gives the parameter setting for the best model, that gives the highest mean score (“search.best_score”). For multi-metric evaluation, this is not available if “refit” is “False”. See “refit” parameter for more information.		

best_params_ best_params_: dict	dict	{'li...ha': np.float64(2....681278128e- 06), 'po...ee': 4, 'po...as': True}
Parameter setting that gave the best results on the hold out data.		
For multi-metric evaluation, this is not available if “refit” is “False“. See “refit” parameter for more information.		
best_score_ best_score_: float	float64	-0.006998
Mean cross-validated score of the best_estimator.		
For multi-metric evaluation, this is not available if “refit” is “False“. See “refit” parameter for more information.		
This attribute is not available if “refit” is a function.		

cv_results_	cv_results_:	dict	{'me...me': array([0.0017..., 0.0019526]), 'me...me': array([0.0009..., 0.0007755]), 'me...re': array([- 1.357...11817247e-01]), 'pa...ha': masked_array(...l_value=1e+20), ...}
-------------	--------------	------	---

A dict with keys as column headers and values as columns, that can be imported into a pandas “DataFrame”.

For instance the below given table

```
+-----+-----+-----+
+-----+---+-----+
| param_kernel |
| param_gamma |
| split0_test_score |
| ...|rank_test_score|
+=====+=====+=====+=====+=====+
| 'rbf' | 0.1 | 0.80 |...| 1 |
+-----+-----+-----+
+-----+---+-----+
| 'rbf' | 0.2 | 0.84 |...| 3 |
+-----+-----+-----+
+-----+---+-----+
| 'rbf' | 0.3 | 0.70 |...| 2 |
+-----+-----+-----+
+-----+---+-----+
```

will be represented by a
“cv_results_“ dict of::

```
{
'param_kernel' :
masked_array(data = ['rbf',
'rbf', 'rbf'],
mask = False),
'param_gamma' :
masked_array(data = [0.1
0.2 0.3], mask = False),
'split0_test_score' : [0.80,
0.84, 0.70],
'split1_test_score' : [0.82,
0.50, 0.70],
'mean_test_score' : [0.81,
0.67, 0.70],
'std_test_score' : [0.01, 0.24,
0.00],
'rank_test_score' : [1, 3, 2],
'split0_train_score' : [0.80,
0.92, 0.70],
'split1_train_score' : [0.82,
```

multimetric__ multimetric__: bool	bool	False
Whether or not the scorers compute several metrics. n_features_in__ n_features_in_: int	int	1
Number of features seen during :term:‘fit’. Only defined if ‘best_estimator_‘ is defined (see the documentation for the ‘refit’ parameter for more details) and that ‘best_estimator_‘ exposes ‘n_features_in_‘ when fit.		
.. versionadded:: 0.24 n_splits__ n_splits_: int	int	20
The number of cross-validation splits (folds/iterations). refit_time__ refit_time_: float	float	0.003007
Seconds used for refitting the best model on the whole dataset.		
This is present only if “refit” is not False.		
.. versionadded:: 0.20		

scorer__ scorer__: function or a dict	__Scorer	make_scorer(m...hod='predict')
---------------------------------------	----------	--------------------------------

Scorer function used on the held out data to choose the best parameters for the model.

For multi-metric evaluation, this attribute holds the validated “scoring“ dict which maps the scorer key to the scorer callable.

degree degree: int or tuple (min_degree, max_degree), default=2	4
---	---

If a single int is given, it specifies the maximal degree of the polynomial features. If a tuple ‘(min_degree, max_degree)’ is passed, then ‘min_degree’ is the minimum and ‘max_degree’ is the maximum polynomial degree of the generated features. Note that ‘min_degree=0’ and ‘min_degree=1’ are equivalent as outputting the degree zero term is determined by ‘include_bias’.

interaction_only interaction_only: bool, default=False If 'True', only interaction features are produced: features that are products of at most 'degree' *distinct* input features, i.e. terms with power of 2 or higher of the same input feature are excluded: - included: 'x[0]', 'x[1]', 'x[0] * x[1]', etc. - excluded: 'x[0] ** 2', 'x[0] ** 2 * x[1]', etc. include_bias include_bias: bool, default=True If 'True' (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model). order order: {'C', 'F'}, default='C' Order of output array in the dense case. 'F' order is faster to compute, but may slow down subsequent estimators. .. versionadded:: 0.21	False True 'C'
---	-------------------------------------

Name	Type	Value
<code>n_features_in_</code> <code>n_features_in_</code> : int	int	1
Number of features seen during :term:`fit`.		
<code>.. versionadded:: 0.24</code> <code>n_output_features_</code> <code>n_output_features_</code> : int	int	5
The total number of polynomial output features. The number of output features is computed by iterating over all suitably sized combinations of input features.		
<code>powers_</code> <code>powers_</code> : ndarray of shape (<code>n_output_features_</code> , <code>n_features_in_</code>)	ndarray[int64](5, 1)	[[0], [1], [2], [3], [4]]
<code>'powers_[i, j]'</code> is the exponent of the jth input in the ith output.		

1
 x_0
 x_0^2
 x_0^3
 x_0^4

alpha	alpha: float or array-like of shape (n_targets,), default=1.0	np.float64(2....681278128e-06)
-------	---	--------------------------------

Constant that multiplies the L2 term, controlling regularization strength. ‘alpha’ must be a non-negative float i.e. in ‘[0, inf)’.

When ‘alpha = 0’, the objective is equivalent to ordinary least squares, solved by the :class:‘LinearRegression’ object. For numerical reasons, using ‘alpha = 0’ with the ‘Ridge’ object is not advised. Instead, you should use the :class:‘LinearRegression’ object.

If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

See :ref:‘sphinx_glr_auto_examples_linear_model_plot_ridge_coeffs.py’ for an illustration of the effect of alpha on the model coefficients.

<code>fit_intercept</code> <code>fit_intercept</code> : bool, default=True	True
Whether to fit the intercept for this model. If set to false, no intercept will be used in calculations (i.e. “X” and “y” are expected to be centered). <code>copy_X</code> <code>copy_X</code> : bool, default=True	True
If True, X will be copied; else, it may be overwritten. <code>max_iter</code> <code>max_iter</code> : int, default=None	None
Maximum number of iterations for conjugate gradient solver. For 'sparse_cg' and 'lsqr' solvers, the default value is determined by <code>scipy.sparse.linalg</code>. For 'sag' solver, the default value is 1000. For 'lbfgs' solver, the default value is 15000.	

tol tol: float, default=1e-4 0.0001

The precision of the solution ('coef_') is determined by 'tol' which specifies a different convergence criterion for each solver:

- 'svd': 'tol' has no impact.
- 'cholesky': 'tol' has no impact.
- 'sparse_cg': norm of residuals smaller than 'tol'.
- 'lsqr': 'tol' is set as atol and btol of `scipy.sparse.linalg.lsqr`, which control the norm of the residual vector in terms of the norms of matrix and coefficients.
- 'sag' and 'saga': relative change of coef smaller than 'tol'.
- 'lbfgs': maximum of the absolute (projected) gradient= $\max|\text{residuals}|$ smaller than 'tol'.

.. versionchanged:: 1.2
Default value changed from 1e-3 to 1e-4 for consistency with other linear models.

```
solver solver: {'auto', 'svd', 'auto',  
'cholesky', 'lsqr', 'sparse_cg',  
'sag', 'saga', 'lbfgs'},  
default='auto'
```

Solver to use in the
computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. It is the most stable solver, in particular more stable for singular matrices than 'cholesky' at the cost of being slower.
- 'cholesky' uses the standard :func:`scipy.linalg.solve` function to obtain a closed-form solution.
- 'sparse_cg' uses the conjugate gradient solver as found in :func:`scipy.sparse.linalg.cg`. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set 'tol' and 'max_iter').
- 'lsqr' uses the dedicated regularized least-squares routine :func:`scipy.sparse.linalg.lsqr`. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its improved, unbiased version named SAGA. Both

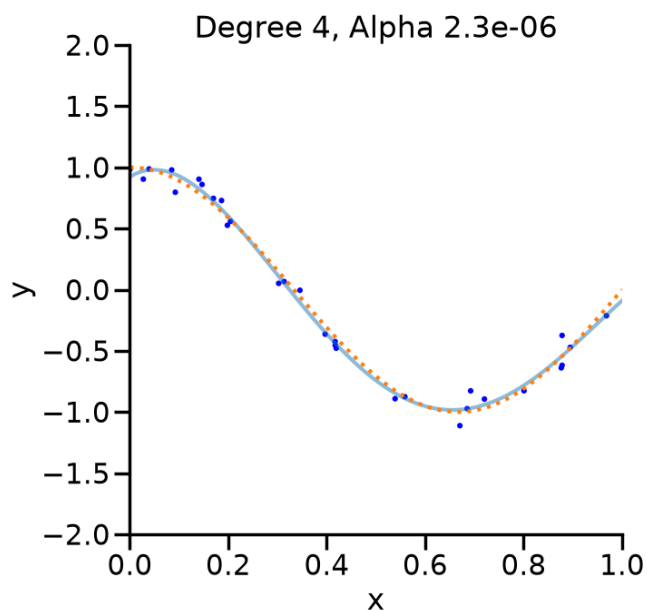
	positive: bool, default=False	False
	When set to “True“, forces the coefficients to be positive. Only 'lbfgs' solver is supported in this case.	
	random_state random_state: int, RandomState instance, default=None	None
	Used when “solver“ == 'sag' or 'saga' to shuffle the data. See :term:‘Glossary <random_state>‘ for details.	
	.. versionadded:: 0.17 ‘random_state‘ to support Stochastic Average Gradient.	

Name	Type	Value
coef_ coef_: ndarray of shape (n_features,) or (n_targets, n_features)	ndarray[float64](5,)	[0. , 2.44,-27.72, 40.46,-16.19]
Weight vector(s). intercept_ intercept_: float or ndarray of shape (n_targets,)	float64	0.9231
Independent term in decision function. Set to 0.0 if “fit_intercept = False“.		
n_features_in_ n_features_in_: int	int	5
Number of features seen during :term:‘fit‘.		
.. versionadded:: 0.24		

<code>n_iter_</code>	<code>n_iter_</code> : None or ndarray of shape (n_targets,)	NoneType	None
Actual number of iterations for each target. Available only for 'sag' and 'lsqr' solvers. Other solvers will return None.			
<code>.. versionadded::</code>	0.17		
<code>solver_</code>	<code>solver_</code> : str	str	'ch...ky'
The solver that was used at fit time by the computational routines.			
<code>.. versionadded::</code>	1.5		

```
{'linear_regression__alpha': np.float64(2.3284176681278128e-06),
 'polynomial_features__degree': 4,
 'polynomial_features__include_bias': True}
```

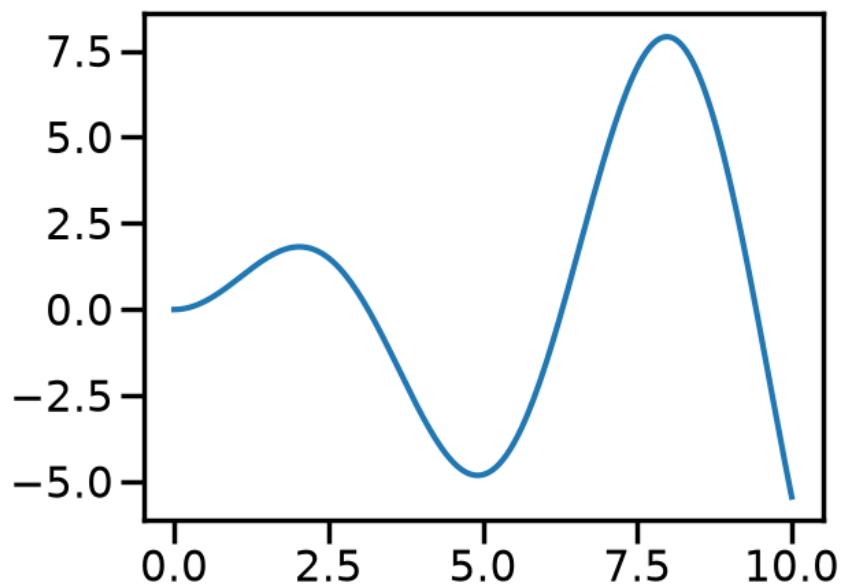
Best MSE for Random Search: 7.00e-03

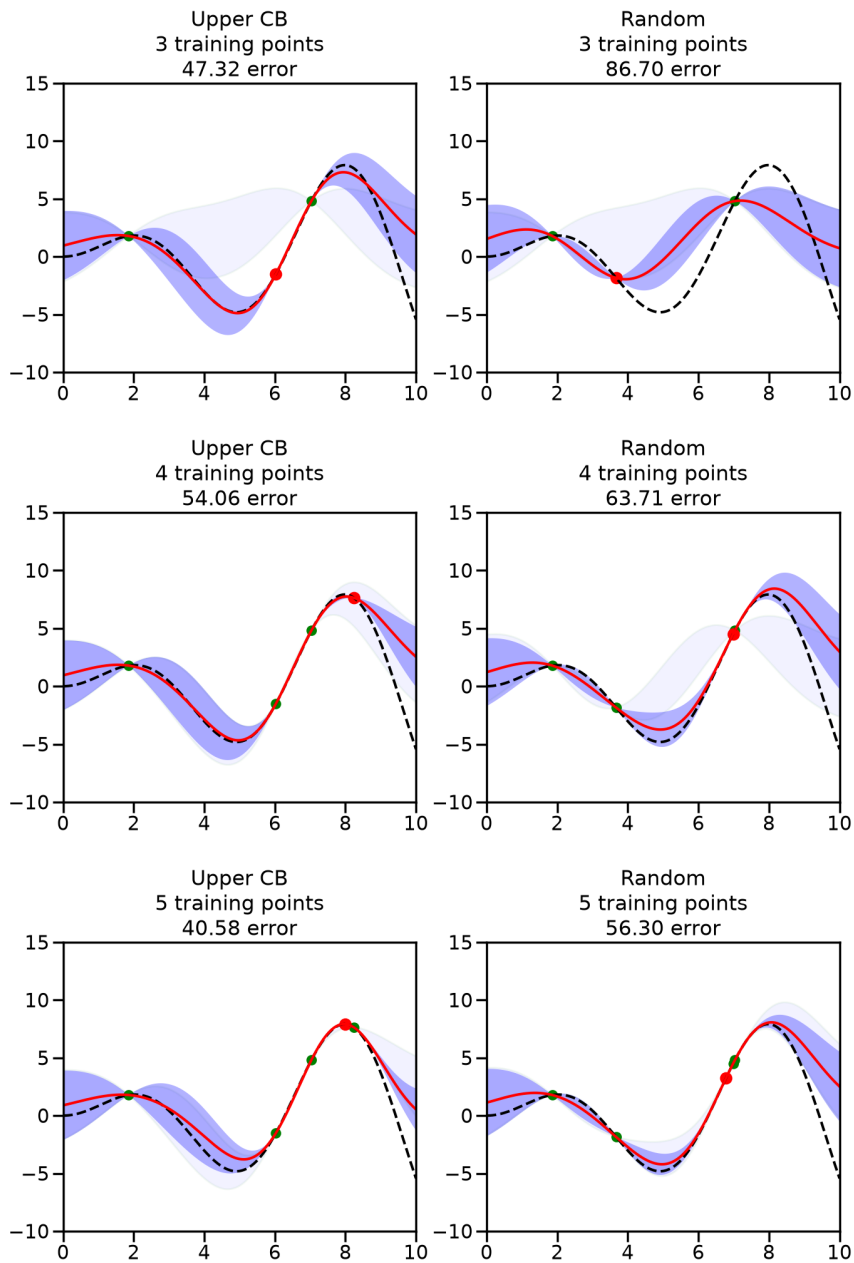


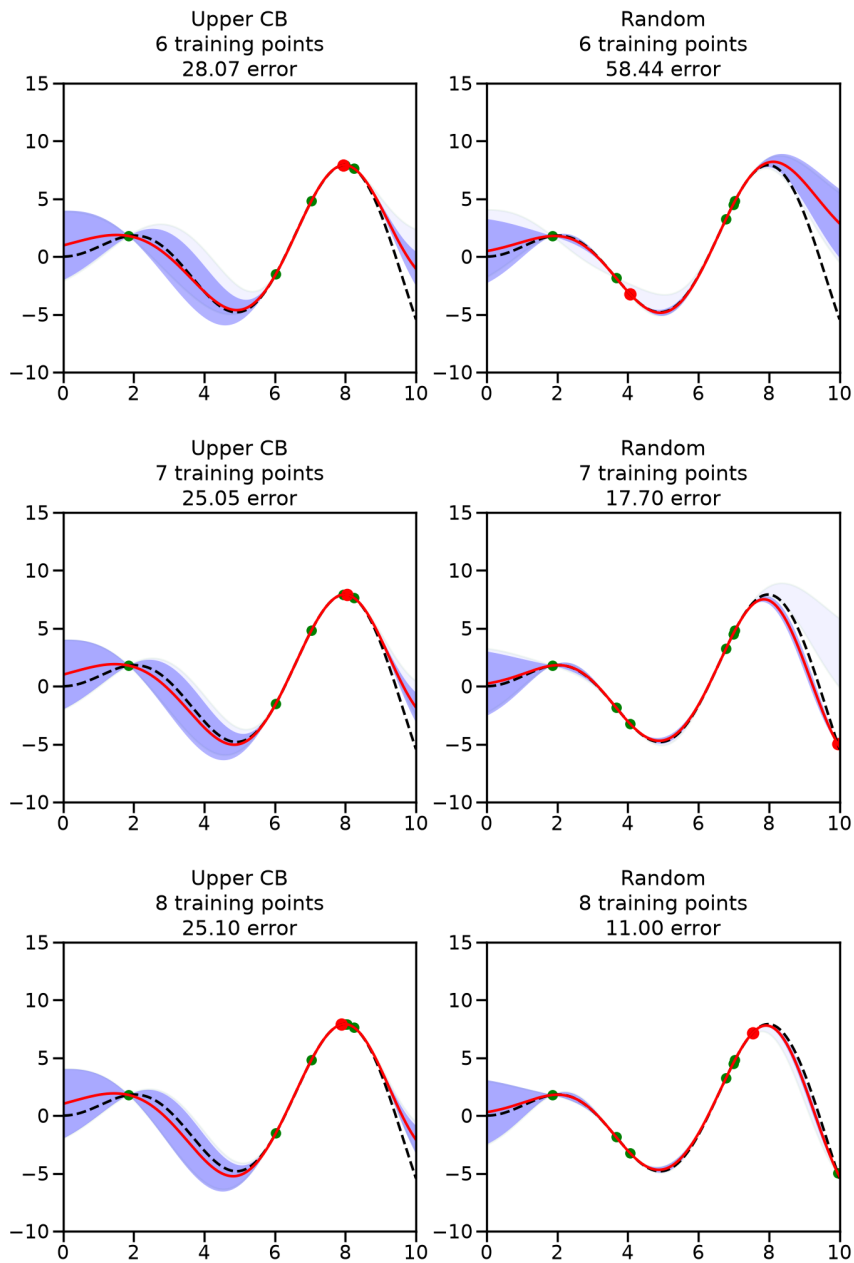
2.7 Global Bayesian Optimization

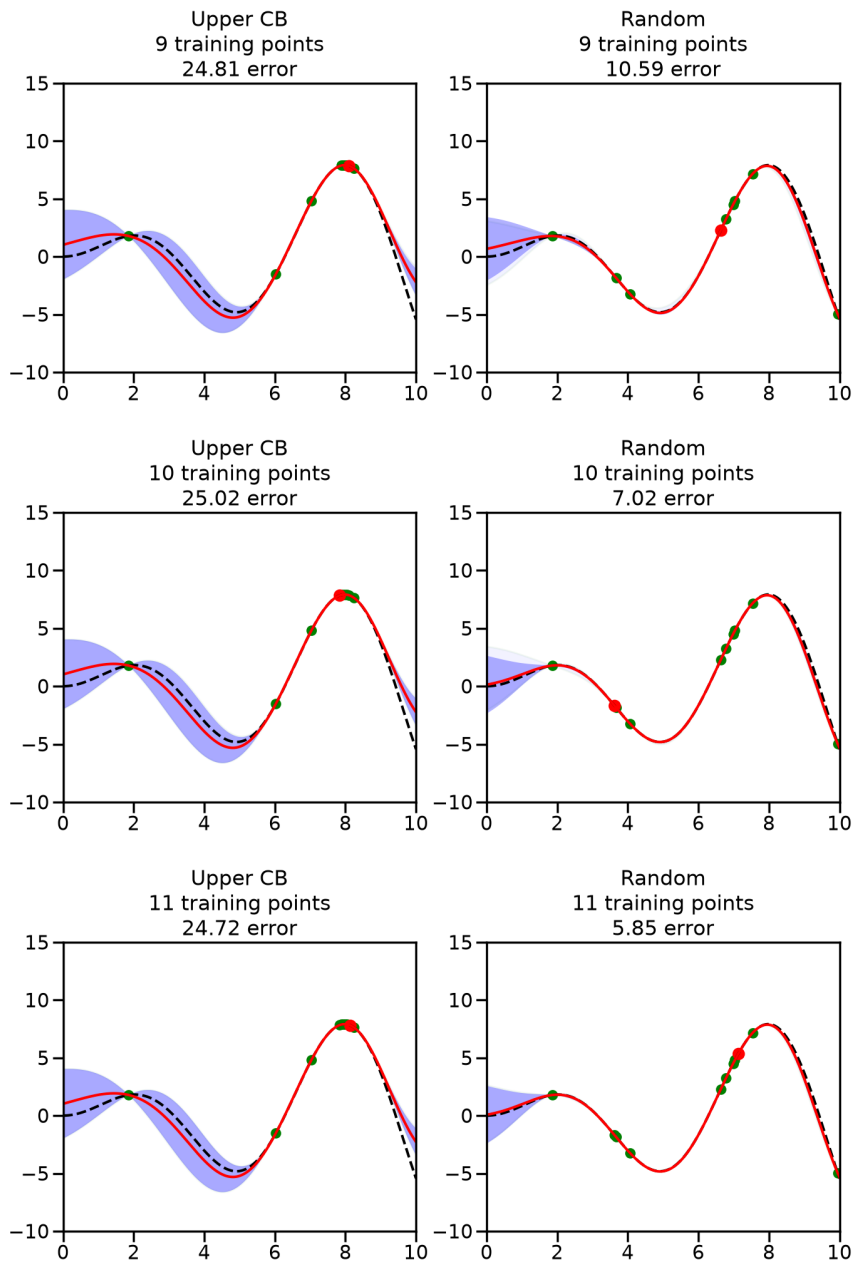
Surely, since we are essentially doing optimization, we could approach hyper-parameter selection as an optimization problem as well, right?

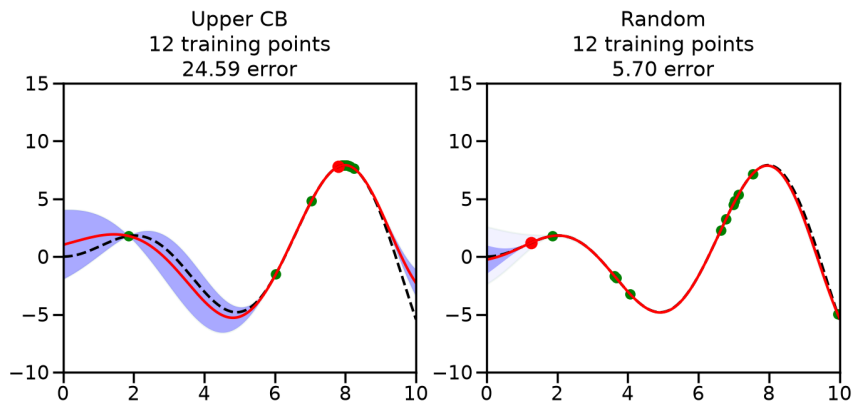
Enter techniques like Global Bayesian Optimization below:





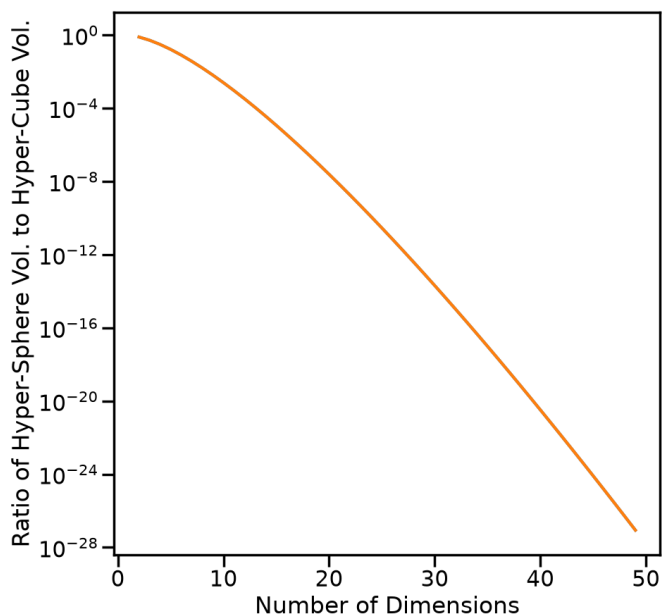






2.8 The Curse of Dimensionality

While the above approaches seem to work well on the small examples we showed, in practice, as the number of hyper-parameters increases we run into (sometimes exponentially) larger problems related to something called the “Curse of Dimensionality.” To get a rough sense of how this works and why it makes search (and especially sampling) difficult, let’s look at one example, which is the ratio between the volume of a sphere with diameter 1 and the volume of a cube with sides length 1. That is, you are essentially fitting a hyper-sphere exactly inside of a hyper-cube and then asking how their volumes compare. We will compute their ratio as N (the number of dimensions) increases:



What we can see in the above is that as N increases most of the volume of the space is actually contained in the **corners** of the hypercube, and thus if we try to sample uniformly in the hyper-volume (or have Bayesian Optimization sample points where it is uncertain), then most of the samples will end up being outside of the main “sphere” at the center of our hyperparameter space. You can see similar effects of other properties such as:

- The distance ratio between min and max distances among points randomly sampled in the space.
- How this affects which points are “near” or “far” from a given point (it won’t change the relative ordering, but the comparative distances will get compressed such that every point looks far away)
- The effect on how sampling points from a Gaussian Density function end up concentrating on “rings” within the hyper-parameter space (see below Exercise)

All of these effects point toward being careful how you interpret space filling strategies when the number of dimensions in the space becomes large.

2.9 Exercises

The exercises reuse the data and code from this chapter, so run the chapter’s cells first. Rough time estimates are in the titles; the last one is meant as a longer take-home task. Worked solutions are in the Exercise Solutions appendix.

Exercise 1 (~5 min): Which split would you use?

For each situation, name the cross-validation strategy you would use (shuffled K-fold, stratified K-fold, group K-fold, or a time series split), say what you would group or stratify on if anything, and describe what data might leak if you just used a plain shuffled split.

- (a) A strain gauge on a bridge has logged one reading per minute for a year; you want to predict the reading one hour ahead.
- (b) Six turbine blades were each laser-scanned 30 times; the model should predict the surface roughness of blades that have not been scanned yet.
- (c) Weld inspection images, of which 2 percent show a defect, and you want to train a model to classify defective versus sound welds based on the images.
- (d) Forty tensile specimens, each cast and tested independently, and you wish to build a model to predict the yield stress of that specimen.
- (e) You buy four different weld inspection imaging machines and deploy each at four production plants, then build a model that can predict their inspection accuracy based on data you collect in production. You wish to estimate how a newly purchased inspection machine might do at a fifth plant, were you to buy another.

- (f) Twenty battery cells have been cycled for months, and you want to forecast the remaining life of *these same cells* from their history so far.

Exercise 2 (~3 min): Read the validation curve

Go back to the one-dimensional sweep over α for the 15-degree polynomial (the cell that produced `alphas_1d` and `scores_1d`) and to the two-dimensional map over degree and α .

- (a) Read off, or compute with `np.argmin`, the value of α with the lowest cross-validated MSE in the 1D sweep.
- (b) The curve is U-shaped. Explain what goes wrong at the right end (large α) and at the left end (tiny α), using the words *bias* and *variance*.
- (c) Look closely at the left half of the curve. Between about 10^{-9} and 10^{-3} it is nearly flat, yet below 10^{-9} it shoots up by two orders of magnitude. Why does a penalty as tiny as 10^{-9} make such a large difference, and why does nothing much change over the next six decades? (Think about the size of the unregularized coefficients of a 15-degree polynomial on $[0, 1]$, which you saw in the previous chapter.)

Exercise 3 (~3 min): Count the fits

Suppose each model fit in the grid search were not a polynomial but a 30-second finite-element simulation.

- (a) From the `parameters` dictionary of the grid search (15 degrees, 10 values of α , 2 options for `include_bias`) and the `shuffle_cv` object (20 splits), how many fits does `GridSearchCV` perform, and how long would that take? Check your count against `len(grid_search.cv_results_['params'])`.
- (b) How many fits did the randomized search use, and why did the chapter choose that number?
- (c) If you could afford only 60 fits in total, how would you split them between hyperparameter settings and cross-validation folds? What do you give up or gain when you trade samples between those two categories?

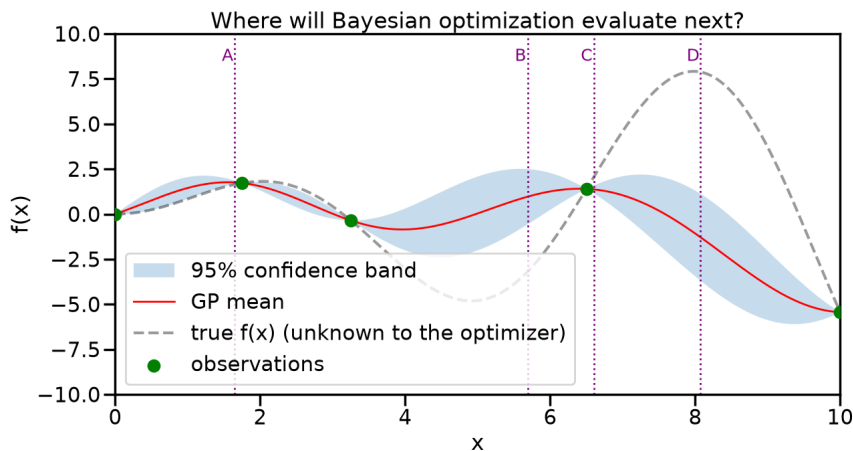
Exercise 4 (~6 min): Pick the sampling distribution

Randomized search needs a probability distribution for every hyperparameter. For each parameter below, propose a distribution from `scipy.stats` (or a list to choose from), give its support, and justify the choice in one sentence. Then implement one of them as an entry in a `param_distributions` dictionary and plot a histogram of 1000 draws to check that it looks the way you intended (the chapter did this for the lognormal).

- (a) The ridge penalty α , plausible anywhere between 10^{-8} and 10^3 .
- (b) The polynomial degree, an integer between 1 and 20.
- (c) The fraction of the training data used in each bootstrap subsample, anywhere in $(0, 1)$.
- (d) The number of hidden units in a neural network layer, between 8 and 1024.
- (e) The phase angle of a sinusoidal feature, anywhere in $(-\pi, \pi]$.

Which of these would be badly served by a plain uniform distribution, and what would go wrong?

Run the cell below to produce the figure for Exercise 5. It fits the chapter's Gaussian process to five observations of $f(x) = x \sin x$ and marks four candidate locations for the next evaluation.

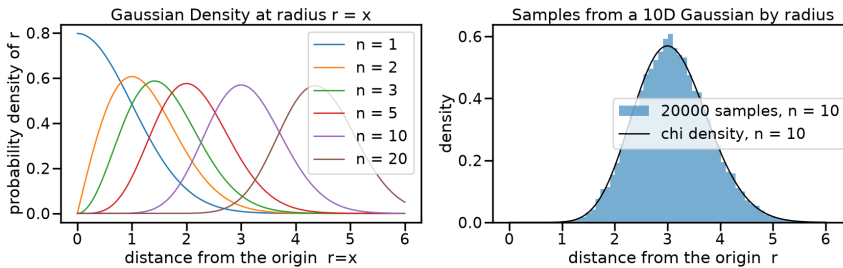


Exercise 5 (~5 min): Where does Bayesian optimization look next?

The figure shows the Gaussian process fit after five evaluations of an expensive function we want to *maximize*: the red line is the posterior mean, the shaded band is the 95 percent confidence region, and A to D are four candidate locations for the next evaluation. The chapter's acquisition function is the upper confidence bound, $UCB(x) = \mu(x) + \kappa \sigma(x)$ with $\kappa = 1.96$.

- (a) Which candidate of the four letters would UCB pick? For each of the other three, say in one sentence why it will not pick that point.
- (b) Which candidate would a purely greedy strategy ($\kappa = 0$) pick? Which one would a very exploratory strategy ($\kappa = 10$) pick?
- (c) One of the candidates would never be picked by any κ . Which one, and why?

The next exercise is about interpreting a figure rather than writing code. The cell below plots the probability density of the *distance from the origin* of a standard Gaussian in n dimensions. (This is the chi distribution $p_R(r) = \frac{2r^{n-1}}{(2\sigma^2)^{n/2} \Gamma(\frac{n}{2})} \exp(-\frac{r^2}{2\sigma^2}) = \frac{2r^{n-1}}{2^{n/2} \Gamma(\frac{n}{2})} \exp(-\frac{r^2}{2})$.) We also plot a histogram of that distance for random samples in $n = 10$ dimensions. Feel free to read the code if it helps you understand where the plots come from.



Exercise 6 (~5 min): A Gaussian is a donut

The density of a Gaussian is $p(\mathbf{x}) = \mathcal{N}(\mathbf{x} \mid \mu, \Sigma) = \frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \exp(-\frac{1}{2}(\mathbf{x} - \mu)^\top \Sigma^{-1}(\mathbf{x} - \mu))$. For a standard Gaussian this simplifies and is proportional to $p(\mathbf{x}) \propto e^{-\|\mathbf{x}\|^2/2}$. We see that the probability is largest at the origin in *every* dimension. Yet the figure says that in ten dimensions almost no sample lands anywhere near the origin.

- (a) Where is the peak of the radial density for $n = 1, 2, 3, 5, 10, 20$? Read the values off the plot and propose a formula in terms of n .
- (b) Roughly what fraction of the $n = 10$ samples fall inside the unit ball $r < 1$? And inside $r < 2$?
- (c) Resolve the apparent contradiction: how can the origin be the most probable *point* while the probability of being *near* the origin is almost zero? (Think about how much volume a thin shell at radius r has in n dimensions.)
- (d) Name one consequence for a method that relies on distances between points, and one for a method that draws random samples in a high-dimensional space.

Exercise 7 (~10 min): Nearest and farthest neighbours

The chapter's volume-ratio plot shows one face of the curse of dimensionality. Here is another, which you will build yourself.

Sample $N = 500$ points uniformly in the unit cube $[0, 1]^d$. For every point, compute the distance to its *nearest* other point and to its *farthest* other point, and form the ratio $d_{\min}/d_{\max}$. Average the ratio over the 500 points. Repeat for $d \in \{1, 2, 5, 10, 20, 50, 100, 500, 1000\}$ with a fixed random seed and plot the mean ratio against d on a logarithmic d -axis. (`scipy.spatial.distance.cdist` gives you all pairwise distances with one function, although remember to exclude each point's zero distance to itself.)

- (a) Describe the trend. What does the ratio approach as d grows, and what does that mean in words about the nearest and the farthest neighbour?
- (b) What does this do to a k -nearest-neighbour method in high dimensions?
- (c) What does it imply for grid search and random search when the number of hyperparameters grows?

3 Introduction to Gradient Descent

We will now review Linear Regression from the standpoint of Gradient Descent (instead of the normal equations), so as to build our intuition about how Gradient Descent works, and also introduce the concept of Stochastic Gradient Descent (SGD).

Let's first set up our notation for the problem:

$$y = w \cdot x + b + \epsilon$$

Or, if we consider $x = [1, x]$ then:

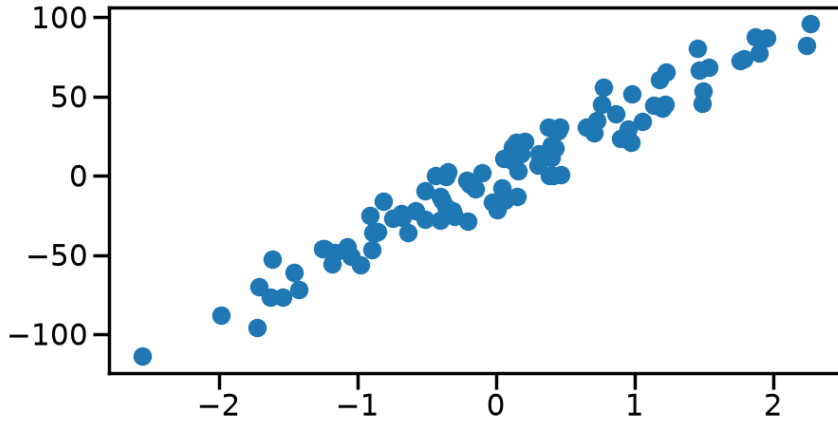
$$y = \mathbf{w}^T \cdot \mathbf{x} + \epsilon$$

Learning Objectives

By the end of this chapter, you should be able to:

1. Derive the gradient of the mean squared error for a linear model and write down the gradient descent update.
2. Relate the step size (learning rate) to convergence, oscillation and divergence.
3. Contrast batch gradient descent with stochastic and mini-batch gradient descent, including decaying step sizes and the Robbins-Monro conditions.
4. Map a hand-coded gradient descent loop onto a library implementation and its hyperparameters.
5. Explain, at a conceptual level, what momentum does, how batch size trades noise for cost, and why normalizing features makes gradient descent converge faster.

Coefficient: 42.38

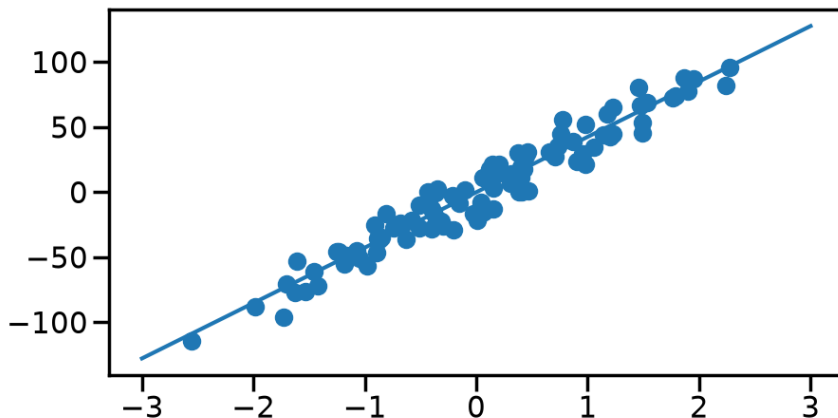


One thing to notice about this dataset before we go on: `make_regression` produces data centred on the origin, so the true line passes through zero and there is no intercept to learn. The chapter therefore fits a single parameter, the slope w , which keeps every plot one-dimensional. The intercept returns in the Normalization section at the end of the chapter, where having two parameters is exactly the point.

From your earlier statistics classes, you likely learned how to solve for the linear regression weights using the [Normal Equations](#):

$$\hat{w} = (X^T X)^{-1} X^T y$$

42.57166255883968



There are ways of solving the normal equations directly without needing to take the inverse (such as using the [Cholesky decomposition](#)), however today we are going to focus on a different kind of solver that has more broader applications: [Gradient Descent](#) and it's cousin [Stochastic Gradient Descent \(SGD\)](#).

We first need to start with some sort of Cost function that we wish to minimize. In general, we will consider costs of the form:

$$Loss = Error + \alpha \cdot Penalty$$

Specifically for Linear Models, we will talk about costs (which I'll call J) of the form:

$$J(w, X) = \frac{1}{N} \sum_{i=1}^N (\mathbf{y}_i - f(\mathbf{w}, \mathbf{x}_i))^2 + \alpha \cdot \Omega(\mathbf{w})$$

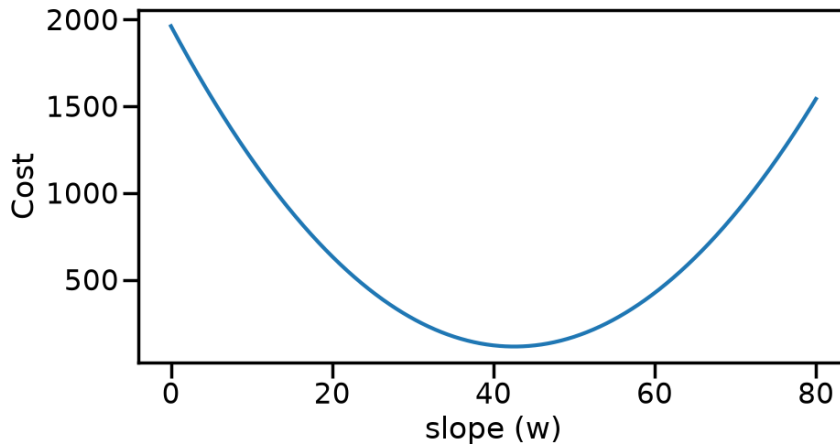
where for Linear Models $f(w, X) = \mathbf{w} \cdot \mathbf{X}$, so that our overall cost becomes:

$$J(w, X) = \frac{1}{N} \sum_{i=1}^N (\mathbf{y}_i - \mathbf{w} \cdot \mathbf{x}_i)^2 + \alpha \cdot \Omega(\mathbf{w})$$

We'll consider the no-penalty case ($\alpha = 0$), so that our loss is just:

$$J(w, X) = \frac{1}{N} \sum_{i=1}^N (\mathbf{y}_i - \mathbf{w} \cdot \mathbf{x}_i)^2$$

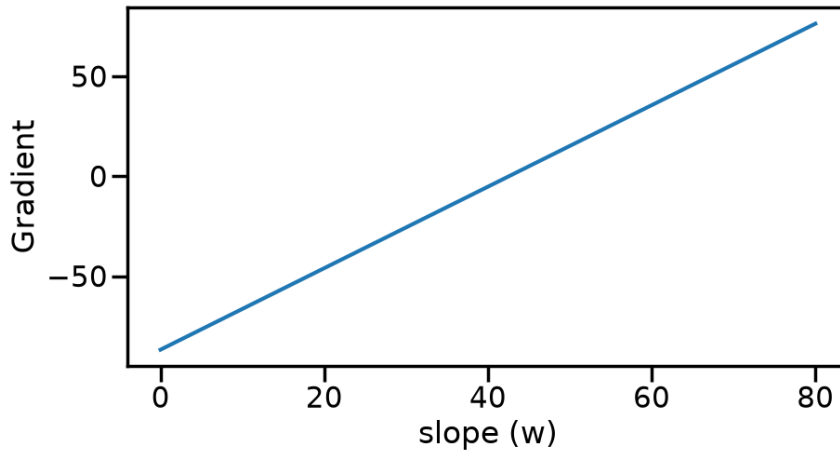
Let's plot this cost as a function of the line slope, just to get an idea of what it looks like:



While it might seem clear to us, visually, where the lowest cost is, actually finding this point automatically via a computer with minimal effort is another story. This is essentially what the field of Optimization tries to do. One simple (but powerful) method of optimization is Gradient Descent. It works by taking a (possibly random) starting point (e.g., $w=60$), and then computing the gradient of the function at that point. Since gradients will point upwards, and we want to minimize the cost, we will instead walk in the *negative* gradient direction, which should move us closer to the bottom. Let's see this on an example, by computing the gradient of our cost function above with respect to the slope (w):

$$\begin{aligned}
\frac{\partial J}{\partial w} &= \frac{\partial}{\partial w} \left(\frac{1}{N} \sum_{i=1}^N (\mathbf{y}_i - \mathbf{w} \cdot \mathbf{x}_i)^2 \right) \\
&= \frac{1}{N} \sum_{i=1}^N \frac{\partial}{\partial w} ((\mathbf{y}_i - \mathbf{w} \cdot \mathbf{x}_i)^2) \\
&= \frac{2}{N} \sum_{i=1}^N (\mathbf{y}_i - \mathbf{w} \cdot \mathbf{x}_i) \frac{\partial}{\partial w} (\mathbf{y}_i - \mathbf{w} \cdot \mathbf{x}_i) \\
&= -\frac{2}{N} \sum_{i=1}^N (\mathbf{y}_i - \mathbf{w} \cdot \mathbf{x}_i) \cdot \mathbf{x}_i
\end{aligned}$$

Let's plot this:



Once we have 1) a starting point, and 2) the gradient at a point, the idea with gradient descent is to take a small step (α) in the direction of the negative gradient:

$$w_{t+1} = w_t - \alpha \frac{\partial J}{\partial w}$$

Note: here we are just considering a single parameter (the slope, w), but this method extends to multiple parameters ($\mathbf{w}$), via the gradient operator:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} J$$

One more thing worth noting before we start walking downhill: the squared-error cost above, and every loss and penalty we met in the previous chapter, is *convex* in the weights. That means there is a single valley with one bottom, so gradient descent (and the stochastic variant we meet below) cannot get stuck somewhere else. This convenient property is what we lose once we move to neural networks in Part 2.

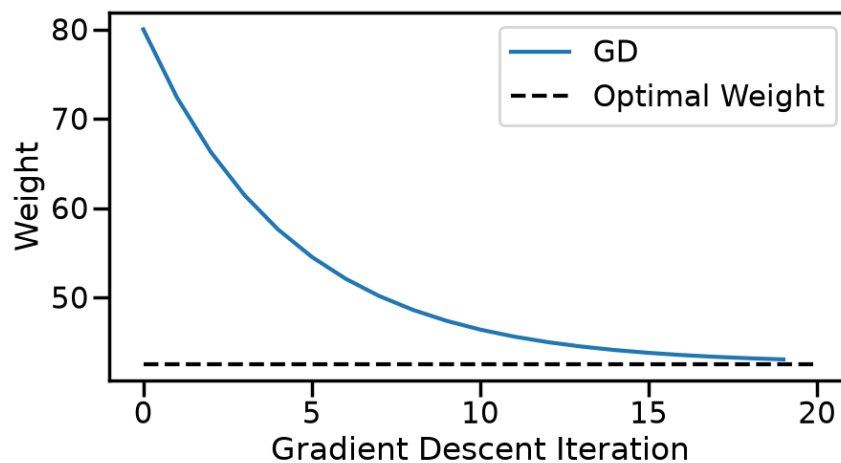
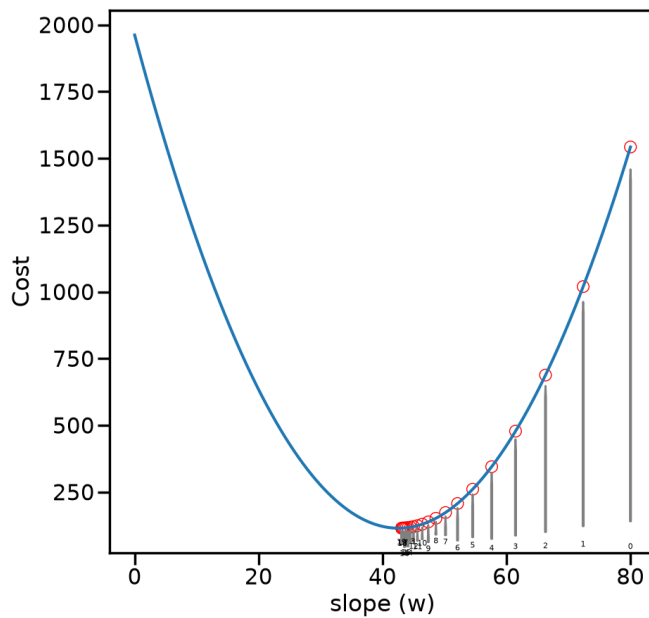
48.4855051874447

💡 Experiment: Starting Point and Step Size

Try modifying the initial guess `wg` and the step size `alpha` in the cell below and re-running the cells that follow. What do you observe?

- Does the starting point change where you end up, or only how long it takes?
- What happens for a very small step size? For a very large one?

Final weight from Gradient Descent is 43.06
Compared to 42.57 from the Normal Equations



3.1 How Large Can the Step Size Be?

In the Experiment above you probably found three kinds of behavior. With a small step size, gradient descent creeps towards the optimum. With a moderate one, it gets there quickly. And somewhere a bit below $\alpha = 1$ it suddenly stops working: the weights jump back and forth across the optimum and fly off to infinity. Can we predict where that happens, without trial and error?

We can, and all we need is a Taylor expansion. To do this, instead of tracking the weight w_t itself, we can track its *error*, that is, the distance $e_t = w_t - w^*$ from the optimum w^* . Let's look at how this error manifests near the optimum. We'll expand the gradient with a first-order Taylor expansion:

$$\frac{\partial J}{\partial w}(w) \approx \frac{\partial J}{\partial w}(w^*) + \frac{\partial^2 J}{\partial w^2}(w^*)(w - w^*) = \lambda e$$

where the first term vanishes because the gradient is zero at the optimum, and $\lambda = \partial^2 J / \partial w^2$ is the *curvature* of the cost. For our squared-error cost, the gradient $-\frac{2}{N} \sum_i (y_i - wx_i) x_i$ is a straight line in w , so this expansion is not an approximation at all: it is exact, with $\lambda = \frac{2}{N} \sum_i x_i^2 = 2 \overline{x^2}$ everywhere.

Now put the gradient back into the update rule $w_{t+1} = w_t - \alpha \partial J / \partial w$ and subtract w^* from both sides:

$$\begin{aligned} e_{t+1} &= e_t - \alpha \lambda e_t = (1 - \alpha \lambda) e_t, \\ e_t &= (1 - \alpha \lambda)^t e_0. \end{aligned}$$

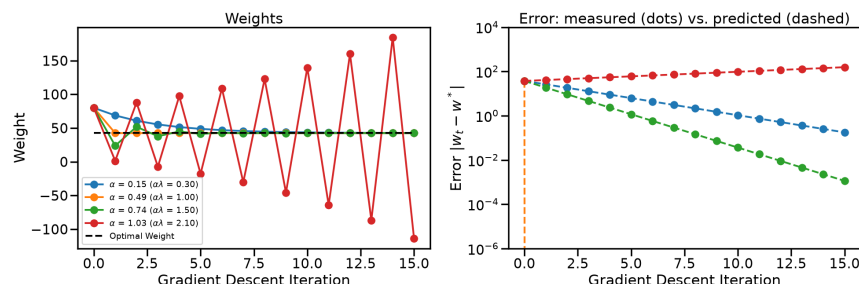
Every step multiplies the error by the same number, $1 - \alpha \lambda$, and from this we can diagnose the behavior we saw above:

- If $0 < \alpha < 1/\lambda$, the factor is between 0 and 1: the error shrinks and keeps its sign, so the weights slide smoothly down one side of the valley.
- If $\alpha = 1/\lambda$, the factor is exactly 0: gradient descent lands on the optimum in a single step. This is the *optimal* step size for this problem.
- If $1/\lambda < \alpha < 2/\lambda$, the factor is between -1 and 0: every step overshoots to the other side of the valley, but by less than before, so the weights zig-zag inwards.
- If $\alpha > 2/\lambda$, the factor is below -1 : every step overshoots by *more* than the error it started with, and the zig-zag grows without limit.

So gradient descent converges exactly when $0 < \alpha < 2/\lambda$. The more sharply curved the valley, the smaller the step you can afford: the gradient changes a lot over one step, so a long stride based on the gradient at the start overshoots badly. Let's compute these numbers for our data.

Curvature of the cost: $\lambda = 2.039$
 Optimal step size: $1/\lambda = 0.490$
 Largest stable step size: $2/\lambda = 0.981$

The largest stable step size is just below 1, which is where the Experiment above broke down. Let's run gradient descent at a few step sizes on either side of these two values, and compare the error we actually measure with the prediction $|e_t| = |1 - \alpha\lambda|^t |e_0|$.



On the right, the dots sit exactly on the dashed lines: for a quadratic cost the error really is multiplied by $1 - \alpha\lambda$ at every step. On a log scale that makes each run a straight line, whose slope is $\log |1 - \alpha\lambda|$. The optimal step size drops straight to zero error after one step (it falls off the bottom of the plot), and the step just above $2/\lambda$ grows. Notice also that the zig-zagging step size $1.5/\lambda$ (factor -0.5) converges *faster* than the small, smooth one (factor $+0.7$): what matters is the size of the factor, not its sign.

💡 Experiment: Predicting the Blow-Up

Change the list `step_sizes` in the cell above and re-run it.

- Go back to the step sizes you tried in the first Experiment. Does $2/\lambda$ explain where your runs started to fail?
- What happens at exactly $\alpha = 2/\lambda$? What does the factor $1 - \alpha\lambda$ tell you to expect, and is that what you see?
- Suppose we had recorded x in millimetres instead of centimetres, so every x_i is ten times larger. What would happen to λ , and to the largest step size you can use? (We will come back to this in the Normalization section at the end of the chapter.)

Our cost above was a perfect parabola, so the curvature is the same everywhere and the Taylor expansion is exact. Most costs are not parabolas, but if we are close enough to a minima many losses can behave like one with a curvature λ equal to the second derivative at its bottom. So $2/\lambda$ still tells you whether gradient descent can settle into that particular minimum, even though far from it the function may behave very differently. When there are many parameters, the second derivative becomes a matrix of second derivatives (the Hessian), and each direction in parameter space has its own curvature. The sharpest direction then limits how large a step

you can take, and the flattest direction decides how long you have to wait. We'll see this below in the Normalization section.

i Aside: Gradient Descent as a Dynamical System

What we just did has a name outside machine learning. The update rule $w_{t+1} = w_t - \alpha \nabla J(w_t)$ is a *discrete-time dynamical system*: a rule that takes the current state and produces the next one. Minima of the cost are its *fixed points* (states that the rule maps to themselves, since the gradient is zero there), and asking “does gradient descent converge?” is the same as asking “is this fixed point stable?”. Linearizing around the fixed point with a Taylor expansion, as we did above, is the standard way to answer that question in dynamics and control.

This view turns out to be surprisingly productive. If you shrink the step size towards zero, the iterates trace out the solution of an ordinary differential equation, $\dot{w}(t) = -\nabla J(w(t))$, called *gradient flow*, and properties of the optimizer can be read off properties of the ODE. The momentum method we will meet later in this chapter corresponds to a *second-order* ODE: a heavy ball rolling through the cost landscape with friction (Su, Boyd and Candès, 2016 work this out for a closely related accelerated method). Stochastic gradient descent adds noise to every step, so it behaves like a *stochastic* differential equation: with a constant step size it does not settle at a point, but keeps jittering in a cloud around the minimum whose width grows with the step size (Mandt, Hoffman and Blei, 2017). We'll see this manifest in Stochastic Gradient Descent in the next section.

3.2 Stochastic Gradient Descent

To perform Gradient Descent, we needed to sum the gradients across all data points (to compute ∇J). We may not want to do this, for many reasons: 1) we may not be able to iterate through all the data points easily (maybe the data are split across many computers or files), 2) we may not want to store the data at all after we are done with it (online learning), 3) maybe computing the gradient for one data point is difficult, and we would like to make forward progress on the model while computing gradients of other points, *etc.*

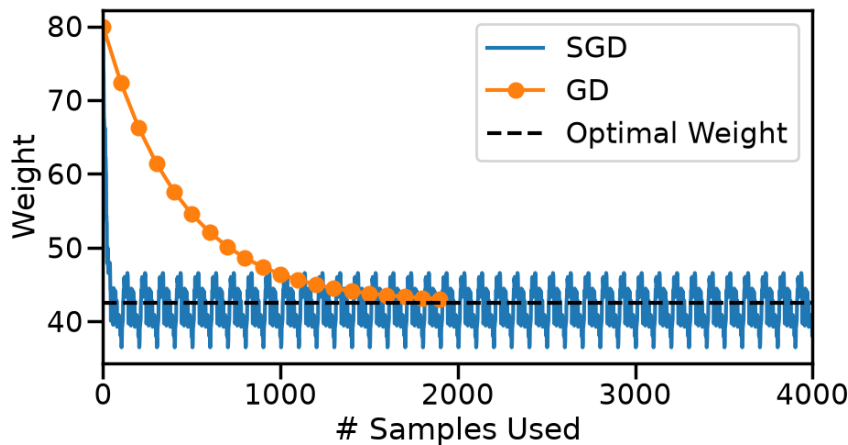
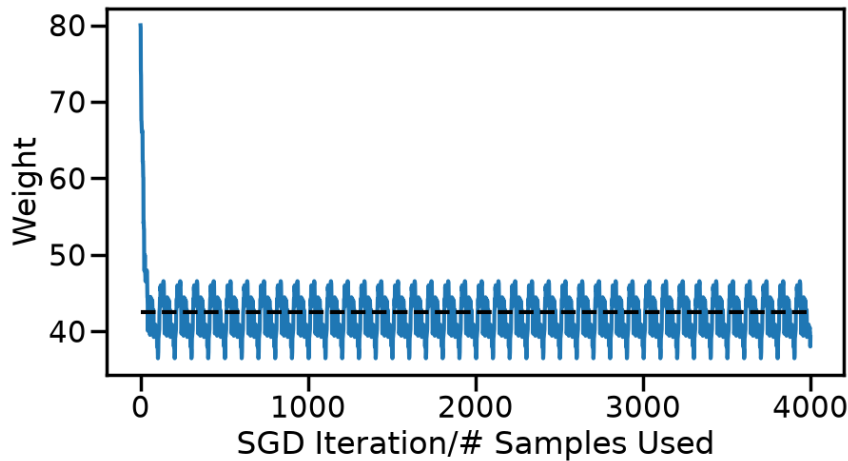
This is where *Stochastic* Gradient Descent comes in. Rather than Gradient Descent, which computes the gradient over all the data ($\mathbf{X}$):

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \frac{\partial J(\mathbf{w}, \mathbf{X})}{\partial \mathbf{w}}$$

Stochastic Gradient Descent computes a similar quantity, but only over one data point ($\mathbf{x}_i$) at a time (or a small batch of data if that is feasible):

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \frac{\partial J(\mathbf{w}, \mathbf{x}_i)}{\partial \mathbf{w}}$$

Initial weight: 80.0
Final Weight from SGD: 39.27
Compared to 42.57 (Normal Equations)



💡 Experiment: The Behaviour of SGD

Go back to the SGD cell above and change the step size and the number of passes.

- What do you notice about the behaviour of SGD compared with gradient descent?
- What happens when α is small versus large?
- If you keep doing more passes through the data, do you eventually converge? Why or why not?

What you are seeing is a result of [Stochastic Approximation](#) (trying to approximate a gradient of a function using noisy estimates of that gradient (where the noise here comes from evaluating the gradient using only one data point)).

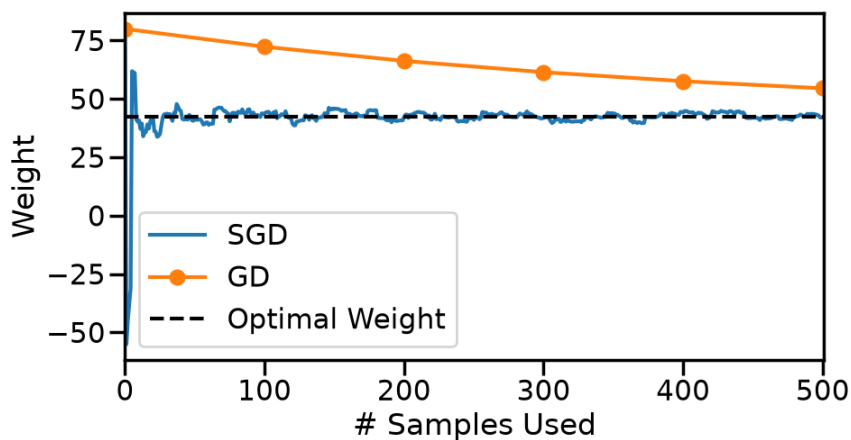
This behavior was studied by multiple people in the 1950s and 60s, with one key result coming from Herbert Robbins and Sutton Monro, in what is now called the [Robbins-Monro Algorithm](#). The central idea is rather than defining a *single* step size, we should let the step size decrease over time. Initially, we need to move the weights a lot, but as we get closer to the goal, they exert less influence so that we *settle* at some point. What they showed was that SGD would converge to the right estimator so long as the sequence satisfies the following properties:

$$\sum_{n=1}^{\infty} a_n = \infty$$

$$\sum_{n=1}^{\infty} a_n^2 < \infty$$

For sequences where $a_n > 0 \forall n > 0$, Robbins and Monro recommended the $a_n = a/n$, however this rate is based on some assumptions about smoothness and convexity which sometimes don't work well in practice. People generally use decay rates on the order of $O(1/\sqrt{(n)})$, however there are entire fields of researchers working on this “optimal step size” problem for SGD and there are many great alternative procedures out there if you know certain things about the function (e.g., can compute things like Hessians, etc.)

Initial weight: 80.0
 Final Weight from SGD: 42.09
 Compared to 42.57 (Normal Equations)



We see from the above that by reducing the step size over time, this can cause the SGD estimate to eventually converge to a stable answer.

3.3 Existing Implementations

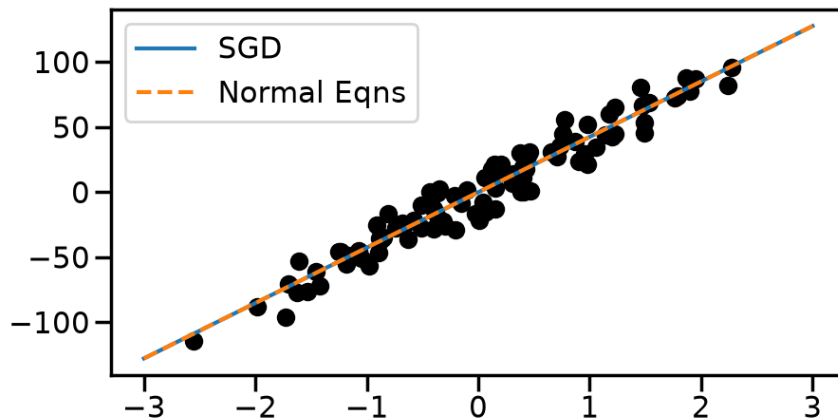
SGD is a fairly simple and popular technique for solving many problems where you can easily express the derivatives of those functions. For certain types of loss function (like the square error/L2 norm we discussed above), many folks have written optimized libraries for just that purpose, such as Scikit-Learn’s [SGD functions](#) including [SGDRegressor](#).

For reference, the SGD Regressor in ScikitLearn uses an update rule similar to:

$$\eta^{(t)} = \frac{\text{eta}_0}{t^{\text{power}_t}}$$

Final Weight from SKLearn SGD: 42.57

Compared to 42.57 (Normal Equations)



3.4 Advanced Techniques

What we presented above was just the “vanilla” version of SGD, and there is an entire sub-field of research focused on more performant variants of this idea. In this section, we’ll try to give you a flavor of some of the main variants or tricks people use to accelerate SGD methods, with the caveat that there are many more interesting rabbit holes you can go down here if you would like in more advanced optimization courses. Most of the popular implementations involve one or more of the following tricks:

1. Using “acceleration” procedures that leverage “momentum” of some type. We’ll illustrate this idea below, but you can read much more about this phenomenon (with some nice interactive visualizations) at: [“Why Momentum Really Works”](#)

2. “Batching” the SGD updates: that is, taking steps that consider $N > n > 1$ data points at a time (for example, averaging the gradients of, say, 5 data points before taking a step). This can help stabilize gradients and improve convergence.
3. “Normalizing” the features or the gradient updates, that is, re-scaling things so that every direction in parameter space is equally easy to walk in. This is commonly used in Neural Networks for things like [Batch Normalization \(Wikipedia\)](#) (or, for a more advanced introduction, you can read the NeurIPS paper [Understanding Batch Normalization](#)).

Let’s look at each of these on the problem we already have. From here on we work with plain NumPy arrays (the cell above already converted x and y back from matrices), so we first restate the cost and its gradient in NumPy array form for convenience.

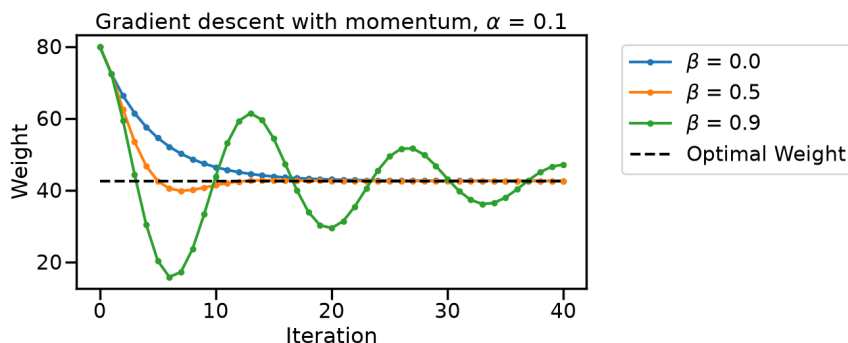
3.4.1 Momentum

Plain gradient descent forgets everything about the previous step. *Momentum* methods keep a running “velocity” that accumulates past gradients, in the same way a heavy ball rolling down a valley keeps moving in the direction it was already going:

$$v_{t+1} = \beta v_t + \nabla_w J(w_t)$$

$$w_{t+1} = w_t - \alpha v_{t+1}$$

With $\beta = 0$ this is exactly gradient descent. As β grows towards 1 the ball gets heavier: it picks up speed along directions where the gradient keeps pointing the same way, and it damps out directions where the gradient keeps flipping sign. Let’s watch how this change affects the behavior of SGD on our one-parameter problem.



💡 Experiment: How Heavy Should the Ball Be?

Change `momentum_step` and the values of `beta` in the cell above.

- With $\beta = 0.5$ the ball arrives faster than plain gradient descent. With $\beta = 0.9$ it overshoots and oscillates around the optimum before settling. What is it about

a heavy ball that produces this behaviour near the bottom of the valley? What about when the step size is really small? What does this behavior remind you of from prior classes you might have taken?

- Increase the step size until plain gradient descent ($\beta = 0$) starts to diverge (should be somewhere a bit before step size = ~ 1.0). What does adding momentum make that step size usable again?

i Aside: Stability with Momentum

Let's redo the stability analysis from *How Large Can the Step Size Be?*, now with momentum. Near the optimum the gradient is still λe_t , where $e_t = w_t - w^*$ is the error and $\lambda = 2\overline{x^2}$ the curvature. In terms of the error, the two momentum updates become

$$\begin{aligned}v_{t+1} &= \beta v_t + \lambda e_t, \\e_{t+1} &= e_t - \alpha v_{t+1}.\end{aligned}$$

The second line, one step earlier, says $\alpha v_t = e_{t-1} - e_t$, so we can substitute that in for the velocity term:

$$e_{t+1} = (1 + \beta - \alpha\lambda) e_t - \beta e_{t-1}.$$

Alternatively, we can express this as:

$$e_{t+1} = a e_t - \beta e_{t-1}.$$

Where $a = (1 + \beta - \alpha\lambda)$, and we can see that with $\beta = 0$ it collapses back to $e_{t+1} = (1 - \alpha\lambda) e_t$. To solve this recurrence equation, we can try the ansatz $e_t = r^t$, which turns the above into a quadratic equation for the per-step factor r :

$$r^{t+1} = a r^t - \beta r^{t-1}.$$

and dividing both sides by r^{t-1} gets us:

$$r^2 - a r + \beta = 0.$$

This quadratic has two roots, r_1 and r_2 , the error is a mix of r_1^t and r_2^t , and gradient descent converges exactly when *both* roots have $|r| < 1$. We can leverage two properties of quadratics to understand this behavior. First, via Vieta's formulas, the product of the roots is $r_1 r_2 = \beta$. Second, a real root can only leave the interval $(-1, 1)$ by passing through $r = +1$ or $r = -1$. Substituting those two values into the quadratic shows that this happens at $\alpha\lambda = 0$ and at $\alpha\lambda = 2(1 + \beta)$. So heavy-ball momentum converges exactly when

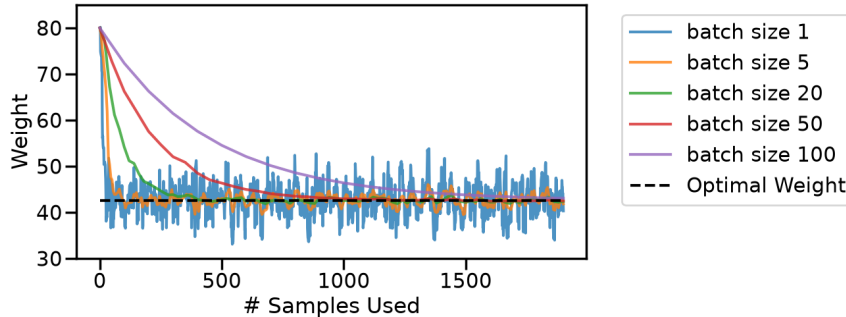
$$0 < \alpha < \frac{2(1 + \beta)}{\lambda}.$$

Two things follow, and both match what you saw in the Experiment above.

- **Momentum widens the stable range.** The largest usable step size grows by a factor $1 + \beta$. For our data, that is 0.98 for plain gradient descent, 1.47 with $\beta = 0.5$, and 1.86 with $\beta = 0.9$. That is why momentum rescued the step sizes at which plain gradient descent diverged. When a step overshoots the valley floor, the velocity still points forward while the new gradient points back, and the two partly cancel.
- **A high β can induce oscillation.** With $\beta > 0$, for most step sizes the two roots are a complex pair. Because their product is β , they then both have size exactly $|r| = \sqrt{\beta}$, no matter what α is. The error spirals in, oscillating around the optimum and shrinking by a factor $\sqrt{\beta}$ per step. With $\beta = 0.5$ that factor is 0.71, a little better than plain gradient descent's $1 - \alpha\lambda = 0.80$ at our step size of 0.1. With $\beta = 0.9$ it is 0.95, so the ball keeps oscillating for a long time before it comes to rest.

3.4.2 Mini-batching

Gradient descent used all N points per step whereas SGD used one. Everything in between is called a *mini-batch*: you average the gradient over k randomly chosen points, take a step, and move on to the next k points. This batch size k is a dial between the two extremes we have already seen.



💡 Experiment: Between SGD and Gradient Descent

The batch-size-100 curve is gradient descent itself (100 points per step), while batch size 1 is the SGD from earlier.

- Where does each curve sit between the two extremes, both in how noisy it is and in how many samples it needs to get close to the optimum?
- If each gradient evaluation cost the same regardless of batch size (as it roughly does

on a GPU), which batch size would you pick?

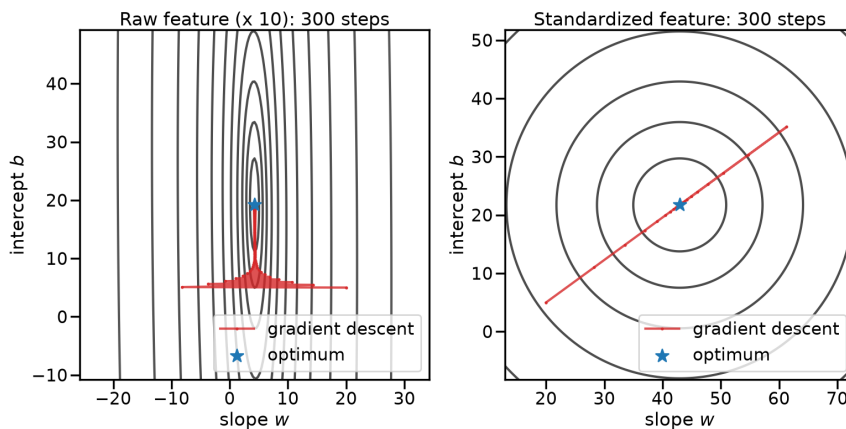
3.4.3 Normalization

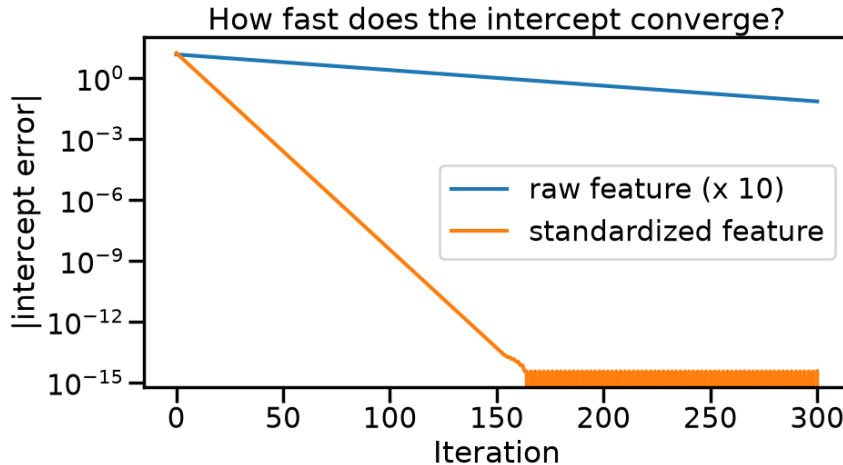
So far our problem had a single parameter, so there was only one direction to walk in. Real problems have many, and gradient descent gets into trouble when those directions have very different curvatures. The simplest place to see this is a line *with an intercept*, $y = wx + b$, when the feature x is recorded in units with a very different scale.

Suppose the same data had been logged in a unit that makes the numbers ten times larger (millimetres instead of centimetres, say), and that the target has an offset the model must learn (*i.e.*, the intercept). The cost is now a function of two parameters, $J(w, b)$, and its curvature in the w direction is $2x^2$ while in the b direction it is just 2. Scaling x by 10 makes that ratio a hundred; scaling by 1000 (metres to millimetres) makes it a million. Let's look at the loss landscape and at what gradient descent does in it.

In both cases below, we will run vanilla SGD, and the only difference will be that for each feature in the training data we will subtract off the mean and then divide it by the standard deviation: `x_standardized = (x_scaled - x_scaled.mean()) / x_scaled.std()`

```
raw feature (x 10): curvature in w = 203.88, in b = 2.00, ratio = 101.9, step size used = 0.0088
standardized feature: curvature in w = 2.00, in b = 2.00, ratio = 1.0, step size used = 0.9000
With the old step size of 0.1 on the raw feature, w after 20 steps is -2.5e+26 (diverged)
```





On the left the contours are a long narrow valley: the cost changes a hundred times faster along w than along b . The largest step size that is stable along w is tiny for b , so gradient descent shoots to the correct slope in a few steps and then crawls along the valley floor towards the correct intercept; after 300 steps it is still not there. On the right, after standardizing the feature, the contours are nearly circular, one step size suits both directions, and the same 300 steps land easily on the optimum.

This is why many machine learning pipelines standardize their inputs, and it is the seed of an idea we will meet again with neural networks: *batch normalization* re-standardizes the signals *inside* the network at every layer, for exactly the same reason, so that one learning rate works for all the weights at once. There are also more advanced SGD variants that attempt to learn or rescale features during optimization to account for this automatically.

3.5 Exercises

The exercises below reuse the data and functions from this chapter (`x_arr`, `y_arr`, `wn`, `mse`, `grad_mse`, `gd_weights`, and the momentum and mini-batch functions), so run the chapter's cells first.

Exercise 1 (~8 min): Common Gradients you will meet again

The chapter derived $\partial J / \partial w$ for the squared error of a slope-only linear model. The same few moves cover almost every loss and penalty you will see in this course. Work these out by hand (a sentence or two of justification each is enough):

- (a) The squared error of a line *with* an intercept, $J(a, b) = \frac{1}{N} \sum_i (y_i - ax_i - b)^2$: find $\partial J / \partial a$ and $\partial J / \partial b$. Consider how you would do this also in the vector form $\mathbf{w}^T \mathbf{x}$.

- (b) The L_2 penalty $\Omega(\mathbf{w}) = \alpha \|\mathbf{w}\|_2^2 = \alpha \sum_j w_j^2$: find $\nabla_{\mathbf{w}} \Omega$.
- (c) The L_1 penalty $\Omega(\mathbf{w}) = \alpha \|\mathbf{w}\|_1 = \alpha \sum_j |w_j|$: find $\nabla_{\mathbf{w}} \Omega$ wherever it exists, and say what goes wrong at $w_j = 0$.
- (d) The Huber loss of a single residual r , which is $\frac{1}{2}r^2$ for $|r| \leq \delta$ and $\delta|r| - \frac{1}{2}\delta^2$ otherwise: find dL/dr and sketch it.
- (e) The logistic loss of one example, $L = \log(1 + e^{-y\mathbf{w}^T\mathbf{x}})$ with $y = \pm 1$: find $\nabla_{\mathbf{w}} L$.
- (f) A bi-linear model of two column vectors u and v : $J(u, v) = u^T v$ where you want $\partial J / \partial u$ and $\partial J / \partial v$.
- (g) In matrix form, with $\mathbf{X}$ an $N \times d$ matrix: (1) $\nabla_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$, (2) $\nabla_{\mathbf{w}} \mathbf{w}^T \mathbf{A} \mathbf{w}$ for a square matrix $\mathbf{A}$, and (3) $\partial \|\mathbf{Y} - \mathbf{X}\mathbf{W}\|_F^2 / \partial \mathbf{W}$ for a matrix of weights $\mathbf{W}$ (the Frobenius norm is the square root of the sum of all squared entries). Note: the [Matrix Cookbook](#) is a good reference for common Matrix operations and derivatives.

You can check (a) numerically: with $b = 0$ your $\partial J / \partial a$ should agree with the chapter's `grad_mse`.

Exercise 2 (~10 min): Chain the pieces together

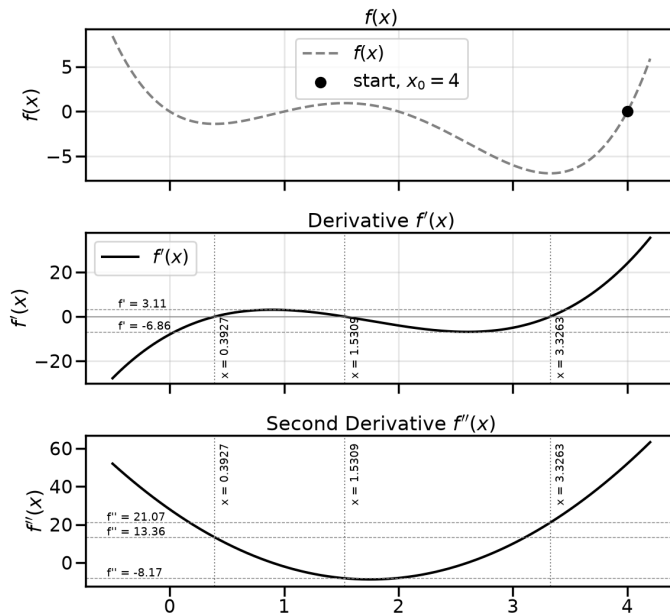
Real training objectives are sums of the pieces from Exercise 1, and their gradients are sums of the piece gradients. Consider the *elastic net* objective for a line with an intercept,

$$J(\mathbf{w}) = \frac{1}{N} \sum_i (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \alpha a^2 + \beta |a|, \quad \mathbf{x}_i = [x_i, 1], \quad \mathbf{w} = [a, b],$$

where the penalties act on the slope a only (it is conventional not to penalize the intercept b , since shifting all the y_i by a constant should only shift b). Use the chapter's data with $\alpha = 0.1$ and $\beta = 5$.

- (a) Write a function `grad_J(w)` that returns the two-component gradient by adding the pieces from Exercise 1.
- (b) Check it against a finite-difference approximation at a random $\mathbf{w}$ (evaluate J at $w_j \pm h$ with $h = 10^{-5}$), or use `scipy.optimize.check_grad`. Avoid $a = 0$ exactly; why?
- (c) Run gradient descent with your gradient from $\mathbf{w}_0 = [80, 0]$ and compare the slope a you reach with `wn`. Which way do the two penalties move it?
- (d) When the weights are large, which of the three terms dominates the gradient? When they are close to zero? What does that suggest about which penalty creates exactly-zero weights?

Run the cell below to produce the figure for Exercise 3. It plots a function with more than one valley, together with its derivative.



Exercise 3 (~8 min): Where does gradient descent end up?

Every cost in this chapter so far has been convex: one valley, one bottom, so gradient descent cannot end up anywhere else. The function in the figure above is not convex:

$$f(x) = x^4 - 7x^3 + 14x^2 - 8x, \quad f'(x) = 4x^3 - 21x^2 + 28x - 8.$$

We run gradient descent with a fixed step size, starting at $x_0 = 4.0$, where $f(4) = 0$ and $f'(4) = 24$. Consider four step sizes: A: $\alpha = 0.025$, B: $\alpha = 0.10$, C: $\alpha = 0.15$ and D: $\alpha = 0.2$.

- (a) Without running any code, predict for each step size where gradient descent ends up if we keep iterating: near which point on the x -axis, or does it diverge? Does it converge? Why? You should only need one gradient descent step by hand per step size, and then some reasoning with the plots and what you know about how gradient descent works.
- (b) Now check your predictions: run gradient descent for each step size (implement this however you like) and plot x_t against the iteration t . Where did your predictions hold, and where did the runs surprise you? Why?

Exercise 4 (~4 min): Does the schedule qualify?

Four colleagues propose step-size schedules for SGD, each with a constant $a > 0$: (i) $a_n = a/n$, (ii) $a_n = a/\sqrt{n}$, (iii) $a_n = a$, (iv) $a_n = a/n^2$.

- (a) For each, decide whether it satisfies the two Robbins-Monro conditions $\sum a_n = \infty$ and $\sum a_n^2 < \infty$, and name the condition that fails when one does.
- (b) Only one schedule passes both tests, yet the chapter says people generally use $1/\sqrt{n}$ in practice. What does each failing condition cost you in practice, and why might that be an acceptable price?

Exercise 5 (~6 min): Match the library

The chapter's decaying-step SGD (initial step 0.5, step $0.5/\sqrt{k}$ at update k , five shuffled passes, no penalty, no intercept) can be reproduced with `SGDRegressor`.

- (a) Which values of `learning_rate`, `eta0`, `power_t`, `max_iter`, `penalty`, `fit_intercept` and `tol` correspond to the hand-written loop? Fit it on X , y with `random_state=0` and compare `coef_` with `wn` and with the hand-coded result.
- (b) Now drop the arguments one at a time and go back to the defaults. Which default changes the answer most, and why? (Look up what `penalty`, `alpha`, `max_iter` and `tol` default to.)

Exercise 6 (~12 min): Tune the learning rate automatically (take-home)

Choosing α by hand is what we did in the Experiments. The previous chapter gave us tools to do it automatically. Define the black-box objective

$$f(\alpha) = \log_{10} J(w_{15}(\alpha)),$$

the (log) cost after 15 steps of gradient descent from $w_0 = 80$ with step size $\alpha \in (0.01, 1.5)$.

- (a) Plot $f(\alpha)$ on a fine grid first, so you know what the two methods below are looking for. Where is the minimum, and where does gradient descent start to diverge? Compare with what you saw in the first Experiment.
- (b) *Random search*: draw 15 values of α from a distribution you consider sensible for a step size and report the best one.
- (c) *Bayesian optimization*: starting from three evaluations, fit the Gaussian process from the previous chapter to f and pick the next α by minimizing the

- lower confidence bound $\mu(\alpha) - 1.96 \sigma(\alpha)$, for 12 iterations. Report the best α found and compare with random search at the same budget of 15 evaluations.
- (d) Extension: repeat with two hyperparameters, step size and momentum β , using `gradient_descent_momentum`. Plot the objective over the (α, β) plane. How does the best step size depend on β , and does that agree with what you saw in the momentum Experiment and the *Stability with Momentum* aside after it?

4 Review of Linear Unsupervised Learning

In this notebook, we will briefly review some of the key concepts of linear unsupervised learning, including Principal Component Analysis (PCA) and the effects of regularizations on linear decompositions, such as SparsePCA and Non-negative Matrix Factorization (NMF). We will use a set of airfoil geometries to demonstrate these effects visually. For those who need to refresh their memory on basic concepts around Matrices and how they affect data, you can see visual examples in [Appendix C](#)

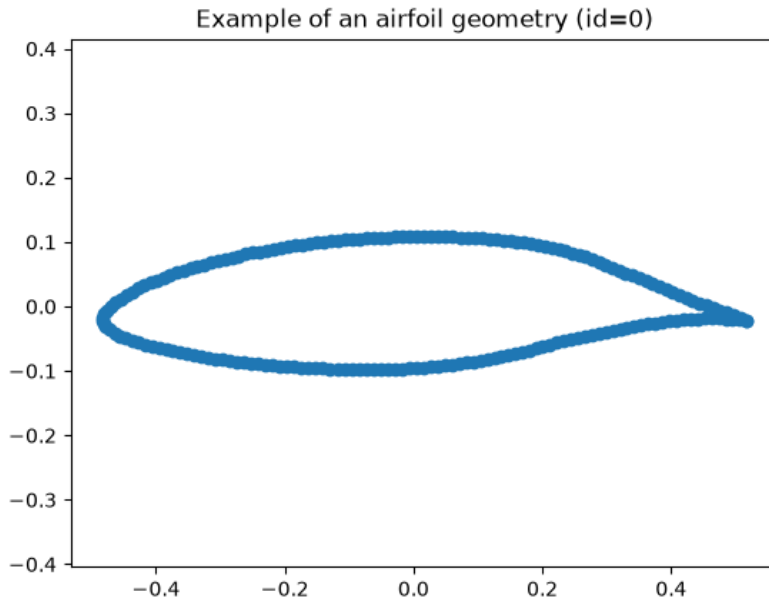
Learning Objectives

By the end of this chapter, you should be able to:

1. Explain PCA as a variance-maximizing orthogonal projection and read component and explained-variance plots.
2. Reconstruct data from k components and relate k to reconstruction error.
3. Contrast PCA, Sparse PCA, Dictionary Learning and NMF by their constraints, interpretability and cost.
4. Interpret decomposition components physically as airfoil shape modes.
5. Evaluate a decomposition on held-out data rather than on the training set.

First, let's load some airfoil geometry coordinates, take a look at the shape of the data matrix, and pick a random one to visualize:

The shape of the airfoil dataset is (1528, 192, 2)



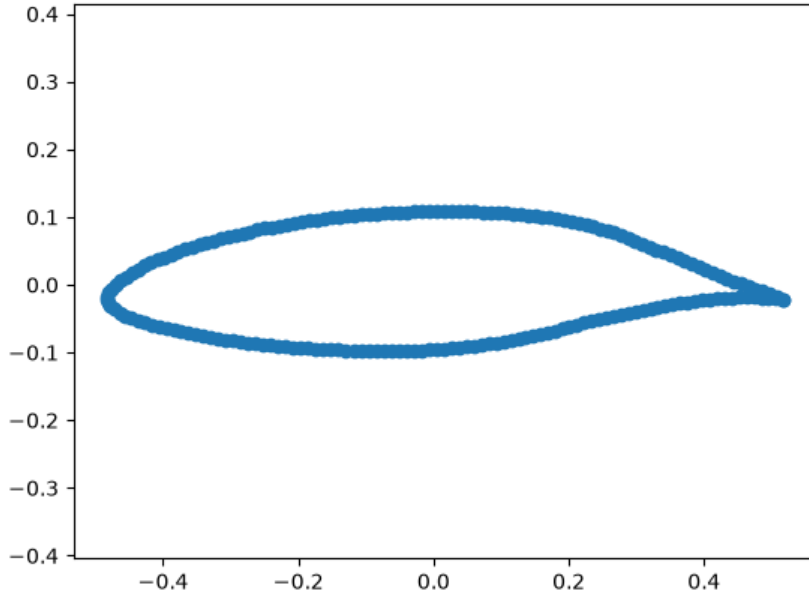
Great, as we can see there are 1528 airfoils, expressed as 192 surface coordinates each with an x and y value. We can turn this into a matrix compatible with a linear decomposition by flattening the last two dimensions, so that each airfoil is a row vector of length 384 (2x192). We keep the first 1222 airfoils for fitting and hold out the remaining 306, which no model in this chapter gets to see; the exercises at the end use them to check how well the fitted decompositions generalize.

```
X_train = make_matrix(X_train)
X_test = make_matrix(X_test)
print("Training Matrix Shape:",X_train.shape)
print("Held-out Matrix Shape:",X_test.shape)
data = X_train

# Transform back to original shape for plotting
# That is, make it back to N by 192 by 2
airfoils = make_airfoils(X_train)
print("Airfoils Data Shape:",airfoils.shape)
plt.figure()
plt.scatter(airfoils[airfoil_id,:,0],
            airfoils[airfoil_id,:,1])
plt.axis('equal')
plt.show()
```

Training Matrix Shape: (1222, 384)

Held-out Matrix Shape: (306, 384)
Airfoils Data Shape: (1222, 192, 2)



Now let's demonstrate how to use various dimension reduction algorithms on this example.

4.1 Principal Component Analysis

Mathematically, given centered data $X \in \mathbb{R}^{n \times d}$, PCA finds k orthonormal components that best reconstruct the data in least-squares sense. One convenient formulation is

$$\min_{W, Z} \|X - ZW\|_F^2 \quad \text{s.t. } Z = XW^T, \quad WW^T = I.$$

Equivalently PCA maximizes the projected variance:

$$\max_{W: WW^T = I} \text{tr}(W\Sigma W^T), \quad \Sigma = \frac{1}{n}X^TX.$$

Notes:

- There is no explicit L1/L2 penalty in vanilla PCA; the constraint $WW^T = I$ enforces orthonormality of components.
- The analytical solution is given by the top- k eigenvectors of the covariance (or the top- k left/right singular vectors from SVD).

```

from sklearn import decomposition

# We can set the maximum number of components that we want to truncate to
# Or can just leave it as None to get all components
n_components = 20
# svd_solver='full' computes the exact SVD; the default (randomized) solver gives slightly different numbers
estimator = decomposition.PCA(n_components=n_components, svd_solver='full')
Z_pca = estimator.fit_transform(data)
components_ = estimator.components_

```

We see that we now possess a matrix (i.e., linear operator) that goes from the target 20 components/dimensions back to the original 384 dimensions.

```
print(f"The shape of the components_ matrix (W) is {components_.shape}")
```

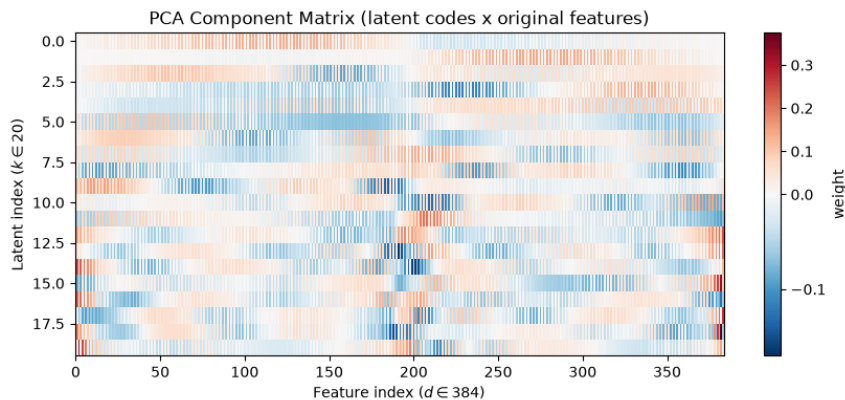
The shape of the components_ matrix (W) is (20, 384)

If we wanted to visualize how each of these components looks like in terms of the original airfoil coordinates, we can reshape each row of the components matrix back to the original airfoil shape:

```
components_.reshape((n_components, -1, 2)).shape
```

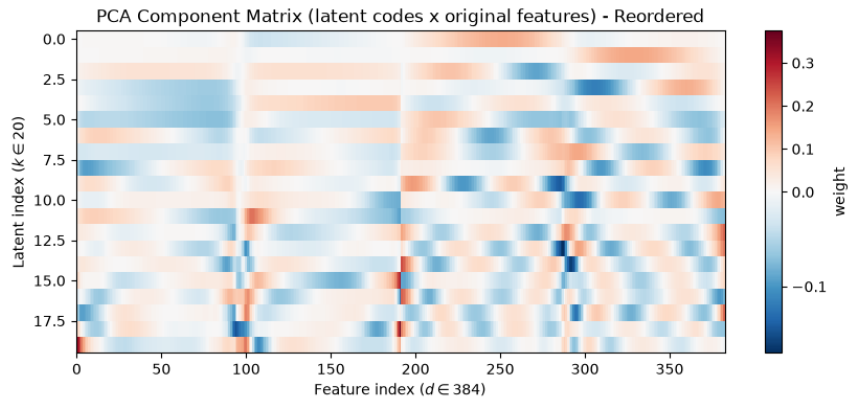
(20, 192, 2)

Let's go ahead and visualize the first all of the learned components:



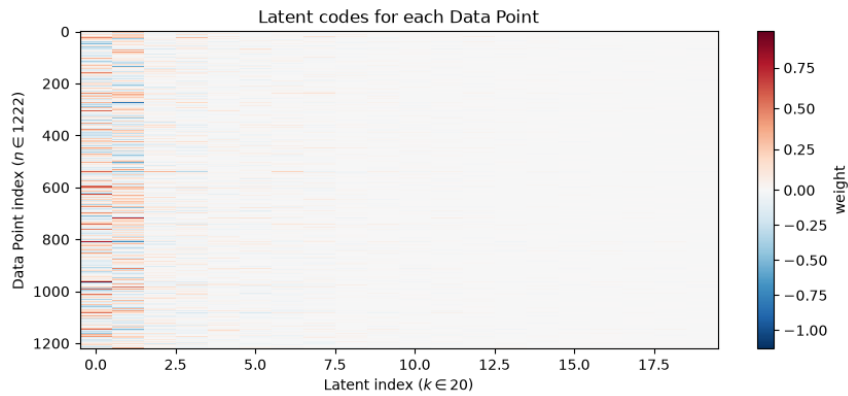
We notice a kind of alternating aliasing pattern in the components, but recall, this is because of how we reshaped the original data, which had rows of x and y coordinates interleaved. To make

this clearer, we can visually re-order the indices of the components so that all the x-coordinates come first (first 192 features), followed by all the y-coordinates (second 192 features):

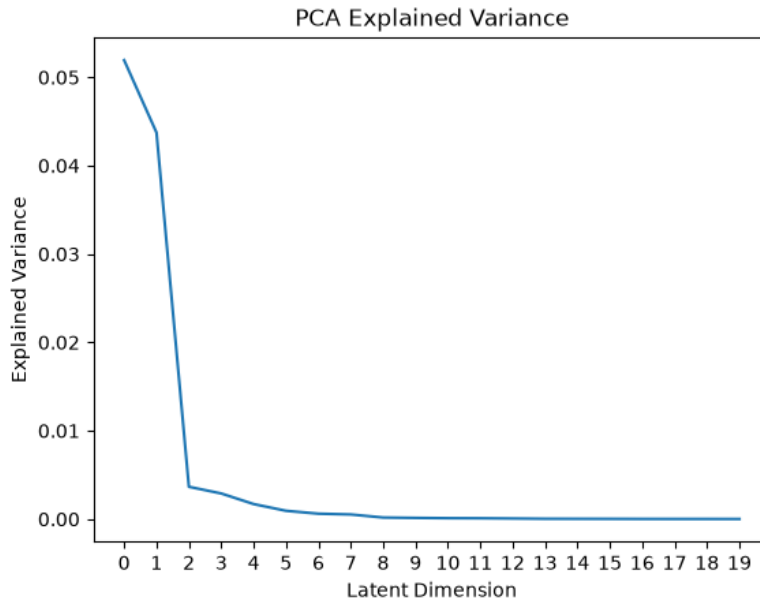


OK, this now looks a little clearer. What do you notice?

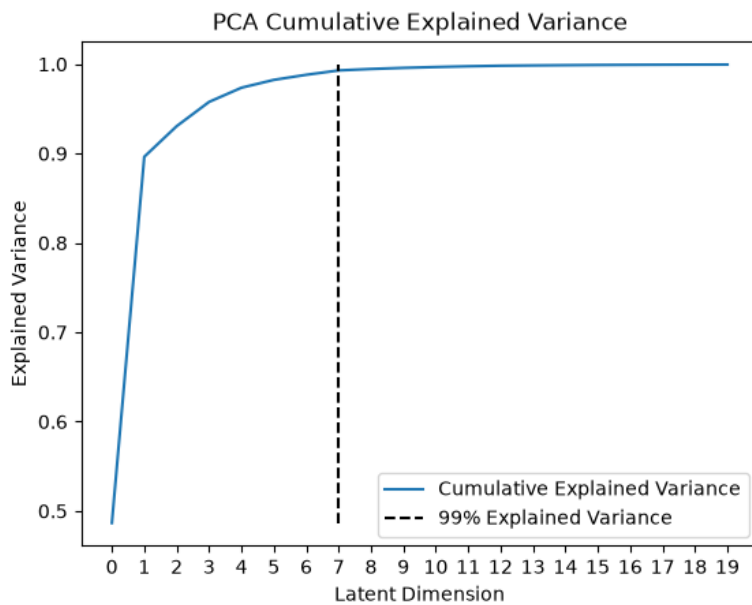
We can next visualize how each data point is mapped to all of the latent coordinates:



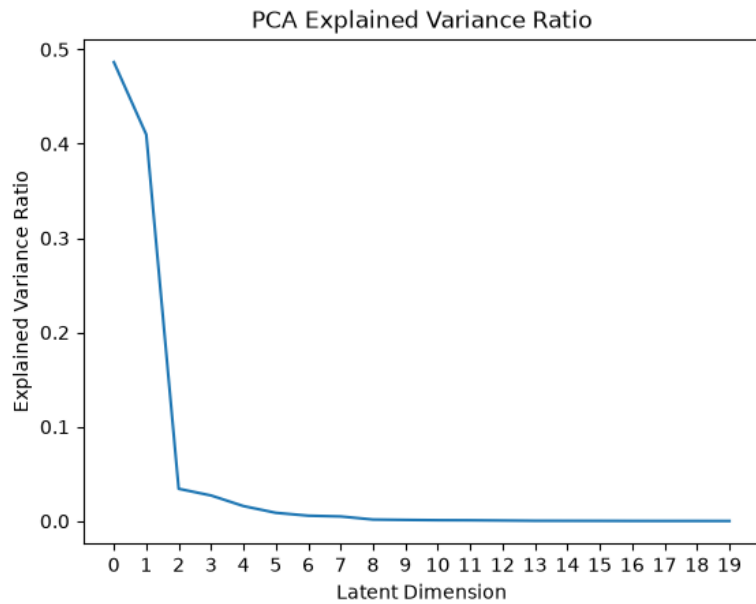
Why do you think the first few components are more important than the later ones? We can gain some intuition here by looking at the amount of variance explained by each component:



We can also visualize the cumulative explained variance to see how many components are needed to explain a certain amount of variance in the data. For example, we can plot a line at the number of dimensions we need to keep to explain 99% of the variance:

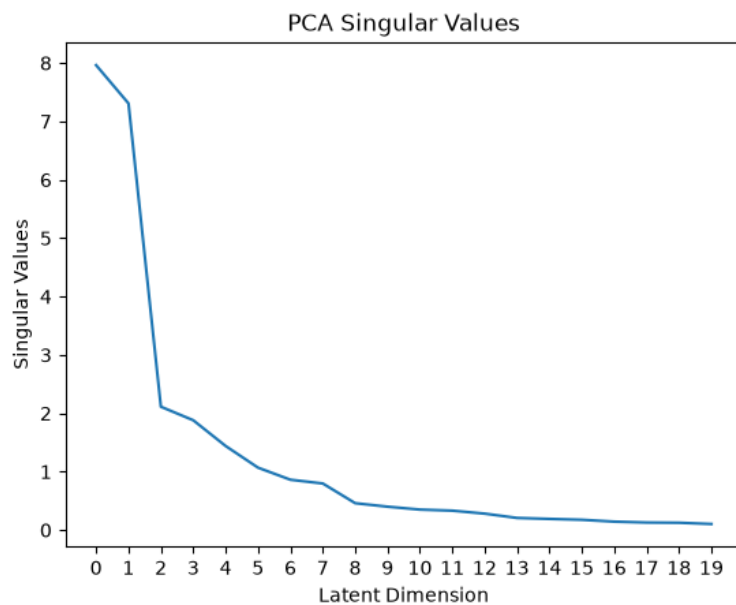


Or plot the explained variance ratio as a function of the number of components, which tells us how much each additional component contributes to the total explained variance:

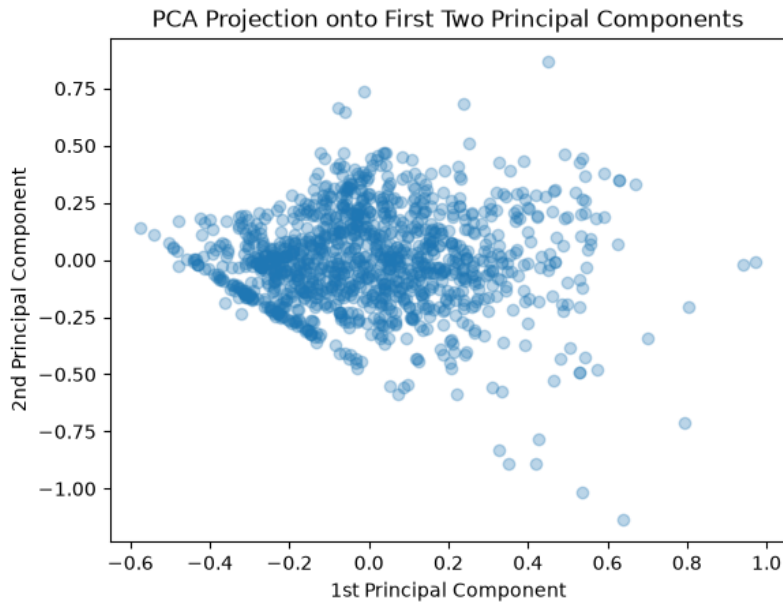


We can see that this approximately follows the singular values of the data matrix:

```
Singular Values: [7.96347431 7.30793278 2.11271967 1.880563    1.43978062 1.06857152
 0.86184436 0.79754077 0.46006406 0.40013427 0.3503857  0.33218949
 0.28107742 0.20708862 0.19063632 0.17582911 0.14392864 0.1288838
 0.12505431 0.10367465]
```



Moving beyond just the singular values, we can also now look at the projection of each airfoil into any of the latent dimensions. It is natural to explore the first two, since those explain the largest variance:



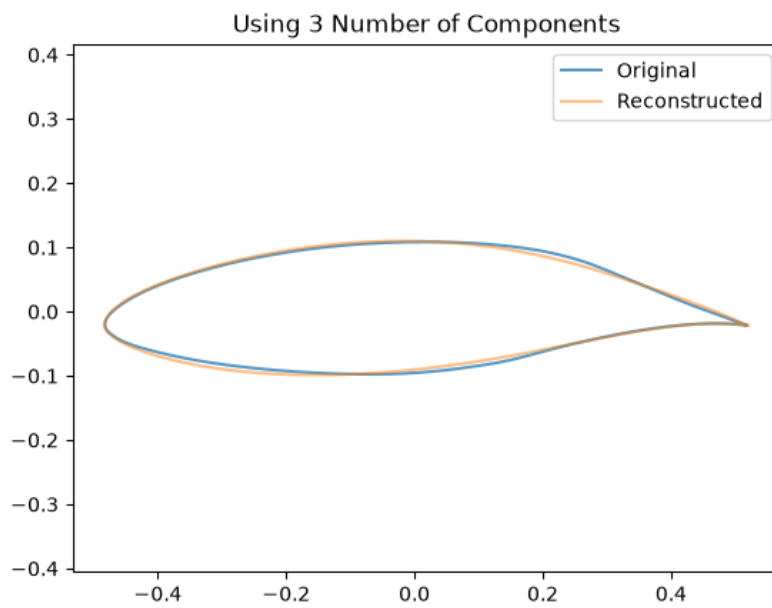
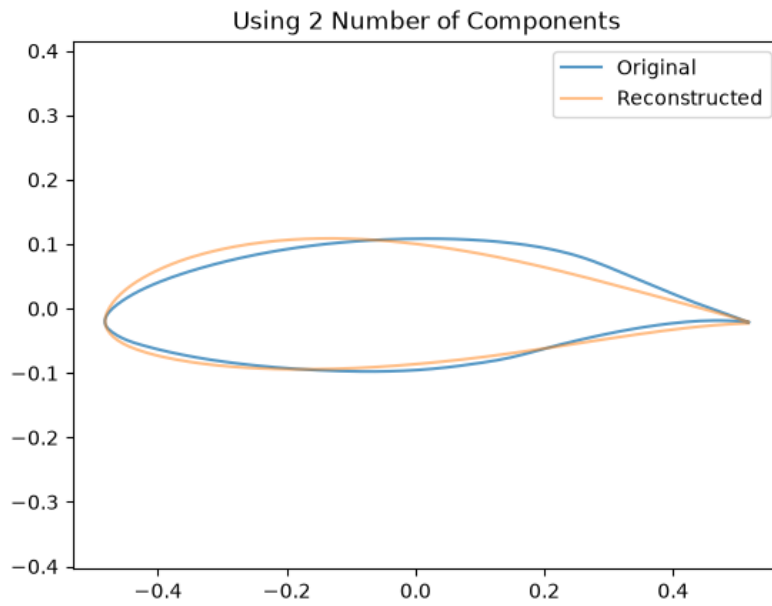
We can also pull out the “latent code/coordinates/vector” for an individual airfoil:

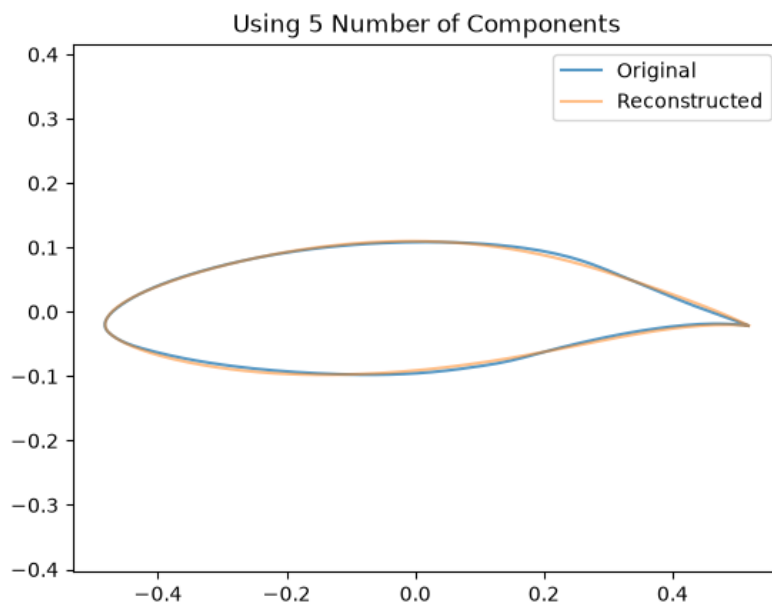
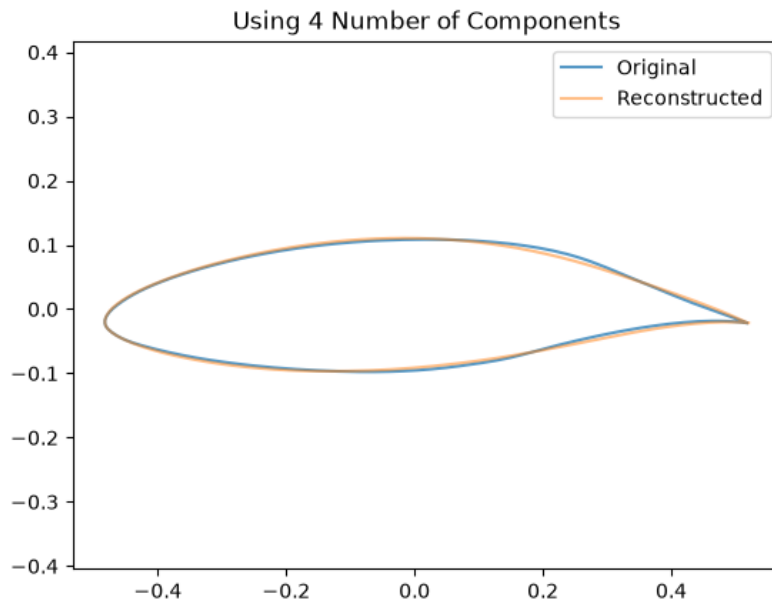
```
z[airfoil_id,:]
```

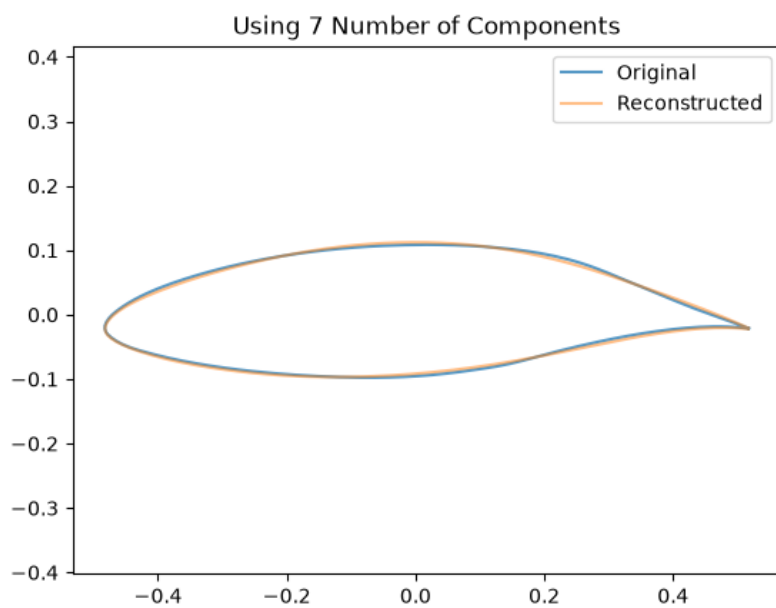
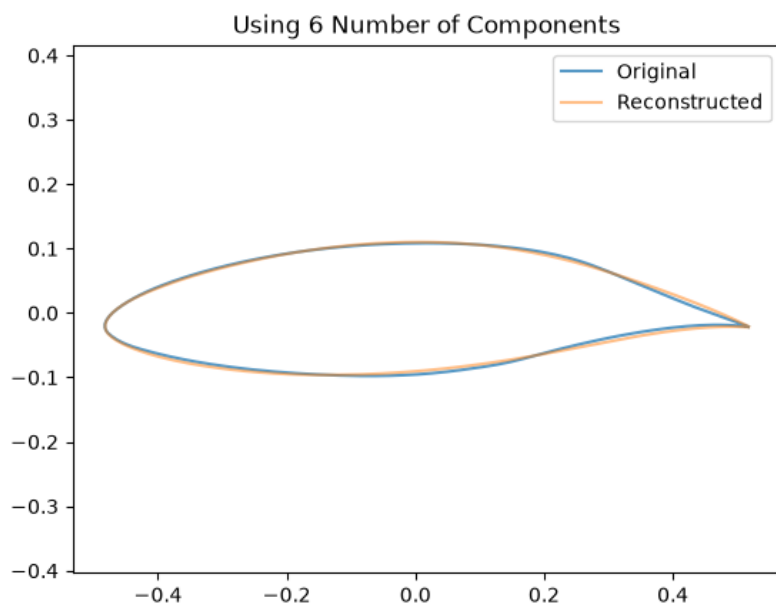
```
array([ 0.26565954, -0.30692116,  0.18797174, -0.0349173 , -0.0307379 ,
        0.02908489, -0.02403219,  0.02460046, -0.01230913, -0.0146706 ,
        0.0124125 ,  0.00509173, -0.01270168,  0.00192036, -0.00573131,
        0.00744313,  0.00506638,  0.00798497,  0.01136562,  0.00033123])
```

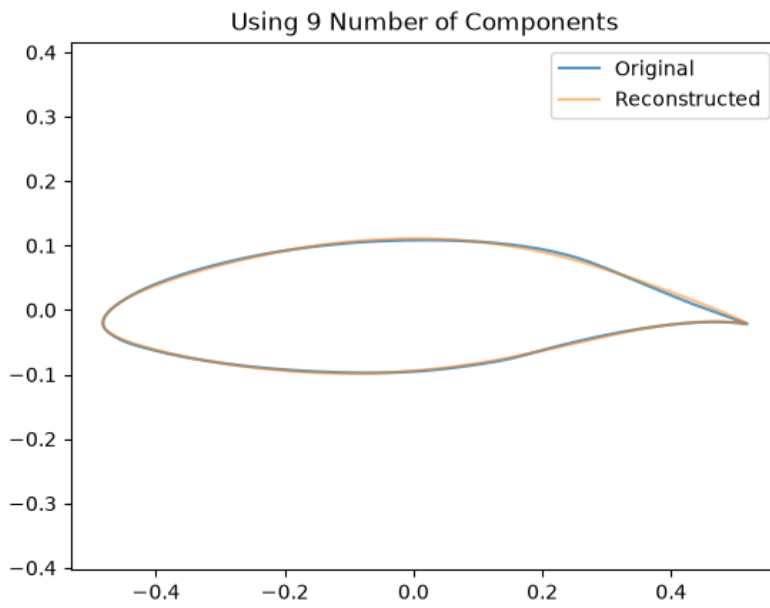
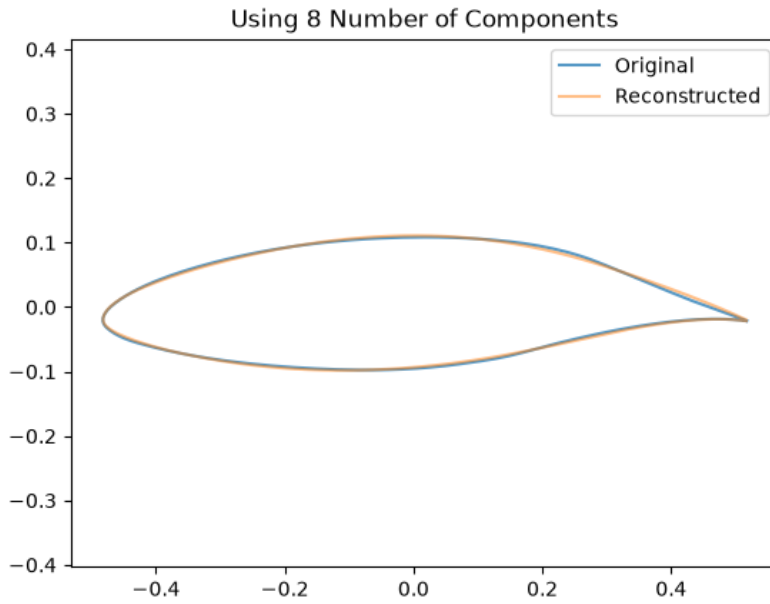
4.1.1 How does the # Components affect the Airfoil Reconstruction?

We can see visually the effect of including an increasing number of components by looking at how this affects the reconstruction of a single airfoil:









[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

4.2 Sparse PCA

Sparse PCA encourages components that have many zeros, which can make the learned components easier to interpret as localized shape features. Below we fit a SparsePCA model and visualize a small set of sparse components and their effect on reconstructing an example airfoil.

Mathematically, Sparse PCA augments a PCA-style reconstruction loss with an L1 penalty on the (unconstrained) components to encourage sparsity. A common optimization is:

$$\min_{Z, W} \|X - ZW\|_F^2 + \alpha \|W\|_1 \quad \text{s.t.} \quad \|Z_i\|_2^2 \leq 1 \quad \forall i,$$

where $W \in \mathbb{R}^{k \times d}$ holds the component vectors (rows), $Z \in \mathbb{R}^{n \times k}$ are the codes, and $\alpha > 0$ controls sparsity. Different implementations (e.g., the sklearn SparsePCA) solve related objectives (sometimes using a LASSO subproblem or elastic-net style updates).

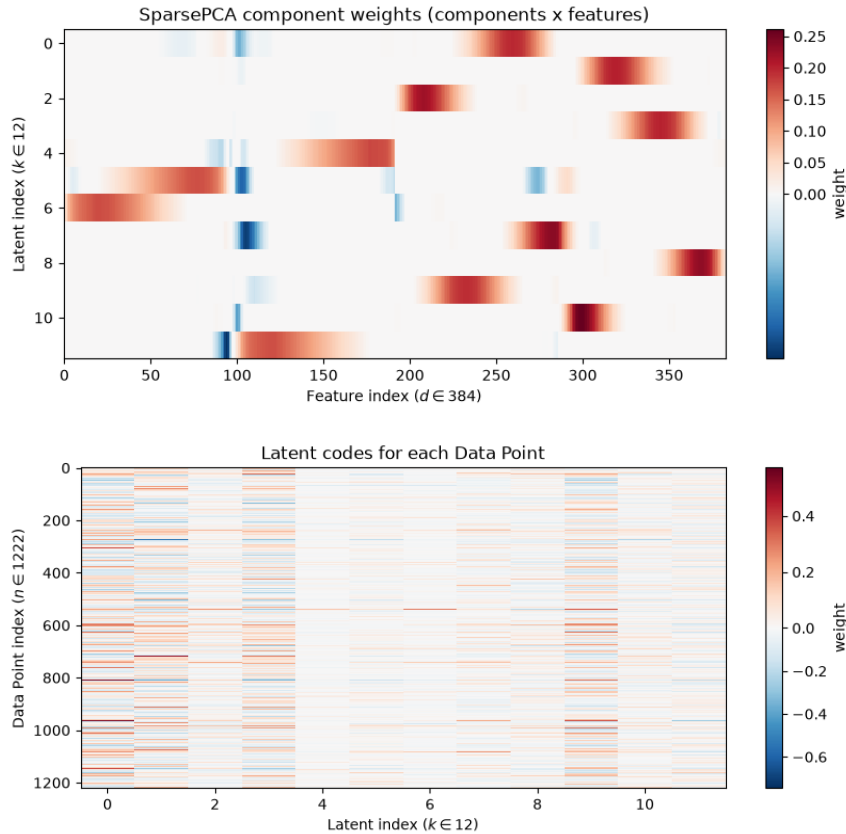
Notes:

- The L1 term on W encourages many component weights to be exactly zero, producing localized/part-like components.
- In practice one often also adds a small L2 (ridge) term to stabilize optimization (an elastic-net variant).

```
from sklearn.decomposition import SparsePCA

#### Try Changing both n_components and alpha ####
n_components = 12
# Warning: setting alpha too low will take the algorithm a long time to converge
alpha = 0.01
#####
spca = SparsePCA(n_components=n_components, alpha=alpha, random_state=42)
# fit on the flattened data matrix 'data' used above
spca.fit(data)
W_sp = spca.components_
Z_sp = spca.transform(data)
```

Now let's take a look at what the Sparse PCA model has learned and how it reconstructs the example airfoil differently than vanilla PCA:



[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

💡 Experiment: Comparison between PCA and SparsePCA

Compare both the component weight matrix and the reconstruction of a single airfoil when we used PCA and SparsePCA.

Consider the following questions:

- What are the main differences in the component weights between PCA and SparsePCA? How does the addition of the L1 penalty within SparsePCA manifest itself in the weight matrix? How does this manifest itself in the learned components, either individually or as we add them up during reconstruction?
- Also compare the latent codes Z between PCA and SparsePCA. How do they differ? What does this tell you about how the data is represented in the latent space? How about which components are most important?
- For SparsePCA, how does changing the value of the sparsity parameter α affect the learned components and reconstruction? What happens when we increase or

decrease α ? Look at both the weight matrix and the components.

- If you set α very low (e.g., 0.001) and increase the number of components, how does the result compare to vanilla PCA with the same number of components? What would you expect to happen, comparing the loss functions? Why are they different?
- Under what practical circumstances or for what types of problems might SparsePCA be preferred over PCA and vice versa?

4.3 Dictionary Learning

Dictionary learning finds a set of atoms (basis elements) and sparse codes that reconstruct the data. This is useful when localized, part-based representations are desirable. Essentially, unlike Sparse PCA (which adds an L1 penalty to the components), dictionary learning adds an L1 penalty to the codes. Below we fit a Dictionary Learning model and visualize a small set of learned atoms and their effect on reconstructing an example airfoil. A word on cost: the scikit-learn implementation is very slow on this data and running it to full convergence it needs about 400 alternating updates and 4 to 8 minutes. So we'll cap the solver at 30 iterations with a looser tolerance so that it finishes in about half a minute.

Mathematically, Dictionary learning (sparse coding) models X as the product of a dictionary $W \in \mathbb{R}^{k \times d}$ (atoms) and sparse codes $Z \in \mathbb{R}^{n \times k}$:

$$\min_{Z, W} \|X - ZW\|_F^2 + \alpha \|Z\|_1 \quad \text{s.t.} \quad \|W_j\|_2 \leq 1 \quad \forall j.$$

Here Z contains the sparse coefficients for each example and $\alpha > 0$ controls the sparsity of the codes. The constraint on W prevents trivial scaling to reduce the penalty term (i.e., just pulling weight into W and shrinking Z). Intuitively, each data point is reconstructed as a *sparse* linear combination of dictionary atoms, essentially selecting a few “parts” (where the parts are elements of W) to compose the whole as a weighted sum.

Notes:

- The L1 penalty acts on the coefficients (Z) rather than W to encourage sparse usage of dictionary elements.
- Many algorithms alternate between solving for Z (a LASSO-type problem) and updating W (a constrained least-squares step).

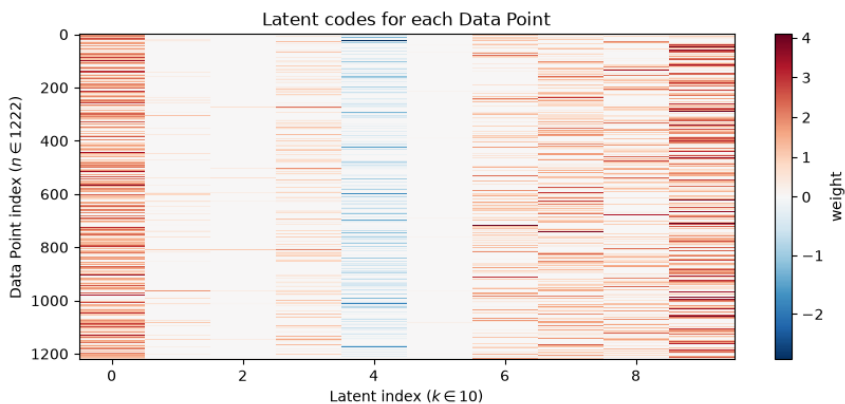
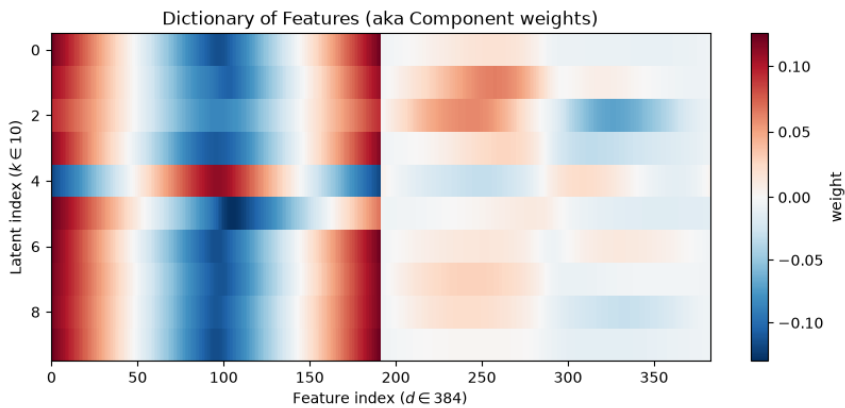
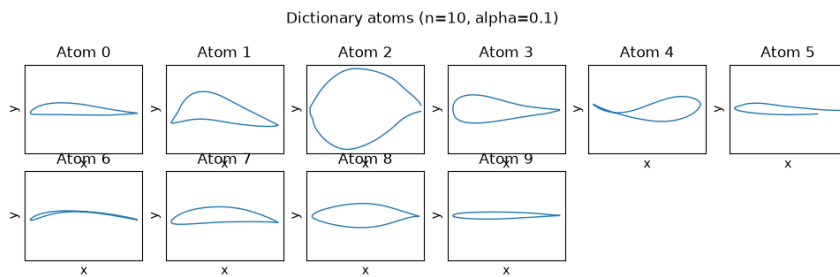
```
# The sklearn implementation is slow on this data: run to full convergence it takes ~400 iterations
# and 4 to 8 minutes. We cap it at 30 iterations with a looser tolerance (about half a minute).
from sklearn.decomposition import DictionaryLearning
# Alternatively, you can use the MiniBatchDictionaryLearning which is faster, but less accurate or stable:
# from sklearn.decomposition import MiniBatchDictionaryLearning as DictionaryLearning
```

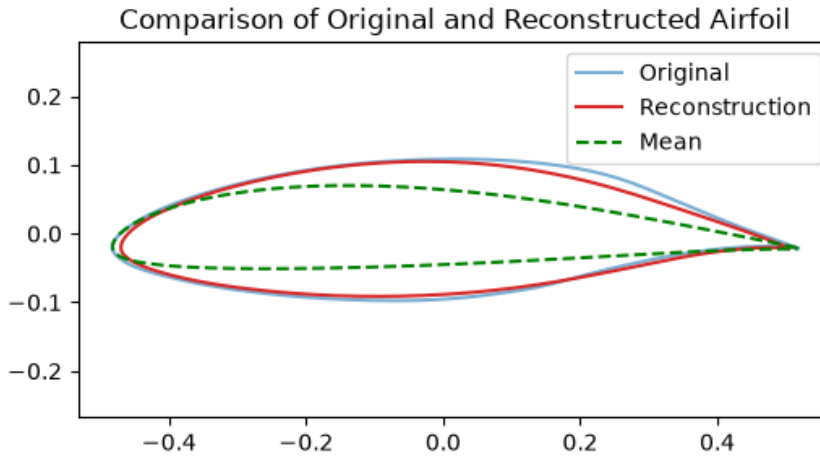
```

### Try changing n_components and alpha #####
n_components = 10
alpha = 0.1
#####

DL = DictionaryLearning(n_components=n_components, alpha=alpha, random_state=42, max_iter=30, tol=1e-4)
Z_dl = DL.fit_transform(data)
W_dl = DL.components_

```





[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

Average fraction of near-zero coefficients: 0.571

💡 Experiment: Comparison between Dictionary Learning (Sparse Coding) and SparsePCA

Compare both the component weight matrix and the reconstruction of a single airfoil when we used Sparse PCA and Dictionary Learning (Sparse Coding).

Consider the following questions:

- Both algorithms use the same sparsity penalty (L1), but on different matrices. How does this difference manifest itself in the learned components and reconstructions? What happens to the component weights versus the latent codes?
- For Dictionary Learning, how does changing the value of the sparsity parameter α affect the learned components and reconstruction? What happens when we increase or decrease α ? Look at both the weight matrix and the components.
- Under what practical circumstances or for what types of problems might Dictionary Learning (Sparse Coding) be preferred over PCA and Sparse PCA, and vice versa?

4.4 Non-Negative Matrix Factorization (NMF)

NMF constrains both the basis elements and coefficients to be non-negative. This often yields parts-based, additive representations which can be intuitive for shape components under certain circumstances.

Mathematically, Non-negative matrix factorization approximates X (with $X > 0$ after shift) as the product of non-negative factors $W \in \mathbb{R}_+^{n \times k}$ and $H \in \mathbb{R}_+^{k \times d}$ by minimizing reconstruction error under non-negativity constraints:

$$\min_{W \geq 0, H \geq 0} \|X - ZW\|_F^2 + \beta_Z \|Z\|_1 + \beta_W \|W\|_1 + \alpha_Z \|Z\|_F^2 + \alpha_W \|W\|_F^2,$$

where optional L1 penalties ($\beta_Z, \beta_W \geq 0$) encourage sparse parts or sparse activations, and L2 penalties ($\alpha_Z, \alpha_W \geq 0$) encourage stability or shrinkage. The essential constraint is $W, Z \geq 0$ which induces additive, parts-based representations.

Notes:

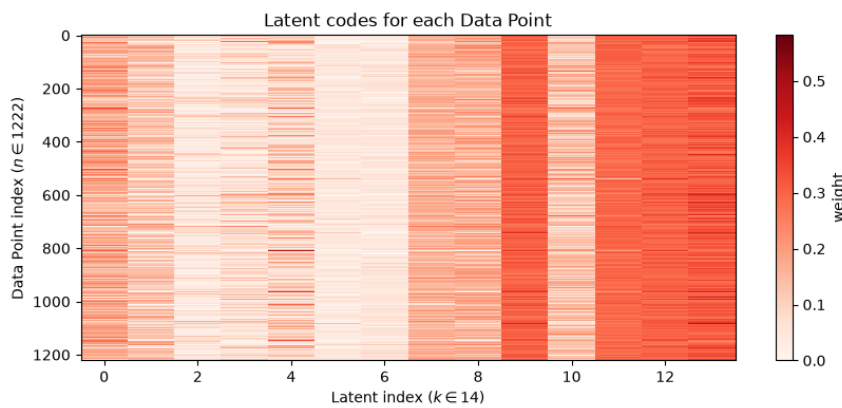
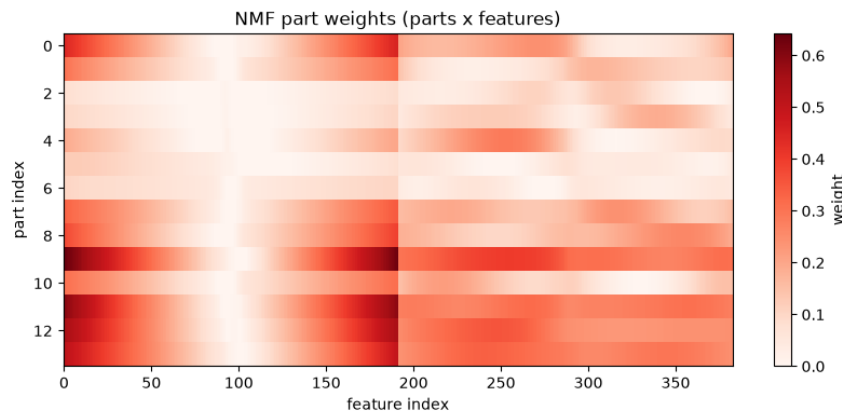
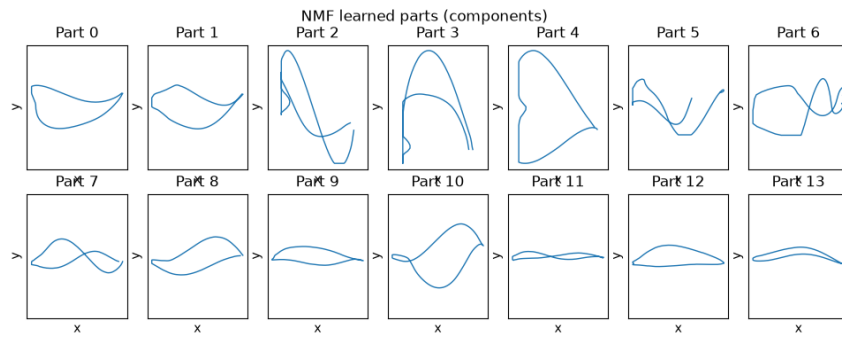
- In this notebook we shift data by the minimum to ensure $X \geq 0$ before fitting and then undo the shift when plotting reconstructions.
- Typical solvers use multiplicative updates or alternating non-negative least squares; regularization (L1 or L2) can be added to encourage sparsity or stability.

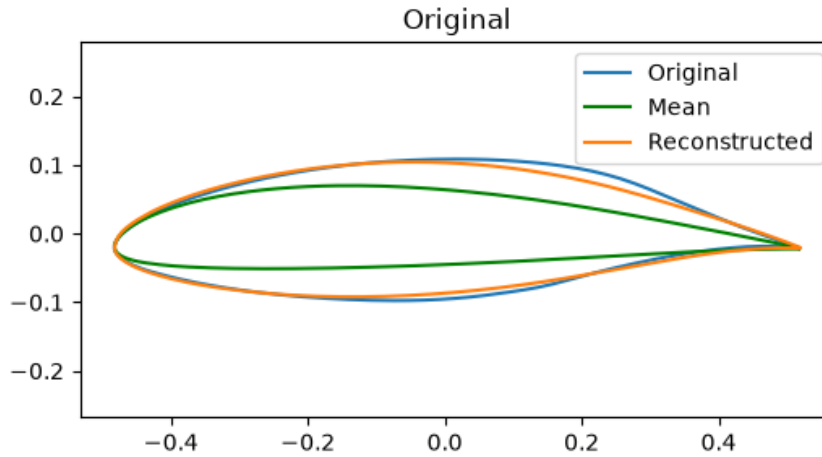
```
from sklearn.decomposition import NMF

#### Try changing n_components #####
n_components = 14
# This is the L2 penalty on the components
alpha_W = 0.001
# This is the L1 ratio on the components (between 0 and 1)
# Set to 0 for pure L2, 1 for pure L1, and in between for a mix
l1_ratio = 0.001
#####

# NMF needs non-negative data. We'll shift by the min and remember the offset.
shift = data.min()
Xpos = data - shift + 1e-6

nmf = NMF(n_components=n_components, init='nndsvda', random_state=42, max_iter=5000,
          alpha_W = alpha_W, l1_ratio=l1_ratio)
Z_nmf = nmf.fit_transform(Xpos)
W_nmf = nmf.components_
```





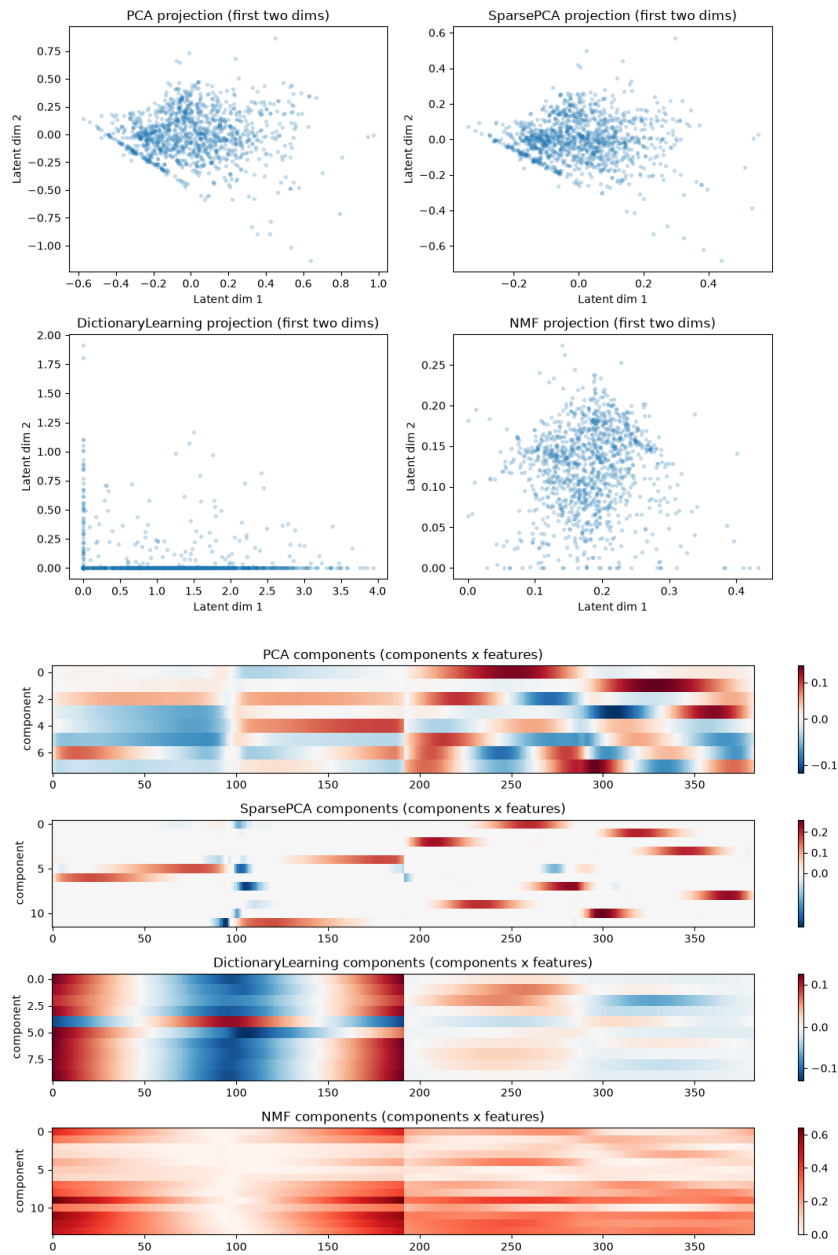
[Interactive output: available in the online version of this chapter.]

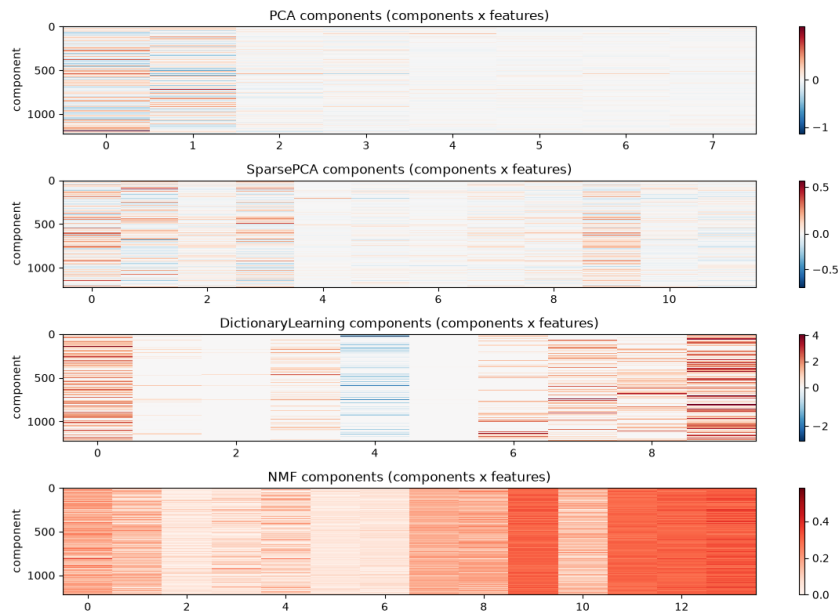
[Interactive output: available in the online version of this chapter.]

Explained variance (approx): 0.994

4.5 Model Comparison

Finally, let's compare the different models side-by-side. We will visualize the two learned projections of the data into the latent space, as well as the component reconstruction matrix that translates those components back into the original feature space, and lastly the mapping of all data points to all latent coordinates. This is similar to plots you have seen before, but it is nice to see them all together for comparison.





4.6 Exercises

The exercises reuse the airfoil data and the fitted models from this chapter, so run the chapter's cells first. `estimator` is the 20-component PCA; `spca`, `DL` and `nmf` are the Sparse PCA, Dictionary Learning and NMF fits; `X_test` holds the 306 held-out airfoils. Rough times are in the titles, and the last exercise is a longer take-home task.

The cell below defines three helpers: `reconstruct_pca` maps airfoils to their first k PCA latent coordinates and back, `rms_error` gives the root-mean-square (RMS) coordinate error of each airfoil, and `plot_airfoil` draws one airfoil.

Exercise 1 (~5 min): Which method has which property?

The chapter fitted four decompositions $X \approx ZW$ to the 1222 training airfoils: PCA (`components_`, `Z_pca`), Sparse PCA (`w_sp`, `Z_sp`), Dictionary Learning (`w_dl`, `Z_dl`) and Non-negative Matrix Factorization, NMF (`w_nmf`, `Z_nmf`).

Property	PCA	Sparse PCA	Dictionary Learning	NMF
The components (rows of W) are mutually orthogonal	?	?	?	?
Many entries of W are exactly zero	?	?	?	?
Many entries of Z are exactly zero	?	?	?	?
No entry of W or Z is negative	?	?	?	?
Asking for fewer components leaves the first component unchanged	?	?	?	?

- (a) Replace each ? with yes or no, using only the objectives given in the chapter. For each yes, name the constraint or penalty that causes it and explain why.
- (b) Check the first four rows with the fitted matrices, for example with WW^T and the fraction of entries that are exactly zero. Which predictions were wrong, and why?
- (c) Check the last row for PCA and NMF: refit each with 5 components and compare the new first component with the old one. (Refitting Sparse PCA takes about two minutes, so reason about it and Dictionary Learning from their objectives.) Why does PCA behave differently?

Exercise 2 (~5 min): PCA by hand

The chapter said that the principal components are the top right singular vectors of the centered data matrix. Check this yourself.

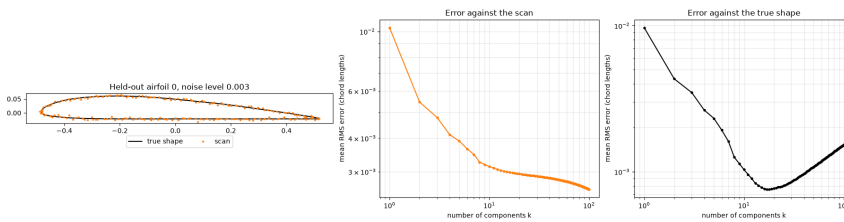
- (a) Subtract the mean airfoil from `x_train`, compute the singular value decomposition $X = USV^T$ with `np.linalg.svd`, and compare the first 20 rows of V^T

with `estimator.components_`. How do they agree, and why can two correct PCA implementations differ in this way?

- (b) Compute the explained variance ratio of each component from the singular values and compare it with `estimator.explained_variance_ratio_`. Why does the variance along a component depend on the square of its singular value?

In the next exercise the held-out airfoils come back from a scanner that adds noise to every coordinate. The cell below simulates the scans. It also fits a PCA with 100 components, because the chapter's `estimator` has only 20. You can change `noise_level` and re-run this cell and the figure cell after it.

Run the cell below to produce the figure for Exercise 3. The left panel shows one scan and the true shape. The other two panels project every scan onto the first k components of `pca_100` and back, for k from 1 to 100, and plot the mean RMS error against the scan (middle) and against the true shape (right). Both axes are logarithmic.



Exercise 3 (~6 min): Cleaning up a noisy scan

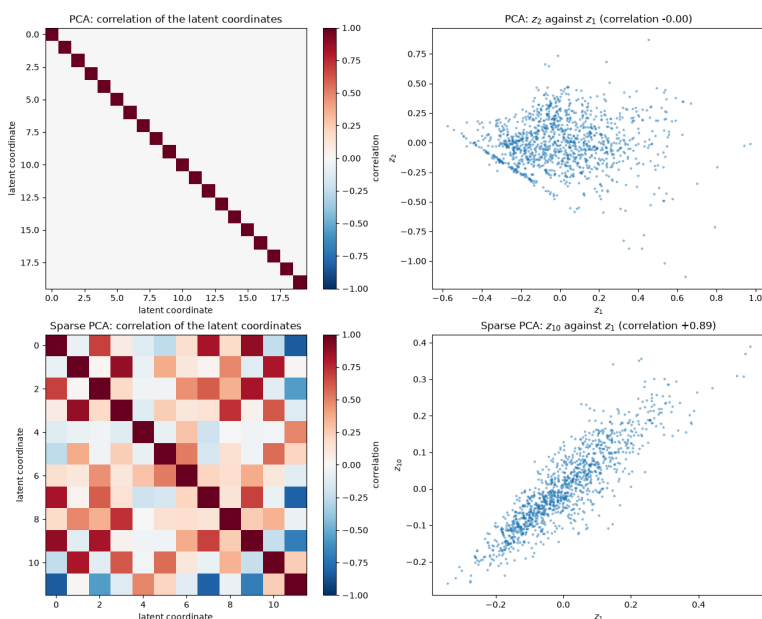
Each scanned coordinate is $\tilde{x}_i = x_i + \varepsilon_i$. The noise ε_i is drawn independently for each coordinate from a normal distribution with density $p(\varepsilon_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\varepsilon_i^2}{2\sigma^2}\right)$, where $\sigma = \text{noise_level}$. Projecting a scan onto a few principal components and back gives a cleaned airfoil. How many components should we use?

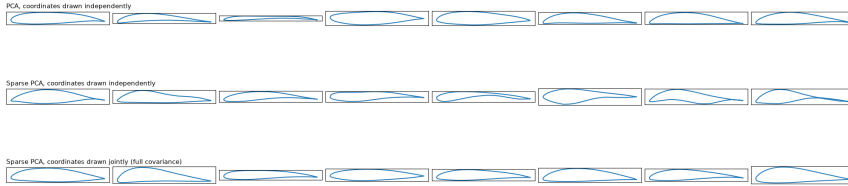
- (a) Why does the error against the scan keep falling as k grows, while the error against the true shape falls and then rises? What do the extra components reconstruct?
- (b) Which k would you use, and why? In practice you only have the scans, not the true shapes. Why can the middle panel not tell you the answer, and how could you choose k from the scans alone?
- (c) Set `noise_level` to 0.001 and then to 0.01, and re-run both cells. Which way does the best k move, and why?

Run the cell below to produce the two figures for Exercise 4. The first figure shows how the latent coordinates of the training airfoils are distributed, for PCA (top) and Sparse PCA (bottom): the correlation matrix on the left and a scatter plot of one pair of coordinates on the right. The second figure draws random latent coordinates z , maps them back to airfoils with W , and plots the shapes. Each row draws z differently:

Row	Model	How the latent coordinates are drawn
1	PCA	each coordinate separately, from a normal distribution with that coordinate's mean μ_j and variance σ_j^2 over the training airfoils
2	Sparse PCA	as in row 1
3	Sparse PCA	all coordinates together, from a multivariate normal distribution with the mean vector μ and covariance matrix Σ of the training airfoils' latent coordinates

Rows 1 and 2 use the density $p(z) = \prod_j \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(z_j - \mu_j)^2}{2\sigma_j^2}\right)$, and row 3 uses $p(z) = \frac{1}{\sqrt{(2\pi)^k \det \Sigma}} \exp\left(-\frac{1}{2}(z - \mu)^\top \Sigma^{-1}(z - \mu)\right)$, where k is the number of components.



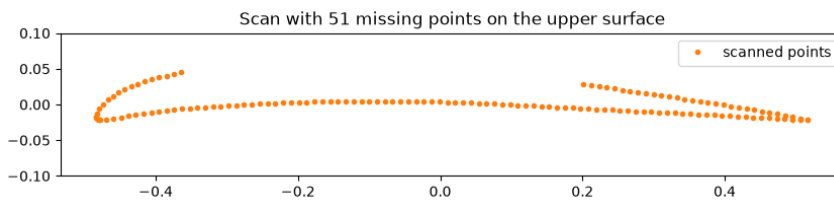


Exercise 4 (~6 min): How are the latent coordinates distributed?

Each fitted airfoil is a point z in the latent space. If we knew how these points are distributed, we could sample new points and get new airfoils. The figures above try the simplest options.

- (a) The PCA correlation matrix is diagonal and the Sparse PCA one is not. Why does the PCA objective make the latent coordinates uncorrelated, and what in the Sparse PCA objective removes that property?
- (b) Which rows contain plausible airfoils? Why does sampling the Sparse PCA coordinates independently fail when it works for PCA, and why does sampling them jointly fix it?
- (c) The PCA coordinates z_1 and z_2 are uncorrelated, but their scatter plot has a sharp straight edge on one side. Are they independent? Which training airfoils lie along that edge (you can look some up), and what does a normal distribution do there that the data does not? What does this mean for using a linear decomposition with a normal distribution to generate new designs?

The last exercise uses a scan with a gap. The cell below takes one held-out airfoil and hides 51 consecutive points on its upper surface, as if a clamp covered that part during the scan. The scanner returns the other 141 points.



Exercise 5 (~12 min): Filling in a gap in a scan

The chapter's PCA basis was fitted on complete airfoils, so it describes how the points of an airfoil move together. We can use it to fill in the gap. Write a function `inpaint(observed_xy, observed_points, k)` that finds the k latent coordinates whose reconstruction best fits the observed points in the least-squares sense, and returns the

full airfoil with all 192 points. This is a linear regression like those in the first chapter, with one equation per observed coordinate. Use $k = 10$. This is a longer take-home exercise.

- (a) Plot your reconstruction with the scanned points and the true airfoil (`airfoil_true`), and compute the RMS error over the missing points. Why can ten numbers fitted to the points around the gap predict where the 51 missing points are?
- (b) Now hide the whole lower surface instead (points 97 to 191) and repeat. You still have almost half of the airfoil, but the result is much worse. Why?
- (c) Your function returns one airfoil. Is it the only airfoil that fits the scanned points? If you had to decide whether the hidden part is within its manufacturing tolerance, what would you want the model to return instead, and why?

5 Taking Derivatives with Automatic Differentiation

Taking derivatives with respect to functions or parameters is one of the most common and fundamental operations that we will need for Machine Learning (and in fact for Scientific Computing, in general). Throughout your life so far, you have probably learned about three main ways to compute derivatives:

1. **Analytical Differentiation:** computing the analytical derivative manually using the rules of calculus and pencil and paper. The benefit of this approach is that it is exact, but the drawback is that it can be tedious, manual, and error-prone for complex functions. Once completed, we get a function that has the exact derivative everywhere.
2. **Symbolic Differentiation:** using a computer algebra system (CAS) to compute the analytical derivative symbolically. The main benefit of this approach is that it is exact and can handle more complex functions than manual differentiation, but the drawback is that it can be slow and may produce very complicated expressions that are difficult to evaluate numerically (a phenomenon known as *expression swell*). This method also produces a function that has the exact derivative everywhere as a single (often very large) function.
3. **Numerical Differentiation:** using finite difference approximations to compute the derivative numerically, for example using the formula $\frac{df}{dx} \approx \frac{f(x+h)-f(x)}{h}$ for some small h . The main benefit of this approach is that it is easy to implement and can handle any function that can be evaluated numerically, but its drawbacks are that it only approximates the derivative, can be sensitive to the choice of h , suffers from numerical precision issues, and only provides the derivative at a single evaluated point (unlike analytical or symbolic approaches that produce a function of the derivative that is valid everywhere).

This chapter won't cover those approaches in detail, as its main goal is to introduce you to a fourth way to compute derivatives called *Automatic Differentiation* (AD). This approach inherits some of the benefits of analytical and symbolic differentiation, in that it computes *exact* derivatives (unlike numerical differentiation). Its main drawback is that the effort used to compute the derivatives will be only useful for a single evaluation point (similar to numerical differentiation). Unlike symbolic differentiation, AD is very efficient and does not suffer from expression swell, although it does require extra bookkeeping to keep track of intermediate values and derivatives (as we will see), which can add some memory and computational overhead.

Table 5.1: Overview of different methods for computing derivatives.

Method	Benefits	Drawbacks
Analytical Differentiation	Exact derivatives, valid everywhere	Tedious, manual, error-prone for complex functions
Symbolic Differentiation	Exact derivatives, can handle complex functions	Slow, may produce complicated expressions (expression swell)
Numerical Differentiation	Easy to implement, handles any numerically evaluable function	Only approximates derivative, sensitive to h , precision issues, pointwise
Automatic Differentiation	Exact derivatives, efficient computation	More complex implementation, potential overhead

With this overview in mind, we can now introduce Automatic Differentiation using a simple example, and then later demonstrate how to use it via some practical examples.

5.1 Automatic Differentiation

To demonstrate how Automatic Differentiation works, let's take a simple example function, for which we can easily compute the derivatives manually/analytically, so that we can check out results. Let's consider the function:

$$f(x_1, x_2) = \ln(x_1) + x_1 \cdot x_2 - \sin(x_2)$$

evaluated at $x_1 = 2, x_2 = 5$.

Analytically, this function is simple enough that we could actually compute the partial derivatives manually:

$$\frac{\partial y}{\partial x_0} = \frac{1}{x_0} + x_1 = 0.5 + 5 = 5.5$$

$$\frac{\partial y}{\partial x_1} = x_0 - \cos(x_1) = 2 - 0.284 = 1.716$$

However, for the sake of this example, we will use Automatic Differentiation to compute these derivatives instead. The first step of Automatic Differentiation is to build up a *Computational Graph* of the function using a library of easy to compute derivatives (e.g., $\frac{\partial}{\partial x} x^n \rightarrow n \cdot x^{n-1}$)

5.1.1 Build the Computational Graph

To build the computational graph, we break the function down into its elementary operations step by step, starting from the beginning (i.e., the inputs to the function). We will introduce intermediate variables ($V_{\#}$) to represent the outputs of these intermediate operations. Let's define:

$V_2 = V_0 \rightarrow V_3$ $V_1 \rightarrow V_3$ $V_1 \rightarrow V_4$

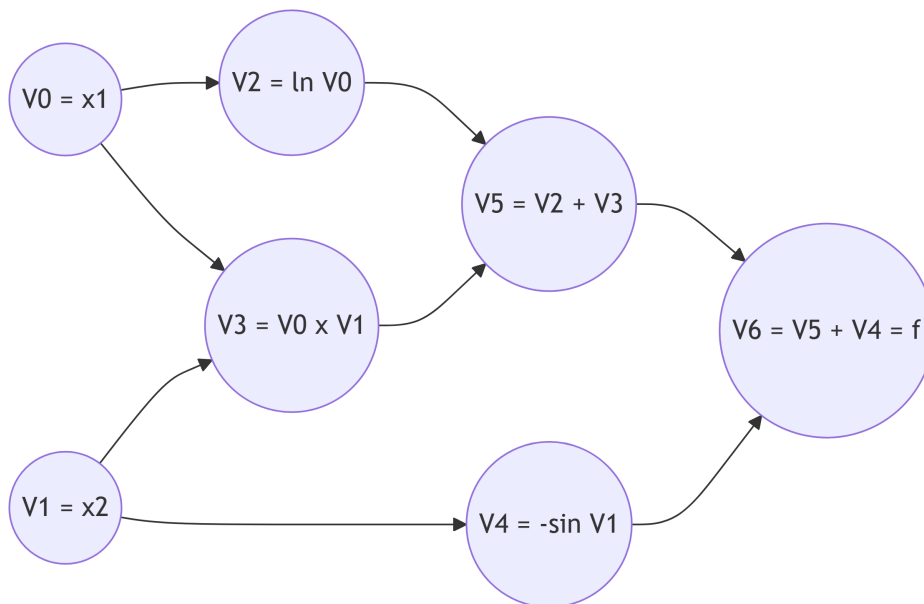
%% Sum nodes

$V_2 \rightarrow V_5$

$V_3 \rightarrow V_5$

$V_5 \rightarrow V_6$

$V_4 \rightarrow V_6$



->

Now with the graph in place, we can compute the first stage of Automatic Differentiation, which is the *forward pass* through the function.

5.1.2 Compute the Forward Pass

The Forward Pass is simply evaluating the function at the given input values, but we will do this step by step, following the computational graph we just built. It will be useful to keep track of the intermediate values in a table for reference later. (In reality, the computer will do this for us, but we are doing it by hand here to illustrate the process.)

Node	Definition	Value
V0	x_1	2
V1	x_2	5
V2	$\ln(V0)$	$\ln(2) \approx 0.6931$
V3	$V0 \cdot V1$	$2 \cdot 5 = 10$
V4	$-\sin(V1)$	$-\sin(5) \approx 0.9589$
V5	$V2 + V3$	$0.6931 + 10 = 10.6931$
V6	$V5 + V4$	$10.6931 + 0.9589 \approx 11.6520$

If we pass in our initial points (x_1, x_2) through this forward pass, we can look at the final node (V6) to get the answer:

$$f(2, 5) \approx 11.6520$$

Great, this matches what we would expect. At this point, we have just evaluated the function forward, and we don't yet have any derivatives. From here, things get interesting and bifurcate into two main types of Automatic Differentiation: Forward Mode AD and Backward Mode AD. Each of these has important but different uses for reasons that will become clear as we work through the example. Let's start with Forward Mode AD. In both cases, we will start with the initial work we already did with the forward pass above.

5.1.3 Computing Forward Mode AD (tangent propagation)

Forward Mode AD allows us to compute *directional derivatives* of the function, as well as that same directional derivative at *any intermediate node* in the computational graph. This is useful for computing derivatives of functions that have a small number of inputs (e.g., 1-10), but potentially a large number of outputs that we might be interested in. For example, if we were computing a trajectory of a dynamical system, we might want to know how the final state of the system changes with respect to some initial condition. In this case, the initial condition is the input, and the final state is the output. There might be many intermediate states along the way that we also want to know how they change with respect to the initial condition, such as the state or total energy at each time step. Forward Mode AD allows us to compute all of these derivatives in only a single pass through the computational graph.

To see how this works, we will introduce a new variable $\dot{V}$ to represent the derivative of each node with respect to some input direction. We will use the notation $\dot{V} = \frac{dV}{dx}$, where x is some input variable.

Case A: derivative wrt x_1 ($\dot{x}_1 = 1, \dot{x}_2 = 0$)

Node	Definition	Value
$\dot{V}_0$	$\dot{x}_1$	1

Node	Definition	Value
$\dot{V}_1$	$\dot{x}_2$	0
$\dot{V}_2$	$d(\ln V_0)/dV_0 = 1/V_0 \dot{V}_0$	$(1/2) \cdot 1 = 0.5$
$\dot{V}_3$	$d(V_0 \cdot V_1)/dV_0 = \dot{V}_0 V_1 + V_0 \dot{V}_1$	$5 \cdot 1 + 2 \cdot 0 = 5$
$\dot{V}_4$	$-\cos(V_1) \dot{V}_1$	$-\cos(5) \cdot 0 = 0$
$\dot{V}_5$	$\dot{V}_3 + \dot{V}_2$	$0.5 + 5 = 5.5$
$\dot{V}_6$	$\dot{V}_5 + \dot{V}_4$	$5.5 + 0 = 5.5$

$$\frac{\partial f}{\partial x_1} = 5.5$$

Case B: derivative wrt x_2 ($\dot{x}_1 = 0, \dot{x}_2 = 1$)

Exercise: Fill in a similar table as above to compute the derivative of the function with respect to x_2 using Forward Mode AD.

You can check your answer using the known analytical derivative of the function that you can compute by hand:

$$\frac{\partial f}{\partial x_2} \approx 1.7163$$

OK, great, we see that with some bookkeeping, we have correctly computed the derivatives of the function with respect to each input variable. Note that we had to do two passes through the computational graph to get both derivatives, since they were different directional derivatives. If we had a third input variable, we would need a third pass, and so on. This is why Forward Mode AD is best suited for functions with a small number of inputs.

On the flip side, if we were interested in the derivative of some intermediate node in the computational graph with respect to an input variable, we would have that information available as well. For example, if we wanted to know how V_3 changes with respect to x_1 , we can see from the table above that $\frac{\partial V_3}{\partial x_1} = 5$. We got this in the process of computing the final function, so this derivative comes “along for the ride” without additional cost on our part. This is a powerful feature of Forward Mode AD that we will see is not available in Backward Mode AD: we can get a specific directional derivatives of any intermediate node from the computational graph via the same Forward Mode pass, but the cost of this scales with the number of input variables or number of directional derivatives we want to compute.

5.1.4 Computing Reverse Mode AD (backpropagation)

While Forward Mode AD efficiently found directional derivatives, if we wanted to compute the full gradient of the output with respect to all of the input variables, we would need to

do a separate pass for each input variable. This means that the cost of computing the full gradient scales with the number of input variables. If we have a function with a large number of input variables (e.g., 1000s or more), this can be very expensive. To address this, we can use Reverse Mode AD, which will allow us to compute the full gradient of every input variable to a function using a single “backward pass” through the computational graph. This is particularly useful for functions that have a small number of outputs (e.g., 1-10), but a very large number of inputs, such as a Neural Network in Machine Learning or mesh coordinates in a Finite Element or Computational Fluid Dynamics simulation.

To see how this works, we will introduce a new variable $\bar{V}$ to represent the *adjoint* of each node with respect to the output. We will use the notation $\bar{V} = \frac{\partial f}{\partial V}$, where f is the final output of the function. The adjoint represents how much a small change in that intermediate node would affect the final output of the function.

Similarly to Forward Mode AD, we will have to pick a specific output that we wish to compute the gradient with respect to. In this case, since we are computing the gradient of the output with respect to all input variables, we can set the final output node with a value of 1, i.e., $\bar{V}_6 = 1$. From there, we will propagate each adjoint backward through the computational graph using the chain rule.

Node	Equation for the adjoint	Value
$\bar{V}_6$	$\frac{\partial f}{\partial V_6}$	1
$\bar{V}_5$	$\frac{\partial f}{\partial V_6} \frac{\partial V_6}{\partial V_5} = \bar{V}_6 1$	1
$\bar{V}_4$	$\frac{\partial f}{\partial V_6} \frac{\partial V_6}{\partial V_4} = \bar{V}_6 1$	1
$\bar{V}_3$	$\frac{\partial f}{\partial V_5} \frac{\partial V_5}{\partial V_3} = \bar{V}_5 1$	1
$\bar{V}_2$	$\frac{\partial f}{\partial V_5} \frac{\partial V_5}{\partial V_2} = \bar{V}_5 1$	1
$\bar{V}_1$	from V3: $\bar{V}_3 \cdot V_1 = 2$ + from V4: $-\cos(5) \cdot \bar{V}_4 \approx -0.2837$	1.7163
$\bar{V}_0$	from V2: $\bar{V}_2(1/V_0) \cdot 1 = 0.5$ + from V3: $\bar{V}_3 \cdot V_1 = 5$	5.5

We can now verify the final gradients, which match our analytical solution:

$$\nabla f(2, 5) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \right) = (5.5, 1.7163)$$

However, unlike Forward Mode AD, we see that we now have access not only to the gradients of the input variables, but also to the gradients of **all intermediate nodes** in the computational graph, and we received all of them via the same amount of work/computation!

5.2 PyTorch Autograd Example with Simple Function

Ok let's use automatic differentiation to compute a simple derivative of our earlier analytical function:

$$y = f(x_0, x_1) = \ln(x_0) + x_0 \cdot x_1 - \sin(x_1)$$

And as with before, we'll evaluate the derivative of this function at

$$x_0 = 2, x_1 = 5$$

We can analytically compute the derivative and code it up so that we can verify accuracy later:

```
import numpy as np
def true_grad(x0,x1):
    return np.array([
        1/x0 + x1,
        x0 - np.cos(x1)
    ])
true_grad(2,5)
```

```
array([5.5      , 1.71633781])
```

But now let's see how to use [PyTorch](#) to get this using Automatic Differentiation:

5.2.1 Automatic Differentiation using PyTorch

```
tensor([2., 5.], requires_grad=True)
```

```
tensor(11.6521, grad_fn=<SubBackward0>)
```

```
tensor([5.5000, 1.7163])
```

Let's see how well it approximated the true gradient:

```
array([0.00000000e+00, 1.45108339e-08])
```

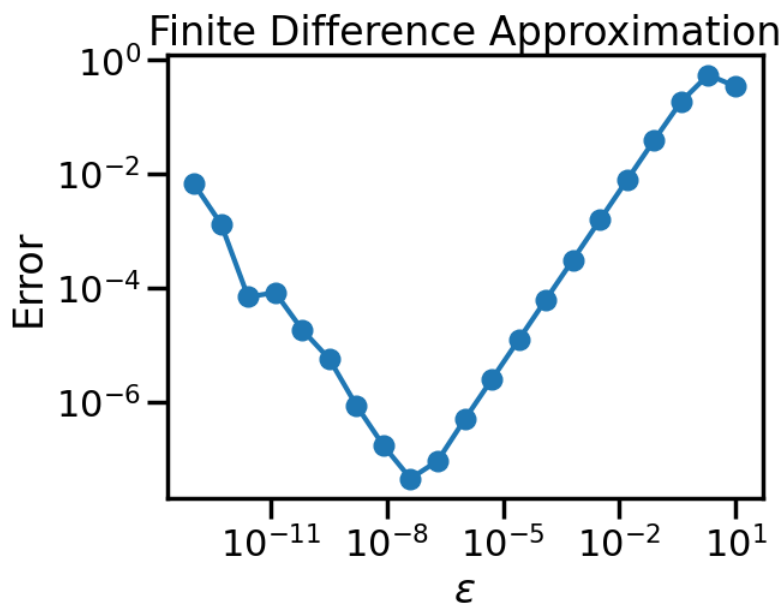
5.2.2 Finite Differences using SciPy

Now let's compare this to computing the same gradient, but using Numerical Differentiation (specifically, Central Finite Differences):

```
array([5.4999875 , 1.71628987])
```

Let's see how well it approximated the true gradient:

```
array([1.24995903e-05, 4.79457300e-05])
```

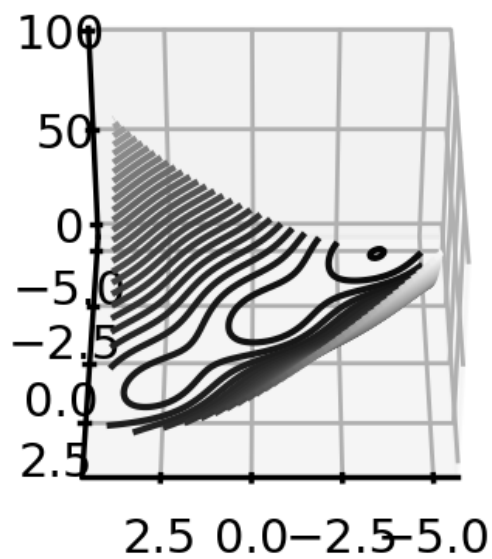
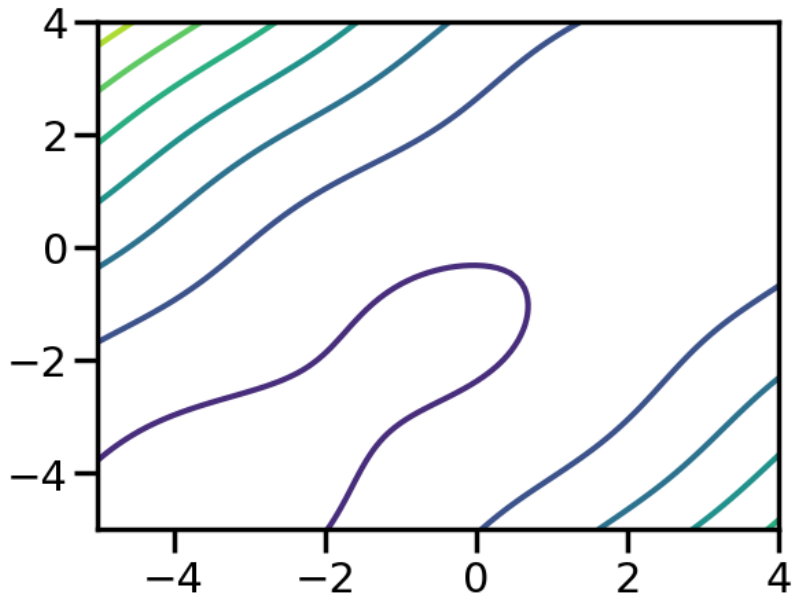


5.3 PyTorch Autograd Example with Optimization

This example shows how to use AD and PyTorch to perform gradient based optimization on a simple test function

What does this function look like?

```
c:\Users\fuge\AppData\Local\miniforge3\envs\ml4me-student\Lib\site-packages\torch\functional.py:554: Us
    return _VF.meshgrid(tensors, **kwargs) # type: ignore[attr-defined]
```



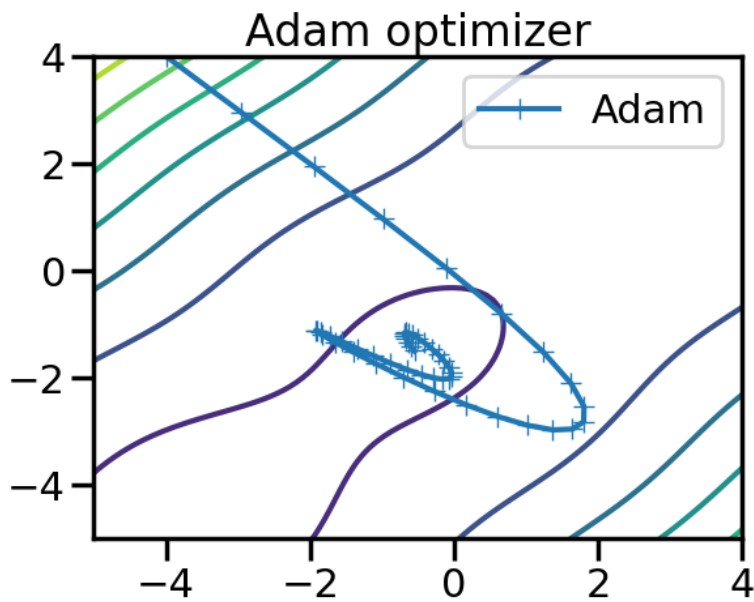
Now let's say I want to optimize this. I could compute the analytical derivative. Or, I could compute the backward-mode AD on the inputs:

```
tensor([-16.5000, 19.5000])
```

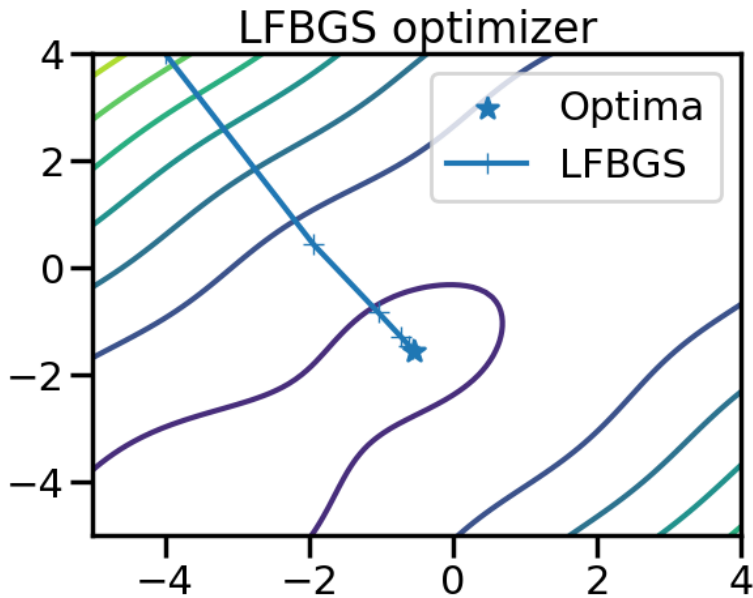
Now we just stick it in a loop and run SGD on it:

```
tensor([-2.9600,  2.9600], requires_grad=True)
tensor([-1.9481,  1.9436], requires_grad=True)
tensor([-0.9857,  0.9653], requires_grad=True)
tensor([-0.1051,  0.0453], requires_grad=True)
tensor([ 0.6508, -0.7906], requires_grad=True)
tensor([ 1.2368, -1.5138], requires_grad=True)
tensor([ 1.6219, -2.0992], requires_grad=True)
tensor([ 1.8020, -2.5325], requires_grad=True)
tensor([ 1.7979, -2.8129], requires_grad=True)
tensor([ 1.6423, -2.9514], requires_grad=True)
tensor([ 1.3704, -2.9664], requires_grad=True)
tensor([ 1.0147, -2.8804], requires_grad=True)
tensor([ 0.6043, -2.7165], requires_grad=True)
tensor([ 0.1659, -2.4983], requires_grad=True)
tensor([-0.2759, -2.2482], requires_grad=True)
tensor([-0.6981, -1.9879], requires_grad=True)
tensor([-1.0796, -1.7375], requires_grad=True)
tensor([-1.4026, -1.5143], requires_grad=True)
tensor([-1.6534, -1.3326], requires_grad=True)
tensor([-1.8238, -1.2020], requires_grad=True)
tensor([-1.9113, -1.1271], requires_grad=True)
tensor([-1.9188, -1.1078], requires_grad=True)
tensor([-1.8539, -1.1396], requires_grad=True)
tensor([-1.7274, -1.2146], requires_grad=True)
tensor([-1.5524, -1.3224], requires_grad=True)
tensor([-1.3433, -1.4512], requires_grad=True)
tensor([-1.1152, -1.5883], requires_grad=True)
tensor([-0.8830, -1.7215], requires_grad=True)
tensor([-0.6606, -1.8395], requires_grad=True)
tensor([-0.4605, -1.9328], requires_grad=True)
tensor([-0.2926, -1.9947], requires_grad=True)
tensor([-0.1641, -2.0210], requires_grad=True)
tensor([-0.0788, -2.0109], requires_grad=True)
tensor([-0.0372, -1.9664], requires_grad=True)
tensor([-0.0366, -1.8921], requires_grad=True)
tensor([-0.0720, -1.7947], requires_grad=True)
tensor([-0.1359, -1.6823], requires_grad=True)
tensor([-0.2198, -1.5637], requires_grad=True)
tensor([-0.3146, -1.4478], requires_grad=True)
tensor([-0.4113, -1.3430], requires_grad=True)
tensor([-0.5017, -1.2561], requires_grad=True)
tensor([-0.5791, -1.1926], requires_grad=True)
tensor([-0.6384, -1.1558], requires_grad=True)
```

```
tensor([-0.6769, -1.1466], requires_grad=True)
tensor([-0.6937, -1.1639], requires_grad=True)
tensor([-0.6901, -1.2045], requires_grad=True)
tensor([-0.6693, -1.2638], requires_grad=True)
tensor([-0.6353, -1.3361], requires_grad=True)
tensor([-0.5934, -1.4148], requires_grad=True)
tensor([-0.5490, -1.4935], requires_grad=True)
```



```
tensor([-1.9382,  0.4356], requires_grad=True)
tensor([-1.0402, -0.8307], requires_grad=True)
tensor([-0.7223, -1.2888], requires_grad=True)
tensor([-0.6097, -1.4543], requires_grad=True)
tensor([-0.5696, -1.5139], requires_grad=True)
```

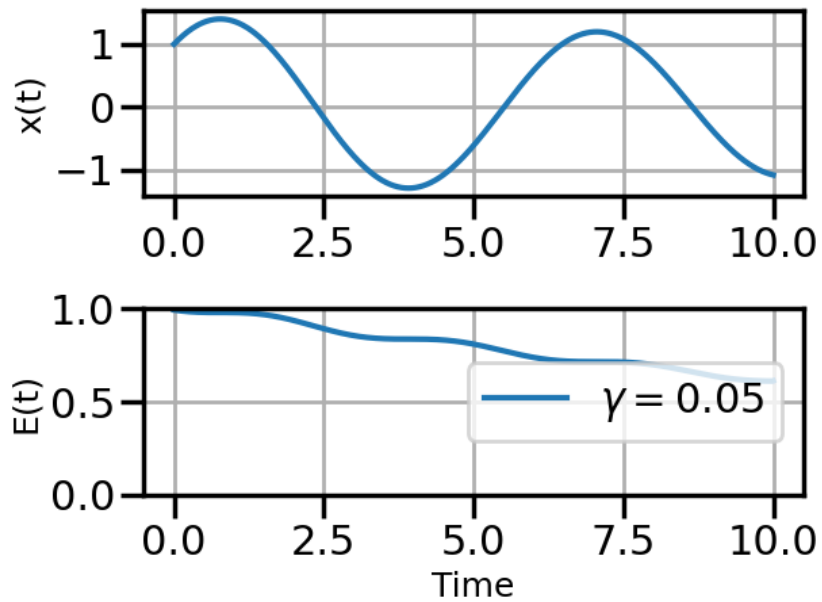



5.4 Demonstration of AD on Verlet Integration

This notebook demonstrates how to use Automatic Differentiation to determine the gradients of the initial conditions of a dynamical system (in this case a damped oscillator). To do this, we will define a numerical routine ([Verlet Integration](#)) and then use Automatic Differentiation to back propagate the gradient information from the output (Total system energy) to the initial conditions.

Below is a typical simulation script that is not current set up for automatic differentiation:

Now note how we modify the code to include AD via PyTorch (see comments and references to `torch`):



```
tensor([0.6137], grad_fn=<MulBackward0>)
```

Now let's print the gradient of the system Energy with respect to some of the initial conditions:

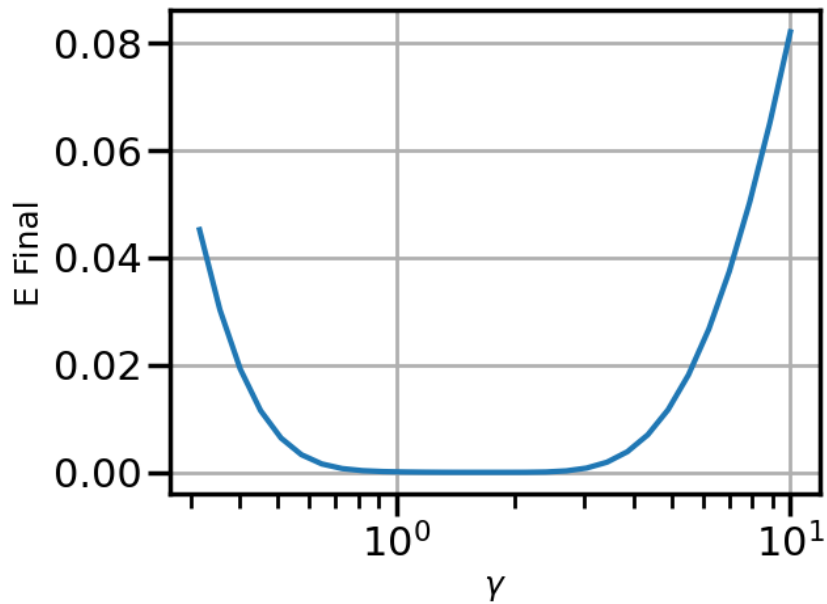
```
tensor([-5.9562])
tensor([0.6026])
tensor([0.6248])
```

```
tensor([-0.2245], grad_fn=<AddBackward0>)
```

```
tensor([-4.7535])
tensor([-0.0437])
tensor([1.0466])
```

5.4.1 Optimizing the Damping Coefficient via SGD and AD

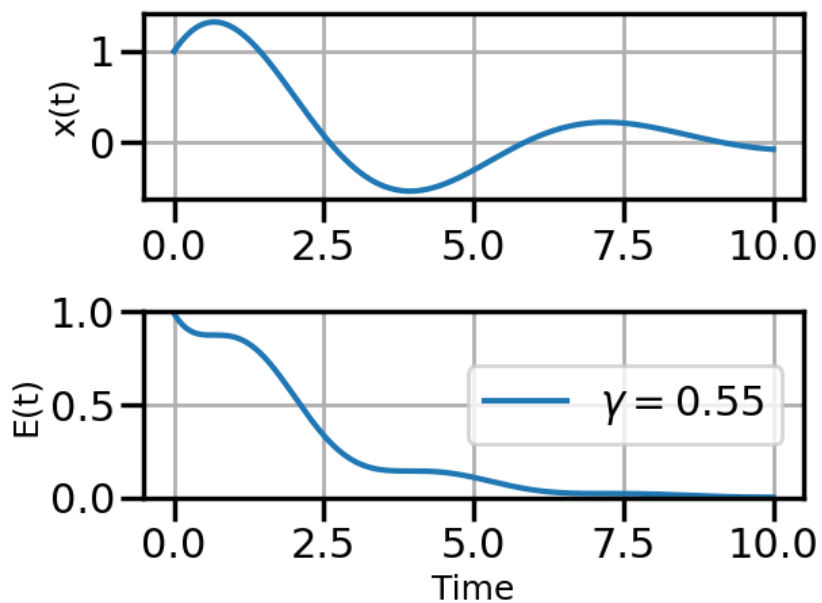
First let's just get a visual intuition for how γ affects the final energy:

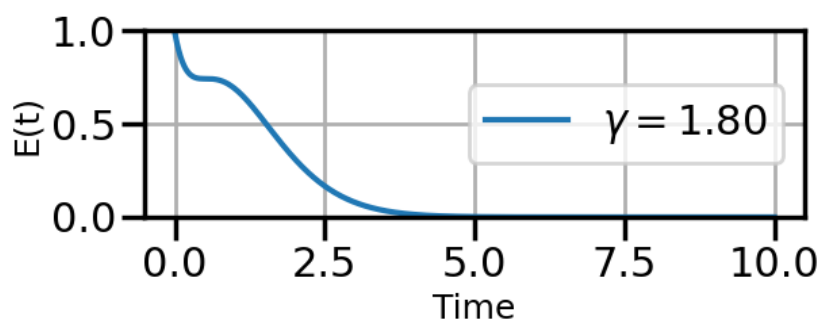
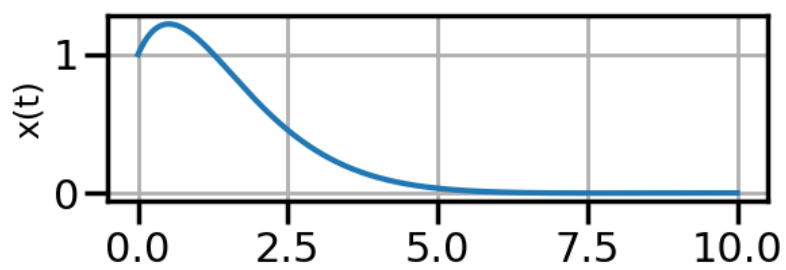
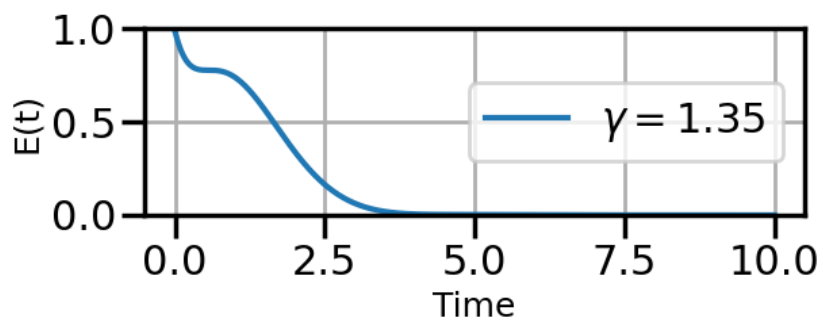
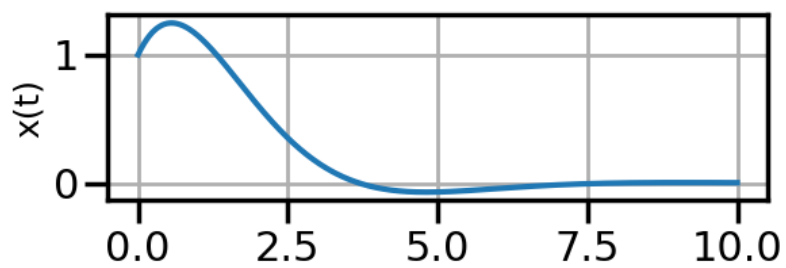


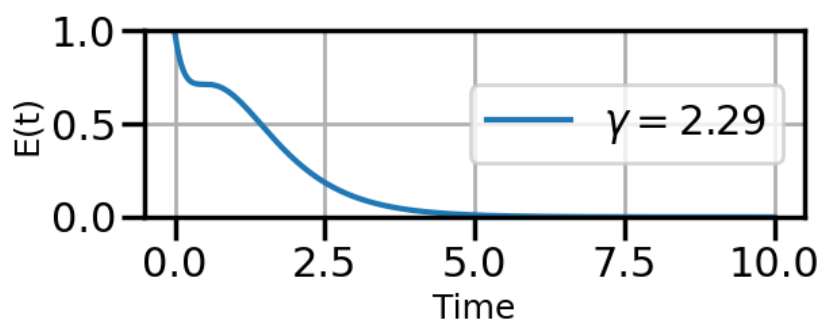
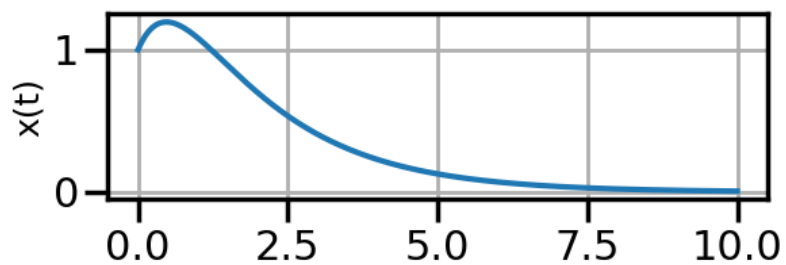
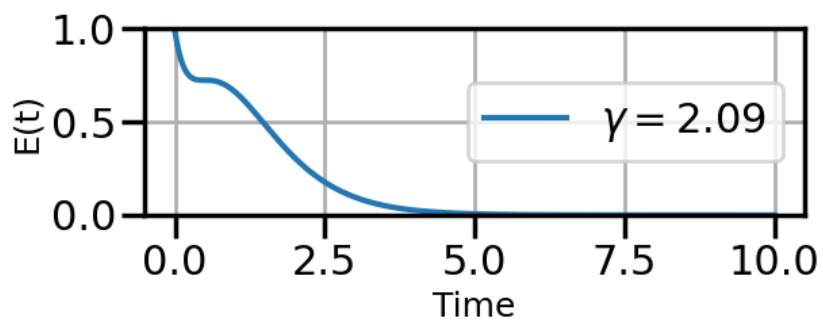
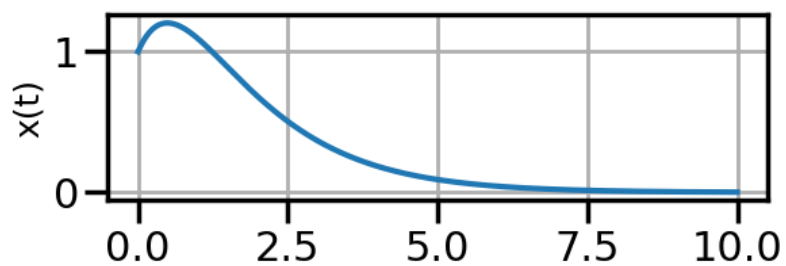
We can see that there is a pretty flat plateau from around $\gamma = 1$ until around $\gamma = 3$.

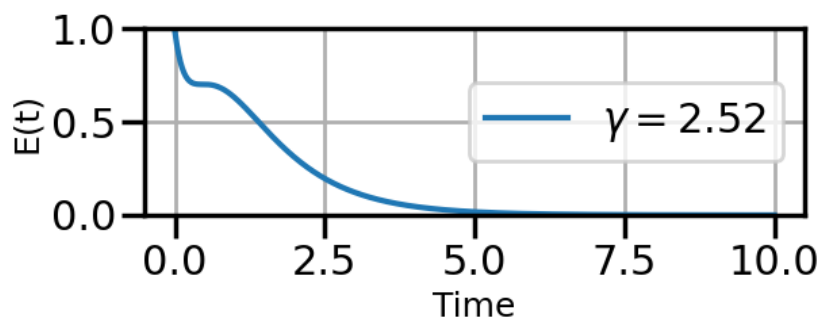
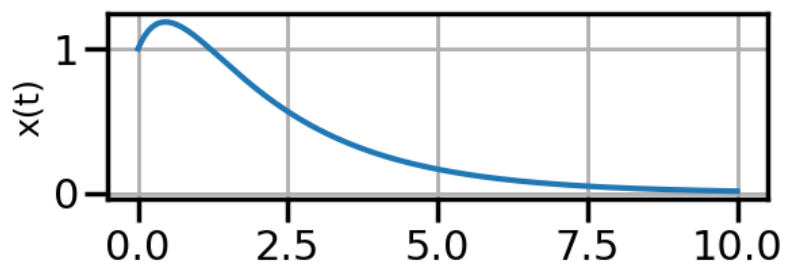
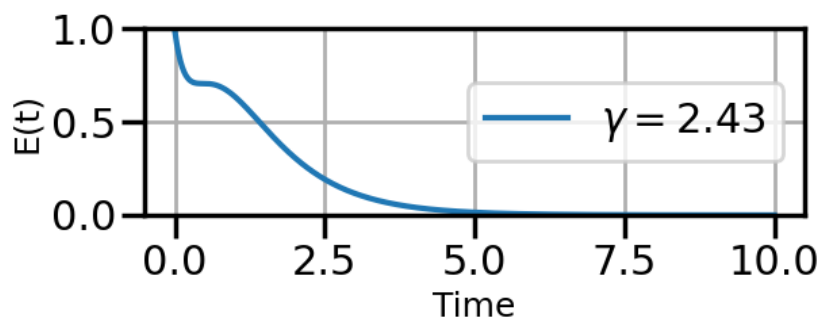
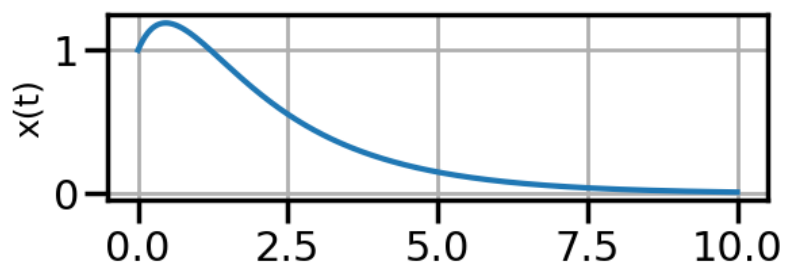
Now let's use our backward mode AD to actually optimize γ directly by calling backward on the output of the final energy of the Verlet integration of the ODE:

ADAM Example

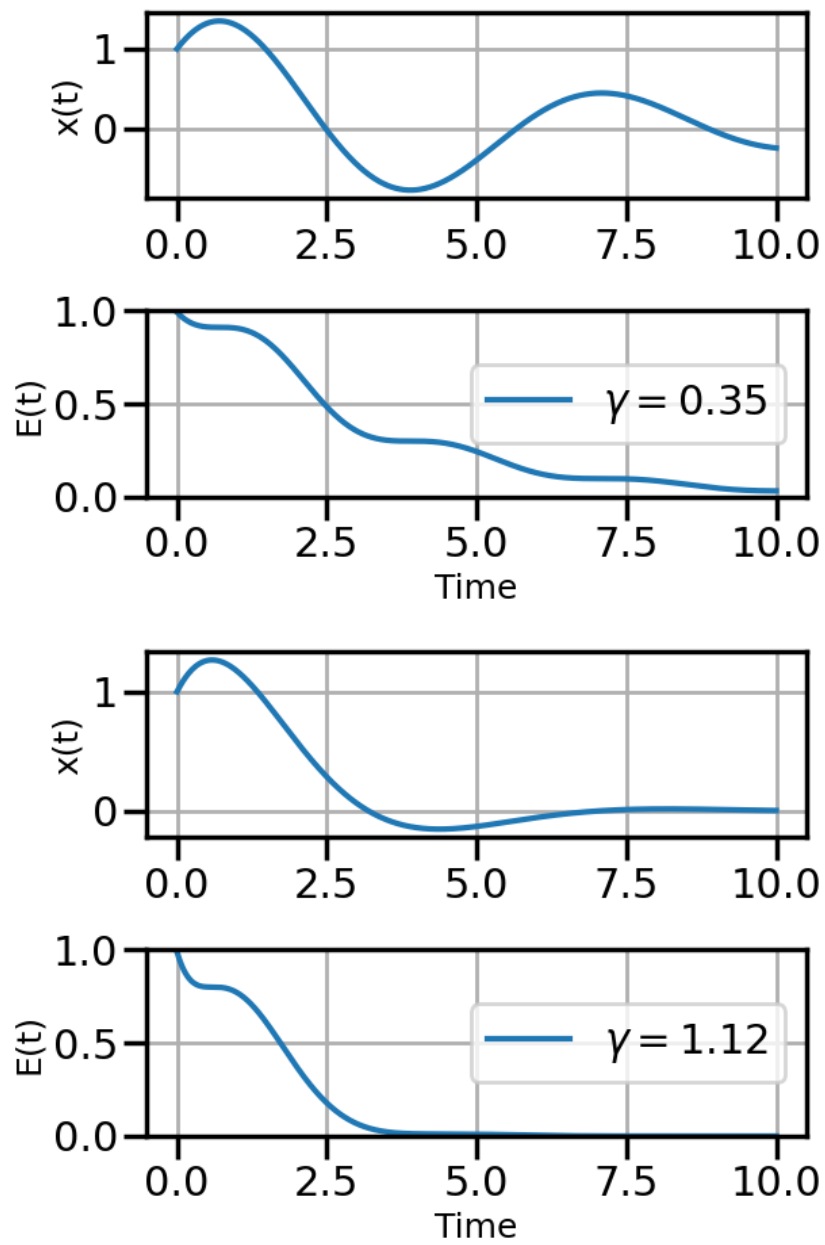


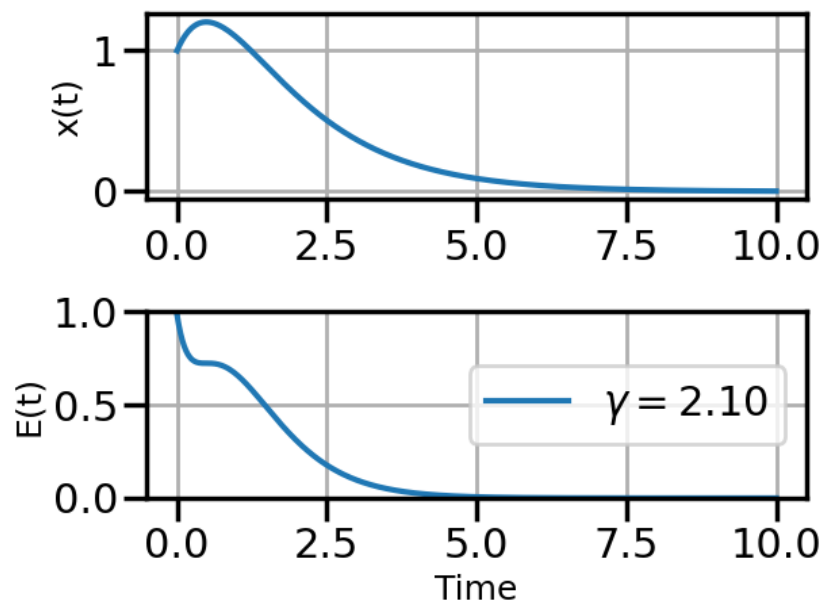
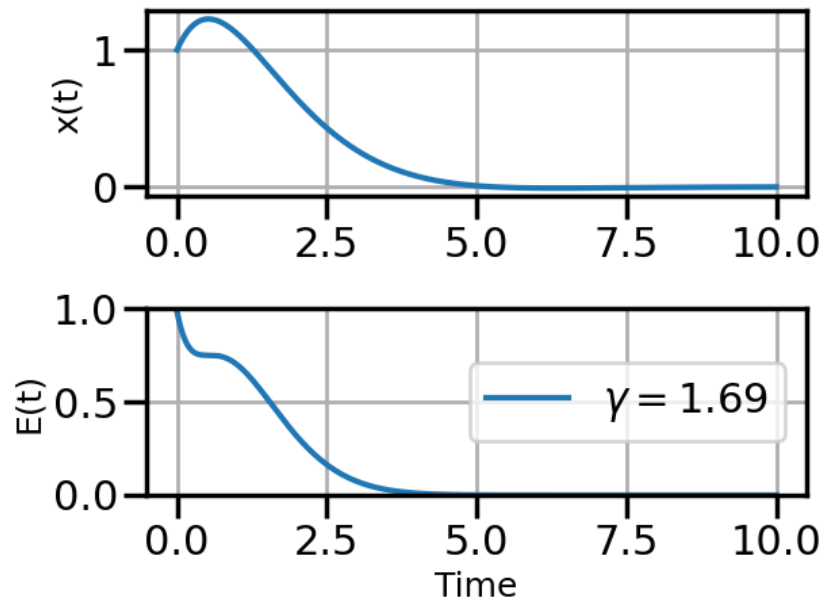


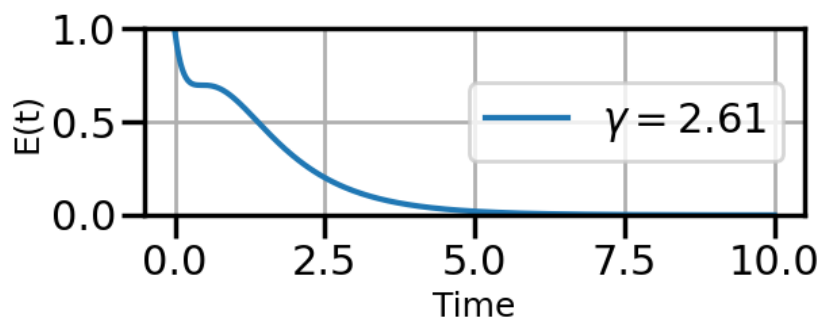
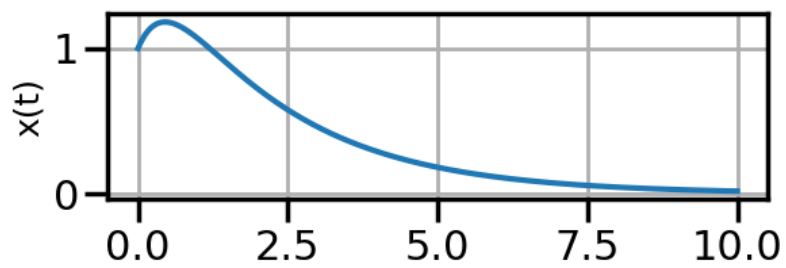
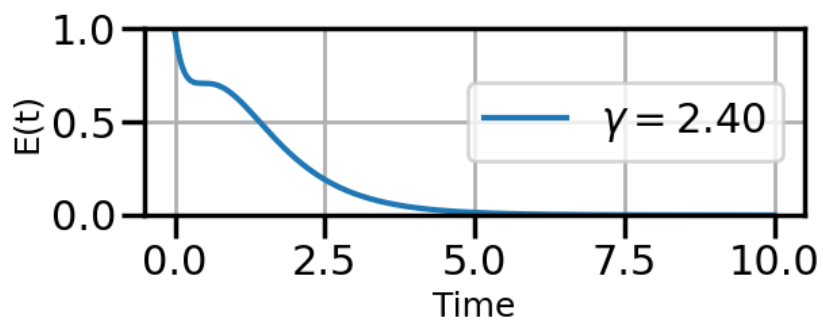
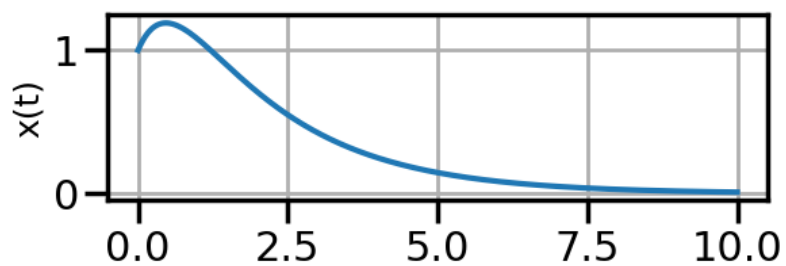


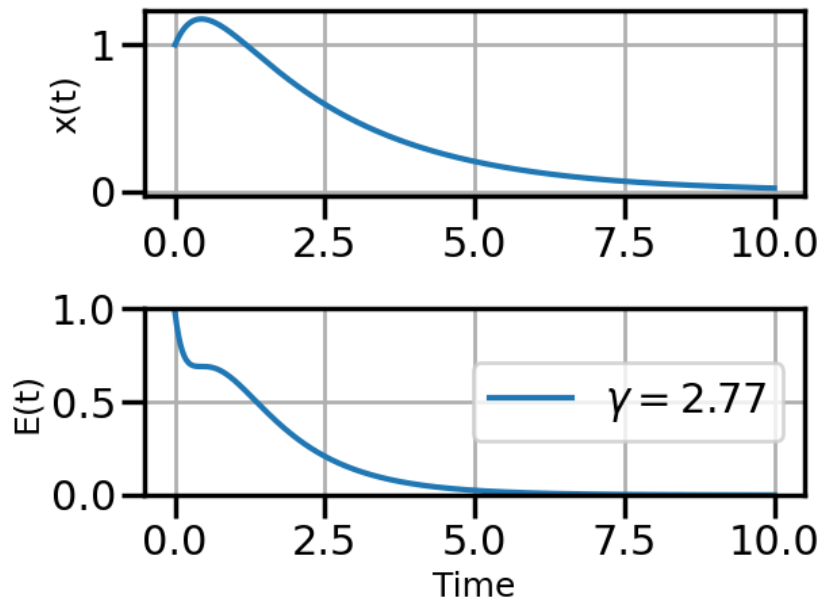


SGD Example





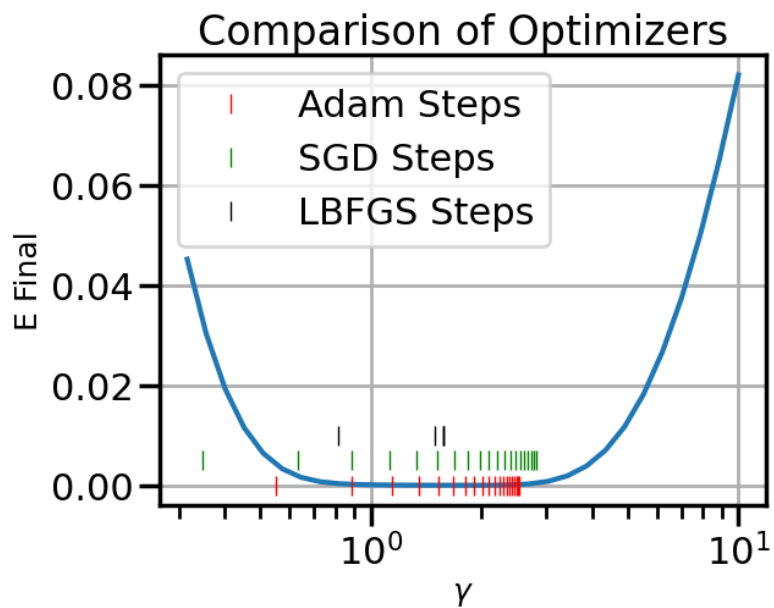




LBFGS Example

(Warning: Per-run solves of LBFGS take a while, so don't set num_steps too high here)

5.4.2 Compare the steps taken



6 Measuring Distribution Distances

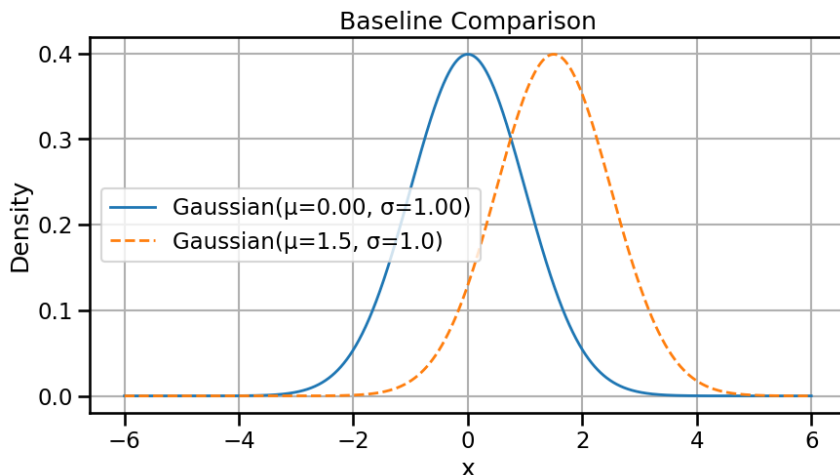
This notebook explores several ways of quantifying distances between one-dimensional probability distributions. We will contrast their mathematical definitions, visualize how they react to parameter changes, and compare the gradients that each metric produces. The goal is to build intuition for why the choice of divergence matters in optimization problems such as generative modeling or system identification. We will explore the following:

- Contrast forward KL, reverse KL, Jensen–Shannon divergence, Maximum Mean Discrepancy, Wasserstein distance, and Fisher–Rao metrics on simple families of distributions.
- Inspect how each distance responds to shifting the mean of a model Gaussian relative to a fixed target distribution.
- Visualize analytical or automatic-differentiation gradients to anticipate their effect on parameter updates.
- Experiment interactively with key hyperparameters (e.g., kernel bandwidth, mixture weights) to see how diagnostics change.

6.1 A Toy Example: Comparing Two Gaussians

We will compare a *target* distribution $p(x)$ against a *model* distribution $q_\mu(x)$ whose mean μ we can adjust. Unless stated otherwise, the target will be the standard normal distribution. You can change that later to a uniform or a bimodal mixture to reveal different behaviours.

Our baseline parameterization keeps the model standard deviation fixed at $\sigma_q = 1$. Adjusting the mean already highlights how asymmetric divergences produce different gradient signals.



6.2 Kullback–Leibler Divergence

Forward KL divergence compares how much mass the target distribution assigns relative to the model:

$$\text{KL}(p \parallel q) = \int p(x) \log \frac{p(x)}{q(x)} dx.$$

Because it integrates expectations under p , it heavily penalizes situations where $q(x)$ is small while $p(x)$ is large, even if the model allocates extra mass elsewhere.

6.2.1 Analytical Gradient for Gaussian vs. Gaussian

For $p = \mathcal{N}(\mu_p, \sigma_p^2)$ and $q = \mathcal{N}(\mu, \sigma_q^2)$, forward KL admits a closed form:

$$\text{KL}(p \parallel q) = \log \frac{\sigma_q}{\sigma_p} + \frac{\sigma_p^2 + (\mu_p - \mu)^2}{2\sigma_q^2} - \frac{1}{2}.$$

Differentiating w.r.t. the model mean gives

$$\frac{\partial}{\partial \mu} \text{KL}(p \parallel q) = \frac{\mu - \mu_p}{\sigma_q^2},$$

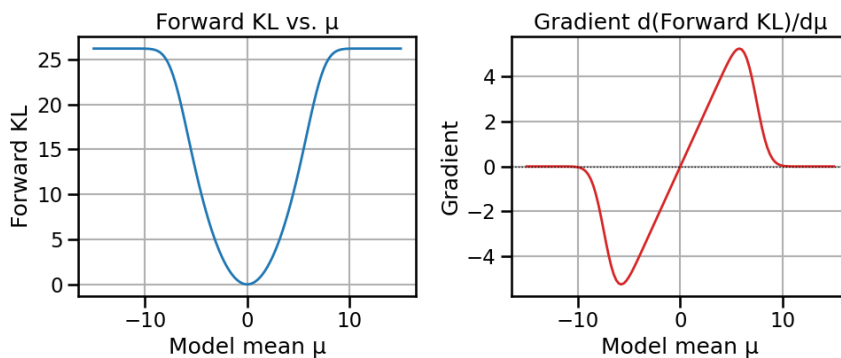
which pushes μ directly toward μ_p . We will evaluate the integral numerically to keep the pipeline consistent when we later switch to mixtures.

Now let's look at how the forward KL behaves as the model mean μ shifts away from the target mean $\mu_p = 0$. We will plot the KL divergence, and its gradient, as we shift the model mean from -15 to 15. What do you notice about the behavior of the KL Divergence and its gradient?

```
# Forward KL distance curve for standard normal target
##### Change the following to see what happens #####
mu_values = np.linspace(-15.0, 15.0, 121)
model_std = 1.0
#####

forward_metrics, forward_grads = evaluate_metric_curve(
    mu_values,
    lambda mu: forward_kl(mu, TARGETS["gaussian"], model_std=model_std),
)
plot_metric_and_gradient(mu_values, forward_metrics, forward_grads, metric_name="Forward KL", target_des
```

Forward KL against Gaussian($\mu=0.00$, $\sigma=1.00$)



💡 Experiment: Forward KL on Gaussian Targets

- What happens to both the KL Divergence and its gradient when the model mean is close to the target mean? What about when it is far away? Why is this?
- How would the curve change if we widened the model variance? Try modifying `model_std` in the code cell above and re-running it.

[Interactive output: available in the online version of this chapter.]

6.2.2 Forward vs. Reverse KL on a Bimodal Target

Reverse KL, $\text{KL}(q \parallel p)$, prefers to avoid regions where the model would place mass but the target does not. On multimodal targets this often leads to *mode seeking*: the optimizer chooses one mode to cover while ignoring others.

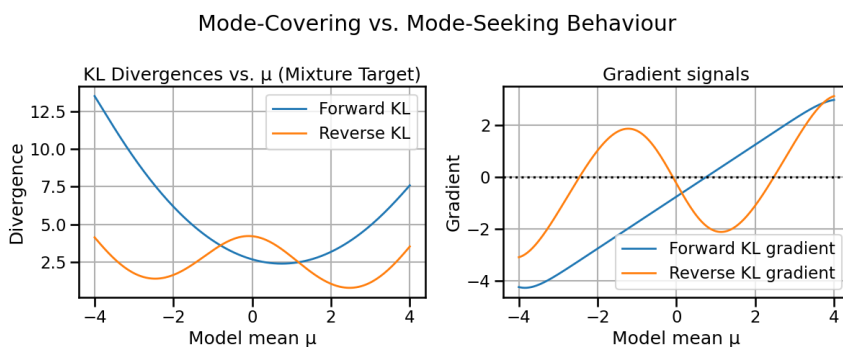
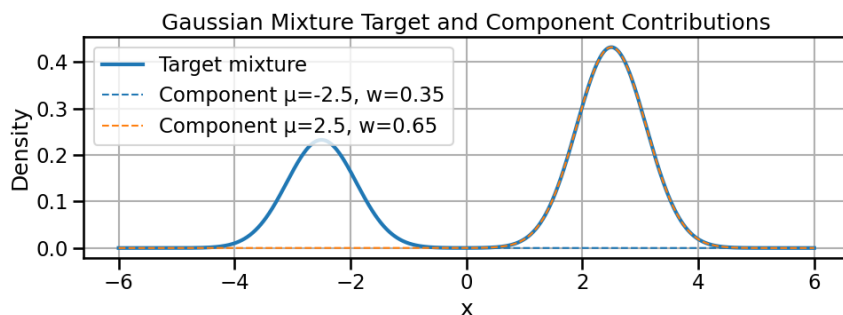
Mathematical Overview

The reverse KL is defined as

$$\text{KL}(q \parallel p) = \int q(x) \log \frac{q(x)}{p(x)} dx,$$

so the expectation is taken under the model distribution q . If $q(x)$ places mass where $p(x)$ is negligible, the logarithm term explodes and strongly penalizes those regions. Conversely, areas where p has mass but q does not contribute nothing, which is why the optimizer can collapse onto a single mode when matching multimodal targets.

Let us compare the two on a mixture of Gaussians.



💡 Reflection: Mode Covering vs. Mode Seeking

- Why does the forward KL gradient stay non-zero even between the two modes?
- Identify the regions where the reverse KL gradient vanishes. How does that explain a model that only tracks one mode?

6.3 Jensen–Shannon Divergence

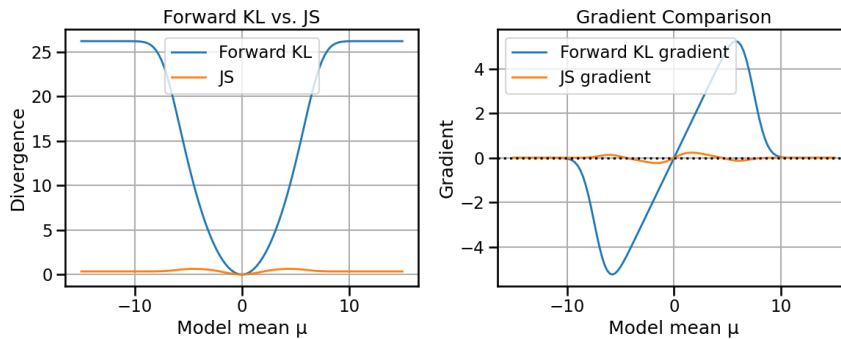
The Jensen–Shannon (JS) divergence symmetrizes KL by averaging the two distributions: $m(x) = \frac{1}{2}(p(x) + q(x))$. It stays finite even when supports do not match and is bounded between 0 and $\log 2$.

6.3.1 Mathematical Formulation

$$\text{JS}(p, q) = \frac{1}{2}\text{KL}(p \parallel m) + \frac{1}{2}\text{KL}(q \parallel m), \quad m = \frac{1}{2}(p + q).$$

We will compute JS numerically on the same grid. Autograd gives us gradients through the logarithms as long as we keep everything in PyTorch tensors.

```
def jensen_shannon(mu: torch.Tensor, target: DistributionSpec, model_std: float = 1.0) -> torch.Tensor:
    p_pdf = get_target_pdf(target)
    q_pdf = gaussian_pdf_torch(GRID_X, mu, model_std)
    m_pdf = 0.5 * (p_pdf + q_pdf)
    term_p = p_pdf * torch.log((p_pdf + EPS) / (m_pdf + EPS))
    term_q = q_pdf * torch.log((q_pdf + EPS) / (m_pdf + EPS))
    return 0.5 * torch.trapz(term_p, GRID_X) + 0.5 * torch.trapz(term_q, GRID_X)
```



💡 Reflection: Bounded Divergence, Softer Gradients

- JS approaches $\log 2$ when the two distributions barely overlap. Where in the plot does that happen?
- Compare the slope near $\mu = 0$ for JS and forward KL. What do you notice?
- What happens when the distributions move far away from each other?

[Interactive output: available in the online version of this chapter.]

6.4 Maximum Mean Discrepancy (MMD)

MMD measures the distance between mean embeddings of distributions in a reproducing kernel Hilbert space (RKHS). With a kernel k ,

$$\text{MMD}^2(p, q) = \mathbb{E}_{x, x' \sim p}[k(x, x')] - 2 \mathbb{E}_{x \sim p, y \sim q}[k(x, y)] + \mathbb{E}_{y, y' \sim q}[k(y, y')].$$

We will use the radial basis function (RBF) kernel

$$k_\sigma(x, y) = \exp\left(-\frac{(x-y)^2}{2\sigma_k^2}\right),$$

which assigns high similarity when two samples are closer than the bandwidth σ_k and decays smoothly otherwise. A small σ_k emphasises very local discrepancies—only nearby points contribute—so MMD becomes sensitive to fine-grained differences but can miss global shifts. A large σ_k blurs the notion of neighbourhood, making the kernel respond to broader trends while down-weighting local wiggles. Balancing σ_k therefore tunes whether we care more about microstructure or coarse alignment between p and q .

```
# Shared noise tensors for deterministic MMD estimates
BASE_NOISE_NORMAL = torch.randn(4096, dtype=DEFAULT_DTYPE, device=device)
BASE_NOISE_UNIFORM = torch.linspace(0.0, 1.0, 4096, dtype=DEFAULT_DTYPE, device=device)

def sample_with_matching_noise(spec: DistributionSpec, sample_count: int) -> torch.Tensor:
    if "Uniform" in spec.name:
        base = BASE_NOISE_UNIFORM[:sample_count]
    else:
        base = BASE_NOISE_NORMAL[:sample_count]
    return spec.sampler(sample_count, base)

def rbf_kernel(x: torch.Tensor, y: torch.Tensor, bandwidth: float) -> torch.Tensor:
    bw = torch.as_tensor(bandwidth, dtype=DEFAULT_DTYPE, device=device)
    diff = x[:, None] - y[None, :]
    return torch.exp(-0.5 * diff.pow(2) / (bw ** 2 + EPS))

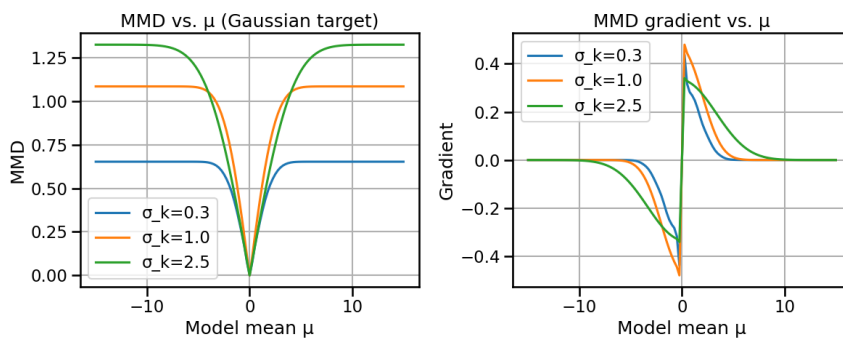
def maximum_mean_discrepancy(
    mu: torch.Tensor,
    target: DistributionSpec,
    model_std: float = 1.0,
    bandwidth: float = 1.0,
    sample_count: int = 512,
```



```

) -> torch.Tensor:
    samples_p = sample_with_matching_noise(target, sample_count)
    base_gauss = BASE_NOISE_NORMAL[:sample_count]
    samples_q = mu + model_std * base_gauss
    xx = rbf_kernel(samples_p, samples_p, bandwidth)
    yy = rbf_kernel(samples_q, samples_q, bandwidth)
    xy = rbf_kernel(samples_p, samples_q, bandwidth)
    xx = xx - torch.diag(torch.diag(xx))
    yy = yy - torch.diag(torch.diag(yy))
    n = sample_count
    mmd2 = xx.sum() / (n * (n - 1) + EPS) - 2.0 * xy.mean() + yy.sum() / (n * (n - 1) + EPS)
    return torch.sqrt(torch.clamp(mmd2, min=0.0))

```



💡 Experiment: Effect of MMD Kernel Bandwidth

- Which bandwidth reacts most strongly to small shifts in μ ?
- Observe how the gradient flattens for large σ_k . When might that be desirable?
- How would you select σ_k automatically in a learning system?

[Interactive output: available in the online version of this chapter.]

6.5 Optimal Transport (Wasserstein-2 Distance)

Optimal Transport (OT) frames distribution comparison as the problem of moving probability mass from a source distribution q to a target distribution p at minimal cost. Given a ground cost $c(x, y)$ measuring how expensive it is to transport mass from x to y , the quadratic OT problem seeks a coupling $\pi(x, y)$ with marginals q and p that minimises the total transport

cost:

$$W_2^2(p, q) = \inf_{\pi \in \Pi(p, q)} \int_{\mathbb{R}^d \times \mathbb{R}^d} \|x - y\|^2 d\pi(x, y),$$

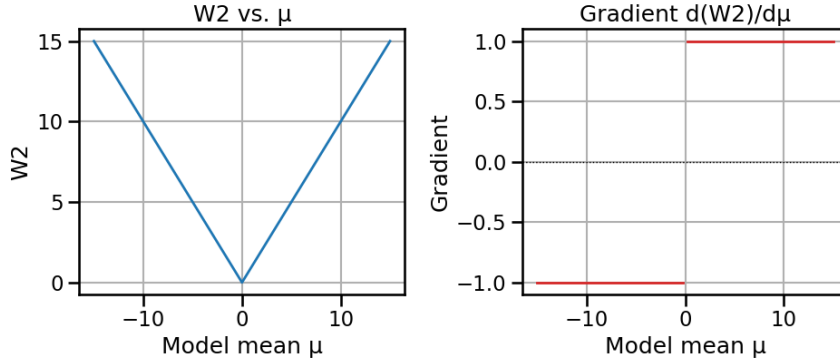
where $\Pi(p, q)$ denotes the set of joint distributions whose marginals are p and q . This formulation differs from divergence-based distances above (KL, JS, MMD) because it explicitly models *where* probability mass must move, rather than comparing densities pointwise or via kernel similarities. As a result, OT remains informative even when supports do not overlap: the metric still reflects how far mass must travel to align the distributions.

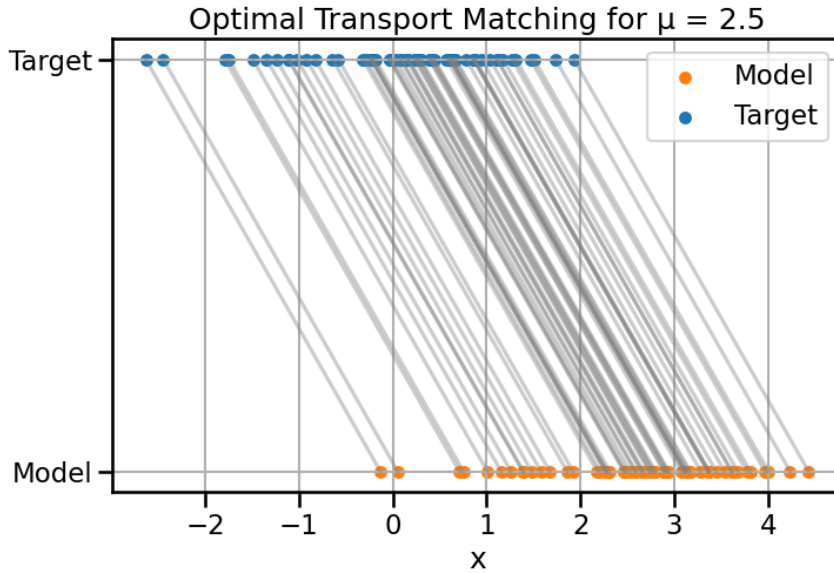
In one dimension, the optimal coupling sorts both distributions and pairs quantiles. The squared Wasserstein-2 distance simplifies to

$$W_2^2(p, q) = \int_0^1 |F_p^{-1}(u) - F_q^{-1}(u)|^2 du,$$

with F^{-1} denoting the quantile function. For Gaussians, this has a closed form $W_2^2 = (\mu_p - \mu)^2 + (\sigma_p - \sigma_q)^2$. We will compute it via quantile functions to keep the numerical pipeline consistent for non-Gaussian targets.

W2 against Gaussian($\mu=0.00$, $\sigma=1.00$)





💡 Reflection: Transport as Matching

- How does the transport map change if you halve the model variance?
- Compare the length of the transport lines with the Wasserstein distance you computed earlier.
- What would happen if the target distribution had gaps (e.g., a mixture)? Sketch the expected matching pattern.

[Interactive output: available in the online version of this chapter.]

6.6 Information Geometry vs. Optimal Transport

Information Geometry treats families of probability distributions as curved manifolds equipped with the Fisher information metric. Intuitively, it asks: *how sensitive are likelihoods to infinitesimal parameter changes?* The geometry that emerges is tailored to statistical inference—geodesics correspond to paths that keep models maximally informative, and inner products reflect the Cramér–Rao notion of efficiency. This is why information-geometric tools appear in natural gradient descent, variational inference, and sensor placement problems where we care about how much information parameters carry about data.

Because the Fisher–Rao metric lives on the parameter manifold, it is invariant to reparameterisations and emphasises directions where the distribution changes *statistically* rather than

spatially. In contrast, Optimal Transport metrics such as W_2 operate directly on the sample space and quantify how far probability mass must move. OT is therefore ideal when spatial structure matters—think of matching shapes, aligning time-series histograms, or enforcing smooth transport maps in PDE-constrained problems.

When would you pick one over the other? - Use Information Geometry when optimizing probabilistic models with latent variables or exponential-family structure, where natural gradients provide preconditioning aligned with likelihood curvature. - Use OT when discrepancies in physical space are key, such as calibrating simulators to sensor data, training generative models that must respect spatial coherence, or comparing distributions with disjoint supports. It is also useful when the target distributions do not have an analytical form but samples are available.

Let us compare the two on Gaussian families and visualize the geodesic paths they induce.

[Interactive output: available in the online version of this chapter.]

Experiment: Comparison of OT versus Fisher–Rao

- OT interpolates linearly in the mean; Fisher–Rao bends the path. How might that influence optimization trajectories?
- Which metric is more sensitive to changes in variance vs. mean in this example?
- If you needed to regularize a generative model toward a reference Gaussian, which geometry would you prefer and why?

6.7 Wrap-Up

We inspected five prominent distribution distances, paying attention to the gradients they induce for a simple Gaussian parameter. Key takeaways: - KL divergences emphasize support mismatch and can be asymmetric in how they penalize missing mass. - JS divergence softens gradients and stays bounded, making it attractive for adversarial training. - MMD exposes the role of kernel bandwidth in shaping sensitivity to local vs. global structure. - Wasserstein distance reasons about mass transport and yields interpretable gradient flows. - Fisher–Rao geometry offers a complementary, information-theoretic notion of proximity with different geodesics.

7 Introduction to Inference

Learning Objectives

By the end of this notebook, you will:

1. Understand **Bayesian inference for regression** and how it quantifies model uncertainty
2. Visualize the **posterior predictive distribution** for linear models
3. Recognize the **limitations of linear models** on nonlinear data
4. Learn how **neural networks** provide flexible nonlinear models
5. Understand why exact Bayesian inference becomes **intractable** for neural networks
6. Implement **Variational Inference (VI)** as an approximate solution
7. Compare VI to MCMC and understand the **speed vs. accuracy tradeoff**
8. Visualize **predictive uncertainty** in function space

The Big Picture

Imagine you have sparse sensor measurements from a mechanical system. You want to:

1. **Fit a model** that captures the underlying relationship
2. **Quantify uncertainty** in predictions (critical for safety-critical systems)
3. **Make predictions** with confidence intervals

This notebook shows you how to do this using **Bayesian inference**, starting with simple linear models and progressing to flexible neural networks where **variational inference** becomes essential.

7.1 Motivation and Toy Problem Definition

Let's start with a fundamental question: **when you fit a model to data, how confident should you be in its predictions?**

In traditional machine learning, we find a single “best” set of parameters. But in engineering, we often need to know: - How uncertain are we about the model? - Where in the input space are predictions reliable? - What happens if we had slightly different data?

Bayesian inference answers these questions by maintaining a **distribution over models** rather than picking a single one.

7.1.1 The Dataset: Nonlinear Sensor Measurements

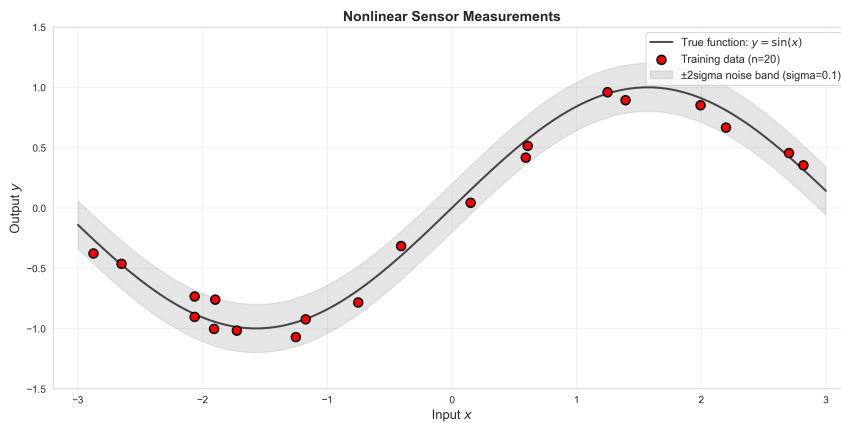
We'll use a simple nonlinear function to represent sensor data:

$$y = \sin(x) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 0.1^2)$$

This represents, for example: - Cyclic behavior (vibrations, temperatures, etc.) - Noisy measurements - Sparse observations (we won't measure everywhere)

We will be interested in understanding how the model can capture the uncertainty of its prediction.

<torch._C.Generator at 0x280584e30f0>



7.2 Linear Bayesian Regression

Let's start with the simplest approach: **linear regression with Bayesian inference**.

7.2.1 The Model

We assume a linear relationship:

$$y = w_0 + w_1x + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

In Bayesian inference, we place a **prior** over the weights:

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$$

After seeing data (X, Y) , the **posterior** over weights is:

$$p(\mathbf{w}|X, Y) = \mathcal{N}(\mathbf{m}_N, \mathbf{S}_N)$$

where (for linear-Gaussian models, we can derive it exactly):

$$\mathbf{S}_N = (\alpha \mathbf{I} + \beta \Phi^T \Phi)^{-1}$$

$$\mathbf{m}_N = \beta \mathbf{S}_N \Phi^T \mathbf{y}$$

where $\beta = 1/\sigma^2$ is the noise precision and Φ is the design matrix.

7.2.2 The Posterior Predictive Distribution

For a new input x_* , the **predictive distribution** integrates over all possible weights:

$$p(y_*|x_*, X, Y) = \int p(y_*|x_*, \mathbf{w}) p(\mathbf{w}|X, Y) d\mathbf{w}$$

For linear-Gaussian models, this is also Gaussian:

$$p(y_*|x_*, X, Y) = \mathcal{N}(\mathbf{m}_N^T \phi(x_*), \sigma_N^2(x_*))$$

The variance $\sigma_N^2(x_*)$ tells us **how uncertain** we are about predictions at x_* .

Let's implement this and see what happens when applied to our non-linear data.

```
class BayesianLinearRegression:
    """
    Bayesian linear regression with Gaussian prior and likelihood.

    Closed-form posterior over weights for polynomial basis functions.
    """

    def __init__(self, degree: int = 1, alpha: float = 1.0, beta: float = 100.0):
        """
        Args:
            degree: Degree of polynomial basis (1 = linear)
            alpha: Prior precision (inverse variance)
            beta: Noise precision (1/sigma^2)
        """
        self.degree = degree
        self.alpha = alpha
```

```

self.beta = beta
self.m_N = None # Posterior mean
self.S_N = None # Posterior covariance

def _design_matrix(self, x: np.ndarray) -> np.ndarray:
    """Compute polynomial design matrix."""
    x = x.reshape(-1, 1)
    Phi = np.concatenate([x**i for i in range(self.degree + 1)], axis=1)
    return Phi

def fit(self, X: np.ndarray, y: np.ndarray):
    """Compute posterior over weights given training data."""
    Phi = self._design_matrix(X)
    n_features = Phi.shape[1]

    # Posterior covariance
    self.S_N = np.linalg.inv(self.alpha * np.eye(n_features) +
                             self.beta * Phi.T @ Phi)

    # Posterior mean
    self.m_N = self.beta * self.S_N @ Phi.T @ y

    return self

def predict(self, X_test: np.ndarray, return_std: bool = True):
    """
    Posterior predictive distribution.

    Returns:
        mean: Predictive mean
        std: Predictive standard deviation (if return_std=True)
    """
    Phi_test = self._design_matrix(X_test)

    # Predictive mean
    y_mean = Phi_test @ self.m_N

    if return_std:
        # Predictive variance (includes noise and weight uncertainty)
        y_var = 1.0/self.beta + np.sum(Phi_test @ self.S_N * Phi_test, axis=1)
        y_std = np.sqrt(y_var)
        return y_mean, y_std

```



```

    return y_mean

def sample_weights(self, n_samples: int = 100) -> np.ndarray:
    """Sample weight vectors from posterior."""
    return np.random.multivariate_normal(self.m_N, self.S_N, size=n_samples)

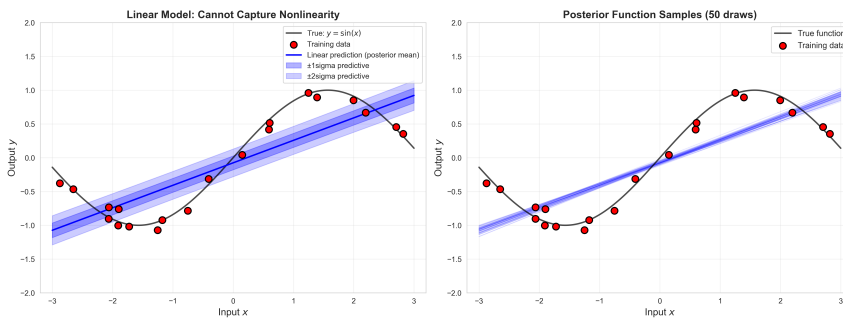
# Fit linear Bayesian regression
blr = BayesianLinearRegression(degree=1, alpha=1.0, beta=1.0/noise_std**2)
blr.fit(x_train, y_train)

# Posterior predictive distribution
y_pred_mean, y_pred_std = blr.predict(x_test)

print("Linear Bayesian Regression Fitted!")
print(f"Posterior mean weights: {blr.m_N}")
print(f"Posterior std of weights: {np.sqrt(np.diag(blr.S_N))}")

```

Linear Bayesian Regression Fitted!
 Posterior mean weights: [-0.07553835 0.33272507]
 Posterior std of weights: [0.0225762 0.01242588]



7.3 A Small Neural Network (Nonlinear Model)

The linear model fails because the true function is nonlinear. What if we try to use a simple **neural network** instead?

7.3.1 Architecture

We'll use a simple 1-hidden-layer network:

$$f(x; \mathbf{w}) = \mathbf{w}_2^T \sigma(\mathbf{w}_1 x + \mathbf{b}_1) + b_2$$

where σ is ReLU activation. With just 5 hidden units, this can approximate smooth nonlinear functions.

7.3.2 The New Challenge: Intractable Posterior

Unlike linear regression, the posterior $p(\mathbf{w}|X, Y)$ is **no longer Gaussian!** We cannot compute it analytically.

However, we can still: 1. Define a **prior** $p(\mathbf{w})$ (e.g., Gaussian on all weights) 2. Sample functions from the prior to see what kinds of functions are admissible under the model 3. Later, we can use **VI or MCMC** to approximate the posterior

Let's first see what the **prior predictive distribution** looks like.

```
class SmallNN(nn.Module):
    """
    Simple 1-hidden-layer neural network for regression.
    """

    def __init__(self, hidden_size: int = 5):
        super().__init__()
        self.fc1 = nn.Linear(1, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

    def forward(self, x):
        x = x.view(-1, 1)
        h = F.relu(self.fc1(x))
        y = self.fc2(h)
        return y.squeeze()

    def get_weights_flat(self) -> torch.Tensor:
        """Get all weights as a flat vector."""
        return torch.cat([p.flatten() for p in self.parameters()])

    def set_weights_flat(self, w_flat: torch.Tensor):
        """Set all weights from a flat vector."""
        offset = 0
```

```

    for p in self.parameters():
        n_params = p.numel()
        p.data = w_flat[offset:offset+n_params].view(p.shape)
        offset += n_params

# Create network and count parameters
hidden_size = 5
model = SmallNN(hidden_size=hidden_size)
n_params = sum(p.numel() for p in model.parameters())

print("Neural Network Architecture:")
print(f" Input → {hidden_size} hidden (ReLU) → 1 output")
print(f" Total parameters: {n_params}")
print("Parameter breakdown:")
for name, param in model.named_parameters():
    print(f" {name}: shape {param.shape}, count {param.numel()}")

```

Neural Network Architecture:

Input → 5 hidden (ReLU) → 1 output

Total parameters: 16

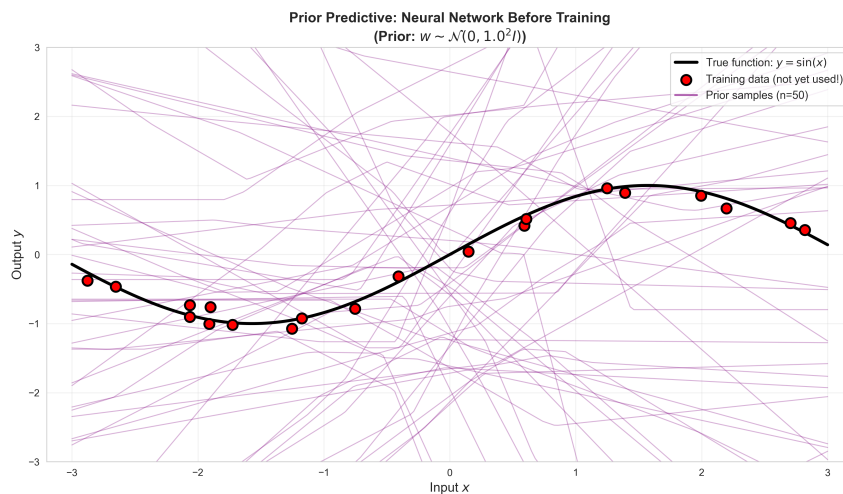
Parameter breakdown:

fc1.weight: shape torch.Size([5, 1]), count 5

fc1.bias: shape torch.Size([5]), count 5

fc2.weight: shape torch.Size([1, 5]), count 5

fc2.bias: shape torch.Size([1]), count 1



We can see that just sampling functions from the prior, the possible Neural Network functions are a lot more diverse than those we could sample from the linear model. But how do we find out which of these weights are (probabilistically) likely?

7.4 MCMC Approximation

The neural network posterior $p(\mathbf{w}|X, Y)$ is intractable, but we can **sample** from it using **Markov Chain Monte Carlo (MCMC)**. Before we dive into how to apply this to neural networks, let's first gain some intuition on how and why MCMC works on a simple 2D example.

7.4.1 Why can't we just compute the posterior using regular Monte Carlo Integration?

A classical Monte Carlo approach would involve: 1. Drawing many samples $\mathbf{w}^{(i)}$ from the prior $p(\mathbf{w})$, or (as is more common) from a proposal distribution $q(\mathbf{w})$ that we hope covers the posterior well. 2. Weighting each sample by its likelihood $p(Y|X, \mathbf{w}^{(i)})$ 3. Normalizing empirically to get an approximate posterior.

This is called **importance sampling** and while it is conceptually simple, it doesn't work for many complex models or distributions because the proposal distribution needs to be well matched to the true posterior. If the posterior is concentrated in a small region of parameter space, most samples will have negligible weight, and this will make our "Effective Sample Size" (ESS) very small, i.e., we will need to do lots of sampling, but ultimately throw out most of those samples, so this is inefficient. For low-dimensional, fairly simple distributions, importance sampling can work well, but for high-dimensional models like neural networks or very complex posterior distributions, it typically fails (as we will see below).

To address this, we can use Markov Chain Monte Carlo (MCMC) methods, which adaptively explore the posterior distribution. The key idea is to construct a Markov chain whose stationary distribution is the target posterior distribution. By running the chain for a long time, we can obtain samples that are approximately distributed according to the posterior. Its main disadvantage is that because many of the samples are correlated, we need many more samples to get a good approximation of the posterior compared to independent sampling methods, but this downside is offset by the fact that MCMC can explore more complex distributions more effectively even using a simple proposal distribution.

7.4.2 What is MCMC?

MCMC methods (like Metropolis-Hastings, Hamiltonian Monte Carlo, and others) generate samples $\mathbf{w}^{(1)}, \mathbf{w}^{(2)}, \dots$ from the posterior by: 1. Starting at a random initial weight vector 2.

Proposing updated samples that respect the posterior density 3. Eventually converging to samples from $p(\mathbf{w}|X, Y)$

So, in a nutshell, it allows you to generate plausible samples from $p(\mathbf{w}|X, Y)$ without ever having to strictly compute the posterior.

Pros: Asymptotically exact (given infinite samples)

Cons: Slow, especially for high-dimensional models

For our small network, MCMC is feasible. We will use it as a reference to compare against variational inference at the end of this notebook.

7.4.3 Note on Implementation

For this notebook, we'll use a simple **Metropolis-Hastings** sampler, although many more advanced algorithms exist.¹

The Metropolis-Hastings Algorithm

The algorithm works as follows:

1. **Initialize:** Start with random weights $\mathbf{w}^{(0)}$
2. **Propose:** Generate a candidate $\mathbf{w}^* \sim q(\mathbf{w}^*|\mathbf{w}^{(t)})$ (typically a Gaussian centered at current weights)
3. **Accept/Reject:** Compute the acceptance probability:

$$\alpha = \min \left(1, \frac{p(\mathbf{w}^*|X, Y)}{p(\mathbf{w}^{(t)}|X, Y)} \right) = \min \left(1, \frac{p(Y|X, \mathbf{w}^*)p(\mathbf{w}^*)}{p(Y|X, \mathbf{w}^{(t)})p(\mathbf{w}^{(t)})} \right)$$

Accept $\mathbf{w}^{(t+1)} = \mathbf{w}^*$ with probability α , otherwise keep $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)}$

4. **Repeat:** Continue for many iterations until the chain converges

The useful thing about this sampler (and indeed MCMC approaches in general), is that we only ever need to evaluate the **unnormalized** posterior (likelihood x prior), and thus can avoid computing the intractable normalizing constant.

Key Caveats (with Metropolis-Hastings or MCMC methods in general): - You cannot just take individual samples and treat them as independent draws from the posterior, since the starting location of one sample was dependent on the location of the last sample. This means that the samples are correlated. In practice, this means you will need to discard

¹In practice, you'd use more sophisticated methods like: Hamiltonian Monte Carlo (HMC), the No-U-Turn Sampler (NUTS) or Stochastic Gradient Langevin Dynamics (SGLD), but these are beyond the scope of this notebook. For an interactive example of samplers, I highly recommend checking out the [MCMC Interactive Gallery](#)

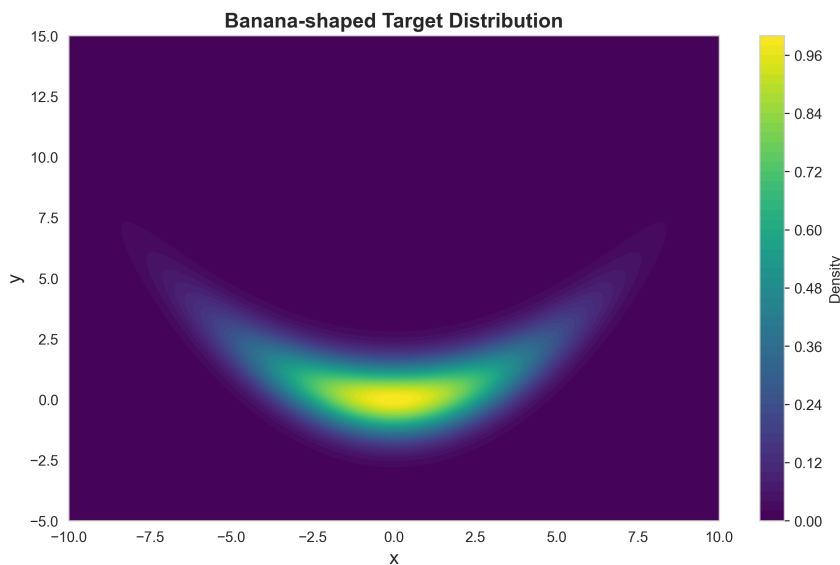
the first certain number of samples (this is called “Burn In”) and also consider only keeping every K number of samples (this is called “Thinning”) so that the MCMC samples become less correlated with one another and more like true independent samples from the posterior. - While the proposal distribution can be simple (e.g., Gaussian), its parameters (like standard deviation) need to be carefully tuned since if it is too large we will have low sample acceptance rates, but if it is too small we will explore the space very slowly, requiring many more samples to converge.

For the 2D example we will do next, MCMC works very well and will give us a good approximation of the posterior. For our 16-parameter network that we will return to later in the notebook, MCMC sampling is still tractable. For modern deep networks with millions of parameters, however, MCMC becomes prohibitively slow and this is where Variational Inference can come to the rescue, as we will see later in the second part of the notebook.

7.4.4 Toy Example: 2D Posterior Sampling from the Banana Distribution

We will start with a simple 2D example to illustrate how MCMC works. This will come from the famous “Banana Distribution”, which is a common test case for MCMC methods due to its curved, non-Gaussian shape, as we can see below.

Defined banana target log-density and Gaussian proposal.



We can attempt to do Monte Carlo integration on this distribution, but because of its curved shape, simple importance sampling is unlikely to cover this distribution well, as we can see below:

```

# Importance Sampling on the banana target
N_is = 10_000

# Draw from q ~ N(0, I)
x_is = rng.normal(size=N_is)*.3
y_is = rng.normal(size=N_is)*.3

# Unnormalized log weights: log w_i = log p~(x_i,y_i) - log q(x_i,y_i)
log_w = log_p_tilde(x_is, y_is) - log_q(x_is, y_is)
# Stabilize weights by subtracting max log_w
log_w_shift = log_w - np.max(log_w)
w = np.exp(log_w_shift)
w_norm = w / np.sum(w)

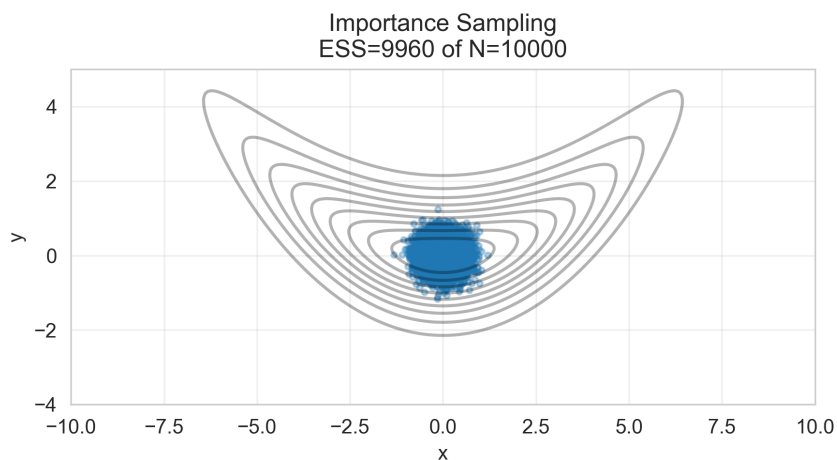
# Effective sample size
ESS = 1.0 / np.sum(w_norm**2)

# Weighted means
mean_x_is = np.sum(w_norm * x_is)
mean_y_is = np.sum(w_norm * y_is)

print(f"Importance Sampling: N={N_is}, ESS={ESS:.1f} ({100*ESS/N_is:.1f}% of N)")
print(f"Weighted means: E[x]={mean_x_is:.3f}, E[y]={mean_y_is:.3f}")

```

Importance Sampling: N=10000, ESS=9959.7 (99.6% of N)
Weighted means: E[x]=-0.003, E[y]=0.007



💡 Experiment: Modifying the Importance Sampling Proposal Distribution

- What happens if you change the proposal distribution to have a larger or smaller variance? Does this improve the coverage of the posterior?
- Compare the Effective Sample Size (ESS) for different proposal variances – does this align with your intuition when you compare the coverage of the sample points with the posterior?

Now that we have seen the limitations of importance sampling, let's see how MCMC can help us better sample from this complex posterior. We will implement Metropolis-Hastings below:

```
# Metropolis-Hastings MCMC on the banana target
def mcmc_banana(N=10_000, sigma=0.5, burn_in=1_000, start=(0.0, 0.0), seed=42):
    """Run a simple MH sampler with isotropic Gaussian proposals."""
    rng_local = np.random.default_rng(seed)
    x_curr, y_curr = float(start[0]), float(start[1])
    logp_curr = log_p_tilde(x_curr, y_curr)

    samples = np.zeros((N, 2), dtype=float)
    accepts = 0

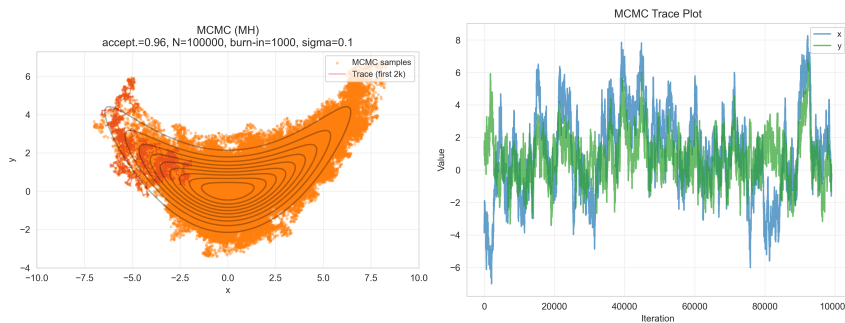
    for i in range(N):
        # Propose from q(x'|x) = N(x, sigma^2 I)
        dx, dy = rng_local.normal(scale=sigma, size=2)
        x_prop, y_prop = x_curr + dx, y_curr + dy
        logp_prop = log_p_tilde(x_prop, y_prop)

        # Symmetric proposal -> Hastings ratio reduces to p~(x')/p~(x)
        log_alpha = logp_prop - logp_curr
        if np.log(rng_local.uniform()) < min(0.0, log_alpha):
            x_curr, y_curr = x_prop, y_prop
            logp_curr = logp_prop
            accepts += 1
        samples[i] = (x_curr, y_curr)

    acc_rate = accepts / N
    chain = samples[burn_in:]
    mean_x_mcmc = float(np.mean(chain[:, 0]))
    mean_y_mcmc = float(np.mean(chain[:, 1]))
    return chain, acc_rate, mean_x_mcmc, mean_y_mcmc
```


MCMC: $N=100000$, burn-in=1000, $\sigma=0.1$, acceptance=0.96

MCMC means: $E[x]=1.061$, $E[y]=0.729$



Summary diagnostics:

Importance Sampling: $ESS=9959.7/10000$ (99.6%), $E[x]=-0.003$, $E[y]=0.007$

MCMC: acceptance=0.96, $E[x]=1.061$, $E[y]=0.729$

💡 Experiment: Metropolis-Hastings sampling parameters

Modify the number of samples, proposal variance, burn-in period, and the starting point.

- What happens if the proposal variance is very small? Is the chain able to explore the posterior well? How is this behavior reflected in the acceptance rate or the trace plot?
- What happens if the starting point is very far from the high-density region of the posterior? How long does it take for the chain to converge to the target distribution? What does this imply for the burn-in period and how is this reflected in the trace plot?
- How do the various factors (proposal variance, starting point, burn-in) interact to affect the acceptance rate and the quality of the samples? Is a higher or lower acceptance rate always better? Why do you think this is?

7.4.5 Why IS struggles and MCMC succeeds

- Importance Sampling (IS) relies on a global reweighting of proposal samples. When q poorly overlaps with the curved high-density region of the banana target, almost all weights are near zero. This leads to a tiny effective sample size (ESS) and high estimator variance.
- MCMC with local proposals uses acceptance ratios that only depend on pairwise density ratios. Even with an unnormalized target $\tilde{p}$, the normalizing constant cancels and the chain can move along the curved ridge, accumulating many effective samples.

- Both methods can use unnormalized p ; the practical difference is normalization: IS requires a global normalization of weights, while MCMC only needs local ratios. As dimensionality and curvature increase, IS becomes very brittle unless q is very well matched to p , whereas MCMC typically remains viable with reasonable proposals.

7.4.6 Returning to Neural Networks (now with MCMC sampling)

Now that we have a good understanding of how MCMC works on a simple 2D example, let's return to our small neural network and use MCMC to sample from its posterior distribution over weights. This will allow us to approximate the posterior predictive distribution for our nonlinear regression problem. We will add on a few tricks that we didn't have in our simple 2D example, namely: - We'll thin out samples in the chain to reduce correlation between samples - We'll discard a burn-in period to allow the chain to converge before collecting samples (as we did before) - Before starting the MCMC chain, we'll pretrain the network using MAP estimation to find a good starting point for the chain.

```
# Simple MCMC sampler for neural network weights
def log_likelihood(model, x, y, noise_std=0.1):
    """Compute log p(y|x,w) for current weights."""
    with torch.no_grad():
        x_tensor = torch.FloatTensor(x)
        y_pred = model(x_tensor).numpy()
        log_lik = -0.5 * np.sum(((y - y_pred) / noise_std) ** 2)
        log_lik -= len(y) * np.log(noise_std * np.sqrt(2 * np.pi))
    return log_lik

def log_prior(model, prior_std=1.0):
    """Compute log p(w) for current weights."""
    log_p = 0
    for param in model.parameters():
        log_p -= 0.5 * torch.sum((param / prior_std) ** 2).item()
    return log_p

def metropolis_hastings_nn(x_train, y_train, n_samples=5000, proposal_std=0.01,
                           prior_std=1.0, noise_std=0.1, thin=10):
    """
    Simple Metropolis-Hastings MCMC for neural network weights.

    Args:
        thin: Only keep every 'thin'-th sample to reduce autocorrelation

    Returns:
```

```

weight_samples: List of weight vectors
acceptance_rate: Fraction of proposals accepted
"""
model = SmallNN(hidden_size=5)

# Initialize at MAP estimate (rough approximation: just train briefly)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
x_tensor = torch.FloatTensor(x_train).view(-1, 1)
y_tensor = torch.FloatTensor(y_train)

print("Finding initial weights (quick MAP estimate)...")
for _ in range(200):
    optimizer.zero_grad()
    loss = F.mse_loss(model(x_tensor.squeeze()), y_tensor)
    loss.backward()
    optimizer.step()

print(f"Starting MCMC sampling ({n_samples} iterations)...")

# Current state
current_log_posterior = log_likelihood(model, x_train, y_train, noise_std) + \
    log_prior(model, prior_std)

weight_samples = []
n_accepted = 0

for i in range(n_samples):
    # Propose new weights
    proposed_model = SmallNN(hidden_size=5)
    with torch.no_grad():
        for p_curr, p_prop in zip(model.parameters(), proposed_model.parameters()):
            p_prop.data = p_curr.data + proposal_std * torch.randn_like(p_curr)

    # Compute proposed log posterior
    proposed_log_posterior = log_likelihood(proposed_model, x_train, y_train, noise_std) + \
        log_prior(proposed_model, prior_std)

    # Accept/reject
    log_alpha = proposed_log_posterior - current_log_posterior
    if np.log(np.random.rand()) < log_alpha:
        # Accept
        model = proposed_model

```

```

        current_log_posterior = proposed_log_posterior
        n_accepted += 1

    # Store sample (with thinning)
    if i % thin == 0:
        weight_samples.append(model.get_weights_flat().clone().detach().numpy())

    if (i + 1) % 2000 == 0:
        acc_rate = n_accepted / (i + 1)
        print(f"  Iteration {i+1}/{n_samples}, acceptance rate: {acc_rate:.3f}")

    acceptance_rate = n_accepted / n_samples
    print(f"MCMC complete! Final acceptance rate: {acceptance_rate:.3f}")

    return weight_samples, acceptance_rate

# Run MCMC (this will take a minute or two)
print("Running MCMC to sample from posterior p(w|X,Y)...")
print("This may take 1-2 minutes for accurate results.")
mcmc_samples, acc_rate = metropolis_hastings_nn(
    x_train, y_train,
    n_samples=10_000,
    proposal_std=0.01,
    prior_std=1.0,
    noise_std=noise_std,
    thin=10
)

print(f"Collected {len(mcmc_samples)} posterior samples (after thinning)")
print(f"Acceptance rate: {acc_rate:.1%}")

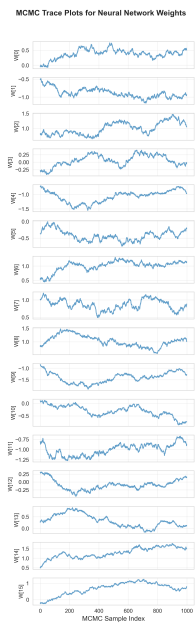
```

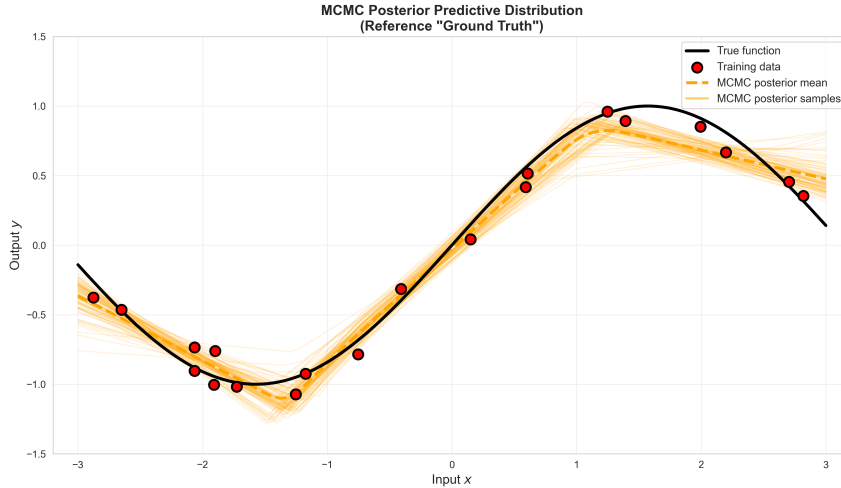
```

Running MCMC to sample from posterior p(w|X,Y)...
This may take 1-2 minutes for accurate results.
Finding initial weights (quick MAP estimate)...
Starting MCMC sampling (10000 iterations)...
Finding initial weights (quick MAP estimate)...
Starting MCMC sampling (10000 iterations)...
  Iteration 2000/10000, acceptance rate: 0.460
  Iteration 2000/10000, acceptance rate: 0.460
  Iteration 4000/10000, acceptance rate: 0.458
  Iteration 4000/10000, acceptance rate: 0.458
  Iteration 6000/10000, acceptance rate: 0.462

```

Iteration 6000/10000, acceptance rate: 0.462
 Iteration 8000/10000, acceptance rate: 0.466
 Iteration 8000/10000, acceptance rate: 0.466
 Iteration 10000/10000, acceptance rate: 0.455
 MCMC complete! Final acceptance rate: 0.455
 Collected 1000 posterior samples (after thinning)
 Acceptance rate: 45.5%
 Iteration 10000/10000, acceptance rate: 0.455
 MCMC complete! Final acceptance rate: 0.455
 Collected 1000 posterior samples (after thinning)
 Acceptance rate: 45.5%





7.4.7 Advanced Sampling: Hamiltonian Monte Carlo (HMC)

There have been many substantial improvements to MCMC sampling methods over the past several decades, and one of the main advances came through the recognition that we need not only sample with knowledge of q and p , but that we could leverage the *gradients* of p . This was previously very limited or difficult to do, but with the widespread use of Automatic Differentiation libraries became much more powerful.

Upon widespread access to gradients of p , we could now think about sampling processes that could leverage the *slope* and *curvature* of the likelihood to guide us toward better samples. One of the mainstream ways to do this is **Hamiltonian Monte Carlo** which treats sampling as analogous to estimating a dynamical system. Specifically, we treat the samples/parameters as positions, introduce an auxiliary momentum variable, and simulate Hamiltonian dynamics that preserve the joint density, typically using a symplectic integrator, such as the leapfrog integrator. Because the leapfrog integrator follows smooth trajectories, HMC avoids the random walk behaviour that slowed down our Metropolis–Hastings chains above. HMC is still fundamentally accepting or rejecting samples like MH did, except that it's method of producing new samples is significantly more likely to lead to acceptable samples, and thus can be far more sample efficient.

It does this through a few basic concepts: - It uses the likelihood as a Potential Energy function $U(\theta) = -\log p(\theta \mid \text{data})$ which pulls particle trajectories toward higher posterior density. - Each particle has Kinetic Energy $K(\mathbf{p}) = \frac{1}{2}\mathbf{p}^\top M^{-1}\mathbf{p}$ which comes from a tunable mass matrix M that can re-scale parameters. - A symplectic integrator (Leapfrog integration) alternates half-steps in momentum with full steps in position and nearly conserves energy.

The algorithm itself is fairly straightforward in that you:

1. Sample a fresh momentum from $\mathcal{N}(0, M)$ to randomize the next trajectory.

2. Integrate the Hamiltonian dynamics for L leapfrog steps with step size ε .
3. Apply a Metropolis accept/reject step using the Hamiltonian.

Some peculiarities with HMC include: - Smaller step size ε improves accuracy but increases computation. If step sizes are too large, it can overshoot and causes many sample rejections. - The number of leapfrog steps L controls how far each proposal travels across the posterior ridge. Too small and the samples become auto-correlated since the sample cannot move far enough away from the current sample. - A well-chosen mass matrix and initial adaptation help HMC cope with areas of strong curvature in the posterior. - HMC relies on gradients, so we need target distributions that are differentiable and numerically stable.

Below we implement a minimal HMC sampler for the banana distribution so you can compare its behaviour with the random-walk MH sampler above.

```
# Hamiltonian Monte Carlo on the banana target
def grad_log_p_tilde(x: np.ndarray, y: np.ndarray) -> np.ndarray:
    """Compute the gradient of the banana log-density with respect to (x, y)."""
    x = np.asarray(x)
    y = np.asarray(y)
    grad_x = -x / 9.0 + 0.2 * x * (y - 0.1 * x**2)
    grad_y = -(y - 0.1 * x**2)
    return np.stack([grad_x, grad_y], axis=-1)

def hmc_banana(num_samples: int = 8_000, burn_in: int = 1_000, step_size: float = 0.25,
               num_leapfrog: int = 20, mass: np.ndarray | float = 1.0,
               start: tuple[float, float] = (0.0, 0.0), seed: int = 2024) -> tuple[np.ndarray, float]:
    """Run a basic HMC sampler with (scalar or diagonal) mass matrix for the banana target."""
    rng_local = np.random.default_rng(seed)
    q_current = np.array(start, dtype=float)
    if q_current.shape != (2,):
        raise ValueError("start must be a length-2 tuple or array.")

    mass_array = np.asarray(mass, dtype=float)
    if mass_array.ndim == 0:
        mass_vec = np.full(2, mass_array.item())
    elif mass_array.shape == (2,):
        mass_vec = mass_array
    else:
        raise ValueError("mass must be a scalar or length-2 array for this example.")
    if np.any(mass_vec <= 0):
        raise ValueError("mass components must be positive.")
    inv_mass = 1.0 / mass_vec
```

```

logp_current = log_p_tilde(q_current[0], q_current[1])
samples = np.zeros((num_samples, 2), dtype=float)
accepts = 0

for i in range(num_samples):
    # Sample fresh momentum from N(0, M) with diagonal mass matrix.
    p_current = rng_local.normal(scale=np.sqrt(mass_vec), size=2)
    q = q_current.copy()
    p = p_current.copy()

    # Leapfrog integration
    grad = grad_log_p_tilde(q[0], q[1])
    p += 0.5 * step_size * grad
    for lf_step in range(num_leapfrog):
        q += step_size * p * inv_mass
        grad = grad_log_p_tilde(q[0], q[1])
        if lf_step != num_leapfrog - 1:
            p += step_size * grad
    p += 0.5 * step_size * grad
    p = -p # Reverse momentum for detailed balance

    logp_prop = log_p_tilde(q[0], q[1])
    current_H = -logp_current + 0.5 * np.sum((p_current**2) * inv_mass)
    proposed_H = -logp_prop + 0.5 * np.sum((p**2) * inv_mass)
    log_alpha = current_H - proposed_H

    if np.log(rng_local.uniform()) < min(0.0, log_alpha):
        q_current = q
        logp_current = logp_prop
        accepts += 1

    samples[i] = q_current

if burn_in >= num_samples:
    raise ValueError("burn_in must be smaller than num_samples.")
chain = samples[burn_in:]
acc_rate = accepts / num_samples
return chain, acc_rate

# Run HMC and summarize diagnostics
N_hmc = 8_000
burn_in_hmc = 1_000

```



```

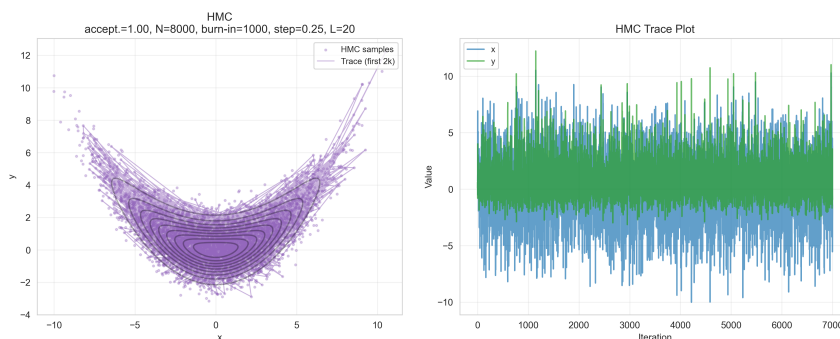
step_size = 0.25
num_leapfrog = 20
mass_vec = np.array([0.7, 1.3]) # Slight re-scaling along x and y
hmc_chain, hmc_acc_rate = hmc_banana(num_samples=N_hmc, burn_in=burn_in_hmc,
                                     step_size=step_size, num_leapfrog=num_leapfrog,
                                     mass=mass_vec, start=(10.0, 10.0), seed=11)

mean_x_hmc = float(np.mean(hmc_chain[:, 0]))
mean_y_hmc = float(np.mean(hmc_chain[:, 1]))

print(f"HMC: N={N_hmc}, burn-in={burn_in_hmc}, step size={step_size}, L={num_leapfrog}, acceptance={hmc_acc_rate}")
print(f"HMC means: E[x]={mean_x_hmc:.3f}, E[y]={mean_y_hmc:.3f}")

```

HMC: N=8000, burn-in=1000, step size=0.25, L=20, acceptance=1.00
HMC means: E[x]=0.046, E[y]=0.860



💡 Experiment: Tuning HMC Hyperparameters

- Decrease the step size while increasing the leapfrog count so the trajectory length stays similar. What happens to acceptance and exploration?
- Try a very large step size or very few leapfrog steps. Do you see the chain reverting to random-walk behaviour?
- Replace the scalar mass with a small diagonal matrix (e.g., different masses for x and y). Does this help the sampler hug the banana ridge more quickly?

7.5 Variational Inference

MCMC works but is slow, since it essentially has to generate thousands of good samples. In contrast, Variational Inference methods turn the **integration** problem into an **optimization** problem. Instead of sampling from $p(\mathbf{w}|X, Y)$, we:

1. Choose a tractable family $q_\phi(\mathbf{w})$ (e.g., Gaussian)
2. Find ϕ that makes q_ϕ close to the true posterior
3. Measure “closeness” using the **ELBO** (Evidence Lower Bound)

7.5.1 Choosing a tractable family $q_\phi(\mathbf{w})$

There are many options for this, as we will see in later more advanced notebooks, but for now let’s use something simple: a **mean-field Gaussian** approximation:

$$q_\phi(\mathbf{w}) = \mathcal{N}(\mu_\phi, \text{diag}(\sigma_\phi^2))$$

We will look at two examples below: the first is just a simple 2D Gaussian with a 2D mean vector and two σ s corresponding to a diagonal covariance. Later, we extend this concept to a Neural Network (like we did with MCMC above) each weight in the network will get its own mean and variance, and for our network with `n_params` parameters, we would then optimize both for all weights, leading to $2 \times n_{params}$ variational parameters.

But, in either case, what should we be optimizing, exactly?

7.5.2 1. Marginal Likelihood (Evidence)

The goal in Bayesian inference is to compute the **marginal likelihood** (also called the evidence):

$$p(Y|X) = \int p(Y|X, \mathbf{w}) p(\mathbf{w}) d\mathbf{w}$$

Which sounds fine in theory, but this integral is usually **intractable** for neural networks or other complex models we might be interested in. How can we solve this?

7.5.3 2. Introducing a Variational Distribution

We introduce a tractable distribution $q_\phi(\mathbf{w})$ to approximate the true posterior $p(\mathbf{w}|X, Y)$.

We can rewrite the log evidence using q_ϕ :

$$\begin{aligned} \log p(Y|X) &= \log \int p(Y|X, \mathbf{w}) p(\mathbf{w}) d\mathbf{w} \\ &= \log \int q_\phi(\mathbf{w}) \frac{p(Y|X, \mathbf{w}) p(\mathbf{w})}{q_\phi(\mathbf{w})} d\mathbf{w} \end{aligned}$$

So far, this mathematical maneuver hasn't really bought us much, but we can apply an approximation to help us separate out some terms.

7.5.4 3. Applying Jensen's Inequality

Jensen's Inequality states that, for a convex function f :

$$f(\mathbb{E}[x]) \leq \mathbb{E}[f(x)]$$

and if we apply Jensen's inequality to the above assuming $f = \log$ and $\mathbb{E} = \int$ (note we flip the sign of the inequality since $\log$ is concave, not convex), we get:

$$\log \mathbb{E}_{q_\phi} \left[\frac{p(Y|X, \mathbf{w}) p(\mathbf{w})}{q_\phi(\mathbf{w})} \right] \geq \mathbb{E}_{q_\phi} \left[\log \frac{p(Y|X, \mathbf{w}) p(\mathbf{w})}{q_\phi(\mathbf{w})} \right]$$

Now we can bring the log inside to help break out some of the terms and rearrange them.

7.5.5 4. The ELBO Expression

This gives us the **Evidence Lower Bound (ELBO)**:

$$\log p(Y|X) \geq \mathbb{E}_{q_\phi} [\log p(Y|X, \mathbf{w})] + \mathbb{E}_{q_\phi} [\log p(\mathbf{w})] - \mathbb{E}_{q_\phi} [\log q_\phi(\mathbf{w})]$$

Or, by grouping $\mathbb{E}_{q_\phi} [\log p(\mathbf{w})]$ and $\mathbb{E}_{q_\phi} [\log q_\phi(\mathbf{w})]$ into the (negative) KL Divergence:

$$\mathcal{L}(\phi) = \mathbb{E}_{q_\phi} [\log p(Y|X, \mathbf{w})] - \text{KL}(q_\phi(\mathbf{w}) \| p(\mathbf{w}))$$

We arrive at the common form of the ELBO, where:

- The **first term** encourages q_ϕ to place mass on weights that explain the data well. This is the **Expected log-likelihood** and measures How well q_ϕ explains the data.
- The **second term** (the **KL divergence**) regularizes q_ϕ to stay close to the prior.

This is the objective we will then optimize when we perform variational inference. We will see this first applied to a simple 2D example so that we can understand what is going on, and then later move to a more complex and higher dimension example of a Neural Network.

7.5.6 Example of using VI for the Banana Distribution

Let's use our concrete example from earlier, the Banana distribution, to illustrate how Variational Inference works. Here we will use a very simple proposal distribution—a Gaussian with a mean and diagonal covariance—which is often called the *mean-field Gaussian approximation* as $q_\phi(\mathbf{w})$. We will then optimize the ELBO to approximate the posterior. That is, we will optimize:

$$\begin{aligned}\mathcal{L}(\phi) &= \mathbb{E}_{q_\phi} [\log p(Y|X, \mathbf{w})] - \text{KL}(q_\phi(\mathbf{w}) \| p(\mathbf{w})) \\ &= \mathbb{E}_{q_\phi} [\log p(Y|X, \mathbf{w})] + \mathbb{E}_{q_\phi} [\log p(\mathbf{w})] - \mathbb{E}_{q_\phi} [\log q_\phi(\mathbf{w})] \\ &= \mathbb{E}_{q_\phi} [\log p(Y, w|X)] - \mathbb{E}_{q_\phi} [\log q_\phi(\mathbf{w})]\end{aligned}$$

The reason that we are only computing $\mathbb{E}_{q_\phi} [\log p(Y, w|X)]$ in this case is because we are using a simple toy distribution where we are computing the joint distribution $p(Y, w|X)$ directly, rather than having a likelihood and prior separately as we would in a more complex model like a neural network.

```
# Variational Inference on the banana target distribution with interactive visualization
rng_vi = np.random.default_rng(0)
LOG_2PI = math.log(2.0 * math.pi)

def mf_elbo(mu_vec: torch.Tensor, raw_diag: torch.Tensor, num_samples: int = None) -> torch.Tensor:
    """Analytic ELBO for a mean-field (diagonal) Gaussian q.

    Assumes raw_diag stores log(sigma) for each dimension.
    Computes E_q[log p] analytically for the banana target:
        log p(x,y) = -0.5 * ( x^2/9 + (y - 0.1 x^2)^2 )
    and uses the closed-form moments of a Gaussian.
    """
    # diagonal std and variances
    sigma = torch.exp(raw_diag)
    var = sigma**2

    mu_x, mu_y = mu_vec[0], mu_vec[1]
    var_x, var_y = var[0], var[1]

    # moments for x
    E_x2 = var_x + mu_x**2
    # Var(x^2) for Gaussian x ~ N(mu_x, var_x)
    Var_x2 = 2.0 * var_x**2 + 4.0 * mu_x**2 * var_x
```

```

# First term in the ELBO
# E_q[ (y - 0.1 x^2)^2 ] = Var(y) + (E[y] - 0.1 E[x^2])^2 + 0.01 Var(x^2)
E_y_term = var_y + (mu_y - 0.1 * E_x2) ** 2 + 0.01 * Var_x2

# Second term in the ELBO
# E_q[log p]
E_log_p = -0.5 * (E_x2 / 9.0 + E_y_term)

# Third term in the ELBO
# E_q[log q] for diagonal Gaussian: -0.5*(D + D*log(2π) + 2 * sum(log sigma))
D = mu_vec.numel()
sum_log_sigma = torch.sum(raw_diag)
E_log_q = -0.5 * (D + D * LOG_2PI + 2.0 * sum_log_sigma)

return E_log_p - E_log_q

def run_vi_optimization(num_steps: int = 800, lr: float = 0.08, record_points: int = 60):
    """Optimize the ELBO using the mean-field (diagonal) analytic ELBO (mf_elbo).

    Records intermediate means, covariances (diagonal), and ELBO values for visualization.
    """
    mu = torch.tensor([0.0, 0.0], requires_grad=True)
    raw_diag = torch.log(torch.ones(2) * 2.5)
    raw_diag.requires_grad_(True)

    optimizer = Adam([mu, raw_diag], lr=lr)
    record_every = 1
    history = {"steps": [], "mu": [], "cov": [], "elbo": []}

    for step in range(num_steps):
        optimizer.zero_grad()
        elbo = mf_elbo(mu, raw_diag)
        loss = -elbo
        loss.backward()
        optimizer.step()

        if (step % record_every == 0) or (step == num_steps - 1):
            # build diagonal cov_diagonal and covariance for recording/plotting
            cov_diagonal = np.diag(np.exp(raw_diag.detach().cpu().numpy()))
            cov_np = cov_diagonal @ cov_diagonal.T
            history["steps"].append(step)
            history["mu"].append(mu.detach().cpu().numpy())

```

```

        history["cov"].append(cov_np)
        history["elbo"].append(elbo.detach().cpu().item())

    return history

# Run VI once and cache the trajectory for the interactive plot
vi_history = run_vi_optimization(num_steps=150, record_points=150)

```

[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

What do we notice about the VI approximation compared to MCMC? In what ways is it better or worse? Comparing the ELBO plot versus the MCMC trace plot, what differences do you notice about the convergence behavior of the two methods?

7.5.7 Monte Carlo Variational Inference

By assuming a certain form of the proposal distribution $q_\phi(\mathbf{w})$, we were able to compute the KL divergence term in closed form and perform VI analytically. However, in more complex models (like neural networks), we often cannot compute these expectations exactly. In such cases, we can resort to using **Monte Carlo Variational Inference** to approximate the ELBO and its gradients.

Instead of integrating analytically, we draw samples from our Gaussian variational family and use the *score-function identity* below to update the parameters. This reliance on Monte Carlo introduces sampling noise, so we will see below how the ELBO estimate bounces around and how the mean and covariance of q_ϕ drift across the banana distribution. The only change from the closed-form solution above is how we estimate expectations, where now our gradients still come from the “score function” of the Gaussian.²

The **score-function identity** states that for any distribution $q_\phi(\mathbf{w})$ parameterized by ϕ :

$$\nabla_\phi \mathbb{E}_{q_\phi}[f(\mathbf{w})] = \mathbb{E}_{q_\phi}[f(\mathbf{w}) \nabla_\phi \log q_\phi(\mathbf{w})]$$

where in our case, $f(\mathbf{w}) = \log p(Y|X, \mathbf{w}) + \log p(\mathbf{w}) - \log q_\phi(\mathbf{w}) = \log p(Y, \mathbf{w}|X) - \log q_\phi(\mathbf{w})$. (This allows us to compute gradients of the ELBO with respect to ϕ using samples from q_ϕ .)

²It is a little silly to resort to Monte Carlo here since we can compute the KL in closed form, but this is just for pedagogical purposes to show you the difference. In real applications with neural networks, we could not compute these expectations analytically.

This score-function identity is also sometimes called the “REINFORCE” gradient estimator in the reinforcement learning literature, as it was first popularized there and uses something called the “log-derivative trick”³ to move the gradient operator inside the expectation:

$$\begin{aligned}
\nabla_\phi \mathbb{E}_{q_\phi}[f(\mathbf{w})] &= \nabla_\phi \int q_\phi(\mathbf{w}) f(\mathbf{w}) d\mathbf{w} \\
&= \int f(\mathbf{w}) \nabla_\phi q_\phi(\mathbf{w}) d\mathbf{w} \\
&= \int f(\mathbf{w}) q_\phi(\mathbf{w}) \nabla_\phi \log q_\phi(\mathbf{w}) d\mathbf{w} \\
&= \mathbb{E}_{q_\phi}[f(\mathbf{w}) \nabla_\phi \log q_\phi(\mathbf{w})]
\end{aligned}$$

So putting this all together for our toy example case here, we can estimate the gradient of the ELBO as:

$$\begin{aligned}
\nabla_\phi \mathcal{L}(\phi) &= \nabla_\phi \mathbb{E}_{q_\phi}[\log p(Y, \mathbf{w}|X)] - \nabla_\phi \mathbb{E}_{q_\phi}[\log q_\phi(\mathbf{w})] \\
&= \mathbb{E}_{q_\phi}[(\log p(Y, \mathbf{w}|X) - \log q_\phi(\mathbf{w})) \nabla_\phi \log q_\phi(\mathbf{w})] \\
&\approx \frac{1}{S} \sum_{s=1}^S (\log p(Y, \mathbf{w}^{(s)}|X) - \log q_\phi(\mathbf{w}^{(s)})) \nabla_\phi \log q_\phi(\mathbf{w}^{(s)}) \quad \text{where } \mathbf{w}^{(s)} \sim q_\phi(\mathbf{w})
\end{aligned}$$

So we can see that this will break down into the following steps:

1. Sample S weight vectors $\mathbf{w}^{(s)}$ from the variational distribution $q_\phi(\mathbf{w})$
2. For each sample, compute the log joint probability $\log p(Y, \mathbf{w}^{(s)}|X)$ and the log variational probability $\log q_\phi(\mathbf{w}^{(s)})$. We subtract these to get what is commonly called the “advantage” term, though we can see that it is just the difference between how well the sample explains the data versus how probable it is under the variational distribution, which is also akin to something called the “likelihood ratio”.
3. Compute the score function $\nabla_\phi \log q_\phi(\mathbf{w}^{(s)})$ for each sample, which in this case is intuitively the direction (in the model weights) that will increase the log likelihood of the variational distribution.
4. We multiply these gradients by the advantage term of that sample from step 2. Intuitively, what this is doing is weighting the gradient direction by how much better that sample explains the data versus how probable it is under the variational distribution. If a sample explains the data well but is improbable under q_ϕ , we want to move q_ϕ in that direction to increase its probability.
5. Average these gradients over all S samples to get an estimate of $\nabla_\phi \mathcal{L}(\phi)$

³Recall that $\nabla_\phi \log(z) = \frac{\nabla_\phi z}{z}$.

```

# Monte Carlo VI using score-function gradients for a diagonal Gaussian

def torch_log_p(z: torch.Tensor) -> torch.Tensor:
    x = z[:, 0]
    y = z[:, 1]
    return -0.5 * ((x**2) / 9.0 + (y - 0.1 * x**2) ** 2)

def log_q_diag(z: torch.Tensor, mu: torch.Tensor, log_sigma: torch.Tensor) -> torch.Tensor:
    diff = z - mu
    inv_var = torch.exp(-2.0 * log_sigma)
    quad = torch.sum(diff**2 * inv_var, dim=1)
    log_det = 2.0 * torch.sum(log_sigma)
    return -0.5 * (quad + log_det + 2.0 * LOG_2PI)

steps = 600
batch_size = 256
lr = 0.08
record_every = 1

mu = torch.tensor([0.0, 0.0])
log_sigma = torch.log(torch.ones(2) * 2.5)

mc_history = {
    "steps": [],
    "mu": [],
    "cov": [],
    "elbo": [],
    "grad_var_mu": [],
    "grad_var_log_sigma": []
}

for step in range(steps):

    # Sample from q(z; mu, sigma)
    sigma = torch.exp(log_sigma)
    eps = torch.randn(batch_size, 2)

    z_samples = (mu + sigma * eps).detach()

    # Compute score-function gradients under samples from q
    log_p = torch_log_p(z_samples)
    log_q = log_q_diag(z_samples, mu, log_sigma)

```



```

# Compute the ELBO gradient estimates (also called the advantage in RL))
advantage = log_p - log_q

# Compute score functions for q at the samples
score_mu = (z_samples - mu) / (sigma**2)
score_log_sigma = -1.0 + (z_samples - mu) ** 2 / (sigma**2)

# Multiply the advantage with the score functions to get gradient estimates
grad_mu_samples = advantage.unsqueeze(1) * score_mu
grad_log_sigma_samples = advantage.unsqueeze(1) * score_log_sigma

# Take the average over the MC samples
grad_mu = grad_mu_samples.mean(dim=0)
grad_log_sigma = grad_log_sigma_samples.mean(dim=0)

# Now we can do a simple gradient ascent step
mu = mu + lr * grad_mu
log_sigma = log_sigma + lr * grad_log_sigma

if step % record_every == 0 or step == steps - 1:
    cov = np.diag(torch.exp(2.0 * log_sigma).cpu().numpy())
    mc_history["steps"].append(step)
    mc_history["mu"].append(mu.cpu().numpy())
    mc_history["cov"].append(cov)
    mc_history["elbo"].append(advantage.mean().item())
    mc_history["grad_var_mu"].append(grad_mu_samples.var(dim=0, unbiased=False).mean().item())
    mc_history["grad_var_log_sigma"].append(grad_log_sigma_samples.var(dim=0, unbiased=False).mean().item())

```

[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

```

Final mean: [0.05686871 0.33599067]
Final marginal std: [1.9444891 1.0056936]
Average gradient variance ( $\mu$ ): 5.516

```

7.5.8 The Reparameterization Trick

From the above we saw that we can now estimate VI updates even if we don't have a closed form solution, provided that we can compute gradients w.r.t. the score function (essentially the log likelihood of q), which we could in principle get through something like Automatic

Differentiation. However, there is a problem with this: to compute the score function gradients, we need to differentiate through samples drawn from $q_\phi(\mathbf{w})$. That is, our expectation is over samples $\mathbf{w} \sim q_\phi(\mathbf{w})$, and the score function $\nabla_\phi \log q_\phi(\mathbf{w})$ depends on ϕ both directly (since q_ϕ is parameterized by ϕ) and indirectly through the samples $\mathbf{w}$. This leads to stochasticity in the gradients due to the sampling process, which can result in high-variance gradient estimates.

To get around this, we can use something called the **reparameterization trick**, whose key idea is to move the stochasticity/randomness **outside** of the gradient computation. This is done by expressing samples from $q_\phi(\mathbf{w})$ as a deterministic function of ϕ and some noise variable ϵ that is independent of ϕ . For example, if $q_\phi(\mathbf{w})$ is a Gaussian with mean μ_ϕ and standard deviation σ_ϕ , we can write:

$$\mathbf{w} = \mu_\phi + \sigma_\phi \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Where the expectation is not over $\mathbf{w} \sim q_\phi(\mathbf{w})$ anymore, but rather over $\epsilon \sim \mathcal{N}(0, I)$.

Note now that $\mathbf{w}$ is still a random variable, but the randomness comes solely from ϵ , which is independent of ϕ . Instead ϕ becomes a deterministic transformation of ϵ . This allows us to rewrite the ELBO expectation as:

$$\begin{aligned} \nabla_\phi \mathcal{L}(\phi) &= \mathbb{E}_{q_\phi} [(\log p(Y, \mathbf{w}|X) - \log q_\phi(\mathbf{w})) \nabla_\phi \log q_\phi(\mathbf{w})] \\ &= \mathbb{E}_{\epsilon \sim \mathcal{N}(0, I)} [(\log p(Y, \mathbf{w}(\epsilon, \phi)|X) - \log q_\phi(\mathbf{w}(\epsilon, \phi))) \nabla_\phi \log q_\phi(\mathbf{w}(\epsilon, \phi))] \\ &\approx \frac{1}{S} \sum_{s=1}^S (\log p(Y, \mathbf{w}(\epsilon^{(s)}, \phi)|X) - \log q_\phi(\mathbf{w}(\epsilon^{(s)}, \phi))) \nabla_\phi \log q_\phi(\mathbf{w}(\epsilon^{(s)}, \phi)) \quad \text{where } \epsilon^{(s)} \sim \mathcal{N}(0, I) \end{aligned}$$

This subtle change has a few important implications:

1. The randomness is now isolated in ϵ , which does not depend on ϕ . This means that when we compute gradients w.r.t. ϕ , we can treat ϵ as a constant. This reduces the variance of our gradient estimates significantly.
2. The gradients $\nabla_\phi \log q_\phi(\mathbf{w}(\epsilon, \phi))$ can now be computed using standard automatic differentiation techniques, as $\mathbf{w}$ is a deterministic function of ϕ and ϵ .
3. This trick is particularly useful when $q_\phi(\mathbf{w})$ is a continuous distribution like a Gaussian, but it can be extended to non-continuous distributions as well, which we may cover in a later notebook.

This approach is sometimes called the “pathwise derivative estimator” because it allows us to compute gradients by following a deterministic path through the parameter space, rather than relying on stochastic sampling, and sits in contrast to “score-function” estimators which rely on sampling directly from the distribution.

```

# Reparameterized VI with automatic differentiation
mu_param = torch.nn.Parameter(torch.tensor([0.0, 0.0]))
log_sigma_param = torch.nn.Parameter(torch.log(torch.ones(2) * 2.5))
optimizer = Adam([mu_param, log_sigma_param], lr=0.05)

steps = 600
batch_size = 256
record_every = 1

reparam_history = {
    "steps": [],
    "mu": [],
    "cov": [],
    "elbo": [],
    "grad_var_mu": [],
    "grad_var_log_sigma": [],
    "grad_norm": []
}

for step in range(steps):
    optimizer.zero_grad()

    # Sample from q(z; mu, sigma) using reparameterization
    sigma = torch.exp(log_sigma_param)
    eps = torch.randn(batch_size, 2)
    z = mu_param + sigma * eps

    # Compute ELBO using reparameterization trick
    log_p = torch_log_p(z)
    log_q = -0.5 * ((eps**2).sum(dim=1) + 2.0 * torch.sum(log_sigma_param) + 2.0 * LOG_2PI)

    # Compute ELBO (aka the advantage)
    elbo = (log_p - log_q).mean()

    # Call AD backward to compute gradients directly on the ELBO
    (-elbo).backward()

    grad_norm = torch.sqrt(sum(param.grad.detach().pow(2).sum() for param in [mu_param, log_sigma_param]))
    optimizer.step()

    if step % record_every == 0 or step == steps - 1:
        with torch.no_grad():

```

```

sigma_det = torch.exp(log_sigma_param)
z_det = (mu_param + sigma_det * eps).detach()
grad_log_p = torch.stack([
    -(z_det[:, 0] / 9.0) + 0.2 * z_det[:, 0] * (z_det[:, 1] - 0.1 * z_det[:, 0] ** 2),
    -(z_det[:, 1] - 0.1 * z_det[:, 0] ** 2)
], dim=1)
grad_mu_samples = grad_log_p
grad_log_sigma_samples = grad_log_p * (sigma_det * eps) + 1.0

cov = np.diag(torch.exp(2.0 * log_sigma_param).cpu().numpy())
reparam_history["steps"].append(step)
reparam_history["mu"].append(mu_param.detach().cpu().numpy())
reparam_history["cov"].append(cov)
reparam_history["elbo"].append(elbo.item())
reparam_history["grad_var_mu"].append(grad_mu_samples.var(dim=0, unbiased=False).mean().item())
reparam_history["grad_var_log_sigma"].append(grad_log_sigma_samples.var(dim=0, unbiased=False).mean().item())
reparam_history["grad_norm"].append(grad_norm)

```

[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

```

Final mean: [0.03177835 0.3603806 ]
Final marginal std: [1.9253347 1.0171496]
Average gradient variance ( $\mu$ ): 1.408

```

7.5.9 Black-Box Variational Inference

We saw above how to use the reparameterization trick to help reduce variance in estimating the VI update in cases where $q_\phi(\mathbf{w})$ is a continuous distribution, such as a Gaussian or even a simpler feedforward Neural Network. In many cases (like we will see next in VAEs) this is sufficient and we are good to go.

However, there are certain cases where basic reparameterization will not work. For example, if I pick a variational family $q_\phi(\mathbf{w})$ that is no longer a nice smooth function of the parameters w , we will not be able to reparameterize $q_\phi(\mathbf{w})$ into a smooth deterministic function. This is common in cases where the desired $q_\phi(\mathbf{w})$ has discrete variables, such as clusters or categories.

In such cases, we actually have three options:

1. We can go back to score estimators like we did earlier in the notebook. We will suffer from higher variance updates but it will at least allow us to train the model even under discrete or non-continuous samples.⁴ This is the approach we will take below.
2. We can essentially try to “relax” discontinuities by finding a distribution that can approximate the “hard” or discrete objects we wish to study by variants that can be differentiated through. The price we will pay for this is that the optimal solution to the relaxed version of the model will be slightly biased away from the optimal solution to the original hard problem, but often this approximation error and bias is acceptable. Classical examples of these distributions include the Gumbel-Softmax or Concrete Distribution for relaxing Categorical variables, or the Birkhoff Polytope for relaxing Permutation Matrices. These extensions go beyond what I want to cover in this book, but are fascinating topics to read more about.
3. There are also classes of “Straight-Through Estimators” which essentially detach certain gradients from the computational graph such that one can sample hard/discrete samples, but then access non-hard gradients when calling automatic differentiation. These inherit similar problems to the relaxed distributions mentioned above, which is that the gradients you compute using Straight-Through estimators are only approximating, in a heuristic sense, the true gradients.

We will demonstrate an example below where reparameterization will not be effective by fitting a three member Gaussian Mixture Model as our $q_\phi(\mathbf{w})$ and resorting to score-function estimators. Note that this below example takes a significantly longer time to fit/train than the examples we have used so far.

```
# Black-box VI with a Gaussian mixture and REINFORCE gradients
try:
    from scipy.stats import gaussian_kde
except ImportError:
    gaussian_kde = None

def log_mixture_prob(z: torch.Tensor, locs: torch.Tensor, log_scales: torch.Tensor, logits: torch.Tensor)
    diff = z.unsqueeze(1) - locs.unsqueeze(0)
    inv_var = torch.exp(-2.0 * log_scales).unsqueeze(0)
    quad = torch.sum(diff**2 * inv_var, dim=2)
    log_det = 2.0 * log_scales.sum(dim=1)
    log_weights = F.log_softmax(logits, dim=0)
    component_log_prob = -0.5 * (quad + log_det.unsqueeze(0) + 2.0 * LOG_2PI)
    return torch.logsumexp(log_weights.unsqueeze(0) + component_log_prob, dim=1)

def sample_mixture(locs: torch.Tensor, log_scales: torch.Tensor, logits: torch.Tensor, num_samples: int)
    weights = F.softmax(logits, dim=0)
```

⁴There are some tricks to reduce the variance by subtracting a moving “baseline” average to the advantage term, which is implemented below, but we will not go into the details of this here.

```

cat = torch.distributions.Categorical(probs=weights)
indices = cat.sample((num_samples,))
sigma = torch.exp(log_scales)
eps = torch.randn(num_samples, 2)
samples = locs[indices] + sigma[indices] * eps
return samples.detach()

def radial_density_estimate(samples: np.ndarray, X: np.ndarray, Y: np.ndarray, bandwidth: float = 0.7, max_centers: int = 100):
    rng_local = np.random.default_rng(0)
    if len(samples) > max_centers:
        idx = rng_local.choice(len(samples), max_centers, replace=False)
        centers = samples[idx]
    else:
        centers = samples
    diff_x = X[None, :, :] - centers[:, 0][:, None, None]
    diff_y = Y[None, :, :] - centers[:, 1][:, None, None]
    density = np.exp(-(diff_x**2 + diff_y**2) / (2.0 * bandwidth**2)).sum(axis=0)
    norm = (2.0 * math.pi * bandwidth**2) * centers.shape[0]
    return density / norm

num_components = 3
logits = torch.nn.Parameter(torch.zeros(num_components))
locs = torch.nn.Parameter(torch.tensor([[6.0, 6.0], [0.0, 0.0], [-6.0, 6.0]]))
log_scales = torch.nn.Parameter(torch.log(torch.ones(num_components, 2) * 2.5))
optimizer = Adam([logits, locs, log_scales], lr=0.05)

steps = 250
batch_size = 512
eval_samples = 6000
record_every = 1
baseline = 0.0
baseline_beta = 0.05

bbvi_history = {
    "steps": [],
    "elbo": [],
    "grad_norm": [],
    "baseline": [],
    "mean": [],
    "cov": [],
    "density": []
}

```

```

for step in range(steps):
    optimizer.zero_grad()
    sigma = torch.exp(log_scales)

    weights = F.softmax(logits, dim=0)
    cat = torch.distributions.Categorical(probs=weights)
    indices = cat.sample((batch_size,))
    eps = torch.randn(batch_size, 2)
    samples = (locs[indices] + sigma[indices] * eps).detach()

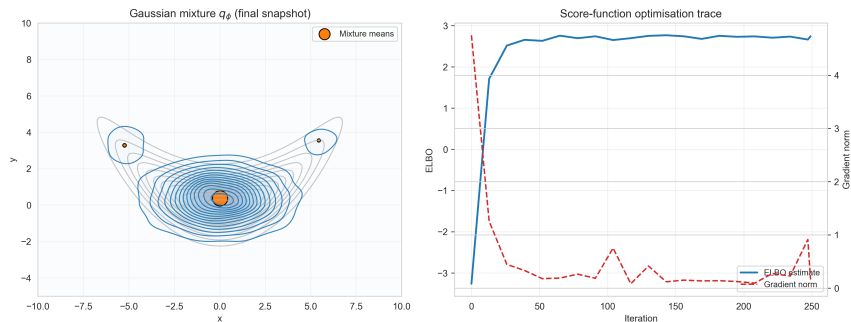
    log_q = log_mixture_prob(samples, locs, log_scales, logits)
    log_p = torch_log_p(samples)
    elbo_estimate = (log_p - log_q).mean().item()

    if step == 0:
        baseline = elbo_estimate
    baseline = (1.0 - baseline_beta) * baseline + baseline_beta * elbo_estimate
    advantage = (log_p - log_q - baseline).detach()

    loss = -(advantage * log_q).mean()
    loss.backward()
    grad_norm = torch.sqrt(sum(param.grad.detach().pow(2).sum() for param in [logits, locs, log_scales])).
    optimizer.step()

    if step % record_every == 0 or step == steps - 1:
        with torch.no_grad():
            eval_samples_tensor = sample_mixture(locs, log_scales, logits, eval_samples)
            eval_np = eval_samples_tensor.cpu().numpy()
            mean_np = eval_np.mean(axis=0)
            cov_np = np.cov(eval_np, rowvar=False)
            if gaussian_kde is not None:
                kde = gaussian_kde(eval_np.T)
                density = kde(np.vstack([X_grid.ravel(), Y_grid.ravel()])).reshape(X_grid.shape)
            else:
                density = radial_density_estimate(eval_np, X_grid, Y_grid)
            bbvi_history["steps"].append(step)
            bbvi_history["elbo"].append(elbo_estimate)
            bbvi_history["grad_norm"].append(grad_norm)
            bbvi_history["baseline"].append(baseline)
            bbvi_history["mean"].append(mean_np)
            bbvi_history["cov"].append(cov_np)
            bbvi_history["density"].append(density)

```



Final mixture weights: [0.043 0.906 0.051]

Final mixture means: [[5.449 3.545]

[0.021 0.365]

[-5.234 3.275]]

Average ELBO estimate: 2.372

With this you now have a high-level overview of some of this major approaches to performing Variational Inference as well as their pros and cons. We can now move on to applying this to a more complex example of the Neural Network from earlier.

7.5.10 Example of using VI for Neural Networks

Since in this case the Neural Network is a continuous and differentiable function, we will leverage the reparameterization trick to allow us to automatically differentiate backwards through the network to perform the VI updates.

```
# Variational Inference for Bayesian Neural Network
class BayesianNNVI(nn.Module):
    """
    Bayesian Neural Network with Variational Inference.

    Each weight has a variational posterior  $q(w) = N(\mu, \sigma^2)$ .
    """

    def __init__(self, hidden_size=5):
        super().__init__()
        self.hidden_size = hidden_size

        # Variational parameters for layer 1: input -> hidden
```



```

        self.fc1_mu_weight = nn.Parameter(torch.randn(hidden_size, 1) * 0.1)
        self.fc1_mu_bias = nn.Parameter(torch.randn(hidden_size) * 0.1)
self.fc1_logsigma_weight = nn.Parameter(torch.ones(hidden_size, 1) * -3) # Start with low variance
        self.fc1_logsigma_bias = nn.Parameter(torch.ones(hidden_size) * -3)

        # Variational parameters for layer 2: hidden -> output
        self.fc2_mu_weight = nn.Parameter(torch.randn(1, hidden_size) * 0.1)
        self.fc2_mu_bias = nn.Parameter(torch.randn(1) * 0.1)
        self.fc2_logsigma_weight = nn.Parameter(torch.ones(1, hidden_size) * -3)
        self.fc2_logsigma_bias = nn.Parameter(torch.ones(1) * -3)

def sample_weights(self):
    """Sample weights using reparameterization trick."""
    # Layer 1
    fc1_weight = self.fc1_mu_weight + torch.exp(self.fc1_logsigma_weight) * torch.randn_like(self.fc1_mu_weight)
    fc1_bias = self.fc1_mu_bias + torch.exp(self.fc1_logsigma_bias) * torch.randn_like(self.fc1_mu_bias)

    # Layer 2
    fc2_weight = self.fc2_mu_weight + torch.exp(self.fc2_logsigma_weight) * torch.randn_like(self.fc2_mu_weight)
    fc2_bias = self.fc2_mu_bias + torch.exp(self.fc2_logsigma_bias) * torch.randn_like(self.fc2_mu_bias)

    return (fc1_weight, fc1_bias, fc2_weight, fc2_bias)

def forward(self, x, sample=True):
    """Forward pass with sampled or mean weights."""
    x = x.view(-1, 1)

    if sample:
        fc1_w, fc1_b, fc2_w, fc2_b = self.sample_weights()
    else:
        fc1_w, fc1_b = self.fc1_mu_weight, self.fc1_mu_bias
        fc2_w, fc2_b = self.fc2_mu_weight, self.fc2_mu_bias

    # Forward propagation
    h = F.relu(F.linear(x, fc1_w, fc1_b))
    y = F.linear(h, fc2_w, fc2_b)

    return y.squeeze()

def kl_divergence(self, prior_std=1.0):
    """Compute KL(q||p) where p is N(0, prior_std^2)."""
    kl = 0

```

```

# KL for each parameter: KL(N(mu, sigma^2) || N(0, prior_std^2))
for mu_param, logsigma_param in [
    (self.fc1_mu_weight, self.fc1_logsigma_weight),
    (self.fc1_mu_bias, self.fc1_logsigma_bias),
    (self.fc2_mu_weight, self.fc2_logsigma_weight),
    (self.fc2_mu_bias, self.fc2_logsigma_bias)
]:
    sigma = torch.exp(logsigma_param)
    kl += 0.5 * torch.sum(
        (mu_param**2 + sigma**2) / prior_std**2 - 1 - 2*logsigma_param + 2*np.log(prior_std)
    )

return kl

def elbo(self, x, y, n_samples=5, noise_std=0.1, prior_std=1.0):
    """
    Compute the Evidence Lower Bound.

    ELBO = E_q[log p(y|x,w)] - KL(q||p)
    """
    x = x.view(-1, 1)

    # Expected log-likelihood (Monte Carlo estimate)
    log_lik = 0
    for _ in range(n_samples):
        y_pred = self.forward(x, sample=True)
        log_lik += -0.5 * torch.sum((y - y_pred) / noise_std) ** 2)
    log_lik = log_lik / n_samples
    log_lik -= len(y) * np.log(noise_std * np.sqrt(2 * np.pi))

    # KL divergence
    kl = self.kl_divergence(prior_std)

    # ELBO
    elbo = log_lik - kl

    return elbo, log_lik, kl

# Initialize variational model
vi_model = BayesianNNVI(hidden_size=5)

# Uncomment below if you would like a listing of all parameters and counts:

```

```
# print("Variational parameters:")
# for name, param in vi_model.named_parameters():
#     print(f" {name}: shape {param.shape}")
#     n_var_params = sum(p.numel() for p in vi_model.parameters())
#     print(f"Total variational parameters: {n_var_params}")
#     print("(2x the number of network weights for mean + variance)")
```

Training Bayesian NN with Variational Inference...

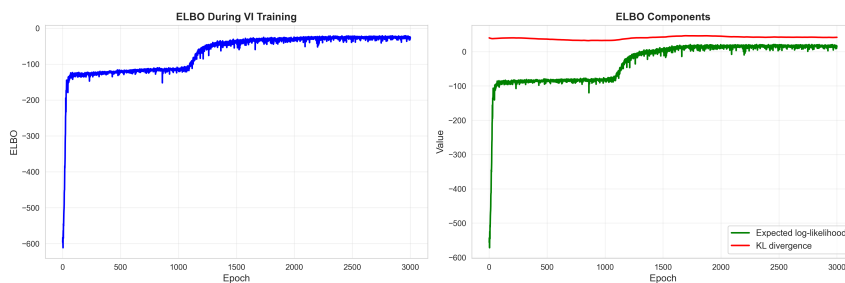
Epochs: 3000, MC samples: 5

Epoch 500	ELBO: -123.28	Log-lik: -86.77	KL: 36.51
Epoch 500	ELBO: -123.28	Log-lik: -86.77	KL: 36.51
Epoch 1000	ELBO: -113.35	Log-lik: -80.70	KL: 32.65
Epoch 1000	ELBO: -113.35	Log-lik: -80.70	KL: 32.65
Epoch 1500	ELBO: -35.02	Log-lik: 7.50	KL: 42.52
Epoch 1500	ELBO: -35.02	Log-lik: 7.50	KL: 42.52
Epoch 2000	ELBO: -32.42	Log-lik: 12.57	KL: 44.99
Epoch 2000	ELBO: -32.42	Log-lik: 12.57	KL: 44.99
Epoch 2500	ELBO: -25.91	Log-lik: 16.31	KL: 42.22
Epoch 2500	ELBO: -25.91	Log-lik: 16.31	KL: 42.22
Epoch 3000	ELBO: -27.61	Log-lik: 14.09	KL: 41.70

Training complete!

Epoch 3000 | ELBO: -27.61 | Log-lik: 14.09 | KL: 41.70

Training complete!

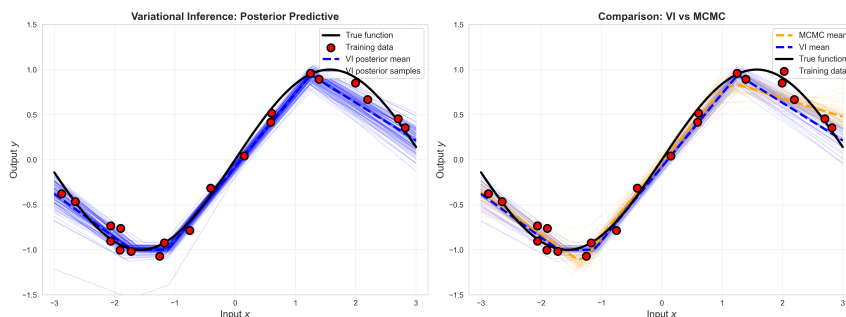


Final ELBO breakdown:

ELBO: -27.61

Expected log-likelihood: 14.09

KL divergence: 41.70



Note the difference in training times between the MCMC Method and VI Method. The main thing we give up for this increase in speed is the

7.6 Possible Additional Experiments

Now that you understand the full pipeline, you can explore below how some of the fundamental parameters of the model might affect the results:

💡 Experiment 1: Vary Hidden Layer Size

Train VI models with `hidden_size = 2, 5, 10, 20`. Observe: - How does model flexibility change? - Does uncertainty increase or decrease? - Is there a risk of overfitting with larger networks?

💡 Experiment 2: Effect of Training Data Size

Generate datasets with `n_train = 5, 10, 20, 50`. For each: - Train the VI model - Plot posterior predictive distributions - Compare the model uncertainty in data-rich vs data-sparse regions What do you notice?

💡 Experiment 3: Prior Sensitivity

Try different prior standard deviations: `prior_std = 0.1, 1.0, 10.0`. Observe: - How does this affect the learned posterior? - Does a strong prior (small std) regularize more? - What happens with a weak prior (large std)?

We can do Experiment 1 together as an example, and leave the remainder for you to do independently:

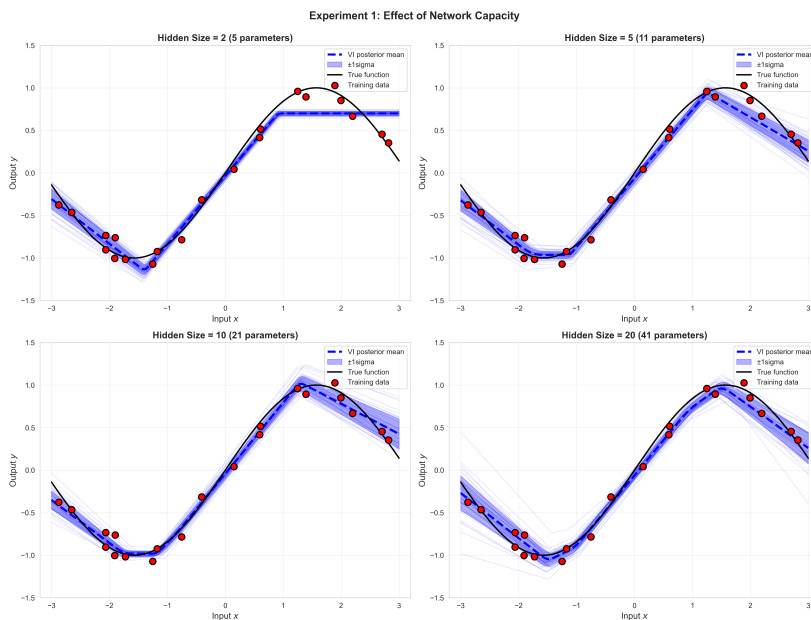
Experiment 1: Effect of Hidden Layer Size

Training VI models with different architectures...

```

Training with hidden_size = 2...
  Final ELBO: -21.87
Training with hidden_size = 5...
  Final ELBO: -21.87
Training with hidden_size = 5...
  Final ELBO: -30.33
Training with hidden_size = 10...
  Final ELBO: -30.33
Training with hidden_size = 10...
  Final ELBO: -45.41
Training with hidden_size = 20...
  Final ELBO: -45.41
Training with hidden_size = 20...
  Final ELBO: -82.38
Training complete!
  Final ELBO: -82.38
Training complete!

```



7.7 Summary and the bridge to VAEs

We saw a few key takeaways in this notebook:

Bayesian Inference Quantifies Uncertainty - Instead of a single “best” model, we maintain a distribution over models - The **posterior predictive distribution** $p(y_*|x_*, X, Y)$ gives us predictions **with uncertainty**

Linear Models Have Limitations - **Bayesian linear regression** has exact, closed-form posteriors (Gaussian) - This allows us to compute predictive mean and variance analytically - However, they cannot capture nonlinear relationships, as we saw in this example

Neural Networks Provide Flexibility at the cost of an intractable posterior - Even a small 1-hidden-layer network can approximate complex functions - **Prior predictive** shows diverse possible functions before seeing data - **Challenge**: Posterior $p(\mathbf{w}|X, Y)$ is no longer tractable.

MCMC: Accurate but Slow - Markov Chain Monte Carlo samples from the true posterior - **Pros**: Asymptotically exact, unbiased - **Cons**: Computationally expensive, slow for high-dimensional models - Useful as a **reference** for validation

Variational Inference: Fast Approximation - Approximate $p(\mathbf{w}|X, Y)$ with a tractable family $q_\phi(\mathbf{w})$ - Optimize ϕ to maximize the **ELBO**:

$$\mathcal{L}(\phi) = \mathbb{E}_{q_\phi} [\log p(Y|X, \mathbf{w})] - \text{KL}(q_\phi(\mathbf{w})||p(\mathbf{w}))$$

- **Reparameterization trick** enables low-variance gradient estimates - **Result**: Posterior approximation in seconds instead of minutes

Function Space vs Weight Space - We don’t visualize high-dimensional weight distributions directly - Instead, we visualize **functions** sampled from the posterior - **Posterior predictive** is what matters for making predictions!

7.7.1 Looking forward to VAEs (Variational Autoencoders)

The VI approach you learned here is the **same mathematical foundation** for VAEs. The key differences are:

- **Here (Bayesian NN)**: Posterior over **weights** $p(\mathbf{w}|X, Y)$
- **VAEs**: Posterior over **latent variables** $p(\mathbf{z}|\mathbf{x})$

The ELBO, reparameterization trick, and optimization strategy are identical.

Moreover, in VAEs, instead of optimizing $q_\phi(z)$ separately for each datapoint x , we learn an **encoder network** $q_\phi(z|x)$ that works for **all** x . This is called **amortized inference**.

Questions to Reflect on

Think about these before moving to the next notebook:

1. **Why does VI underestimate uncertainty?**

Hint: Is the Mean-field approximation where we assume independence between weights reasonable? When might it not be?

2. **When would MCMC be essential despite being slow?**

3. **Could we use a more expressive variational family?**

We used a Mean-field Gaussian here, but are we limited to this?

8 Introduction to Probabilistic Programming

In past chapters, we have looked at manually defining different generative models, as well as covered the fundamentals of inference algorithms (e.g., MCMC, Variational Inference). However, as models grow in complexity, or if we wish to combine elements of probabilistic models with Deep Learning, manually implementing these models and inference algorithms can become cumbersome and error-prone. *Probabilistic Programming Languages* (PPLs) were developed to help address this challenge. They work by providing high-level abstractions for defining probabilistic models as well as routines for performing inference on those models. There are [lists of PPLs available online](#), but for the purposes of this chapter, our examples will use a PyTorch-based PPL called [Pyro](#).

We will demonstrate this by considering the example of inferring unknown material properties from observed stress-strain data. First, we will define a simple linear elastic model where the stress is proportional to strain via Young's modulus (E). We will then use Pyro to infer the posterior distribution of E given some observed data. It is a bit unnecessary to use a PPL for such a simple model, but it will serve to illustrate the basic concepts of how they work. We will then move onto a more complex model involving plasticity, where the benefits of using a PPL become more apparent, and we will see how we can recover a posterior distribution over multiple material parameters.

8.1 Learning Objectives

- Connect probabilistic programming concepts (random variables, guides, inference) to stress-strain modeling tasks.
- Wrap physics-based constitutive laws in Pyro to infer interpretable parameters with credible intervals.

Using device: cuda

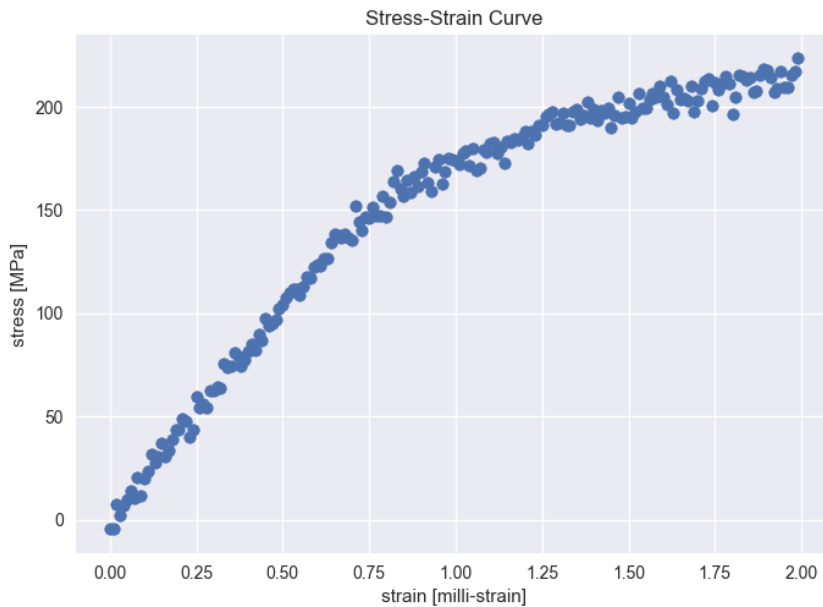
8.2 Problem Setup: Inferring Material Properties from Stress-Strain Data

As the running example in this notebook, we will consider the problem of inferring material properties from observed stress-strain data. Specifically, we will assume that we have existing

experimental data that relates stress (σ) to strain (ϵ) for a given material, and we wish to infer the underlying material parameters that govern this relationship. We will use a Ramberg-Osgood model to describe the stress-strain behavior of the material, which is given by the equation:

$$\epsilon = \frac{\sigma}{E} + \alpha \left(\frac{\sigma}{\sigma_0} \right)^n$$

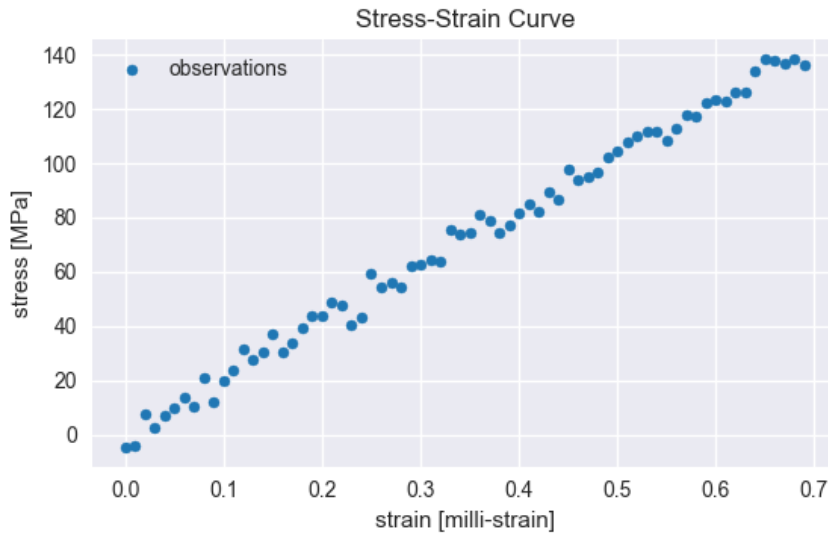
where E is Young's modulus, σ_0 is the yield stress, α is a material constant, and n is the strain hardening exponent. Our goal will be to infer the parameters E , σ_0 , α , and n from observed stress-strain data using probabilistic programming. For our example below, we will use generic properties of a mild steel (e.g., S235) although this could be adapted to any material or to a different constitutive model. We will also add in a bit of measurement noise (which you can also feel free to update as you see fit to see how this changes the later uncertainty bounds). Let's plot some example data:



8.3 Bayesian Linear Regression

Let's first start with a simple model that assumes Hooke's law: $\sigma = E\epsilon + \epsilon$. For this, we can use Linear Regression, where our goal is to compute the posterior distribution over E . While we could compute this in closed form for such a simple model, the goal here is to expose Pyro's main primitives: `pyro.sample` for declaring priors/likelihoods, `pyro.param` inside a guide, and stochastic variational inference (SVI) with the ELBO. Even though this toy example is small, it mirrors the workflow for more complex models that we will see later.

Since we are dealing with a linear model, let's restrict the training data to small strains where the linear approximation holds decently well:



For this Bayesian linear regression, we will: 1. Define a model with a prior on Young's modulus E and a likelihood for observed stresses. 2. Define a guide (variational distribution) to approximate the posterior over E . 3. Set up Stochastic Variational Inference (SVI) to optimize the ELBO and learn the posterior parameters.

```
# Define Pyro model, guide, and run SVI
pyro.clear_param_store()

def linear_model(strain: torch.Tensor, stress_obs: torch.Tensor | None = None) -> None:
    E = pyro.sample("E", dist.LogNormal(math.log(2.0e5), 0.2))
    sigma_noise = pyro.sample("sigma_noise", dist.LogNormal(math.log(5.0), 0.4))
    with pyro.plate("data", strain.shape[0]):
        mean_stress = E * strain
        pyro.sample("obs", dist.Normal(mean_stress, sigma_noise), obs=stress_obs)

# Here we define what Pyro calls a "guide" which is just
# the variational distribution over the latent variables.
def linear_guide(strain: torch.Tensor, stress_obs: torch.Tensor | None = None) -> None:
    qE_loc = pyro.param("qE_loc", torch.tensor(12.2))
    qE_scale = pyro.param("qE_scale", torch.tensor(0.1), constraint=constraints.positive)
    pyro.sample("E", dist.LogNormal(qE_loc, qE_scale))

    # Guide for the measurement noise
    q_noise_loc = pyro.param("q_noise_loc", torch.tensor(math.log(6.0)))
```

```

q_noise_scale = pyro.param("q_noise_scale", torch.tensor(0.1), constraint=constraints.positive)
pyro.sample("sigma_noise", dist.LogNormal(q_noise_loc, q_noise_scale))

# Now we train the model using Stochastic Variational Inference (SVI)
optimizer = AdamW({"lr": 0.01})
svi = SVI(linear_model, linear_guide, optimizer, loss=Trace_ELBO())

loss_history = []
num_steps = 3000
for step in range(num_steps):
    loss = svi.step(strain, stress)
    loss_history.append(loss / strain.shape[0])
    if (step + 1) % 500 == 0:
        print(f"Step {step + 1:04d} | per-sample ELBO: {loss_history[-1]:.2f}")

```

```

Step 0500 | per-sample ELBO: 2.82
Step 1000 | per-sample ELBO: 2.89
Step 1500 | per-sample ELBO: 2.77
Step 2000 | per-sample ELBO: 2.76
Step 2500 | per-sample ELBO: 2.78
Step 3000 | per-sample ELBO: 2.76

```

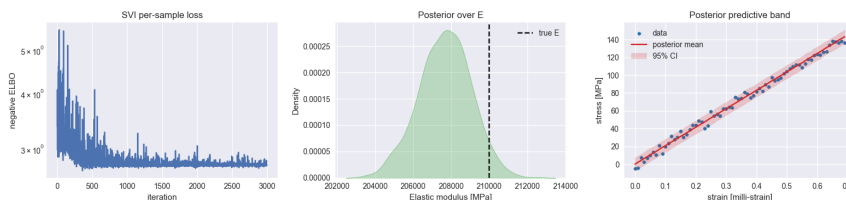
Now that the model has been trained, we can sample the posterior predictive of the model to sample possible values of E :

```

# Posterior diagnostics and predictive uncertainty
predictive = Predictive(
    linear_model,
    guide=linear_guide,
    num_samples=1500,
    return_sites=("E", "obs"),
)
posterior_samples = predictive(strain)

```

And we can go ahead and plot it:



We see the convergence of the SVI loss over training, as well as the posterior distribution over E , and also the posterior predictive distribution over stress-strain curves. The shaded regions represent credible intervals, indicating the uncertainty in our predictions due to the uncertainty in E . We can see from the middle plot that the estimate for E is lower than the true value, which is due both to noise, as well as model misspecification (the true model is not perfectly linear). However, the true value of E is on the border of the 95% credible interval.

8.4 Bayesian Material Model Identification (Ramberg-Osgood)

For that simple example above, we didn't really need a Probabilistic Programming Language and could have computed the posterior in closed form. Now let's move onto a more complex material model where a closed form solution is not as straightforward, yet a PPL has a remarkably simple implementation. Specifically, we will infer the parameters of the Ramberg-Osgood model from above, using the noisy stress-strain data:

$$\epsilon = \frac{\sigma}{E} + \alpha \left(\frac{\sigma}{\sigma_0} \right)^n$$

Notice below that we can directly define the model in the code itself and assign individual variables with priors using `pyro.sample`. The likelihood is also defined in a similar manner.

```
# Pyro model and inference for Ramberg-Osgood parameters
pyro.clear_param_store()

def ramberg_osgood_model(stress: torch.Tensor, strains: torch.Tensor | None = None) -> None:
    """
    Probabilistic model for Ramberg-Osgood material parameter inference.

    We infer E (Young's modulus), sigma_y (yield stress), and n (hardening exponent)
    from observed strain-stress pairs.
    """
    # Priors for material parameters
    # Note how we directly assign the priors in the code
    E = pyro.sample("E", dist.LogNormal(math.log(2.1e5), 0.2))
    sigma_y = pyro.sample("sigma_y", dist.LogNormal(math.log(235.0), 0.25))
    n = pyro.sample("n", dist.LogNormal(math.log(8.0), 0.3))

    # Prior for measurement noise
    sigma_noise = pyro.sample("sigma_noise", dist.LogNormal(math.log(2e-4), 0.3))

    # Forward model: compute predicted strain from stress using Ramberg-Osgood
```

```

#  $\epsilon = \sigma/E + \alpha*(\sigma/\sigma_y)^n$ 
alpha = 0.002 # Fixed material constant

with pyro.plate("data", stress.shape[0]):
    # Add small epsilon to avoid division by zero issues and clamp the stress ratio
    eps = 1e-10
    stress_ratio = torch.clamp(stress / (sigma_y + eps), min=eps, max=1e3)
    strain_pred = stress / (E + eps) + alpha * torch.pow(stress_ratio, n)

    # Likelihood: observed strain given predicted strain and noise
    pyro.sample("obs", dist.Normal(strain_pred, sigma_noise), obs=strains)

# Use AutoNormal guide for automatic variational inference
# This uses a Normal distribution with a diagonal covariance matrix as the
# Variational Distribution
#guide_ro = AutoNormal(ramberg_osgood_model)

# Alternatively, you can use a distribution with a full covariance:
guide_ro = AutoMultivariateNormal(ramberg_osgood_model)

# Set up SVI with optimizer
optimizer = ClippedAdam({"lr": 0.005, "clip_norm": 5.0})
svi = SVI(ramberg_osgood_model, guide_ro, optimizer, loss=Trace_ELBO())

# Run inference
ro_losses = []
num_steps = 10000
print("Running Ramberg-Osgood parameter inference...")
for step in range(num_steps):
    loss = svi.step(stresses, strains)
    ro_losses.append(loss / stresses.shape[0]+8)
    if (step + 1) % 800 == 0:
        print(f"Step {step + 1:04d} | per-sample loss: {ro_losses[-1]:.6f}")

print(f"\nFinal loss: {ro_losses[-1]:.6f}")

```

```

Running Ramberg-Osgood parameter inference...
Step 0800 | per-sample loss: 0.103314
Step 1600 | per-sample loss: 0.165335
Step 2400 | per-sample loss: 0.097428
Step 3200 | per-sample loss: 0.091438
Step 4000 | per-sample loss: 0.098814

```

```

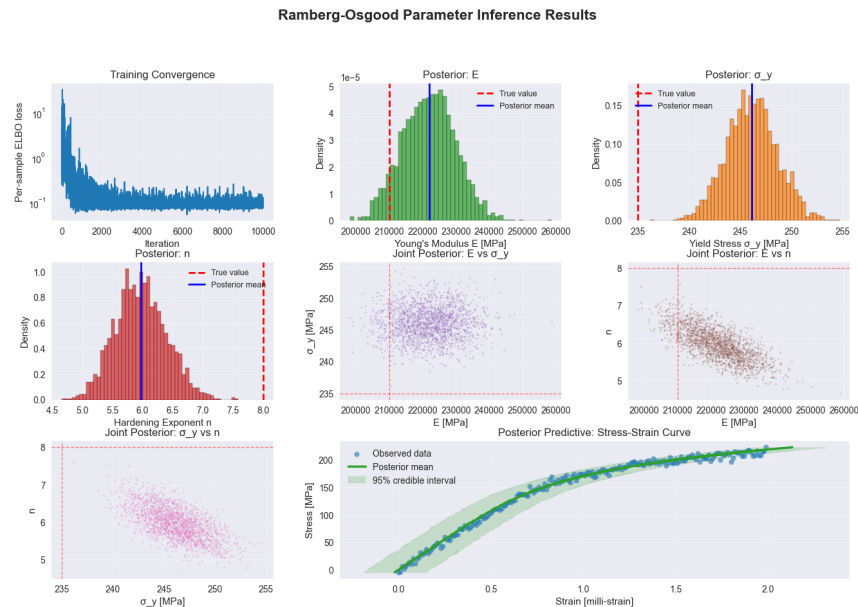
Step 4800 | per-sample loss: 0.102443
Step 5600 | per-sample loss: 0.081144
Step 6400 | per-sample loss: 0.082515
Step 7200 | per-sample loss: 0.082739
Step 8000 | per-sample loss: 0.082601
Step 8800 | per-sample loss: 0.087190
Step 9600 | per-sample loss: 0.087617

```

Final loss: 0.100413

===== POSTERIOR SUMMARY STATISTICS =====

Parameter	True Value	Posterior Mean	95% CI
E (MPa)	210000.0	221922.6	[206535.0, 237921.2]
σ_y (MPa)	235.0	246.1	[241.2, 251.0]
n (hardening exp)	8.0	5.97	[5.19, 6.82]



We see that while the posterior mean and credible intervals for the stress-strain curve fit the data well, there is still significant uncertainty in the inferred parameters, with the true values lying in some cases outside of the credible intervals. You can experiment with different levels of noise in the observed data, as well as different prior distributions to see how they affect

the posterior estimates. Also, if you look closely at the joint distributions of the parameters, you can see that there are correlations between them, indicating that certain combinations of parameters can produce similar stress-strain behavior. This highlights the importance of considering the full posterior distribution rather than just point estimates when inferring material properties.

For completeness, we also demonstrate below how to use Pyro's MCMC functionality, instead of SVI::

```
# Pyro Example using MCMC Inference:
nuts_kernel = NUTS(ramberg_osgood_model, adapt_step_size=True)
mcmc = MCMC(nuts_kernel, num_samples=2000, warmup_steps=300)
mcmc.run(stresses, strains)
mcmc.get_samples()['E'].mean(0)
```

```
Sample: 100%|██████████| 2300/2300 [02:53, 13.24it/s, step size=1.37e-01, acc. prob=0.940]
```

```
tensor(219903.2188)
```

```
MCMC Estimated E: 219903.22 MPa
```

```
MCMC Estimated  $\sigma_y$ : 244.88 MPa
```

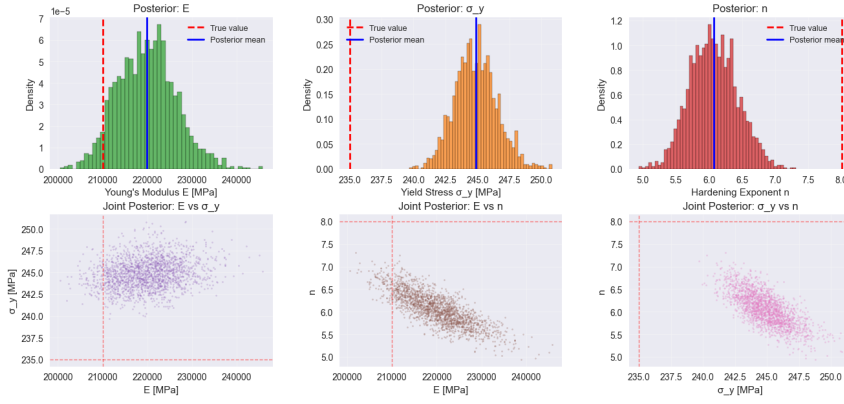
```
MCMC Estimated n: 6.07
```

```
=====
POSTERIOR SUMMARY STATISTICS
=====
```

Parameter	True Value	Posterior Mean	95% CI
E (MPa)	210000.0	219903.2	[207923.0, 233500.7]
σ_y (MPa)	235.0	244.9	[241.8, 248.2]
n (hardening exp)	8.0	6.07	[5.41, 6.79]

```
=====
```

Ramberg-Osgood Parameter Inference Results (MCMC)



8.5 Summary and Next Steps

This chapter introduced the concept of Probabilistic Programming Languages (PPLs) and demonstrated their application in inferring material properties from stress-strain data using Pyro. We started with a simple Bayesian linear regression example to illustrate the basic workflow of defining a model, guide, and performing inference using Stochastic Variational Inference (SVI). We then extended this to a more complex Ramberg-Osgood material model, showcasing how PPLs can simplify the implementation of probabilistic models and inference for more intricate scenarios. While there wasn't much new in this chapter that we didn't already know how to do from prior chapters, the key takeaway is that PPLs in general can significantly streamline the process of defining and inferring complex probabilistic models, making them more accessible for practical applications in engineering.

Part II

Foundations of Generative Models

These chapters look more like a “traditional” textbook, in the sense that they cover individual techniques or concepts that I thought were maximally relevant to Mechanical Engineers at the time that I wrote the book. These chapters will be most immediately useful in terms of getting up to speed with specific types of models, but they are also the most likely to become out of date quickly as newer/better models are invented.

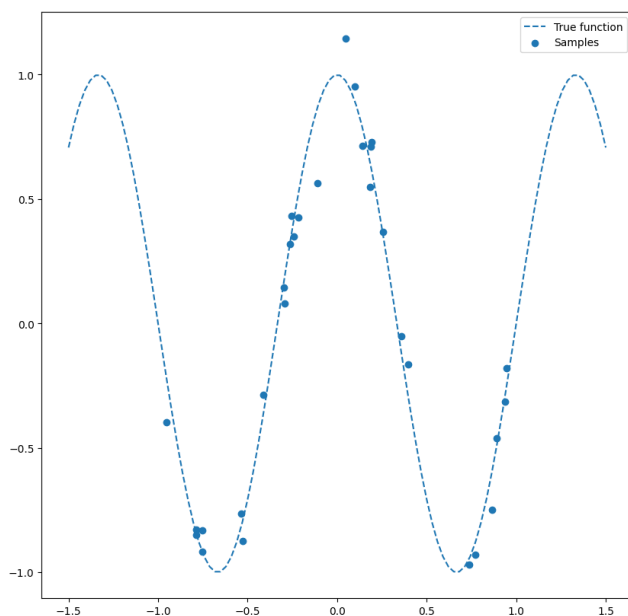
- Review of Prior Course Models – CNNs, UNets, RNNs, AEs
- Advanced Neural Models
 - The Attention Mechanism
 - Introduction to Transformers
 - Regularization of Neural Networks
 - Introduction to Geometric Deep Learning (Message Passing, GNNs, Working with Meshes)
 - Failure Mechanisms in Neural Models
- Probabilistic Models and Kernels
 - Introduction to Probabilistic Graphical Models
 - Exact Inference (MLE, MAP, EM)
 - Approximate Inference (MCMC, VI)
 - Introduction to Probabilistic Programming
 - Failure Mechanisms in Probabilistic Models
- Ensembles (Not included in current course for scope reasons)
 - Bagging
 - Boosting

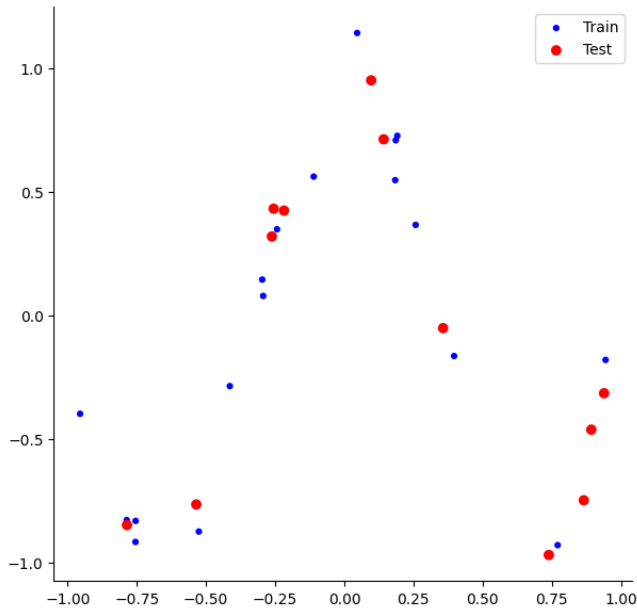
9 Review of Neural Networks

This notebook reviews some basic notation and concepts related to neural networks. It will skip many details that would have been covered in your earlier Stochastics and Machine Learning course, but will set a basis for us jumping into more advanced topics in a few chapters.

Before we get started with PyTorch code, I encourage you first to interactive with the ConvNetJS demo, specifically the [interactive 1D Regression](#) where you can toggle on and off the different layers and plotting functions, as well as the [2D Classification demo](#) to see how a neural network is just performing a series of feature transformations that ultimately lead to a linear classification boundary. These demos help build a good intuition for what is going on before we move onto more obtuse PyTorch code.

9.1 Example of PyTorch SGD for Linear Regression and a Simple Feed-Forward Network





9.1.1 Linear Regression

```
input_size = 1
output_size = 1
# Linear regression model
model = nn.Linear(input_size, output_size)

# Loss and optimizer
criterion = nn.MSELoss()
learning_rate = 0.1 # alpha
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
# Train the model
num_epochs = 100
for epoch in range(num_epochs):
    # Convert numpy arrays to torch tensors
    inputs = torch.from_numpy(X_train)
    targets = torch.from_numpy(y_train)

    # Forward pass
    outputs = model(inputs)
    loss = criterion(outputs, targets)
```

```

# Backward and optimize
optimizer.zero_grad()
loss.backward()
optimizer.step()

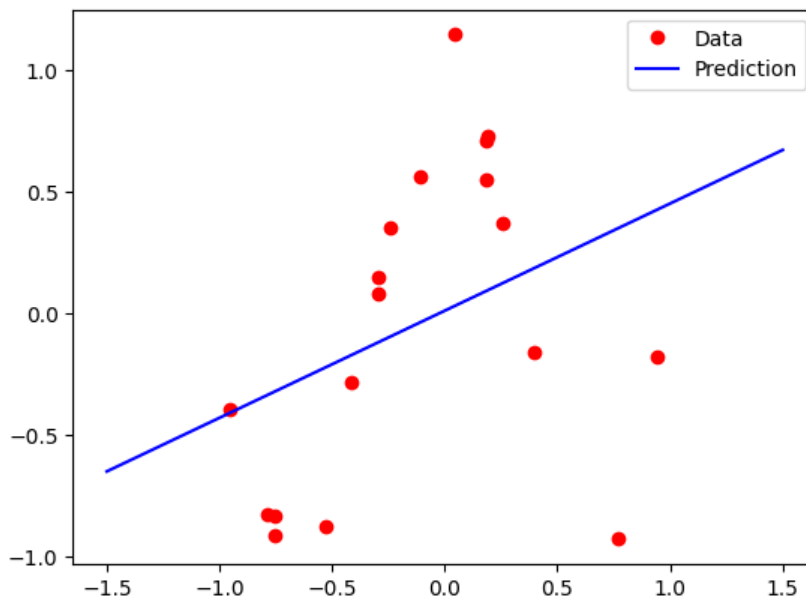
if (epoch+1) % 20 == 0:
    print ('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))

```

```

Epoch [20/100], Loss: 0.3992
Epoch [40/100], Loss: 0.3596
Epoch [60/100], Loss: 0.3553
Epoch [80/100], Loss: 0.3548
Epoch [100/100], Loss: 0.3548

```



9.1.2 Neural Network Examples

Example of Defining a Network via the full Module class

```

class Net(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Net, self).__init__()

```

```

# Fully-Connected Layer: 1 (input data) -> 5 (hidden node)
self.fc1 = nn.Linear(input_size, hidden_size)

# Non-Linear Layer
self.sigmoid = nn.Sigmoid()
# You can try other kinds as well
# self.relu = nn.ReLU()
# self.elu = nn.ELU()

# Fully-Connected Layer: 5 (hidden node) -> 1 (output)
self.fc2 = nn.Linear(hidden_size, 1)

# Forward pass builds the model prediction from the inputs
def forward(self, x):
    out = self.fc1(x)
    out = self.sigmoid(out)
    out = self.fc2(out)
    return out

# Build the network -- is it not trained yet
model = Net(input_size=1, hidden_size=5)

```

Example of building a model using the Sequential helper function

Example using Python list expansions to help build deeper networks

```
[Linear(in_features=1, out_features=7, bias=True), ReLU(), Linear(in_features=7, out_features=7, bias=True)]
```

9.1.3 Now let's do the actual training

```

# What Loss function should we use? MSE!
criterion = nn.MSELoss()

# What Optimization procedure should we use?

##### Change these and let's see how it affects the model fit #####
learning_rate = 0.05
weight_decay = 0.0

```

```
#####
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate, weight_decay=0.0)

### Train the model

# Convert numpy arrays to torch tensors
inputs = torch.from_numpy(X_train)
targets = torch.from_numpy(y_train)

num_epochs = 5000
for epoch in range(num_epochs):

    ## Do Forward pass
    # Make predictions
    outputs = model(inputs)
    # Compute the loss function
    loss = criterion(outputs, targets)

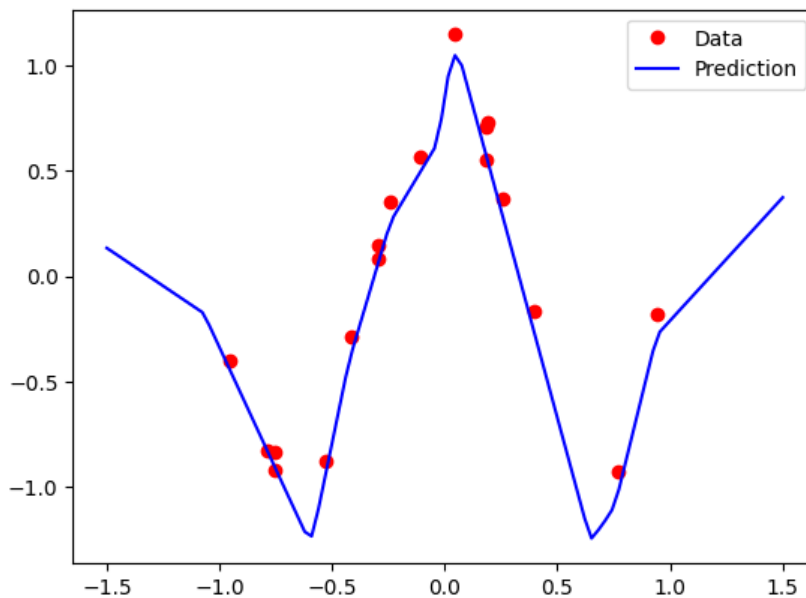
    ## Update the model
    # Reset the optimizer gradients
    optimizer.zero_grad()
    # Compute the gradient of the loss function
    loss.backward()
    # Do an optimization step
    optimizer.step()

    # Print the loss
    if (epoch+1) % 200 == 0:
        print ('Epoch [{:4}/{:}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
```

```
Epoch [ 200/5000], Loss: 0.0064
Epoch [ 400/5000], Loss: 0.0025
Epoch [ 600/5000], Loss: 0.0029
Epoch [ 800/5000], Loss: 0.0023
Epoch [1000/5000], Loss: 0.0023
Epoch [1200/5000], Loss: 0.0023
Epoch [1400/5000], Loss: 0.0023
Epoch [1600/5000], Loss: 0.0023
Epoch [1800/5000], Loss: 0.0022
Epoch [2000/5000], Loss: 0.0023
Epoch [2200/5000], Loss: 0.0022
Epoch [2400/5000], Loss: 0.0033
```

```
Epoch [2600/5000], Loss: 0.0023
Epoch [2800/5000], Loss: 0.0209
Epoch [3000/5000], Loss: 0.0218
Epoch [3200/5000], Loss: 0.0023
Epoch [3400/5000], Loss: 0.0022
Epoch [3600/5000], Loss: 0.0020
Epoch [3800/5000], Loss: 0.0070
Epoch [4000/5000], Loss: 0.0020
Epoch [4200/5000], Loss: 0.0019
Epoch [4400/5000], Loss: 0.0023
Epoch [4600/5000], Loss: 0.0019
Epoch [4800/5000], Loss: 0.0030
Epoch [5000/5000], Loss: 0.0118
```

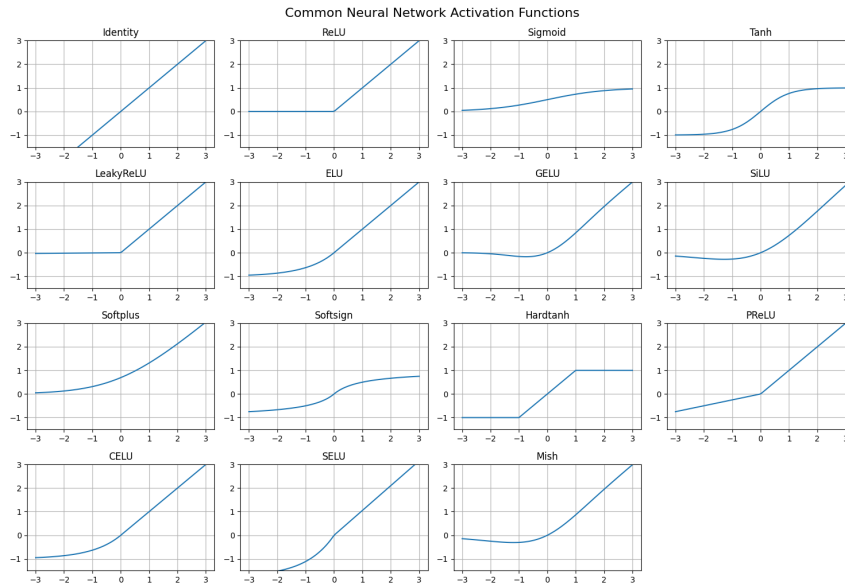
Now let's plot the prediction:



💡 Experiment: Effect of Regularization on Simple Toy Regression Problem

Revisit the above regression code and experiment with the weight decay parameter in the Adam optimizer. How does this affect the learned function? Why do you think this is?:

You can also play around with different activation functions, e.g., `nn.ReLU()`, `nn.Sigmoid()`, `nn.Tanh()`, etc., and the below plot pulls out several options from PyTorch for you to visualize the activation functions:

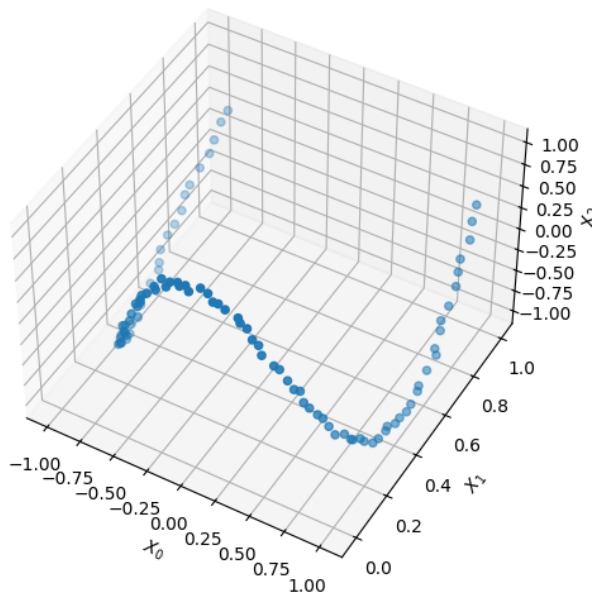


9.2 Unsupervised Learning using Autoencoders

For this demonstration, we will construct what is fundamentally a 1D function (t) but then embed it in a higher-dimensional space (3D) using a non-linear transformation. This will allow us to compare what a linear method (PCA) can do versus a non-linear method (Autoencoder), on a simple example.

```
# First we create a simple line (t)
t = np.linspace(-1,1,100)
```

```
# Now let's make it 3D and add some (optional) noise
noise_level = 0.01
# You can try out different functions below by uncommenting/commenting them
#X = np.vstack([t,1*t**2,-1*t**3, 0.4*t**5,t**2-t**3,-0.4*t**4]).T
X = np.vstack([2*np.sin(3*t),1*t**2,-1*t**3, 2*np.sin(6*t+1)-t**3,2*np.cos(3*t)]).T
X = np.vstack([np.sin(2*t),1*t**2,1*np.cos(5*t)]).T + noise_level*np.random.randn(len(t),3)
```



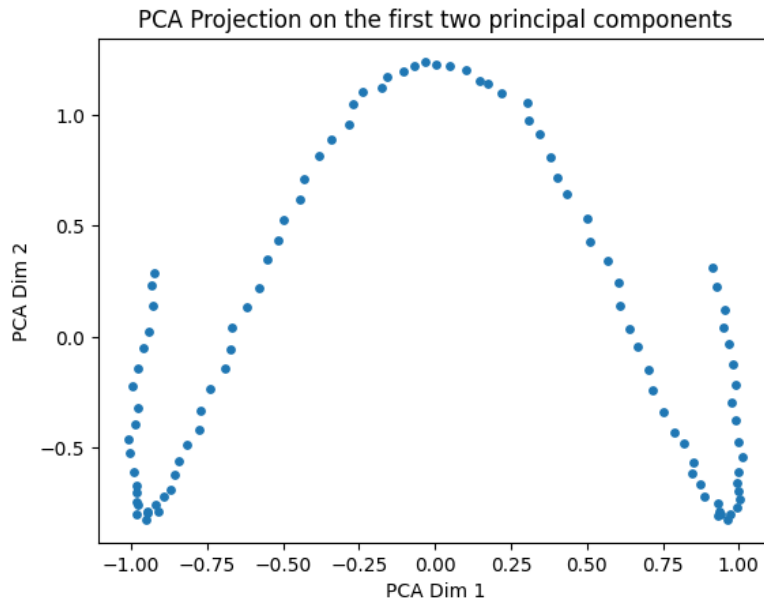
We can see that it results in 100 points each of which has three dimensions:

```
X.shape
```

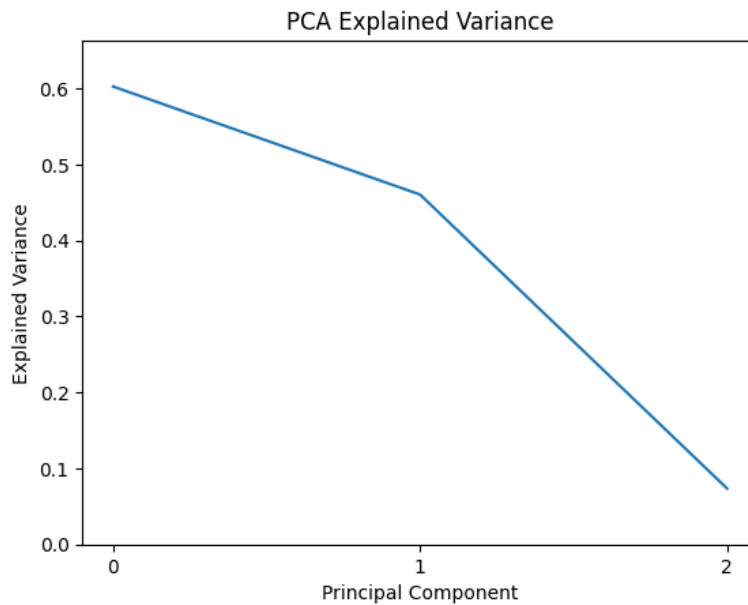
```
(100, 3)
```

We can attempt to reduce the dimensionality of this data using PCA, but as we can already see from the 3D plot, the data is not linearly embedded in 3D space, so PCA will not be able to find a good low-dimensional representation:

```
pca = PCA(3)
Z_PCA =pca.fit_transform(X)
plt.figure()
plt.scatter(Z_PCA[:,0],Z_PCA[:,1],s=15)
plt.xlabel('PCA Dim 1')
plt.ylabel('PCA Dim 2')
plt.title('PCA Projection on the first two principal components')
plt.show()
```

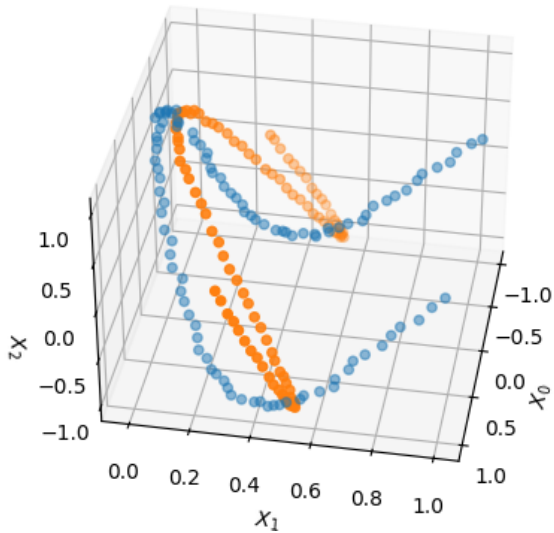


And we can also see this reflected in the explained variance, which shows that we need all three original dimensions to explain the variance in the data, even though we know that the data fundamentally lies on a 1D manifold:



We can see below what happens if we try to truncate PCA to only two dimensions and then reconstruct back to 3D space. The reconstruction is not very good, as expected, and what do you notice about the shape of the reconstructed data?

PCA Reconstruction with only two components



Now let's see how this works using an autoencoder. We will use a very simple architecture with just one hidden layer in the encoder and one hidden layer in the decoder, and we will use 1D latent space (since we know that the data is fundamentally 1D). You will have the option in the below code to change the number of hidden units in the encoder and decoder, as well as the number of latent dimensions, and see how this affects the results.

```
class Encoder(nn.Module):
    def __init__(self, num_input, num_latent, num_hidden):
        super().__init__()
        self.num_input = num_input
        self.num_latent = num_latent
        self.num_hidden = num_hidden

        # I encourage you to modify the architecture here by adding more layers or changing activation functions
        self.encode = nn.Sequential(
            nn.Linear(self.num_input, self.num_hidden),
            nn.ReLU(),
            #nn.Linear(self.num_hidden, self.num_hidden),
            #nn.ReLU(),
            nn.Linear(self.num_hidden, self.num_latent),
        )

    def forward(self, X):
        encoded = self.encode(X)
        return encoded
```

```

class Decoder(nn.Module):
    def __init__(self, num_input, num_latent,num_hidden):
        super().__init__()
        self.num_input = num_input
        self.num_latent = num_latent
        self.num_hidden = num_hidden

        self.decode = nn.Sequential(
            nn.Linear(self.num_latent, self.num_hidden),
            nn.ReLU(),
            #nn.Linear(self.num_hidden, self.num_hidden),
            #nn.ReLU(),
            nn.Linear(self.num_hidden, self.num_input)
        )

    def forward(self, Z):
        decoded = self.decode(Z)
        return decoded

class AutoEncoder(nn.Module):
    def __init__(self, num_input,num_latent,num_hidden):
        super().__init__()
        self.num_input = num_input
        self.num_latent = num_latent
        self.num_hidden = num_hidden

        self.encoder = Encoder(num_input = self.num_input,
                                num_latent = self.num_latent,
                                num_hidden = self.num_hidden)
        self.decoder = Decoder(num_input = self.num_input,
                                num_latent = self.num_latent,
                                num_hidden = self.num_hidden)

    def forward(self, X):
        encoded = self.encoder(X)
        decoded = self.decoder(encoded)
        return decoded, encoded # <- return a tuple of two values

    def transform(self,X):
        '''Take X and encode to latent space'''
        return self.encoder(X)

```

```

def inverse_transform(self,Z):
    '''Take Z and decode to X space'''
    return self.decoder(Z)

```

```

# Here we can set some parameters for the autoencoder that we are about to train
# What happens if you change them?
# e.g., increase/decrease num_latent, num_hidden, learning rate, weight decay
num_points, D_orig = X.shape
num_latent = 3
num_hidden = 5
model = AutoEncoder(D_orig,num_latent,num_hidden)
# Create the optimizer object:
# Adam optimizer with learning rate and weight decay
# I encourage you to try out different learning rates and weight decays and
# observe their effect on the model
optimizer = optim.AdamW(model.parameters(),
                        lr=1e-3,
                        weight_decay=1e-2)
# Add a mean-squared error loss
criterion = nn.MSELoss()

```

```

X_torch = torch.from_numpy(X)
X_torch = X_torch.float()

```

```

# Depending on the model architecture you use, you may need to increase or decrease this to get good training
epochs=10000
for epoch in range(epochs):
    loss = 0

    optimizer.zero_grad()

    # compute reconstructions
    decoded, encoded = model(X_torch)

    # compute training reconstruction loss
    train_loss = criterion(decoded, X_torch)

    # Total Loss
    loss = train_loss

    # compute accumulated gradients
    loss.backward()

```

```

# perform parameter update based on current gradients
optimizer.step()

# display the epoch training loss
if epoch%500==0:
    print("epoch : {}/{}", loss = {:.6f}".format(epoch + 1, epochs, loss.item()))

```

```

epoch : 1/10000, loss = 0.716544
epoch : 501/10000, loss = 0.350305
epoch : 1001/10000, loss = 0.090813
epoch : 1501/10000, loss = 0.010054
epoch : 2001/10000, loss = 0.000795
epoch : 2501/10000, loss = 0.000162
epoch : 3001/10000, loss = 0.000120
epoch : 3501/10000, loss = 0.000097
epoch : 4001/10000, loss = 0.000081
epoch : 4501/10000, loss = 0.000073
epoch : 5001/10000, loss = 0.000062
epoch : 5501/10000, loss = 0.000054
epoch : 6001/10000, loss = 0.000035
epoch : 6501/10000, loss = 0.000016
epoch : 7001/10000, loss = 0.000006
epoch : 7501/10000, loss = 0.000000
epoch : 8001/10000, loss = 0.000000
epoch : 8501/10000, loss = 0.000000
epoch : 9001/10000, loss = 0.000000
epoch : 9501/10000, loss = 0.000000

```

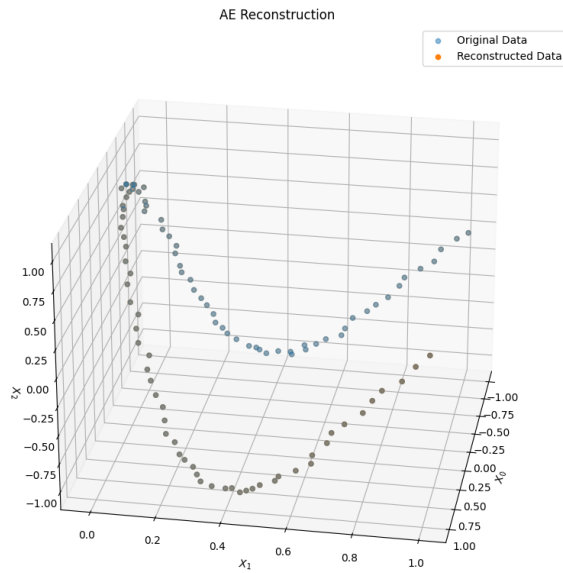
Now that the model is trained, we can put the data through the encoder and decoder to pull out both the encoded (i.e., latent) representation of each point, as well as the decoded (i.e., reconstructed) version of each point, and visualize the results.

```

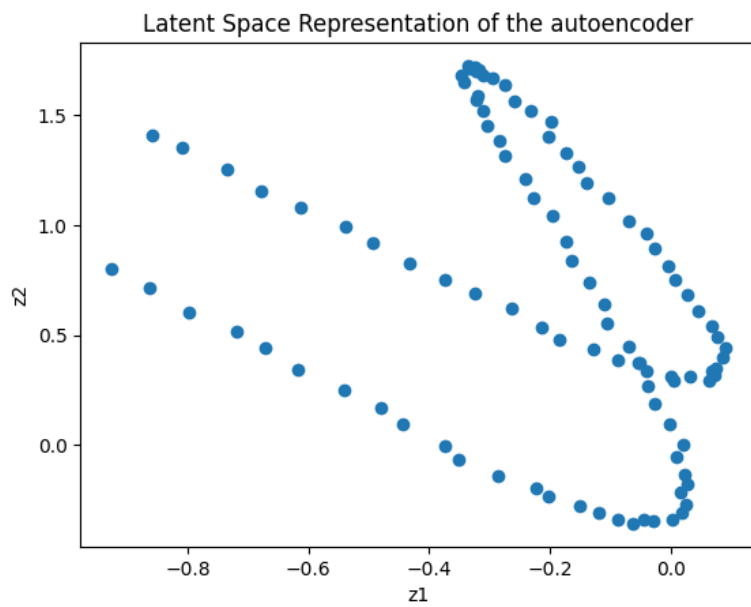
decoded, encoded = model(X_torch)

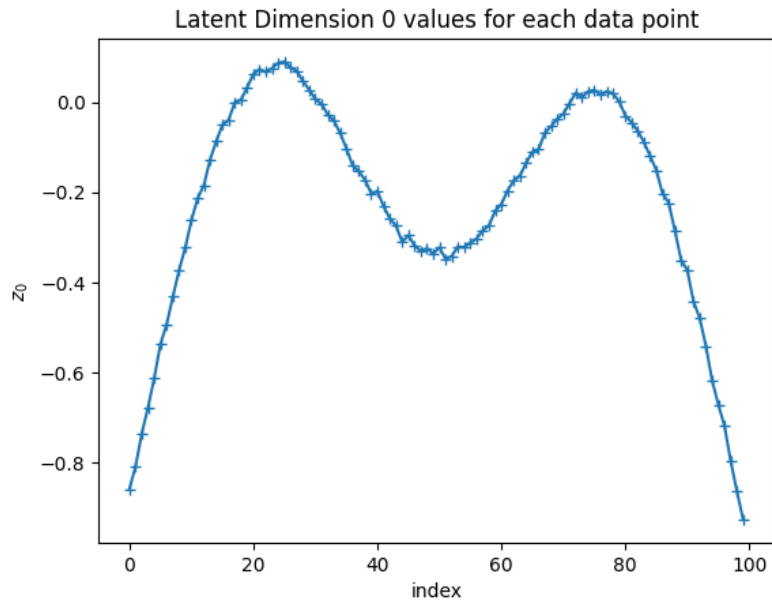
```

First, let's take a look at the reconstructed data in the original 3D space, compared to the original data:



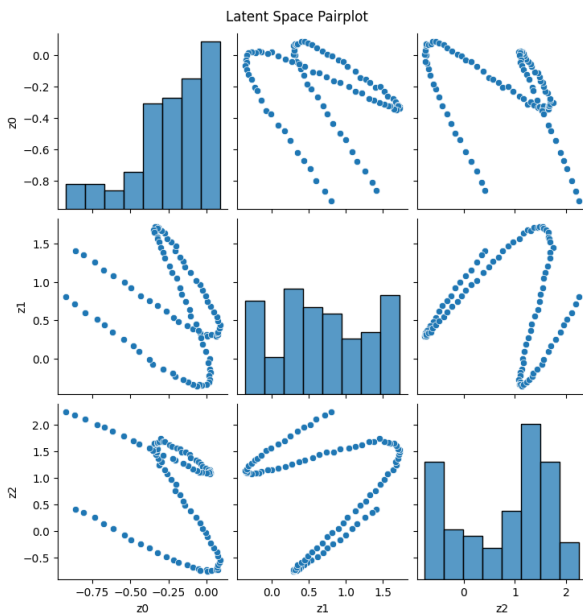
Let's see how the encoded (i.e., latent) points actually look in Z space:





We can also look at a pairplot of the points in the latent space to get a sense of how the different dimensions correlate with each other:

<Figure size 640x480 with 0 Axes>



💡 Experiment: Effect on Autoencoder Architecture on Reconstruction Accuracy and Latent Space Behavior

Revisit the above autoencoder code and experiment by changing the number of hidden units, the bottleneck (latent) dimension, and the number of layers in the encoder and decoder. How do these changes affect the reconstruction accuracy and the behavior of the learned latent space? Consider some of the below questions:

- We know that the data is fundamentally 1D, but why does setting the autoencoder latent dimension to 1 not work well? Why might it be useful to have a latent dimension larger than the true dimensionality of the data for this type of model?
- What happens if you set the latent dimension to be three or larger? Why do you think this happens? What have we given up by doing this?
- As you increase or decrease the number of hidden units in Autoencoder, how does this affect the reconstruction accuracy? Why do you think this is? Consider in particular the case where the number of hidden units is one or two.
- Unlike in PCA, when we re-run the autoencoder training, we get different results each time. Why do you think this is?
- Unlike PCA, the autoencoder does not guarantee that the latent dimensions are orthogonal or ordered by importance (e.g., z_0 being more important than z_1 , etc.). Do you see any evidence of this in the learned latent space? Why do you think this is?
- The Autoencoder used a ReLU activation function in the hidden layers. How does this manifest in the way that the network reconstructs the data? (Note, this may only be visible if you set the latent dimension to 1 and the number of hidden units to a small number, e.g., 2 or 3).
- We can see that when we set the latent dimension to two or greater, the autoencoder can reconstruct the data well, but it is not capturing the intrinsic dimensionality of the data. One way we might do this is by adding L1 regularization to the latent space coordinates, which is commented out in the code above. Try adding in this regularization. Does it fix the problem?

9.3 Least Volume Regularization

One of the problems that we saw in Autoencoders is that the latent space does not have any particular structure, and in particular, it is not guaranteed to be ordered, in the same sense as PCA. It was also difficult for us to determine the exact “size” of the latent dimension, since, as we saw, setting the latent dimension to 1 did not work well, even though we knew that the data was fundamentally 1D, due to training variability and the fact that the autoencoder could get trapped in local minima. One possible solution to this is to add a regularization term that encourages the latent space to be small in some sense. One such regularization is

called [Least Volume Regularization](#), which encourages the latent space to have a small volume by penalizing the volume of the encoded points in the latent space.

```
ambient_dim = X.shape[1]

# Change the below latent dimension to see what happens to the embedded points
latent_dim = 3

width = ambient_dim * 16
encoder = MLP(ambient_dim, latent_dim, [width] * 4)
decoder = MLP(latent_dim, ambient_dim, [width] * 4)

opt = torch.optim.Adam(list(encoder.parameters()) + list(decoder.parameters()), lr=1e-4)
```

```
# Set X to a torch tensor:
X_torch = torch.from_numpy(X).float()

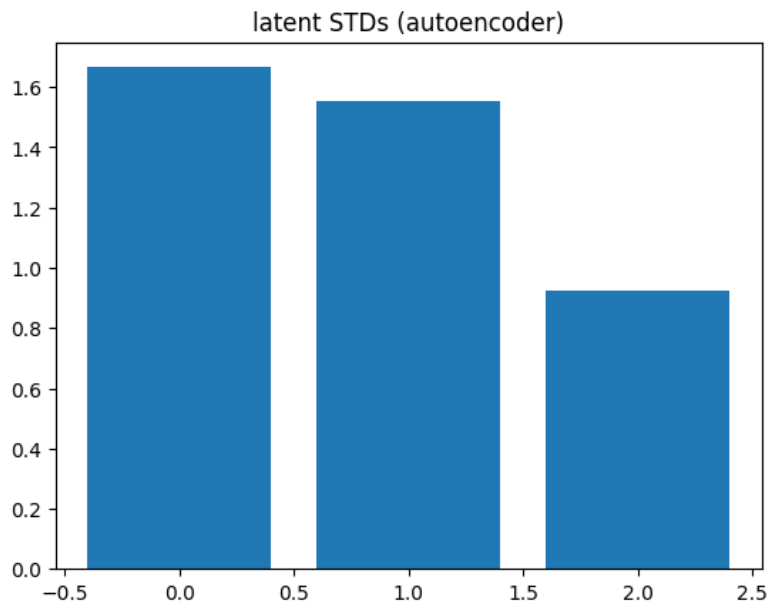
for i in range(5000):
    opt.zero_grad()
    z = encoder(X_torch)
    rec_loss = F.mse_loss(decoder(z), X_torch)
    loss = rec_loss
    # If you want, you can even add an L1 penalty on the latent space
    # to try to encourage sparsity by uncommenting the line below:
    #loss += 1e-3 * torch.mean(torch.abs(z))

    loss.backward()
    opt.step()
    if (i+1) % 100 == 0:
        print(f'Epoch {i:4}: rec = {rec_loss:.5g}')
```

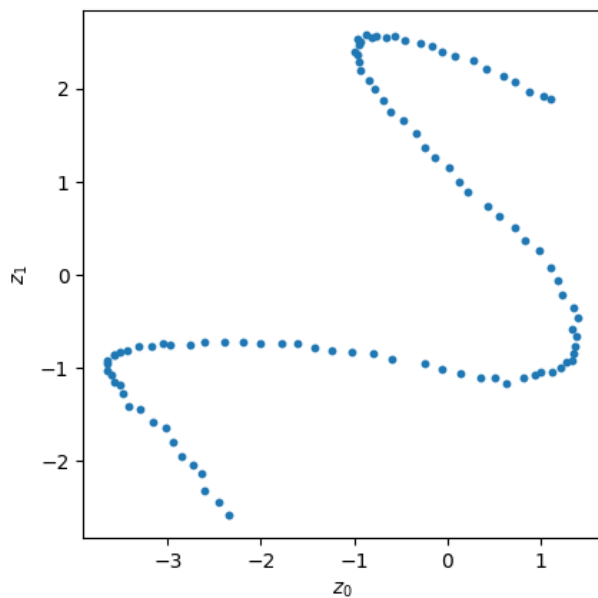
```
Epoch   99: rec = 0.39166
Epoch  199: rec = 0.31308
Epoch  299: rec = 0.048501
Epoch  399: rec = 0.023062
Epoch  499: rec = 0.015194
Epoch  599: rec = 0.010049
Epoch  699: rec = 0.0068286
Epoch  799: rec = 0.0048156
Epoch  899: rec = 0.0035384
Epoch  999: rec = 0.0026096
Epoch 1099: rec = 0.0018915
```

```
Epoch 1199: rec = 0.001313
Epoch 1299: rec = 0.00084967
Epoch 1399: rec = 0.00053819
Epoch 1499: rec = 0.00027124
Epoch 1599: rec = 0.00014737
Epoch 1699: rec = 0.00010135
Epoch 1799: rec = 8.2536e-05
Epoch 1899: rec = 7.4167e-05
Epoch 1999: rec = 6.9067e-05
Epoch 2099: rec = 6.5697e-05
Epoch 2199: rec = 6.2941e-05
Epoch 2299: rec = 6.0584e-05
Epoch 2399: rec = 5.8694e-05
Epoch 2499: rec = 5.6992e-05
Epoch 2599: rec = 5.5272e-05
Epoch 2699: rec = 5.3765e-05
Epoch 2799: rec = 5.2397e-05
Epoch 2899: rec = 5.1138e-05
Epoch 2999: rec = 5.0006e-05
Epoch 3099: rec = 4.8935e-05
Epoch 3199: rec = 4.7868e-05
Epoch 3299: rec = 4.6667e-05
Epoch 3399: rec = 4.5643e-05
Epoch 3499: rec = 4.4691e-05
Epoch 3599: rec = 4.3831e-05
Epoch 3699: rec = 4.3016e-05
Epoch 3799: rec = 4.2142e-05
Epoch 3899: rec = 4.1268e-05
Epoch 3999: rec = 4.0287e-05
Epoch 4099: rec = 3.9385e-05
Epoch 4199: rec = 3.8568e-05
Epoch 4299: rec = 3.7773e-05
Epoch 4399: rec = 3.7031e-05
Epoch 4499: rec = 3.6334e-05
Epoch 4599: rec = 3.5686e-05
Epoch 4699: rec = 3.5056e-05
Epoch 4799: rec = 3.4465e-05
Epoch 4899: rec = 3.388e-05
Epoch 4999: rec = 3.332e-05
```

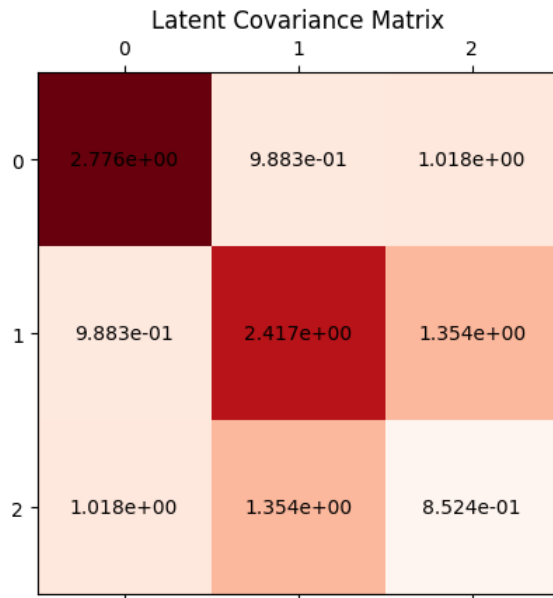
Now that the autoencoder has been trained, let's take a look at the standard deviation of the embedded points (i.e., in z). We can sort the latent dimensions according to which dimensions have the highest standard deviation in Z .



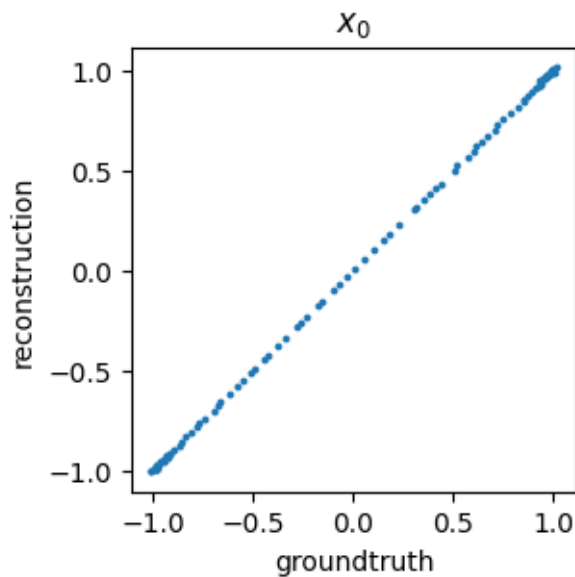
As with before, we can plot some of the data points in Z space:

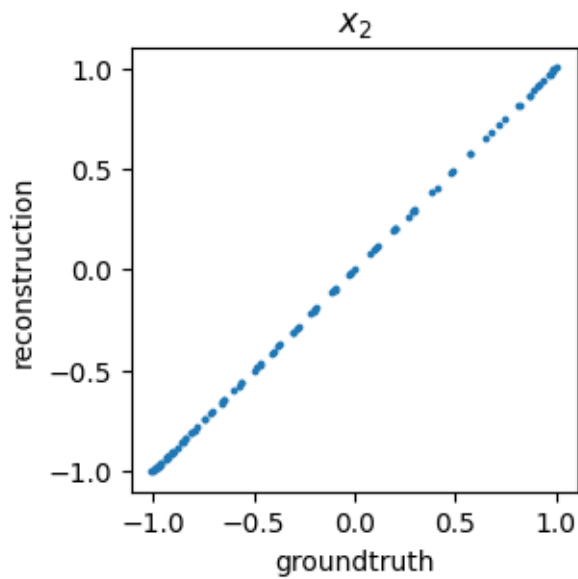
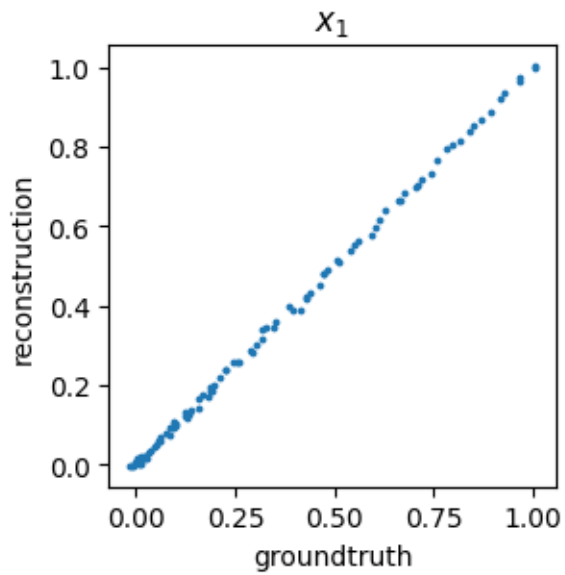


We can also plot the covariance among the latent codes of the embedded data:



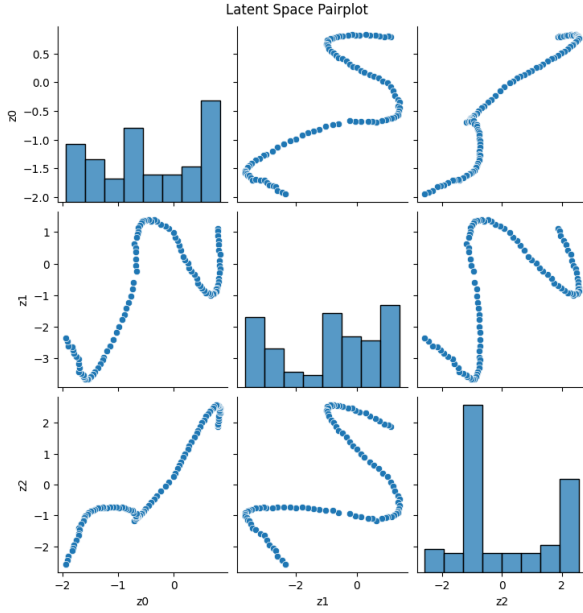
We can also get a general idea about how well we are reconstructing the original data by comparing the ground truth values versus predicted (i.e., encoded then decoded) data points – this is often called a parity plot:





Lastly, we can visualize a pairplot of the latent space to see how the different dimensions correlate with each other:

<Figure size 640x480 with 0 Axes>



9.4 How might we select the right order in an Autoencoder? Building the case for Least Volume Analysis (LVA)

Our explorations above exposed both the advantages and disadvantages of using non-linear maps to attempt to embed and capture the underlying distribution and topology of the data.

Below we describe the basic principle of “Least Volume” regularization in Autoencoders and demonstrate how it can be useful in providing automated order selection in over-parameterized Autoencoders. It can allow us to capture relevant topological structure, but with minimal dimension.

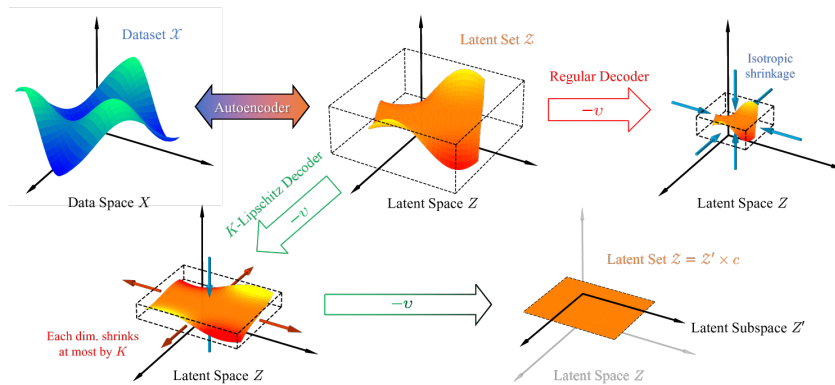


Image above from: Qiuyi Chen and Mark Fuge, “Least Volume Analysis”

In general, the idea behind least volume is that we want to encourage the latent space to take up as little volume as possible, while still being able to reconstruct the data well. This can be achieved by adding a regularization term to the loss function that penalizes the volume of the latent space. A simple way to do this is to penalize the geometric mean of the standard deviation of the latent dimensions, which encourages the latent space to be small in all dimensions. Specifically, we can minimize the product of all elements of the latent code's standard deviation vector $\prod \sigma$, which is equivalent to minimizing the exponential of the mean of the log of the standard deviation vector:

```
vol_loss = torch.exp(torch.log(z.std(0) + η).mean())
```

We add a small constant η to avoid numerical issues when any one of the standard deviation's in any dimension approaches zero – that is when the autoencoder eliminates a dimension, and thus $\prod \sigma$ would have a zero in the product. This loss term can be added to the reconstruction loss, weighted by a hyperparameter λ , to form the total loss.

In principle, while this loss can encourage the latent space to reduce its volume, there is one catch: the autoencoder could simply scale up the weights in the encoder and decoder to make the latent space arbitrarily small, while still being able to reconstruct the data well. To prevent this, we have to prevent the decoder from being able to arbitrarily increase its weights, and one easy way to enforce this is through *spectral normalization* on the weights of the decoder, which constrains the Lipschitz constant of the decoder to be at most 1. By preventing the decoder from scaling up its weights too much, and the encoder cannot easily defeat the volume penalty by isotropically shrinking the weights, and thus the only way to achieve a good volume penalty is to actually reduce dimensions.

Fun Fact: It turns out that in the case of an Autoencoder that only uses Linear layers, and with no activation functions, the least volume penalty is equivalent to PCA. That is, PCA can be seen as a special case of Least Volume Autoencoder. For more details on the mathematical proof, see [Proposition 15](#) in the original paper.

Below code implements the spectral normalized decoder:

```
from torch.nn.utils.parametrizations import spectral_norm

class SNLinearCombo(_Combo):
    def __init__(self, in_features, out_features, activation=nn.LeakyReLU(0.2)):
        super().__init__()
        self.model = nn.Sequential(
            spectral_norm(nn.Linear(in_features, out_features)),
            activation
        )

class SNMLP(MLP):
    def __init__(
```

```

self, in_features: int, out_features: int, layer_width: list,
    combo=SNLinearCombo):
    super().__init__(in_features, out_features, layer_width, combo)

def _build_model(self, combo):
    model = nn.Sequential()
    idx = -1
    for idx, (in_ftr, out_ftr) in enumerate(self.layer_sizes[:-1]):
        model.add_module(str(idx), combo(in_ftr, out_ftr))
    # Note here is the main difference: the last layer also has spectral normalization
    # This was not the case in the previous MLP definition
    model.add_module(str(idx+1), spectral_norm(nn.Linear(*self.layer_sizes[-1])))
    return model

```

```

width = ambient_dim * 16
# Note in particular the lack of the bottleneck choice below
# That is, we don't need to actually pick a bottleneck dimension -- LVA automatically determines this, like
encoder = MLP(ambient_dim, ambient_dim, [width] * 4)
# Note also the change in the decoder to have spectral normalization
decoder = SNMLP(ambient_dim, ambient_dim, [width] * 4)

opt = torch.optim.Adam(list(encoder.parameters()) + list(decoder.parameters()), lr=1e-3)

```

```

η, λ = 0.01, 0.01

for i in range(20000):
    opt.zero_grad()
    z = encoder(X_torch)
    rec_loss = F.mse_loss(decoder(z), X_torch)
    # Note below the least volume loss
    vol_loss = torch.exp(torch.log(z.std(0) + η).mean())
    loss = rec_loss + λ * vol_loss
    loss.backward()
    opt.step()

    if (i+1) % 1000 == 0:
        # Print floats with 5 significant digits and fill epoch with leading spaces if under 4 digits
        print(f'Epoch {i:4}: rec = {rec_loss:.5g}, vol = {vol_loss:.5g}')

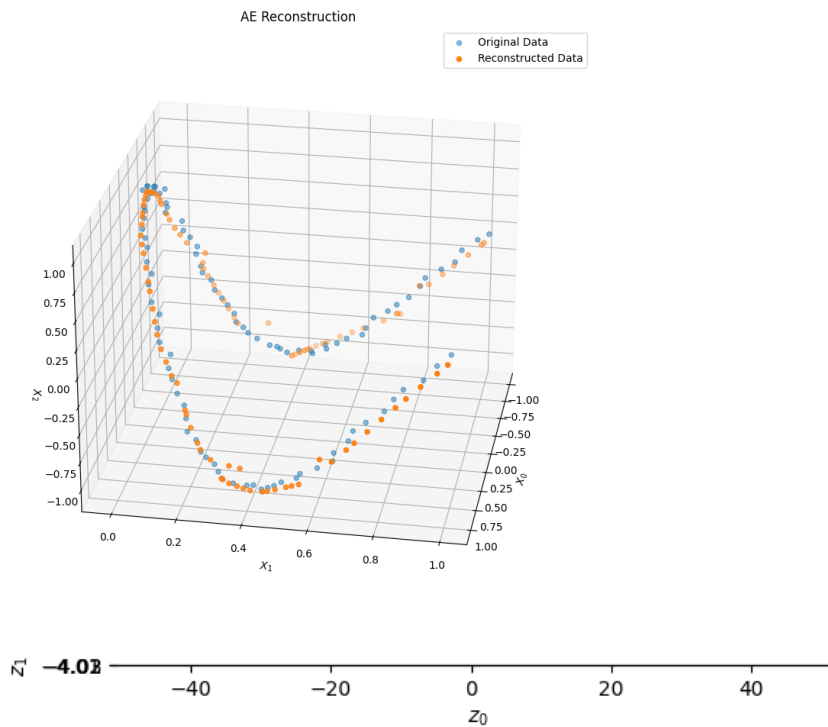
```

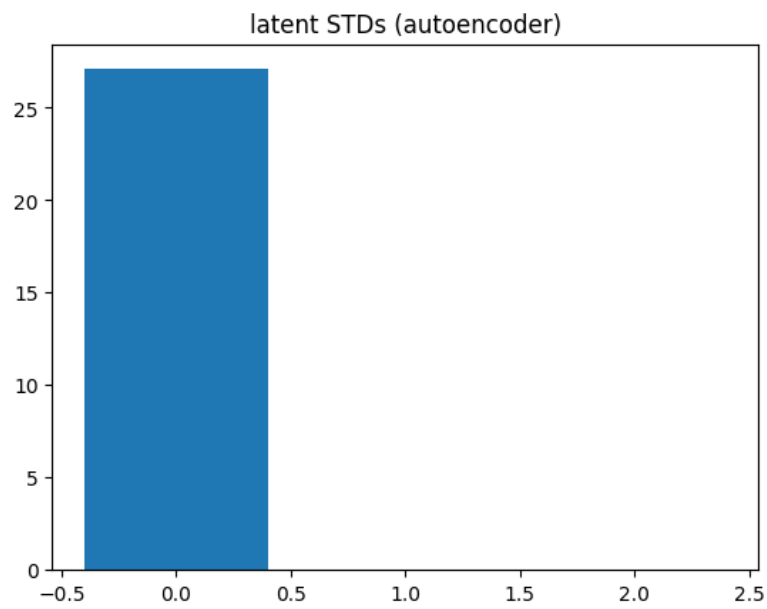
```

Epoch 999: rec = 0.015768, vol = 0.20888
Epoch 1999: rec = 0.0093937, vol = 0.2092

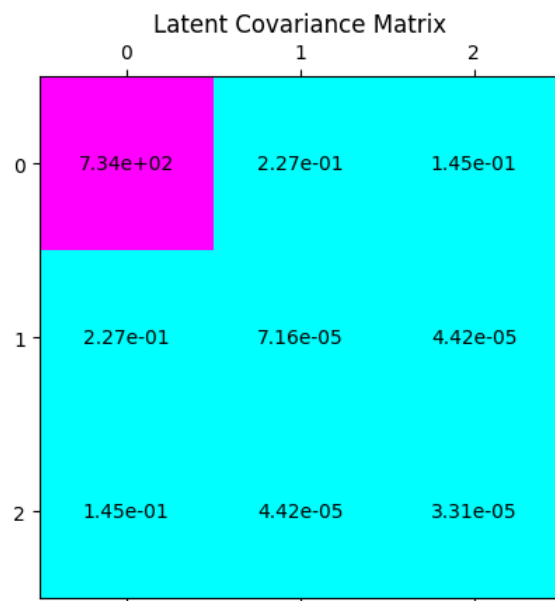
```

Epoch 2999: rec = 0.0075273, vol = 0.21866
 Epoch 3999: rec = 0.0061228, vol = 0.26539
 Epoch 4999: rec = 0.0049839, vol = 0.23421
 Epoch 5999: rec = 0.0042767, vol = 0.1931
 Epoch 6999: rec = 0.0035521, vol = 0.17556
 Epoch 7999: rec = 0.0034988, vol = 0.22726
 Epoch 8999: rec = 0.0025694, vol = 0.1861
 Epoch 9999: rec = 0.0028511, vol = 0.202
 Epoch 10999: rec = 0.0022147, vol = 0.17631
 Epoch 11999: rec = 0.0021955, vol = 0.17034
 Epoch 12999: rec = 0.0020662, vol = 0.19222
 Epoch 13999: rec = 0.0019016, vol = 0.20411
 Epoch 14999: rec = 0.0015431, vol = 0.15869
 Epoch 15999: rec = 0.0014341, vol = 0.18272
 Epoch 16999: rec = 0.0015622, vol = 0.18977
 Epoch 17999: rec = 0.0013786, vol = 0.17417
 Epoch 18999: rec = 0.0017483, vol = 0.15937
 Epoch 19999: rec = 0.0014537, vol = 0.20069

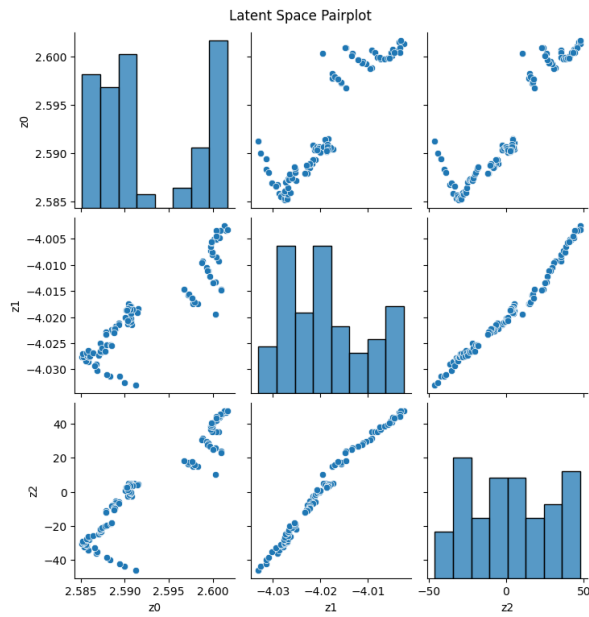




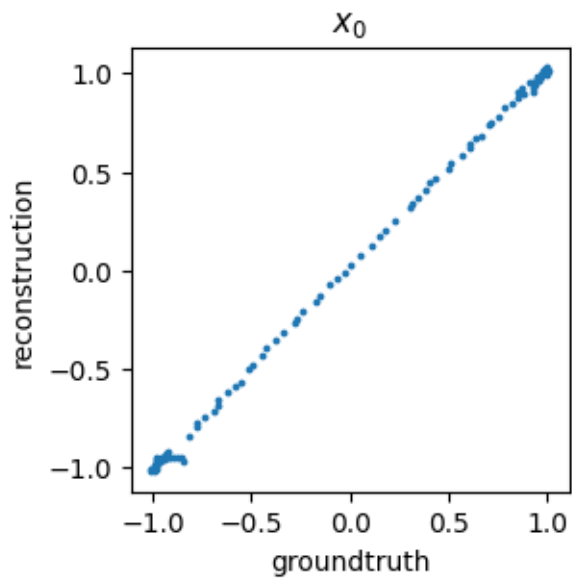
Plotting the latent code covariances:

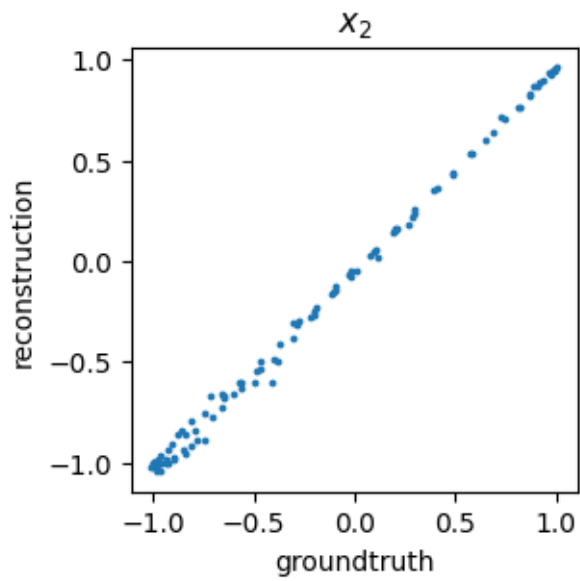
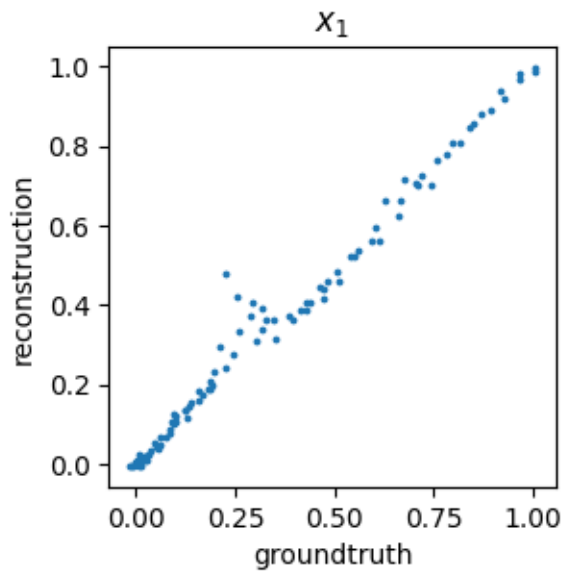


<Figure size 640x480 with 0 Axes>



Let's check again the reconstruction errors:





10 Introduction to Push-Forward Generative Models – Generative Adversarial Networks (GANs)

In this chapter, we will build geometric intuition for push-forward generative models, which are essentially trying to transport probability distributions from one space to another. We will start with simple linear maps to gain a sense of how they modify a simple 2D data space, then scale up to a small MLP GAN on a ring-of-Gaussians toy dataset. We will also talk about common metrics to evaluate generative models. Even though GANs may be comparatively simple compared to more advanced generative models that we will explore later, they will nevertheless be useful in building intuition for how push-forward generative models work.

Using device: cuda

10.1 What are Generative Models really doing?

Fundamentally, generative models are really just functions that transform a probability distribution from one space to another – you can think of them as “distribution transformers” or, move intuitively, as moving probability mass around in space. As an analogy, consider that you are sitting in a sandbox with a smooth mound of sand in front of you – the 2D Gaussian that we will use below is not too far from this, actually. A generative model is kind of like your hands, which you can use to push the sand around, creating hills and valleys, and moving the sand from one place to another. In this way, we are essentially moving the probability mass (sand, in this analogy) from a simple distribution (the smooth mound of sand) to a more complex distribution (the hills and valleys that you create with your hands). Different models that we will explore later (e.g., VAEs, normalizing flows, diffusion models) have different ways of doing this, but the core goal is the same, and they essentially all try to do one or more of three operations:

- $f(z) \rightarrow x$: Mapping points from one space (typically called the *latent space*) to another (typically, the *data space*).
- $f^{-1}(x) \rightarrow z$: Mapping points from data space back to latent space.
- $p(x) \Leftrightarrow p(z)$: Mapping probability densities from data space to latent space (and vice versa).

As we go forward, we will see how different types of models are better or worse at these different operations, and, in some cases, how they can be combined to create more powerful models.

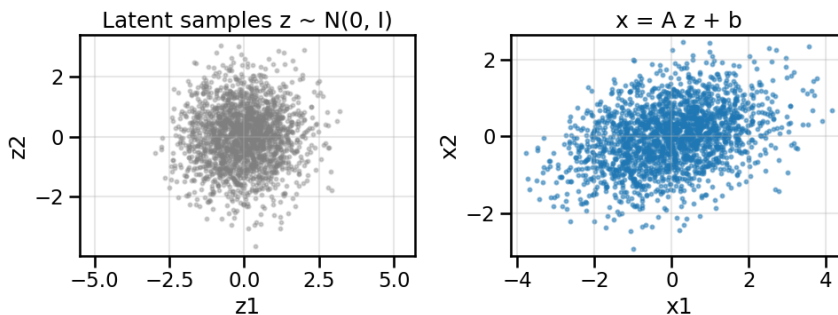
10.2 Basic Deterministic Push-Forward Models: From Latent Space to Data Space

A push-forward generative model defines a mapping $x = f(z)$, where z is sampled from a simple latent distribution (often a standard Gaussian, though it need not be) and f is a deterministic function (e.g., linear map or neural network).

We'll start with a 2D latent $z \sim N(0, I)$ (i.e., a standard Gaussian) and inspect how different choices of f reshape the distribution in data space. In this case, the latent space is 2D and the data space is also 2D, so we can visualize both spaces directly, however, in general, the latent space is often lower-dimensional than the data space.

```
# Linear push-forward demo with interactive sliders
def linear_pushforward_demo(A: np.ndarray, b: np.ndarray, n: int = 1000, seed: int = RNG_SEED):
    rng = np.random.default_rng(seed)
    z = rng.standard_normal(size=(n, 2))
    x = (z @ A.T) + b
    return z, x

# Default transform
A0 = np.array([[1.2, 0.4], [0.0, 0.8]], dtype=float)
b0 = np.array([0.0, 0.0], dtype=float)
z_lin, x_lin = linear_pushforward_demo(A0, b0, n=2000)
plot_pushforward(z_lin, x_lin, title='x = A z + b')
```

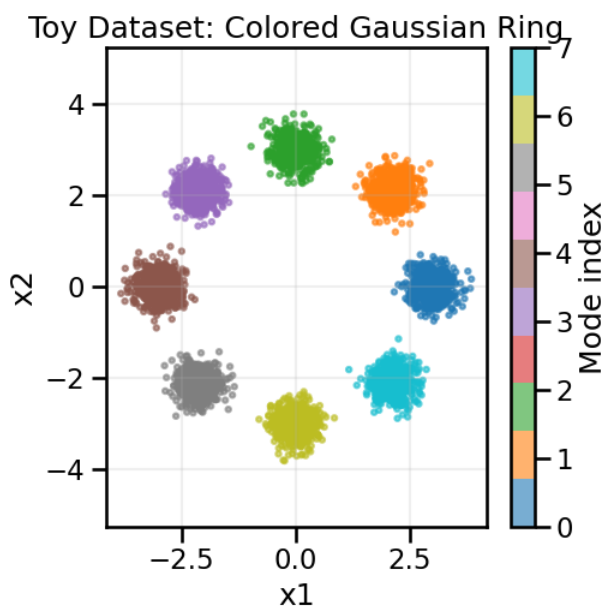


We can see from above that the simple linear map has slightly “moved” or shifted the location of the probability mass, just like if you were sculpting sand in your sandbox. You can play below with some of the sliders to manipulate the simple 2x2 weight matrix and 2x1 bias vector to see how this works interactively:

[Interactive output: available in the online version of this chapter.]

10.3 Toy Dataset: Ring of Gaussians

OK, now that we have some intuition for how simple linear maps can move probability mass around in space, let's try a more complex example where it is not so clear that a simple linear map will be sufficient. Below, we will create a common toy dataset consisting of a “ring of Gaussians” – that is, several Gaussian blobs arranged in a circle. You can see from above that no combination of weights in a simple linear map will be able to move our original probability mass from a standard Gaussian to this ring of Gaussians, so we will need something more powerful. A simple starting point for this is something called a Generative Adversarial Network (GAN), which we will explore next.



10.4 What is a Generative Adversarial Network (GAN)?

A Generative Adversarial Network (GAN) is a framework for training generative models using two competing neural networks: a **generator** and a **discriminator**. The generator, G , learns to map random noise from a latent space (e.g., $z \sim p_z(z)$ or $f(z) = x$, where G functions as f here) to data space, aiming to produce samples that resemble the real data. It is tempting to want to train just the Generator by minimizing Mean Squared Error between z and $f(z) = x$, except we have one big problem for right now – we do not know apriori which samples of

x correspond to which samples of z , and so it is not straightforward to compute this MSE.¹ Instead, what we will do is train a separate network, the Discriminator, that can help us push the samples produced by the Generator closer to the real data distribution.

Specifically, the discriminator, D , tries to distinguish between real data samples and those produced by the generator and our loss function will encourage the discriminator to become better and better at this task. In turn, the generator will be trained to produce samples that the discriminator classifies as real – that is to try to fool the discriminator.

The two networks are trained simultaneously in something called a *minimax game*:

- The **generator** tries to “fool” the discriminator by generating realistic samples.
- The **discriminator** tries to correctly classify real vs. generated samples.

10.4.1 Mathematical Formulation

The standard GAN objective, as introduced by Goodfellow et al. (2014), is:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

- $p_{\text{data}}(x)$: Distribution of real data.
- $p_z(z)$: Prior distribution over latent variables (often standard normal).
- $G(z)$: Generator’s output given latent input z .
- $D(x)$: Discriminator’s estimate of the probability that x is real.

The generator and discriminator are typically neural networks trained with stochastic gradient descent. The generator improves by producing samples that the discriminator cannot distinguish from real data (the second term on the right-hand side of the equation above), while the discriminator improves by getting better at distinguishing real from fake (the first term on the right-hand side of the equation above).

10.5 A Simple Generative Adversarial Network (GAN)

Now we will define a small GAN and see how it moves probability mass around after passing samples through the Generator. You can feel free to modify the architecture below if you like and see how that impacts the below results, but for now we will use a single hidden layer MLP with LeakyReLU activation for both the Generator and Discriminator.

¹Indeed, we will return to this idea in later notebooks once we have introduced the concept of Optimal Transport, which solves this mapping problem, but for now, let’s assume that we don’t know how to do this.

```

class MLPGenerator(nn.Module):
    def __init__(self, noise_dim: int = 2, hidden_dim: int = 256, out_dim: int = 2):
        super().__init__()
        self.main = nn.Sequential(
            nn.Linear(noise_dim, hidden_dim), nn.LeakyReLU(),
            nn.Linear(hidden_dim, hidden_dim), nn.LeakyReLU(),
            nn.Linear(hidden_dim, out_dim),
        )

    def forward(self, z: torch.Tensor) -> torch.Tensor:
        return self.main(z)

class MLPDiscriminator(nn.Module):
    def __init__(self, input_dim: int = 2, hidden_dim: int = 256):
        super().__init__()
        self.main = nn.Sequential(
            nn.Linear(input_dim, hidden_dim), nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, hidden_dim), nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, 1),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.main(x).squeeze(-1)

def build_generator(noise_dim=2, hidden_dim=256):
    return MLPGenerator(noise_dim=noise_dim, hidden_dim=hidden_dim).to(device)

def build_discriminator(hidden_dim=256):
    return MLPDiscriminator(hidden_dim=hidden_dim).to(device)

```

10.6 Basic Training Loop

With the models defined, we will now set up the training loop to optimize both the generator and discriminator.

```

def train_vanilla_gan(
    data: np.ndarray, *, noise_dim: int = 2, batch_size: int = 256, epochs: int = 120,
    lr_g: float = 2e-4, lr_d: float = 2e-4, hidden_dim: int = 256, print_every: int = 40) -> tuple[nn.Module,
    # Load the data into a DataLoader for batching and make PyTorch happy
    loader = make_loader(data, batch_size)

    # Set up the basic networks
    G = build_generator(noise_dim=noise_dim, hidden_dim=hidden_dim)

```

```

D = build_discriminator(hidden_dim=hidden_dim)
# Instantiate the optimizers for each model
opt_g = optim.Adam(G.parameters(), lr=lr_g, betas=(0.5, 0.999))
opt_d = optim.Adam(D.parameters(), lr=lr_d, betas=(0.5, 0.999))
bce = nn.BCEWithLogitsLoss()
# Record the loss history for plotting later
hist = GanHistory([], [], [], [], [])

# Now we do the training loop for # epochs defined in `epochs`
for ep in range(epochs):
    d_losses=[]
    g_losses=[]
    real_scores=[]
    fake_scores=[]
    for (xb,) in loader:
        # Send the data to the GPU, if using.
        xb = xb.to(device)

        # Take a Discriminator step
        z = torch.randn(xb.size(0), noise_dim, device=device)
        with torch.no_grad():
            x_fake = G(z)
        opt_d.zero_grad()
        d_real = D(xb)
        d_fake = D(x_fake)
        loss_d = bce(d_real, torch.ones_like(d_real)) + bce(d_fake, torch.zeros_like(d_fake))
        loss_d.backward()
        opt_d.step()
        d_losses.append(float(loss_d.detach().cpu().item()))
        real_scores.append(d_real.mean().item())
        fake_scores.append(d_fake.mean().item())

    # Take a Generator step
    z = torch.randn(xb.size(0), noise_dim, device=device)
    opt_g.zero_grad()
    xg = G(z)
    dg = D(xg)
    loss_g = bce(dg, torch.ones_like(dg))
    loss_g.backward()
    opt_g.step()
    g_losses.append(float(loss_g.detach().cpu().item()))

```

```

# We'll record some epoch metrics
with torch.no_grad():
    z_eval = torch.randn(2048, noise_dim, device=device)
    samples = G(z_eval)
    div = compute_diversity_metric(samples)

# Now we'll record the metrics for plotting later and reporting
hist.d_loss.append(float(np.mean(d_losses)))
hist.g_loss.append(float(np.mean(g_losses)))
hist.diversity.append(div)
hist.real_scores.append(float(np.mean(real_scores)))
hist.fake_scores.append(float(np.mean(fake_scores)))
if (ep+1) % max(1, print_every) == 0 or ep==0:
    print(f"Epoch {ep+1:03d}/{epochs} | D {hist.d_loss[-1]:.3f} | G {hist.g_loss[-1]:.3f} | Div {div:.3f}")
return G, D, hist

```

You can feel free to modify some of the training elements here, such as the epoch length or learning rates of the Generator or Discriminator, respectively.

```

G, D, H = train_vanilla_gan(
    X_ring, epochs=120, batch_size=256,
    lr_g=2e-4, lr_d=2e-4,
    hidden_dim=256, noise_dim=2,
    print_every=40)

```

```

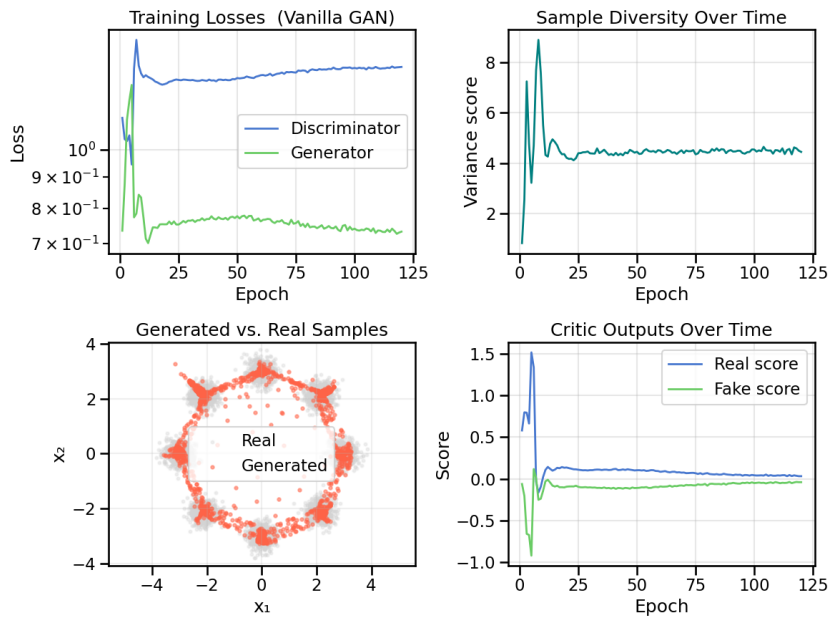
Epoch 001/120 | D 1.128 | G 0.734 | Div 0.809
Epoch 040/120 | D 1.300 | G 0.769 | Div 4.310
Epoch 080/120 | D 1.340 | G 0.754 | Div 4.409
Epoch 120/120 | D 1.368 | G 0.732 | Div 4.444

```

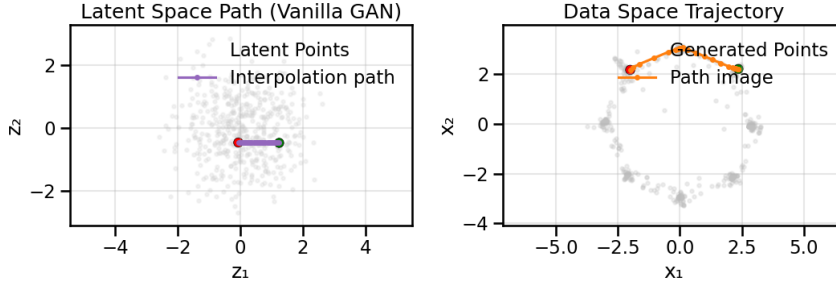
OK, now the model is trained, so let's look at some basic visualizations of how it did. In the below plots, which we will re-use for other models later on, we will show four things:

- Upper Left: Any training losses for the model, as a function of training epoch. In the case of a GAN, this will plot both the Generator and Discriminator losses. This plot type allows us to assess something about the convergence and stability of training.
- Upper Right: The Sample Diversity, as a function of Epoch. This is computing and plotting the variance of $\mathbf{x}$ as a function of epoch, where $\mathbf{x}$ are samples drawn from the Generator at each epoch. This plot type allows us one way to assess whether the model is suffering from mode collapse (i.e., low diversity in the generated samples) or not.

- Lower Right: A plot of the real data samples (in light grey) and samples drawn from the Generator (in reddish-orange). This plot type allows us to visually assess how well the model is capturing the data distribution. This will be easy to compare in this simple 2D case, but will be harder in higher dimensions, and in those cases you might have to resort to just comparing selected samples or overall distribution summary statistics.
- Lower Right: This plot will vary depending on the specific model we are studying, but in this case it will show the specific scores of the Discriminator (often called the “Critic”) with respect to its classification accuracy on real vs. fake samples. In the context of a GAN model, this allows us to assess how well a Generator is fooling the discriminator, since a well-trained Generator should produce samples that the Discriminator classifies as real, and thus both the “Real Score” and “Fake Score” should have around 50% accuracy. (i.e., the Discriminator is effectively guessing randomly).



In addition to the above training diagnostics, it can also be useful to plot interpolations in latent space to see how smoothly the Generator can move between different modes of the data distribution. In the case of a well-trained GAN, we would expect that interpolating between two points in latent space should produce a smooth transition in data space, moving through intermediate samples that also look realistic. In this case, this means hopping smoothly between the different Gaussian blobs in our ring-of-Gaussians dataset, and not jumping back-and-forth randomly between them.



To give you some interactive control on visualizing the forward mapping $f(z) \rightarrow x$, use the sliders below to move a single latent vector (left-side plot) and observe the corresponding generated point in data space (right-side plot). By moving around in z you can try to align the generated point with a particular mode on the ring.

[Interactive output: available in the online version of this chapter.]

10.7 Conditional Generation Examples

So far, we’ve explored unconditional generative models that learn to produce samples from the overall data distribution. However, in many practical applications, we want to control what we generate by conditioning on additional information. For example, we might want to generate samples from a specific mode of the distribution, or generate samples with particular properties.

In this section, we’ll explore two types of conditioning:

1. **Discrete Conditioning (One-Hot Mode Index):** Condition the GAN on which Gaussian mode in our ring dataset the sample should come from.
2. **Continuous Conditioning (Radius):** Condition the GAN on a continuous scalar representing the radius from the origin.

These conditioning strategies are widely applicable and form the basis for more complex conditional generation tasks, such as class-conditional image generation or text-to-image synthesis.

10.7.1 Conditional Example A — One-Hot Mode Conditioning

In this example, we’ll condition the GAN on which of the 8 Gaussian modes in the ring dataset we want to generate from. We provide a one-hot encoded vector indicating the target mode to both the generator and discriminator.

The key idea is simple: we concatenate the one-hot label to the inputs of both networks. The generator receives both random noise z and the label y , learning to map $(z, y) \rightarrow x$. The

discriminator receives both the data point x and the label y , learning to distinguish real pairs (x, y) from fake pairs $(G(z, y), y)$.

```
# Conditional Generator: takes noise + one-hot label
class ConditionalMLPGenerator(nn.Module):
    def __init__(self, noise_dim: int = 2, label_dim: int = 8, hidden_dim: int = 256, out_dim: int = 2):
        super().__init__()
        self.main = nn.Sequential(
            nn.Linear(noise_dim + label_dim, hidden_dim), nn.LeakyReLU(),
            nn.Linear(hidden_dim, hidden_dim), nn.LeakyReLU(),
            nn.Linear(hidden_dim, out_dim),
        )
    def forward(self, z: torch.Tensor, y: torch.Tensor) -> torch.Tensor:
        # Concatenate noise and label
        zy = torch.cat([z, y], dim=1)
        return self.main(zy)

# Conditional Discriminator: takes data + one-hot label
class ConditionalMLPDiscriminator(nn.Module):
    def __init__(self, input_dim: int = 2, label_dim: int = 8, hidden_dim: int = 256):
        super().__init__()
        self.main = nn.Sequential(
            nn.Linear(input_dim + label_dim, hidden_dim), nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, hidden_dim), nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, 1),
        )
    def forward(self, x: torch.Tensor, y: torch.Tensor) -> torch.Tensor:
        xy = torch.cat([x, y], dim=1)
        return self.main(xy).squeeze(-1)

def train_conditional_gan_onehot(
    data: np.ndarray, labels: np.ndarray, n_classes: int = 8, *,
    noise_dim: int = 2, batch_size: int = 256, epochs: int = 120,
    lr_g: float = 2e-4, lr_d: float = 2e-4, hidden_dim: int = 256, print_every: int = 40
) -> tuple[nn.Module, nn.Module, GanHistory]:
    loader = make_conditional_loader_onehot(data, labels, batch_size, n_classes)
    G = ConditionalMLPGenerator(noise_dim=noise_dim, label_dim=n_classes, hidden_dim=hidden_dim).to(device)
    D = ConditionalMLPDiscriminator(input_dim=2, label_dim=n_classes, hidden_dim=hidden_dim).to(device)
    opt_g = optim.Adam(G.parameters(), lr=lr_g, betas=(0.5, 0.999))
    opt_d = optim.Adam(D.parameters(), lr=lr_d, betas=(0.5, 0.999))
    bce = nn.BCEWithLogitsLoss()
    hist = GanHistory([], [], [], [], [])
```



```

for ep in range(epochs):
    d_losses, g_losses, real_scores, fake_scores = [], [], [], []
    for xb, yb in loader:
        xb, yb = xb.to(device), yb.to(device)
        # Discriminator step
        z = torch.randn(xb.size(0), noise_dim, device=device)
        with torch.no_grad():
            x_fake = G(z, yb)
        opt_d.zero_grad()
        d_real = D(xb, yb)
        d_fake = D(x_fake, yb)
    loss_d = bce(d_real, torch.ones_like(d_real)) + bce(d_fake, torch.zeros_like(d_fake))
    loss_d.backward()
    opt_d.step()
    d_losses.append(float(loss_d.detach().cpu().item()))
    real_scores.append(d_real.mean().item())
    fake_scores.append(d_fake.mean().item())

    # Generator step
    z = torch.randn(xb.size(0), noise_dim, device=device)
    opt_g.zero_grad()
    xg = G(z, yb)
    dg = D(xg, yb)
    loss_g = bce(dg, torch.ones_like(dg))
    loss_g.backward()
    opt_g.step()
    g_losses.append(float(loss_g.detach().cpu().item()))

    with torch.no_grad():
        z_eval = torch.randn(2048, noise_dim, device=device)
        y_eval = torch.nn.functional.one_hot(
            torch.randint(0, n_classes, (2048,), device=device), num_classes=n_classes
        ).to(torch.float32)
        samples = G(z_eval, y_eval)
        div = compute_diversity_metric(samples)

    hist.d_loss.append(float(np.mean(d_losses)))
    hist.g_loss.append(float(np.mean(g_losses)))
    hist.diversity.append(div)
    hist.real_scores.append(float(np.mean(real_scores)))
    hist.fake_scores.append(float(np.mean(fake_scores)))
    if (ep+1) % max(1, print_every) == 0 or ep==0:

```

```

        print(f"Epoch {ep+1:03d}/{epochs} | D {hist.d_loss[-1]:.3f} | G {hist.g_loss[-1]:.3f} | Div {div:}
    return G, D, hist

# Train the conditional GAN
G_cond_onehot, D_cond_onehot, H_cond_onehot = train_conditional_gan_onehot(
    X_ring, y_ring, n_classes=8, epochs=480, batch_size=256,
    lr_g=2e-4, lr_d=2e-4, hidden_dim=256, noise_dim=2, print_every=40
)

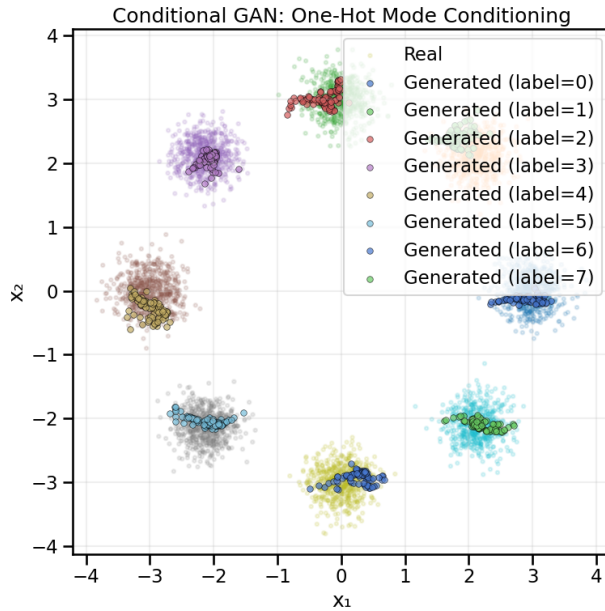
```

```

Epoch 001/480 | D 1.250 | G 0.706 | Div 0.029
Epoch 040/480 | D 1.399 | G 0.691 | Div 4.734
Epoch 080/480 | D 1.386 | G 0.697 | Div 4.626
Epoch 120/480 | D 1.387 | G 0.689 | Div 4.322
Epoch 160/480 | D 1.385 | G 0.707 | Div 4.171
Epoch 200/480 | D 1.401 | G 0.658 | Div 3.566
Epoch 240/480 | D 1.390 | G 0.710 | Div 4.966
Epoch 280/480 | D 1.387 | G 0.702 | Div 4.233
Epoch 320/480 | D 1.384 | G 0.708 | Div 4.220
Epoch 360/480 | D 1.386 | G 0.680 | Div 4.759
Epoch 400/480 | D 1.388 | G 0.699 | Div 4.590
Epoch 440/480 | D 1.387 | G 0.682 | Div 4.703
Epoch 480/480 | D 1.388 | G 0.700 | Div 4.429

```

Now let's visualize the conditional samples. We'll generate samples for each of the 8 modes and plot them to see if the generator has learned to respect the conditioning signal.



10.7.2 Conditional Example B — Continuous Radius Conditioning

Instead of conditioning on discrete labels, we can also condition on continuous scalar values. In this example, we'll condition the GAN on the radius (distance from the origin) of the generated samples. This allows us to control where on the ring samples are generated.

To handle continuous conditioning, we typically embed the scalar through a small MLP before concatenating it with other inputs. This gives the network more flexibility in how it uses the conditioning signal.

```
from gen_models_utilities import make_conditional_loader_radius, plot_conditional_samples_continuous

# Conditional Generator with radius embedding
class ConditionalMLPGeneratorRadius(nn.Module):
    def __init__(self, noise_dim: int = 2, cond_emb_dim: int = 16, hidden_dim: int = 256, out_dim: int = 2):
        super().__init__()
        # Embed the scalar radius into a higher-dimensional space
        self.cond_embed = nn.Sequential(
            nn.Linear(1, cond_emb_dim),
            nn.LeakyReLU(),
            nn.Linear(cond_emb_dim, cond_emb_dim),
        )
        self.main = nn.Sequential(
            nn.Linear(noise_dim + cond_emb_dim, hidden_dim), nn.LeakyReLU(),
```

```

        nn.Linear(hidden_dim, hidden_dim), nn.LeakyReLU(),
        nn.Linear(hidden_dim, out_dim),
    )
    def forward(self, z: torch.Tensor, r: torch.Tensor) -> torch.Tensor:
        r_emb = self.cond_embed(r)
        zr = torch.cat([z, r_emb], dim=1)
        return self.main(zr)

# Conditional Discriminator with radius embedding
class ConditionalMLPDiscriminatorRadius(nn.Module):
    def __init__(self, input_dim: int = 2, cond_emb_dim: int = 16, hidden_dim: int = 256):
        super().__init__()
        self.cond_embed = nn.Sequential(
            nn.Linear(1, cond_emb_dim),
            nn.LeakyReLU(),
            nn.Linear(cond_emb_dim, cond_emb_dim),
        )
        self.main = nn.Sequential(
            nn.Linear(input_dim + cond_emb_dim, hidden_dim), nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, hidden_dim), nn.LeakyReLU(0.2),
            nn.Linear(hidden_dim, 1),
        )
    def forward(self, x: torch.Tensor, r: torch.Tensor) -> torch.Tensor:
        r_emb = self.cond_embed(r)
        xr = torch.cat([x, r_emb], dim=1)
        return self.main(xr).squeeze(-1)

def train_conditional_gan_radius(
    data: np.ndarray, *,
    noise_dim: int = 2, batch_size: int = 256, epochs: int = 120,
    lr_g: float = 2e-4, lr_d: float = 2e-4, hidden_dim: int = 256, print_every: int = 40
) -> tuple[nn.Module, nn.Module, GanHistory]:
    loader = make_conditional_loader_radius(data, batch_size)
    G = ConditionalMLPGeneratorRadius(noise_dim=noise_dim, hidden_dim=hidden_dim).to(device)
    D = ConditionalMLPDiscriminatorRadius(input_dim=2, hidden_dim=hidden_dim).to(device)
    opt_g = optim.Adam(G.parameters(), lr=lr_g, betas=(0.5, 0.999))
    opt_d = optim.Adam(D.parameters(), lr=lr_d, betas=(0.5, 0.999))
    bce = nn.BCEWithLogitsLoss()
    hist = GanHistory([], [], [], [], [])

    for ep in range(epochs):
        d_losses, g_losses, real_scores, fake_scores = [], [], [], []

```

```

for xb, rb in loader:
    xb, rb = xb.to(device), rb.to(device)
    # Discriminator step
    z = torch.randn(xb.size(0), noise_dim, device=device)
    with torch.no_grad():
        x_fake = G(z, rb)
    opt_d.zero_grad()
    d_real = D(xb, rb)
    d_fake = D(x_fake, rb)
    loss_d = bce(d_real, torch.ones_like(d_real)) + bce(d_fake, torch.zeros_like(d_fake))
    loss_d.backward()
    opt_d.step()
    d_losses.append(float(loss_d.detach().cpu().item()))
    real_scores.append(d_real.mean().item())
    fake_scores.append(d_fake.mean().item())

    # Generator step
    z = torch.randn(xb.size(0), noise_dim, device=device)
    opt_g.zero_grad()
    xg = G(z, rb)
    dg = D(xg, rb)
    loss_g = bce(dg, torch.ones_like(dg))
    loss_g.backward()
    opt_g.step()
    g_losses.append(float(loss_g.detach().cpu().item()))

    with torch.no_grad():
        z_eval = torch.randn(2048, noise_dim, device=device)
        r_eval = torch.norm(torch.randn(2048, 2), dim=1, keepdim=True).to(device)
        samples = G(z_eval, r_eval)
        div = compute_diversity_metric(samples)

    hist.d_loss.append(float(np.mean(d_losses)))
    hist.g_loss.append(float(np.mean(g_losses)))
    hist.diversity.append(div)
    hist.real_scores.append(float(np.mean(real_scores)))
    hist.fake_scores.append(float(np.mean(fake_scores)))
    if (ep+1) % max(1, print_every) == 0 or ep==0:
        print(f"Epoch {ep+1:03d}/{epochs} | D {hist.d_loss[-1]:.3f} | G {hist.g_loss[-1]:.3f} | Div {div:.3f}")
return G, D, hist

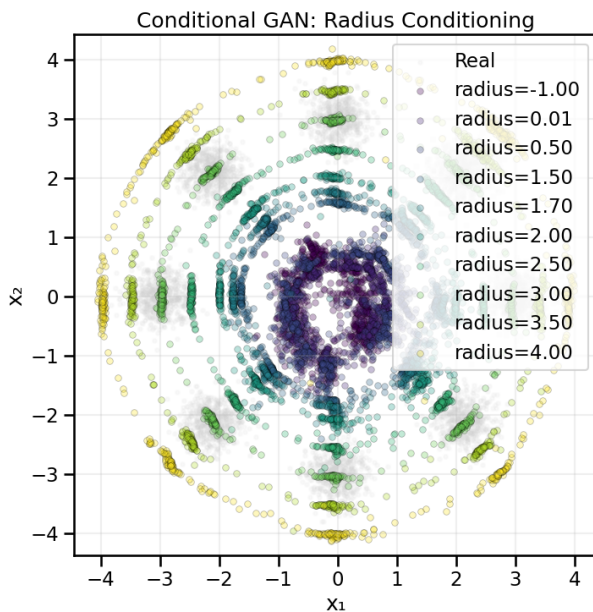
# Train the conditional GAN with radius conditioning

```

```
G_cond_radius, D_cond_radius, H_cond_radius = train_conditional_gan_radius(
    X_ring, epochs=480, batch_size=256,
    lr_g=2e-4, lr_d=2e-4, hidden_dim=256, noise_dim=2, print_every=40
)
```

```
Epoch 001/480 | D 1.295 | G 0.732 | Div 0.012
Epoch 040/480 | D 1.100 | G 1.071 | Div 0.335
Epoch 080/480 | D 1.265 | G 0.834 | Div 3.910
Epoch 120/480 | D 1.242 | G 0.807 | Div 1.787
Epoch 160/480 | D 1.284 | G 0.801 | Div 1.194
Epoch 200/480 | D 1.323 | G 0.774 | Div 1.310
Epoch 240/480 | D 1.341 | G 0.758 | Div 1.311
Epoch 280/480 | D 1.351 | G 0.736 | Div 1.299
Epoch 320/480 | D 1.355 | G 0.731 | Div 1.255
Epoch 360/480 | D 1.367 | G 0.729 | Div 1.294
Epoch 400/480 | D 1.362 | G 0.722 | Div 1.245
Epoch 440/480 | D 1.359 | G 0.721 | Div 1.224
Epoch 480/480 | D 1.365 | G 0.725 | Div 1.226
```

Now let's generate samples at different radius values to see how the generator responds to continuous conditioning. We'll sweep through several radius values and visualize the results.



10.8 Summary and Next Steps:

OK, so we have seen how a basic GAN is set up and trained, and covered some diagnostic plots that give us insight into how the training went. This gave us the following sets of useful tools:

- We could learn a forward mapping function $f(z) \rightarrow x$ that could push forward a given sample in z to a corresponding sample in x . This could learn distributions that were far more complex than a simple linear map could achieve above.
- This f is a deterministic function – i.e., the same z will always produce the same x .
- We could produce latent interpolations in z that produced smooth transitions in x and seemed to capture the overall clustering behavior of certain distributions.

Unfortunately, GANs did not give us any of the following properties, which will turn out to be useful later and motivate other types of generative models:

- We did not learn an inverse mapping $f^{-1}(x) \rightarrow z$. This means that we cannot easily encode real data samples into the latent space, which would be useful for many applications. For example, if I wanted to slightly modify x – e.g., $x' = x + \delta x$ – and compute the corresponding latent coordinate $z' = z + \delta z$, GANS do not provide a built-in mechanism to do this.² We will see how both Variational Autoencoders (VAEs) and Normalizing Flows (NFs) address this problem.
- We also cannot yet easily talk about or compute probability mappings between $p(x)$ and $p(z)$, which would be useful for computing likelihoods of data samples or for evaluating how well the model fits the data distribution, both of which are quite useful. Again, we will see later how both VAEs and NFs address this problem.

Before we move beyond GANs, however, we will first explore some of the complex training dynamics and other subtle issues that arise when applying them in practice. This will help you build up your intuition and experience in how to assess problems in generative models and will serve as a strong basis upon which to launch into more complex models later. It will also give us an opportunity to introduce the concept of **Optimal Transport**, which will be useful in its own right applications later on, and will also help us understand some of the limitations of GANs should you encounter their use in the future.

²Of course, I could try to get at this another way, for example, by applying automatic differentiation to the Generator to compute $\frac{\partial f}{\partial z}$ and then use that to estimate $\delta z \approx (\frac{\partial f}{\partial z})^{-1} \delta x$, but this requires extra work on our part and this is not always straightforward, especially if the Jacobian is ill-conditioned.

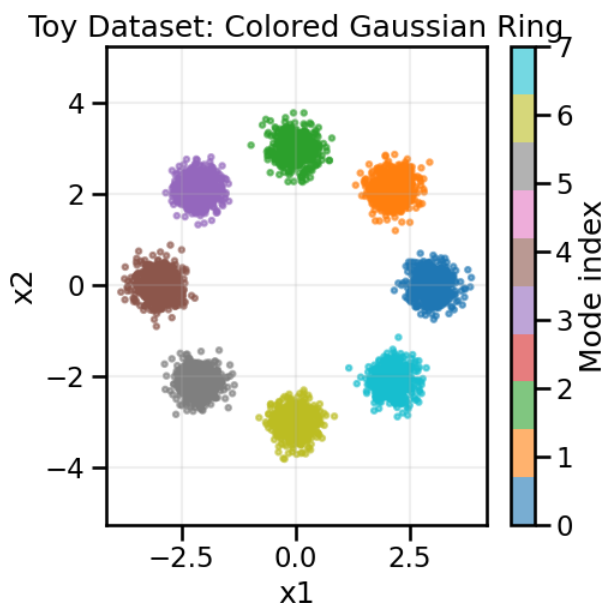
11 GAN Training Pitfalls

Using device: cuda

This notebook will highlight some of the original problems with GAN models, some of the techniques you can use to diagnose model errors, and also introduce common remedies for simple feed-forward GAN models. The next notebook will introduce the concept of Optimal Transport, and show how this mitigates some of the problems caused by the original GAN formulation as a minimax game.

As with the last notebook, we will use the simple Ring of Gaussian Toy example to illustrate this for pedagogical purposes. In later notebooks, we will see how these problems can manifest in more complex settings.

Let's start by loading the dataset again:



We will now run a series of experiments to show you common problems and how they manifest in the plotting visualizations.

11.1 Experiment 1: When the Discriminator Overpowers the Generator

In this experiment, we will see what happens with the Discriminator part of the GAN significantly overpowers the Generator. This can occur because of several reasons:

1. The Discriminator is a much larger (or higher capacity) network than the Generator, and thus is able to move its classification decision boundary in more complex ways to distinguish real from fake samples, and the Generator doesn't have sufficient capacity to compensate for this.
2. The Discriminator trains much faster than the Generator, and thus can move its decision boundary faster than the Generator can keep up.
3. The Discriminator is initialized (either on purpose or by random chance) such that it has an initial significant advantage over the Generator. In this case, even if the Generator has high capacity and can train quickly, it may not be able to recover from this initial disadvantage, as the discriminator make lock it into a portion of the sample space from which local gradient descent on the Generator cannot recover.

We can induce this below by just significantly increasing the learning rate of the Discriminator relative to the Generator. Consider the following questions and experiment:

💡 Experiment: What happens when the Discriminator overpowers the Generator?

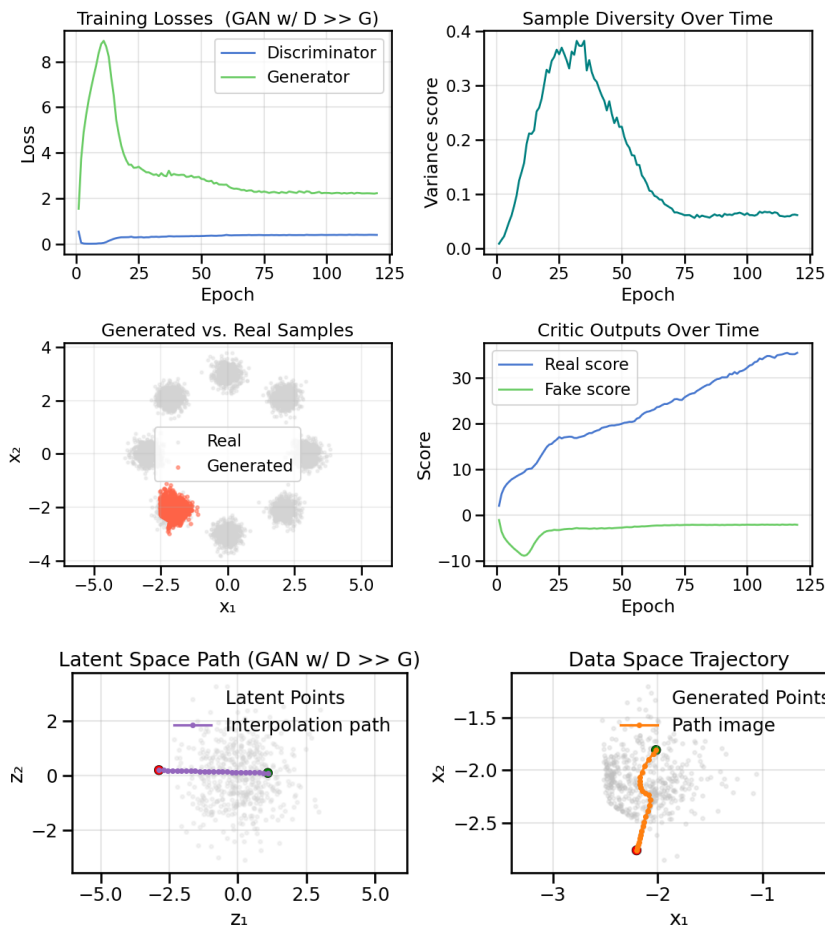
Below you can change the relative learning rates of the Discriminator versus the Generator:

- What happens when the Discriminator is able to train significantly faster than the Generator? What about the other way around?
- How can we tell from the various loss plots that the Discriminator is overpowering the Generator? How does this behavior manifest itself in the generated samples?
- Why is it that the generator is unable to recover from this situation, even though it has the capacity to represent the data distribution? What about the training procedure and dynamics of the minimax game prevents the Generator from catching up?
- In the latent interpolation plots, as the GAN spans its latent set of coordinates, what happens to the generated samples?

```
G, D, H = train_vanilla_gan(  
    X_ring, epochs=120, batch_size=256,  
    lr_g=1e-5,  
    lr_d=1e-3,  
    hidden_dim=256, noise_dim=2,
```

```
checkpoint_interval=10,  
print_every=40)
```

```
Epoch 001/120 | D 0.537 | G 1.542 | Div 0.009  
Epoch 040/120 | D 0.331 | G 3.036 | Div 0.307  
Epoch 040/120 | D 0.331 | G 3.036 | Div 0.307  
Epoch 080/120 | D 0.388 | G 2.256 | Div 0.062  
Epoch 080/120 | D 0.388 | G 2.256 | Div 0.062  
Epoch 120/120 | D 0.394 | G 2.226 | Div 0.062  
Epoch 120/120 | D 0.394 | G 2.226 | Div 0.062
```



We can also take a look at this interactively, by viewing various snapshots during training:

[Interactive output: available in the online version of this chapter.]

11.2 Experiment 2: When the Generator Overpowers the Discriminator

Above we saw what happens when the Discriminator overpowers the Generator. But what about the other way around? The next experiment will help illuminate what can go wrong.

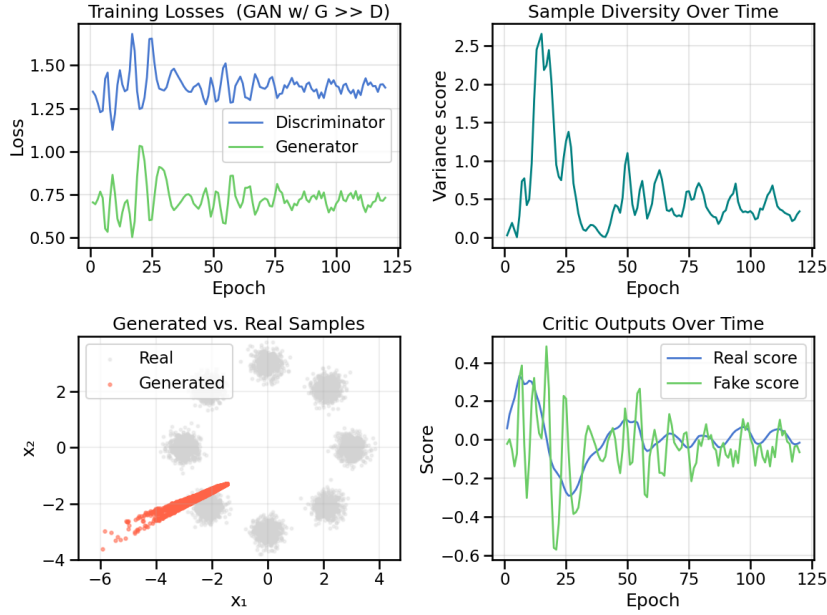
💡 Experiment: What happens when the Generator overpowers the Discriminator?

Below you can change the relative learning rates of the Discriminator versus the Generator to give the Generator a significant advantage:

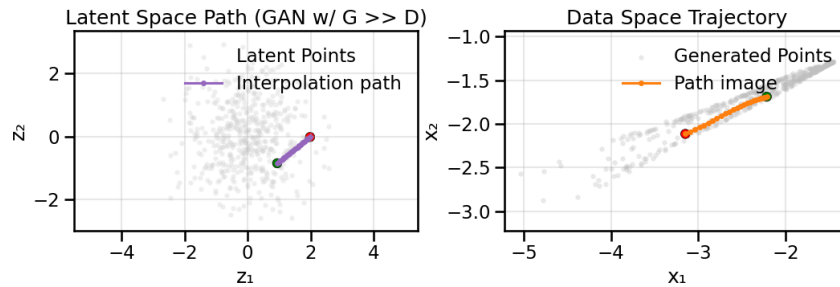
- What happens when we turn the tables on the Discriminator, and give the Generator comparatively more power?
- How are the loss plots this time significantly different than what you saw in the first experiment?
- As you look through the training curves as well as snapshots during training, what do you notice about the generated samples? Why would an underpowered Discriminator lead to this behavior? (Hint: Play out conceptually in your head how the generator would respond in cases where the Discriminator is very weak or slow.)

```
G, D, H = train_vanilla_gan(
    X_ring, epochs=120, batch_size=256,
    lr_g=1e-4,
    lr_d=1e-5,
    hidden_dim=256, noise_dim=2,
    checkpoint_interval=10,
    print_every=40)
```

```
Epoch 001/120 | D 1.347 | G 0.704 | Div 0.030
Epoch 040/120 | D 1.346 | G 0.751 | Div 0.015
Epoch 040/120 | D 1.346 | G 0.751 | Div 0.015
Epoch 080/120 | D 1.368 | G 0.716 | Div 0.650
Epoch 080/120 | D 1.368 | G 0.716 | Div 0.650
Epoch 120/120 | D 1.370 | G 0.731 | Div 0.343
Epoch 120/120 | D 1.370 | G 0.731 | Div 0.343
```



Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
 Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
 Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.



You can also view this interactively at different training snapshots below, which might help illuminate some behavior:

[Interactive output: available in the online version of this chapter.]

11.3 Experiment 3: Effect of Discriminator Spectral Normalization

Part of the issue in Experiment 1 (where the Discriminator overpowers the Generator) is that the Discriminator can make very sudden and extreme changes to its decision boundary, which the Generator cannot keep up with. One way to mitigate this is to use Spectral Normalization

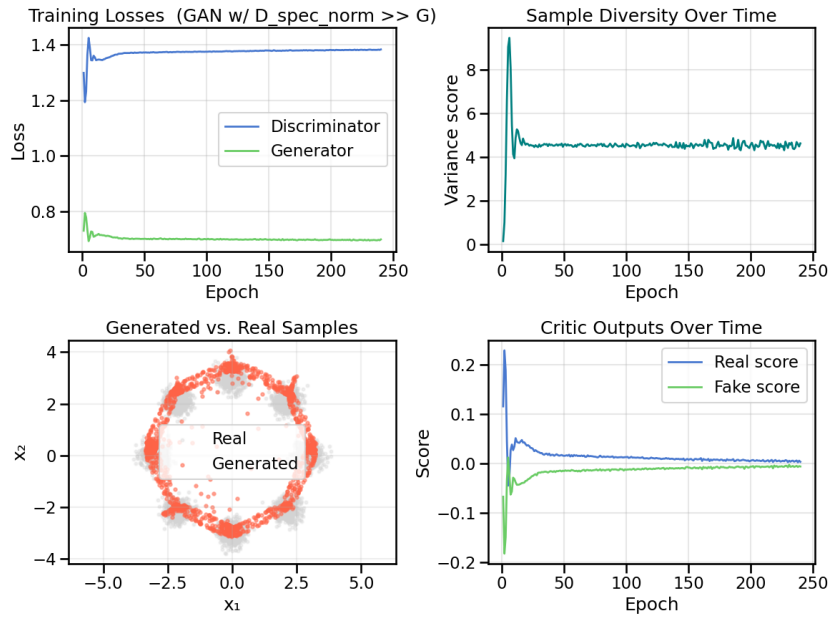
on the Discriminator weights, which constrains the Lipschitz constant of the Discriminator function, and thus prevents it from making extreme changes to its decision boundary.

💡 Experiment: How does Spectral Normalization affect the Discriminator?

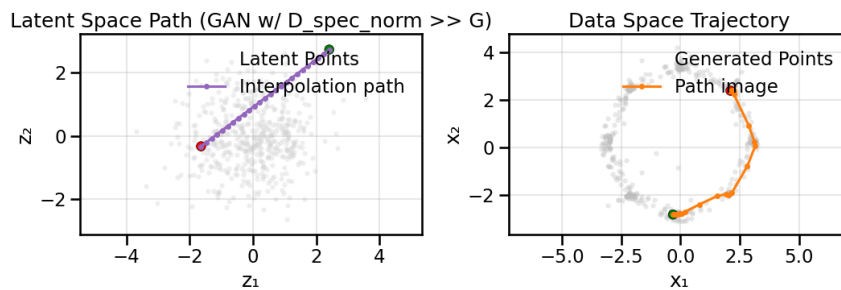
Below we will activate `spectral_normalization=True` in the Discriminator, and see how this affects the training dynamics when the Discriminator is given a significant learning rate advantage over the Generator.

- Keeping the learning rates from Experiment 1, what do you now notice about the Discriminator's training losses?
- Do different learning rates modulate this effect? You can repeat a version of Experiment 2 here if you are curious.
- In what ways has applying this spectral normalization improved the model performance? In what ways has it made it worse? Why do you think this is? (Hint: Think about what Spectral Normalization controls within a network, and what effect that would have if applied to the Discriminator in a mini-max game against the Generator.)
- Looking only at the various training curves, do you notice the effect of the Spectral normalization? Where do the effect of the Spectral Normalization show itself within the diagnostics?

```
Epoch 001/240 | D 1.299 | G 0.731 | Div 0.143
Epoch 040/240 | D 1.371 | G 0.703 | Div 4.572
Epoch 040/240 | D 1.371 | G 0.703 | Div 4.572
Epoch 080/240 | D 1.374 | G 0.700 | Div 4.610
Epoch 080/240 | D 1.374 | G 0.700 | Div 4.610
Epoch 120/240 | D 1.376 | G 0.700 | Div 4.528
Epoch 160/240 | D 1.378 | G 0.697 | Div 4.569
Epoch 200/240 | D 1.381 | G 0.697 | Div 4.568
Epoch 240/240 | D 1.383 | G 0.699 | Div 4.630
```



Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.



11.4 Summary and Next Steps

We have seen above how the mini-max formulation of a Generative Adversarial Network leads to unwanted training dynamics that need to be carefully tuned if the GAN is to perform well. In large part, we can see that this is the result of the two networks fighting each other. But is this fight really necessary? After all, the Generator is trying to learn the data distribution, and the Discriminator is just a tool to help it do so, because we didn't have a clear way to map the generated samples to the real samples in a way that directly allowed us to directly optimize something like an MSE.

In the next notebook, we will see how we can reformulate this problem in terms of Optimal Transport, which will allow us to sidestep some of these issues and remove some of the problems caused by the mini-max game.

12 Optimal Transport for Generative Models

In the previous notebooks, we explored how Generative Adversarial Networks (GANs) use a minimax game between a Generator and Discriminator to learn data distributions. However, we saw that this adversarial training can lead to instability, mode collapse, and other training difficulties. In this notebook, we'll explore an alternative approach based on **Optimal Transport (OT)**, which provides a more direct way to measure the distance between probability distributions.

The key insight is this: instead of training a discriminator to distinguish real from fake samples, we can directly minimize the distance between the generated distribution and the real data distribution using optimal transport metrics. This often leads to more stable training and better coverage of the data distribution.

12.1 What is Optimal Transport?

Let's go back to our earlier analogy where we were imagining probability distributions as two piles of sand, and our goal is to transform one pile into the other. Optimal transport addresses the question: **What is the most efficient way to move the sand from one configuration to another?**

More formally, given two probability distributions z and x , optimal transport finds a transport plan π that moves mass from z to x while minimizing the total *transport cost*. There are many possible definitions of cost here, and it is common to think of a distance as a form of cost, with the **Wasserstein distance** (also called the Earth Mover's Distance) as a common one with the technical form:

$$W_p(z, x) = \left(\inf_{\pi \in \Pi(z, x)} \int \|x - y\|^p d\pi(x, y) \right)^{1/p}$$

where $\Pi(z, x)$ is the set of all joint distributions with marginals z and x .

OK, so far so good in principle – I just need to find the transport plan that minimizes some p-norm over all joint distributions of z and x . However, in practice, this is not so straightforward,

since finding a minimum over all possible joint distributions of z and x is not so computationally tractable.¹

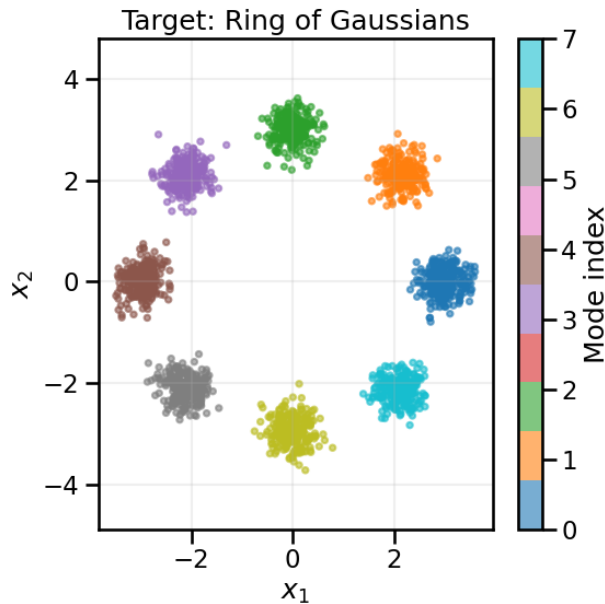
Instead, of computing the Wasserstein distance directly, we will compute an approximate version of it that regularizes the transport plan, and is called computing the **Sinkhorn Divergence**. Going over the specific implementation details of the Sinkhorn Divergence (which rely on Sinkhorn Iteration and knowledge of doubly stochastic matrices) are beyond the scope of what I want to cover in these notebooks, but interested students can check out [Computational Optimal Transport](#) by Gabriel Peyré and Marco Cuturi for further details.

The important thing to know in the context of a course at this level is that the Sinkhorn Divergence can only *approximate* the true Wasserstein distance, and that it does so via what is often called a “blur” parameter. This parameter is essentially a smoothing term that determines how much we penalize the complexity of the transport map. Some small amount of blur will help us compute gradients and use the Sinkhorn Divergence in ML model training, but too much of this will prevent a model from capturing fine details in the data distribution. You will see an interactive example of this next before we move on to using OT for GAN training.

12.1.1 Simple Optimal Transport Example

Let’s start with a familiar and concrete example that we have been using in the prior GAN notebooks. We’ll take a simple 2D Gaussian and compute its optimal transport to our ring of Gaussians dataset. We’ll visualize the transport map by computing the transport vectors (i.e., in what direction we move the probability mass) and also demonstrate how moving in those directions shifts our 2D Gaussian towards the ring.

¹At a high level, this is due to a combinatoric assignment problem in something called the coupling matrix, where you are trying to match generated and real datapoints to one another and optimize for the lowest distance. Because this assignment matrix is ultimately a binary matrix, this makes it not easily differentiable.



Now what we will do below is place a simple 2D Gaussian distribution centered at the origin, and then compute the Sinkhorn Divergence (loss) between each point in the real dataset and each point in the simple 2D Gaussian. Using Automatic Differentiation, we can then compute how to move each point in the generated distribution to minimize this loss. We will plot a sample of these gradient vectors so that you can see what the transport map looks like and also take a (very large) step in that direction for each point, so you can see the visual effect of the transport.

In this case, we are only taking a single, giant step along the transport map for pedagogical purposes, but in reality (and as we will do later), you would move slowly along the transport map over many iterations to gradually morph the generated distribution into the real data distribution.

```
# Create a simple 2D Gaussian source distribution
n_source = 2000
source_samples = np.random.randn(n_source, 2).astype(np.float32) * 0.5

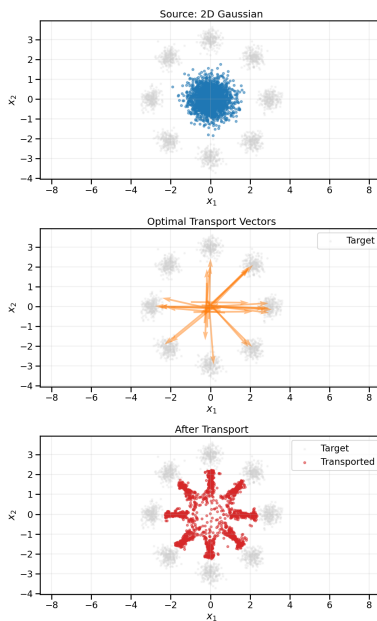
# Convert to torch tensors
source_torch = torch.from_numpy(source_samples).to(device)
target_torch = torch.from_numpy(X_ring).to(device)

# Compute optimal transport using Sinkhorn algorithm
# The blur parameter controls entropic regularization (larger = more regularization)
sinkhorn_loss = SamplesLoss("sinkhorn", p=2, blur=0.01, scaling=0.9)
```

```
# Compute transport plan by taking gradient
source_torch.requires_grad_(True)
loss = sinkhorn_loss(source_torch, target_torch)
loss.backward()

# This is effectively a giant step size so the we can visualize the gradients
# and see a meaningful change in the distribution.
# In reality, we would take much smaller steps than this and do it over iterations
magnitude_scaling = 2000

# The gradient points in the direction of optimal transport
transport_direction = source_torch.grad.detach().cpu().numpy()
transported_points = source_samples + transport_direction*magnitude_scaling
```



Sinkhorn divergence: 2.7932

In the visualization above, you can see how optimal transport naturally moves mass from the source Gaussian to the target ring distribution. The transport vectors (middle plot) show the direction and magnitude of how each point should move to minimize the total transport cost.

💡 Experiment: Effect of Sinkhorn Divergence Parameters

The Sinkhorn divergence has several key parameters that affect the transport:

1. **blur** (ϵ): Controls the amount of entropic regularization. Larger values make the transport “smoother” but less accurate.
2. **p**: The p-norm used for measuring distances. This is typically $p=1$ for Manhattan distance (sum of absolute differences) or $p=2$ for Euclidean distance (standard L2 norm)).

Use the slider below to gain intuition about changing the effects of these three parameters:
- What effect does moving from a small blur to a large blur have? - How does the transport pattern differ between $p=1$ and $p=2$?

[Interactive output: available in the online version of this chapter.]

12.2 Building an Entropic OT GAN

Now that we are beginning to understand optimal transport and how it might be useful, let’s try to use it to train a generative model. Instead of using a discriminator (as in standard GANs), we’ll directly minimize the Sinkhorn divergence between generated samples and real data. We can do this because the Sinkhorn divergence is now substituting for the original role of the Discriminator (i.e., to move the generator closer to real-world data), and so the Discriminator is no longer necessary.²

This approach has several advantages:

1. **No discriminator needed:** We only need the network for the Generator now, which is simpler and we fewer total parameters to train. We don’t need to worry about two learning rates or different capacities in each network, like we saw in the GAN Pitfalls notebook.
2. **More stable to train:** Since we no longer have a minimax game to balance, we do not have to contend with oscillatory behavior of the optimizer and potentially getting trapped in a loop. This can also equate to faster training times as a result, if the learning rate is suitably tuned.
3. **Better coverage:** OT naturally encourages covering all modes of the data distribution, rather than hoping that a Discriminator pushes the Generator to cover all modes.

²One could argue that we are no longer really doing a “GAN” here because we do not have an “Adversary” now that the Discriminator is gone, so it is perhaps misleading to call it this, but since earlier papers refer to this style of Generative Model as an Entropic GAN, I will be consistent with them, even if it isn’t the best name in my view.

4. **Meaningful gradients:** Because OT computes a pairwise distance among all data points, so long as the gradient of our distance/cost measure remains finite and non-zero at far away distances, we can still perform useful gradient descent steps in the generative model. This is not the case for some other cost functions (such as the KL Divergence) where gradients might vanish if the two distributions are not close enough.

```
class GeneratorMLP(nn.Module):
    """Simple MLP generator for 2D data."""
    def __init__(self, noise_dim: int = 2, hidden_dim: int = 256, out_dim: int = 2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(noise_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, out_dim)
        )

    def forward(self, z: torch.Tensor) -> torch.Tensor:
        return self.net(z)
```

```
from dataclasses import dataclass
from typing import List

@dataclass
class SinkhornHistory:
    """Training history for Sinkhorn-based generator."""
    loss: List[float]
    diversity: List[float]

def train_sinkhorn_generator(
    data: np.ndarray,
    noise_dim: int = 2,
    batch_size: int = 256,
    epochs: int = 200,
    lr: float = 1e-3,
    hidden_dim: int = 256,
    blur: float = 0.05,
    p: int = 2,
    print_every: int = 50
) -> tuple:
    """
```

Train a generator using Sinkhorn divergence.

Args:

data: Real data samples (numpy array)
noise_dim: Dimension of latent noise
batch_size: Batch size for training
epochs: Number of training epochs
lr: Learning rate
hidden_dim: Hidden dimension for generator
blur: Entropic regularization parameter
p: p-norm for distance metric
print_every: Print progress every N epochs

Returns:

Tuple of (trained generator, training history)

"""

if not geomloss_available:

raise ImportError("GeomLoss is required for Sinkhorn training. Install with: pip install geomloss")

Setup data loader

loader = make_loader(data, batch_size)

Initialize generator and optimizer

G = GeneratorMLP(noise_dim=noise_dim, hidden_dim=hidden_dim, out_dim=2).to(device)

optimizer = optim.Adam(G.parameters(), lr=lr, betas=(0.5, 0.999))

Setup Sinkhorn loss

sinkhorn_loss = SamplesLoss("sinkhorn", p=p, blur=blur, scaling=0.9, debias=True)

Training history

history = SinkhornHistory(loss=[], diversity=[])

print(f"Training Sinkhorn OT Generator for {epochs} epochs...")

print(f"Parameters: blur={blur}, p={p}, lr={lr}, hidden_dim={hidden_dim}")

for epoch **in** range(epochs):

epoch_losses = []

for (real_batch,) **in** loader:

real_batch = real_batch.to(device)

Generate fake samples

```

        z = torch.randn(batch_size, noise_dim, device=device)
        fake_batch = G(z)

        # Compute Sinkhorn divergence
        loss = sinkhorn_loss(fake_batch, real_batch)

        # Update generator
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_losses.append(float(loss.detach().cpu().item()))

    # Record metrics
    mean_loss = float(np.mean(epoch_losses))

    # Compute diversity
    with torch.no_grad():
        z_eval = torch.randn(2048, noise_dim, device=device)
        samples = G(z_eval)
        diversity = compute_diversity_metric(samples)

    history.loss.append(mean_loss)
    history.diversity.append(diversity)

    if (epoch + 1) % print_every == 0 or epoch == 0:
        print(f"Epoch {epoch+1:03d}/{epochs} | Loss: {mean_loss:.4f} | Diversity: {diversity:.4f}")
    return G, history

```

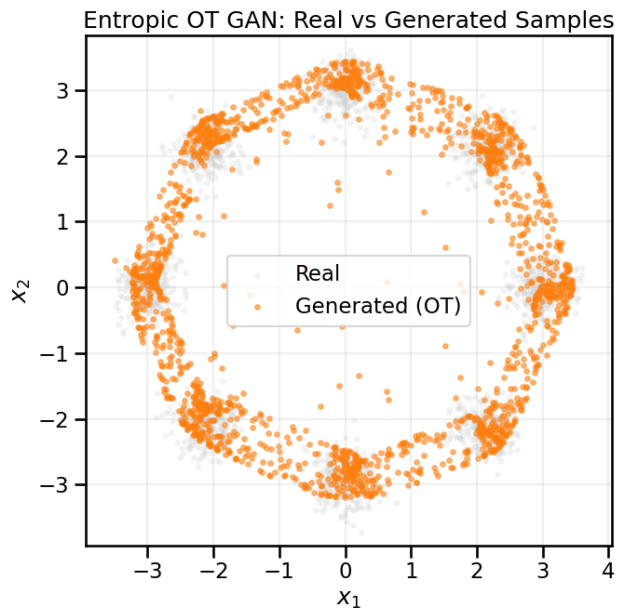
Let's train the OT generator on our ring dataset:

```

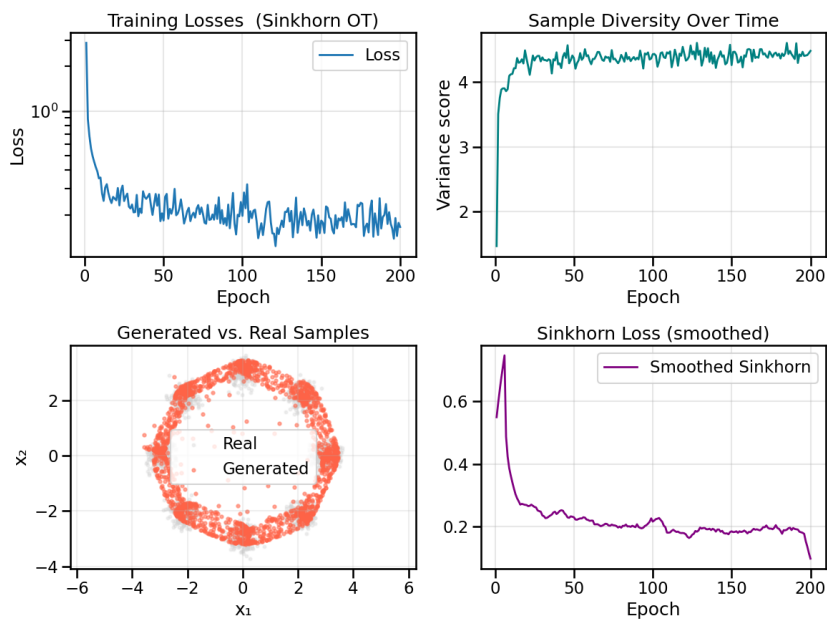
Training Sinkhorn OT Generator for 200 epochs...
Parameters: blur=0.02, p=2, lr=0.001, hidden_dim=256
Epoch 001/200 | Loss: 2.8801 | Diversity: 1.4708
Epoch 001/200 | Loss: 2.8801 | Diversity: 1.4708
Epoch 050/200 | Loss: 0.1830 | Diversity: 4.3724
Epoch 050/200 | Loss: 0.1830 | Diversity: 4.3724
Epoch 100/200 | Loss: 0.1967 | Diversity: 4.4298
Epoch 100/200 | Loss: 0.1967 | Diversity: 4.4298
Epoch 150/200 | Loss: 0.1828 | Diversity: 4.2631
Epoch 150/200 | Loss: 0.1828 | Diversity: 4.2631
Epoch 200/200 | Loss: 0.1656 | Diversity: 4.4816

```

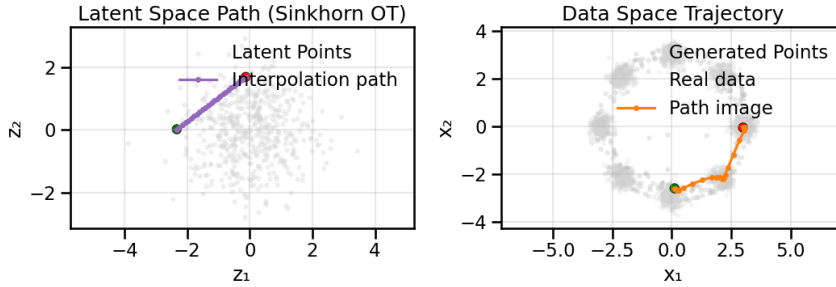
Epoch 200/200 | Loss: 0.1656 | Diversity: 4.4816



Let's use our standard diagnostic plots from the GAN notebooks to evaluate the OT generator's performance:



Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.



12.3 Summary and Next Steps

In this notebook, we explored how Optimal Transport provides an alternative to adversarial training for generative models:

1. **Optimal Transport Intuition:** OT finds the most efficient way to transform one distribution into another, providing meaningful gradients even when distributions don't overlap.
2. **Sinkhorn Divergence:** By adding entropic regularization, we make OT computationally tractable while preserving most benefits. The `blur` parameter controls the trade-off between precision and smoothness.
3. **Entropic OT GANs:** We can train generative models by directly minimizing Sinkhorn divergence, eliminating the need for a discriminator and the associated minimax game.
4. **Advantages over Traditional GANs:**
 - Simpler architecture (no discriminator)
 - More stable training (no adversarial dynamics)
 - Better mode coverage (OT naturally spreads mass)
 - Meaningful loss values (directly related to distribution distance)
5. **Trade-offs:** OT-based training requires careful tuning of the `blur` parameter and can be computationally expensive for very large datasets. In particular, OT-based distances are very good when the source and target distributions are already well aligned, but less so when the transport distance is high (since then the optimal map computed by Sinkhorn Iteration may not be as discriminative). As a consequence, many real-world applications of OT in a generative model context use OT as a fine-tuning step after doing an alignment or registration step (e.g., when matching a mesh of an organ to data from a CT scan).

So, at this point we have addressed one of the key weaknesses of GANs – the instability of adversarial training – by removing the discriminator entirely and replacing it with a direct measure of distribution distance. However, there is still one big missing piece to be desired by Entropic GAN-style models. While we have a better forward map now of $f(z) \rightarrow x$ via the generator, we still do not have an inverse map $f^{-1}(x) \rightarrow z$ that would allow us to encode real data points back into the latent space. This is something that Variational Autoencoders (VAEs) and Normalizing Flows provide, and is where we turn our attention next. We will see that the general concept of OT will raise its head again after Normalizing Flows in the context of something called “Flow Matching”, and we will return to this in a few notebooks.

13 Variational Autoencoders (VAEs)

In the previous notebooks, we explored push-forward generative models (GANs) and then moved to optimal transport approaches. These models excel at generating samples by learning a mapping $f(z) \rightarrow x$ from latent space to data space. However, they have a significant limitation: they don't provide a way to go backwards – to encode a real data point x back into the latent space z .

Variational Autoencoders (VAEs) address this limitation by learning both directions simultaneously:

- An **encoder** that maps data to latent space: $q_\phi(z|x)$
- A **decoder** that maps latent space to data: $p_\theta(x|z)$

We saw this kind of encoder-decoder structure before when we discussed (non-variational) Autoencoders, but only in the context of deterministic mappings for dimension reduction. Now, we will extend this idea to probabilistic mappings that allow us to both generate new data and infer latent representations with explicit likelihoods.

13.1 Learning Objectives

- Understand the architecture and motivation behind Variational Autoencoders
- Learn about the Evidence Lower Bound (ELBO) and why it's used for training
- Explore the role of the KL divergence regularization term
- Build and train a simple VAE on 2D data
- Visualize the learned latent space and understand the encoder-decoder relationship
- Compare VAEs to GANs and understand their respective strengths

Using device: cuda

13.2 What is a Variational Autoencoder?

To understand VAEs, let's start with regular **autoencoders**. An autoencoder is a neural network trained to compress data into a lower-dimensional representation (encoding) and then reconstruct the original data from that compressed form (decoding). The key insight is

that if we can successfully compress and reconstruct data, the compressed representation must capture the essential features of the original dataset, in a possibly more interpretable form.

However, regular autoencoders have a problem: the latent space can be irregular and discontinuous. Two similar data points might end up far apart in latent space, making it hard to generate new samples or interpolate smoothly. This is not a problem as far as a regular Autoencoder is concerned, since its only goal in life is to reconstruct the training data well (it doesn't care about interpolation or what the latent space might mean).

Variational Autoencoders solve this by imposing structure on the latent space. Instead of encoding a data point x to a single point z , a VAE encodes it to a **probability distribution** over z . This is typically a Gaussian distribution with mean μ and standard deviation σ :

$$q_\phi(z|x) = \mathcal{N}(z; \mu_\phi(x), \sigma_\phi^2(x))$$

where ϕ represents the encoder parameters (typically those of a neural network).

The decoder then samples from this distribution to reconstruct the data:

$$p_\theta(x|z) = \mathcal{N}(x; \mu_\theta(z), \sigma_\theta^2(z))$$

where θ represents the decoder parameters.

With this, we actually have two types of probabilistic models, one that can compute the likelihood of a data point (x) given a latent code (z), and a second that can infer a distribution over latent codes (z) given a data point (x).

13.2.1 The ELBO: Why VAEs Work

VAEs are trained by maximizing the **Evidence Lower Bound (ELBO)**, which decomposes into two terms:

$$\text{ELBO} = \underbrace{\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]}_{\text{Reconstruction term}} - \underbrace{D_{KL}(q_\phi(z|x) \| p(z))}_{\text{Regularization term}}$$

Let's break this down:

1. **Reconstruction term:** This measures how well the decoder can reconstruct the original data from the latent representation. It's similar to a standard autoencoder loss.
2. **KL divergence term:** This regularizes the latent space by encouraging $q_\phi(z|x)$ to be close to a prior distribution $p(z)$ (typically a standard normal $\mathcal{N}(0, I)$). This ensures the latent space is well-behaved and continuous.

The KL term has a nice closed-form solution when both distributions are Gaussian:

$$D_{KL}(q_\phi(z|x)||p(z)) = \frac{1}{2} \sum_{i=1}^d (\mu_i^2 + \sigma_i^2 - \log(\sigma_i^2) - 1)$$

Let’s step back a moment and think about what this term is encouraging. $p(z)$ is fixed, so the only thing we are changing when we optimize the KL divergence is q_ϕ (via the Neural Network weights ϕ). What is ϕ trying to do? Given an x from our training data, ϕ will transform it to z using the encoder’s neural network, and this will constitute a distribution over z influenced by x , specifically, $z \sim \mathcal{N}(z; \mu_\phi(x), \sigma_\phi^2(x))$. So this will produce a tiny little Gaussian blob in z for that given x , and the KL divergence will penalize this $q_\phi(z|x)$ from being “far” (in a KL sense) from $p(z)$. However, this doesn’t mean that $q_\phi(z|x)$ will look at all like $p(z)$, rather, $p(z)$ is just sort of gently encouraging $q_\phi(z|x)$ to remaining close by to $p(z)$. In fact, $q_\phi(z|x)$ can’t look too much like $p(z)$ because then $q_\phi(z|x)$ would be conditionally independent from x and this would make it hard for the decoder ($p_\theta(x|z)$) to reproduce x well.

13.2.2 The Reparameterization Trick

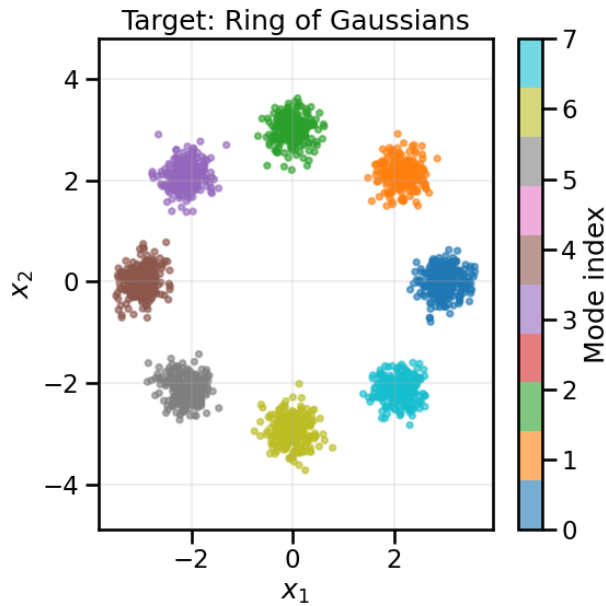
To train VAEs with backpropagation, we need gradients to flow through the sampling operation. The **reparameterization trick** achieves this by rewriting the sampling as:

$$z = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Now the randomness is moved to ϵ , and gradients can flow through μ and σ .

13.3 Building a Simple VAE

Let’s implement a VAE for our familiar 2D ring dataset. We’ll keep the architecture simple so we can visualize and understand what’s happening.



13.3.1 VAE Architecture

Our VAE will consist of two networks:

1. **Encoder:** Takes 2D data and outputs the parameters (μ , $\log \sigma^2$) of a Gaussian distribution in latent space
2. **Decoder:** Takes a latent code z and reconstructs the 2D data point

For this simple example, we'll use 2D latent space so we can visualize it directly.

```
class Encoder(nn.Module):
    """Encoder network that maps data to latent distribution parameters."""
    def __init__(self, x_dim: int = 2, hidden_dim: int = 128, z_dim: int = 2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(x_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )
        # Output mean and log-variance of the latent distribution
        self.mu_layer = nn.Linear(hidden_dim, z_dim)
        self.logvar_layer = nn.Linear(hidden_dim, z_dim)
```

```

def forward(self, x: torch.Tensor):
    h = self.net(x)
    mu = self.mu_layer(h)
    logvar = self.logvar_layer(h)
    return mu, logvar

class Decoder(nn.Module):
    """Decoder network that maps latent codes back to data space."""
    def __init__(self, z_dim: int = 2, hidden_dim: int = 128, x_dim: int = 2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(z_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, x_dim)
        )

    def forward(self, z: torch.Tensor):
        return self.net(z)

class VAE(nn.Module):
    """Variational Autoencoder combining encoder and decoder."""
    def __init__(self, x_dim: int = 2, z_dim: int = 2, hidden_dim: int = 128):
        super().__init__()
        self.encoder = Encoder(x_dim, hidden_dim, z_dim)
        self.decoder = Decoder(z_dim, hidden_dim, x_dim)
        self.z_dim = z_dim

    def reparameterize(self, mu: torch.Tensor, logvar: torch.Tensor):
        """Reparameterization trick:  $z = \mu + \sigma * \epsilon$ """
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x: torch.Tensor):
        # Encode
        mu, logvar = self.encoder(x)
        # Sample from latent distribution
        z = self.reparameterize(mu, logvar)

```

```

# Decode
x_recon = self.decoder(z)
return x_recon, mu, logvar, z

def encode(self, x: torch.Tensor):
    """Encode data to latent space (returns mean of distribution)."""
    mu, _ = self.encoder(x)
    return mu

def decode(self, z: torch.Tensor):
    """Decode latent codes to data space."""
    return self.decoder(z)

```

13.3.2 The VAE Loss Function

The VAE loss combines reconstruction error and KL divergence. For reconstruction, we'll use mean squared error (MSE), which corresponds to assuming a Gaussian likelihood with a mean at the decoder value ($\hat{x} = \mu_\theta(z)$) and a variance σ_θ^2 that could be output by the decoder:

$$\mathcal{L} = \underbrace{\frac{1}{2\sigma_\theta^2} \|x - \hat{x}\|^2}_{\text{Reconstruction}} + \underbrace{D_{KL}(q_\phi(z|x) \| p(z))}_{\text{KL divergence}}$$

In a simplified case, we could set σ_θ^2 to be a constant value (σ^2) in which case the only job of the decoder would be to output a single vector (the mean value $\hat{x} = \mu_\theta(z)$) and then the constant σ^2 would effectively control the trade-off between reconstruction accuracy and regularization strength.

```

def vae_loss(x: torch.Tensor, x_recon: torch.Tensor, mu: torch.Tensor, logvar: torch.Tensor,
            beta: float = 1.0, reconstruction_variance: float = 0.1):
    """
    Compute VAE loss = Reconstruction loss + beta * KL divergence.

    Args:
        x: Original data
        x_recon: Reconstructed data
        mu: Mean of latent distribution
        logvar: Log-variance of latent distribution
        beta: Weight for KL term (beta=1 is standard VAE, beta>1 is beta-VAE)
        reconstruction_variance: Assumed variance of reconstruction distribution
    """

```



```

# Reconstruction loss (MSE scaled by assumed variance)
recon_loss = torch.sum((x - x_recon) ** 2, dim=1) / (2 * reconstruction_variance)

# KL divergence between q(z|x) and p(z) = N(0,I)
# KL(N(mu, sigma^2) || N(0,1)) = 0.5 * sum(mu^2 + sigma^2 - log(sigma^2) - 1)
kl_div = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), dim=1)

# Total loss (negative ELBO)
return (recon_loss + beta * kl_div).mean(), recon_loss.mean(), kl_div.mean()

```

13.3.3 Training the VAE

Now let's train our VAE on the ring dataset. We'll track both the reconstruction loss and KL divergence separately to understand how the model learns.

```

@dataclass
class VAEHistory:
    """Training history for VAE."""
    loss: List[float]
    recon_loss: List[float]
    kl_loss: List[float]
    diversity: List[float]

def train_vae(
    data: np.ndarray,
    x_dim: int = 2,
    z_dim: int = 2,
    hidden_dim: int = 128,
    batch_size: int = 256,
    epochs: int = 200,
    lr: float = 1e-3,
    beta: float = 1.0,
    reconstruction_variance: float = 0.1,
    print_every: int = 50
):
    """
    Train a VAE on the provided data.

    Args:
        data: Training data (numpy array)
    """

```

```

x_dim: Dimension of data space
z_dim: Dimension of latent space
hidden_dim: Hidden layer dimension
batch_size: Batch size for training
epochs: Number of training epochs
lr: Learning rate
beta: Weight for KL term (beta-VAE parameter)
reconstruction_variance: Assumed variance of reconstruction distribution
print_every: Print progress every N epochs

Returns:
    Trained VAE model and training history
"""
# Setup data loader
loader = make_loader(data, batch_size)

# Initialize model and optimizer
vae = VAE(x_dim=x_dim, z_dim=z_dim, hidden_dim=hidden_dim).to(device)
optimizer = optim.Adam(vae.parameters(), lr=lr)

# Training history
history = VAHistory(loss=[], recon_loss=[], kl_loss=[], diversity=[])

print(f"Training VAE for {epochs} epochs...")
print(f"Parameters: z_dim={z_dim}, beta={beta}, recon_var={reconstruction_variance}, lr={lr}")

vae.train()
for epoch in range(epochs):
    epoch_losses = []
    epoch_recon = []
    epoch_kl = []

    for (batch,) in loader:
        batch = batch.to(device)

        # Forward pass
        x_recon, mu, logvar, z = vae(batch)

        # Compute loss
        loss, recon, kl = vae_loss(batch, x_recon, mu, logvar, beta, reconstruction_variance)

        # Backward pass

```

```

optimizer.zero_grad()
loss.backward()
optimizer.step()

epoch_losses.append(loss.item())
epoch_recon.append(recon.item())
epoch_kl.append(kl.item())

# Record metrics
mean_loss = np.mean(epoch_losses)
mean_recon = np.mean(epoch_recon)
mean_kl = np.mean(epoch_kl)

# Compute diversity (sample from prior and decode)
with torch.no_grad():
    z_samples = torch.randn(2048, z_dim, device=device)
    x_samples = vae.decode(z_samples)
    diversity = compute_diversity_metric(x_samples)

history.loss.append(mean_loss)
history.recon_loss.append(mean_recon)
history.kl_loss.append(mean_kl)
history.diversity.append(diversity)

if (epoch + 1) % print_every == 0 or epoch == 0:
    print(f"Epoch {epoch+1:03d}/{epochs} | Loss: {mean_loss:.3f} | "
          f"Recon: {mean_recon:.3f} | KL: {mean_kl:.3f} | Div: {diversity:.3f}")
return vae, history

```

Training VAE for 400 epochs...

Parameters: z_dim=2, beta=1.0, recon_var=0.1, lr=0.001

```

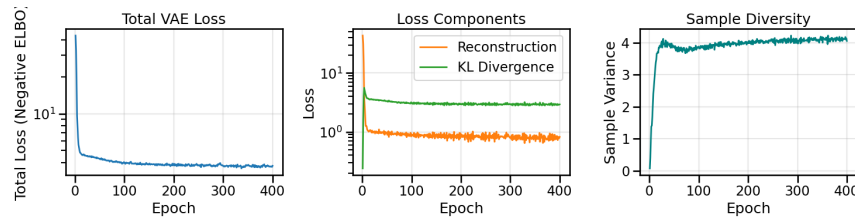
Epoch 001/400 | Loss: 43.550 | Recon: 43.310 | KL: 0.240 | Div: 0.074
Epoch 001/400 | Loss: 43.550 | Recon: 43.310 | KL: 0.240 | Div: 0.074
Epoch 050/400 | Loss: 4.296 | Recon: 0.964 | KL: 3.332 | Div: 3.860
Epoch 050/400 | Loss: 4.296 | Recon: 0.964 | KL: 3.332 | Div: 3.860
Epoch 100/400 | Loss: 3.996 | Recon: 0.917 | KL: 3.079 | Div: 3.879
Epoch 100/400 | Loss: 3.996 | Recon: 0.917 | KL: 3.079 | Div: 3.879
Epoch 150/400 | Loss: 3.904 | Recon: 0.842 | KL: 3.062 | Div: 3.965
Epoch 150/400 | Loss: 3.904 | Recon: 0.842 | KL: 3.062 | Div: 3.965
Epoch 200/400 | Loss: 3.813 | Recon: 0.813 | KL: 3.001 | Div: 4.029
Epoch 200/400 | Loss: 3.813 | Recon: 0.813 | KL: 3.001 | Div: 4.029
Epoch 250/400 | Loss: 3.797 | Recon: 0.986 | KL: 2.810 | Div: 4.013

```

Epoch 250/400	Loss: 3.797	Recon: 0.986	KL: 2.810	Div: 4.013
Epoch 300/400	Loss: 3.734	Recon: 0.822	KL: 2.912	Div: 4.059
Epoch 300/400	Loss: 3.734	Recon: 0.822	KL: 2.912	Div: 4.059
Epoch 350/400	Loss: 3.773	Recon: 0.873	KL: 2.900	Div: 4.172
Epoch 350/400	Loss: 3.773	Recon: 0.873	KL: 2.900	Div: 4.172
Epoch 400/400	Loss: 3.758	Recon: 0.824	KL: 2.935	Div: 4.051
Epoch 400/400	Loss: 3.758	Recon: 0.824	KL: 2.935	Div: 4.051

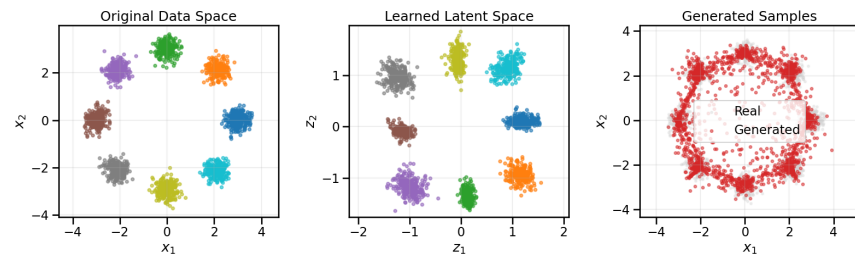
13.3.4 Visualizing Training Progress

Let's look at how the different loss components evolved during training:



13.4 Exploring the Learned Latent Space

One of the most powerful aspects of VAEs is that we can visualize the latent space directly (since we're using 2D latent codes). Let's see how the VAE has organized the ring data in latent space.



13.4.1 Visualizing $q_\phi(z|x)$

The encoder learns to map each input data point x to a *distribution* in the latent space, not just a single point. Let's pick two points from our dataset and visualize the corresponding Gaussian distributions $q_\phi(z|x)$ that the encoder has learned for them.

```

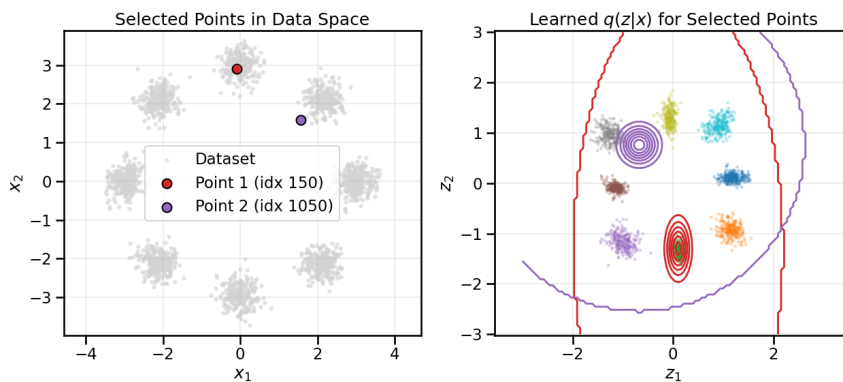
# Select two points from the dataset
point_1_idx, point_2_idx = 150, 1050
x_1 = torch.from_numpy(X_ring[point_1_idx]).float().to(device)
x_2 = torch.from_numpy(X_ring[point_2_idx]).float().to(device)

# Get the parameters of  $q(z|x)$  for these two points
vae.eval()
with torch.no_grad():
    mu_1, logvar_1 = vae.encoder(x_1.unsqueeze(0))
    mu_2, logvar_2 = vae.encoder(x_2.unsqueeze(0))

# Create distributions
std_1 = torch.exp(0.5 * logvar_1)
dist_1 = torch.distributions.MultivariateNormal(mu_1.squeeze(), torch.diag(std_1.squeeze().pow(2)))

std_2 = torch.exp(0.5 * logvar_2)
dist_2 = torch.distributions.MultivariateNormal(mu_2.squeeze(), torch.diag(std_2.squeeze().pow(2)))

```



13.4.2 Visualizing the Latent Space Manifold

Since both our data and latent space are 2D, we can create a grid in latent space and see what the decoder produces at each point. This helps us understand how the VAE has organized the latent space.

```

# Create a grid in latent space
n_points = 20
z_range = 2.0
z1 = np.linspace(-z_range, z_range, n_points)
z2 = np.linspace(-z_range, z_range, n_points)

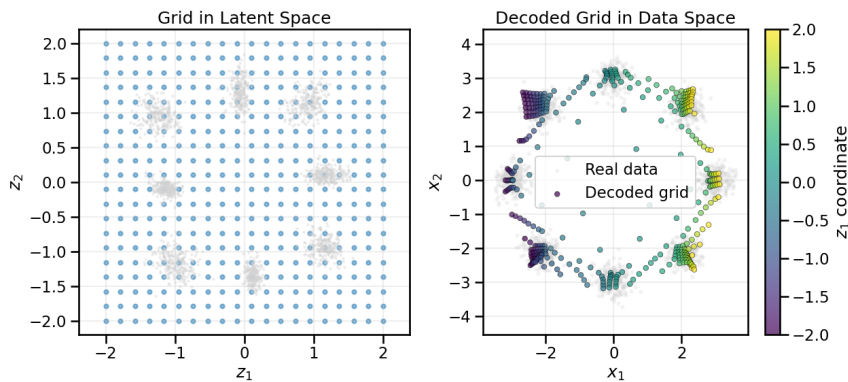
```

```

Z1, Z2 = np.meshgrid(z1, z2)
z_grid = np.stack([Z1.flatten(), Z2.flatten()], axis=1).astype(np.float32)

# Decode the grid
with torch.no_grad():
    z_grid_tensor = torch.from_numpy(z_grid).to(device)
    x_grid = vae.decode(z_grid_tensor).cpu().numpy()

```



💡 Experiment: Effect of re-weighting the KL Divergence term

We can see above that `reconstruction_variance` essentially sets the variances of the Gaussian outputs for the reconstructed data points. - Mathematically, what does setting this to a small variance imply for the loss function? What about a big variance? - Try modifying the `reconstruction_variance` above from a small number to a large one, e.g., 0.01 to 5.0. What happens to the output of the VAE? Why is this? (Note: if you go below about 0.1 then you may have to increase the learning rate or number of epochs unless it converges.)

13.5 Conditional Generation Examples

Just like with GANs, we can extend VAEs to conditional generation by providing additional information to both the encoder and decoder. This allows us to control what we generate based on labels or continuous attributes.

In conditional VAEs (CVAEs), we modify both the encoder and decoder to accept conditioning information:

- The **encoder** receives both the data x and condition y , producing $q_\phi(z|x, y)$
- The **decoder** receives both the latent code z and condition y , producing $p_\theta(x|z, y)$

This conditioning allows us to: 1. Generate samples with specific properties (e.g., from a particular mode) 2. Learn more structured latent representations that disentangle the conditioning variable from other factors of variation

Let's explore both discrete (one-hot mode index) and continuous (radius) conditioning for VAEs.

13.5.1 Conditional Example A — One-Hot Mode Conditioning

We'll first condition the VAE on the discrete mode index using one-hot encoding. The conditioning vector is concatenated to the inputs of both the encoder and decoder networks.

```
# Conditional Encoder
class ConditionalEncoder(nn.Module):
    def __init__(self, x_dim: int = 2, label_dim: int = 8, hidden_dim: int = 128, z_dim: int = 2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(x_dim + label_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )
        self.mu_layer = nn.Linear(hidden_dim, z_dim)
        self.logvar_layer = nn.Linear(hidden_dim, z_dim)

    def forward(self, x: torch.Tensor, y: torch.Tensor):
        xy = torch.cat([x, y], dim=1)
        h = self.net(xy)
        mu = self.mu_layer(h)
        logvar = self.logvar_layer(h)
        return mu, logvar

# Conditional Decoder
class ConditionalDecoder(nn.Module):
    def __init__(self, z_dim: int = 2, label_dim: int = 8, hidden_dim: int = 128, x_dim: int = 2):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(z_dim + label_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, x_dim)
```

```

    )

    def forward(self, z: torch.Tensor, y: torch.Tensor):
        zy = torch.cat([z, y], dim=1)
        return self.net(zy)

# Conditional VAE
class ConditionalVAE(nn.Module):
    def __init__(self, x_dim: int = 2, z_dim: int = 2, label_dim: int = 8, hidden_dim: int = 128):
        super().__init__()
        self.encoder = ConditionalEncoder(x_dim, label_dim, hidden_dim, z_dim)
        self.decoder = ConditionalDecoder(z_dim, label_dim, hidden_dim, x_dim)
        self.z_dim = z_dim

    def reparameterize(self, mu: torch.Tensor, logvar: torch.Tensor):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x: torch.Tensor, y: torch.Tensor):
        mu, logvar = self.encoder(x, y)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decoder(z, y)
        return x_recon, mu, logvar, z

    def decode(self, z: torch.Tensor, y: torch.Tensor):
        return self.decoder(z, y)

def train_conditional_vae_onehot(
    data: np.ndarray, labels: np.ndarray, n_classes: int = 8,
    x_dim: int = 2, z_dim: int = 2, hidden_dim: int = 128,
    batch_size: int = 256, epochs: int = 200, lr: float = 1e-3,
    beta: float = 1.0, reconstruction_variance: float = 0.1, print_every: int = 50
):
    loader = make_conditional_loader_onehot(data, labels, batch_size, n_classes)
    vae = ConditionalVAE(x_dim=x_dim, z_dim=z_dim, label_dim=n_classes, hidden_dim=hidden_dim).to(device)
    optimizer = optim.Adam(vae.parameters(), lr=lr)
    history = VAHHistory([], [], [], [])

    for epoch in range(epochs):
        total_loss, total_recon, total_kl = 0.0, 0.0, 0.0
        for xb, yb in loader:

```



```

        xb, yb = xb.to(device), yb.to(device)
        x_recon, mu, logvar, z = vae(xb, yb)
        loss, recon_loss, kl_loss = vae_loss(xb, x_recon, mu, logvar, beta, reconstruction_variance)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        total_recon += recon_loss.item()
        total_kl += kl_loss.item()

    # Compute diversity on generated samples
    with torch.no_grad():
        z_sample = torch.randn(2048, z_dim, device=device)
        y_sample = torch.nn.functional.one_hot(
            torch.randint(0, n_classes, (2048,), device=device), num_classes=n_classes
        ).to(torch.float32)
        samples = vae.decode(z_sample, y_sample)
        div = compute_diversity_metric(samples)

    n_batches = len(loader)
    history.loss.append(total_loss / n_batches)
    history.recon_loss.append(total_recon / n_batches)
    history.kl_loss.append(total_kl / n_batches)
    history.diversity.append(div)

    if (epoch + 1) % print_every == 0 or epoch == 0:
        print(f"Epoch {epoch+1:03d}/{epochs} | Loss: {history.loss[-1]:.3f} | "
              f"Recon: {history.recon_loss[-1]:.3f} | KL: {history.kl_loss[-1]:.3f} | Div: {div:.3f}")

    return vae, history

# Train conditional VAE
cvae_onehot, hist_cvae_onehot = train_conditional_vae_onehot(
    X_ring, y_ring, n_classes=8, epochs=500, batch_size=256,
    lr=1e-3, beta=1.0, hidden_dim=128, z_dim=2, print_every=50
)

```

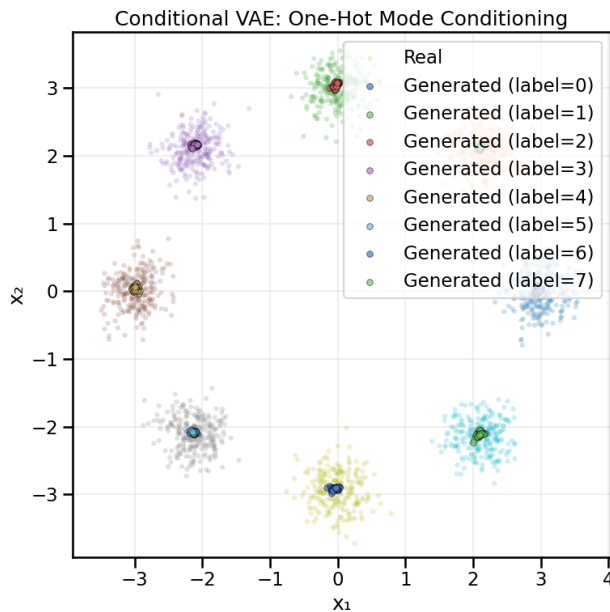
```

Epoch 001/500 | Loss: 43.496 | Recon: 43.452 | KL: 0.045 | Div: 0.012
Epoch 050/500 | Loss: 0.638 | Recon: 0.633 | KL: 0.004 | Div: 4.465
Epoch 050/500 | Loss: 0.638 | Recon: 0.633 | KL: 0.004 | Div: 4.465
Epoch 100/500 | Loss: 0.636 | Recon: 0.635 | KL: 0.001 | Div: 4.463
Epoch 100/500 | Loss: 0.636 | Recon: 0.635 | KL: 0.001 | Div: 4.463

```

Epoch 150/500	Loss: 0.639	Recon: 0.638	KL: 0.000	Div: 4.465
Epoch 150/500	Loss: 0.639	Recon: 0.638	KL: 0.000	Div: 4.465
Epoch 200/500	Loss: 0.624	Recon: 0.624	KL: 0.000	Div: 4.438
Epoch 200/500	Loss: 0.624	Recon: 0.624	KL: 0.000	Div: 4.438
Epoch 250/500	Loss: 0.640	Recon: 0.639	KL: 0.000	Div: 4.479
Epoch 250/500	Loss: 0.640	Recon: 0.639	KL: 0.000	Div: 4.479
Epoch 300/500	Loss: 0.628	Recon: 0.628	KL: 0.001	Div: 4.398
Epoch 300/500	Loss: 0.628	Recon: 0.628	KL: 0.001	Div: 4.398
Epoch 350/500	Loss: 0.626	Recon: 0.625	KL: 0.000	Div: 4.434
Epoch 350/500	Loss: 0.626	Recon: 0.625	KL: 0.000	Div: 4.434
Epoch 400/500	Loss: 0.632	Recon: 0.632	KL: 0.000	Div: 4.438
Epoch 400/500	Loss: 0.632	Recon: 0.632	KL: 0.000	Div: 4.438
Epoch 450/500	Loss: 0.629	Recon: 0.628	KL: 0.000	Div: 4.492
Epoch 450/500	Loss: 0.629	Recon: 0.628	KL: 0.000	Div: 4.492
Epoch 500/500	Loss: 0.625	Recon: 0.625	KL: 0.000	Div: 4.452
Epoch 500/500	Loss: 0.625	Recon: 0.625	KL: 0.000	Div: 4.452

Let's visualize the conditional samples generated by the CVAE for each mode:



13.5.2 Conditional Example B — Continuous Radius Conditioning

Now let's condition the VAE on a continuous scalar—the radius. We'll embed the radius through a small MLP before concatenating with the encoder and decoder inputs.

```

from gen_models_utilities import make_conditional_loader_radius, plot_conditional_samples_continuous

# Conditional Encoder with radius embedding
class ConditionalEncoderRadius(nn.Module):
    def __init__(self, x_dim: int = 2, cond_emb_dim: int = 16, hidden_dim: int = 128, z_dim: int = 2):
        super().__init__()
        self.cond_embed = nn.Sequential(
            nn.Linear(1, cond_emb_dim),
            nn.ReLU(),
            nn.Linear(cond_emb_dim, cond_emb_dim),
        )
        self.net = nn.Sequential(
            nn.Linear(x_dim + cond_emb_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )
        self.mu_layer = nn.Linear(hidden_dim, z_dim)
        self.logvar_layer = nn.Linear(hidden_dim, z_dim)

    def forward(self, x: torch.Tensor, r: torch.Tensor):
        r_emb = self.cond_embed(r)
        xr = torch.cat([x, r_emb], dim=1)
        h = self.net(xr)
        mu = self.mu_layer(h)
        logvar = self.logvar_layer(h)
        return mu, logvar

# Conditional Decoder with radius embedding
class ConditionalDecoderRadius(nn.Module):
    def __init__(self, z_dim: int = 2, cond_emb_dim: int = 16, hidden_dim: int = 128, x_dim: int = 2):
        super().__init__()
        self.cond_embed = nn.Sequential(
            nn.Linear(1, cond_emb_dim),
            nn.ReLU(),
            nn.Linear(cond_emb_dim, cond_emb_dim),
        )
        self.net = nn.Sequential(
            nn.Linear(z_dim + cond_emb_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),

```

```

        nn.Linear(hidden_dim, x_dim)
    )

    def forward(self, z: torch.Tensor, r: torch.Tensor):
        r_emb = self.cond_embed(r)
        zr = torch.cat([z, r_emb], dim=1)
        return self.net(zr)

# Conditional VAE with radius
class ConditionalVAERadius(nn.Module):
    def __init__(self, x_dim: int = 2, z_dim: int = 2, cond_emb_dim: int = 16, hidden_dim: int = 128):
        super().__init__()
        self.encoder = ConditionalEncoderRadius(x_dim, cond_emb_dim, hidden_dim, z_dim)
        self.decoder = ConditionalDecoderRadius(z_dim, cond_emb_dim, hidden_dim, x_dim)
        self.z_dim = z_dim

    def reparameterize(self, mu: torch.Tensor, logvar: torch.Tensor):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def forward(self, x: torch.Tensor, r: torch.Tensor):
        mu, logvar = self.encoder(x, r)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decoder(z, r)
        return x_recon, mu, logvar, z

    def decode(self, z: torch.Tensor, r: torch.Tensor):
        return self.decoder(z, r)

def train_conditional_vae_radius(
    data: np.ndarray,
    x_dim: int = 2, z_dim: int = 2, hidden_dim: int = 128,
    batch_size: int = 256, epochs: int = 200, lr: float = 1e-3,
    beta: float = 1.0, reconstruction_variance: float = 0.1, print_every: int = 50
):
    loader = make_conditional_loader_radius(data, batch_size)
    vae = ConditionalVAERadius(x_dim=x_dim, z_dim=z_dim, hidden_dim=hidden_dim).to(device)
    optimizer = optim.Adam(vae.parameters(), lr=lr)
    history = VAHistory([], [], [], [])

    for epoch in range(epochs):

```

```

total_loss, total_recon, total_kl = 0.0, 0.0, 0.0
for xb, rb in loader:
    xb, rb = xb.to(device), rb.to(device)
    x_recon, mu, logvar, z = vae(xb, rb)
    loss, recon_loss, kl_loss = vae_loss(xb, x_recon, mu, logvar, beta, reconstruction_variance)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    total_loss += loss.item()
    total_recon += recon_loss.item()
    total_kl += kl_loss.item()

    with torch.no_grad():
        z_sample = torch.randn(2048, z_dim, device=device)
        r_sample = torch.norm(torch.randn(2048, 2), dim=1, keepdim=True).to(device)
        samples = vae.decode(z_sample, r_sample)
        div = compute_diversity_metric(samples)

n_batches = len(loader)
history.loss.append(total_loss / n_batches)
history.recon_loss.append(total_recon / n_batches)
history.kl_loss.append(total_kl / n_batches)
history.diversity.append(div)

if (epoch + 1) % print_every == 0 or epoch == 0:
    print(f"Epoch {epoch+1:03d}/{epochs} | Loss: {history.loss[-1]:.3f} | "
          f"Recon: {history.recon_loss[-1]:.3f} | KL: {history.kl_loss[-1]:.3f} | Div: {div:.3f}")

return vae, history

# Train conditional VAE with radius
cvae_radius, hist_cvae_radius = train_conditional_vae_radius(
    X_ring, epochs=500, batch_size=256, lr=1e-3, beta=1.0, hidden_dim=128, z_dim=2, print_every=50
)

```

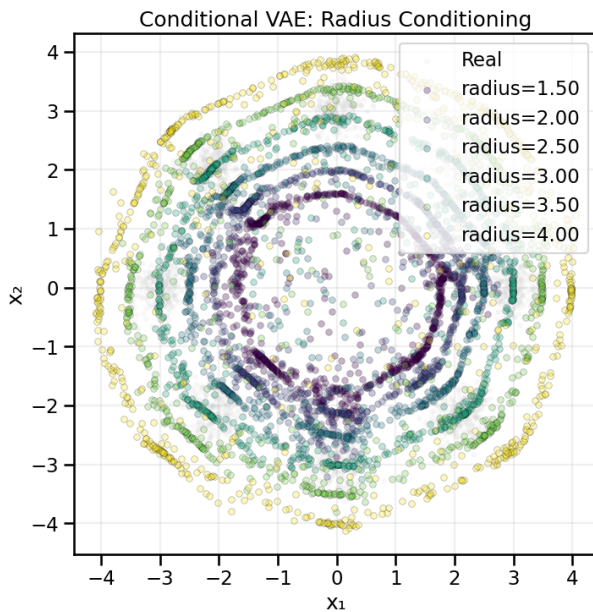
```

Epoch 001/500 | Loss: 45.153 | Recon: 45.136 | KL: 0.017 | Div: 0.003
Epoch 050/500 | Loss: 4.302 | Recon: 0.906 | KL: 3.396 | Div: 1.842
Epoch 050/500 | Loss: 4.302 | Recon: 0.906 | KL: 3.396 | Div: 1.842
Epoch 100/500 | Loss: 3.996 | Recon: 0.843 | KL: 3.152 | Div: 1.444
Epoch 100/500 | Loss: 3.996 | Recon: 0.843 | KL: 3.152 | Div: 1.444
Epoch 150/500 | Loss: 3.829 | Recon: 0.673 | KL: 3.156 | Div: 1.113
Epoch 150/500 | Loss: 3.829 | Recon: 0.673 | KL: 3.156 | Div: 1.113

```

Epoch 200/500	Loss: 3.773	Recon: 0.634	KL: 3.139	Div: 1.073
Epoch 200/500	Loss: 3.773	Recon: 0.634	KL: 3.139	Div: 1.073
Epoch 250/500	Loss: 3.681	Recon: 0.624	KL: 3.057	Div: 1.158
Epoch 250/500	Loss: 3.681	Recon: 0.624	KL: 3.057	Div: 1.158
Epoch 300/500	Loss: 3.606	Recon: 0.604	KL: 3.002	Div: 1.216
Epoch 300/500	Loss: 3.606	Recon: 0.604	KL: 3.002	Div: 1.216
Epoch 350/500	Loss: 3.643	Recon: 0.656	KL: 2.987	Div: 1.280
Epoch 350/500	Loss: 3.643	Recon: 0.656	KL: 2.987	Div: 1.280
Epoch 400/500	Loss: 3.640	Recon: 0.563	KL: 3.077	Div: 1.373
Epoch 400/500	Loss: 3.640	Recon: 0.563	KL: 3.077	Div: 1.373
Epoch 450/500	Loss: 3.597	Recon: 0.636	KL: 2.961	Div: 1.344
Epoch 450/500	Loss: 3.597	Recon: 0.636	KL: 2.961	Div: 1.344
Epoch 500/500	Loss: 3.594	Recon: 0.542	KL: 3.051	Div: 1.392
Epoch 500/500	Loss: 3.594	Recon: 0.542	KL: 3.051	Div: 1.392

Let's generate samples at different radius values and visualize how the CVAE responds to continuous conditioning:



13.6 Comparing VAEs to GANs and OT Models

Now that we understand VAEs, let's compare them to the other generative models we've studied:

Aspect	GAN	OT GAN	VAE
Architecture	Generator + Discriminator	Generator only	Encoder + Decoder
Training	Adversarial (minimax)	Direct divergence minimization	Maximize ELBO
Stability	Can be unstable	More stable	Generally stable
Forward map	Yes ($z \rightarrow x$)	Yes ($z \rightarrow x$)	Yes ($z \rightarrow x$)
Inverse map	No	No	Yes ($x \rightarrow z$)
Mode coverage	Prone to mode collapse	Good coverage	Good coverage
Interpretability	Limited	Limited	Good (latent space mirrors $p(z)$)

13.6.1 When to use VAEs:

1. **When you need an encoder:** If you want to map real data to latent codes (for analysis, interpolation, or compression)
2. **For smooth interpolation:** The regularized latent space makes interpolation meaningful
3. **When interpretability matters:** VAEs often learn more interpretable latent representations
4. **When training stability is important:** VAEs are generally easier to train than GANs

13.6.2 When to use GANs or OT models:

1. **When sample quality is paramount:** GANs often produce sharper, more realistic samples
2. **When you only need generation:** If you don't need to encode real data
3. **For specific domains:** GANs can excel at images, videos, and other perceptual data compared to VAEs (at least historically)

13.7 Summary and Looking Forward

In this notebook, we explored Variational Autoencoders (VAEs), which provide a principled approach to learning bidirectional mappings between data and latent space:

13.7.1 Key Takeaways:

1. **VAEs learn distributions:** Unlike deterministic autoencoders, VAEs encode data as distributions in latent space, enabling smooth interpolation and principled sampling.
2. **The ELBO objective:** VAEs maximize a lower bound on the data likelihood, balancing reconstruction accuracy with latent space regularization.
3. **KL divergence regularization:** This term ensures the latent space is well-behaved (close to a standard normal), making sampling and interpolation meaningful.
4. **Bidirectional mapping:** VAEs provide both encoding ($x \rightarrow z$) and decoding ($z \rightarrow x$), enabling applications like representation learning and data analysis.

13.7.2 Limitations and Extensions:

While VAEs are powerful, they have some limitations: - **Blurry samples:** The MSE reconstruction loss can lead to averaged, blurry outputs - **Posterior collapse:** In high dimensions, the KL term can dominate, causing the model to ignore the encoder - **Limited expressiveness:** The Gaussian assumption may be too restrictive for complex distributions

Modern extensions address these issues (not covered here): - **β -VAE:** Adjustable β for better disentanglement - **Normalizing Flow VAEs:** Use more expressive posterior distributions - **Vector Quantized VAEs (VQ-VAE):** Discrete latent codes for sharper reconstructions - **Hierarchical VAEs:** Multiple layers of latent variables for richer representations

13.7.3 Next Steps:

VAEs created an important step beyond GAN models in that they allowed us to model probability distributions over both the output data samples from the generator $p_\theta(z|x)$ but also the possible latent codes via an encoder $q_\phi(z|x)$. However, the VAE was only able to predict the latent codes using an approximation method (variational inference) by assuming a q and minimizing the ELBO.

What if it were possible to learn both $p(x|z)$ and $q(z|x)$ exactly? In the next notebook, we'll explore **Normalizing Flows**, which provide another way to learn bidirectional mappings with *exact* likelihood computation. Unlike VAEs, flows are deterministic and invertible, allowing for exact inference without variational approximations. This makes them particularly useful when we need precise likelihood estimates or want to avoid the approximation errors inherent in VAEs.

14 Normalizing Flows

In the previous chapter we trained a Variational Autoencoder (VAE) to learn a latent representation that could reconstruct and generate samples. VAEs rely on a probabilistic encoder-decoder pair and optimize the Evidence Lower Bound (ELBO). In this notebook we take a different perspective: we build models that exactly transform a simple base distribution (e.g., a Gaussian) into our target data distribution using a sequence of invertible mappings known as *normalizing flows*.

We will reuse the same 2D ring-of-Gaussians dataset that we have been using in prior notebooks so far, so that we can compare models directly and focus on the geometric intuition behind these transformations.

14.1 Learning Objectives

- Understand how normalizing flows differ from VAEs in the way they model probability densities
- Apply the change-of-variables formula to relate base and transformed distributions
- Experiment with invertible transformations in 1D and 2D to build intuition
- Implement and train a RealNVP-style normalizing flow on the ring dataset
- Summarize the advantages and limitations of flows and motivate continuous or stochastic extensions

Using device: cuda

14.2 VAEs vs. Normalizing Flows

VAEs *approximate* the data likelihood by maximizing an evidence lower bound, balancing a reconstruction term with a KL regularizer on the latent posterior. Normalizing flows instead learn an *exactly* normalizable density by composing invertible transformations that map a simple base distribution to the target data distribution using the change-of-variables formula.

The key trade-off is between tractable posterior inference (VAE) and exact likelihood modeling with potentially sharper samples (flow). Both rely on simple base distributions, but flows *must* ensure every transformation remains invertible with a tractable Jacobian determinant.

14.3 Key Idea: The Change-of-Variables Formula

Suppose we want to transform a simple base random variable z with known density $p_Z(z)$ into a new variable $x = f(z)$. If f is invertible and differentiable, then the density of x is given by the Change-of-Variables formula:

$$p_X(x) = p_Z(f^{-1}(x)) \left| \det \frac{\partial f^{-1}(x)}{\partial x} \right| = p_Z(z) \left| \det \frac{\partial f(z)}{\partial z} \right|^{-1}.$$

This formula is for a single function f . In Normalizing Flows, the idea is to instead construct f as a **composition** of many simple, easily invertible transformations, each of which can be computed using the Change-of-Variables formula above. If we can do this, then computing the log-density reduces to **summing** the log-determinants of the Jacobians of those transformations. Let's ground this idea in an accessible one-dimensional example before moving to higher dimensions.

Specifically, we will consider first a simple tanh-based transformation, which is both invertible and differentiable. The forward function f and its inverse f^{-1} are given by:

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad f^{-1}(x) = \tanh^{-1}(x) = \frac{1}{2} \ln \left(\frac{1+x}{1-x} \right).$$

Let's see what this transformation does to a simple base distribution, such as a standard normal.

```
# 1D change-of-variables example with a tanh transform
num_samples = 10_000
base_normal = Normal(loc=torch.tensor(0.0, device=device), scale=torch.tensor(1.0, device=device))

# We can compute all of the below functions analytically for a given simple transformation:
def forward_transform(u: torch.Tensor) -> torch.Tensor:
    """Smooth, monotonic transform mapping R -> (-1, 1)."""
    return torch.tanh(u)

def inverse_transform(y: torch.Tensor) -> torch.Tensor:
    clipped = torch.clamp(y, min=-0.999_999, max=0.999_999)
    return 0.5 * torch.log((1 + clipped) / (1 - clipped))

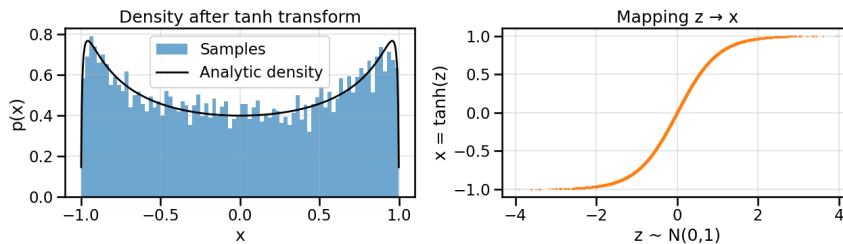
def log_abs_det_jacobian(y: torch.Tensor) -> torch.Tensor:
    return -torch.log1p(-y.pow(2))

with torch.no_grad():
    z_samples = base_normal.sample((num_samples,))
```

```

x_samples = forward_transform(z_samples)
x_grid = torch.linspace(-0.999, 0.999, 500, device=device)
z_grid = inverse_transform(x_grid)
log_px = base_normal.log_prob(z_grid) + log_abs_det_jacobian(x_grid)

```



In this simple transformation, we can see how it modifies the density of a standard normal distribution, and also how the transformation is invertible, since we can go both ways. We can draw a sample from z and pass it through the forward transformation to get x , and we can also take a value of x and map it back to z using the inverse transformation. We can see from the mapping that the transformation is indeed invertible.

14.3.1 Building Intuition Through Simple Transforms

Before we move onto more complex normalizing flows, it will help us to build geometric intuition for individual transformations. We will:

- Look at several common 1D functions that appear in flow architectures to understand how they bend the real line.
- Apply those transforms to a base Gaussian and inspect how the probability density changes.
- Explore higher-dimensional analogues (rotations, couplings, planar flows) to see how they reshape contours while keeping the transformation invertible.

Keep the change-of-variables formula in mind: steep regions of a transform squeeze probability mass, while flat regions stretch it.

```

# Visualizing common 1D flow transformations before touching the density
import torch.nn.functional as F

z_plot = torch.linspace(-3.0, 3.0, 400, device=device)

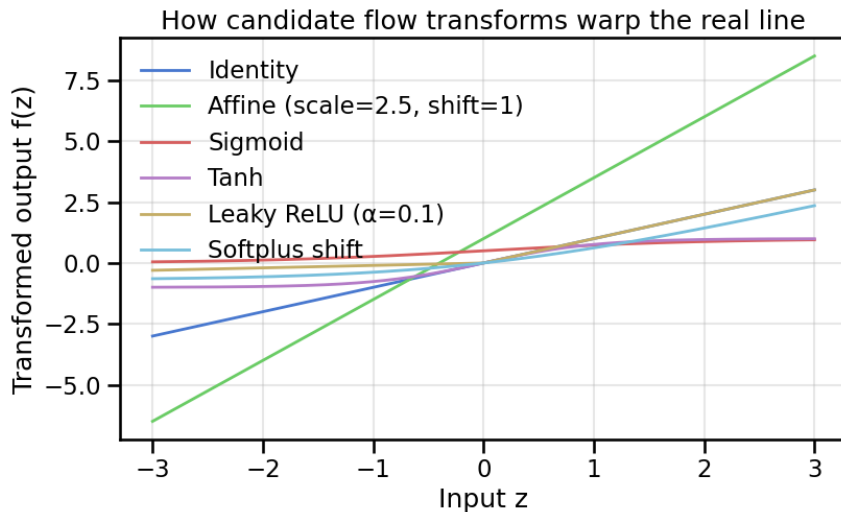
transform_fns = {
    "Identity": lambda z: z,
    "Affine (scale=2.5, shift=1)": lambda z: 2.5 * z + 1.0,
}

```

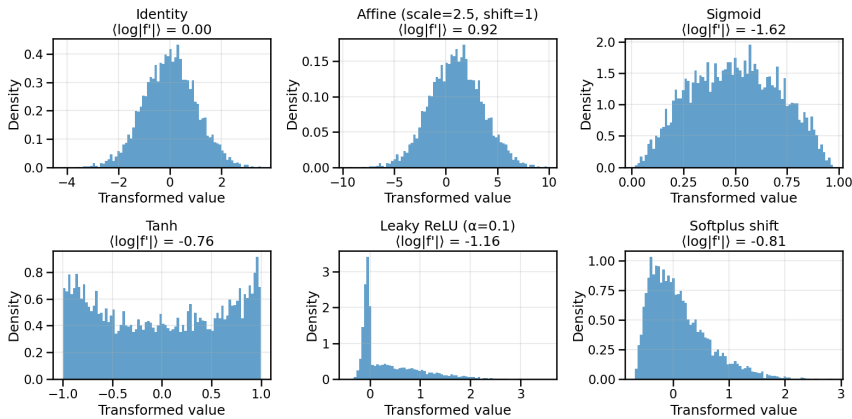
```

"Sigmoid": torch.sigmoid,
"Tanh": torch.tanh,
"Leaky ReLU ( $\alpha=0.1$ )": lambda z: torch.where(z >= 0, z, 0.1 * z),
"Softplus shift": lambda z: F.softplus(z) - np.log(2.0),
}

```



Note how in the above all warpings are monotonic functions, ensuring invertibility. Now let's see how they modify a simple Gaussian base density.



14.3.2 More Complex Transforms

OK, so far so good, and the 1D change of variables is fairly intuitive. But how do we extend this to higher dimensions? For higher dimensions, the change-of-variables formula generalizes

to:

$$p_X(\mathbf{x}) = p_Z(f^{-1}(\mathbf{x})) \left| \det \frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right| = p_Z(\mathbf{z}) \left| \det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right|^{-1}.$$

where $\mathbf{x}, \mathbf{z} \in \mathbb{R}^D$ are now D -dimensional vectors, and the Jacobian (J) $\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}}$ is a $D \times D$ matrix of partial derivatives:

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \begin{bmatrix} \frac{\partial f_1(\mathbf{z})}{\partial z_1} & \frac{\partial f_1(\mathbf{z})}{\partial z_2} & \dots & \frac{\partial f_1(\mathbf{z})}{\partial z_D} \\ \frac{\partial f_2(\mathbf{z})}{\partial z_1} & \frac{\partial f_2(\mathbf{z})}{\partial z_2} & \dots & \frac{\partial f_2(\mathbf{z})}{\partial z_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_D(\mathbf{z})}{\partial z_1} & \frac{\partial f_D(\mathbf{z})}{\partial z_2} & \dots & \frac{\partial f_D(\mathbf{z})}{\partial z_D} \end{bmatrix}$$

So, if we unpack the new formula, we see that the density at a point $\mathbf{x}$ depends on the density at the corresponding point $\mathbf{z} = f^{-1}(\mathbf{x})$ in the base distribution, scaled by the volume change induced by the transformation f at that point, as captured by the **determinant of the Jacobian**. As such, in normalizing flows, most of the effort goes into designing transformations f (or compositions of transformations) that are:

1. **Invertible**: So we can compute $f^{-1}(\mathbf{x})$.
2. **Tractable Jacobian Determinant**: So we can efficiently compute $\log \left| \det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right|$.
3. **Expressive**: So that the composition of multiple such transformations can model complex target distributions.

A Refresher on Determinants

Before going into specific transformations or architectures, let's briefly refresh our understanding of determinants, in general.¹ We will see that many of the common transformations in Normalizing Flows are designed to take advantage of one or more of the following properties of determinants:

- **Product Rule**: For two square matrices $\mathbf{A}$ and $\mathbf{B}$ of the same size, $\det(\mathbf{AB}) = \det(\mathbf{A}) \cdot \det(\mathbf{B})$. This means that for a composition of functions $f = f_2 \circ f_1$, the Jacobian determinant is the product of the individual determinants: $\det(J_f) = \det(J_{f_2}) \cdot \det(J_{f_1})$.
- **Matrix Inversion**: The determinant of the inverse of a matrix is the reciprocal of the determinant: $\det(\mathbf{A}^{-1}) = 1/\det(\mathbf{A})$. This is useful when computing the density of the transformed variable using the inverse transformation.
- **Eigenvalues**: The determinant of a matrix can also be computed as the product of its eigenvalues. This property is sometimes used in flow architectures that involve spectral methods or transformations based on eigen-decompositions.

¹For some common matrix identities, I can recommend reviewing the [Matrix Cookbook](#)

- **Triangular Matrices:** The determinant of a triangular matrix (upper or lower) is simply the product of its diagonal elements: $\det J = \prod_{i=1}^d \frac{\partial f_i}{\partial x_i}$ and the log-determinant is just the sum of those diagonal terms. This property is often exploited in flow architectures to design transformations with triangular Jacobians, making the determinant computation efficient. This is because, for a matrix of size $D \times D$, computing the determinant directly is $O(D^3)$, while for triangular matrices this reduces to $O(D)$. We can see that a diagonal matrix is a special case of a triangular matrix, as thus has the same formula for the determinant.
- **Block Matrices:** For block diagonal matrices, the determinant is the product of the determinants of the blocks. This allows for designing transformations that operate on subsets of dimensions independently, simplifying the Jacobian structure.
- **Orthogonal Matrices:** For orthogonal matrices (where $\mathbf{A}^T \mathbf{A} = \mathbf{I}$), the determinant is either +1 or -1. This property is useful in designing volume-preserving transformations, since we can make sure that the determinant of the Jacobian is one, and thus simplify density computations or ensure that volume doesn't change. This is also useful in special matrices like Permutation matrices, which are orthogonal, and we will see later are useful for shuffling dimensions in coupling layers.
- **Volume Preserving Matrices:** Some transformations are designed to be volume-preserving, meaning that they do not change the volume of regions in space. For such transformations, the determinant of the Jacobian is one. These will have important analogues to continuous flows which we will see in later notebooks, where this property corresponds to divergence-free vector fields ($\nabla \cdot \mathbf{f} = 0$).
- **Low-Rank Updates:** For matrices that can be expressed as a low-rank update to another matrix, the determinant can be computed using the [matrix determinant lemma](#). This is particularly useful in certain flow architectures that involve low-rank transformations, such as planar flows. Specifically, if $\mathbf{A}$ is an invertible matrix and $\mathbf{u}, \mathbf{v}$ are column vectors, then: $\det(\mathbf{A} + \mathbf{u}\mathbf{v}^T) = \det(\mathbf{A}) \cdot (1 + \mathbf{v}^T \mathbf{A}^{-1} \mathbf{u})$. In the special case where $\mathbf{A}$ is the identity matrix, this simplifies to: $\det(\mathbf{I} + \mathbf{u}\mathbf{v}^T) = 1 + \mathbf{v}^T \mathbf{u}$.

With these properties in mind, we can now explore some common transformations used in normalizing flows.

Affine Transforms

These are the simplest layers, applying a linear transformation followed by a shift: $f(\mathbf{z}) = \mathbf{A}\mathbf{z} + \mathbf{b}$. The Jacobian is simply $\mathbf{A}$, so its log-determinant is $\log |\det \mathbf{A}|$. If $\mathbf{A}$ is a rotation matrix, the transform is volume-preserving ($\log |\det| = 0$). If $\mathbf{A}$ is diagonal, it applies axis-aligned scaling.

Planar Flows

These layers stretch and compress the space along a specific direction (a hyperplane). The transformation is $f(\mathbf{z}) = \mathbf{z} + \mathbf{u}h(\mathbf{w}^T\mathbf{z} + b)$, where $\mathbf{u}, \mathbf{w}$ are vectors, b is a scalar, and h is a smooth nonlinearity like $\tanh$. This creates contractions or expansions perpendicular to the hyperplane $\mathbf{w}^T\mathbf{z} + b = 0$. Because this is a low-rank update, the log-determinant can be efficiently computed as $\log |1 + \mathbf{u}^T \mathbf{v}(\mathbf{z})|$, where $\mathbf{v}(\mathbf{z}) = h'(\mathbf{w}^T\mathbf{z} + b)\mathbf{w}$.

Radial Flows

These layers modify the space around a reference point $\mathbf{z}_0$: $f(\mathbf{z}) = \mathbf{z} + \beta \frac{\mathbf{z} - \mathbf{z}_0}{\alpha + r}$, where $r = \|\mathbf{z} - \mathbf{z}_0\|$ is a radius from the reference point and $\alpha > 0, \beta \in \mathbb{R}$ are fixed or learned scalars parameterizing the function. This can create or fill holes in the distribution around $\mathbf{z}_0$ by pushing or pulling points in $\mathbf{z}$ toward/away from $\mathbf{z}_0$. Its Jacobian is given by:

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \mathbf{I} + h(r)\mathbf{I} + h'(r) \frac{(\mathbf{z} - \mathbf{z}_0)(\mathbf{z} - \mathbf{z}_0)^T}{r}$$

where $h(r) = \frac{\beta}{\alpha + r}$ and $h'(r) = -\frac{\beta}{(\alpha + r)^2}$. As in planar flows, this is a low-rank update, so we can use the matrix determinant lemma to compute the determinant efficiently:

$$\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = (1 + h(r))^{D-1} (1 + h(r) + h'(r)r) = \left(1 + \frac{\beta}{\alpha + r}\right)^{D-1} \left(1 + \frac{\beta\alpha}{(\alpha + r)^2}\right)$$

Affine Coupling Layers

We are not only restricted to transforming the entire base distribution in one function. We can also split the base distribution into two parts, $\mathbf{z}_1$ and $\mathbf{z}_2$. One part is used to parameterize an affine transformation on the other:

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{z}_1 \\ \mathbf{y}_2 &= \mathbf{z}_2 \odot \exp(s(\mathbf{z}_1)) + t(\mathbf{z}_1) \end{aligned}$$

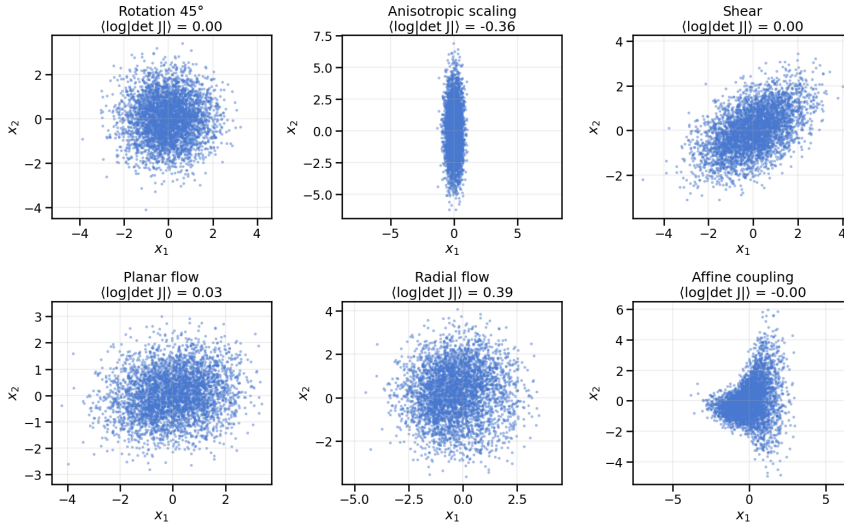
Here, s (scale) and t (translate) are neural networks. Let's assume for this simple example that $\mathbf{z}_1$ and $\mathbf{z}_2$ are each half of the dimensions of $\mathbf{z}$. The inverse transformation is straightforward:

$$\begin{aligned} \mathbf{z}_1 &= \mathbf{y}_1 \\ \mathbf{z}_2 &= (\mathbf{y}_2 - t(\mathbf{y}_1)) \odot \exp(-s(\mathbf{y}_1)) \end{aligned}$$

We can also see that this approach endowed the Jacobian with a triangular structure:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{z}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{y}_2}{\partial \mathbf{z}_1} & \text{diag}(\exp(s(\mathbf{z}_1))) \end{bmatrix}$$

Since the Jacobian is triangular, this makes its determinant just the product of the diagonal elements: $\sum_j s(\mathbf{z}_1)_j$. This is expressive, efficient, and easily invertible. Of course the cost of doing this is that we can only transform half of the dimensions in one layer, leaving the rest unchanged. To address this, we can stack multiple coupling layers, alternating which half of the dimensions are transformed in each layer. Additionally, we can add permutation layers (which are orthogonal and volume-preserving) between coupling layers to shuffle the dimensions, ensuring that all dimensions get transformed over multiple layers.



14.3.3 From Transforms to Complex Flows and Probabilities

In practice, we can stack many such layers of different types to build a deep normalizing flow that can model complex target distributions. Each layer contributes to the overall transformation, and we can compute the log-density of the transformed variable by summing the log-determinants of each layer's Jacobian. Each transform above comes with a Jacobian determinant that tells us how probability mass re-scales. In one dimension, the recipe is

$$\log p_Y(y) = \log p_Z(f^{-1}(y)) + \log \left| \frac{df^{-1}(y)}{dy} \right| = \log p_Z(z) - \log |f'(z)|.$$

For multivariate flows with $x = f(z)$, we track the log-determinant at every layer:

$$\log p_X(x) = \log p_Z(z_0) + \sum_{\ell=1}^K \log |\det J_{f_\ell}(z_{\ell-1})|,$$

Computing these terms is why we prefer architectures (like affine coupling, planar, or radial flows) with Jacobians that are either triangular or have closed-form determinants, so that we can avoid very expensive determinant calculations. When training a flow we maximise this exact log-likelihood, and during sampling we simply draw $z \sim p_Z$ and apply the inverses f_ℓ^{-1} in reverse order.

14.4 Constructing a Flow with Coupling Layers using RealNVP

One of the first models for building tractable flows was the RealNVP (Real-valued Non-Volume Preserving) architecture. It splits the vector into two parts, using a coupling layer as we saw previously. This leaves one part unchanged, and uses it to condition an affine transformation on the other part. Because the transformation is triangular, the Jacobian determinant is the product of diagonal entries and is therefore easy to compute. Below implements a RealNVP flow using coupling layers where we swap in each layer which one of the two input dimensions is fixed. For a larger or more complex example with more inputs, these could be selected randomly using permutations, but our simple approach below is just to illustrate the concept.

```
class AffineCoupling(nn.Module):
    """RealNVP-style affine coupling layer for 2D inputs."""
    def __init__(self, dim: int, hidden_dim: int, mask: torch.Tensor):
        super().__init__()
        self.dim = dim
        self.register_buffer('mask', mask)
        self.scale_net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, dim)
        )
        self.translate_net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, dim)
        )
        nn.init.zeros_(self.scale_net[-1].weight)
        nn.init.zeros_(self.scale_net[-1].bias)
        nn.init.zeros_(self.translate_net[-1].weight)
        nn.init.zeros_(self.translate_net[-1].bias)
```

```

def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    x_masked = x * self.mask
    s = self.scale_net(x_masked) * (1 - self.mask)
    t = self.translate_net(x_masked) * (1 - self.mask)
    s = torch.tanh(s)
    y = x_masked + (1 - self.mask) * (x * torch.exp(s) + t)
    log_det = ((1 - self.mask) * s).sum(dim=1)
    return y, log_det

def inverse(self, y: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    y_masked = y * self.mask
    s = self.scale_net(y_masked) * (1 - self.mask)
    t = self.translate_net(y_masked) * (1 - self.mask)
    s = torch.tanh(s)
    x = y_masked + (1 - self.mask) * ((y - t) * torch.exp(-s))
    log_det = -((1 - self.mask) * s).sum(dim=1)
    return x, log_det

class RealNVP(nn.Module):
    """Stack of affine coupling layers implementing a normalizing flow."""
    def __init__(self, dim: int = 2, hidden_dim: int = 128, n_layers: int = 6):
        super().__init__()
        masks: List[torch.Tensor] = []
        for i in range(n_layers):
            if i % 2 == 0:
                mask = torch.tensor([1.0, 0.0])
            else:
                mask = torch.tensor([0.0, 1.0])
            masks.append(mask)
        self.layers = nn.ModuleList([
            AffineCoupling(dim=dim, hidden_dim=hidden_dim, mask=mask)
            for mask in masks
        ])

    def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        log_det_total = torch.zeros(x.shape[0], device=x.device)
        z = x
        for layer in self.layers:
            z, log_det = layer(z)
            # Note here how we can just accumulate log-determinants forward:
            log_det_total = log_det_total + log_det

```

```

        return z, log_det_total

    def inverse(self, z: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        log_det_total = torch.zeros(z.shape[0], device=z.device)
        x = z
        for layer in reversed(self.layers):
            x, log_det = layer.inverse(x)
            # Likewise, here we can accumulate log-determinants backward:
            log_det_total = log_det_total + log_det
        return x, log_det_total

    def inverse_with_intermediates(self, z: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor, List[torch.Tensor]]:
        """Inverse transformation that also returns intermediate layer outputs."""
        log_det_total = torch.zeros(z.shape[0], device=z.device)
        x = z
        intermediates = [z.clone()]
        for layer in reversed(self.layers):
            x, log_det = layer.inverse(x)
            log_det_total = log_det_total + log_det
            intermediates.append(x.clone())
        return x, log_det_total, intermediates

    def log_prob(self, x: torch.Tensor, base_dist: MultivariateNormal) -> torch.Tensor:
        z, log_det = self.forward(x)
        log_prob_base = base_dist.log_prob(z)
        # Now we can just combine base log-probability and log-determinant, which we accumulated earlier
        return log_prob_base + log_det

    def sample(self, num_samples: int, base_dist: MultivariateNormal) -> torch.Tensor:
        with torch.no_grad():
            z = base_dist.sample((num_samples,))
            x, _ = self.inverse(z)
        return x

```

14.4.1 Training a Flow on the Ring-of-Gaussians Dataset

We now train a small RealNVP model on the same ring dataset used in the VAE notebook. Unlike the VAE, the flow optimizes the exact log-likelihood of the data, so we do not need a separate encoder or a variational bound. The key ingredients are:

1. A base density $p_0(z)$ (we use a standard 2D Gaussian).
2. A flow f_θ composed of coupling layers.

3. An optimizer to maximize $\log p_X(x) = \log p_0(f(x)) + \log |\det J_f(x)|$.

```
@dataclass
class FlowHistory:
    loss: List[float]
    log_prob: List[float]
    diversity: List[float]

def train_flow_model(
    data: np.ndarray,
    flow: RealNVP,
    base_dist: MultivariateNormal,
    epochs: int = 300,
    batch_size: int = 256,
    lr: float = 1e-3,
    print_every: int = 50
) -> Tuple[RealNVP, FlowHistory]:
    flow.train()
    loader = make_loader(data, batch_size)
    optimizer = optim.Adam(flow.parameters(), lr=lr)
    history = FlowHistory(loss=[], log_prob=[], diversity=[])

    for epoch in range(epochs):
        batch_losses = []
        batch_log_probs = []
        for (batch,) in loader:
            batch = batch.float().to(device)
            log_prob = flow.log_prob(batch, base_dist)
            loss = -log_prob.mean()
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            batch_losses.append(loss.item())
            batch_log_probs.append(log_prob.mean().item())

        mean_loss = float(np.mean(batch_losses))
        mean_log_prob = float(np.mean(batch_log_probs))

        with torch.no_grad():
            samples = flow.sample(2048, base_dist)
            diversity = compute_diversity_metric(samples)
```

```

        history.loss.append(mean_loss)
        history.log_prob.append(mean_log_prob)
        history.diversity.append(diversity)

        if (epoch + 1) % print_every == 0 or epoch == 1:
            print(f'Epoch {epoch + 1:03d}/{epochs} | Loss: {mean_loss:.3f} | Log p(x): {mean_log_prob:.3f} | D

    flow.eval()
    return flow, history

X_ring, y_ring = create_ring_gaussians(n_samples=2000)

```

You can modify the below RealNVP parameters (number of layers, hidden units) and retrain the flow to see how it affects expressiveness and training time, as we mention for the experiment at the end of the notebook.

```

# Train the flow on the ring dataset
flow = RealNVP(dim=2, hidden_dim=128, n_layers=6).to(device)
base_dist = MultivariateNormal(
    loc=torch.zeros(2, device=device),
    covariance_matrix=torch.eye(2, device=device)
)
flow, flow_history = train_flow_model(
    data=X_ring,
    flow=flow,
    base_dist=base_dist,
    epochs=300,
    batch_size=256,
    lr=1e-3,
    print_every=50
)

```

```

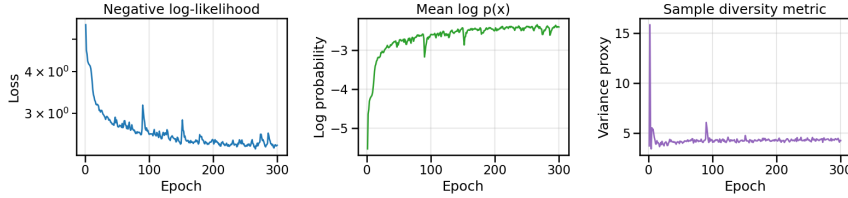
Epoch 002/300 | Loss: 4.634 | Log p(x): -4.634 | Div: 15.833
Epoch 050/300 | Loss: 2.800 | Log p(x): -2.800 | Div: 4.141
Epoch 050/300 | Loss: 2.800 | Log p(x): -2.800 | Div: 4.141
Epoch 100/300 | Loss: 2.591 | Log p(x): -2.591 | Div: 4.106
Epoch 100/300 | Loss: 2.591 | Log p(x): -2.591 | Div: 4.106
Epoch 150/300 | Loss: 2.430 | Log p(x): -2.430 | Div: 4.275
Epoch 150/300 | Loss: 2.430 | Log p(x): -2.430 | Div: 4.275
Epoch 200/300 | Loss: 2.490 | Log p(x): -2.490 | Div: 4.298
Epoch 200/300 | Loss: 2.490 | Log p(x): -2.490 | Div: 4.298
Epoch 250/300 | Loss: 2.383 | Log p(x): -2.383 | Div: 4.308

```

Epoch 250/300 | Loss: 2.383 | Log $p(x)$: -2.383 | Div: 4.308
Epoch 300/300 | Loss: 2.401 | Log $p(x)$: -2.401 | Div: 4.274
Epoch 300/300 | Loss: 2.401 | Log $p(x)$: -2.401 | Div: 4.274

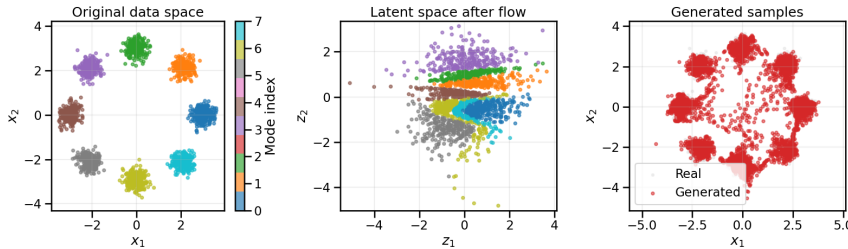
14.4.2 Monitoring Training Dynamics

The total loss is the negative log-likelihood (NLL). Because we optimize the exact density, improvements in the loss correspond directly to better calibrated probabilities. We also track a simple diversity statistic computed from generated samples to monitor coverage.



14.4.3 Inspecting the Learned Mapping

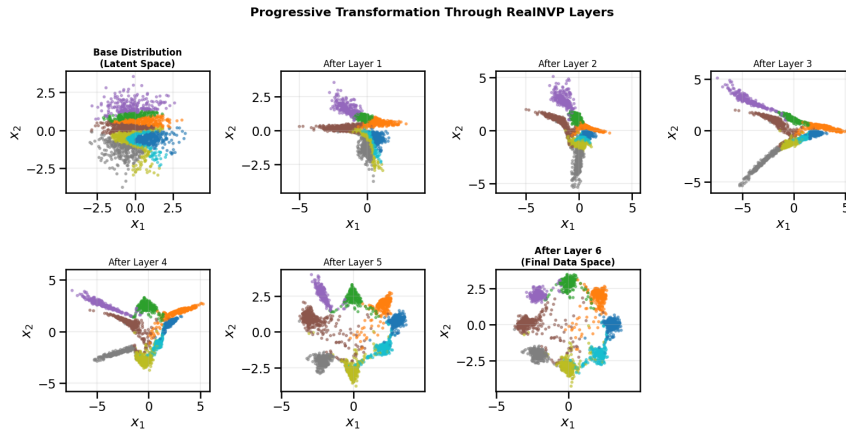
With a trained flow we can examine three complementary views: the data space, the latent space obtained by applying the forward transformation $f(x)$, and samples generated by pushing base noise through the inverse transformation $f^{-1}(z)$. A well-trained flow should map the multimodal ring distribution onto a near-Gaussian latent distribution while recovering the ring geometry when sampling.



14.4.4 Layer-by-Layer Transformation

To better understand how the normalizing flow works, let's visualize how each coupling layer progressively transforms the base distribution into the target distribution. We'll sample from the base Gaussian and then apply each layer's inverse transformation one at a time, coloring the points by their eventual mode assignment (determined by which ring Gaussian they end up closest to).

This visualization shows the geometric action of each layer, illustrating how the flow gradually warps the simple base distribution into the complex ring structure. Note in particular how the RealNVP coupling layers show up here: in the first layer only one of the dimensions is changed, while the other remains fixed. In the next layer, the roles are swapped, and so on.



💡 Experiment: Effect of Flow Depth and Width

Go back up to where we defined the RealNVP architecture and try it again with the following changes and note the behavior that you see:

- Vary the number of coupling layers (e.g., 2, 4, 8, 16). What happens to the quality of the learned distribution and the training time?
- What does varying the number of hidden units in the scale and translate networks do to the expressiveness and stability of training?
- As you make the number of coupling layers extremely large (50,100), what do you notice about the training dynamics as well as the effect of individual layers on the movement of the distribution?

14.5 Conditional Generation Examples

Normalizing flows can also be extended to conditional generation by incorporating conditioning information into the transformation layers. This allows us to learn conditional densities $p(x|y)$ while maintaining exact likelihood computation and efficient sampling.

The key idea is to condition the coupling layers on additional information y . For discrete conditioning (like one-hot labels), we concatenate the label to the unchanged part of the input before passing it through the scale and translation networks. For continuous conditioning (like radius), we embed the scalar and concatenate it similarly.

This approach preserves the invertibility of the flow while allowing the transformations to adapt based on the conditioning signal.

14.5.1 Conditional Example A — One-Hot Mode Conditioning

We'll first implement a conditional affine coupling layer that takes both the data and a one-hot label as input. The label is concatenated with the masked input before computing the scale and translation parameters.

```
# Conditional affine coupling layer
class ConditionalAffineCoupling(nn.Module):
    def __init__(self, dim: int, cond_dim: int, hidden_dim: int, mask: torch.Tensor):
        super().__init__()
        self.dim = dim
        self.register_buffer('mask', mask)
        # Condition on both masked input and label
        self.scale_net = nn.Sequential(
            nn.Linear(dim + cond_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, dim)
        )
        self.translate_net = nn.Sequential(
            nn.Linear(dim + cond_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, dim)
        )
        nn.init.zeros_(self.scale_net[-1].weight)
        nn.init.zeros_(self.scale_net[-1].bias)
        nn.init.zeros_(self.translate_net[-1].weight)
        nn.init.zeros_(self.translate_net[-1].bias)

    def forward(self, x: torch.Tensor, y: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        x_masked = x * self.mask
        xy = torch.cat([x_masked, y], dim=1)
        s = self.scale_net(xy) * (1 - self.mask)
        t = self.translate_net(xy) * (1 - self.mask)
        s = torch.tanh(s)
        out = x_masked + (1 - self.mask) * (x * torch.exp(s) + t)
```



```

log_det = ((1 - self.mask) * s).sum(dim=1)
return out, log_det

def inverse(self, y_out: torch.Tensor, y_cond: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    y_masked = y_out * self.mask
    yc = torch.cat([y_masked, y_cond], dim=1)
    s = self.scale_net(yc) * (1 - self.mask)
    t = self.translate_net(yc) * (1 - self.mask)
    s = torch.tanh(s)
    x = y_masked + (1 - self.mask) * ((y_out - t) * torch.exp(-s))
    log_det = -((1 - self.mask) * s).sum(dim=1)
    return x, log_det

# Conditional RealNVP
class ConditionalRealNVP(nn.Module):
    def __init__(self, dim: int = 2, cond_dim: int = 8, hidden_dim: int = 128, n_layers: int = 6):
        super().__init__()
        masks = []
        for i in range(n_layers):
            mask = torch.tensor([1.0, 0.0]) if i % 2 == 0 else torch.tensor([0.0, 1.0])
            masks.append(mask)
        self.layers = nn.ModuleList([
            ConditionalAffineCoupling(dim=dim, cond_dim=cond_dim, hidden_dim=hidden_dim, mask=mask)
            for mask in masks
        ])

    def forward(self, x: torch.Tensor, y: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        log_det_total = torch.zeros(x.shape[0], device=x.device)
        z = x
        for layer in self.layers:
            z, log_det = layer(z, y)
            log_det_total = log_det_total + log_det
        return z, log_det_total

    def inverse(self, z: torch.Tensor, y: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        log_det_total = torch.zeros(z.shape[0], device=z.device)
        x = z
        for layer in reversed(self.layers):
            x, log_det = layer.inverse(x, y)
            log_det_total = log_det_total + log_det
        return x, log_det_total

```

```

def log_prob(self, x: torch.Tensor, y: torch.Tensor, base_dist: MultivariateNormal) -> torch.Tensor:
    z, log_det = self.forward(x, y)
    log_prob_base = base_dist.log_prob(z)
    return log_prob_base + log_det

def sample(self, y: torch.Tensor, base_dist: MultivariateNormal) -> torch.Tensor:
    with torch.no_grad():
        z = base_dist.sample((y.shape[0],))
        x, _ = self.inverse(z, y)
    return x

def train_conditional_flow_onehot(
    data: np.ndarray, labels: np.ndarray, n_classes: int = 8,
    epochs: int = 300, batch_size: int = 256, lr: float = 1e-3, print_every: int = 50
):
    loader = make_conditional_loader_onehot(data, labels, batch_size, n_classes)
    flow = ConditionalRealNVP(dim=2, cond_dim=n_classes, hidden_dim=128, n_layers=6).to(device)
    base_dist = MultivariateNormal(
        loc=torch.zeros(2, device=device),
        covariance_matrix=torch.eye(2, device=device)
    )
    optimizer = optim.Adam(flow.parameters(), lr=lr)
    history = FlowHistory(loss=[], log_prob=[], diversity=[])

    for epoch in range(epochs):
        batch_losses, batch_log_probs = [], []
        for xb, yb in loader:
            xb, yb = xb.to(device), yb.to(device)
            log_prob = flow.log_prob(xb, yb, base_dist)
            loss = -log_prob.mean()
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            batch_losses.append(loss.item())
            batch_log_probs.append(log_prob.mean().item())

        with torch.no_grad():
            y_sample = torch.nn.functional.one_hot(
                torch.randint(0, n_classes, (2048,), device=device), num_classes=n_classes
            ).to(torch.float32)
            samples = flow.sample(y_sample, base_dist)
            diversity = compute_diversity_metric(samples)

```

```

        history.loss.append(float(np.mean(batch_losses)))
        history.log_prob.append(float(np.mean(batch_log_probs)))
        history.diversity.append(diversity)

    if (epoch + 1) % print_every == 0 or epoch == 0:
        print(f'Epoch {epoch + 1:03d}/{epochs} | Loss: {history.loss[-1]:.3f} | '
              f'Log p(x|y): {history.log_prob[-1]:.3f} | Div: {diversity:.3f}')

    flow.eval()
    return flow, base_dist, history

# Train conditional flow
cflow_onehot, cflow_base_dist, cflow_hist_onehot = train_conditional_flow_onehot(
    X_ring, y_ring, n_classes=8, epochs=300, batch_size=256, lr=1e-3, print_every=50
)

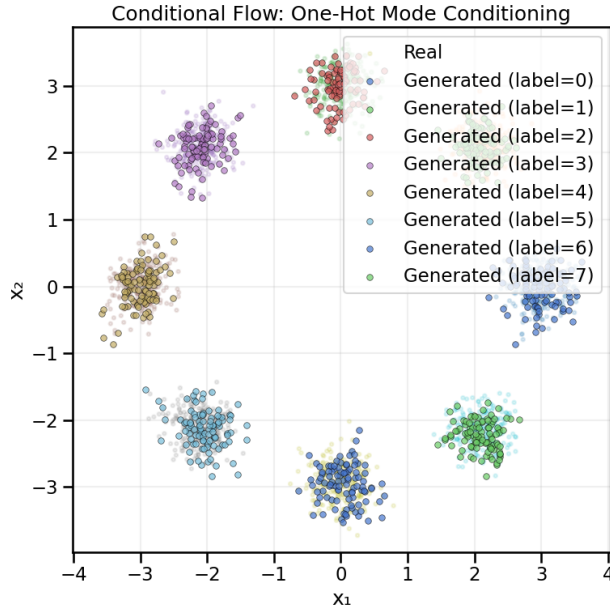
```

```

Epoch 001/300 | Loss: 5.696 | Log p(x|y): -5.696 | Div: 1.979
Epoch 050/300 | Loss: 0.038 | Log p(x|y): -0.038 | Div: 4.453
Epoch 050/300 | Loss: 0.038 | Log p(x|y): -0.038 | Div: 4.453
Epoch 100/300 | Loss: 0.034 | Log p(x|y): -0.034 | Div: 4.593
Epoch 100/300 | Loss: 0.034 | Log p(x|y): -0.034 | Div: 4.593
Epoch 150/300 | Loss: 0.031 | Log p(x|y): -0.031 | Div: 4.486
Epoch 150/300 | Loss: 0.031 | Log p(x|y): -0.031 | Div: 4.486
Epoch 200/300 | Loss: 0.026 | Log p(x|y): -0.026 | Div: 4.487
Epoch 200/300 | Loss: 0.026 | Log p(x|y): -0.026 | Div: 4.487
Epoch 250/300 | Loss: 0.026 | Log p(x|y): -0.026 | Div: 4.538
Epoch 250/300 | Loss: 0.026 | Log p(x|y): -0.026 | Div: 4.538
Epoch 300/300 | Loss: 0.030 | Log p(x|y): -0.030 | Div: 4.522
Epoch 300/300 | Loss: 0.030 | Log p(x|y): -0.030 | Div: 4.522

```

Let's visualize the conditional samples generated by the conditional flow for each mode:



14.5.2 Conditional Example B — Continuous Radius Conditioning

Now let's condition the flow on a continuous scalar (radius). We'll embed the radius through a small MLP before concatenating it with the masked input in the coupling layers.

```
# Conditional affine coupling with radius embedding
class ConditionalAffineCouplingRadius(nn.Module):
    def __init__(self, dim: int, cond_emb_dim: int, hidden_dim: int, mask: torch.Tensor):
        super().__init__()
        self.dim = dim
        self.register_buffer('mask', mask)
        self.cond_embed = nn.Sequential(
            nn.Linear(1, cond_emb_dim),
            nn.ReLU(),
            nn.Linear(cond_emb_dim, cond_emb_dim),
        )
        self.scale_net = nn.Sequential(
            nn.Linear(dim + cond_emb_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, dim)
        )
        self.translate_net = nn.Sequential(
```

```

        nn.Linear(dim + cond_emb_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, dim)
    )
    nn.init.zeros_(self.scale_net[-1].weight)
    nn.init.zeros_(self.scale_net[-1].bias)
    nn.init.zeros_(self.translate_net[-1].weight)
    nn.init.zeros_(self.translate_net[-1].bias)

    def forward(self, x: torch.Tensor, r: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        x_masked = x * self.mask
        r_emb = self.cond_embed(r)
        xr = torch.cat([x_masked, r_emb], dim=1)
        s = self.scale_net(xr) * (1 - self.mask)
        t = self.translate_net(xr) * (1 - self.mask)
        s = torch.tanh(s)
        out = x_masked + (1 - self.mask) * (x * torch.exp(s) + t)
        log_det = ((1 - self.mask) * s).sum(dim=1)
        return out, log_det

    def inverse(self, y_out: torch.Tensor, r: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        y_masked = y_out * self.mask
        r_emb = self.cond_embed(r)
        yr = torch.cat([y_masked, r_emb], dim=1)
        s = self.scale_net(yr) * (1 - self.mask)
        t = self.translate_net(yr) * (1 - self.mask)
        s = torch.tanh(s)
        x = y_masked + (1 - self.mask) * ((y_out - t) * torch.exp(-s))
        log_det = -((1 - self.mask) * s).sum(dim=1)
        return x, log_det

# Conditional RealNVP with radius
class ConditionalRealNVPRadius(nn.Module):
    def __init__(self, dim: int = 2, cond_emb_dim: int = 16, hidden_dim: int = 128, n_layers: int = 6):
        super().__init__()
        masks = []
        for i in range(n_layers):
            mask = torch.tensor([1.0, 0.0]) if i % 2 == 0 else torch.tensor([0.0, 1.0])
            masks.append(mask)
        self.layers = nn.ModuleList([

```

```

        ConditionalAffineCouplingRadius(dim=dim, cond_emb_dim=cond_emb_dim, hidden_dim=hidden_dim, mask=mask)
        for mask in masks:
    ])

def forward(self, x: torch.Tensor, r: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    log_det_total = torch.zeros(x.shape[0], device=x.device)
    z = x
    for layer in self.layers:
        z, log_det = layer(z, r)
        log_det_total = log_det_total + log_det
    return z, log_det_total

def inverse(self, z: torch.Tensor, r: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
    log_det_total = torch.zeros(z.shape[0], device=z.device)
    x = z
    for layer in reversed(self.layers):
        x, log_det = layer.inverse(x, r)
        log_det_total = log_det_total + log_det
    return x, log_det_total

def log_prob(self, x: torch.Tensor, r: torch.Tensor, base_dist: MultivariateNormal) -> torch.Tensor:
    z, log_det = self.forward(x, r)
    log_prob_base = base_dist.log_prob(z)
    return log_prob_base + log_det

def sample(self, r: torch.Tensor, base_dist: MultivariateNormal) -> torch.Tensor:
    with torch.no_grad():
        z = base_dist.sample((r.shape[0],))
        x, _ = self.inverse(z, r)
    return x

def train_conditional_flow_radius(
    data: np.ndarray, epochs: int = 300, batch_size: int = 256, lr: float = 1e-3, print_every: int = 50
):
    loader = make_conditional_loader_radius(data, batch_size)
    flow = ConditionalRealNVPRadius(dim=2, cond_emb_dim=16, hidden_dim=128, n_layers=6).to(device)
    base_dist = MultivariateNormal(
        loc=torch.zeros(2, device=device),
        covariance_matrix=torch.eye(2, device=device)
    )
    optimizer = optim.Adam(flow.parameters(), lr=lr)
    history = FlowHistory(loss=[], log_prob=[], diversity=[])

```

```

for epoch in range(epochs):
    batch_losses, batch_log_probs = [], []
    for xb, rb in loader:
        xb, rb = xb.to(device), rb.to(device)
        log_prob = flow.log_prob(xb, rb, base_dist)
        loss = -log_prob.mean()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        batch_losses.append(loss.item())
        batch_log_probs.append(log_prob.mean().item())

    with torch.no_grad():
        r_sample = torch.norm(torch.randn(2048, 2), dim=1, keepdim=True).to(device)
        samples = flow.sample(r_sample, base_dist)
        diversity = compute_diversity_metric(samples)

    history.loss.append(float(np.mean(batch_losses)))
    history.log_prob.append(float(np.mean(batch_log_probs)))
    history.diversity.append(diversity)

    if (epoch + 1) % print_every == 0 or epoch == 0:
        print(f'Epoch {epoch + 1:03d}/{epochs} | Loss: {history.loss[-1]:.3f} | '
              f'Log p(x|r): {history.log_prob[-1]:.3f} | Div: {diversity:.3f}')

    flow.eval()
    return flow, base_dist, history

# Train conditional flow with radius
cflow_radius, cflow_base_dist_r, cflow_hist_radius = train_conditional_flow_radius(
    X_ring, epochs=1000, batch_size=256, lr=1e-3, print_every=50
)

```

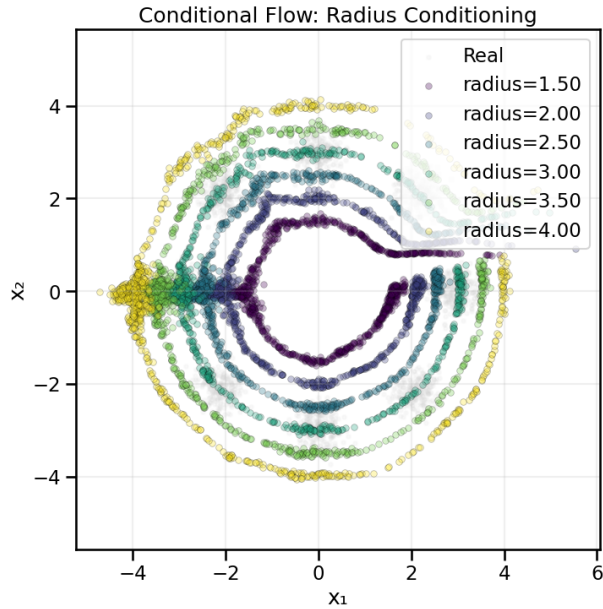
```

Epoch 001/1000 | Loss: 5.550 | Log p(x|r): -5.550 | Div: 1.782
Epoch 050/1000 | Loss: 3.520 | Log p(x|r): -3.520 | Div: 1.769
Epoch 050/1000 | Loss: 3.520 | Log p(x|r): -3.520 | Div: 1.769
Epoch 100/1000 | Loss: 2.914 | Log p(x|r): -2.914 | Div: 1.207
Epoch 100/1000 | Loss: 2.914 | Log p(x|r): -2.914 | Div: 1.207
Epoch 150/1000 | Loss: 2.231 | Log p(x|r): -2.231 | Div: 1.432
Epoch 150/1000 | Loss: 2.231 | Log p(x|r): -2.231 | Div: 1.432
Epoch 200/1000 | Loss: 1.646 | Log p(x|r): -1.646 | Div: 1.463

```

Epoch 200/1000	Loss: 1.646	Log p(x r): -1.646	Div: 1.463
Epoch 250/1000	Loss: 1.415	Log p(x r): -1.415	Div: 1.364
Epoch 250/1000	Loss: 1.415	Log p(x r): -1.415	Div: 1.364
Epoch 300/1000	Loss: 1.422	Log p(x r): -1.422	Div: 1.379
Epoch 300/1000	Loss: 1.422	Log p(x r): -1.422	Div: 1.379
Epoch 350/1000	Loss: 1.006	Log p(x r): -1.006	Div: 1.336
Epoch 350/1000	Loss: 1.006	Log p(x r): -1.006	Div: 1.336
Epoch 400/1000	Loss: 1.299	Log p(x r): -1.299	Div: 1.354
Epoch 400/1000	Loss: 1.299	Log p(x r): -1.299	Div: 1.354
Epoch 450/1000	Loss: 0.982	Log p(x r): -0.982	Div: 1.283
Epoch 450/1000	Loss: 0.982	Log p(x r): -0.982	Div: 1.283
Epoch 500/1000	Loss: 1.962	Log p(x r): -1.962	Div: 1.277
Epoch 500/1000	Loss: 1.962	Log p(x r): -1.962	Div: 1.277
Epoch 550/1000	Loss: 0.960	Log p(x r): -0.960	Div: 1.249
Epoch 550/1000	Loss: 0.960	Log p(x r): -0.960	Div: 1.249
Epoch 600/1000	Loss: 1.101	Log p(x r): -1.101	Div: 1.212
Epoch 600/1000	Loss: 1.101	Log p(x r): -1.101	Div: 1.212
Epoch 650/1000	Loss: 1.320	Log p(x r): -1.320	Div: 1.249
Epoch 650/1000	Loss: 1.320	Log p(x r): -1.320	Div: 1.249
Epoch 700/1000	Loss: 0.844	Log p(x r): -0.844	Div: 1.241
Epoch 700/1000	Loss: 0.844	Log p(x r): -0.844	Div: 1.241
Epoch 750/1000	Loss: 1.324	Log p(x r): -1.324	Div: 1.296
Epoch 750/1000	Loss: 1.324	Log p(x r): -1.324	Div: 1.296
Epoch 800/1000	Loss: 0.794	Log p(x r): -0.794	Div: 1.246
Epoch 800/1000	Loss: 0.794	Log p(x r): -0.794	Div: 1.246
Epoch 850/1000	Loss: 1.003	Log p(x r): -1.003	Div: 1.241
Epoch 850/1000	Loss: 1.003	Log p(x r): -1.003	Div: 1.241
Epoch 900/1000	Loss: 0.865	Log p(x r): -0.865	Div: 1.259
Epoch 900/1000	Loss: 0.865	Log p(x r): -0.865	Div: 1.259
Epoch 950/1000	Loss: 1.090	Log p(x r): -1.090	Div: 1.219
Epoch 950/1000	Loss: 1.090	Log p(x r): -1.090	Div: 1.219
Epoch 1000/1000	Loss: 0.639	Log p(x r): -0.639	Div: 1.280
Epoch 1000/1000	Loss: 0.639	Log p(x r): -0.639	Div: 1.280

Let's generate samples at different radius values and visualize the results:



14.6 Advantages and Limitations of Normalizing Flows

Strengths	Limitations
Exact log-likelihood evaluation enables principled model comparison	Computing Jacobians can be expensive for high-dimensional data
Invertible architecture provides bidirectional mapping without a separate encoder	Designing expressive yet tractable transforms is challenging
Amenable to gradient-based training with standard optimizers	Coupling layers may struggle with highly non-local dependencies
Samples are sharp because no explicit reconstruction loss is used	Memory footprint grows with the number of layers

Normalizing flows shine when calibrated densities and invertible mappings are required, such as uncertainty-aware control or anomaly detection in sensor networks. However, they can become cumbersome in very high dimensions or when local affine transformations are insufficient.

14.7 Summary and Looking Ahead

- Normalizing flows learn an exact, differentiable transformation between a base distribution and the data, turning density modeling into a sequence of Jacobian adjustments.
- The change-of-variables formula provides the theoretical backbone, allowing us to evaluate log-likelihoods exactly.
- Coupling layers (as in RealNVP) offer an efficient way to construct expressive flows with tractable inverses and determinants.
- On the ring dataset, flows match the multimodal structure while keeping a simple latent Gaussian distribution.

If you ran the last experiment, you may have noticed that as we increase the number of layers, the individual transformations become smaller and more incremental. What would happen if we set the number of layers to infinity? In this case, each individual transformation would become an almost infinitesimal transformation from one probability distribution to another, and the integral over all of those many transformations would get us the flow from z to x . This is exactly the idea behind *continuous normalizing flows* or neural ODE-based flows, where the model evolves samples through a learned vector field. Introducing stochastic dynamics yields *stochastic flows* that blend diffusion models with flow-based ideas. These extensions retain the change-of-variables logic while trading discrete layers for continuous-time dynamics, setting the stage for the next chapter.

15 From Discrete Transformations to Continuous Flows

We saw in the last chapter how normalizing flows transform simple base distributions into complex target distributions via a sequence of invertible mappings. Each mapping requires computing the determinant of the Jacobian to track how densities change.

In continuous normalizing flows, we ask: what happens if we let the number of layers go to infinity while shrinking each individual transformation to an infinitesimal step? That is, rather than a handful of large invertible maps, we stack infinitely many tiny, smooth updates. We will see that taking this limit produces an ordinary differential equation (ODE) whose solution transports probability mass.

15.1 Learning Objectives

By the end of this notebook you should be able to:

- Recognize that a CNF is the infinitesimal limit of a discrete normalizing flow.
- Explain how the log-density evolves through the divergence of a velocity field.
- Derive the continuity equation from the change-of-variables formula.
- Simulate simple continuous flows and verify how densities change.
- Connect CNFs to probability transport ideas and the diffusion models we will meet later.

15.2 From Finite Jacobians to Differential Updates

As we saw last chapter, in a discrete flow the change-of-variables formula is:

$$\log p(x) = \log p(z_K) = \log p(z_0) - \sum_{i=1}^K \log |\det J_{f_i}(z_{i-1})|.$$

for a sequence of K invertible maps f_i transforming a base sample z_0 to a target sample $x = z_K$. Here, J_{f_i} is the Jacobian of the i -th transformation.

If we instead consider an infinite sequence of infinitesimal transformations, we can think of each layer as a tiny residual step: $z_{i+1} = z_i + \varepsilon v(z_i, t_i)$ where ε is a small step size, v is a velocity field, and t_i is a time index. The change in probability density can then be thought of as an integral over these infinitesimal changes, like particles flowing through a fluid governed by the velocity field. The main learning challenge here is to learn the right velocity field $v(z, t)$ that transports our simple base distribution to our desired target distribution over time, and also to solve this ODE inexpensively.

Concretely, rather than consider a single step of the change of variables formula, we consider the limit as $\varepsilon \rightarrow 0$ of many small steps, which gives rise to the *instantaneous change of variables* formula^[1]:

$$\frac{\partial}{\partial t} \log p(z(t)) = -\text{tr}(\nabla_x v(z(t), t)),$$

where $\nabla_x v$ is the Jacobian of the velocity field with respect to z , and tr denotes the trace of a matrix.¹

Now, given a vector field $v(z, t)$, we can compute how a base density $p(z_0)$ evolves into the final distribution $p(x) = p(z_K)$ via:

$$\log p(x) = \log p(z_0) - \int_0^T \text{tr}(\nabla_x v(z(t), t)) \, dt.$$

We can think of this as starting a distribution at time $t = 0$ (i.e., at $p(z_0)$) and then integrating it through the velocity field $v(z, t)$ until time $t = T$, where we hope to recover the distribution of the training data (i.e., $p(x)$). To find the best velocity field, we can optimize the above expression for $\log p(x)$ with respect to the parameters of $v(z, t)$ using maximum likelihood. For neural network-based parameterizations of $v(z, t)$ we would parameterize the velocity field via neural network weights and denote this $v_\theta(z, t)$ where θ are the neural network weights. The main requirement we have for the velocity field $v(z, t)$ is that it (and its first derivative) must be Lipschitz continuous so that the resulting ODE has a unique solution (and thus can be in principle inverted as well). We can enforce the Lipschitz continuity of v_θ by choosing activations that have bounded Lipschitz constants, and then further control the smoothness of the velocity field by regularizing the weights (e.g., via spectral normalization).

Putting all of this together, and following the notation/construction of [Grathwohl et al. 2019](#), we can summarize the continuous normalizing flow (CNF) model as the following Initial Value Problem (IVP), based on the above ODE:

$$\underbrace{\begin{bmatrix} \mathbf{z}_0 \\ \log p(\mathbf{x}) - \log p_{\mathbf{z}_0}(\mathbf{z}_0) \end{bmatrix}}_{\text{solutions}} = \int_{t_1}^{t_0} \underbrace{\begin{bmatrix} v_\theta(\mathbf{z}(t), t) \\ -\text{tr}\left(\frac{\partial v_\theta}{\partial \mathbf{z}(t)}\right) \end{bmatrix}}_{\text{dynamics}} dt, \quad \underbrace{\begin{bmatrix} \mathbf{z}(t_1) \\ \log p(\mathbf{x}) - \log p(\mathbf{z}(t_1)) \end{bmatrix}}_{\text{initial values}} = \begin{bmatrix} \mathbf{x} \\ 0 \end{bmatrix}.$$

¹For the full derivation, see e.g. [Chen et al., 2018](#) Appendix A.

Which we can see involves solving the ODE backwards in time from $t_1 = T$ to $t_0 = 0$ and can recover both the latent variable $\mathbf{z}_0$ and the log-density $\log p(\mathbf{x})$ given a training sample $\mathbf{x}$.

Solving this IVP can be done in many ways, including using existing ODE solvers like Runge-Kutta methods or adaptive solvers, and the cost is largely determined by the complexity of the velocity field and the stiffness of the ODE, since more complex velocity fields may require more function evaluations to accurately solve the ODE. An important caveat to this is that the solver must be itself differentiable, since we need to be able to differentiate through it to see how to modify the vector field. This is not the case for many existing ODE solvers you may have used.² Further techniques are used to reduce the computational burden, such through the use of Jacobian-vector products to compute the trace term efficiently, the use of the Hutchinson’s trace estimator, as well as regularization methods to ensure smoothness of the velocity field, which can help in reducing the number of function evaluations needed during ODE solving, but these details are beyond the scope of this book. For further details see the original paper by [Chen et al., 2018](#) and the follow-up work by [Grathwohl et al., 2019](#).

To gain some intuition about continuous flows, we will do a couple of simple experiments using a simple flow integrator using Runge-Kutta 45 – in practice you could use an existing solver or customized code with some of the advanced tricks mentioned above, but this will allow us to get the main concept across with minimal code complexity.

```
Tensor = torch.Tensor

# Fourth-order Runge-Kutta (RK4) integrator step
def rk4_step(field: Callable[[Tensor, float], Tensor], x: Tensor, t: float, dt: float) -> Tensor:
    k1 = field(x, t)
    k2 = field(x + 0.5 * dt * k1, t + 0.5 * dt)
    k3 = field(x + 0.5 * dt * k2, t + 0.5 * dt)
    k4 = field(x + dt * k3, t + dt)
    return x + (dt / 6.0) * (k1 + 2.0 * k2 + 2.0 * k3 + k4)

# Integrate flow using RK4
def integrate_flow(
    field: Callable[[Tensor, float], Tensor],
    x0: Tensor,
    t_span: Tuple[float, float] = (0.0, 1.0),
    steps: int = 100,
    record: bool = False,
):
    t0, t1 = t_span
```

²Interestingly enough, we showed in our [review of automatic differentiation](#) how to make a Verlet Integrator differentiable. In practice, one could use an existing ODE solver that was designed to be differentiable, such as `torchdiffeq` in PyTorch, which we will use in the experiments below.

```

    dt = float(t1 - t0) / steps
    x = x0.clone()
    t = float(t0)
    history = []
    if record:
        history.append(x.detach().cpu().numpy())
    for _ in range(steps):
        x = rk4_step(field, x, t, dt)
        t += dt
        if record:
            history.append(x.detach().cpu().numpy())
    if record:
        times = np.linspace(t0, t1, steps + 1)
        return x, np.stack(history), times
    return x

# Integrate flow and log-probability using RK4
def integrate_flow_with_logp(
    field: Callable[[Tensor, float], Tensor],
    divergence_fn: Callable[[Tensor, float], Tensor],
    x0: Tensor,
    logp0: Tensor,
    t_span: Tuple[float, float] = (0.0, 1.0),
    steps: int = 100,
):
    t0, t1 = t_span
    dt = float(t1 - t0) / steps
    x = x0.clone()
    logp = logp0.clone()
    t = float(t0)
    for _ in range(steps):
        div = divergence_fn(x, t)
        logp = logp - dt * div
        x = rk4_step(field, x, t, dt)
        t += dt
    return x, logp

# Estimate divergence
def estimate_divergence_autograd(field: Callable[[Tensor, float], Tensor], x: Tensor, t: float) -> Tensor:
    x = x.clone().detach().requires_grad_(True)
    v = field(x, t)
    components = []

```

```

for idx in range(v.shape[1]):
    grad = torch.autograd.grad(v[:, idx].sum(), x, retain_graph=idx < v.shape[1] - 1)[0][:, idx]
    components.append(grad)
return torch.stack(components, dim=1).sum(dim=1)

```

15.3 Example: Divergence-Free Rotation

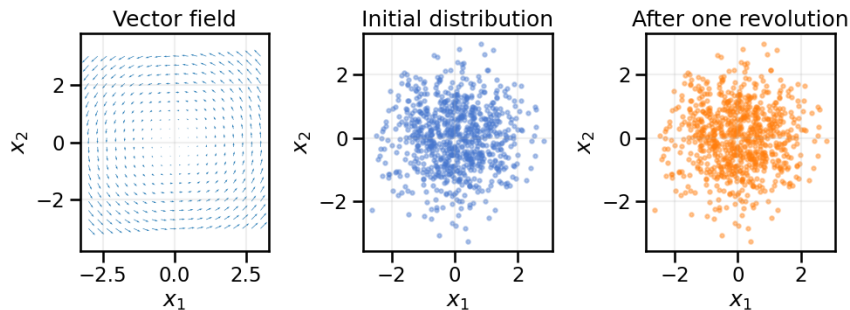
Consider the velocity field $v(x, y) = (-y, x)$. Its divergence is zero, so the flow should rotate particles without changing their density. We integrate a few trajectories and compare a cloud of points before and after one full revolution.

```

def rotation_field(x: Tensor, t: float) -> Tensor:
    return torch.stack([-x[:, 1], x[:, 0]], dim=1)

cloud0 = torch.randn(800, 2, device=device)
cloudT = integrate_flow(rotation_field, cloud0, t_span=(0.0, 2 * np.pi), steps=200)

```



Because $\nabla_x \cdot v = 0$, every particle completes a rotation on a circle of constant radius. The density of the cloud is unchanged, illustrating how divergence controls instantaneous volume change.

15.4 Example: Contracting Divergence Controls Density Growth

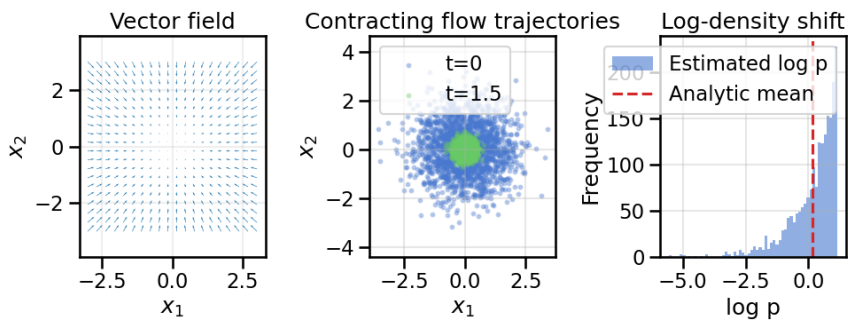
Now take the compressible velocity field $v(x, y) = (-x, -y)$. The divergence is -2 , so the log-density should increase as $\frac{d}{dt} \log p = 2$. We'll integrate particles, track log-density, and compare to the analytic solution $x(t) = e^{-t}x(0)$.

```

def contracting_field(x: Tensor, t: float) -> Tensor:
    return -x

divergence_contracting = -2.0
torch.manual_seed(21)
z0 = torch.randn(2000, 2, device=device)
logp0 = -0.5 * torch.sum(z0 ** 2, dim=1) - np.log(2 * np.pi)
T_final = 1.5
zT, logpT_est = integrate_flow_with_logp(
    contracting_field,
    lambda x, t: torch.full((x.shape[0],), divergence_contracting, device=x.device),
    z0,
    logp0,
    t_span=(0.0, T_final),
    steps=200,
)
scale = torch.exp(torch.tensor(-T_final))
analytic_zT = z0 * scale
analytic_logp = logp0 - divergence_contracting * T_final

```



RMSE between simulated and analytic endpoints: 7.975e-08

Mean absolute log-density error: 4.175e-06

The numerical flow contracts every trajectory toward the origin while the histogram of $\log p$ shifts by approximately $2T$. The errors stay small because the velocity field is linear and the RK4 integrator tracks it well.

15.5 Example 1D Continuous Normalizing Flow with Neural ODEs

Finally, we'll implement a simple continuous normalizing flow using a neural ODE to learn a target distribution. We'll define a neural network velocity field, integrate the ODE, and

optimize the parameters to maximize likelihood on a simple 1D distribution.

Note: this demo requires the `torchdiffeq` package. You can install it via `pip` if you haven't already if you wish to run the demo.

```
from torchdiffeq import odeint

def sample_base_gaussian(n: int) -> Tensor:
    return torch.normal(0.0, 1.0, size=(n, 1), device=device)

def log_prob_base_gaussian(x: Tensor) -> Tensor:
    const = -0.5 * torch.log(2 * torch.pi * torch.ones_like(x))
    return -0.5 * x.pow(2) + const

_UNIF_LOW, _UNIF_HIGH = -3.0, 3.0
def sample_base_uniform(n: int, low: float = _UNIF_LOW, high: float = _UNIF_HIGH) -> Tensor:
    return low + (high - low) * torch.rand(n, 1, device=device)

def log_prob_base_uniform(x: Tensor, low: float = _UNIF_LOW, high: float = _UNIF_HIGH) -> Tensor:
    width = high - low
    logp = -torch.log(torch.tensor(width, device=x.device)) * torch.ones_like(x)
    inside = (x >= low) & (x <= high)
    # out-of-support points get -inf log-probability
    return torch.where(inside, logp, torch.full_like(x, float("-inf")))

# We will use the uniform base distribution, but you can switch it to the Gaussian if desired
# by changing these assignments below.
sample_base = sample_base_uniform
log_prob_base = log_prob_base_uniform

def sample_target(n):
    # mixture of two Gaussians, 0.5 each
    mix = torch.randint(0, 2, (n, 1), device=device).float()
    means = -2 * (1 - mix) + 2 * mix
    return means + 0.4 * torch.randn(n, 1, device=device)

def log_prob_target(x):
    # mixture density for visualization only
    denom = 0.4 * torch.sqrt(2 * torch.pi * torch.ones_like(x))
    p1 = torch.exp(-0.5 * ((x + 2) / 0.4)**2) / denom
    p2 = torch.exp(-0.5 * ((x - 2) / 0.4)**2) / denom
    return torch.log(0.5 * (p1 + p2) + 1e-9)
```

```

# Define the neural network vector field f(x,t)
class CNF1D(nn.Module):
    def __init__(self, hidden=32):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1 + 1, hidden),
            nn.Tanh(),
            nn.Linear(hidden, hidden),
            nn.Tanh(),
            nn.Linear(hidden, 1)
        )

    def forward(self, t, x):
        if isinstance(t, torch.Tensor):
            t = t.to(x.device)
        t_in = torch.ones_like(x) * t
        return self.net(torch.cat([x, t_in], dim=1))

# CNF dynamics: dx/dt = f, d(logp)/dt = -∂f/∂x
class CNFDynamics(nn.Module):
    def __init__(self, fnet):
        super().__init__()
        self.fnet = fnet

    def forward(self, t, states):
        x, logp = states
        x = x.requires_grad_(True)
        f = self.fnet(t, x)
        df_dx = torch.autograd.grad(f, x, torch.ones_like(f), create_graph=True)[0]
        dlogp = -df_dx
        return f, dlogp

fnet = CNF1D(hidden=32).to(device)
dynamics = CNFDynamics(fnet)
optimizer = optim.AdamW(fnet.parameters(), lr=1e-3)

for step in range(600):
    x0 = sample_base(256)
    logp0 = log_prob_base(x0)

    # ensure time tensor is on the same device as the model/state

```

```

t = torch.tensor([0.0, 1.0], device=device)
x1, logp1 = odeint(dynamics, (x0, logp0), t)
x1, logp1 = x1[-1], logp1[-1]

# Negative log-likelihood under target (maximum likelihood)
logpt = log_prob_target(x1)
loss = -(logpt - logp1).mean()

optimizer.zero_grad()
loss.backward()
optimizer.step()

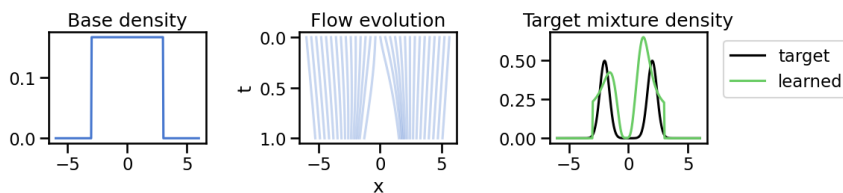
if step % 50 == 0:
    print(f"Step {step}, loss {loss.item():.4f}")
print("Training complete.")

```

```

Step 0, loss 2.3605
Step 50, loss 1.5631
Step 100, loss 1.0494
Step 150, loss 0.4459
Step 200, loss 0.1519
Step 250, loss 0.1067
Step 300, loss 0.1081
Step 350, loss 0.0705
Step 400, loss 0.0784
Step 450, loss 0.1421
Step 500, loss 0.0940
Step 550, loss 0.0485
Training complete.

```



We can see from the center plot how the initial distribution (blue) is smoothly transformed into the target distribution over time via the learned velocity field, although not perfectly in this example given the small network and simulation time we gave it.

💡 Experiment: Effect of Base Distribution and Smoothness on the Flow Field

Changing some of the below will help you see some possible failure modes of continuous normalizing flows:

- Try changing the base distribution from standard normal to a uniform distribution. How does this affect the learned flow? What happens if the base distribution has bounded support that is different from the target distribution? How does the flow handle this?
- Experiment with the smoothness of the velocity field by changing the architecture of the neural network (e.g., number of layers, activation functions, weight decay to promote regularization). How does this impact the quality of the learned distribution and the stability of training?
- Change `sample_base` to use more or fewer samples. How does the number of samples used to estimate the likelihood affect the training dynamics, total time taken, and the final learned distribution?

15.6 Summary and Next Steps

- Continuous normalizing flows are the infinitesimal limit of discrete flows, replacing layerwise Jacobians with a divergence term integrated over time.
- The continuity equation formalizes conservation of probability mass and links particle dynamics to density evolution.
- Divergence-free flows preserve density while compressive fields increase log-density.
- Neural ODEs show that we can learn velocity fields that sculpt data distributions without explicit discrete blocks. Inference now involves solving an ODE rather than evaluating a fixed sequence of transformations.
- However, ODE solving can be computationally intensive, especially in high dimensions or with complex velocity fields, since this will necessitate longer numerical integration times. Moreover, computing the trace of the Jacobian remains a bottleneck.

What if we could get some of the benefits of continuous flows without the computational overhead of ODE solving? In the next chapter, we will explore *score-based generative models*, which learn to model data distributions by estimating something called a *score function*. These models can be seen as a special case of continuous flows, but they avoid the need for expensive ODE integration by leveraging denoising techniques and stochastic sampling methods that will ultimately lead us to diffusion models and flow matching approaches in the coming chapters.

16 From Continuous Flows to Score Matching

In the previous chapter, we explored Continuous Normalizing Flows (CNFs), where we learned a vector field v_t to transform a simple base distribution into a complex data distribution. The training objective involved maximizing the log-likelihood of the data, which required computing the trace of the Jacobian of the vector field:

$$\log p_1(x) = \log p_0(z) - \int_0^1 \text{Tr}(\nabla_x v_t(x(t))) dt$$

While powerful, this approach has a significant computational bottleneck: the trace term $\text{Tr}(\nabla_x v_t)$. For high-dimensional data like images, computing this trace is prohibitively expensive, often requiring clever estimation tricks or restrictive model architectures that can slow down training.

This raises a critical question: **Can we define and train a generative model based on a vector field without ever needing to compute a trace?**

This chapter introduces **Score Matching**, a powerful alternative that achieves exactly this.

16.1 Learning Objectives

By the end of this notebook, you should be able to:

- Explain the computational bottleneck of Continuous Normalizing Flows.
- Define the **score function** and its relationship to the data distribution.
- Describe how the score function can be used for generative sampling.
- Understand the challenge of learning the score function directly.
- Explain how **Denoising Score Matching (DSM)** provides a practical, trace-free objective for learning the score.
- Implement a simple score-based model and use it for generative sampling with Langevin dynamics.

Using device: cpu

16.2 Sampling by “Climbing” the Probability Landscape

Imagine our data distribution $p_{data}(x)$ as a “probability landscape”—a mountain range where peaks correspond to likely data points (e.g., realistic images) and valleys represent unlikely ones (e.g., noise). If we were placed randomly in this landscape, how could we find (aka, generate) a “good” data point? A natural strategy would be to climb uphill. We saw this idea previously in the context of MCMC sampling, for example where Hamiltonian Monte Carlo (HMC) uses gradients to propose moves towards higher probability regions. Likewise, we will use similar ideas in Score Matching to define a generative process. Specifically, the direction of steepest ascent on the log-probability landscape is given by its gradient. This leads us to the definition of the **score function**.

16.2.1 The Score Function

The score function is the gradient of the log-probability density with respect to the data x :

$$s(x) \equiv \nabla_x \log p(x)$$

This function defines a vector field where every vector points “uphill” towards the nearest mode (peak) of the data distribution. If we could learn this vector field, we could invent a new generative process:

1. Start from a random point x_0 .
2. Follow the score vectors to “climb” the landscape.
3. Arrive at a peak, which corresponds to a high-likelihood data sample.

16.3 The First Hurdle: How to Learn the Score?

Let’s imagine that we want to train a neural network $s_\theta(x)$ to approximate the true data score $s_{p_{data}}(x)$. The most straightforward objective is to minimize the L2 distance between our model’s score and the true score, an objective known as the **Fisher Divergence**:

$$J(\theta) = \mathbb{E}_{x \sim p_{data}(x)} [\|s_{p_{data}}(x) - s_\theta(x)\|^2]$$

However, this objective is intractable. To compute it, we need the true data score $s_{p_{data}}(x) = \nabla_x \log p_{data}(x)$. But we don’t know the underlying data distribution $p_{data}(x)$; we only have samples from it. This is the central challenge of score matching.

16.4 Solution 1: Hyvärinen's Score Matching

In 2005, Aapo Hyvärinen used techniques from partial integration to prove that the intractable Fisher Divergence objective is equivalent to a tractable one, under certain regularity conditions. The key result is:

$$J(\theta) \propto \mathbb{E}_{x \sim p_{data}(x)} \left[\text{Tr}(\nabla_x s_\theta(x)) + \frac{1}{2} \|s_\theta(x)\|^2 \right]$$

What his result allowed is for us to eliminate the need for the unknown true score $s_{p_{data}}(x)$, allowing us to compute an objective using only our model s_θ and data samples $x \sim p_{data}(x)$, which we have (this is our training data).

However, look closely at the first term: $\text{Tr}(\nabla_x s_\theta(x))$ ¹. We are right back where we started with continuous flow matching. We've traded one generative model (CNF) for another (Score Matching), but we are still stuck with the same computationally expensive trace of the Jacobian. This is why classic Score Matching wasn't a silver bullet for high-dimensional cases where CNFs struggled.

16.5 Solution 2: Denoising Score Matching (DSM)

The breakthrough came in 2011 from Pascal Vincent, who asked: **What if we try to learn the score of *noisy* data instead of clean data?**

To see what we might mean by this, consider the following setup:

1. **Define a noise kernel:** We perturb our data x with Gaussian noise: $p_\sigma(\tilde{x}|x) = \mathcal{N}(\tilde{x}|x, \sigma^2 I)$.
2. **Create a noisy data distribution:** This process defines a new, noisy data distribution $p_\sigma(\tilde{x}) = \int p_\sigma(\tilde{x}|x) p_{data}(x) dx$.
3. **Set a new goal:** Our goal is now to learn the score of this noisy distribution: $s_\theta(\tilde{x}) \approx \nabla_{\tilde{x}} \log p_\sigma(\tilde{x})$.

It turns out that the objective for this new problem, $\mathbb{E}_{p_\sigma(\tilde{x})} [\|\nabla_{\tilde{x}} \log p_\sigma(\tilde{x}) - s_\theta(\tilde{x})\|^2]$, also simplifies, but to something much better. The optimal score model $s_\theta(\tilde{x})$, which is the thing we are trying to learn, for the Denoising objective is the one that minimizes:

$$J_\sigma(\theta) = \mathbb{E}_{x \sim p_{data}, \tilde{x} \sim p_\sigma(\cdot|x)} [\|s_\theta(\tilde{x}) - \nabla_{\tilde{x}} \log p_\sigma(\tilde{x}|x)\|^2]$$

¹Note that in some texts this is referred to as $\nabla \cdot s_\theta(x)$ (the divergence of the score function) instead of $\text{Tr}(\nabla_x s_\theta(x))$, but these are mathematically equivalent.

This new target, $\nabla_{\tilde{x}} \log p_{\sigma}(\tilde{x}|x)$, is the score of our noise kernel, which we know analytically. Since the log-probability of our Gaussian noise kernel is:

$$\log p_{\sigma}(\tilde{x}|x) = C - \frac{1}{2\sigma^2} \|\tilde{x} - x\|^2$$

its gradient with respect to $\tilde{x}$ is simply:

$$\nabla_{\tilde{x}} \log p_{\sigma}(\tilde{x}|x) = -\frac{\tilde{x} - x}{\sigma^2}$$

We can substitute this back into our original objective:

$$J_{\sigma}(\theta) = \mathbb{E}_{x \sim p_{data}, \tilde{x} \sim p_{\sigma}(\cdot|x)} \left[\left\| s_{\theta}(\tilde{x}) + \frac{\tilde{x} - x}{\sigma^2} \right\|^2 \right]$$

And rename $\tilde{x}$ as $x + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$, to arrive at the final form:

$$J_{\sigma}(\theta) = \mathbb{E}_{x \sim p_{data}, \epsilon \sim \mathcal{N}(0, \sigma^2 I)} \left[\left\| s_{\theta}(x + \epsilon) + \frac{\epsilon}{\sigma^2} \right\|^2 \right]$$

This gives us our final, practical, **trace-free** Denoising Score Matching (DSM) loss:

$$J_{\sigma}(\theta) = \mathbb{E}_{x \sim p_{data}, \epsilon \sim \mathcal{N}(0, \sigma^2 I)} \left[\left\| s_{\theta}(x + \epsilon) + \frac{\epsilon}{\sigma^2} \right\|^2 \right]$$

This is just a simple L2 regression! We train a neural network to: 1. Take a noisy sample $\tilde{x} = x + \epsilon$. 2. Predict the “ground truth” score, which is proportional to the noise that was added: $-\epsilon/\sigma^2$.

There is no trace and no intractable integrals. We have successfully found a way to learn a generative vector field that completely bypasses the CNF bottleneck. To enable this, we had to assume that we were okay with learning the score of a noisy version of the data distribution, which may seem a bit silly but, as we will see in the next chapter, this turns out to be a powerful and flexible approach for generative modeling.

16.6 Example: Learning a Score Model for a 2D Ring of Gaussians

Let's apply this to a 2D problem to build some intuition. We'll generate data from the Ring of Gaussians we have been using in earlier chapters, and then train a neural network to learn its score function s_θ using the DSM objective. This will allow us to visualize the learned score as a vector field.

```
#### Try changing the noise level sigma here and see
# how it affects the learned score function.
# Try values like 0.01, 0.1, 0.5, 5.0
sigma = 0.1 # Noise level
#####3

# Create the dataset
data, _ = create_ring_gaussians(n_samples=5000)
loader = make_loader(data, batch_size=256)

# Define the neural network to model the score function s_theta(x)
class ScoreNet2D(nn.Module):
    def __init__(self, hidden=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(2, hidden),
            nn.Softplus(),
            nn.Linear(hidden, hidden),
            nn.Softplus(),
            nn.Linear(hidden, 2)
        )

    def forward(self, x):
        return self.net(x)

# Training parameters
lr = 1e-3
n_epochs = 400

# Model and optimizer
score_net = ScoreNet2D().to(device)
optimizer = optim.AdamW(score_net.parameters(), lr=lr)

# Training loop
for epoch in range(n_epochs):
```

```

for (x,) in loader:
    x = x.to(device)

    # 1. Add noise
    noise = torch.randn_like(x) * sigma
    x_tilde = x + noise

    # 2. Predict the score
    predicted_score = score_net(x_tilde)

    # 3. Compute the ground truth score (target)
    target_score = -noise / (sigma**2)

    # 4. Calculate the DSM loss
    loss = ((predicted_score - target_score)**2).mean()

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

if epoch % 20 == 0:
    print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

print("Training complete.")

```

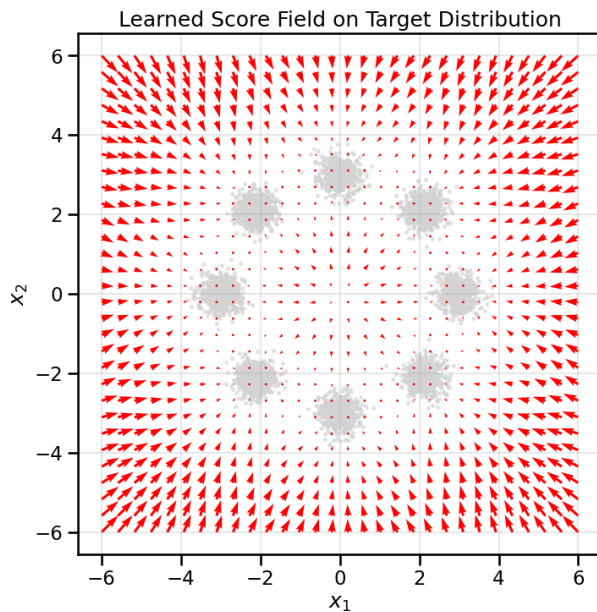
```

Epoch 0, Loss: 99.9621
Epoch 20, Loss: 94.5389
Epoch 40, Loss: 96.6329
Epoch 60, Loss: 94.0892
Epoch 80, Loss: 95.9352
Epoch 100, Loss: 89.9783
Epoch 120, Loss: 91.1169
Epoch 140, Loss: 91.3176
Epoch 160, Loss: 89.9263
Epoch 180, Loss: 81.9100
Epoch 200, Loss: 96.8376
Epoch 220, Loss: 91.9254
Epoch 240, Loss: 82.5780
Epoch 260, Loss: 89.1135
Epoch 280, Loss: 78.5972
Epoch 300, Loss: 89.9539
Epoch 320, Loss: 88.9281

```

Epoch 340, Loss: 90.0029
Epoch 360, Loss: 81.8739
Epoch 380, Loss: 78.4657
Training complete.

Now let's visualize the learned score function as a vector field. We will overlay the score vectors on a contour plot of the target data distribution. This will show us the direction that the learned score function will push particles in.



Training for animation...
epoch 1 | loss 103.260
epoch 50 | loss 97.533
epoch 100 | loss 98.608
epoch 150 | loss 88.190
epoch 200 | loss 88.734
epoch 250 | loss 83.404
epoch 300 | loss 80.596
epoch 350 | loss 78.519
epoch 400 | loss 90.604

The plot shows that our network has successfully learned the score function. The learned score vectors point away from the low-density region and towards the peaks of each Gaussian, just as expected.

💡 Experiment: Effect of Noise Level on Denoising Score Matching

We saw above that the key trick in Denoising Score Matching is to add noise to the data before learning the score function. The amount of noise added is controlled by the parameter σ . But what effect does the choice of σ have on the learned score function and the quality of generated samples?

- Try changing the noise level σ to different values (e.g., 0.01, 0.1, 0.5, 5.0). How does this affect the learned score function?
- Reflect on how Denoising Score Matching achieves its approximation compared to the original score matching objective. What effect does the noise level have on this approximation? How is this reflected in the learned score function as you vary σ ?

16.7 How do we use the learned Score Function?

Now that we have a trained model $s_\theta(x)$ that can estimate the score of a data distribution, we can finally implement our “sampling by climbing” idea.

A naive approach would be to start from random noise and iteratively update the sample by taking small steps in the direction of the score:

$$x_{k+1} \leftarrow x_k + \alpha s_\theta(x_k)$$

This is just gradient ascent on the log-probability landscape defined by our score model. However, this approach has a critical flaw: it will quickly get stuck in the nearest local maximum (peak) of the distribution and fail to explore the full data distribution. So while this is OK for optimization or finding modes of the distribution, it is not sufficient for generative sampling.

Instead, we need to introduce some randomness into the process to allow exploration of the landscape. This is done using **Langevin Dynamics** which involves taking a small step in the direction of the score (the gradient ascent “climb”) and then adding a small amount of noise. This noise prevents the sampler from just finding the nearest peak and stopping; instead, it allows it to explore the full distribution.

The update rule for Langevin Dynamics is:

$$x_{k+1} \leftarrow x_k + \alpha s_\theta(x_k) + \sqrt{2\alpha} z_k$$

where α is the step size and $z_k \sim \mathcal{N}(0, I)$ is standard Gaussian noise. The $s_\theta(x_k)$ term performs the climb, while the z_k term makes the process stochastic, turning it into a sampler (like MCMC) rather than just an optimizer.

The choice of step size α is important: too large and the sampler will be unstable, too small and it will mix slowly. In practice, α is often decreased over time (as we will see later) to allow for finer exploration as the sampler approaches high-density regions.

It perhaps seems magical that this simple procedure will converge to samples from the data distribution. The theoretical justification comes from the fact that Langevin Dynamics is a discretization of a continuous-time stochastic differential equation (SDE) whose stationary distribution is exactly the target distribution defined by the score function. Thus, as the number of steps goes to infinity and the step size goes to zero, the samples generated by Langevin Dynamics converge to samples from the target distribution.

Specifically, the SDE is given by:

$$dx_t = s_\theta(x_t)dt + \sqrt{2}dW_t$$

where W_t is standard Brownian motion. We will see this connection to SDEs in more detail in the next chapter when we see its connection to Diffusion Models.

```
# Compare deterministic gradient-ascent (no noise) vs Langevin dynamics (with noise)

def deterministic_ascent(score_model, init_points, steps=200, alpha=0.05):
    """Move points by x <- x + alpha * score(x) without noise and record trajectories."""
    x = init_points.clone().to(device)
    traj = [x.cpu().numpy()]
    score_model.eval()
    with torch.no_grad():
        for _ in range(steps):
            s = score_model(x)
            x = x + alpha * s
            traj.append(x.cpu().numpy())
    return np.array(traj)

# Langevin Dynamics Sampler for 2D
def langevin_sampler_2d(score_model, n_samples=500, n_steps=200, alpha=0.01, initial_noise_scale=5.0, track_trajectories=True):
    """Generate samples using Langevin Dynamics, with optional trajectory tracking."""
    if track_trajectories and tracked_points is not None:
        x = tracked_points.to(device)
        n_samples = tracked_points.shape[0]
    else:
        x = torch.randn(n_samples, 2, device=device) * initial_noise_scale

    trajectory = [x.cpu().detach().numpy()]

    score_model.eval()
```

```

with torch.no_grad():
    for _ in range(n_steps):
        score = score_model(x)
        noise = torch.randn_like(x) * np.sqrt(2 * alpha)
        x = x + alpha * score + noise
        trajectory.append(x.cpu().detach().numpy())

    return x, np.array(trajectory)

# Choose three representative initial points (same as earlier examples)
init_points = torch.tensor([[ -1.0, -1.0], [ 0.0, 1.5], [ 1.0, -0.5]], dtype=torch.float32)

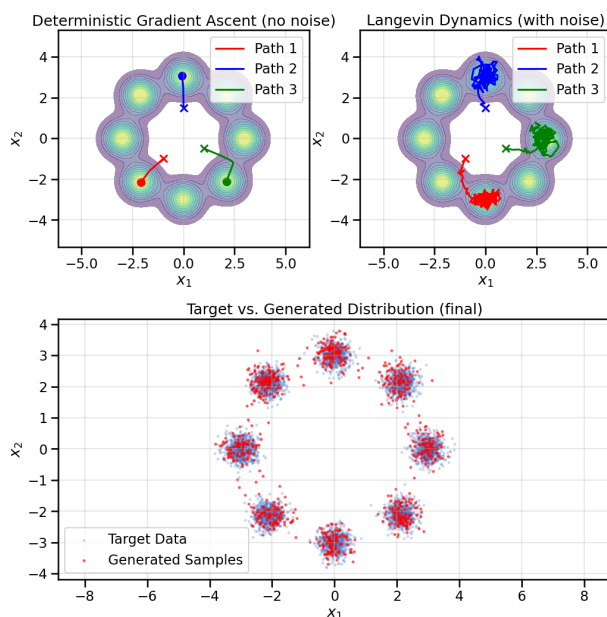
# Deterministic ascent trajectories (no noise)
det_traj = deterministic_ascent(score_net, init_points, steps=200, alpha=0.05)

# Langevin trajectories (with noise) using the sampler we defined earlier
_, lang_traj = langevin_sampler_2d(score_net, track_trajectories=True,
                                   tracked_points=init_points, n_steps=200, alpha=0.01)

# Generate a larger set of samples for density comparison
generated_samples, _ = langevin_sampler_2d(score_net, n_samples=1000, alpha=0.01)

```

C:\Users\mafuge\AppData\Local\Temp\ipykernel_14996\2162259831.py:46: UserWarning: This figure includes A
plt.tight_layout()



16.8 The Final Bridge to Diffusion Models

This works, but we have one big new problem: the distribution depends on noise level, which remains constant during both training and sampling. If we set σ too high, the learned score will be too smooth and miss important details of the data distribution. If we set σ too low, the score will be too noisy and hard to learn accurately.

What if we allowed ourselves to change the noise level as a function of time during sampling? Perhaps we could start with a high noise level (large σ) to capture the broad structure of the data, and then gradually reduce the noise (small σ) to refine the samples such that they begin to resemble the true data distribution.

This is **exactly what a Diffusion Model is**: a time-conditional score model $s_\theta(x_t, t)$ that is trained via Denoising Score Matching at all noise levels t simultaneously. This allows for a much more powerful and stable generative process, which we will explore in the next chapter.

16.9 Summary and Next Steps

- Score Matching provides a way to train a generative vector field without computing the expensive trace of the Jacobian required by CNFs.
- The core idea is to learn the **score function** $(\nabla_x \log p(x))$, which points towards regions of high data density.
- **Denoising Score Matching (DSM)** makes this practical by reframing the problem as a simple L2 regression: predicting the noise added to a data sample.
- Once the score function is learned, **Langevin Dynamics** can be used to generate new data by iteratively following the score field and adding noise.
- This framework directly leads to **Diffusion Models**, which use a time-conditional score network to reverse a gradual noising process.

17 From Score Matching to Diffusion Models

In the previous chapter, we introduced Denoising Score Matching (DSM) as a way to learn a generative vector field, the **score function** $\nabla_x \log p(x)$, without the computationally expensive trace term that plagued Continuous Normalizing Flows. We used this learned score to generate samples via Langevin Dynamics, which follows the score field uphill while adding noise to explore the data distribution.

However, this approach has its own limitations. The score function is only well-defined and easy to learn *near* the data. Far from the data manifold, where there are no samples, the score can be chaotic or undefined, making it difficult to initialize a generative process from random noise. Furthermore, we had to choose a single, fixed noise level σ for training, which created a trade-off: too much noise and we lose important details; too little and the score is hard to learn.

This chapter introduces **Diffusion Models**, which solve these problems by learning a score function not just for one noise level, but for a *continuum* of noise levels. This creates a robust, time-dependent vector field that can smoothly transform pure noise into structured data.

17.1 Learning Objectives

By the end of this notebook, you should have a strong intuition for:

- Why diffusion models can be seen as “time-extended” score models.
- How the **forward process** gradually adds noise to data, creating a series of tractable distributions.
- How the **reverse process** uses a time-conditional score model to denoise a sample back onto the data manifold.
- How noise levels (t) structure the learning problem and stabilize training.
- How to implement and train a simple diffusion model.

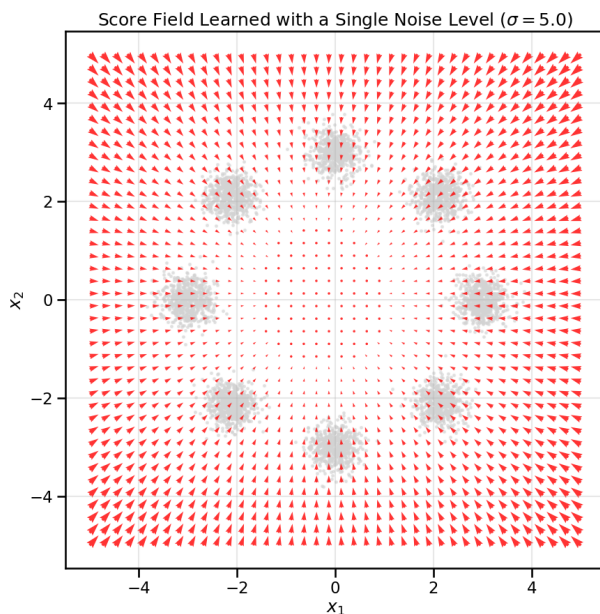
Using device: cpu

17.2 Why Go Beyond Single-Noise Score Matching?

Let's first revisit the limitations of the score model we trained in the previous chapter. That model learned the score function for a *single* noise level, σ . This meant trading off detail and stability: a small σ captures fine details but is hard to learn, while a large σ smooths out the data but loses the ability to describe higher-frequency contours of the true log probability.

The key idea behind diffusion models is that they fix this problem by defining a continuum of noisy data distributions $p_t(x)$, which are smooth and well-behaved everywhere, even far from the data, yet can still create score functions that capture fine details when t is small.

Finished training score network with sigma=5.0



17.3 The Forward Diffusion Process $q(x_t|x_0)$

The first key component of a diffusion model is the **forward process**, where we systematically destroy the structure in our data by gradually adding Gaussian noise over a series of timesteps. We define a “noise schedule” that controls how much noise is added at each step.

Let x_0 be a sample from our data distribution. The forward process $q(x_t|x_0)$ defines a noisy version of this sample at time t :

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Here, $\bar{\alpha}_t$ is a parameter that goes from 1 down to (almost) 0 as t goes from 0 to T . - When $t = 0$, $\bar{\alpha}_0 = 1$, so x_0 is just the original data. - As t increases, $\bar{\alpha}_t$ decreases, and more noise is added. - When $t = T$ is large, $\bar{\alpha}_T \approx 0$, so x_T is almost pure Gaussian noise.

This process creates a family of intermediate, noisy distributions $p_t(x)$. Let's visualize this by taking our ring of Gaussians and applying the forward process for different values of t .

```
def get_alphas(timesteps=1000, beta_start=0.0001, beta_end=0.02):
    """Create the noise schedule parameters alpha and alpha_bar."""
    betas = torch.linspace(beta_start, beta_end, timesteps)
    alphas = 1. - betas
    alphas_cumprod = torch.cumprod(alphas, axis=0)
    return alphas, alphas_cumprod

def forward_diffusion_sample(x_0, t, alphas_cumprod, device="cpu"):
    """Takes an datapoint and a timestep and returns the noisy version."""
    noise = torch.randn_like(x_0)
    sqrt_alphas_cumprod_t = torch.sqrt(alphas_cumprod[t])[:, None]
    sqrt_one_minus_alphas_cumprod_t = torch.sqrt(1. - alphas_cumprod[t])[:, None]

    # send to device
    sqrt_alphas_cumprod_t = sqrt_alphas_cumprod_t.to(device)
    sqrt_one_minus_alphas_cumprod_t = sqrt_one_minus_alphas_cumprod_t.to(device)
    noise = noise.to(device)

    # Equation for x_t
    noisy_x = sqrt_alphas_cumprod_t * x_0 + sqrt_one_minus_alphas_cumprod_t * noise
    return noisy_x, noise

# Get the noise schedule
T = 200
alphas, alphas_cumprod = get_alphas(timesteps=T)

# Visualize the forward process
data_tensor = torch.tensor(data, dtype=torch.float32).to(device)

fig, axes = plt.subplots(5, 1, figsize=(6, 20))
plot_times = [0, T//4, T//2, 3*T//4, T-1]

for i, t_int in enumerate(plot_times):
    t = torch.full((data_tensor.shape[0],), t_int, dtype=torch.long)
    noisy_data, _ = forward_diffusion_sample(data_tensor, t, alphas_cumprod, device=device)
    noisy_data = noisy_data.cpu().numpy()
```

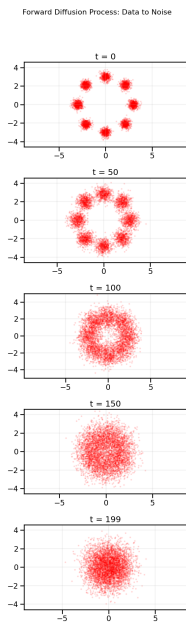
```

ax = axes[i]
ax.scatter(noisy_data[:, 0], noisy_data[:, 1], s=5, alpha=0.1, color="red")
ax.set_title(f"t = {t_int}")
ax.axis('equal')
#ax.set_xlim(-7, 7)
ax.set_ylim(-5, 5)
ax.grid(True, alpha=0.2)

fig.suptitle("Forward Diffusion Process: Data to Noise", fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

```

Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.



As you can see, the forward process gradually transforms the structured ring of Gaussians into a fuzzy, isotropic Gaussian blob. At each intermediate step t , we have a distribution $p_t(x)$ that is smoother and more spread out than the original data distribution. This gives us a family

of intermediate densities where score estimation is much easier and better-behaved, which is the key insight we will leverage next.

17.4 The Reverse Process and Multi-Scale Score Estimation

The magic of diffusion models lies in the **reverse process**. If we can reverse the arrow of time, we can start from pure noise $x_T \sim \mathcal{N}(0, I)$ and gradually denoise it step-by-step to generate a clean sample x_0 .

It turns out that the reverse of the diffusion process is also a diffusion, described by a stochastic differential equation (SDE). While the full derivation is beyond our scope, the key insight is that the “drift” of this reverse process depends on the score of the noisy data distribution at that time, $\nabla_{x_t} \log p_t(x_t)$.

$$dx_t = f(x_t, t)dt + g(t)^2 \nabla_{x_t} \log p_t(x_t)dt + g(t)d\bar{W}_t$$

where $f(x_t, t)$ is the drift term of the forward process, $g(t)$ is the diffusion coefficient, and $d\bar{W}_t$ is a Wiener process in reverse time. For the time being, let's put aside the exact forms of f and g and focus on the score term.

This is the crucial connection: **to reverse the diffusion, we need to know the score function at every noise level t** . This is why diffusion models are essentially **time-conditional score models**. Instead of learning a single score function for a fixed σ , we will learn a single neural network $s_\theta(x_t, t)$ that can estimate the score for any noisy sample x_t at any time t .

Let's visualize what these score fields look like at different noise levels. We'll train separate score networks for low, medium, and high noise to see how the target vector field changes.

- **Low Noise (t is small):** The score field should be sharp and detailed, pointing strongly towards the modes of the original data.
- **Medium Noise (t is medium):** The score field will be smoother, capturing the global structure of the data (the ring) rather than individual modes.
- **High Noise (t is large):** The data is almost pure noise, so the score field will be largely diffuse, with weak gradients pointing towards the center of the distribution.

```
# Map t to sigma. The variance of the noise added at time t is 1 - alpha_bar_t.
# So, sigma_t = sqrt(1 - alpha_bar_t)
T = 200
_, alphas_cumprod = get_alphas(timesteps=T)

# Times and corresponding sigmas to visualize
viz_times = {'Low Noise (t=10)': 2, 'Medium Noise (t=80)': 80, 'High Noise (t=199)': 199}
```

```

score_nets = {}
data_vis = None

for name, t_int in viz_times.items():
    sigma_t = torch.sqrt(1. - alphas_cumprod[t_int]).item()
    print(f"Training for {name}, sigma={sigma_t:.3f}")
    net, data_vis = train_simple_score_net(sigma=sigma_t)
    score_nets[name] = net

```

```

Training for Low Noise (t=10), sigma=0.024
Finished training score network with sigma=0.024492062628269196
Training for Medium Noise (t=80), sigma=0.532
Finished training score network with sigma=0.024492062628269196
Training for Medium Noise (t=80), sigma=0.532
Finished training score network with sigma=0.5321973562240601
Training for High Noise (t=199), sigma=0.932
Finished training score network with sigma=0.5321973562240601
Training for High Noise (t=199), sigma=0.932
Finished training score network with sigma=0.9315670728683472
Finished training score network with sigma=0.9315670728683472

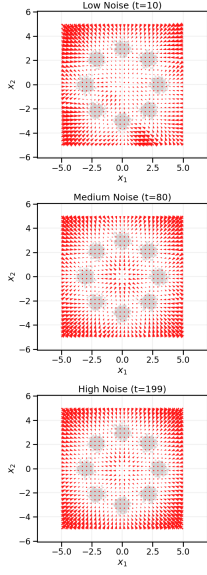
```

```

Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.

```

Score Fields at Different Noise Levels (Time t)



17.5 How Training Works (Denoising Score Matching Perspective)

Now that we understand the goal—learning a time-conditional score function—how do we train such a model? The answer is a natural extension of the Denoising Score Matching (DSM) objective we learned in the previous chapter.

Instead of training on a single noise level, we train on **all noise levels simultaneously**. We'll create a single neural network, $s_\theta(x_t, t)$, that takes both the noisy data x_t and the timestep t as input. The network's job is to predict the score of the noisy data distribution $p_t(x)$ at that specific time.

The loss function is almost identical to the one for simple DSM from last chapter, but now we take the expectation over uniformly sampled time t as well:

$$L(\theta) = \mathbb{E}_{x_0 \sim p_{data}, t \sim \text{Uniform}(0, T), \epsilon \sim \mathcal{N}(0, I)} \left[\left\| s_\theta(x_t, t) - \nabla_{x_t} \log p_t(x_t | x_0) \right\|^2 \right]$$

Just like before, the target score $\nabla_{x_t} \log p_t(x_t | x_0)$ is analytically known and simplifies to $-\frac{\epsilon}{\sqrt{1-\bar{\alpha}_t}}$. So, our practical loss becomes:

$$L(\theta) = \mathbb{E}_{x_0, t, \epsilon} \left[\left\| s_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t) + \frac{\epsilon}{\sqrt{1-\bar{\alpha}_t}} \right\|^2 \right]$$

In many modern diffusion model implementations, it's more common to reparameterize the network to predict the noise ϵ directly, rather than the score. Let's call this network $\epsilon_\theta(x_t, t)$. The relationship is: $s_\theta(x_t, t) = -\frac{\epsilon_\theta(x_t, t)}{\sqrt{1-\alpha_t}}$. Substituting this into the loss function gives a simpler, more stable objective:

$$L(\theta) = \mathbb{E}_{x_0, t, \epsilon} \left[\left\| \epsilon_\theta(\sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon, t) - \epsilon \right\|^2 \right]$$

This is (again) just a simple L2 regression! The network takes a noisy sample x_t and its corresponding time t and learns to predict the exact noise ϵ that was added to the original sample x_0 .

17.5.1 The Training Pipeline

The training process looks like this for each step:

1. **Sample Data:** Pick a clean sample x_0 from the training data (our ring of Gaussians).
2. **Sample Time:** Pick a random timestep t uniformly from $\{0, 1, \dots, T-1\}$.
3. **Add Noise:** Create the noisy sample x_t using the forward process equation: $x_t = \sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon$.
4. **Predict Noise:** Feed x_t and t into the neural network $\epsilon_\theta(x_t, t)$ to get the predicted noise.
5. **Compute Loss:** Calculate the Mean Squared Error between the network's prediction and the true noise ϵ .
6. **Update:** Backpropagate the loss and update the network's weights.

By repeating this process, the single network ϵ_θ learns the entire time-varying vector field.

```
class DiffusionModel(nn.Module):
    def __init__(self, n_steps=200, time_emb_dim=16, hidden_dim=128):
        super().__init__()
        self.time_embed = nn.Embedding(n_steps, time_emb_dim)
        self.net = nn.Sequential(
            nn.Linear(2 + time_emb_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 2) # Predicts the noise
        )

    def forward(self, x, t):
        t_emb = self.time_embed(t)
```

```

        x_in = torch.cat([x, t_emb], dim=-1)
        return self.net(x_in)

# Training parameters
T = 200 # Number of diffusion steps
lr = 1e-3
n_epochs = 1000
batch_size = 256

# Data
#data, _ = create_ring_gaussians(n_samples=10000)
loader = make_loader(data, batch_size=batch_size)

# Model and optimizer
model = DiffusionModel(n_steps=T).to(device)
optimizer = optim.Adam(model.parameters(), lr=lr)
alphas, alphas_cumprod = get_alphas(timesteps=T)

# Training loop
losses = []
for epoch in range(n_epochs):
    for (x_0,) in loader:
        x_0 = x_0.to(device)

        # 1. Sample a random time t
        t = torch.randint(0, T, (x_0.shape[0],), device=device).long()

        # 2. Create noisy sample x_t and get the true noise
        x_t, noise = forward_diffusion_sample(x_0, t, alphas_cumprod, device)

        # 3. Predict the noise
        predicted_noise = model(x_t, t)

        # 4. Calculate the loss
        loss = nn.functional.mse_loss(predicted_noise, noise)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if epoch % 50 == 0:
        losses.append(loss.item())

```



```

        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

print("Training complete.")

# Plot the loss curve
plt.figure(figsize=(8, 4))
plt.plot(range(0, n_epochs, 50), losses)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.yscale('log')
plt.grid(True)
plt.show()

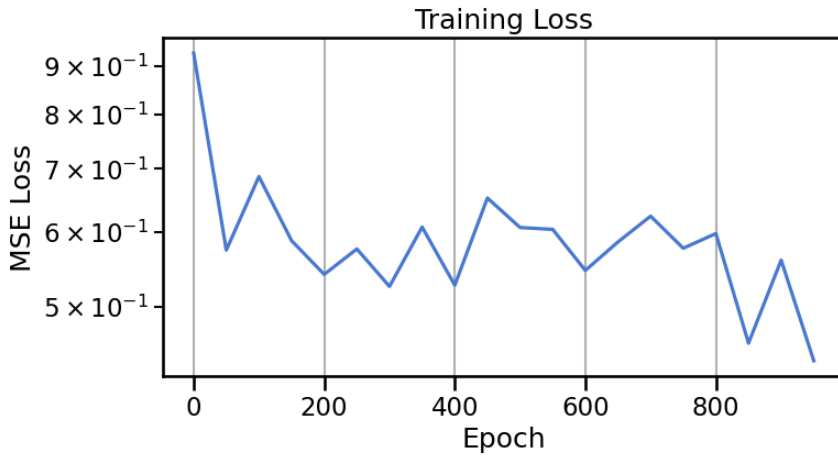
```

```

Epoch 0, Loss: 0.9294
Epoch 50, Loss: 0.5738
Epoch 50, Loss: 0.5738
Epoch 100, Loss: 0.6870
Epoch 100, Loss: 0.6870
Epoch 150, Loss: 0.5872
Epoch 150, Loss: 0.5872
Epoch 200, Loss: 0.5406
Epoch 200, Loss: 0.5406
Epoch 250, Loss: 0.5754
Epoch 250, Loss: 0.5754
Epoch 300, Loss: 0.5250
Epoch 300, Loss: 0.5250
Epoch 350, Loss: 0.6071
Epoch 350, Loss: 0.6071
Epoch 400, Loss: 0.5266
Epoch 400, Loss: 0.5266
Epoch 450, Loss: 0.6519
Epoch 450, Loss: 0.6519
Epoch 500, Loss: 0.6064
Epoch 500, Loss: 0.6064
Epoch 550, Loss: 0.6037
Epoch 550, Loss: 0.6037
Epoch 600, Loss: 0.5459
Epoch 600, Loss: 0.5459
Epoch 650, Loss: 0.5853
Epoch 650, Loss: 0.5853
Epoch 700, Loss: 0.6235

```

Epoch 700, Loss: 0.6235
 Epoch 750, Loss: 0.5764
 Epoch 750, Loss: 0.5764
 Epoch 800, Loss: 0.5976
 Epoch 800, Loss: 0.5976
 Epoch 850, Loss: 0.4568
 Epoch 850, Loss: 0.4568
 Epoch 900, Loss: 0.5599
 Epoch 900, Loss: 0.5599
 Epoch 950, Loss: 0.4376
 Epoch 950, Loss: 0.4376
 Training complete.
 Training complete.



17.6 Generating Samples via Reverse Sampling

With our time-conditional noise predictor $\epsilon_{\theta}(x_t, t)$ fully trained, we can now perform the reverse process to generate new data. This is the most intuitive and visually satisfying part of diffusion models.

We start with pure Gaussian noise, $x_T \sim \mathcal{N}(0, I)$, and then iteratively “denoise” it by stepping backward in time from $t = T-1$ down to $t = 0$. Each reverse step updates the sample’s position based on the predicted noise (which is related to the score as we saw above).

The update rule for a single reverse step (from t to $t-1$) is:

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(x_t - \frac{1 - \alpha_t}{\sqrt{1 - \bar{\alpha}_t}} \epsilon_{\theta}(x_t, t) \right) + \sigma_t z$$

where $z \sim \mathcal{N}(0, I)$ is new noise added at each step (for a stochastic sampler), and σ_t is its standard deviation. This equation looks complex, but the intuition is simple:

1. **Predict Noise:** Use the network $\epsilon_\theta(x_t, t)$ to estimate the noise that was added to get to the current state x_t .
2. **Estimate Clean Data:** Use the predicted noise to get a rough estimate of the original clean data, x_0 .
3. **Step Back:** Use this estimate to take one step backward in the diffusion process, arriving at a slightly less noisy sample, x_{t-1} .
4. **Add Stochasticity:** Add a small amount of random noise to explore the distribution (this corresponds to the SDE sampler). If we omit this term, we get a deterministic sampler (an ODE).

Let's implement this reverse sampling process and watch as a cloud of random noise transforms into our target ring of Gaussians. This animation is the core “aha” moment for understanding how diffusion models work.

```
@torch.no_grad()
def reverse_sample(model, n_samples=50, n_steps=20, initial_noise_scale=1.0):
    """Generate samples by running the reverse diffusion process."""
    model.eval()

    # Start with pure noise
    x = torch.randn(n_samples, 2, device=device) * initial_noise_scale

    # Store trajectory for animation
    trajectory = [x.cpu().numpy()]

    # Get noise schedule
    alphas, alphas_cumprod = get_alphas(timesteps=n_steps)
    betas = 1. - alphas

    for i in range(n_steps - 1, -1, -1):
        t = torch.full((n_samples,), i, device=device, dtype=torch.long)

        # Predict noise
        predicted_noise = model(x, t)

        # Get parameters for this timestep
        alpha_t = alphas[t][:, None]
        alpha_t_cumprod = alphas_cumprod[t][:, None]
        beta_t = betas[t][:, None]
```

```

# The update rule
coeff = (1. - alpha_t) / torch.sqrt(1. - alpha_t_cumprod)
x = (1. / torch.sqrt(alpha_t)) * (x - coeff * predicted_noise)

if i > 0:
    # Add noise for stochastic sampling
    variance = beta_t
    noise = torch.randn_like(x) * torch.sqrt(variance)
    x = x + noise

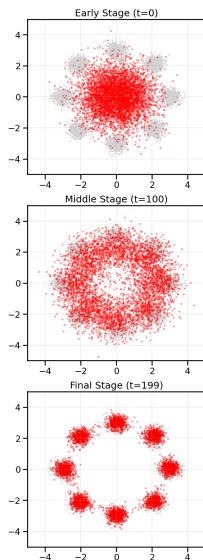
trajectory.append(x.cpu().numpy())

return x.cpu().numpy(), np.array(trajectory)

# Generate samples and their trajectories
num_generated_samples = 5000
generated_samples, trajectory = reverse_sample(model,
                                              n_samples=num_generated_samples,
                                              n_steps=T)

```

Reverse Diffusion Process: Noise to Data



We can also animate the entire reverse diffusion process to visualize how samples evolve from pure noise to structured data:

The animation illustrates the generative process: the points start as a random cloud and are gradually “guided” by the learned, time-varying score field. As time decreases, the field

becomes sharper, pulling the points onto the intricate structure of original data distribution.

17.7 Connection to Langevin Dynamics

How does this relate to the Langevin Dynamics we used in the previous chapter?

You can think of the reverse diffusion process as a more sophisticated, “annealed” version of Langevin Dynamics.

- **Langevin Dynamics** (or at least how we used it in the last chapter) operates on a single, fixed energy landscape (defined by $\log p(x)$). It explores this landscape locally and is dependent on a single noise level applied to the score function.
- **Reverse Diffusion** operates on a changing landscape. It starts on a very smooth, simple landscape (at high t) where the global structure is easy to find. As t decreases, the landscape becomes more detailed, allowing the sampler to refine the samples it generates.

At $t \approx 0$, the reverse SDE step is mathematically very similar to a Langevin dynamics step on the true data score. So, diffusion models can be seen as a way to reliably initialize a sampler in a high-density region before switching to local exploration. This is why they are so effective at generating high-quality, diverse samples.

17.8 Conditional Generation Examples

Diffusion models naturally extend to conditional generation by providing additional conditioning information to the denoising network. This is particularly powerful because the time-conditional architecture already accepts auxiliary inputs, making it straightforward to add class labels, continuous attributes, or even images as conditioning signals.

The key modification is to pass both the noisy data x_t , the timestep t , and the conditioning variable y to the denoising network: $\epsilon_\theta(x_t, t, y)$. During training, we learn to predict noise conditioned on y , and during sampling, we can specify y to control what we generate.

This approach has been used successfully in many applications, including class-conditional image generation (e.g., ImageNet), text-to-image synthesis (e.g., DALL-E 2, Stable Diffusion), and more.

17.8.1 Conditional Example A — One-Hot Mode Conditioning

We'll condition the diffusion model on the discrete mode index using one-hot encoding. The conditioning vector is concatenated with the noisy data and time embedding before passing through the network.

```
from gen_models_utilities import make_conditional_loader_onehot, plot_conditional_samples_discrete

# Conditional Diffusion Model
class ConditionalDiffusionModel(nn.Module):
    def __init__(self, n_steps=200, time_emb_dim=16, label_dim=8, hidden_dim=128):
        super().__init__()
        self.time_embed = nn.Embedding(n_steps, time_emb_dim)
        self.net = nn.Sequential(
            nn.Linear(2 + time_emb_dim + label_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 2)
        )

    def forward(self, x, t, y):
        t_emb = self.time_embed(t)
        x_in = torch.cat([x, t_emb, y], dim=-1)
        return self.net(x_in)

def train_conditional_diffusion_onehot(
    data: np.ndarray, labels: np.ndarray, n_classes: int = 8,
    T: int = 200, lr: float = 1e-3, n_epochs: int = 5000, batch_size: int = 256, print_every: int = 500
):
    loader = make_conditional_loader_onehot(data, labels, batch_size, n_classes)
    model = ConditionalDiffusionModel(n_steps=T, label_dim=n_classes).to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    alphas, alphas_cumprod = get_alphas(timesteps=T)
    losses = []

    for epoch in range(n_epochs):
        for xb, yb in loader:
            xb, yb = xb.to(device), yb.to(device)
            t = torch.randint(0, T, (xb.shape[0],), device=device).long()
            x_t, noise = forward_diffusion_sample(xb, t, alphas_cumprod, device)
            predicted_noise = model(x_t, t, yb)
            loss = nn.functional.mse_loss(predicted_noise, noise)
```

```

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if epoch % print_every == 0:
        losses.append(loss.item())
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

    return model, alphas, alphas_cumprod, losses

# Conditional sampling function
@torch.no_grad()
def sample_conditional_diffusion(
    model: nn.Module, y: torch.Tensor, alphas: torch.Tensor, alphas_cumprod: torch.Tensor,
    n_samples: int = 100, T: int = 200
):
    model.eval()
    x = torch.randn(n_samples, 2, device=device)

    for t_idx in reversed(range(T)):
        t = torch.full((n_samples,), t_idx, device=device, dtype=torch.long)
        predicted_noise = model(x, t, y)

        alpha_t = alphas[t_idx]
        alpha_bar_t = alphas_cumprod[t_idx]

        # Compute mean of the reverse step
        if t_idx > 0:
            noise = torch.randn_like(x)
            alpha_bar_t_prev = alphas_cumprod[t_idx - 1]
            beta_t = 1 - alpha_t
            sigma_t = torch.sqrt(beta_t * (1 - alpha_bar_t_prev) / (1 - alpha_bar_t))
        else:
            noise = torch.zeros_like(x)
            sigma_t = 0.0

    x = (1 / torch.sqrt(alpha_t)) * (x - (1 - alpha_t) / torch.sqrt(1 - alpha_bar_t) * predicted_noise)
    x = x + sigma_t * noise

    return x.cpu().numpy()

```

```

# Train conditional diffusion
print("Training conditional diffusion model (one-hot)...")
# Generate ring of Gaussians data with labels
data, y_ring = create_ring_gaussians(n_samples=5000, n_modes=8, radius=3.0, cluster_std=0.25)

# Train conditional diffusion
print("Training conditional diffusion model (one-hot)...")
cdiff_onehot, alphas_oh, alphas_cumprod_oh, losses_oh = train_conditional_diffusion_onehot(
    data, y_ring, n_classes=8, T=200, lr=1e-3, n_epochs=1000, batch_size=256, print_every=500
)

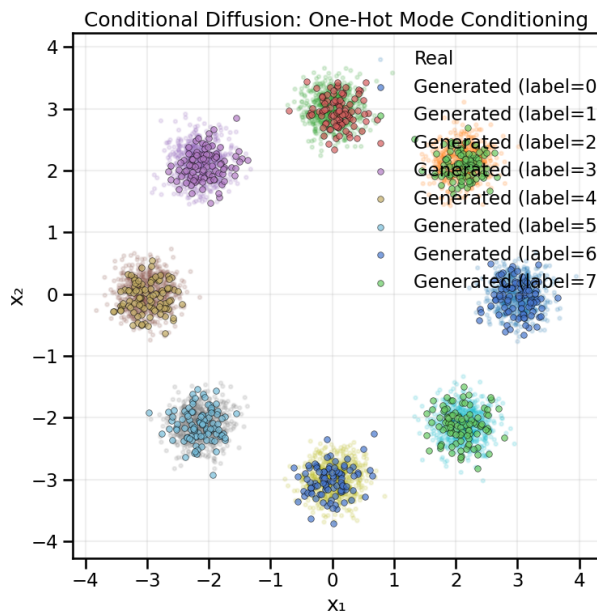
```

```

Training conditional diffusion model (one-hot)...
Training conditional diffusion model (one-hot)...
Epoch 0, Loss: 0.8252
Epoch 500, Loss: 0.2527
Epoch 1000, Loss: 0.2527

```

Let's generate conditional samples for each mode:



17.8.2 Conditional Example B — Continuous Radius Conditioning

Now let's condition the diffusion model on a continuous scalar—the radius. We'll embed the radius through a small MLP before concatenating it with the noisy data and time embedding.


```

from gen_models_utilities import make_conditional_loader_radius, plot_conditional_samples_continuous

# Conditional Diffusion Model with radius embedding
class ConditionalDiffusionModelRadius(nn.Module):
    def __init__(self, n_steps=200, time_emb_dim=16, cond_emb_dim=16, hidden_dim=128):
        super().__init__()
        self.time_embed = nn.Embedding(n_steps, time_emb_dim)
        self.cond_embed = nn.Sequential(
            nn.Linear(1, cond_emb_dim),
            nn.ReLU(),
            nn.Linear(cond_emb_dim, cond_emb_dim),
        )
        self.net = nn.Sequential(
            nn.Linear(2 + time_emb_dim + cond_emb_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 2)
        )

    def forward(self, x, t, r):
        t_emb = self.time_embed(t)
        r_emb = self.cond_embed(r)
        x_in = torch.cat([x, t_emb, r_emb], dim=-1)
        return self.net(x_in)

def train_conditional_diffusion_radius(
    data: np.ndarray, T: int = 200, lr: float = 1e-3,
    n_epochs: int = 5000, batch_size: int = 256, print_every: int = 500
):
    loader = make_conditional_loader_radius(data, batch_size)
    model = ConditionalDiffusionModelRadius(n_steps=T).to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    alphas, alphas_cumprod = get_alphas(timesteps=T)
    losses = []

    for epoch in range(n_epochs):
        for xb, rb in loader:
            xb, rb = xb.to(device), rb.to(device)
            t = torch.randint(0, T, (xb.shape[0],), device=device).long()
            x_t, noise = forward_diffusion_sample(xb, t, alphas_cumprod, device)
            predicted_noise = model(x_t, t, rb)

```

```

        loss = nn.functional.mse_loss(predicted_noise, noise)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if epoch % print_every == 0:
        losses.append(loss.item())
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

    return model, alphas, alphas_cumprod, losses

@torch.no_grad()
def sample_conditional_diffusion_radius(
    model: nn.Module, r: torch.Tensor, alphas: torch.Tensor, alphas_cumprod: torch.Tensor,
    n_samples: int = 100, T: int = 200
):
    model.eval()
    x = torch.randn(n_samples, 2, device=device)

    for t_idx in reversed(range(T)):
        t = torch.full((n_samples,), t_idx, device=device, dtype=torch.long)
        predicted_noise = model(x, t, r)

        alpha_t = alphas[t_idx]
        alpha_bar_t = alphas_cumprod[t_idx]

        if t_idx > 0:
            noise = torch.randn_like(x)
            alpha_bar_t_prev = alphas_cumprod[t_idx - 1]
            beta_t = 1 - alpha_t
            sigma_t = torch.sqrt(beta_t * (1 - alpha_bar_t_prev) / (1 - alpha_bar_t))
        else:
            noise = torch.zeros_like(x)
            sigma_t = 0.0

        x = (1 / torch.sqrt(alpha_t)) * (x - (1 - alpha_t) / torch.sqrt(1 - alpha_bar_t) * predicted_noise)
        x = x + sigma_t * noise

    return x.cpu().numpy()

# Train conditional diffusion with radius
print("Training conditional diffusion model (radius)...")

```

```
cdiff_radius, alphas_r, alphas_cumprod_r, losses_r = train_conditional_diffusion_radius(
    data, T=200, lr=1e-3, n_epochs=1000, batch_size=256, print_every=500
)
```

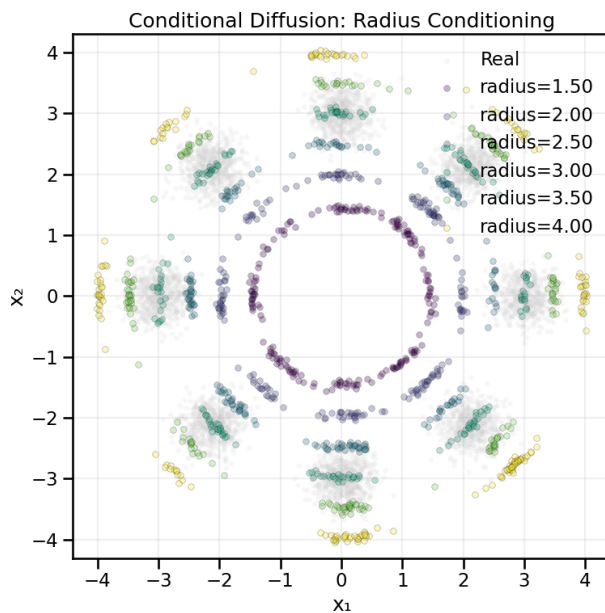
Training conditional diffusion model (radius)...

Epoch 0, Loss: 0.9504

Epoch 500, Loss: 0.4312

Epoch 500, Loss: 0.4312

Let's generate samples at different radius values:



Note that while these examples are simple, and thus work well for simple conditioning schemes where we just concatenate the conditioning vector, more complex conditioning (e.g., images or text) often requires specialized architectures like cross-attention layers or what are referred to as “guidance” methods (e.g., Classifier Guidance or Classifier-Free Guidance) to effectively incorporate the conditioning information. These techniques are beyond the scope of what I will cover in this introductory book but are important for state-of-the-art conditional diffusion models.

17.9 Summary and Next Steps

- **Diffusion Models as Time-Extended Score Models:** Diffusion models solve the problems of single-noise score matching by learning a score function across a continuum

of noise levels.

- **Forward Process:** Data is gradually noised into an isotropic Gaussian distribution, creating a series of well-behaved intermediate distributions $p_t(x)$.
- **Reverse Process:** A time-conditional neural network, $\epsilon_\theta(x_t, t)$, learns to predict the noise at any given time t . This network is trained with a simple MSE loss.
- **Generative Sampling:** By starting with pure noise and running the process in reverse, we can iteratively denoise the samples, guiding them onto the data manifold to produce realistic new data.

While we used a toy example of a 2D Ring of Gaussians to explore diffusion models, the general principles behind them are applied to generate high-resolution images, audio, and other complex data, often using more sophisticated network architectures like U-Nets. At their core, the concepts of the forward and reverse processes remain exactly the same. The main thing we had to give up to enable this new kind of generative model was the stochasticity of the sampling process; by introducing noise at each step, we traded off deterministic sampling for a more robust and flexible generative framework. In the next chapter, we will explore an idea that borrows concepts from both score matching and Continuous Normalizing Flows to define deterministic, yet flexible generative models called **Flow Matching**.

18 Flow Matching

In the previous chapters, we explored how Denoising Score Matching and Diffusion Models can learn to reverse a stochastic (noise-based) process to generate data. These models gradually destroy structure with noise and then learn to painstakingly reverse that process, step by noisy step.

This raises a question: **Can we achieve the same goal without noise?** What if we could learn a deterministic transformation that smoothly ‘flows’ a simple base distribution (like a Gaussian) into a complex data distribution?

This is the core idea behind **Flow Matching**. Instead of learning to reverse a noisy, stochastic process, we learn a deterministic vector field of velocities that directly transports probability mass. We saw this idea earlier in two cases: first with Optimal Transport GANs, where we had to solve a complex and expensive transport matching problem, and then later with (Continuous) Normalizing Flows, which required computing Jacobian determinants in high dimensions. Flow Matching inherits some of these ideas – notably the idea of learning a vector field to move data points from CNFs and the notion of pairing sets of points from Optimal Transport – but avoids many of their computational complexities by making certain approximations. In practice, this approach is often simpler to train, faster to sample from, and offers a different but powerful perspective on generative modeling that those we have looked at so far.

18.1 Learning Objectives

By the end of this notebook, you should have a strong intuition for:

- The conceptual difference between stochastic diffusion and deterministic flow matching.
- How to define a simple ‘path’ between a noise distribution and a data distribution.
- How to train a neural network to predict the ‘velocity’ along this path.
- How to generate samples by integrating this learned velocity field using an ODE solver.

Using device: cuda

18.2 The Noise-Free Limit

As we saw last chapter, Diffusion models work by reversing a stochastic process. The sampling is noisy and requires many steps because it has to undo the randomness that was added during the forward process.

Flow matching takes a more direct route. It learns a deterministic vector field $v_\theta(x_t, t)$ that defines the velocity of particles as they move from a base distribution p_0 to the target data distribution p_1 . The generative process becomes the solution to an ordinary differential equation (ODE):

$$\frac{dx_t}{dt} = v_\theta(x_t, t)$$

Starting with a sample $x_0 \sim p_0$, we can find the corresponding data sample x_1 by integrating this equation from $t = 0$ to $t = 1$. The path is smooth and deterministic. This is similar in concept to what we saw in the previous chapter on Continuous Normalizing Flows (CNFs), however, we will see below that they have different training strategies and properties.

18.3 Interpolating a Path

Like with CNFs, we need to define a velocity field $v_\theta(x_t, t)$ that smoothly transports samples from the base distribution to the data distribution over time. But how should we define the ‘correct’ velocity field to learn? In CNFs, we derived it from the instantaneous change of variables formula, but that required computing Jacobian determinants which can be expensive and required the velocity field to have certain regularity properties so that the integral could be computed efficiently. Moreover, training the velocity field required estimating an integral over the entire flow path, which can be computationally intensive, and the numerical ODE solver used to compute the integral had to be differentiable to allow backpropagation back to the learned velocity field.

In contrast, Flow Matching uses a simple but brilliant trick. We don’t need to simulate a complex forward process (as in Diffusion Models) or estimate some integral of a flow path (as in CNFs). Instead, we just define a synthetic ‘path’ by simply drawing a straight line between a sample from our base distribution and a sample from our data distribution. This pair of samples is often called a **synthetic coupling pair**.

Let $x_0 \sim p_{\text{base}}$ (e.g., a standard Gaussian) and $x_1 \sim p_{\text{data}}$ (e.g., our ring of Gaussians). Given these two sampled points, we define an interpolated point x_t at time $t \in [0, 1]$ as:

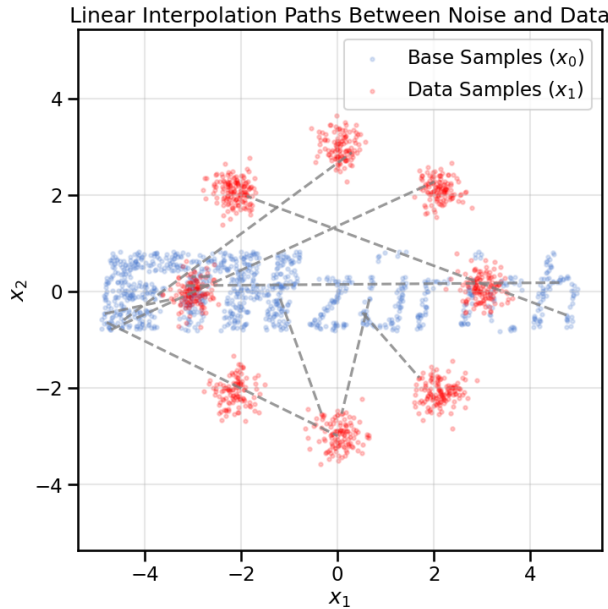
$$x_t = (1 - t)x_0 + tx_1$$

This gives us a point on the straight line connecting the noise sample to the data sample. The velocity along this path is constant and easy to calculate:

$$v_t(x_t) = \frac{dx_t}{dt} = x_1 - x_0$$

This is our **ground truth** velocity. For any interpolated point x_t , the target vector is simply the difference vector between the data point and the noise point that created it. This provides a direct, local supervision signal for training our neural network. Another nice benefit of this approach is that it doesn't place any particular restrictions on the base distribution – it can be Gaussian, uniform, or any other simple distribution from which we can easily sample, since all we need are coupled pairs of points (x_0, x_1) to define our paths. That is, Flow Matching doesn't require any specific form for p_{base} , unlike models such as Diffusion Models that assume (and in fact rely upon) on Gaussian noise to be the final base distribution.

To understand what this synthetic coupling means, let's visualize these straight-line paths.



Each dashed line represents a simple path from a noise sample to a data sample. Our goal is to train a vector field that will push particles along these kinds of paths.

i Why pick the pairs of points randomly?

It may first seem odd to randomly pair points from the base and data distributions. Why not try to find the 'best' matching pairs, as we did earlier when we covered Optimal Transport? In that chapter, we saw that by computing the optimal transport map, we

could minimize the total cost of moving probability mass, which also seems like it would be useful in flow matching and lead to better paths. One major downside of Optimal Transport is that finding the optimal matching can be computationally expensive, especially in high dimensions or with large datasets. Flow Matching sidesteps this complexity by using completely random pairings as we can see above.

But why should this work? Does a velocity field trained on random pairings still learn to transport mass correctly from the base to the data distribution?

This was the main insight from the [original Flow Matching paper by Lipman et al. \(2022\)](#). They showed (see Appendix A) that even with random pairings, the *expected* velocity field learned over many such random pairs still approximates the *true* optimal transport interpolant needed to move mass from the base to the data distribution. Intuitively, while any single random pairing may not represent an optimal path, averaging over many such paths provides a good approximation of the overall flow needed to transform one distribution into the other. Of course, you could also use more sophisticated pairing strategies, such as actually computing the optimal transport coupling, but what was impressive about the original Flow Matching paper was that the random pairing approach is simple, efficient, and works surprisingly well in practice.

18.4 Training via Velocity Regression

The training objective for Flow Matching is a straightforward regression problem. We want our neural network $v_\theta(x_t, t)$ to predict the true velocity, $x_1 - x_0$, and the loss function is a simple Mean Squared Error:

$$L(\theta) = \mathbb{E}_{t, x_0, x_1} \left[\|v_\theta((1-t)x_0 + tx_1, t) - (x_1 - x_0)\|^2 \right]$$

This is even simpler than the diffusion model objective. There are no noise schedules (α_t, β_t) to worry about.

18.4.1 The Training Pipeline

The training process for each step is as follows:

1. **Sample a Pair:** Pick a noise sample x_0 at random from the base distribution and a data sample x_1 at random from the training data.
2. **Sample Time:** Pick a random timestep t uniformly from $[0, 1]$.
3. **Interpolate:** Compute the point on the path: $x_t = (1-t)x_0 + tx_1$.
4. **Compute Target Velocity:** The target is simply $v_{target} = x_1 - x_0$.
5. **Predict Velocity:** Feed x_t and t into the neural network $v_\theta(x_t, t)$ to get the predicted velocity.

6. **Compute Loss:** Calculate the Mean Squared Error between the network's prediction and the target velocity.
7. **Update:** Backpropagate and update the network's weights.

Let's implement this. The network architecture will be very similar to the one we used for our diffusion model, as it also needs to be conditioned on time.

```
# Time-conditional network to predict velocity
class VelocityField(nn.Module):
    def __init__(self, time_emb_dim=16, hidden_dim=128):
        super().__init__()
        # We can't use an embedding layer for continuous time, so we'll use a small MLP
        self.time_mlp = nn.Sequential(
            nn.Linear(1, time_emb_dim),
            nn.ReLU(),
            nn.Linear(time_emb_dim, time_emb_dim),
            nn.ReLU(),
            nn.Linear(time_emb_dim, time_emb_dim)
        )
        self.net = nn.Sequential(
            nn.Linear(2 + time_emb_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 2) # Predicts the velocity vector
        )

    def forward(self, x, t):
        # t is expected to be of shape [batch_size, 1]
        t_emb = self.time_mlp(t)
        x_in = torch.cat([x, t_emb], dim=-1)
        return self.net(x_in)

# Training parameters
lr = 1e-3
if BASE_DISTRIBUTION == 'ethz_logo':
    n_epochs = 4000
else:
    n_epochs = 500
```

```

batch_size = 1000
loader = make_loader(data, batch_size=batch_size)

# Model and optimizer
flow_model = VelocityField().to(device)
optimizer = optim.AdamW(flow_model.parameters(), lr=lr)

# Training loop
losses = []
for epoch in range(n_epochs):
    for (x1_batch,) in loader:
        x1 = x1_batch.to(device)

        # 1. Sample noise
        if BASE_DISTRIBUTION == 'gaussian':
            x0 = torch.randn_like(x1)
        elif BASE_DISTRIBUTION == 'uniform':
            x0 = torch.rand_like(x1)*7.0 - 3.5 # Uniform noise in [-3.5, 3.5]
        elif BASE_DISTRIBUTION == 'ethz_logo':
            x0 = torch.Tensor(load_eth_z_logo_data(n_samples=x1.shape[0]))
        x0 = x0.to(device)

        # 2. Sample time
        t = torch.rand(x1.shape[0], 1, device=device)

        # 3. Interpolate
        xt = (1 - t) * x0 + t * x1

        # 4. Compute target velocity
        target_v = x1 - x0

        # 5. Predict velocity
        predicted_v = flow_model(xt, t)

        # 6. Compute loss
        loss = nn.functional.mse_loss(predicted_v, target_v)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if epoch % 50 == 0:

```

```

        losses.append(loss.item())
        print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

print("Training complete.")

# Plot the loss curve
plt.figure(figsize=(8, 4))
plt.plot(range(0, n_epochs, 50), losses)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.grid(True)
plt.show()

```

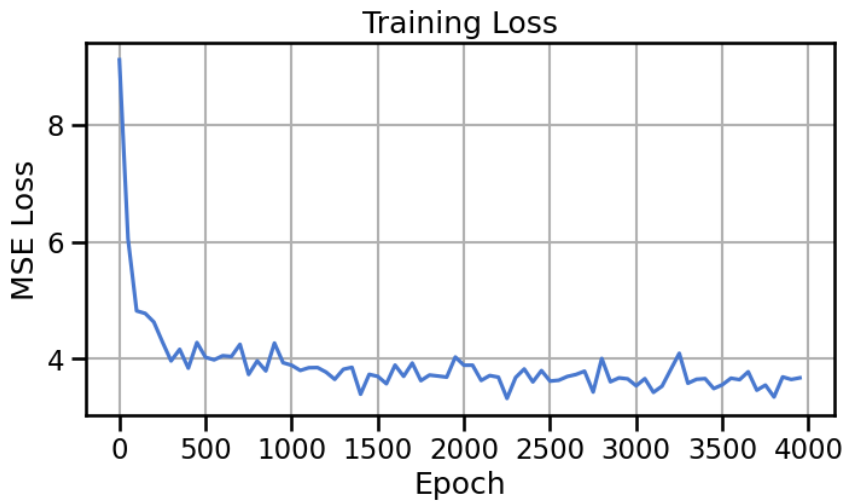
```

Epoch 0, Loss: 9.1081
Epoch 50, Loss: 6.0522
Epoch 100, Loss: 4.8192
Epoch 150, Loss: 4.7758
Epoch 200, Loss: 4.6283
Epoch 250, Loss: 4.2880
Epoch 300, Loss: 3.9641
Epoch 350, Loss: 4.1624
Epoch 400, Loss: 3.8411
Epoch 450, Loss: 4.2785
Epoch 500, Loss: 4.0259
Epoch 550, Loss: 3.9824
Epoch 600, Loss: 4.0530
Epoch 650, Loss: 4.0388
Epoch 700, Loss: 4.2475
Epoch 750, Loss: 3.7330
Epoch 800, Loss: 3.9644
Epoch 850, Loss: 3.7940
Epoch 900, Loss: 4.2690
Epoch 950, Loss: 3.9338
Epoch 1000, Loss: 3.8900
Epoch 1050, Loss: 3.8017
Epoch 1100, Loss: 3.8478
Epoch 1150, Loss: 3.8510
Epoch 1200, Loss: 3.7722
Epoch 1250, Loss: 3.6503
Epoch 1300, Loss: 3.8214
Epoch 1350, Loss: 3.8547

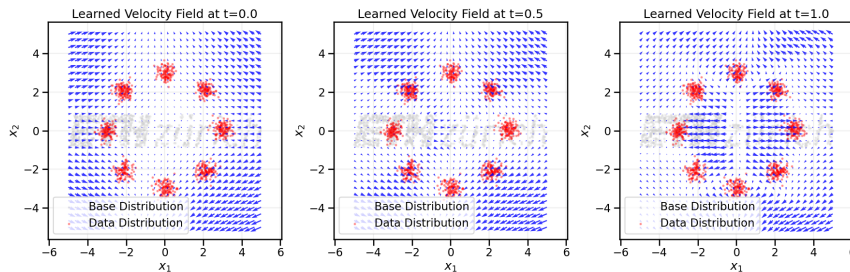
```

Epoch 1400, Loss: 3.3953
Epoch 1450, Loss: 3.7348
Epoch 1500, Loss: 3.6998
Epoch 1550, Loss: 3.5748
Epoch 1600, Loss: 3.8926
Epoch 1650, Loss: 3.7013
Epoch 1700, Loss: 3.9276
Epoch 1750, Loss: 3.6276
Epoch 1800, Loss: 3.7231
Epoch 1850, Loss: 3.7062
Epoch 1900, Loss: 3.6872
Epoch 1950, Loss: 4.0305
Epoch 2000, Loss: 3.8905
Epoch 2050, Loss: 3.8915
Epoch 2100, Loss: 3.6312
Epoch 2150, Loss: 3.7150
Epoch 2200, Loss: 3.6866
Epoch 2250, Loss: 3.3240
Epoch 2300, Loss: 3.6812
Epoch 2350, Loss: 3.8264
Epoch 2400, Loss: 3.6060
Epoch 2450, Loss: 3.7987
Epoch 2500, Loss: 3.6199
Epoch 2550, Loss: 3.6342
Epoch 2600, Loss: 3.6981
Epoch 2650, Loss: 3.7324
Epoch 2700, Loss: 3.7892
Epoch 2750, Loss: 3.4332
Epoch 2800, Loss: 4.0062
Epoch 2850, Loss: 3.6090
Epoch 2900, Loss: 3.6746
Epoch 2950, Loss: 3.6584
Epoch 3000, Loss: 3.5399
Epoch 3050, Loss: 3.6614
Epoch 3100, Loss: 3.4281
Epoch 3150, Loss: 3.5333
Epoch 3200, Loss: 3.8152
Epoch 3250, Loss: 4.0943
Epoch 3300, Loss: 3.5832
Epoch 3350, Loss: 3.6506
Epoch 3400, Loss: 3.6617
Epoch 3450, Loss: 3.4921
Epoch 3500, Loss: 3.5575

Epoch 3550, Loss: 3.6688
Epoch 3600, Loss: 3.6447
Epoch 3650, Loss: 3.7789
Epoch 3700, Loss: 3.4644
Epoch 3750, Loss: 3.5511
Epoch 3800, Loss: 3.3482
Epoch 3850, Loss: 3.6907
Epoch 3900, Loss: 3.6479
Epoch 3950, Loss: 3.6755
Training complete.



Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed y limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed x limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed x limits to fulfill fixed data aspect with adjustable data limits.
Ignoring fixed x limits to fulfill fixed data aspect with adjustable data limits.



18.5 Generating Samples by Solving the ODE

Once the velocity field $v_\theta(x, t)$ is trained, we can generate samples by solving the initial value problem defined by our ODE. We start with a random sample from the base distribution, $x_0 \sim \mathcal{N}(0, I)$, and integrate the velocity field from $t = 0$ to $t = 1$:

$$x_1 = x_0 + \int_0^1 v_\theta(x_t, t) dt$$

In practice, we use a numerical ODE solver to approximate this integral. A simple choice is the **Euler method**, where we take small, discrete steps in time:

$$x_{t+\Delta t} = x_t + v_\theta(x_t, t)\Delta t$$

We repeat this process from $t = 0$ to $t = 1$ to get our final sample. More sophisticated solvers like Runge-Kutta (RK4) can provide more accurate solutions with fewer steps.

Let's implement a simple Euler integrator and use it to generate samples. We will animate the process to see the smooth, deterministic flow from noise to data.

```
@torch.no_grad()
def generate_with_ode_solver(model, n_samples=500, n_steps=100, initial_noise_scale=1.0):
    """Generate samples by solving the ODE with Euler's method.
        Note here that unlike in CNFs, we don't require the ODE solver to be
        differentiable, because we are only using it for sampling points not training.
    """
    model.eval()

    # Start with pure noise
    if BASE_DISTRIBUTION == 'gaussian':
        x = torch.randn(n_samples, 2, device=device) * initial_noise_scale
    elif BASE_DISTRIBUTION == 'uniform':
        x = torch.rand(n_samples, 2, device=device)*7.0 - 3.5
    elif BASE_DISTRIBUTION == 'ethz_logo':
        x = torch.tensor(load_eth_z_logo_data(n_samples=n_samples), dtype=torch.float32, device=device)

    # Store trajectory for animation
    trajectory = [x.cpu().numpy()]

    dt = 1. / n_steps

    for i in range(n_steps):
```

```

        t = torch.full((n_samples, 1), i / n_steps, device=device)

        # Predict velocity
        v = model(x, t)

        # Euler step
        x = x + v * dt

        trajectory.append(x.cpu().numpy())

    return x.cpu().numpy(), np.array(trajectory)

# Generate samples and their trajectories
generated_samples_flow, trajectory_flow = generate_with_ode_solver(flow_model, n_samples=1000, n_steps=5

```

The animation shows the smooth, deterministic paths the samples take as they are pushed by the learned velocity field. Unlike the stochastic, zig-zagging paths of the diffusion sampler, these particles flow directly and efficiently toward the target distribution.

18.6 Conditional Generation with Flow Matching

As with diffusion models, Flow Matching can be extended to conditional generation tasks. By conditioning the velocity field on additional information (like class labels or text embeddings), we can guide the flow to generate samples that match specific conditions. For example, if we want to generate images of specific classes, we can modify our velocity network to take an additional input representing the class label. During training, we would sample pairs of points (x_0, x_1) along with their corresponding class labels and train the network to predict the velocity conditioned on these labels. Likewise, during sampling, we would start with a noise sample and integrate the ODE while providing the desired condition to the velocity network at each step.

Below, we will repeat our earlier exercises on conditional generation, this time using Flow Matching instead of Diffusion Models. As before, we will first train on a discrete set of classes (the ring of Gaussians) and note how the learned flow fields change as we modify the conditioning label. Then we will show a continuous conditioning example, where we condition on a continuous variable (the radius) to show how flow matching can condition on continuous variables.

```

class ConditionalVelocityField(nn.Module):
    """Velocity field conditioned on one-hot encoded mode labels."""

```

```

def __init__(self, n_classes=8, time_emb_dim=16, hidden_dim=128):
    super().__init__()
    self.time_mlp = nn.Sequential(
        nn.Linear(1, time_emb_dim),
        nn.ReLU(),
        nn.Linear(time_emb_dim, time_emb_dim),
        nn.ReLU(),
        nn.Linear(time_emb_dim, time_emb_dim),
    )
    # Concatenate: x (2) + time_emb (time_emb_dim) + one_hot (n_classes)
    self.net = nn.Sequential(
        nn.Linear(2 + time_emb_dim + n_classes, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, 2),
    )

def forward(self, x, t, c):
    """
    Args:
        x: [batch, 2] - spatial coordinates
        t: [batch, 1] - time
        c: [batch, n_classes] - one-hot encoded class labels
    """
    t_emb = self.time_mlp(t)
    x_in = torch.cat([x, t_emb, c], dim=-1)
    return self.net(x_in)

def train_conditional_flow_onehot(
    data, labels, n_epochs=500, batch_size=128, lr=1e-3, n_classes=8
):
    """Train the conditional velocity field with one-hot conditioning.
    """

    # Create conditional data loader
    cond_loader = make_conditional_loader_onehot(

```



```

    data=data, labels=labels, batch_size=batch_size, n_classes=n_classes
)

# Model and optimizer
model = ConditionalVelocityField(n_classes=8).to(device)
optimizer = optim.AdamW(model.parameters(), lr=lr)

# Training loop
losses = []
for epoch in range(n_epochs):
    epoch_loss = 0.0
    n_batches = 0
    for x1_batch, labels_batch in cond_loader:
        x1 = x1_batch.to(device)
        c = labels_batch.to(device)

        # Sample noise
        x0 = torch.randn_like(x1)

        # Sample time
        t = torch.rand(x1.shape[0], 1, device=device)

        # Interpolate
        xt = (1 - t) * x0 + t * x1

        # Compute target velocity
        target_v = x1 - x0

        # Predict velocity with conditioning
        predicted_v = model(xt, t, c)

        # Compute loss
        loss = nn.functional.mse_loss(predicted_v, target_v)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
        n_batches += 1

    avg_loss = epoch_loss / n_batches

```

```

        losses.append(avg_loss)

    if epoch % 50 == 0:
        print(f"Epoch {epoch}, Loss: {avg_loss:.4f}")

print("Conditional training complete.")

# Plot the loss curve
plt.figure(figsize=(8, 4))
plt.plot(range(n_epochs), losses)
plt.title("Conditional Flow Matching Training Loss (One-Hot)")
plt.xlabel("Epoch")
plt.ylabel("MSE Loss")
plt.grid(True)
plt.show()

return model

@torch.no_grad()
def sample_conditional_flow(model, condition, n_samples=100, n_steps=100):
    """Generate samples using conditional flow matching.

    Args:
        model: Conditional velocity field model
        condition: [batch, n_classes] one-hot encoded condition
        n_samples: Number of samples to generate
        n_steps: Number of ODE integration steps
    """
    model.eval()

    # Start with pure noise
    x = torch.randn(n_samples, 2, device=device)

    # Expand condition to match batch size
    c = condition.repeat(n_samples, 1)

    dt = 1.0 / n_steps

    for i in range(n_steps):
        t = torch.full((n_samples, 1), i / n_steps, device=device)
        v = model(x, t, c)

```

```

        x = x + v * dt

    return x.cpu().numpy()

def visualize_conditional_samples_onehot(model, n_modes=8, n_samples_per_mode=200):
    """Generate and visualize samples for each mode."""
    # Build samples dict expected by the utility plotting function
    samples_by_label = {}
    for mode_idx in range(n_modes):
        condition = torch.zeros(1, n_modes, device=device)
        condition[0, mode_idx] = 1.0
        samples = sample_conditional_flow(
            model, condition, n_samples=n_samples_per_mode
        )
        samples_by_label[int(mode_idx)] = samples

    # Load reference real data for background and labels
    real_data, real_labels = create_ring_gaussians(n_samples=5000, n_modes=n_modes)

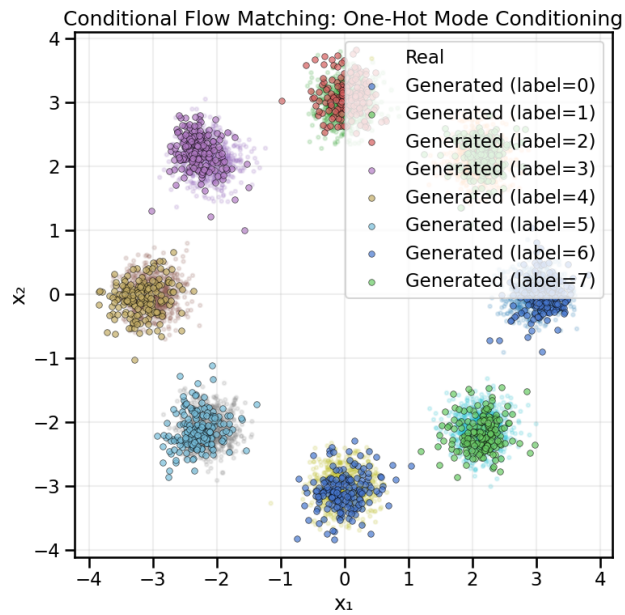
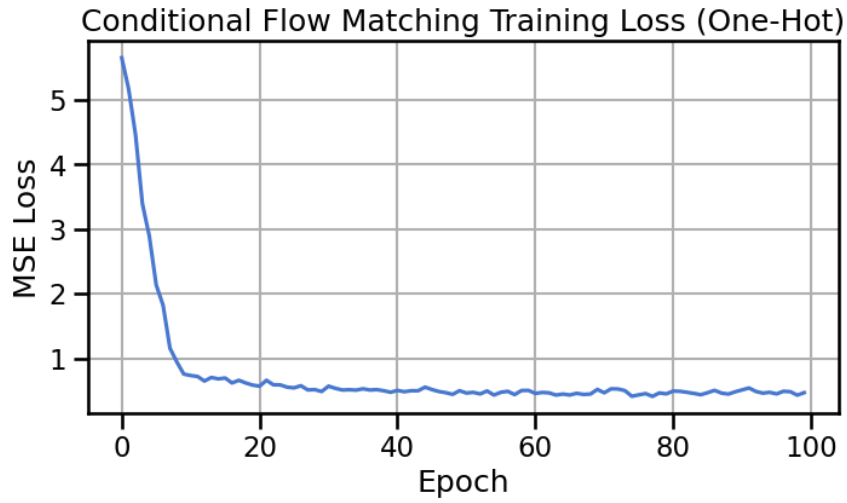
    # Visualize using utilities (expects dict and real data/labels)
    plot_conditional_samples_discrete(
        samples_by_label,
        real_data=real_data,
        real_labels=real_labels,
        title="Conditional Flow Matching: One-Hot Mode Conditioning",
    )

# Train conditional model with one-hot encoding
model_onehot = train_conditional_flow_onehot(data = data, labels=labels,
                                             n_epochs=100, batch_size=128, lr=1e-3)

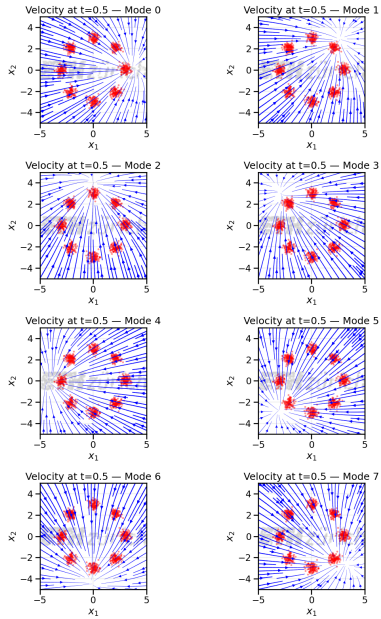
# Visualize results
visualize_conditional_samples_onehot(
    model_onehot, n_modes=8, n_samples_per_mode=200
)

```

Epoch 0, Loss: 5.6546
 Epoch 50, Loss: 0.4630
 Conditional training complete.



By looking at the flow fields below, we can see how the conditioning label influences the direction and magnitude of the velocity vectors, guiding samples toward the desired mode in the ring of Gaussians example.



18.6.1 Example 2: Conditioning on the Continuous Radius

```
class ConditionalVelocityFieldRadius(nn.Module):
    """Velocity field conditioned on continuous radius values."""

    def __init__(self, time_emb_dim=16, radius_emb_dim=16, hidden_dim=128):
        super().__init__()
        self.time_mlp = nn.Sequential(
            nn.Linear(1, time_emb_dim),
            nn.ReLU(),
            nn.Linear(time_emb_dim, time_emb_dim),
            nn.ReLU(),
            nn.Linear(time_emb_dim, time_emb_dim),
        )
        self.radius_mlp = nn.Sequential(
            nn.Linear(1, radius_emb_dim),
            nn.ReLU(),
            nn.Linear(radius_emb_dim, radius_emb_dim),
            nn.ReLU(),
            nn.Linear(radius_emb_dim, radius_emb_dim),
        )
        # Concatenate: x (2) + time_emb + radius_emb
        self.net = nn.Sequential(
```

```

        nn.Linear(2 + time_emb_dim + radius_emb_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, 2),
    )

def forward(self, x, t, r):
    """
    Args:
        x: [batch, 2] - spatial coordinates
        t: [batch, 1] - time
        r: [batch, 1] - radius values
    """
    t_emb = self.time_mlp(t)
    r_emb = self.radius_mlp(r)
    x_in = torch.cat([x, t_emb, r_emb], dim=-1)
    return self.net(x_in)

def train_conditional_flow_radius(data, n_epochs=500, batch_size=128, lr=1e-3):
    """Train the conditional velocity field with radius conditioning.
    """
    # if data is None:
    #     data, _ = create_ring_gaussians(n_samples=5000)

    # Create conditional data loader with radius
    radius_loader = make_conditional_loader_radius(data=data, batch_size=batch_size)

    # Model and optimizer
    model = ConditionalVelocityFieldRadius().to(device)
    optimizer = optim.AdamW(model.parameters(), lr=lr)

    # Training loop
    losses = []
    for epoch in range(n_epochs):
        epoch_loss = 0.0
        n_batches = 0

```

```

for x1_batch, radius_batch in radius_loader:
    x1 = x1_batch.to(device)
    r = radius_batch.to(device)

    # Sample noise
    x0 = torch.randn_like(x1)

    # Sample time
    t = torch.rand(x1.shape[0], 1, device=device)

    # Interpolate
    xt = (1 - t) * x0 + t * x1

    # Compute target velocity
    target_v = x1 - x0

    # Predict velocity with radius conditioning
    predicted_v = model(xt, t, r)

    # Compute loss
    loss = nn.functional.mse_loss(predicted_v, target_v)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    epoch_loss += loss.item()
    n_batches += 1

avg_loss = epoch_loss / n_batches
losses.append(avg_loss)

if epoch % 50 == 0:
    print(f"Epoch {epoch}, Loss: {avg_loss:.4f}")

print("Conditional training with radius complete.")

# Plot the loss curve
plt.figure(figsize=(8, 4))
plt.plot(range(n_epochs), losses)
plt.title("Conditional Flow Matching Training Loss (Radius)")
plt.xlabel("Epoch")

```

```

plt.ylabel("MSE Loss")
plt.grid(True)
plt.show()

return model

@torch.no_grad()
def sample_conditional_flow_radius(model, radius_value, n_samples=100, n_steps=100):
    """Generate samples using conditional flow matching with radius conditioning.

    Args:
        model: Conditional velocity field model
        radius_value: Scalar radius value to condition on
        n_samples: Number of samples to generate
        n_steps: Number of ODE integration steps
    """
    model.eval()

    # Start with pure noise
    x = torch.randn(n_samples, 2, device=device)

    # Create radius condition
    r = torch.full((n_samples, 1), radius_value, device=device)

    dt = 1.0 / n_steps

    for i in range(n_steps):
        t = torch.full((n_samples, 1), i / n_steps, device=device)
        v = model(x, t, r)
        x = x + v * dt

    return x.cpu().numpy()

def visualize_conditional_samples_radius(
    model, radius_values=None, n_samples_per_radius=200
):
    """Generate and visualize samples at different radius values."""

    if radius_values is None:
        radius_values = [1.0, 1.5, 2.0, 2.5, 3.0, 3.5]

```



```

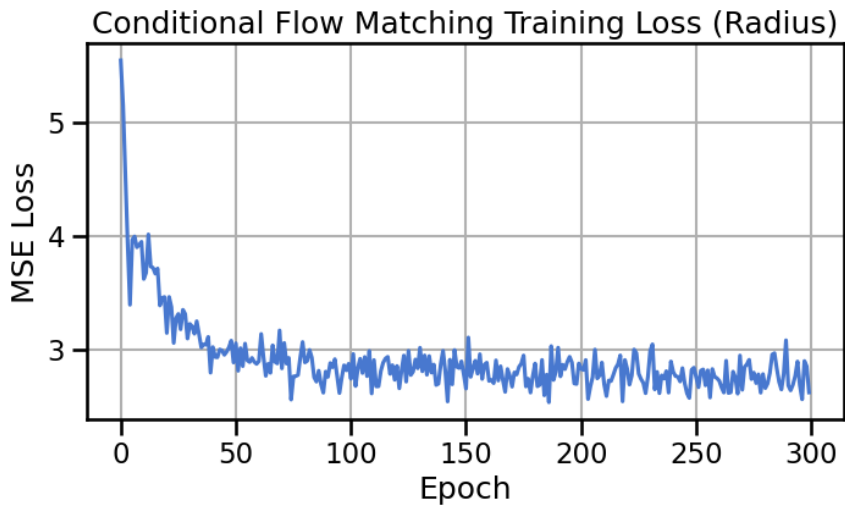
samples_by_value = {}
for radius_val in radius_values:
    samples = sample_conditional_flow_radius(
        model, radius_val, n_samples=n_samples_per_radius
    )
    samples_by_value[float(radius_val)] = samples

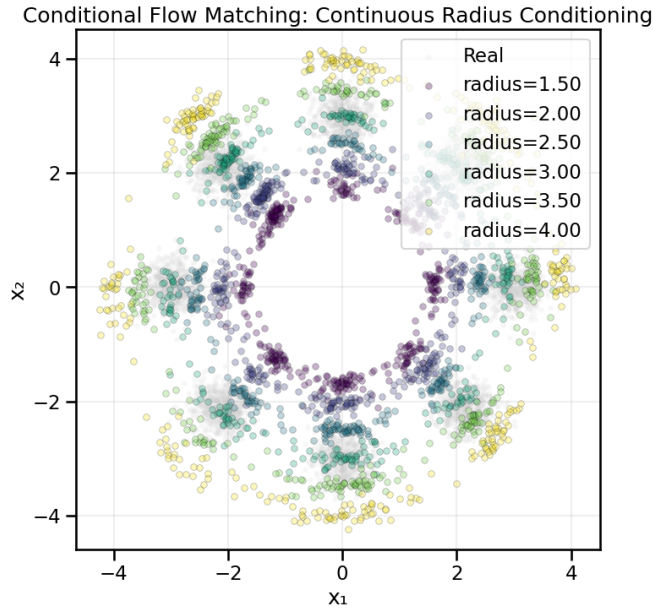
# Load real data for background
real_data, _ = create_ring_gaussians(n_samples=5000)

# Visualize
plot_conditional_samples_continuous(
    samples_by_value,
    real_data=real_data,
    title="Conditional Flow Matching: Continuous Radius Conditioning",
)

```

Epoch 0, Loss: 5.5464
 Epoch 50, Loss: 3.0524
 Epoch 100, Loss: 2.7391
 Epoch 150, Loss: 2.6528
 Epoch 200, Loss: 2.8269
 Epoch 250, Loss: 2.7486
 Conditional training with radius complete.





18.7 Summary and Next Steps

Flow Matching combines some of the best aspects of both Normalizing Flows and Diffusion Models:

Aspect	Diffusion Models	Normalizing Flows (CNF)	Flow Matching
Dynamics	Stochastic (SDE)	Deterministic (ODE)	Deterministic (ODE)
Training	Score/Noise Regression	Max Likelihood (trace)	Velocity Regression
Path	Random, noisy	Invertible, complex	Simple, straight-line
Sampling Speed	Slow (many steps)	Fast (single pass)	Fast (few ODE steps)
Stability	Very stable	Can be unstable	Very stable

Flow Matching avoids the expensive trace computation of CNFs and the slow, stochastic sampling of diffusion models. It provides a stable, efficient, and conceptually simple way to learn a generative model.

- **Flow Matching vs. Diffusion:** Flow Matching learns a deterministic ODE, while diffusion models learn to reverse a stochastic SDE. This leads to faster, noise-free sampling.

- **Simple Training:** The model is trained by regressing a time-conditional neural network on the simple velocity vector defined by a straight-line path between noise and data samples.
- **ODE-based Generation:** Samples are generated by integrating the learned velocity field from $t = 0$ to $t = 1$ using a numerical ODE solver.

Through this part of the book we have covered a wide variety of generative models, ranging from GANs to VAEs, Normalizing Flows, Diffusion Models, and now Flow Matching. Yet many of these models struggle in high-dimensional settings. A popular and commonly used approach is to combine these Generative Models with reduced or latent dimensional representations, such as those learned by Autoencoders or Variational Autoencoders. In the next chapter we will see how to combine these two threads together to produce “Latent Generative Models”, such as Latent Diffusion Models or Latent Flow Models, which leverage the strengths of both generative modeling and representation learning.

19 Latent Generative Models

In the previous chapters, we’ve seen powerful generative models like GANs, VAEs, and Flow Matching that can learn complex data distributions. However, many of these models face challenges when data lies on a complicated, low-dimensional manifold embedded in a high-dimensional space. For example, imagine trying to learn a distribution of images of faces. The “space of all possible faces” is a complex, twisted surface within the vast space of all possible pixel combinations.

A powerful strategy to tackle this is to combine **representation learning** with **generative modeling**. The idea is to first learn a “simpler” latent space that “unrolls” or “flattens” the data manifold. Then, we can train a generative model in this much simpler latent space, where tasks like density estimation or interpolation are significantly easier. Finally, we can map the generated latent points back to the original data space.

This is the core idea behind **Latent Generative Models**. In this notebook, we will explore this concept by:

1. **Learning a Latent Space:** We’ll train a simple autoencoder to “unroll” the classic Swiss Roll dataset, a 2D spiral manifold.
2. **Training Generative Models:** We will train two generative models: one directly on the complex Swiss Roll data and another on the simple latent representation. We will use GANs and Flow Matching, just to compare different models.
3. **Comparing the Results:** We will visualize the generated samples from both approaches to see how learning in the latent space can lead to superior results.

19.1 Learning Objectives

- Understand the motivation for latent generative models.
- Implement a simple autoencoder to learn a latent representation of a manifold.
- Train a Flow Matching and GAN model in both the original data space and the learned latent space.
- Visualize and compare the quality of samples generated from both approaches.

19.2 Setup and Data Generation

We'll import the Swiss Roll dataset and visualize it. We can see that the data lies on a complex 2D manifold embedded in 3D space.

Using device: cuda

Data shape (should be N x 3): torch.Size([2000, 3])

[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

19.3 Initial Baselines: Generative Models in the Original Data Space

First we will train two generative models on the original data space: a GAN and a Flow Matching model to see how they do on this complex distribution. This will form a baseline for comparison when we introduce learning within a latent space. These are largely identical to the implementations we used in prior chapters, with minor adjustments to make them compatible with the Swiss Roll data.

19.3.1 Flow Matching Model

```
# Train data-space flow model
print("Training data-space flow model (3D)...")
flow_data = train_flow(VelocityField(data_dim=ambient_dim).to(device), X,
                        epochs=epochs_generative_model)

# Generate samples
samples_flow_data, traj_data = sample_flow(flow_data, data_dim=ambient_dim, n_samples=num_generated_samples)
samples_flow_data_np = samples_flow_data.detach().cpu().numpy() if hasattr(samples_flow_data, 'detach')
```

```
Training data-space flow model (3D)...
Flow Loss (0): 2.1931
Flow Loss (1000): 1.5469
Flow Loss (2000): 1.5517
Flow Loss (3000): 1.5562
Flow Loss (4000): 1.5008
```

```
Flow Loss (5000): 1.4315
Flow Loss (6000): 1.5096
Flow Loss (7000): 1.5889
Flow Loss (8000): 1.5116
Flow Loss (9000): 1.4904
Final loss: 1.4904
```

19.3.2 GAN Model

```
# Train GAN in original 3D data space
print("Training data-space GAN (3D)...")
G_data, D_data = train_gan(X, data_dim=ambient_dim, noise_dim=8, epochs=epochs_generative_model, device=device)

# Generate samples
samples_gan_data = sample_gan(G_data, n_samples=num_generated_samples, noise_dim=8, device=device)
samples_gan_data_np = samples_gan_data.detach().cpu().numpy() if hasattr(samples_gan_data, 'detach') else
```

```
Training data-space GAN (3D)...
GAN step 1000: D=1.191 | G=0.967
GAN step 2000: D=1.283 | G=0.817
GAN step 3000: D=1.305 | G=0.795
GAN step 4000: D=1.341 | G=0.805
GAN step 5000: D=1.352 | G=0.740
GAN step 6000: D=1.372 | G=0.690
GAN step 7000: D=1.332 | G=0.733
GAN step 8000: D=1.341 | G=0.779
GAN step 9000: D=1.339 | G=0.785
GAN step 10000: D=1.295 | G=0.762
```

19.3.3 Comparison of the Baseline Models in the Original Data Space

Now let's compare these baseline models with the original data distribution, in 3D space.

[Interactive output: available in the online version of this chapter.]

As we can see above, training the generative model directly in data space can perform OK in this simple example, although for higher-dimensional problems it may struggle. To demonstrate a different strategy, let's first learn a low-dimensional compressed representation, and then train the generative models on that latent representation to see how it compares. These are generally referred to as **Latent Generative Models**.

19.4 Build a Least Volume Autoencoder to Learn the Latent Space

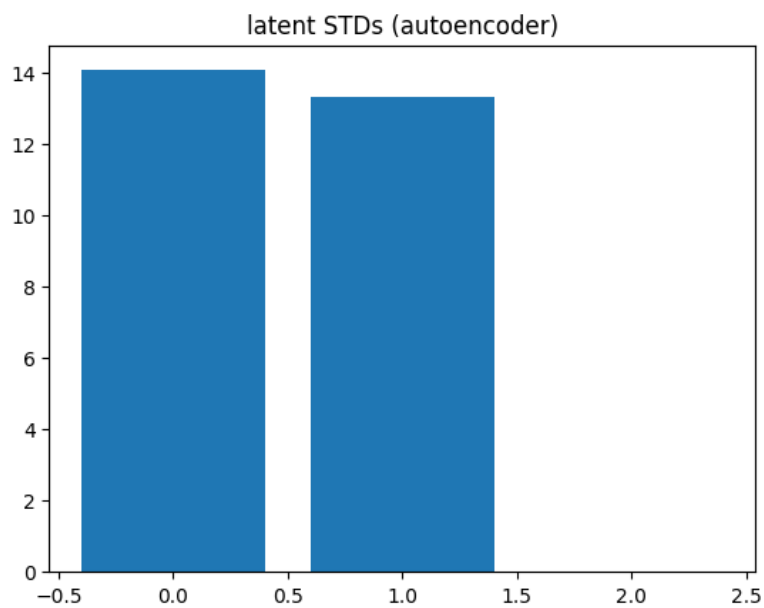
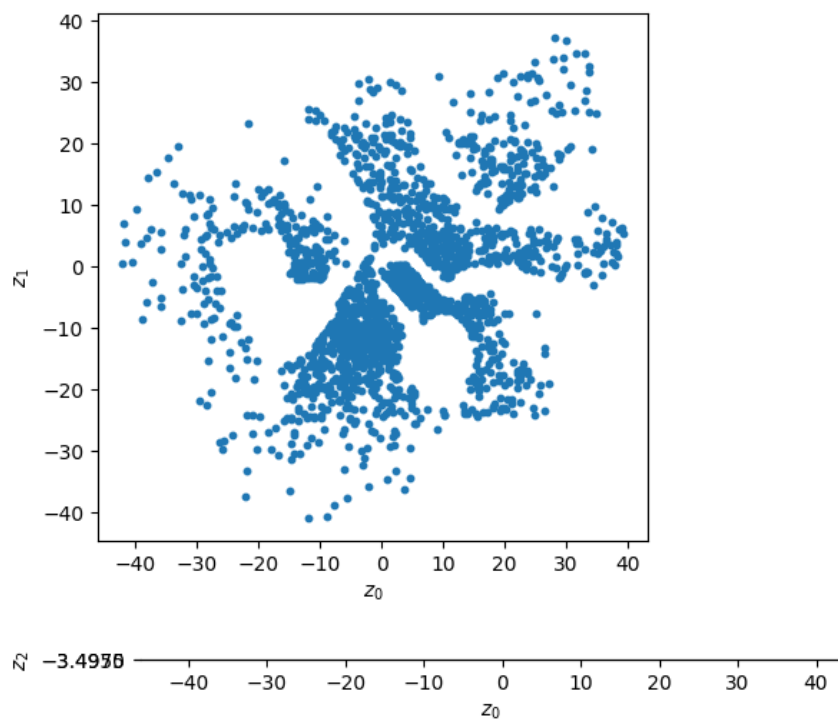
Our first goal is to learn a mapping from the complex Swiss Roll manifold to a simpler, “unrolled” latent space. As we saw in prior chapters, an autoencoder can be a good tool for this, although any dimension reduction method that allows for both compression and reconstruction could work. As we saw in the prior chapter on unsupervised Neural Network models, it consists of two parts:

1. **Encoder:** A neural network that compresses the input data into a low-dimensional latent representation.
2. **Decoder:** A neural network that reconstructs the original data from the latent representation.

By training the autoencoder to minimize the reconstruction error (the difference between the original and reconstructed data), the encoder learns to capture the most important features of the data in its latent space. Specifically, we will use a Least Volume Autoencoder, so that we can identify and select the lowest dimensional representation of the data. For the Swiss Roll, we can see from above that this should result in a latent space where the spiral structure is flattened out, in which case the generative models only has to learn to reconstruct a 2D distribution, rather than a 3D one.

```
Epoch 999: rec = 0.041504, vol = 0.90292
Epoch 1999: rec = 0.026399, vol = 1.0715
Epoch 2999: rec = 0.019554, vol = 1.1883
Epoch 3999: rec = 0.015653, vol = 1.28
Epoch 4999: rec = 0.013543, vol = 1.3123
Epoch 5999: rec = 0.010756, vol = 1.3138
Epoch 6999: rec = 0.0091636, vol = 1.3596
Epoch 7999: rec = 0.0096072, vol = 1.3558
Epoch 8999: rec = 0.0073822, vol = 1.3531
Epoch 9999: rec = 0.0071178, vol = 1.3433
Epoch 10999: rec = 0.0066327, vol = 1.3494
Epoch 11999: rec = 0.0060909, vol = 1.34
Epoch 12999: rec = 0.0060377, vol = 1.3203
Epoch 13999: rec = 0.0056103, vol = 1.3167
Epoch 14999: rec = 0.0053546, vol = 1.3116
Epoch 15999: rec = 0.005258, vol = 1.3159
Epoch 16999: rec = 0.0050283, vol = 1.2932
Epoch 17999: rec = 0.0050594, vol = 1.2874
Epoch 18999: rec = 0.004842, vol = 1.2819
Epoch 19999: rec = 0.0050161, vol = 1.2694
```

[Interactive output: available in the online version of this chapter.]



As you can see, the autoencoder has unrolled the spiral structure of the Swiss Roll into a 2D distribution in the latent space. This is the “unrolled” manifold. Let’s see if learning a

generative model on this simple distribution is easier for the two generative models, and then we can use the decoder to project those generated points back into 3D space.

19.5 Retrain the Generative Models on the Latent Space

Now we will repeat our two generative models from above, except this time we will train them on the latent space learned by the autoencoder (i.e., $Z = \text{encoder}(X)$). We will keep the same architectures and training procedures for the generative models, but the key difference is just the input data, which now lives in a 2D space.

```
##### Latent Flow Matching #####
# Train latent-space flow model using LVA encoder (top-k dims)
with torch.no_grad():
    Z_full = encoder(X)                # shape [N, ambient_dim]
    latent_std = Z_full.std(0)
    top_k = 3                          # choose latent dimensionality to model
    latent_idx = torch.argsort(latent_std, descending=True)[:top_k]
    Z = Z_full[:, latent_idx]          # shape [N, top_k]

print(f"Training latent-space flow model (LVA top-{top_k} latent)...")
flow_latent = train_flow(VelocityField(data_dim=top_k).to(device), Z,
                        epochs=epochs_generative_model)

# Use the same top-k (and indices) selected during training for latent flow
samples_latent_flow, traj_latent = sample_flow(flow_latent, data_dim=top_k, n_samples=num_generated_samples)
with torch.no_grad():
    # Pad latent samples back to ambient_dim with zeros in the non-selected dims
    full_latent_flow = torch.zeros(samples_latent_flow.shape[0], ambient_dim, device=device)
    full_latent_flow[:, latent_idx] = samples_latent_flow.to(device)
    decoded_latent = decoder(full_latent_flow).detach().cpu()
print("Latent Flow sampling complete.")

print(f"Training latent-space GAN (top-{top_k} LVA dims)...")
G_latent, D_latent = train_gan(Z, data_dim=top_k, noise_dim=8, epochs=epochs_generative_model, device=device)

# Sample from both GANs
samples_latent_gan = sample_gan(G_latent, n_samples=num_generated_samples, noise_dim=8, device=device)

# Decode latent GAN samples back to 3D
with torch.no_grad():
    full_latent_gan = torch.zeros(samples_latent_gan.shape[0], ambient_dim, device=device)
```

```

full_latent_gan[:, latent_idx] = samples_latent_gan.to(device)
decoded_gan_latent = decoder(full_latent_gan).detach().cpu()

print("Latent GAN sampling complete.")

```

Training latent-space flow model (LVA top-3 latent)...

Flow Loss (0): 134.2461

Flow Loss (1000): 14.5121

Flow Loss (2000): 16.1711

Flow Loss (3000): 13.6451

Flow Loss (4000): 14.6157

Flow Loss (5000): 12.9395

Flow Loss (6000): 10.8772

Flow Loss (7000): 16.4385

Flow Loss (8000): 15.1911

Flow Loss (9000): 12.5841

Final loss: 12.5841

Latent Flow sampling complete.

Training latent-space GAN (top-3 LVA dims)...

GAN step 1000: D=1.067 | G=0.994

GAN step 2000: D=1.104 | G=0.885

GAN step 3000: D=1.140 | G=0.879

GAN step 4000: D=1.440 | G=0.645

GAN step 5000: D=1.300 | G=0.675

GAN step 6000: D=1.252 | G=0.813

GAN step 7000: D=1.295 | G=0.842

GAN step 8000: D=1.330 | G=0.839

GAN step 9000: D=1.420 | G=0.791

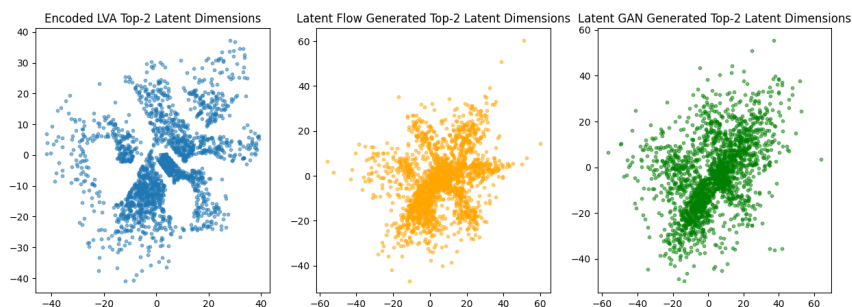
GAN step 10000: D=1.360 | G=0.844

Latent GAN sampling complete.

19.6 Sampling and Decoding to Compare to Original Data

Unlike the previous baseline models trained directly in data space, we now need to decode the generated latent samples back into the original 3D space for comparison. To do this, after generating samples in the latent space (2D), we must pass them through the `decoder` to map them back into the original 3D data space.

Below we will first plot how well the generated models do in the latent space itself, and then we will decode those samples back to 3D and compare to the original data and the data-space generative model samples.



[Interactive output: available in the online version of this chapter.]

19.7 Summary

This notebook demonstrated the concept of Latent Generative Models by combining representation learning with generative modeling. By first learning a low-dimensional latent space that “unrolls” the complex Swiss Roll manifold, we were able to train generative models that only need to sample points in a lower-dimensional space, which we hope is simpler. In this particular example, given that the original 3D distribution was fairly straightforward and not too difficult, both Flow Matching and GANs did fine, and we perhaps did not need the additional compression offered by the Autoencoder. However, this technique of Latent Generative Models is typically deployed on examples that have much higher complexity and dimensionality, such as images or 3D shapes. For example, commonly used latent generative models include VAE-GANs or Latent Diffusion Models, which have shown great success in generating high-quality images or 3D data. We also see similar approaches used in robotics and control, such as in model-based reinforcement learning, where learning a latent representation of the environment dynamics can greatly simplify planning and control tasks (e.g., [PlaNet](#), [Dreamer](#)).

19.8 Review of Generative Models Covered So Far and Next Steps

This chapter concludes our exploration of generative models in this textbook so far, with the exception of auto-regressive models (e.g., transformers) which we will cover later. However, now is a good time to briefly recap what we’ve discussed so far and to compare their strengths and weaknesses or areas of common application. Specifically, we have covered:

- **Generative Adversarial Networks (GANs):** Powerful for generating high-quality samples, especially in image domains. However, they can be difficult to train and may suffer from mode collapse.
- **Variational Autoencoders (VAEs):** Provide a principled way to learn latent representations and generate samples. They are generally more stable to train than GANs but historically were known to produce blurrier samples, however this is not always the case

in modern implementations. Importantly, they provide a fully probabilistic framework that allows us to compute posterior distributions over latent variables as well as data likelihoods.

- **Normalizing Flows:** Allow for exact likelihood computation and efficient sampling through the use of invertible transformations and the change of variables formula. They are flexible but can be computationally intensive for many layers, and their main downside is that they require careful architecture design to maintain invertibility. Like VAEs, they also provide a fully probabilistic framework without requiring the variational approximations used in VAEs.
- **Continuous Normalizing Flows (CNFs) and Flow Matching:** These offer a continuous-time perspective on flows, enabling more flexible transformations that do not require chaining together many discrete transformations and log determinant terms. They can be more efficient in certain scenarios but can also be complex to implement (e.g., CNFs require differentiable flow solvers and may not work well in high dimensions). Flow Matching simplifies some of these issues by avoiding the need for backpropagating through an ODE solver.
- **Diffusion Models:** These model the data generation process as a Stochastic Differential Equation (SDE) or a discrete-time process that gradually denoises data from pure noise to a data sample via Score Matching. Are known for their ability to generate high-quality samples, particularly in image generation tasks. They are generally more robust than GANs but can be much slower to sample from due to the iterative denoising process needed to generate samples. In contrast, direct push-forward models like GANs or VAEs can generate samples significantly faster.
- **Latent Generative Models:** These combine representation learning with generative modeling to often handle complex high dimensional data distributions more effectively. They are particularly useful when data lies on low-dimensional manifolds within high-dimensional spaces that can be “unrolled” easily into a simpler latent space.

To put some of these models into an engineering context, the next chapter will demonstrate or explore common applications of generative models in practical engineering settings, such as Inverse Design or Surrogate modeling. After that, we will address the last major class of generative models: Auto-Regressive Models, which have become extremely popular in recent years, especially in the context of large language models (LLMs) and transformers, and generally for data expressed as sequences.

20 Introduction to Deep Reinforcement Learning

In this chapter we will briefly review key reinforcement learning (RL) algorithms, and implement them on the Acrobot-v1 environment from the [Gymnasium library](#). Specifically, we will review the differences between value-based (sometimes called “Critic Methods”) and policy-based methods (sometimes called “Actor Methods”), and we to demonstrate the differences, we will implement Deep Q-Networks (DQN), Policy Gradient (REINFORCE), Vanilla Actor-Critic (AC), and finally Proximal Policy Optimization.

20.1 Learning Objectives

- Understand how value-based, policy-based, and actor-critic methods differ conceptually and algorithmically.
- Interpret diagnostic plots (learning curves, value plots, policy visualizations).
- Understand the strengths and weaknesses of each method on a common example.
- Gain intuition about fundamental parameters of each model (learning rate, discount factor, exploration rate, entropy regularization, etc.) and how they affect learning.

```
Using cpu (USE_GPU=False)
```

First let’s take a look at the Acrobot-v1 environment:

```
The action space is Discrete(3)
A sample action is: 2
The observation space is Box([ -1.      -1.      -1.      -1.      -12.566371 -28.274334], [ 1.      1.      1.      1.      12.566371 28.274334])
A sample observation is: [-0.5119918 -0.57492304 0.10808961 0.90896344 -3.5385666 15.979161 ]
```

We see that the goal of the agent is to swing the lower link up to a given height by applying torques at the joint between the two links, much like you would do with your own hips if you had to swing yourself upward while hanging on a bar. What is not obvious from the printed action/observation spaces and the environment render is that the agent receives a reward of -1 for each time step until it reaches the goal, encouraging it to reach the goal as quickly as

possible. We can see this below by running a random policy (which shouldn't do too well) for several episodes and summing up the total rewards received.

```
# Reset environment to start a new episode
observation, info = env.reset()

print(f"Starting observation: {observation}")

episode_over = False
total_reward = 0

for i in range(500):
    # Choose an action
    action = env.action_space.sample() # Random action for now - real agents will be smarter!

    # Take the action and see what happens
    observation, reward, terminated, truncated, info = env.step(action)

    total_reward += reward
    episode_over = terminated or truncated

print(f"Episode finished! Total reward: {total_reward}")
env.close()
```

```
Starting observation: [ 0.99901444 -0.04438601  0.9955945  -0.09376328  0.00158421 -0.04944538]
Episode finished! Total reward: -500.0
```

20.2 Deep Q-Network (DQN)

In Deep Q-Networks (DQN), we learn a state-action value function $Q(s, a)$ that estimates the expected return from taking action a in state s and following the current policy thereafter. The policy is derived from the Q-function by selecting the action with the highest estimated value, often with some exploration strategy like ϵ -greedy, to prevent getting stuck in suboptimal policies. Specifically, a small MLP predicts $Q(s, a)$ a continuous space of angles and velocities, and then has to predict the Q-values for a discrete set of actions: applying positive torque, negative torque, or no torque. To select actions, the agent picks the action with the highest Q-value for the current state:

$$a_t = \arg \max_a Q_\theta(s_t, a).$$

To encourage exploration the policy is ϵ -greedy, which means that it takes the Q-maximizing action with probability $1 - \epsilon$ and a random action with probability ϵ .

To train the Q-function, we use the Bellman equation as a target for supervised learning:

$$y_t = r_t + \gamma \max_{a'} Q_{\theta}(s_{t+1}, a'),$$

where r_t is the reward received after taking action a_t in state s_t , and γ is the discount factor that determines the importance of future rewards. The target y_t represents the estimated return for taking action a_t in state s_t . The Q-function parameters θ are updated to minimize the difference between the predicted Q-value and the target using mean-squared error loss. Specifically, we minimize the mean-squared Temporal Difference (TD) error:

$$\mathcal{L}_{\text{TD}} = (Q_{\theta}(s_t, a_t) - y_t)^2$$

typically using a variants of stochastic gradient descent, and over a batch of transitions sampled from a replay buffer to improve stability and sample efficiency.

```
# Simple Discrete Deep Q-Network (DQN) Agent
class DQNAgent:
    def __init__(self, state_dim: int, action_dim: int, hidden_dim: int = 128, lr: float = 1e-3):
        self.q_network = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim)
        ).to(DEVICE)
        self.optimizer = torch.optim.Adam(self.q_network.parameters(), lr=lr)
        self.criterion = nn.MSELoss()

    def select_action(self, state: np.ndarray, epsilon: float = 0.0) -> int:
        if random.random() < epsilon:
            return random.randint(0, self.q_network[-1].out_features - 1)
        state_tensor = torch.FloatTensor(state).unsqueeze(0).to(DEVICE)
        with torch.no_grad():
            q_values = self.q_network(state_tensor)
        return q_values.argmax().item()

    def train(self, batch: List[Tuple[np.ndarray, int, float, np.ndarray, bool]], gamma: float):
        states, actions, rewards, next_states, dones = zip(*batch)

        states_tensor = torch.FloatTensor(states).to(DEVICE)
        actions_tensor = torch.LongTensor(actions).unsqueeze(1).to(DEVICE)
        rewards_tensor = torch.FloatTensor(rewards).unsqueeze(1).to(DEVICE)
        next_states_tensor = torch.FloatTensor(next_states).to(DEVICE)
        dones_tensor = torch.FloatTensor(dones).unsqueeze(1).to(DEVICE)
```

```

current_q_values = self.q_network(states_tensor).gather(1, actions_tensor)
with torch.no_grad():
    max_next_q_values = self.q_network(next_states_tensor).max(1)[0].unsqueeze(1)
    target_q_values = rewards_tensor + (gamma * max_next_q_values * (1 - dones_tensor))

loss = self.criterion(current_q_values, target_q_values)

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()

```

```

# Now we can instantiate the DQNAgent and use it to interact with Acrobot-v1 environment.
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n # Ensure the action space is discrete for DQN
dqn_agent = DQNAgent(state_dim, action_dim)

```

```

# The main training loop for the DQN agent
def train_dqn_agent(env: gym.Env, agent: DQNAgent, num_episodes: int = 700, batch_size: int = 64,
                    gamma: float = 0.99, epsilon_start: float = 1.0, epsilon_end: float = 0.1,
                    epsilon_decay: float = 0.995, replay_buffer_size: int = 10000,
                    max_steps_per_episode: int = 500) -> List[float]:
    replay_buffer = deque(maxlen=replay_buffer_size)
    epsilon = epsilon_start
    all_rewards = []

    visited_states = set()
    for episode in trange(num_episodes):
        state, _ = env.reset()
        episode_reward = 0
        done = False

        for step in range(max_steps_per_episode):
            action = agent.select_action(state, epsilon)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            visited_states.add(tuple(state))
            replay_buffer.append((state, action, reward, next_state, done))
            state = next_state
            episode_reward += reward

        if len(replay_buffer) >= batch_size:

```



```

        batch = random.sample(replay_buffer, batch_size)
        agent.train(batch, gamma)
    if done:
        break

    all_rewards.append(episode_reward)
    epsilon = max(epsilon_end, epsilon * epsilon_decay)

    return all_rewards, visited_states

```

```

env = gym.make('Acrobot-v1')
dqn_rewards, dqn_visited_states = train_dqn_agent(env, dqn_agent, num_episodes=300)

```

```

0%|          | 0/300 [00:00<?, ?it/s]

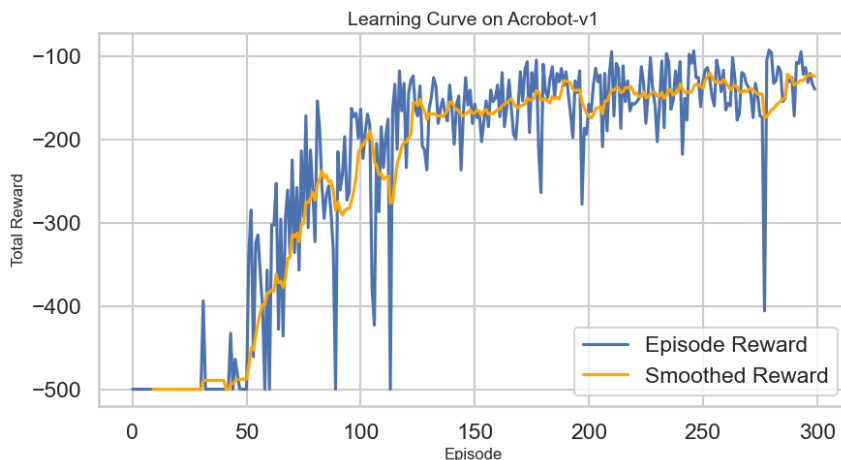
```

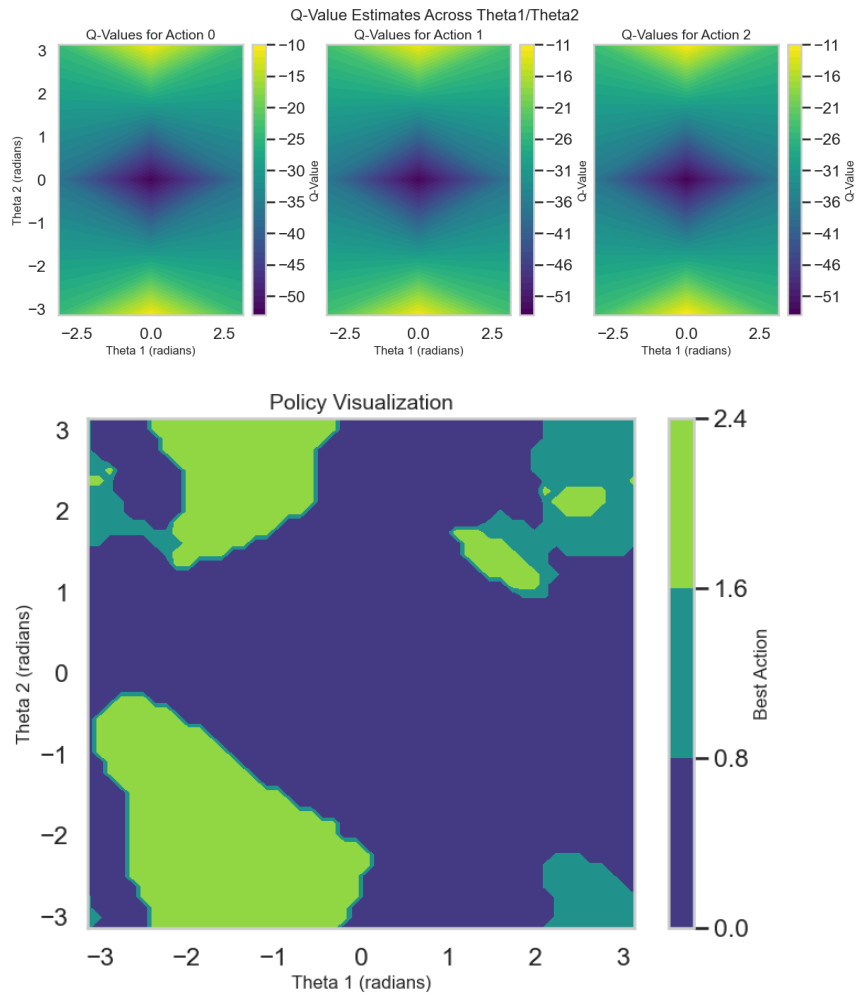
C:\Users\mafuge\AppData\Local\Temp\ipykernel_1116\1304865866.py:24: UserWarning: Creating a tensor from states_tensor = torch.FloatTensor(states).to(DEVICE)

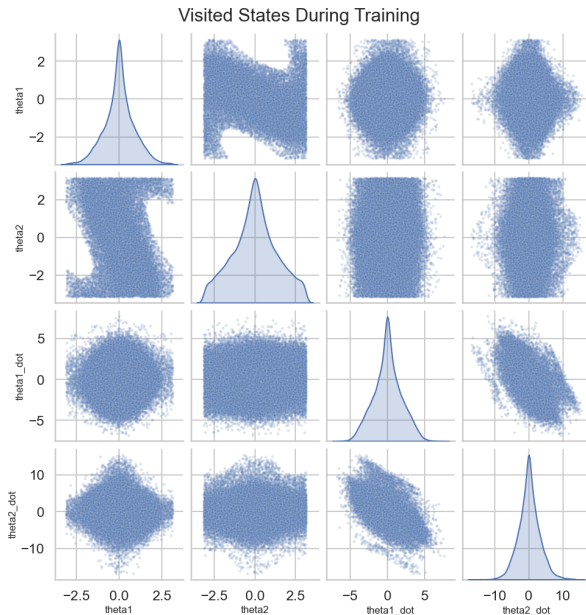
Total reward during evaluation: -500.0

Beyond just visualizing the performance on a sample environment, we can also inspect several aspects of the model training, including:

1. **Learning Curves:** Plotting the average return over episodes to see how quickly the agent learns.
2. **Q-Value Estimates:** Visualizing the predicted Q-values for different states and actions to assess how well the model is learning the value function.
3. **Policy Visualization:** Showing the selected actions as a function of space to understand the agent's learned behavior.







We can also interactively change the angular velocities to see how this updates the agent's q-value estimates and policy:

[Interactive output: available in the online version of this chapter.]

💡 Experiment: Effect the DQN architecture and training parameters on performance

The behavior of the DQN depends heavily on the architecture of the neural network used to approximate the Q-function, as well as various training parameters. Here are some aspects to experiment with so that you can get an intuition for how they affect learning:

- **Network Architecture:** Try changing the number of layers and neurons in the MLP. A deeper or wider network may capture more complex patterns but could also lead to overfitting or instability – what do you observe?
- **Discount Factor (γ):** Experiment with different values of the discount factor. A higher γ places more emphasis on future rewards, while a lower γ focuses more on immediate rewards. How does this affect the learned policy? Start with it set to 0.99 and then decrement to values like 0.95, 0.90, etc. and note how this affects the Learning Curve plot.
- **Exploration Rate (ϵ):** Adjust the exploration rate in the ϵ -greedy policy. A higher ϵ encourages more exploration, which can help discover better policies, but may also slow down learning. Try values like 0.1, 0.2, and 0.3 and see how quickly the agent learns. What if you set it to 0 (no exploration)? Could it improve learning to decay ϵ over time? If so, what possible downsides could you see to this?
- **Visited States versus Learned Policy:** After training, visualize the states vis-

ited during training versus the learned policy. Are there areas of the state space that the agent has not explored? Use the interactive visualization to look at the policy learned at this “out of distribution” set of states. What do you see?

20.3 Policy Gradients (REINFORCE)

In contrast to value-based methods like DQN, policy gradient methods directly optimize the policy $\pi_\theta(a|s)$ without relying on a value function. This has several advantages, including the ability to naturally handle continuous action spaces and stochastic policies. To do this, Policy Gradient methods optimize the expected return by adjusting the policy parameters θ in the direction that increases the likelihood of actions that lead to higher returns. It does this via the same Log-Derivative trick we learned about during Variational Inference, which allows us to express the gradient of a function under an Expectation (i.e., the expected return here) with respect to the policy parameters as an expectation over the policy itself:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a|s) R(\tau)],$$

where $R(\tau)$ is the total return from trajectory τ .

```
# Basic Policy Gradient (REINFORCE) Agent
class PolicyGradientAgent:
    """
    Basic REINFORCE policy gradient agent with optional entropy regularization.
    """
    def __init__(self, state_dim: int, action_dim: int, hidden_dim: int = 128,
                  lr: float = 1e-3, entropy_coef: float = 1e-3, normalize_rewards: bool = True):
        self.policy_network = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim),
            nn.Softmax(dim=-1)
        ).to(DEVICE)
        self.optimizer = torch.optim.Adam(self.policy_network.parameters(), lr=lr)
        self.entropy_coef = entropy_coef
        self.normalize_rewards = normalize_rewards

    def select_action(self, state: np.ndarray) -> int:
        """
        Sample an action from the policy for a single state.
        """
        state_tensor = torch.FloatTensor(state).unsqueeze(0).to(DEVICE)
```

```

    with torch.no_grad():
        action_probs = self.policy_network(state_tensor).squeeze(0).cpu().numpy()
        action = np.random.choice(len(action_probs), p=action_probs)
        return int(action)

def train(self, trajectories: List[Tuple[np.ndarray, int, float]], gamma: float):
    """
    Train the policy using a single episode trajectory list of (state, action, reward).
    Adds entropy regularization to encourage exploration.
    """
    states, actions, rewards = zip(*trajectories)

    # Compute discounted returns
    discounted_rewards = []
    G = 0.0
    for r in reversed(rewards):
        G = r + gamma * G
        discounted_rewards.insert(0, G)
    returns = torch.FloatTensor(discounted_rewards).to(DEVICE)

    # Optionally normalize returns (helps training stability)
    if self.normalize_rewards:
        returns = (returns - returns.mean()) / (returns.std(unbiased=False) + 1e-8)

    states_tensor = torch.FloatTensor(np.vstack(states)).to(DEVICE)
    actions_tensor = torch.LongTensor(actions).to(DEVICE)

    # Use torch.distributions for stable log-probs and entropy
    action_probs = self.policy_network(states_tensor)
    dist = torch.distributions.Categorical(probs=action_probs)
    log_probs = dist.log_prob(actions_tensor)
    entropy = dist.entropy()

    # Policy loss with entropy regularization (we maximize both returns and entropy)
    policy_loss = -torch.mean(log_probs * returns) - self.entropy_coef * torch.mean(entropy)

    self.optimizer.zero_grad()
    policy_loss.backward()
    self.optimizer.step()

# Instantiate and train the Policy Gradient agent similarly to the DQN agent above.
state_dim = env.observation_space.shape[0]

```

```

action_dim = env.action_space.n
pg_agent = PolicyGradientAgent(state_dim, action_dim, entropy_coef=1e-7)

```

```

# Training loop for Policy Gradient Agent

```

```

def train_pg_agent(env: gym.Env, agent: PolicyGradientAgent, num_episodes: int = 700, gamma: float = 0.99,
                  max_steps_per_episode: int = 500) -> List[float]:
    all_rewards = []
    visited_states = set()

    for episode in trange(num_episodes):
        state, _ = env.reset()
        episode_reward = 0
        done = False
        trajectory = []

        for step in range(max_steps_per_episode):
            action = agent.select_action(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            trajectory.append((state, action, reward))
            visited_states.add(tuple(state))
            state = next_state
            episode_reward += reward

        if done:
            break

        agent.train(trajectory, gamma)
        all_rewards.append(episode_reward)

    return all_rewards, visited_states

```

```

env = gym.make('Acrobot-v1')
pg_rewards, pg_visted_states = train_pg_agent(env, pg_agent, num_episodes=1000)

```

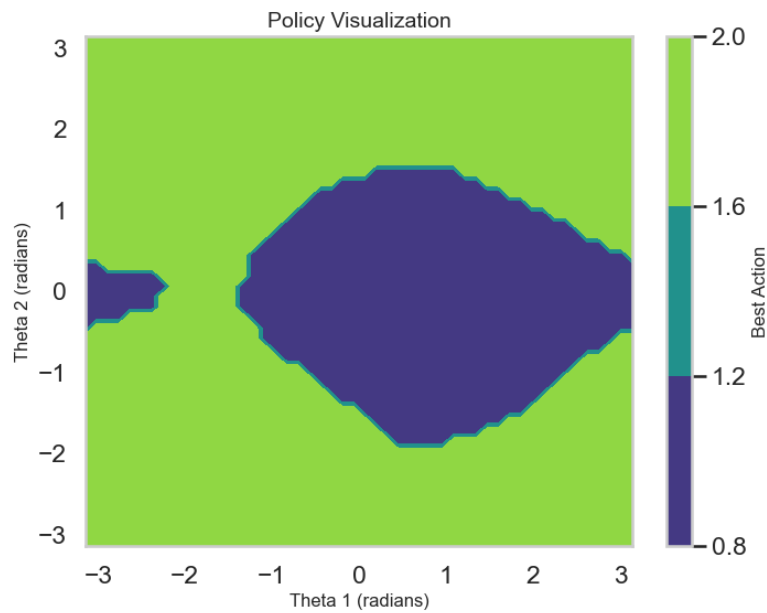
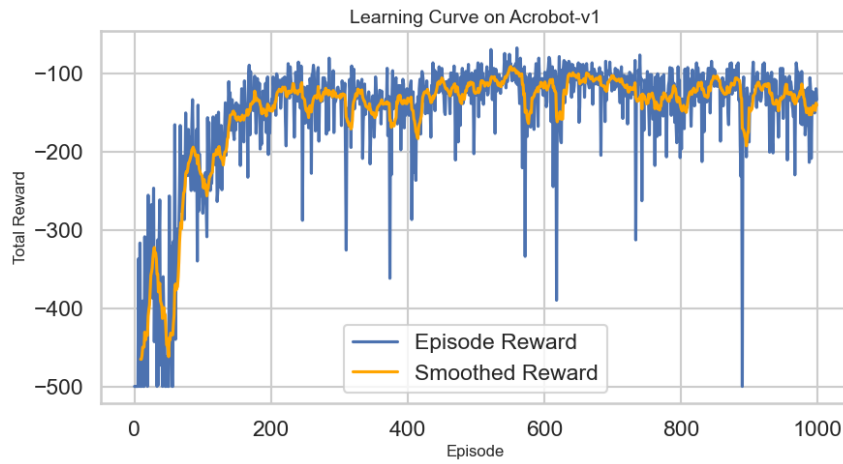
```

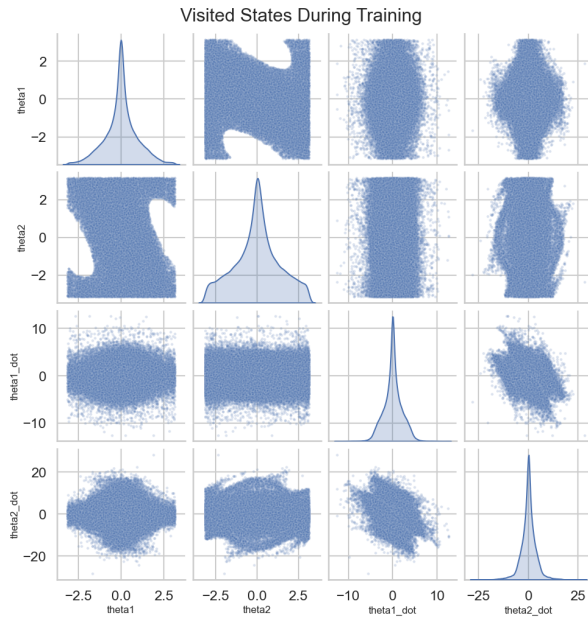
0%|          | 0/1000 [00:00<?, ?it/s]

```

Like with DQN, we can visualize: 1. **Learning Curves**: Tracking the average return to evaluate learning progress. 2. **Policy Distribution**: Visualizing the action distributions for given states to see how the policy evolves.

However, note that unlike DQN, REINFORCE does not learn a value function, so we won't have Q-value estimates to visualize, just the learned policy.





We can also interactively change the angular velocities like before to see how this updates the agent's policy:

[Interactive output: available in the online version of this chapter.]

💡 Experiment: Effect of PG architecture and training parameters on performance

While Policy Gradient methods do not rely on a value function, the architecture of the policy network and training parameters still play a crucial role in learning. Here are some aspects you can experiment with to gain some intuition:

- **Network Architecture:** As with before, you can try modifying the MLP. As you do this, what do you observe? Pay particular attention to the variability in the learning curve and also how it affects the overall policy visualization.
- **Entropy Regularization:** We added an entropy bonus to the policy loss to encourage exploration and hopefully prevent the policy from collapsing into a suboptimal mode. Experiment with different weights for this bonus. A higher weight encourages more exploration, while a lower weight focuses more on exploitation of the current policy. How does this affect learning and the learned policy? What happens if you set it to zero? What about a very high value? How does this compare to your experiments above with the DQN parameter that controlled exploration in its case (e.g., ϵ in the ϵ -greedy policy)? How does this affect the pairplot of the visited states during training?
- **Visited States versus Learned Policy:** As we did with the DQN, compare the learned policy using the interactive visualization with the states visited by the PG

Agent. What do you see, especially in comparison to the DQN agent and as you move outside of the visited states?

20.4 Vanilla Actor-Critic (AC)

While Policy Gradient methods like REINFORCE are powerful, they can suffer from high variance in their gradient estimates, which can slow down learning. This is because they rely solely on sampled returns to estimate the expected return, which can be noisy, especially in environments with high variability, long horizons, or sparse rewards. Moreover, in environments where rewards differ significantly across states and actions, the variance can be exacerbated, making it difficult for the policy to converge to an optimal solution.

To address these issues, researchers developed Actor-Critic methods, which combine the strengths of both value-based and policy-based approaches. The main idea behind Actor-Critic methods is to use a value function (the critic) to provide a more stable estimate of the expected return, which can then be used to guide the policy updates (the actor). This helps reduce the variance of the policy gradient estimates while still allowing for direct optimization of the policy. In this way, the actor policy only needs to learn the relative advantages of actions, rather than their absolute returns, which can be easier and more stable. This is referred to as learning the “advantage function,” which measures how much better an action is compared to the average action in a given state. By normalizing the returns in this way, Actor-Critic methods can in principle achieve lower variance in their gradient estimates, leading to more stable and efficient learning.

Specifically, Actor-Critic methods use two function approximators: one for the policy $\pi_\theta(a|s)$ (the actor) and another for the value function $V_\phi(s)$ (the critic). The Actor-Critic combines value-based bootstrapping (like in Value Iteration) with policy gradients. With TD(0), the critic target is

$$y_t = r_t + \gamma V_\phi(s_{t+1}).$$

The advantage estimate is $A_t = y_t - V_\phi(s_t)$ and the critic minimizes

$$\mathcal{L}_{\text{critic}} = (V_\phi(s_t) - y_t)^2.$$

The actor update uses $\nabla \log \pi_\theta(a_t|s_t) A_t$, which we can understand intuitively as increasing the log-probability of actions that lead to higher-than-expected returns (positive advantage) and decreasing it for actions that lead to lower-than-expected returns (negative advantage).

```
# Vanilla Actor-Critic Agent with Entropy Regularization
class ActorCriticAgent:
    """
    Simple actor-critic agent with separate policy and value networks.
```

Uses TD(0) bootstrapping for the critic and a policy gradient update for the actor. Adds entropy regularization to the policy loss to encourage exploration.

```
"""
def __init__(self, state_dim: int, action_dim: int, hidden_dim: int = 128,
              lr: float = 1e-3, entropy_coef: float = 1e-3):
    self.policy_network = nn.Sequential(
        nn.Linear(state_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, action_dim),
        nn.Softmax(dim=-1)
    ).to(DEVICE)

    self.value_network = nn.Sequential(
        nn.Linear(state_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, 1)
    ).to(DEVICE)

    self.policy_optimizer = torch.optim.Adam(self.policy_network.parameters(), lr=lr)
    # give the value network a slightly larger LR by default (optional)
    self.value_optimizer = torch.optim.Adam(self.value_network.parameters(), lr=lr * 3)

    self.entropy_coef = entropy_coef

def select_action(self, state: np.ndarray) -> int:
    """
    Sample an action from the policy for a single state (used during interaction).
    Returns:
        action (int): sampled discrete action
    """
    state_tensor = torch.FloatTensor(state).unsqueeze(0).to(DEVICE)
    with torch.no_grad():
        action_probs = self.policy_network(state_tensor).squeeze(0)
        dist = torch.distributions.Categorical(probs=action_probs)
        action = dist.sample().item()
    return int(action)

def train(self, trajectories: List[Tuple[np.ndarray, int, float, np.ndarray, bool]], gamma: float):
    """
    Perform one update of the actor and critic from a single episode trajectory.
    trajectories: list of (state, action, reward, next_state, done)
    """
```

```

"""
if len(trajectories) == 0:
    return

states, actions, rewards, next_states, dones = zip(*trajectories)

states_tensor = torch.FloatTensor(np.vstack(states)).to(DEVICE)
actions_tensor = torch.LongTensor(actions).to(DEVICE)
rewards_tensor = torch.FloatTensor(rewards).to(DEVICE)
next_states_tensor = torch.FloatTensor(np.vstack(next_states)).to(DEVICE)
dones_tensor = torch.FloatTensor(dones).to(DEVICE)

# Compute TD(0) targets for the critic (handle terminal states)
with torch.no_grad():
    next_state_values = self.value_network(next_states_tensor).squeeze(1)
    targets = rewards_tensor + gamma * next_state_values * (1.0 - dones_tensor)

# Update value network (critic)
values = self.value_network(states_tensor).squeeze(1)
value_loss = F.mse_loss(values, targets)

self.value_optimizer.zero_grad()
value_loss.backward()
self.value_optimizer.step()

# Update policy network (actor) with entropy regularization
action_probs = self.policy_network(states_tensor)
dist = torch.distributions.Categorical(probs=action_probs)
log_probs = dist.log_prob(actions_tensor)
entropy = dist.entropy()

# Advantage: TD error (target - baseline)
advantages = (targets - values).detach()

policy_loss = -torch.mean(log_probs * advantages) - self.entropy_coef * torch.mean(entropy)

self.policy_optimizer.zero_grad()
policy_loss.backward()
self.policy_optimizer.step()

```

```

# Training loop for Actor-Critic Agent
def train_ac_agent(env: gym.Env, agent: ActorCriticAgent, num_episodes: int = 700, gamma: float = 0.99,

```

```

        max_steps_per_episode: int = 500) -> List[float]:
    all_rewards = []
    visited_states = set()

    for episode in trange(num_episodes):
        state, _ = env.reset()
        episode_reward = 0
        done = False
        trajectory = []

        for step in range(max_steps_per_episode):
            action = agent.select_action(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            trajectory.append((state, action, reward, next_state, done))
            visited_states.add(tuple(state))
            state = next_state
            episode_reward += reward

        if done:
            break

        agent.train(trajectory, gamma)
        all_rewards.append(episode_reward)

    return all_rewards, visited_states

```

```

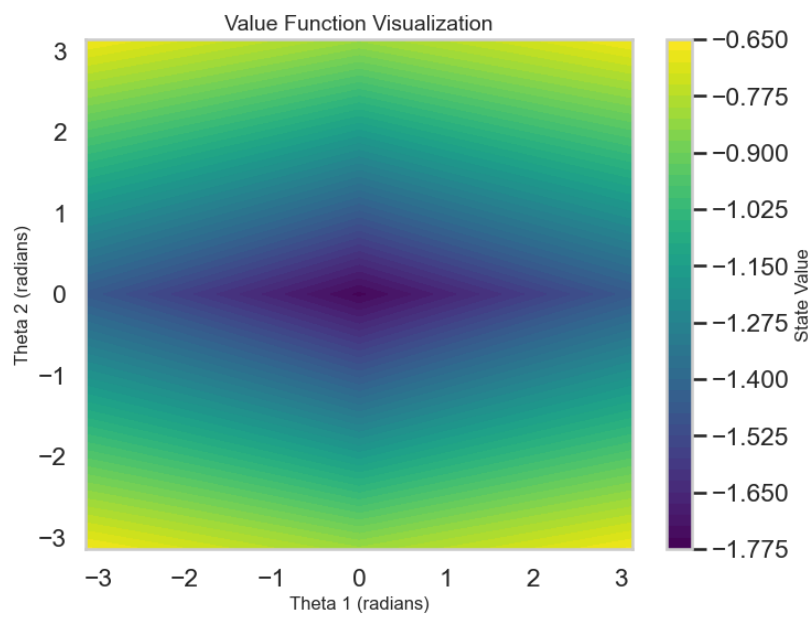
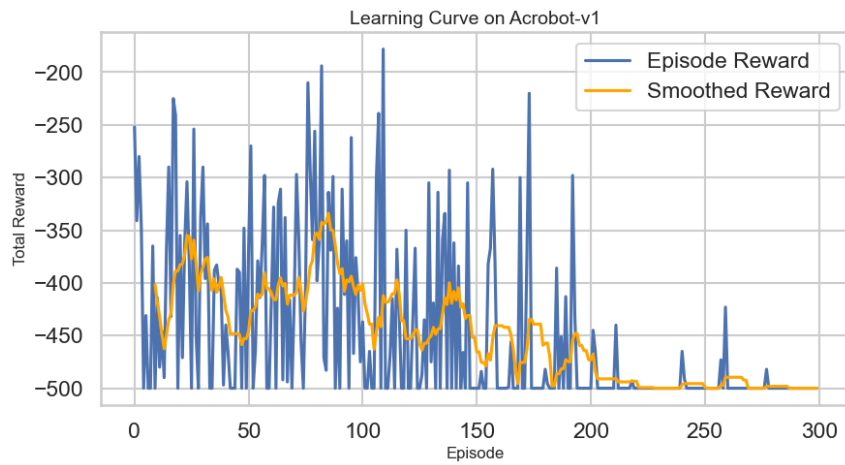
env = gym.make('Acrobot-v1')
ac_agent = ActorCriticAgent(state_dim, action_dim, lr=5e-5, entropy_coef=5e-3)
ac_rewards, ac_visited_states = train_ac_agent(env, ac_agent, num_episodes=300, max_steps_per_episode=1000)

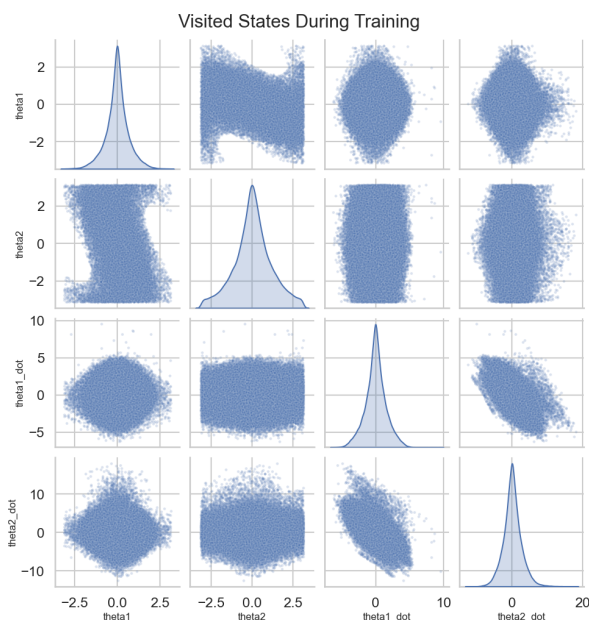
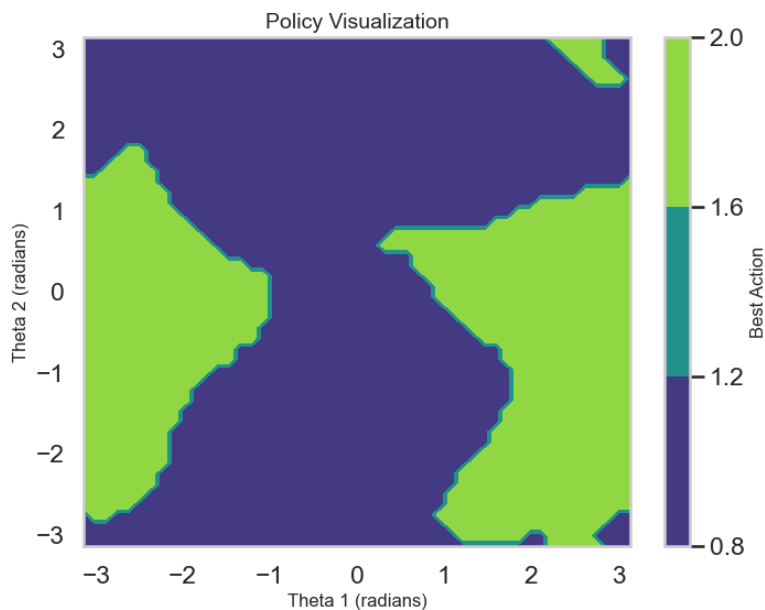
```

```

0%|          | 0/300 [00:00<?, ?it/s]

```





[Interactive output: available in the online version of this chapter.]

As we see above, while in principle Actor-Critic methods should have lower variance than pure policy gradient methods, in practice they can still suffer from instability due to the interaction between the actor and critic updates, much in the same way as we saw GANS struggle in an early chapter. For example, if the critic provides poor value estimates, it can lead to suboptimal policy updates, which in turn can degrade the quality of the value

estimates (since the critic thinks a state is bad when it is really just following a bad policy). To mitigate these issues, various techniques such as target networks, experience replay, and entropy regularization are often employed, but in their basic form, Actor-Critic methods can still be sensitive to hyperparameters and require careful tuning.

💡 Experiment: Effect of AC architecture and training parameters on performance

As mentioned above, the behavior of the Vanilla Actor-Critic can be difficult to tune due to the interaction between the actor and critic. Here are some aspects you can experiment with to gain some intuition:

- **Relative Strength of the Critic versus Actor Updates:** Try adjusting the learning rates for the actor and critic separately. A higher learning rate for the critic may lead to faster value function updates, but could also destabilize the policy updates if the value estimates become too noisy. Conversely, a higher learning rate for the actor may lead to more aggressive policy updates, but could also result in suboptimal policies if the value estimates are not accurate enough. Experiment with different combinations of learning rates and observe how they affect the learning curve and the learned policy. Typically, starting with a higher learning rate for the critic compared to the actor can help stabilize learning – you can compare this to your own observations.
- **Entropy Regularization:** As with the Policy Gradient method, we added an entropy bonus to the policy loss to encourage exploration and hopefully prevent the policy from collapsing into a suboptimal mode. Experiment with different weights for this bonus. A higher weight encourages more exploration, while a lower weight focuses more on exploitation of the current policy. How does this affect learning and the learned policy? What happens if you set it to zero? What about a very high value? How does this compare to your experiments above with the DQN and PG agents?
- **Learning Collapse:** Monitor the learning curves for signs of collapse or divergence. If you observe instability, try adjusting the learning rates, entropy regularization weight, or even the network architectures for the actor and critic. What combinations seem to work best for stabilizing learning in this environment? How sensitive is this Vanilla AC method to these hyperparameters compared to the DQN and PG methods you experimented with earlier?

20.5 Proximal Policy Optimization (PPO)

Building on the original idea of Actor-Critic methods, Proximal Policy Optimization (PPO), which (as of writing) is one of the most widely used Deep RL methods, introduces several enhancements to improve stability and performance. One of the key innovations in PPO is

the use of a clipped surrogate objective to limit the size of policy updates, preventing large, destabilizing changes to the policy. This is achieved by defining a ratio of the new policy to the old policy:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

and then clipping this ratio to be within a specified range (typically $[1 - \epsilon, 1 + \epsilon]$) when computing the policy loss. This prevents the policy from changing too much in a single update, which could lead to instability in the Vanilla AC case above. The PPO objective can be expressed as:

$$\mathcal{L}_{\text{PPO}}(\theta) = \mathbb{E}_t [\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)A_t)]$$

where A_t is the advantage estimate, similar to that used in Vanilla Actor-Critic methods. Breaking the above formula down, we see that if the policy update would lead to a large increase in the probability of taking action a_t (i.e., $r_t(\theta) > 1 + \epsilon$), the clipped term limits the contribution of that action to the loss, effectively preventing the policy from changing too much. Similarly, if the update would lead to a large decrease in the probability (i.e., $r_t(\theta) < 1 - \epsilon$), the clipping also limits the impact of that action on the loss. By taking the minimum of the unclipped and clipped objectives, PPO ensures that the policy update is conservative, leading to more stable learning.

The $r_t(\theta)A_t$ term encourages the policy to increase the probability of actions with positive advantage and decrease the probability of actions with negative advantage, similar to the Vanilla AC update, and the $\pi_{\theta_{\text{old}}}(a_t|s_t)$ term in $r_t(\theta)$ ensures that the updates are made relative to the previous policy, maintaining a connection to the past behavior of the agent.

```
# Proximal Policy Optimization (PPO) Agent
class PPOAgent:
    """
    Proximal Policy Optimization (PPO) agent with clipped surrogate objective.
    Uses separate policy and value networks.
    """
    def __init__(self, state_dim: int, action_dim: int, hidden_dim: int = 128,
                  lr: float = 3e-4, clip_epsilon: float = 0.2, entropy_coef: float = 1e-3):
        self.policy_network = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, action_dim),
            nn.Softmax(dim=-1)
        ).to(DEVICE)

        self.value_network = nn.Sequential(
```



```

        nn.Linear(state_dim, hidden_dim),
        nn.ReLU(),
        nn.Linear(hidden_dim, 1)
    ).to(DEVICE)

self.policy_optimizer = torch.optim.Adam(self.policy_network.parameters(), lr=lr)
self.value_optimizer = torch.optim.Adam(self.value_network.parameters(), lr=lr)

self.clip_epsilon = clip_epsilon
self.entropy_coef = entropy_coef
def select_action(self, state: np.ndarray) -> Tuple[int, float]:
    """ Sample an action from the policy for a single state.
    Returns:
        action (int): sampled discrete action
        action_prob (float): probability of the sampled action
    """
    state_tensor = torch.FloatTensor(state).unsqueeze(0).to(DEVICE)
    with torch.no_grad():
        action_probs = self.policy_network(state_tensor).squeeze(0)
    dist = torch.distributions.Categorical(probs=action_probs)
    action = dist.sample().item()
    action_prob = action_probs[action].item()
    return int(action)

def train(self, trajectories: List[Tuple[np.ndarray, int, float, np.ndarray, bool]], gamma: float,
       ppo_epochs: int = 4, batch_size: int = 64):
    """
    Perform multiple PPO updates from a single episode trajectory.
    trajectories: list of (state, action, reward, next_state, done)
    """
    if len(trajectories) == 0:
        return
    states, actions, rewards, next_states, dones = zip(*trajectories)

    states_tensor = torch.FloatTensor(np.vstack(states)).to(DEVICE)
    actions_tensor = torch.LongTensor(actions).to(DEVICE)
    rewards_tensor = torch.FloatTensor(rewards).to(DEVICE)
    next_states_tensor = torch.FloatTensor(np.vstack(next_states)).to(DEVICE)
    dones_tensor = torch.FloatTensor(dones).to(DEVICE)

    # Compute TD(0) targets for the critic
    with torch.no_grad():

```

```

        next_state_values = self.value_network(next_states_tensor).squeeze(1)
        targets = rewards_tensor + gamma * next_state_values * (1.0 - dones_tensor)
        values = self.value_network(states_tensor).squeeze(1)
        advantages = (targets - values).detach()

# Store old action probabilities
with torch.no_grad():
    action_probs = self.policy_network(states_tensor)
    dist = torch.distributions.Categorical(probs=action_probs)
    old_log_probs = dist.log_prob(actions_tensor)

dataset_size = len(trajectories)
for _ in range(ppo_epochs):
    for start in range(0, dataset_size, batch_size):
        end = start + batch_size
        batch_indices = slice(start, end)
        batch_states = states_tensor[batch_indices]
        batch_actions = actions_tensor[batch_indices]
        batch_advantages = advantages[batch_indices]
        batch_old_log_probs = old_log_probs[batch_indices]

        # Update value network (critic)
        batch_values = self.value_network(batch_states).squeeze(1)
        batch_targets = targets[batch_indices]
        value_loss = F.mse_loss(batch_values, batch_targets)
        self.value_optimizer.zero_grad()
        value_loss.backward()
        self.value_optimizer.step()

        # Update policy network (actor) with PPO clipped objective
        action_probs = self.policy_network(batch_states)
        dist = torch.distributions.Categorical(probs=action_probs)
        log_probs = dist.log_prob(batch_actions)
        entropy = dist.entropy()

        ratios = torch.exp(log_probs - batch_old_log_probs)
        surr1 = ratios * batch_advantages
        surr2 = torch.clamp(ratios, 1.0 - self.clip_epsilon, 1.0 + self.clip_epsilon) * batch_advantages

        policy_loss = -torch.mean(torch.min(surr1, surr2)) - self.entropy_coef * torch.mean(entropy)

        self.policy_optimizer.zero_grad()

```

```

        policy_loss.backward()
        self.policy_optimizer.step()

# Training loop for PPO Agent
def train_ppo_agent(env: gym.Env, agent: PPOAgent, num_episodes: int = 700, gamma: float = 0.99,
                    max_steps_per_episode: int = 500) -> List[float]:
    all_rewards = []
    visited_states = set()

    for episode in trange(num_episodes):
        state, _ = env.reset()
        episode_reward = 0
        done = False
        trajectory = []
        for step in range(max_steps_per_episode):
            action = agent.select_action(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated

            trajectory.append((state, action, reward, next_state, done))
            visited_states.add(tuple(state))
            state = next_state
            episode_reward += reward
            if done:
                break
        agent.train(trajectory, gamma)
        all_rewards.append(episode_reward)
    return all_rewards, visited_states

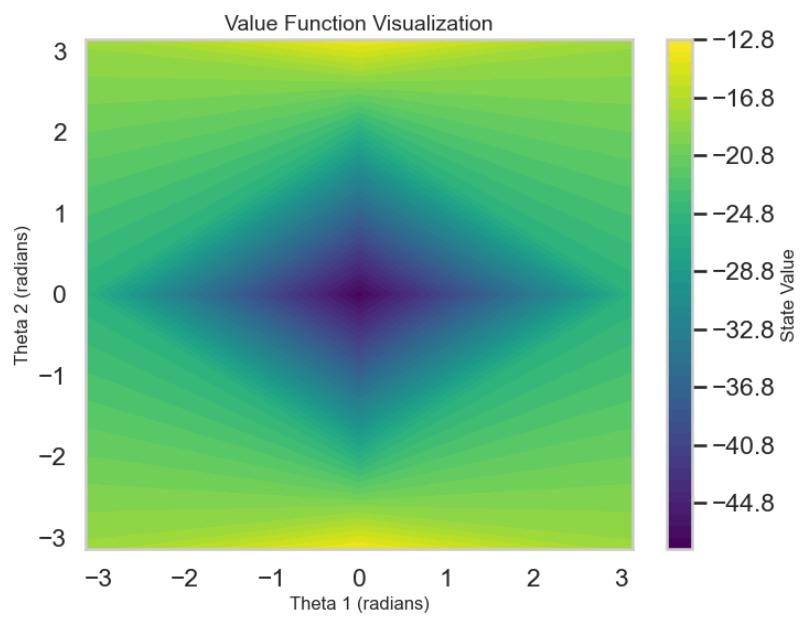
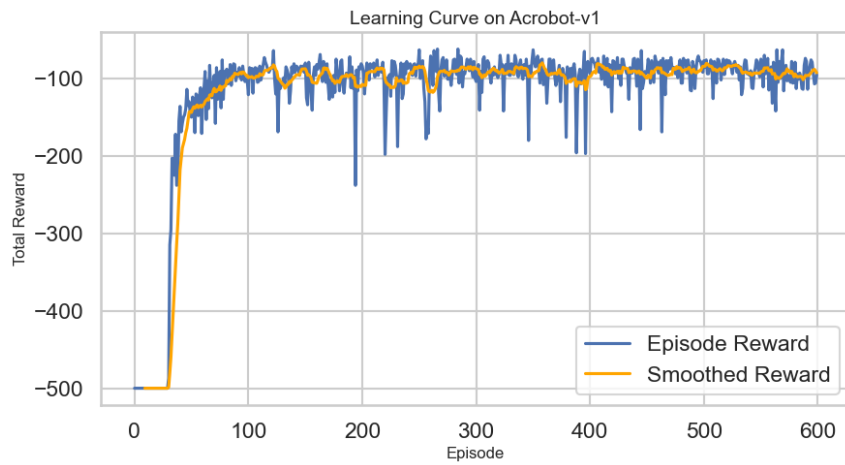
env = gym.make('Acrobot-v1')
ppo_agent = PPOAgent(state_dim, action_dim, lr=1e-3, clip_epsilon=0.2, entropy_coef=1e-3)
ppo_rewards, ppo_visited_states = train_ppo_agent(env, ppo_agent, num_episodes=600, max_steps_per_episode=500)

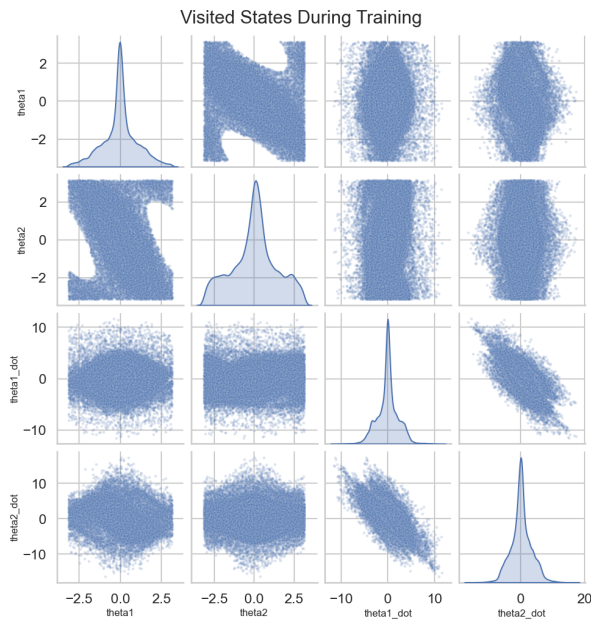
```

```

0%|          | 0/600 [00:00<?, ?it/s]

```





[Interactive output: available in the online version of this chapter.]

💡 Experiment: Effect of PPO architecture and training parameters on performance

In addition to the type of experiments we described above for the other agents, PPO has one new hyperparameter that you can experiment with to see how it affects learning:

- **Clipping Parameter (ϵ):** Experiment with different values of the clipping parameter ϵ . A smaller ϵ leads to more conservative updates, which can improve stability but may slow down learning. Conversely, a larger ϵ allows for more aggressive updates, which can speed up learning but may also lead to instability. Try values like 0.1, 0.2, and 0.3 and observe how they affect the learning curve and the learned policy.

20.6 Summary and Next Steps

We saw above two related paradigms for solving reinforcement learning problems: value-based methods (DQN) and policy-based methods (REINFORCE), plus a merger of the two approaches in Actor-Critic and PPO. Each has its own strengths and weaknesses, and the choice of which to use often depends on the specific characteristics of the problem at hand, such as the nature of the action space (discrete vs. continuous), the complexity of the environment, and the desired trade-off between exploration and exploitation. Specifically:

- **Value-Based Methods (DQN):** These methods are effective for discrete action spaces and can leverage the power of value functions to guide exploration. However, they can struggle with stability and convergence, especially in high-dimensional state spaces or continuous action spaces.
- **Policy-Based Methods (REINFORCE):** These methods can naturally handle continuous action spaces and stochastic policies, making them more flexible in certain environments. However, they can suffer from high variance in their gradient estimates, which can slow down learning.
- **Actor-Critic Methods:** By combining value-based and policy-based approaches, Actor-Critic methods can reduce variance in policy updates while still allowing for direct optimization of the policy. However, they can still be sensitive to hyperparameters and require careful tuning.
- **Proximal Policy Optimization (PPO):** PPO builds on Actor-Critic methods by introducing a clipped surrogate objective to limit the size of policy updates, improving stability and performance. It is widely used in practice due to its robustness and effectiveness across a variety of tasks. Like other policy-based methods, it can handle continuous action spaces and stochastic policies, which is one reason it is commonly used in Engineering and Robotics applications, where continuous control tasks are prevalent.

We could only cover this topic very briefly in this chapter, but there are many more advanced topics and techniques in reinforcement learning that are worth exploring. Some potential next steps include:

- **Exploring Advanced Algorithms:** Investigate more sophisticated RL algorithms such as Deep Deterministic Policy Gradient (DDPG), Twin Delayed DDPG (TD3), Soft Actor-

Critic (SAC), and others that are designed for continuous action spaces and offer improved performance and stability.

- **Hierarchical Reinforcement Learning:** Study methods that decompose complex tasks into simpler subtasks, allowing for more efficient learning and transfer of knowledge across tasks.
- **Multi-Agent Reinforcement Learning:** Explore scenarios where multiple agents interact within the same environment, leading to challenges such as cooperation, competition, and communication.
- **Model-Based Reinforcement Learning:** Investigate approaches that learn a model of the environment to improve sample efficiency and planning capabilities.

All of these are active areas of research and have significant implications for real-world applications of reinforcement learning within Engineering and beyond.

21 Introduction to Transformers

In prior chapters we explored various push-forward generative models as well as flow-based models. We have not yet addressed auto-regressive models, which are a third major class of generative models. Auto-regressive models are based on the idea of factorizing the joint distribution of data into a product of conditional distributions. There are many such models, including RNNs, LSTMs, and GRUs. However, in recent years, a new architecture called the *Transformer* has revolutionized the field of natural language processing (NLP) and has also been successfully applied to other domains such as computer vision and audio processing. This chapter introduces the Transformer architecture, its key components, and builds intuition behind how it works using simple 2D examples.

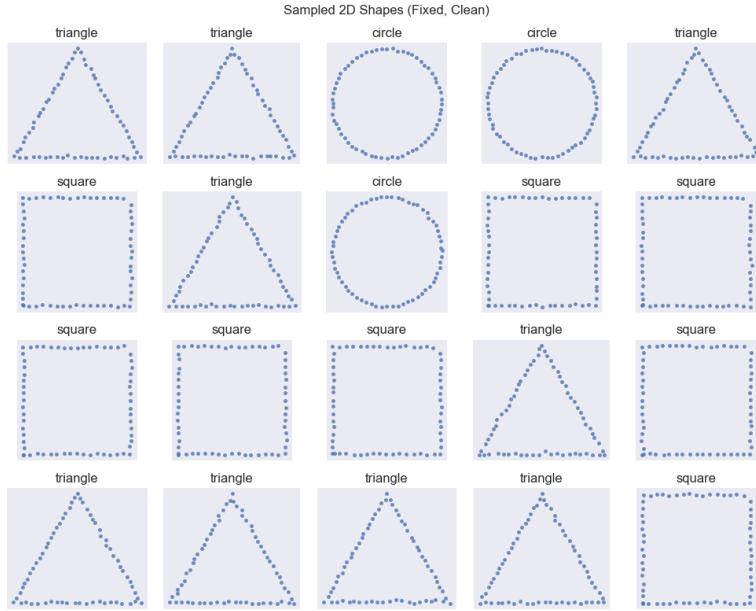
21.1 Learning Objectives

- Connect geometric intuition (shapes and point clouds) to transformer mechanics such as self-attention and global pooling.
- Compare MLPs, PointNet-style models, and transformers on fixed vs. variable set classification problems.
- Practice building PyTorch training loops, visualization helpers, and attention diagnostics that generalize across datasets.

[Interactive output: available in the online version of this chapter.]

21.2 Example: 2D Shapes — Fixed-Size Set Classification

To build intuition, we will start with a simple classification problem: given a set of 2D points sampled from a shape, classify which shape it is. We will start with the simplest model we could imagine for this task and then later show how this breaks down. Specifically, for a fixed-size input set of n points in 2D, we can flatten the input and feed it into a standard MLP classifier, as we show below. First, let's visualize some example data and labels:



As we can see, the dataset consists of points sampled from three different shapes: circles, squares, and triangles. Each shape is represented by a set of 2D points of a fixed length, and our goal is to classify the shape based on these points. For this, let's build a simple MLP classifier that takes the flattened set of points as input and outputs the predicted shape class:

```
class ShapeMLP(nn.Module):
    def __init__(self, input_dim: int, hidden_dims: Tuple[int, ...] = (256, 128, 64), num_classes: int = 3, dropout: float = 0.5):
        super().__init__()
        layers: List[nn.Module] = []
        dims = [input_dim, *hidden_dims]
        for in_dim, out_dim in zip(dims[:-1], dims[1:]):
            layers.append(nn.Linear(in_dim, out_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout))
        self.feature_extractor = nn.Sequential(*layers)
        self.classifier = nn.Linear(hidden_dims[-1], num_classes)

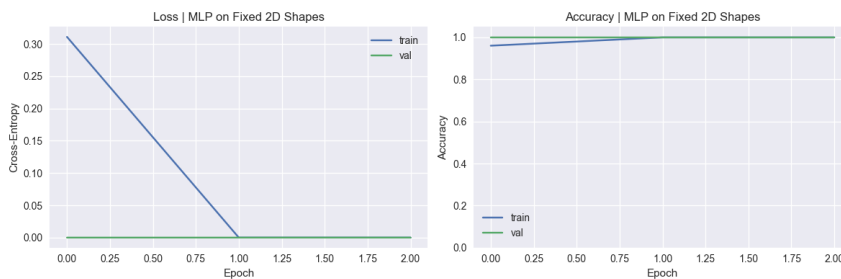
    def forward(self, x: torch.Tensor, return_features: bool = False):
        features = self.feature_extractor(x)
        logits = self.classifier(features)
        if return_features:
            return logits, features
        return logits
```

```
# Now train the MLP on the fixed-size shape dataset
mlp_fixed = ShapeMLP(input_dim=64 * 2, hidden_dims=(256, 128, 64), num_classes=len(SHAPE_TYPES))
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(mlp_fixed.parameters(), lr=1e-3, weight_decay=1e-4)

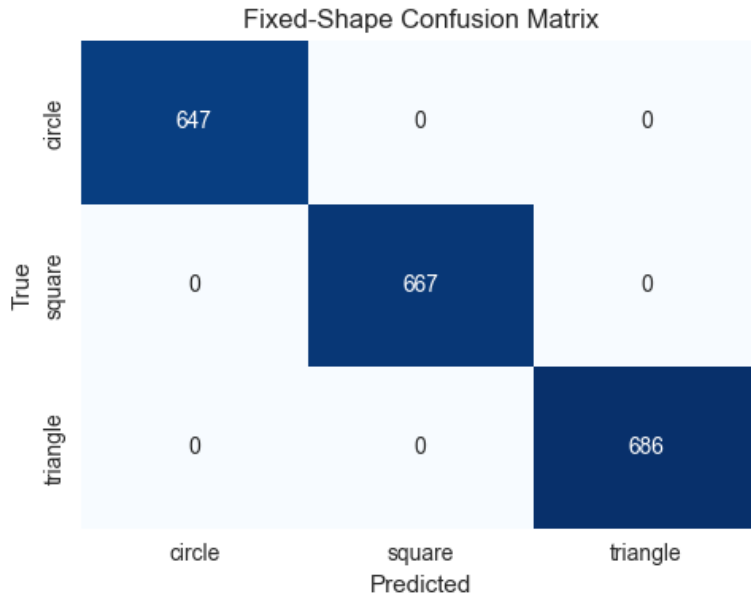
def mlp_forward(model, batch, device):
    return model(batch["inputs"].to(device))

history_fixed = train_classifier(
    mlp_fixed,
    train_loader_fixed,
    optimizer,
    criterion,
    epochs=3,
    forward_fn=mlp_forward,
    val_loader=val_loader_fixed,
)
```

```
Epoch 01/3 | train_loss=0.3110 train_acc=0.960 | val_loss=0.0000 val_acc=1.000
Epoch 02/3 | train_loss=0.0000 train_acc=1.000 | val_loss=0.0000 val_acc=1.000
Epoch 03/3 | train_loss=0.0000 train_acc=1.000 | val_loss=0.0000 val_acc=1.000
```



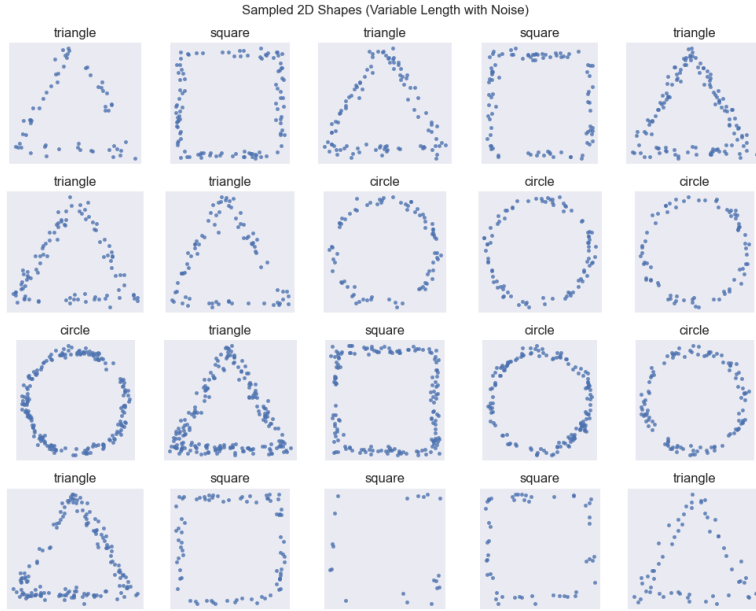
Validation accuracy: 100.00%



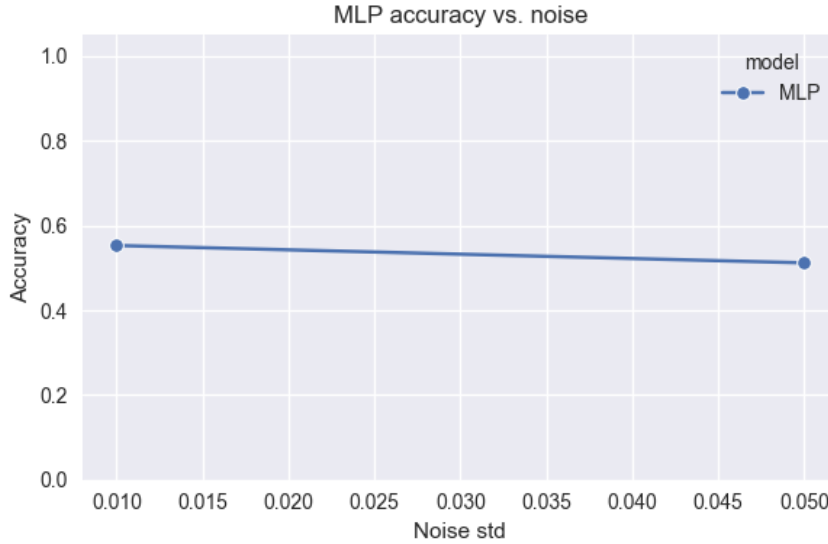
As we can see from the results above, the MLP does a great job on this particular task, achieving high accuracy on the test set. However, one reason for this is that every shape (1) had the same number of points and (2) the points were always presented in the same order. What happens to the MLP if we change these assumptions? Let's explore this next by permuting the order of the points in each set.

21.3 2D Shapes — Variable-Length Sets with Noise and Shuffling

In reality, we cannot always guarantee that our input sets will have a fixed size or that the points will be presented in a consistent order. For example, if the points we receive come from a 3D scanner or automated unstructured meshing algorithm. To simulate this on our toy example, we will modify our dataset by randomizing the number of points sampled from each shape, including the order, and adding some jitter to the point locations to simulate measurement noise. Let's see how the MLP does in this case. We keep the same MLP architecture but because we do not know the number of points in advance, we are now forced to pad every set to the maximum possible length so the linear layers see a fixed-size vector.



Epoch 01/10		train_loss=1.0766	train_acc=0.351		val_loss=1.0515	val_acc=0.373
Epoch 02/10		train_loss=0.9691	train_acc=0.520		val_loss=1.0408	val_acc=0.437
Epoch 03/10		train_loss=0.7116	train_acc=0.697		val_loss=1.0556	val_acc=0.502
Epoch 04/10		train_loss=0.3864	train_acc=0.847		val_loss=1.1267	val_acc=0.540
Epoch 05/10		train_loss=0.2029	train_acc=0.933		val_loss=1.3843	val_acc=0.553
Epoch 06/10		train_loss=0.1280	train_acc=0.955		val_loss=1.6696	val_acc=0.537
Epoch 07/10		train_loss=0.0800	train_acc=0.975		val_loss=1.7424	val_acc=0.551
Epoch 08/10		train_loss=0.0627	train_acc=0.978		val_loss=2.1464	val_acc=0.517
Epoch 09/10		train_loss=0.0548	train_acc=0.982		val_loss=2.1447	val_acc=0.551
Epoch 10/10		train_loss=0.0493	train_acc=0.983		val_loss=2.2817	val_acc=0.553
Noise 0.01		MLP accuracy: 55.3%				
Epoch 01/10		train_loss=1.0808	train_acc=0.358		val_loss=1.0528	val_acc=0.368
Epoch 02/10		train_loss=1.0001	train_acc=0.472		val_loss=1.0448	val_acc=0.436
Epoch 03/10		train_loss=0.7855	train_acc=0.669		val_loss=1.0365	val_acc=0.525
Epoch 04/10		train_loss=0.4514	train_acc=0.826		val_loss=1.1060	val_acc=0.547
Epoch 05/10		train_loss=0.2393	train_acc=0.914		val_loss=1.2745	val_acc=0.549
Epoch 06/10		train_loss=0.1336	train_acc=0.956		val_loss=1.6240	val_acc=0.543
Epoch 07/10		train_loss=0.1078	train_acc=0.960		val_loss=1.9251	val_acc=0.539
Epoch 08/10		train_loss=0.0826	train_acc=0.975		val_loss=2.0061	val_acc=0.523
Epoch 09/10		train_loss=0.0770	train_acc=0.974		val_loss=2.2888	val_acc=0.529
Epoch 10/10		train_loss=0.0660	train_acc=0.980		val_loss=2.3854	val_acc=0.512
Noise 0.05		MLP accuracy: 51.2%				



We see now that unlike before, the MLP’s performance degrades significantly as we introduce variability in the input sets. This is because the MLP relies on a consistent ordering and number of points to learn meaningful patterns, and when these assumptions are violated, it struggles to generalize. We can fix this by instead operating on *sets* of unordered points rather than flattened vectors where we assume that everything is consistently ordered. To enable this, we will use a transformer architecture that can handle variable-length sets and is invariant to the order of the points.

21.4 What Is a Transformer?

At a high level, a transformer is a **function on sets or sequences of vectors** that learns *how much each element should pay attention to every other element* when computing a new representation.

Given an input set (or sequence)

$$X = [x_1, x_2, \dots, x_n], \quad x_i \in \mathbb{R}^d,$$

a single **self-attention layer** produces new vectors z_i :

$$Z = [z_1, z_2, \dots, z_n], \quad z_i \in \mathbb{R}^{d_{\text{model}}}.$$

It involves three key steps:

1. **Linear projections** (learned):

$$q_i = W_Q x_i, \quad k_i = W_K x_i, \quad v_i = W_V x_i,$$

where W_Q, W_K, W_V are matrices.

2. **Attention weights** from pairwise similarity of queries and keys:

$$a_{ij} = \text{softmax}_j \left(\frac{q_i^\top k_j}{\sqrt{d_k}} \right),$$

which say how much point i listens to point j . The division by $\sqrt{d_k}$ is a scaling factor that helps with training stability since the dot products can grow large in magnitude as the dimension d_k increases, and thus push the softmax into regions with very small gradients, so the square root helps normalize the variance of the dot products. The softmax ensures the weights sum to 1 across j for each fixed i .¹

3. **Weighted aggregation of values:**

$$z_i = \sum_{j=1}^n a_{ij} v_j.$$

which produces the new representation for point i .

Typically, multiple attention heads are run in parallel (with different W_Q, W_K, W_V) and their outputs are concatenated and mixed with a small MLP. Stacking these layers produces a network that can attend globally to the whole set with content-dependent weights, unlike convolutions which can only attend across a short window.

For transformers in general, there is a rich set of possible variants and add-ons that one can include or study, but for this chapter we will keep it simple and try to build intuition using only unordered point sets. This means we will not be doing certain common operations in transformers like adding positional encoders or similar.

¹The softmax function is defined as follows: for a vector $z \in \mathbb{R}^n$, the softmax of z is a vector $\text{softmax}(z) \in \mathbb{R}^n$ where each component is given by

$$\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}.$$

21.4.1 MLPs vs. Transformers on Sets

It is useful to pause and contrast **how** the two architectures see the same data.

MLP view of a point set

For the variable-length experiment, we padded each set to a common maximum size $N_{\max}$ and then *flatten* all coordinates into a single long vector:

$$x = [p_{1,x}, p_{1,y}, \dots, p_{N_{\max},x}, p_{N_{\max},y}]^\top \in \mathbb{R}^{2N_{\max}}.$$

An L -layer MLP implements a composition of affine maps and pointwise nonlinearities

$$f_{\text{MLP}}(x) = W_L \sigma(W_{L-1} \sigma(\dots \sigma(W_1 x + b_1)) + b_{L-1}) + b_L.$$

Key properties of this representation:

- Each coordinate position (e.g., “entry 17”) has a **fixed semantic meaning** during training.
- If we change the **ordering or count** of points, the mapping from “position index” to “physical location in the shape” is destroyed.
- As a result, the MLP is **not permutation-invariant**: shuffling points changes the prediction in an uncontrolled way.

Transformer view of the same set

The transformer instead keeps the set as a matrix

$$X \in \mathbb{R}^{N_{\max} \times d}, \quad d = 2 \text{ for 2D}; X_i = p_i,$$

and produces a new matrix $Z \in \mathbb{R}^{N_{\max} \times d_{\text{model}}}$ by self-attention. Classification then uses either

- a special **CLS token** z_{CLS} , or
- a permutation-invariant pooling (mean or max) over rows of Z .

If we mask out padded rows and ignore them in attention, then:

- Reordering the **actual points** corresponds to permuting the rows of X .
- Self-attention plus pooling can be made **permutation-invariant** over the true points.

This means that the output of the transformer depends only on the **set** of points, not on their ordering, which is exactly what we need if we cannot rely on a fixed ordering or input sequence. The experiment below shows this numerically: once we break the ordering assumption the transformer will still be able to perform well, where the MLP degraded above.

21.4.2 Transformer Encoder for Sets

We now treat each point as a token, project (x, y) into a latent space, and apply stacked self-attention blocks with a learned CLS token for global pooling. Padding masks ensure attention ignores fake points. Before we jump into the full transformer encoder, let's first build and train a **tiny single-head self-attention model** on top of simple 2D shapes. The goal is not to get state-of-the-art accuracy, but to have a model whose **entire internal state fits in 2 dimensions and one head** so that we can literally plot:

- the **linear projections** W_Q, W_K, W_V applied to each point,
- the resulting **attention weights** a_{ij} across the set,
- the **aggregated features** $z_i = \sum_j a_{ij} v_j$.

We will first train a very small transformer on a toy classification task (only a few 2D shapes), and then visualize what it has learned.

```
class MiniSelfAttention(nn.Module):
    """Single-head self-attention with d_model = 2 for visualization.

    This block omits residual connections and feedforward layers on purpose so
    that you can directly see the impact of Q, K, V and the attention
    matrix on a set of 2D points.
    """

    def __init__(self, input_dim: int = 2, d_model: int = 2):
        super().__init__()
        self.d_model = d_model
        self.W_q = nn.Linear(input_dim, d_model, bias=False)
        self.W_k = nn.Linear(input_dim, d_model, bias=False)
        self.W_v = nn.Linear(input_dim, d_model, bias=False)

    def forward(self, x: torch.Tensor) -> Dict[str, torch.Tensor]:
        """Compute Q, K, V, attention weights, and output features.

        Parameters
        -----
        x : (N, 2) tensor
```



```

        Input 2D points for a single shape.
        """

        # Compute projections
        Q = self.W_q(x) # (N, d_model)
        K = self.W_k(x) # (N, d_model)
        V = self.W_v(x) # (N, d_model)

        # Scaled dot-product attention logits and weights
        scale = math.sqrt(self.d_model)
        logits = Q @ K.t() / scale # (N, N)
        A = torch.softmax(logits, dim=-1) # (N, N)

        # Aggregate values
        Z = A @ V # (N, d_model)

        return {"Q": Q, "K": K, "V": V, "A": A, "Z": Z}

class SelfAttentionBlock(nn.Module):
    """Batched single-head self-attention block plus optional feedforward.

    This block provides the interface expected by ToySetTransformer:
    - forward(x, key_padding_mask=None) -> (h, attn)
      where x: (B, N, d_model), h: (B, N, d_model), attn: (B, N, N)
    """

    def __init__(self, d_model: int = 2, n_heads: int = 1, dim_feedforward: int = 16, dropout: float = 0.0):
        super().__init__()
        self.d_model = d_model

        # Projections operate on d_model inputs (assumes input already projected to d_model)
        self.W_q = nn.Linear(d_model, d_model, bias=False)
        self.W_k = nn.Linear(d_model, d_model, bias=False)
        self.W_v = nn.Linear(d_model, d_model, bias=False)

        # small position-wise feedforward for demonstration
        self.ff = nn.Sequential(
            nn.Linear(d_model, dim_feedforward),
            nn.ReLU(),
            nn.Linear(dim_feedforward, d_model),
            nn.Dropout(dropout),
        )

```

```

def forward(self, x: torch.Tensor, key_padding_mask: Optional[torch.Tensor] = None):
    """Forward pass for batched inputs.

    Parameters
    -----
    x : (B, N, d_model)
    key_padding_mask : (B, N) boolean mask where True indicates padding positions to ignore
    """
    # Projections
    Q = self.W_q(x) # (B, N, d)
    K = self.W_k(x) # (B, N, d)
    V = self.W_v(x) # (B, N, d)

    # Scaled dot-product attention logits and weights
    scale = math.sqrt(self.d_model)
    logits = torch.matmul(Q, K.transpose(-2, -1)) / scale # (B, N, N)

    if key_padding_mask is not None:
        # key_padding_mask: True for padded tokens -> we want to set logits to -inf
        # Expand to (B, N, N) to mask keys for each query
        mask = key_padding_mask.unsqueeze(1).expand(-1, logits.size(1), -1) # (B, N, N)
        logits = logits.masked_fill(mask, float("-inf"))

    A = torch.softmax(logits, dim=-1) # (B, N, N)

    # Aggregate values
    Z = torch.matmul(A, V) # (B, N, d)

    # Apply a simple feedforward (keeps dimensionality)
    h = Z + self.ff(Z) # (B, N, d) -- include residual-like skip for stability

    return h, A

class ToySetTransformer(nn.Module):
    """Tiny transformer on 2D point sets for interpretability.

    - Single head
    - d_model = 2 (so we can plot everything directly)
    - One encoder block
    - Mean pooling over points for classification
    """

```

```

def __init__(self, num_classes: int = 3, d_model: int = 2):
    super().__init__()
    self.input_proj = nn.Linear(2, d_model)
    self.attn_block = SelfAttentionBlock(d_model=d_model, n_heads=1, dim_feedforward=16, dropout=0.0)
    self.classifier = nn.Sequential(
        nn.LayerNorm(d_model),
        nn.Linear(d_model, num_classes),
    )

def forward(self, x: torch.Tensor, mask: Optional[torch.Tensor] = None, return_internals: bool = False)
    # x: (B, N, 2)
    h = self.input_proj(x)
    h, attn = self.attn_block(h, key_padding_mask=mask)

    if mask is not None:
        valid = (~mask).unsqueeze(-1) # (B, N, 1)
        summed = (h * valid).sum(dim=1)
        counts = valid.sum(dim=1).clamp_min(1.0)
        pooled = summed / counts
    else:
        pooled = h.mean(dim=1)

    logits = self.classifier(pooled)

    if return_internals:
        return logits, {"h": h, "attn": attn}
    return logits

```

```

toy_model = ToySetTransformer(num_classes=len(SHAPE_TYPES), d_model=3)
toy_optimizer = torch.optim.AdamW(toy_model.parameters(), lr=5e-3, weight_decay=1e-4)

def toy_forward(model, batch, device):
    return model(batch["points"].to(device), mask=batch["mask"].to(device))

toy_history = train_classifier(
    toy_model,
    toy_train_loader,
    toy_optimizer,
    criterion,
    epochs=150,
    forward_fn=toy_forward,
    val_loader=toy_val_loader,

```

)

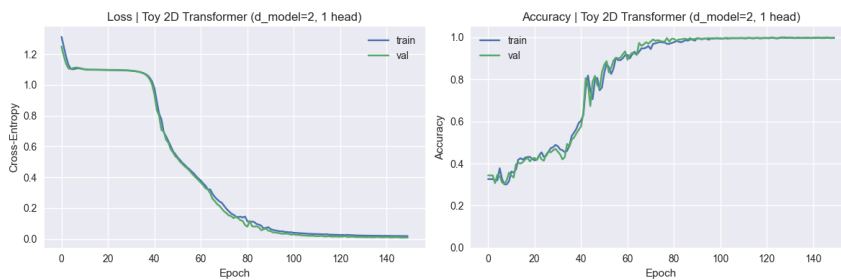
Epoch 01/150		train_loss=1.3112	train_acc=0.326		val_loss=1.2489	val_acc=0.343
Epoch 02/150		train_loss=1.2458	train_acc=0.326		val_loss=1.1850	val_acc=0.343
Epoch 03/150		train_loss=1.1798	train_acc=0.326		val_loss=1.1349	val_acc=0.343
Epoch 04/150		train_loss=1.1286	train_acc=0.320		val_loss=1.1066	val_acc=0.307
Epoch 05/150		train_loss=1.1043	train_acc=0.320		val_loss=1.1019	val_acc=0.347
Epoch 06/150		train_loss=1.1010	train_acc=0.378		val_loss=1.1088	val_acc=0.340
Epoch 07/150		train_loss=1.1047	train_acc=0.327		val_loss=1.1124	val_acc=0.310
Epoch 08/150		train_loss=1.1073	train_acc=0.301		val_loss=1.1116	val_acc=0.303
Epoch 09/150		train_loss=1.1061	train_acc=0.301		val_loss=1.1079	val_acc=0.320
Epoch 10/150		train_loss=1.1028	train_acc=0.316		val_loss=1.1040	val_acc=0.357
Epoch 11/150		train_loss=1.1002	train_acc=0.362		val_loss=1.1011	val_acc=0.343
Epoch 12/150		train_loss=1.0981	train_acc=0.357		val_loss=1.0994	val_acc=0.333
Epoch 13/150		train_loss=1.0977	train_acc=0.372		val_loss=1.0986	val_acc=0.397
Epoch 14/150		train_loss=1.0978	train_acc=0.418		val_loss=1.0984	val_acc=0.403
Epoch 15/150		train_loss=1.0978	train_acc=0.425		val_loss=1.0982	val_acc=0.400
Epoch 16/150		train_loss=1.0977	train_acc=0.418		val_loss=1.0982	val_acc=0.407
Epoch 17/150		train_loss=1.0975	train_acc=0.428		val_loss=1.0980	val_acc=0.420
Epoch 18/150		train_loss=1.0972	train_acc=0.431		val_loss=1.0978	val_acc=0.427
Epoch 19/150		train_loss=1.0970	train_acc=0.432		val_loss=1.0975	val_acc=0.410
Epoch 20/150		train_loss=1.0968	train_acc=0.422		val_loss=1.0972	val_acc=0.423
Epoch 21/150		train_loss=1.0966	train_acc=0.415		val_loss=1.0970	val_acc=0.427
Epoch 22/150		train_loss=1.0963	train_acc=0.418		val_loss=1.0969	val_acc=0.417
Epoch 23/150		train_loss=1.0960	train_acc=0.439		val_loss=1.0968	val_acc=0.440
Epoch 24/150		train_loss=1.0958	train_acc=0.453		val_loss=1.0970	val_acc=0.420
Epoch 25/150		train_loss=1.0954	train_acc=0.432		val_loss=1.0969	val_acc=0.413
Epoch 26/150		train_loss=1.0950	train_acc=0.437		val_loss=1.0963	val_acc=0.443
Epoch 27/150		train_loss=1.0945	train_acc=0.456		val_loss=1.0955	val_acc=0.457
Epoch 28/150		train_loss=1.0939	train_acc=0.473		val_loss=1.0950	val_acc=0.453
Epoch 29/150		train_loss=1.0933	train_acc=0.477		val_loss=1.0944	val_acc=0.463
Epoch 30/150		train_loss=1.0925	train_acc=0.488		val_loss=1.0936	val_acc=0.470
Epoch 31/150		train_loss=1.0915	train_acc=0.482		val_loss=1.0924	val_acc=0.457
Epoch 32/150		train_loss=1.0900	train_acc=0.468		val_loss=1.0905	val_acc=0.443
Epoch 33/150		train_loss=1.0880	train_acc=0.463		val_loss=1.0881	val_acc=0.420
Epoch 34/150		train_loss=1.0855	train_acc=0.454		val_loss=1.0852	val_acc=0.430
Epoch 35/150		train_loss=1.0826	train_acc=0.460		val_loss=1.0811	val_acc=0.493
Epoch 36/150		train_loss=1.0776	train_acc=0.486		val_loss=1.0760	val_acc=0.480
Epoch 37/150		train_loss=1.0709	train_acc=0.531		val_loss=1.0687	val_acc=0.513
Epoch 38/150		train_loss=1.0614	train_acc=0.547		val_loss=1.0557	val_acc=0.523
Epoch 39/150		train_loss=1.0455	train_acc=0.565		val_loss=1.0351	val_acc=0.543
Epoch 40/150		train_loss=1.0209	train_acc=0.593		val_loss=1.0004	val_acc=0.560

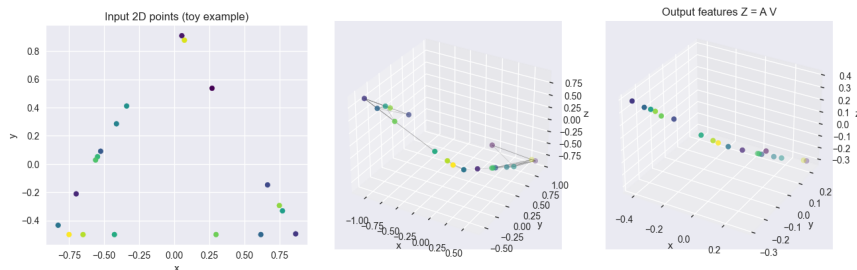
Epoch 41/150		train_loss=0.9766	train_acc=0.604		val_loss=0.9355	val_acc=0.577
Epoch 42/150		train_loss=0.8936	train_acc=0.630		val_loss=0.8360	val_acc=0.640
Epoch 43/150		train_loss=0.8132	train_acc=0.755		val_loss=0.7998	val_acc=0.807
Epoch 44/150		train_loss=0.7757	train_acc=0.818		val_loss=0.7071	val_acc=0.780
Epoch 45/150		train_loss=0.6926	train_acc=0.749		val_loss=0.6972	val_acc=0.673
Epoch 46/150		train_loss=0.6656	train_acc=0.706		val_loss=0.6436	val_acc=0.790
Epoch 47/150		train_loss=0.6341	train_acc=0.797		val_loss=0.6239	val_acc=0.817
Epoch 48/150		train_loss=0.6027	train_acc=0.807		val_loss=0.5822	val_acc=0.773
Epoch 49/150		train_loss=0.5688	train_acc=0.748		val_loss=0.5610	val_acc=0.757
Epoch 50/150		train_loss=0.5496	train_acc=0.762		val_loss=0.5360	val_acc=0.833
Epoch 51/150		train_loss=0.5287	train_acc=0.832		val_loss=0.5219	val_acc=0.870
Epoch 52/150		train_loss=0.5126	train_acc=0.875		val_loss=0.5013	val_acc=0.887
Epoch 53/150		train_loss=0.4939	train_acc=0.858		val_loss=0.4858	val_acc=0.833
Epoch 54/150		train_loss=0.4808	train_acc=0.826		val_loss=0.4709	val_acc=0.857
Epoch 55/150		train_loss=0.4649	train_acc=0.853		val_loss=0.4587	val_acc=0.890
Epoch 56/150		train_loss=0.4527	train_acc=0.902		val_loss=0.4444	val_acc=0.903
Epoch 57/150		train_loss=0.4379	train_acc=0.892		val_loss=0.4271	val_acc=0.900
Epoch 58/150		train_loss=0.4242	train_acc=0.892		val_loss=0.4099	val_acc=0.903
Epoch 59/150		train_loss=0.4059	train_acc=0.901		val_loss=0.3958	val_acc=0.920
Epoch 60/150		train_loss=0.3911	train_acc=0.916		val_loss=0.3803	val_acc=0.933
Epoch 61/150		train_loss=0.3779	train_acc=0.917		val_loss=0.3648	val_acc=0.893
Epoch 62/150		train_loss=0.3595	train_acc=0.900		val_loss=0.3481	val_acc=0.913
Epoch 63/150		train_loss=0.3449	train_acc=0.918		val_loss=0.3281	val_acc=0.927
Epoch 64/150		train_loss=0.3207	train_acc=0.932		val_loss=0.3212	val_acc=0.923
Epoch 65/150		train_loss=0.3209	train_acc=0.918		val_loss=0.2965	val_acc=0.917
Epoch 66/150		train_loss=0.2924	train_acc=0.932		val_loss=0.2605	val_acc=0.973
Epoch 67/150		train_loss=0.2715	train_acc=0.948		val_loss=0.2448	val_acc=0.960
Epoch 68/150		train_loss=0.2565	train_acc=0.948		val_loss=0.2255	val_acc=0.967
Epoch 69/150		train_loss=0.2431	train_acc=0.951		val_loss=0.2139	val_acc=0.977
Epoch 70/150		train_loss=0.2340	train_acc=0.959		val_loss=0.1966	val_acc=0.970
Epoch 71/150		train_loss=0.2145	train_acc=0.944		val_loss=0.1787	val_acc=0.980
Epoch 72/150		train_loss=0.1949	train_acc=0.969		val_loss=0.1670	val_acc=0.990
Epoch 73/150		train_loss=0.1801	train_acc=0.972		val_loss=0.1521	val_acc=0.983
Epoch 74/150		train_loss=0.1649	train_acc=0.975		val_loss=0.1483	val_acc=0.987
Epoch 75/150		train_loss=0.1571	train_acc=0.978		val_loss=0.1348	val_acc=0.983
Epoch 76/150		train_loss=0.1455	train_acc=0.976		val_loss=0.1436	val_acc=0.980
Epoch 77/150		train_loss=0.1411	train_acc=0.976		val_loss=0.1343	val_acc=0.980
Epoch 78/150		train_loss=0.1443	train_acc=0.969		val_loss=0.1215	val_acc=0.997
Epoch 79/150		train_loss=0.1389	train_acc=0.969		val_loss=0.1164	val_acc=0.980
Epoch 80/150		train_loss=0.1449	train_acc=0.973		val_loss=0.0899	val_acc=0.993
Epoch 81/150		train_loss=0.1144	train_acc=0.977		val_loss=0.0793	val_acc=0.993
Epoch 82/150		train_loss=0.1088	train_acc=0.983		val_loss=0.1161	val_acc=0.983
Epoch 83/150		train_loss=0.1128	train_acc=0.980		val_loss=0.0826	val_acc=0.990

Epoch 84/150		train_loss=0.1091	train_acc=0.977		val_loss=0.0797	val_acc=0.990
Epoch 85/150		train_loss=0.0962	train_acc=0.981		val_loss=0.0816	val_acc=0.993
Epoch 86/150		train_loss=0.0929	train_acc=0.987		val_loss=0.0745	val_acc=0.993
Epoch 87/150		train_loss=0.0851	train_acc=0.984		val_loss=0.0562	val_acc=0.990
Epoch 88/150		train_loss=0.0676	train_acc=0.990		val_loss=0.0629	val_acc=0.997
Epoch 89/150		train_loss=0.0721	train_acc=0.993		val_loss=0.0669	val_acc=0.987
Epoch 90/150		train_loss=0.0760	train_acc=0.984		val_loss=0.0537	val_acc=0.993
Epoch 91/150		train_loss=0.0638	train_acc=0.996		val_loss=0.0506	val_acc=0.993
Epoch 92/150		train_loss=0.0584	train_acc=0.994		val_loss=0.0418	val_acc=0.993
Epoch 93/150		train_loss=0.0549	train_acc=0.991		val_loss=0.0418	val_acc=0.993
Epoch 94/150		train_loss=0.0525	train_acc=0.994		val_loss=0.0390	val_acc=0.993
Epoch 95/150		train_loss=0.0489	train_acc=0.994		val_loss=0.0365	val_acc=0.993
Epoch 96/150		train_loss=0.0493	train_acc=0.989		val_loss=0.0332	val_acc=0.997
Epoch 97/150		train_loss=0.0457	train_acc=0.995		val_loss=0.0338	val_acc=0.993
Epoch 98/150		train_loss=0.0442	train_acc=0.995		val_loss=0.0345	val_acc=0.993
Epoch 99/150		train_loss=0.0431	train_acc=0.995		val_loss=0.0315	val_acc=0.997
Epoch 100/150		train_loss=0.0429	train_acc=0.995		val_loss=0.0276	val_acc=0.997
Epoch 101/150		train_loss=0.0394	train_acc=0.995		val_loss=0.0295	val_acc=0.993
Epoch 102/150		train_loss=0.0397	train_acc=0.993		val_loss=0.0262	val_acc=0.993
Epoch 103/150		train_loss=0.0371	train_acc=0.997		val_loss=0.0263	val_acc=0.997
Epoch 104/150		train_loss=0.0368	train_acc=0.997		val_loss=0.0242	val_acc=0.993
Epoch 105/150		train_loss=0.0356	train_acc=0.995		val_loss=0.0225	val_acc=0.997
Epoch 106/150		train_loss=0.0350	train_acc=0.996		val_loss=0.0224	val_acc=0.997
Epoch 107/150		train_loss=0.0335	train_acc=0.996		val_loss=0.0209	val_acc=0.997
Epoch 108/150		train_loss=0.0328	train_acc=0.996		val_loss=0.0201	val_acc=0.997
Epoch 109/150		train_loss=0.0320	train_acc=0.996		val_loss=0.0208	val_acc=0.997
Epoch 110/150		train_loss=0.0316	train_acc=0.997		val_loss=0.0192	val_acc=0.997
Epoch 111/150		train_loss=0.0313	train_acc=0.996		val_loss=0.0175	val_acc=0.997
Epoch 112/150		train_loss=0.0309	train_acc=0.995		val_loss=0.0183	val_acc=0.997
Epoch 113/150		train_loss=0.0299	train_acc=0.998		val_loss=0.0201	val_acc=0.997
Epoch 114/150		train_loss=0.0296	train_acc=0.997		val_loss=0.0161	val_acc=0.997
Epoch 115/150		train_loss=0.0303	train_acc=0.995		val_loss=0.0162	val_acc=0.997
Epoch 116/150		train_loss=0.0282	train_acc=0.997		val_loss=0.0178	val_acc=0.997
Epoch 117/150		train_loss=0.0273	train_acc=0.998		val_loss=0.0159	val_acc=0.997
Epoch 118/150		train_loss=0.0275	train_acc=0.995		val_loss=0.0147	val_acc=0.997
Epoch 119/150		train_loss=0.0260	train_acc=0.996		val_loss=0.0158	val_acc=0.997
Epoch 120/150		train_loss=0.0264	train_acc=0.998		val_loss=0.0146	val_acc=0.997
Epoch 121/150		train_loss=0.0250	train_acc=0.997		val_loss=0.0144	val_acc=0.997
Epoch 122/150		train_loss=0.0261	train_acc=0.996		val_loss=0.0150	val_acc=0.997
Epoch 123/150		train_loss=0.0242	train_acc=0.998		val_loss=0.0179	val_acc=1.000
Epoch 124/150		train_loss=0.0262	train_acc=0.998		val_loss=0.0129	val_acc=0.997
Epoch 125/150		train_loss=0.0249	train_acc=0.995		val_loss=0.0133	val_acc=0.997
Epoch 126/150		train_loss=0.0245	train_acc=0.996		val_loss=0.0150	val_acc=0.997

Epoch 127/150		train_loss=0.0235	train_acc=0.998		val_loss=0.0134	val_acc=1.000
Epoch 128/150		train_loss=0.0224	train_acc=0.998		val_loss=0.0122	val_acc=1.000
Epoch 129/150		train_loss=0.0220	train_acc=0.998		val_loss=0.0120	val_acc=1.000
Epoch 130/150		train_loss=0.0216	train_acc=0.998		val_loss=0.0125	val_acc=0.997
Epoch 131/150		train_loss=0.0218	train_acc=0.998		val_loss=0.0119	val_acc=1.000
Epoch 132/150		train_loss=0.0212	train_acc=0.998		val_loss=0.0108	val_acc=0.997
Epoch 133/150		train_loss=0.0212	train_acc=0.997		val_loss=0.0118	val_acc=0.997
Epoch 134/150		train_loss=0.0207	train_acc=0.998		val_loss=0.0115	val_acc=0.997
Epoch 135/150		train_loss=0.0204	train_acc=0.998		val_loss=0.0104	val_acc=0.997
Epoch 136/150		train_loss=0.0205	train_acc=0.997		val_loss=0.0110	val_acc=0.997
Epoch 137/150		train_loss=0.0205	train_acc=0.998		val_loss=0.0125	val_acc=0.997
Epoch 138/150		train_loss=0.0204	train_acc=0.998		val_loss=0.0110	val_acc=0.997
Epoch 139/150		train_loss=0.0195	train_acc=0.998		val_loss=0.0101	val_acc=0.997
Epoch 140/150		train_loss=0.0200	train_acc=0.997		val_loss=0.0106	val_acc=0.997
Epoch 141/150		train_loss=0.0197	train_acc=0.998		val_loss=0.0118	val_acc=0.997
Epoch 142/150		train_loss=0.0193	train_acc=0.998		val_loss=0.0103	val_acc=0.997
Epoch 143/150		train_loss=0.0193	train_acc=0.998		val_loss=0.0093	val_acc=1.000
Epoch 144/150		train_loss=0.0189	train_acc=0.998		val_loss=0.0102	val_acc=0.997
Epoch 145/150		train_loss=0.0189	train_acc=0.998		val_loss=0.0109	val_acc=0.997
Epoch 146/150		train_loss=0.0189	train_acc=0.998		val_loss=0.0104	val_acc=0.997
Epoch 147/150		train_loss=0.0188	train_acc=0.998		val_loss=0.0086	val_acc=1.000
Epoch 148/150		train_loss=0.0186	train_acc=0.998		val_loss=0.0089	val_acc=0.997
Epoch 149/150		train_loss=0.0185	train_acc=0.998		val_loss=0.0097	val_acc=0.997
Epoch 150/150		train_loss=0.0178	train_acc=0.998		val_loss=0.0092	val_acc=0.997

Toy transformer validation accuracy: 99.7%

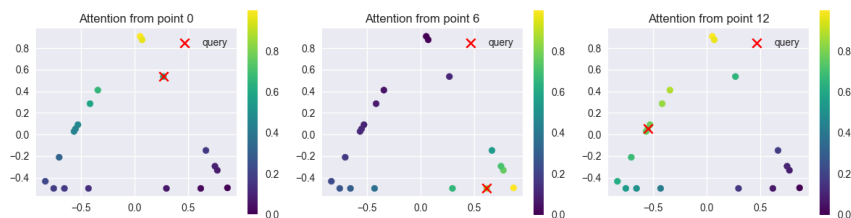




We can note a few things from this simple example:

- Even though the original points were shuffled in a random order (as you can see from the colors in the right plot above), we can see that the projected queries and keys (left plot) have learned to organize themselves in a meaningful way that almost mirrors the original x-y locations of the points.
- We can also see from the grey lines in the center plot that points attend most strongly to other points that are nearby in space, even though the model has no explicit knowledge of the spatial arrangement of the points.

Now let's pick a few example points and visualize how they attend to the rest of the set more directly by coloring the points according to the attention weights from a selected query point:



Here again we see that the attention weights are highest for points that are spatially close to the query point.

21.4.3 Scaling up to a Full Transformer Encoder

The above example was a single-head, single-layer, low-dimension attention model, so that we could visualize everything in 2D and gain intuition. However, in practice, we want to use multiple layers, multiple heads, and higher-dimensional latent spaces to get good performance. Below we build a larger transformer model (labeled **SetTransformer**) that can be trained on the same 2D shapes dataset, but with more capacity.


```

class SetTransformer(nn.Module):
    def __init__(
        self,
        input_dim: int,
        num_classes: int,
        d_model: int = 64,
        n_heads: int = 4,
        num_layers: int = 3,
        dim_feedforward: int = 128,
        dropout: float = 0.1,
        pooling: str = "cls",
    ):
        super().__init__()
        self.input_proj = nn.Linear(input_dim, d_model)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, d_model))
        self.layers = nn.ModuleList(
            [SelfAttentionBlock(d_model, n_heads, dim_feedforward, dropout) for _ in range(num_layers)]
        )
        self.pooling = pooling
        self.classifier = nn.Sequential(nn.LayerNorm(d_model), nn.Linear(d_model, num_classes))

    def forward(
        self,
        x: torch.Tensor,
        src_key_padding_mask: Optional[torch.Tensor] = None,
        return_attention: bool = False,
        return_features: bool = False,
    ):
        batch_size = x.size(0)
        proj = self.input_proj(x)
        cls_tokens = self.cls_token.expand(batch_size, -1, -1)
        tokens = torch.cat([cls_tokens, proj], dim=1)

        if src_key_padding_mask is not None:
            pad = torch.zeros(src_key_padding_mask.size(0), 1, dtype=src_key_padding_mask.dtype, device=src_
                attn_mask = torch.cat([pad, src_key_padding_mask], dim=1)
        else:
            attn_mask = None

        attention_maps = []
        for layer in self.layers:
            tokens, attn = layer(tokens, key_padding_mask=attn_mask)

```

```

        attention_maps.append(attn)

    if self.pooling == "cls":
        features = tokens[:, 0]
    else:
        valid = (~attn_mask[:, 1:]).unsqueeze(-1) if attn_mask is not None else torch.ones_like(tokens[:, 1:])
        summed = (tokens[:, 1:] * valid).sum(dim=1)
        counts = valid.sum(dim=1).clamp_min(1.0)
        features = summed / counts

    logits = self.classifier(features)

    outputs = (logits,)
    if return_attention:
        outputs += (attention_maps,)
    if return_features:
        outputs += (features,)

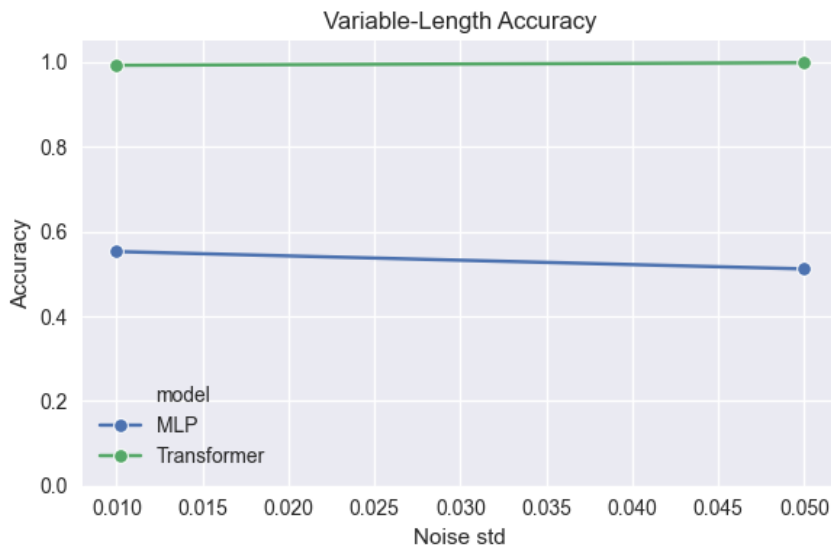
    if len(outputs) == 1:
        return outputs[0]
    return outputs

```

```

Epoch 01/8 | train_loss=1.0918 train_acc=0.372 | val_loss=1.0390 val_acc=0.603
Epoch 02/8 | train_loss=0.7943 train_acc=0.677 | val_loss=0.3791 val_acc=0.914
Epoch 03/8 | train_loss=0.2146 train_acc=0.954 | val_loss=0.1356 val_acc=0.960
Epoch 04/8 | train_loss=0.0836 train_acc=0.980 | val_loss=0.0920 val_acc=0.968
Epoch 05/8 | train_loss=0.0446 train_acc=0.991 | val_loss=0.0378 val_acc=0.992
Epoch 06/8 | train_loss=0.0269 train_acc=0.995 | val_loss=0.0282 val_acc=0.995
Epoch 07/8 | train_loss=0.0225 train_acc=0.996 | val_loss=0.0248 val_acc=0.995
Epoch 08/8 | train_loss=0.0196 train_acc=0.996 | val_loss=0.0249 val_acc=0.993
Noise 0.01 | Transformer accuracy: 99.3%
Epoch 01/8 | train_loss=1.0979 train_acc=0.353 | val_loss=1.0361 val_acc=0.424
Epoch 02/8 | train_loss=0.8103 train_acc=0.699 | val_loss=0.4268 val_acc=0.898
Epoch 03/8 | train_loss=0.2339 train_acc=0.953 | val_loss=0.0956 val_acc=0.989
Epoch 04/8 | train_loss=0.0703 train_acc=0.987 | val_loss=0.0427 val_acc=0.994
Epoch 05/8 | train_loss=0.0336 train_acc=0.994 | val_loss=0.0277 val_acc=0.994
Epoch 06/8 | train_loss=0.0247 train_acc=0.995 | val_loss=0.0207 val_acc=0.994
Epoch 07/8 | train_loss=0.0224 train_acc=0.995 | val_loss=0.0165 val_acc=0.996
Epoch 08/8 | train_loss=0.0214 train_acc=0.996 | val_loss=0.0148 val_acc=0.999
Noise 0.05 | Transformer accuracy: 99.9%

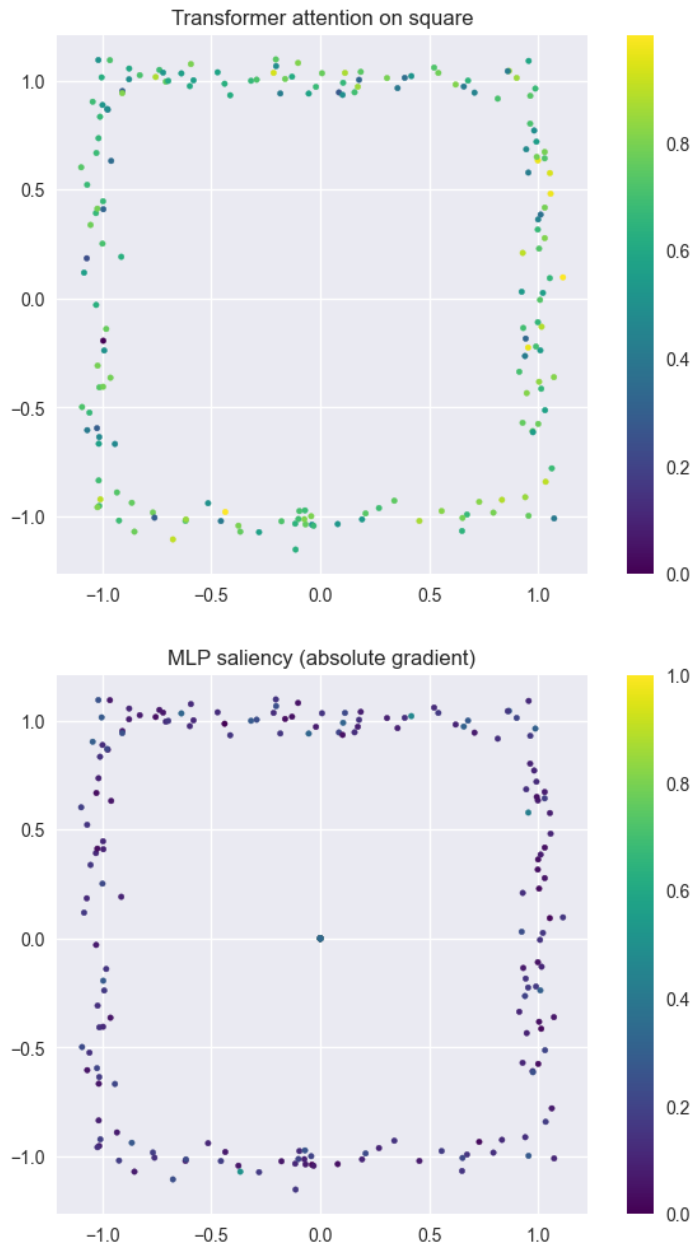
```



We can also visualize the attention patterns learned by this larger model, as well as how the MLP is weighting points after pooling:

```
Epoch 01/50 | train_loss=1.1410 train_acc=0.347
Epoch 02/50 | train_loss=1.0896 train_acc=0.365
Epoch 03/50 | train_loss=0.9695 train_acc=0.532
Epoch 04/50 | train_loss=0.5630 train_acc=0.798
Epoch 05/50 | train_loss=0.2649 train_acc=0.935
Epoch 06/50 | train_loss=0.1571 train_acc=0.957
Epoch 07/50 | train_loss=0.1191 train_acc=0.960
Epoch 08/50 | train_loss=0.0893 train_acc=0.968
Epoch 09/50 | train_loss=0.0819 train_acc=0.973
Epoch 10/50 | train_loss=0.0574 train_acc=0.982
Epoch 11/50 | train_loss=0.0449 train_acc=0.983
Epoch 12/50 | train_loss=0.0529 train_acc=0.983
Epoch 13/50 | train_loss=0.0480 train_acc=0.983
Epoch 14/50 | train_loss=0.0382 train_acc=0.990
Epoch 15/50 | train_loss=0.0329 train_acc=0.990
Epoch 16/50 | train_loss=0.0468 train_acc=0.983
Epoch 17/50 | train_loss=0.0371 train_acc=0.985
Epoch 18/50 | train_loss=0.0409 train_acc=0.985
Epoch 19/50 | train_loss=0.0172 train_acc=0.995
Epoch 20/50 | train_loss=0.0347 train_acc=0.987
Epoch 21/50 | train_loss=0.0125 train_acc=0.997
Epoch 22/50 | train_loss=0.0134 train_acc=0.997
Epoch 23/50 | train_loss=0.0229 train_acc=0.992
```

Epoch 24/50 | train_loss=0.0202 train_acc=0.993
Epoch 25/50 | train_loss=0.0207 train_acc=0.992
Epoch 26/50 | train_loss=0.0095 train_acc=0.998
Epoch 27/50 | train_loss=0.0079 train_acc=0.998
Epoch 28/50 | train_loss=0.0222 train_acc=0.995
Epoch 29/50 | train_loss=0.0064 train_acc=0.998
Epoch 30/50 | train_loss=0.0087 train_acc=0.998
Epoch 31/50 | train_loss=0.0061 train_acc=0.998
Epoch 32/50 | train_loss=0.0097 train_acc=0.995
Epoch 33/50 | train_loss=0.0438 train_acc=0.982
Epoch 34/50 | train_loss=0.0336 train_acc=0.992
Epoch 35/50 | train_loss=0.0061 train_acc=0.998
Epoch 36/50 | train_loss=0.0090 train_acc=0.998
Epoch 37/50 | train_loss=0.0023 train_acc=1.000
Epoch 38/50 | train_loss=0.0024 train_acc=1.000
Epoch 39/50 | train_loss=0.0023 train_acc=1.000
Epoch 40/50 | train_loss=0.0020 train_acc=1.000
Epoch 41/50 | train_loss=0.0018 train_acc=1.000
Epoch 42/50 | train_loss=0.0015 train_acc=1.000
Epoch 43/50 | train_loss=0.0016 train_acc=1.000
Epoch 44/50 | train_loss=0.0015 train_acc=1.000
Epoch 45/50 | train_loss=0.0014 train_acc=1.000
Epoch 46/50 | train_loss=0.0013 train_acc=1.000
Epoch 47/50 | train_loss=0.0013 train_acc=1.000
Epoch 48/50 | train_loss=0.0012 train_acc=1.000
Epoch 49/50 | train_loss=0.0012 train_acc=1.000
Epoch 50/50 | train_loss=0.0012 train_acc=1.000



21.5 3D Shapes — Synthetic Point Clouds

In the above examples, we showed a simple 2D shape classification task to build intuition about how transformers work on sets of points. However, the technique is more broadly applicable than this and can be used for more complex problems where something like a 2D or 3D

Convolutional Neural Network (CNN) would not be appropriate. In engineering a common task is dealing with point clouds or meshes, and these can often be un-ordered and of variable size, just like in the above case.

We will demonstrate this on a simple 3D shape classification task using synthetic point clouds. We will vary the density, jitter, and rotations, and also compare Transformers to point-cloud specific architectures like PointNet.

21.5.1 PointNet vs. Transformer for Point Clouds

Before training the 3D models, it is helpful to review the [PointNet](#) architecture, since it is the “convolutional cousin” of our set transformer, and will also act as a stronger baseline comparison than the MLP we used above.

PointNet

Specifically, given a set of 3D points $X = \{p_1, \dots, p_n\}$ with $p_i \in \mathbb{R}^3$, PointNet applies a **shared MLP** to every point and then aggregates with a symmetric pooling function:

$$\begin{aligned}h_i &= g(p_i; \theta) \in \mathbb{R}^m, \\H &= [h_1, \dots, h_n] \in \mathbb{R}^{m \times n}, \\z &= \text{pool}(H) = \max_i h_i \quad (\text{or mean}), \\y &= f(z; \phi).\end{aligned}$$

Here g and f are small MLPs, and m is the dimension of our model’s latent space, which we get to choose. In our implementation:

- g is the 1D-convolutional `shared_mlp` (kernel size 1 = pointwise MLP),
- `torch.max` over points implements the permutation-invariant pooling, and
- `fc` maps the global feature z to class logits.

Key idea: permutation invariance is enforced *only* by the final pooling. Every point is processed independently before pooling, so there is **no explicit message passing between points**.

Transformer for point clouds

The set transformer uses almost the opposite design:

- Each point is first embedded into a latent space $h_i = Wx_i$.
- Self-attention layers repeatedly **mix information across points** via learned attention weights a_{ij} .
- A CLS token or pooling over attended features produces the final global descriptor.

From a functional perspective:

- PointNet learns $y = F(\max_i g(p_i))$, where all interactions are implicit through the max.
- The transformer learns $y = F_{\text{Trans}}(\{p_1, \dots, p_n\})$ where each layer already blends local and global structure through attention.

In practice, this means that PointNet is very strong for **coarse global shape** (e.g., sphere vs. cube) but may struggle to capture fine-grained relational patterns unless we widen g a lot. Transformers can model those relations directly, at the cost of more computation.

What we will do now is construct a dataset of 3D point clouds sampled from simple shapes (spheres, cubes, cones, cylinders), and then train both a PointNet and a Transformer model on this dataset to compare their performance. Let's first take a look at some example data samples from our synthetic 3D shapes dataset:

[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

```
class PointNetClassifier(nn.Module):
    def __init__(self, input_dim: int = 3, widths: Tuple[int, ...] = (64, 128, 256), num_classes: int = 4, dr
        super().__init__()
        layers: List[nn.Module] = []
        dims = [input_dim, *widths]
        for in_dim, out_dim in zip(dims[:-1], dims[1:]):
            layers.append(nn.Conv1d(in_dim, out_dim, kernel_size=1))
            layers.append(nn.BatchNorm1d(out_dim))
            layers.append(nn.ReLU())
        self.shared_mlp = nn.Sequential(*layers)
        self.dropout = nn.Dropout(dropout)
        self.fc = nn.Sequential(
            nn.Linear(widths[-1], 256),
            nn.ReLU(),
```

```

        nn.Dropout(dropout),
        nn.Linear(256, num_classes),
    )

    def forward(self, x: torch.Tensor, return_features: bool = False):
        x = x.transpose(1, 2)
        features = self.shared_mlp(x)
        pooled = torch.max(features, dim=2).values
        logits = self.fc(self.dropout(pooled))
        if return_features:
            return logits, pooled
        return logits

mlp_cloud = ShapeMLP(input_dim=cloud_dataset.max_points * 3, hidden_dims=(16, 16, 16), num_classes=len(SH
pointnet = PointNetClassifier(num_classes=len(SHAPE_TYPES_3D), widths = (16, 16, 16))
transformer_cloud = SetTransformer(
    input_dim=3,
    num_classes=len(SHAPE_TYPES_3D),
    d_model= 15,
    n_heads=2,
    num_layers=3,
    dim_feedforward=20,
)

def pointnet_forward(model, batch, device):
    return model(batch["points"].to(device))

def transformer_cloud_forward(model, batch, device):
    return model(batch["points"].to(device), src_key_padding_mask=batch["mask"].to(device))

models_cloud = {
    "MLP": (mlp_cloud, mlp_forward),
    "PointNet": (pointnet, pointnet_forward),
    "Transformer": (transformer_cloud, transformer_cloud_forward),
}

# Print the parameter counts for each of the models in models_cloud:
for name, (model, _) in models_cloud.items():
    param_count = sum(p.numel() for p in model.parameters())

```



```
print(f"{name} parameter count: {param_count:,}")
```

MLP parameter count: 49,780

PointNet parameter count: 6,084

Transformer parameter count: 4,099

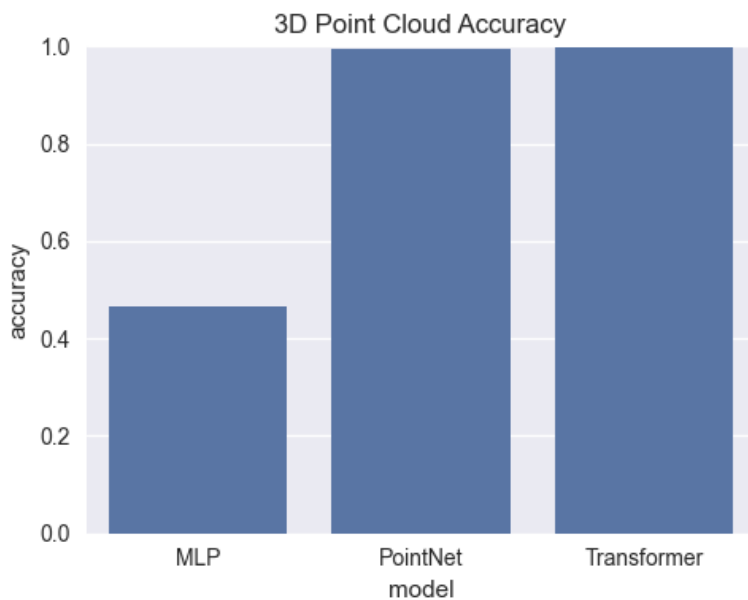
Epoch 01/15		train_loss=1.3906	train_acc=0.250		val_loss=1.3825	val_acc=0.275
Epoch 02/15		train_loss=1.3708	train_acc=0.310		val_loss=1.3693	val_acc=0.261
Epoch 02/15		train_loss=1.3708	train_acc=0.310		val_loss=1.3693	val_acc=0.261
Epoch 03/15		train_loss=1.3203	train_acc=0.353		val_loss=1.3281	val_acc=0.306
Epoch 03/15		train_loss=1.3203	train_acc=0.353		val_loss=1.3281	val_acc=0.306
Epoch 04/15		train_loss=1.2075	train_acc=0.451		val_loss=1.2738	val_acc=0.405
Epoch 04/15		train_loss=1.2075	train_acc=0.451		val_loss=1.2738	val_acc=0.405
Epoch 05/15		train_loss=1.0446	train_acc=0.561		val_loss=1.2201	val_acc=0.410
Epoch 05/15		train_loss=1.0446	train_acc=0.561		val_loss=1.2201	val_acc=0.410
Epoch 06/15		train_loss=0.8864	train_acc=0.594		val_loss=1.2035	val_acc=0.414
Epoch 06/15		train_loss=0.8864	train_acc=0.594		val_loss=1.2035	val_acc=0.414
Epoch 07/15		train_loss=0.7685	train_acc=0.644		val_loss=1.2120	val_acc=0.410
Epoch 07/15		train_loss=0.7685	train_acc=0.644		val_loss=1.2120	val_acc=0.410
Epoch 08/15		train_loss=0.6742	train_acc=0.736		val_loss=1.2601	val_acc=0.416
Epoch 08/15		train_loss=0.6742	train_acc=0.736		val_loss=1.2601	val_acc=0.416
Epoch 09/15		train_loss=0.5971	train_acc=0.838		val_loss=1.3152	val_acc=0.420
Epoch 09/15		train_loss=0.5971	train_acc=0.838		val_loss=1.3152	val_acc=0.420
Epoch 10/15		train_loss=0.5040	train_acc=0.910		val_loss=1.3748	val_acc=0.451
Epoch 10/15		train_loss=0.5040	train_acc=0.910		val_loss=1.3748	val_acc=0.451
Epoch 11/15		train_loss=0.4454	train_acc=0.938		val_loss=1.4523	val_acc=0.459
Epoch 11/15		train_loss=0.4454	train_acc=0.938		val_loss=1.4523	val_acc=0.459
Epoch 12/15		train_loss=0.3808	train_acc=0.952		val_loss=1.5529	val_acc=0.459
Epoch 12/15		train_loss=0.3808	train_acc=0.952		val_loss=1.5529	val_acc=0.459
Epoch 13/15		train_loss=0.3262	train_acc=0.957		val_loss=1.6719	val_acc=0.460
Epoch 13/15		train_loss=0.3262	train_acc=0.957		val_loss=1.6719	val_acc=0.460
Epoch 14/15		train_loss=0.2943	train_acc=0.958		val_loss=1.7680	val_acc=0.455
Epoch 14/15		train_loss=0.2943	train_acc=0.958		val_loss=1.7680	val_acc=0.455
Epoch 15/15		train_loss=0.2600	train_acc=0.963		val_loss=1.8689	val_acc=0.468
MLP accuracy: 46.8%						
Epoch 15/15		train_loss=0.2600	train_acc=0.963		val_loss=1.8689	val_acc=0.468
MLP accuracy: 46.8%						
Epoch 01/15		train_loss=1.4902	train_acc=0.268		val_loss=1.3391	val_acc=0.255
Epoch 01/15		train_loss=1.4902	train_acc=0.268		val_loss=1.3391	val_acc=0.255
Epoch 02/15		train_loss=1.3876	train_acc=0.297		val_loss=1.2545	val_acc=0.639
Epoch 02/15		train_loss=1.3876	train_acc=0.297		val_loss=1.2545	val_acc=0.639

Epoch 03/15		train_loss=1.3141	train_acc=0.360		val_loss=1.1578	val_acc=0.574
Epoch 03/15		train_loss=1.3141	train_acc=0.360		val_loss=1.1578	val_acc=0.574
Epoch 04/15		train_loss=1.2116	train_acc=0.433		val_loss=1.0201	val_acc=0.748
Epoch 04/15		train_loss=1.2116	train_acc=0.433		val_loss=1.0201	val_acc=0.748
Epoch 05/15		train_loss=1.1339	train_acc=0.484		val_loss=0.8737	val_acc=0.876
Epoch 05/15		train_loss=1.1339	train_acc=0.484		val_loss=0.8737	val_acc=0.876
Epoch 06/15		train_loss=1.0209	train_acc=0.545		val_loss=0.7659	val_acc=0.856
Epoch 06/15		train_loss=1.0209	train_acc=0.545		val_loss=0.7659	val_acc=0.856
Epoch 07/15		train_loss=0.9293	train_acc=0.585		val_loss=0.6443	val_acc=0.902
Epoch 07/15		train_loss=0.9293	train_acc=0.585		val_loss=0.6443	val_acc=0.902
Epoch 08/15		train_loss=0.8323	train_acc=0.643		val_loss=0.5531	val_acc=0.917
Epoch 08/15		train_loss=0.8323	train_acc=0.643		val_loss=0.5531	val_acc=0.917
Epoch 09/15		train_loss=0.7074	train_acc=0.715		val_loss=0.4403	val_acc=0.974
Epoch 09/15		train_loss=0.7074	train_acc=0.715		val_loss=0.4403	val_acc=0.974
Epoch 10/15		train_loss=0.6615	train_acc=0.717		val_loss=0.3780	val_acc=0.960
Epoch 10/15		train_loss=0.6615	train_acc=0.717		val_loss=0.3780	val_acc=0.960
Epoch 11/15		train_loss=0.6140	train_acc=0.747		val_loss=0.3815	val_acc=0.858
Epoch 11/15		train_loss=0.6140	train_acc=0.747		val_loss=0.3815	val_acc=0.858
Epoch 12/15		train_loss=0.5653	train_acc=0.772		val_loss=0.3253	val_acc=0.975
Epoch 12/15		train_loss=0.5653	train_acc=0.772		val_loss=0.3253	val_acc=0.975
Epoch 13/15		train_loss=0.5264	train_acc=0.791		val_loss=0.2857	val_acc=0.981
Epoch 13/15		train_loss=0.5264	train_acc=0.791		val_loss=0.2857	val_acc=0.981
Epoch 14/15		train_loss=0.5014	train_acc=0.801		val_loss=0.2917	val_acc=0.948
Epoch 14/15		train_loss=0.5014	train_acc=0.801		val_loss=0.2917	val_acc=0.948
Epoch 15/15		train_loss=0.4934	train_acc=0.805		val_loss=0.2345	val_acc=0.995
PointNet accuracy: 99.5%						
Epoch 15/15		train_loss=0.4934	train_acc=0.805		val_loss=0.2345	val_acc=0.995
PointNet accuracy: 99.5%						
Epoch 01/15		train_loss=1.4022	train_acc=0.253		val_loss=1.3634	val_acc=0.271
Epoch 01/15		train_loss=1.4022	train_acc=0.253		val_loss=1.3634	val_acc=0.271
Epoch 02/15		train_loss=1.1292	train_acc=0.516		val_loss=0.8269	val_acc=0.671
Epoch 02/15		train_loss=1.1292	train_acc=0.516		val_loss=0.8269	val_acc=0.671
Epoch 03/15		train_loss=0.6382	train_acc=0.769		val_loss=0.4080	val_acc=0.865
Epoch 03/15		train_loss=0.6382	train_acc=0.769		val_loss=0.4080	val_acc=0.865
Epoch 04/15		train_loss=0.3927	train_acc=0.868		val_loss=0.2545	val_acc=0.927
Epoch 04/15		train_loss=0.3927	train_acc=0.868		val_loss=0.2545	val_acc=0.927
Epoch 05/15		train_loss=0.2578	train_acc=0.912		val_loss=0.1931	val_acc=0.939
Epoch 05/15		train_loss=0.2578	train_acc=0.912		val_loss=0.1931	val_acc=0.939
Epoch 06/15		train_loss=0.2062	train_acc=0.933		val_loss=0.1657	val_acc=0.948
Epoch 06/15		train_loss=0.2062	train_acc=0.933		val_loss=0.1657	val_acc=0.948
Epoch 07/15		train_loss=0.1952	train_acc=0.941		val_loss=0.1688	val_acc=0.938
Epoch 07/15		train_loss=0.1952	train_acc=0.941		val_loss=0.1688	val_acc=0.938
Epoch 08/15		train_loss=0.1261	train_acc=0.960		val_loss=0.0951	val_acc=0.975

```

Epoch 08/15 | train_loss=0.1261 train_acc=0.960 | val_loss=0.0951 val_acc=0.975
Epoch 09/15 | train_loss=0.1043 train_acc=0.968 | val_loss=0.0896 val_acc=0.976
Epoch 09/15 | train_loss=0.1043 train_acc=0.968 | val_loss=0.0896 val_acc=0.976
Epoch 10/15 | train_loss=0.1486 train_acc=0.955 | val_loss=0.0628 val_acc=0.983
Epoch 10/15 | train_loss=0.1486 train_acc=0.955 | val_loss=0.0628 val_acc=0.983
Epoch 11/15 | train_loss=0.0598 train_acc=0.984 | val_loss=0.0439 val_acc=0.989
Epoch 11/15 | train_loss=0.0598 train_acc=0.984 | val_loss=0.0439 val_acc=0.989
Epoch 12/15 | train_loss=0.0442 train_acc=0.989 | val_loss=0.0534 val_acc=0.990
Epoch 12/15 | train_loss=0.0442 train_acc=0.989 | val_loss=0.0534 val_acc=0.990
Epoch 13/15 | train_loss=0.0311 train_acc=0.992 | val_loss=0.0170 val_acc=0.995
Epoch 13/15 | train_loss=0.0311 train_acc=0.992 | val_loss=0.0170 val_acc=0.995
Epoch 14/15 | train_loss=0.0127 train_acc=0.997 | val_loss=0.0154 val_acc=0.995
Epoch 14/15 | train_loss=0.0127 train_acc=0.997 | val_loss=0.0154 val_acc=0.995
Epoch 15/15 | train_loss=0.0104 train_acc=0.998 | val_loss=0.0130 val_acc=0.998
Epoch 15/15 | train_loss=0.0104 train_acc=0.998 | val_loss=0.0130 val_acc=0.998
Transformer accuracy: 99.8%
Transformer accuracy: 99.8%

```



[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

21.6 Summary

In this chapter we introduced the concept of transformers, which use self-attention to decide which elements in an un-ordered set to use for downstream prediction tasks. While these were developed in the context of Natural Language Processing (where the un-ordered set consisted of characters, words, or phrases), we saw in this notebook how it could extend to arbitrary other objects like unordered point-cloud data. What we saw in our comparative experiments were:

- **MLP (flattened inputs)**
 - *Strengths*: simple to implement; fast on small, fixed-size inputs; works well when point order and count are strictly controlled (our first 2D experiment).
 - *Weaknesses*: not permutation-invariant; brittle under reordering, variable-length sets, and padding artifacts; has no built-in sense of locality or geometry beyond what you manually encode in the features.
- **PointNet (shared MLP + symmetric pooling)**
 - *Strengths*: explicitly permutation-invariant via max/mean pooling; handles variable point counts; very effective for coarse global 3D shape classification with relatively few parameters.
 - *Weaknesses*: points do not interact before pooling, so capturing subtle **relations** (e.g., “distance between two features” or “joint configuration of surfaces”) may require very wide hidden layers; global pooling can throw away spatial structure that might matter for fine-grained tasks.
- **Set Transformer (self-attention on points)**
 - *Strengths*: every point can attend to every other, so the model naturally captures edges, corners, support planes, and long-range geometric relations; masking and pooling make it robust to variable-length, noisy, and unordered sets; the same architecture scales from toy 2D sets to realistic 3D scans.
 - *Weaknesses*: attention is more computationally expensive than pointwise MLPs (roughly $\mathcal{O}(n^2)$ in the number of points per set); requires more care in tuning depth, width, and regularization.

Part III

Engineering-Specific Considerations

These chapters deal with issues that are particularly prevalent in Mechanical Engineering contexts and may cut across specific models mentioned in Part 2. These are likely to persist over time, even as new models or approaches are invented, although they will likely get easier to address as research fields expand.

22 Active and Semi-Supervised Learning

So far, all of the models we have used in this book have assumed that data was given to us, and in many cases fully labeled. However, in many real-world scenarios, obtaining data of any kind (let alone labeled data) can be expensive and time-consuming. Instead, we are often in a position where we can choose what data to collect or label, and we want to make the most of our limited resources. Likewise, we may have access to a large amount of unlabeled data, but labeling it all may be infeasible or too costly. In these situations, we can use techniques from *active learning* and *semi-supervised learning* to make the most of the data we can collect.

22.1 Learning Objectives

By the end of this chapter, you will be able to:

- Understand the concepts of active learning and semi-supervised learning.
- Implement basic active learning strategies, such as Uncertainty sampling, to select informative data points for labeling.
- Apply semi-supervised learning techniques, such as label propagation, to leverage unlabeled data in model training.
- Evaluate the comparative performance of models trained using active and semi-supervised learning methods on several baseline cases.

22.2 Active Learning

Active learning is a machine learning paradigm where the model can interactively query a user (or some other information source) to obtain the desired outputs for new data points. The main idea is to select the most informative data points to label, thereby maximizing the model's performance while minimizing the amount of labeled data required. [Settles 2010](#) categorizes active learning approaches into three main settings depending on the type of queries you are able to make:

1. **Membership Query Synthesis:** The model can generate new data points and request labels for them. This is often used in scenarios where the model can create synthetic examples. Common examples that would occur in Engineering include:

- Running simulations with specific input parameters and querying the output. For example, running a CFD simulation with a given geometry or boundary condition. In these cases, I can choose which samples/datapoints I want to run the simulation for and can decide at any time to generate new or different ones.
 - Conducting experiments with specific configurations and querying the results. For example, in a materials science context, I might choose to synthesize a material with certain properties and then measure its performance.
2. **Stream-based Selective Sampling:** The model receives a stream of unlabeled data points and must decide whether to query the label for each point as it arrives. This is common in scenarios where data arrives in real-time, such as:
- Monitoring sensor data from an IoT device (e.g., a video or audio sensor on a production line or mobile robot) and deciding whether to request human annotation for specific readings that are uncertain or unusual.
 - Analyzing streams of incoming user interactions on a website or device and choosing to label certain actions (e.g., clicks, purchases, motions) based on their potential informativeness for improving your prediction system.
3. **Pool-based Sampling:** The model has access to a large pool of unlabeled data and can select specific points from this pool to query for labels. This is common in engineering scenarios where a large dataset is available, but labeling is expensive, such as:
- Selecting parts from a production run to inspect or possibly perform destructive testing on, in order to improve a quality control model.
 - You can synthesize a large number of design variations (e.g., different geometries, materials, or configurations) and then select a subset to simulate or test in order to improve a predictive model for performance or reliability. For example, if you have a manufacturing process parameter that can produce a variety of randomized material microstructures (e.g., metal foams with various pore sizes and distributions), you might generate a large pool of these microstructures and then select only a few to experimentally test in order to improve your model for predicting mechanical properties.

Depending on your use case, one of these settings may be more appropriate than the others, and the below algorithms may need to be adapted accordingly. In addition, you may need to consider whether you sample data *sequentially* (i.e., one at a time, updating your model after each new label) or *in batches* (i.e., selecting multiple data points to label at once before updating your model). The latter is referred to as *batch-mode active learning* and is often more practical in real-world scenarios where labeling multiple data points at once is more efficient. For example, the setup costs for a metal powder-bed laser fusion test may be high, so it makes sense to run multiple parts in one go rather than setting up the machine for each individual test. In these cases, we need to consider not only the value of a given datapoint, but rather the value of a set of datapoints.

22.3 Types of Active Learning Strategies

There are several strategies for selecting which data points to label in active learning. The most common general strategies (outlined in [Settles 2010](#)) include:

1. **Uncertainty Sampling:** The model selects data points for which it is least certain about the predicted label. The advantage of this approach is that it focuses on the areas of the input space where the model is most unsure, which can lead to significant improvements in performance with fewer labeled examples. It is also often relatively simple to implement. The main disadvantage is that it may focus too much on outliers or noisy data points, which can lead to suboptimal performance, and also the points where the model is most uncertain may not always be the most informative for improving the model. Specific common uncertainty measures are:
 - Least Confidence: Select the data point with the lowest predicted probability for the most likely class.
 - Margin Sampling: Select the data point with the smallest difference between the top two predicted class probabilities.
 - Entropy-based Sampling: Select the data point with the highest entropy in the predicted class probabilities.
2. **Query-by-Committee:** A committee of models is trained on the current labeled data, and data points are selected based on the disagreement among the committee members. The idea is that if different models disagree on a data point, it is likely to be informative. The advantage of this approach is that it can work across a wide variety of model types, since all one has to compute is the final decision of each model in the committee. The disadvantage is that it can be computationally expensive to train multiple models, and the choice of committee members can significantly impact performance.
3. **Expected Model Change:** Data points are selected based on the expected change in the model parameters if the point were to be labeled and added to the training set. This approach aims to select points that will have the most significant impact on the model. The main advantage of this approach is that it directly targets points that will change the model, potentially leading to faster learning compared to, say, uncertainty sampling. The disadvantage is that it can be computationally intensive, as it requires estimating the impact of each potential data point on the model parameters. In the general case, this could mean retraining the model for each candidate point, which is often infeasible. However, for certain models (e.g., linear models), efficient approximations can be used that make this approach more practical.
4. **Expected Error Reduction:** Data points are selected based on the expected reduction in the model's generalization error if the point were to be labeled and added to the training set. This approach aims to select points that will most improve the model's performance on unseen data. This is in some sense the most "natural" metric of performance with which to select new datapoints, and thus the main advantage of this approach is that it directly targets points that will improve the model's performance on unseen data.

The disadvantage is that it can be computationally intensive, as it requires estimating the impact of each potential data point on the model's generalization error. This often involves not only retraining the model multiple times, but also then testing it on a held-out validation set. This can become infeasible for complex models or large datasets. However, for certain models (e.g., Gaussian processes), efficient approximations can be used that make this approach more practical.

5. **Variance Reduction:** Data points are selected based on the expected reduction in the model's output variance if the point were to be labeled and added to the training set. This approach aims to select points that will directly reduce the uncertainty in the model's predictions. The advantage of this approach is that it can lead to more confident predictions, which can be particularly useful in scenarios where uncertainty quantification is important. It is also often the case that data points that are most useful for reducing the model's variance are not necessarily those points about which the model is most uncertain. The disadvantage is that it can be computationally intensive, as it requires estimating the impact of each potential data point on the model's output variance, which may not be easily computable compared to, say, a decision surface or likelihood. However, for certain models (e.g., Gaussian processes), efficient approximations can be used that make this approach more practical.
6. **Density-Weighted Methods:** These methods take into account the *distribution* of the unlabeled data when selecting points to label. The idea is to prefer points that are not only uncertain but also representative of the overall data distribution. The advantage of this approach is that it can help avoid selecting outliers or noisy points that may not be representative of the underlying data distribution. The disadvantage is that it can be more complex to implement, as it requires estimating the density of the unlabeled data, which can be challenging or have large numerical variance in high-dimensional spaces.

For this chapter, we will focus only on implementing Uncertainty Sampling due to its relative simplicity and wide applicability in many practical scenarios.

22.4 Uncertainty Sampling Example: Gaussian Processes

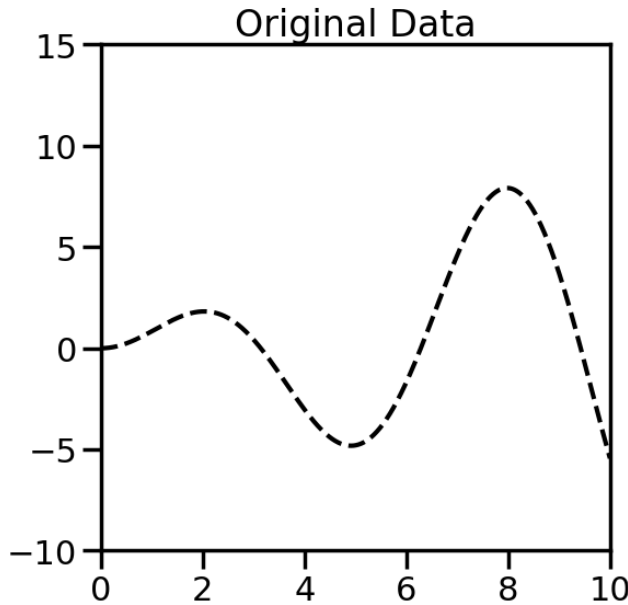
In the example below, we will use a Gaussian Process regression model to select data points to label based on uncertainty sampling. At a high level, a Gaussian Process (GP) is a non-parametric model that defines a distribution over functions. It is particularly well-suited for active learning because it provides not only predictions but also uncertainty estimates for those predictions. Specifically, the mean and variance of the GP's predictions can be expressed in the following closed-form equations:

$$\begin{aligned}\mu(x_*) &= k(x_*, X)[K(X, X) + \sigma_n^2 I]^{-1}y \\ \sigma^2(x_*) &= k(x_*, x_*) - k(x_*, X)[K(X, X) + \sigma_n^2 I]^{-1}k(X, x_*)\end{aligned}$$

where:

- $\mu(x_*)$ is the predicted mean at the new input point x_* .
- $\sigma^2(x_*)$ is the predicted variance at the new input point x_* .
- $k(x_*, X)$ is the covariance vector between the new input point x_* and the training inputs X .
- $K(X, X)$ is the covariance matrix of the training inputs.

We can see from the above that the uncertainty of the model's predictions is directly given by the predicted variance $\sigma^2(x_*)$. Thus, in uncertainty sampling, we can just test the model on several candidate x locations that we haven't yet measured and select the data points with the highest predicted variance for labeling. Below we will create a sample "true" function (the dashed line), and then generate noisy observations of this function at a few initial points. We will then iteratively select new points to label based on uncertainty sampling and update our GP model accordingly. For comparison sakes, we will also implement a random sampling strategy where new points are selected randomly from the pool of unlabeled data.



We will now create a Gaussian Process model and implement the uncertainty sampling strategy. For this, we will need to pick a kernel function for the GP, which we will choose to be the Radial Basis Function (RBF) kernel for this example. (However, you can modify the code cell below if you want to experiment with other kernel functions.) In addition, we will need to decide on the number and placement of the initial labeled data points, since otherwise the model will have no information to start with. Common strategies here include random sampling, Latin Hypercube Sampling, or selecting points that are evenly spaced (or at the boundaries of) the input space. For simplicity, we will use random sampling for the initial points in this example, although you can modify the code cell below to experiment with other strategies if you wish.

What we will see in the plots below is a red line and shaded region representing the GP model (both mean and variance) *before* we sampled a new datapoint, and then we will overlay the GP model *after* we have sampled and added the new datapoint to the training set. The new model will be shown with a green line and shaded region. The selected datapoint will be marked with a large red circle. In Uncertainty Sampling, we should see that the selected datapoint is located where the initial model (red line) has high uncertainty (wide red shaded region). In Random Sampling, the selected datapoint will be chosen randomly from the input space. You can see what happens to the model's uncertainty as we add the point, as well as the overall Mean Squared Error (MSE) on a held-out test set after each iteration.

```
# Create the Gaussian Process model
# We can start with a basic RBF kernel
kernel = RBF(length_scale=1) #+ WhiteKernel(noise_level=0.01)
# You can also experiment with adding noise to the kernel to see how this affects the sampling
# kernel = RBF(length_scale=1) + WhiteKernel(noise_level=0.01)

gp = GaussianProcessRegressor(kernel=kernel, alpha=0.01,
                              optimizer=None, normalize_y=True, random_state=1)

def sample_data_points(X, y, gp, num_points=35):
    """Compare Uncertainty Sampling vs Random Sampling for Active Learning."""
    # Now, we are ready to begin learning!
    # We will have two different models that we'll test out:
    # 1) Uncertainty Sampling: Pick points where we are most uncertain
    # 2) Random Sampling: Pick points randomly
    # train_ind holds the indices of what training samples we'll use
    train_ind = {
        'Uncertainty': np.zeros(len(X), dtype=bool),
        'Random':      np.zeros(len(X), dtype=bool)
    }
    options = train_ind.keys()

    # We can now initialize the first few training points.
    possible_points = np.array(list(range(len(X))))
    # Possible Initialization options
    # 1. Start with the same two randomly selected points
    for ind in np.random.choice(possible_points, 2):
        for o in options:
            train_ind[o][ind] = True
    # Or
    # 2. Start with end-points
    # for o in options:
```

```

# train_ind[o][0] = True
# train_ind[o][-1] = True

# How much data should we collect?
num_total_samples = 35

# Let's store the errors for the models:
errors = {'Uncertainty':np.zeros(num_total_samples),
          'Random':np.zeros(num_total_samples)}

# Let's select an increasing number of new points to try:
for i in range(num_total_samples):
    # As i increases, we increase the number of points
    plt.figure(figsize=(16,6))
    for j,o in enumerate(options): # For each sampling option
        plt.subplot(1,2,j+1) # Create different subplot

        # Fit the original GP Model (without the new point):
        gp.fit(X[train_ind[o],:],y[train_ind[o]])

        # Plot the original model's bounds (just for comparison sake)
        yp,sigma = gp.predict(x, return_std=True)
        plt.fill(np.concatenate([x, x[::-1]]),
                 np.concatenate([yp - 1.9600 * sigma,
                                (yp + 1.9600 * sigma)[::-1]]), 'r',
                 alpha=0.2, ec='g', label='95% confidence interval')
        # Plot the predicted function
        plt.plot(x,yp,'r-',alpha=0.2)

    #####
    # Now do the Active Learning part
    #####
    # Here is Uncertainty Sampling:
    if o == 'Uncertainty':
        # First, we need to predict the MSE (uncertainty) at
        # each of the non-training points:
        yp,sigma = gp.predict(X[~train_ind[o],:], return_std=True)

        # The points that we'll choose from are the points that have the
        # maximum mean square error in the GP (highest uncertainty)
        candidates = np.extract(sigma == np.amax(sigma),X[~train_ind[o],:])

```

```

# Side note:
# If you wanted to turn Uncertainty Sampling into Global Bayesian
# Optimization instead, then uncomment the below two lines, which implements
# Upper Confidence Bound (UCB) sampling:
# ucb = yp + 1.96*sigma
# candidates = np.extract(ucb == np.amax(ucb),X[~train_ind[o],:])

# Choose the next sampling point randomly from the max candidates
# (It is unlikely that len(candidates)>1, but you never know)
next_point = np.random.choice(candidates.flatten())
# Find the training sample index corresponding to the next sample
next_ind = np.argwhere(X.flatten() == next_point)

# Here is Random Sampling:
elif o == 'Random':
    # Just pick the point randomly from the points that we
    # haven't chosen already (i.e., ~train_ind[o])
    next_ind = np.random.choice(possible_points[~train_ind[o]],1)

# Add the selected point to the training samples
train_ind[o][next_ind] = True

#####
# Done with Active Learning part
#####

# Fit the GP again, this time with the added training point:
gp.fit(X[train_ind[o],:],y[train_ind[o]])

# Now plot the updated model so we can see the difference:
yp,sigma = gp.predict(x, return_std=True)

# Draw the 95% Confidence Interval
plt.fill(np.concatenate([x, x[:-1]]),
        np.concatenate([yp - 1.9600 * sigma,
                        (yp + 1.9600 * sigma)[:-1]]), 'g',
        alpha=0.3, ec='None', label='95% confidence interval')
# Plot the original function
plt.plot(x,f(x),'k--',alpha=1)
# Plot the predicted function
plt.plot(x,yp,'g-',alpha=1)
# Plot the original training points

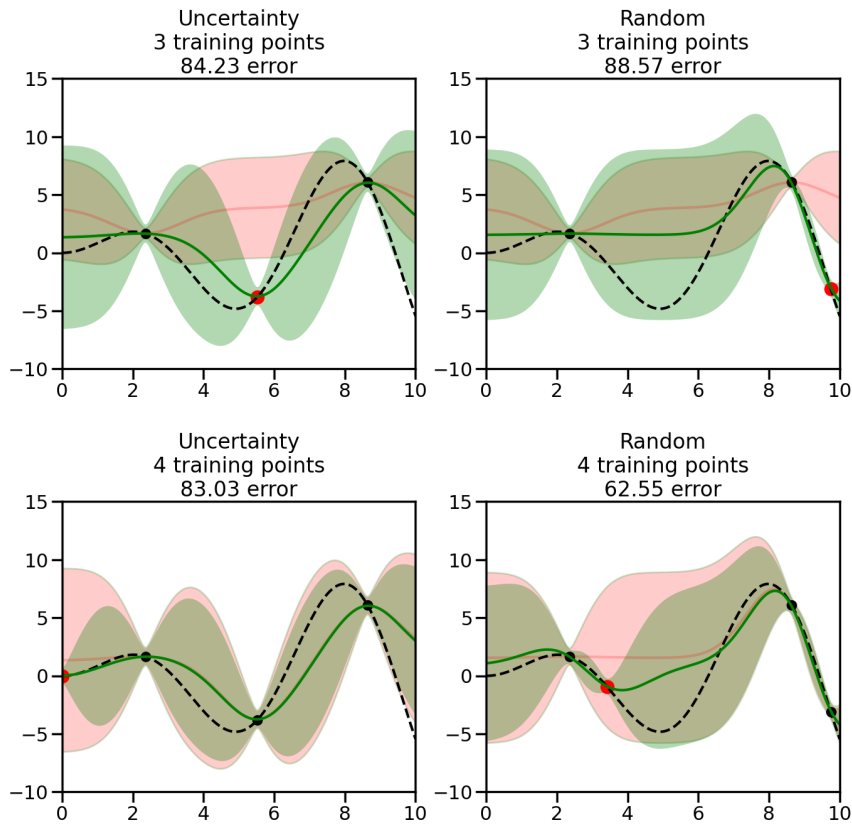
```

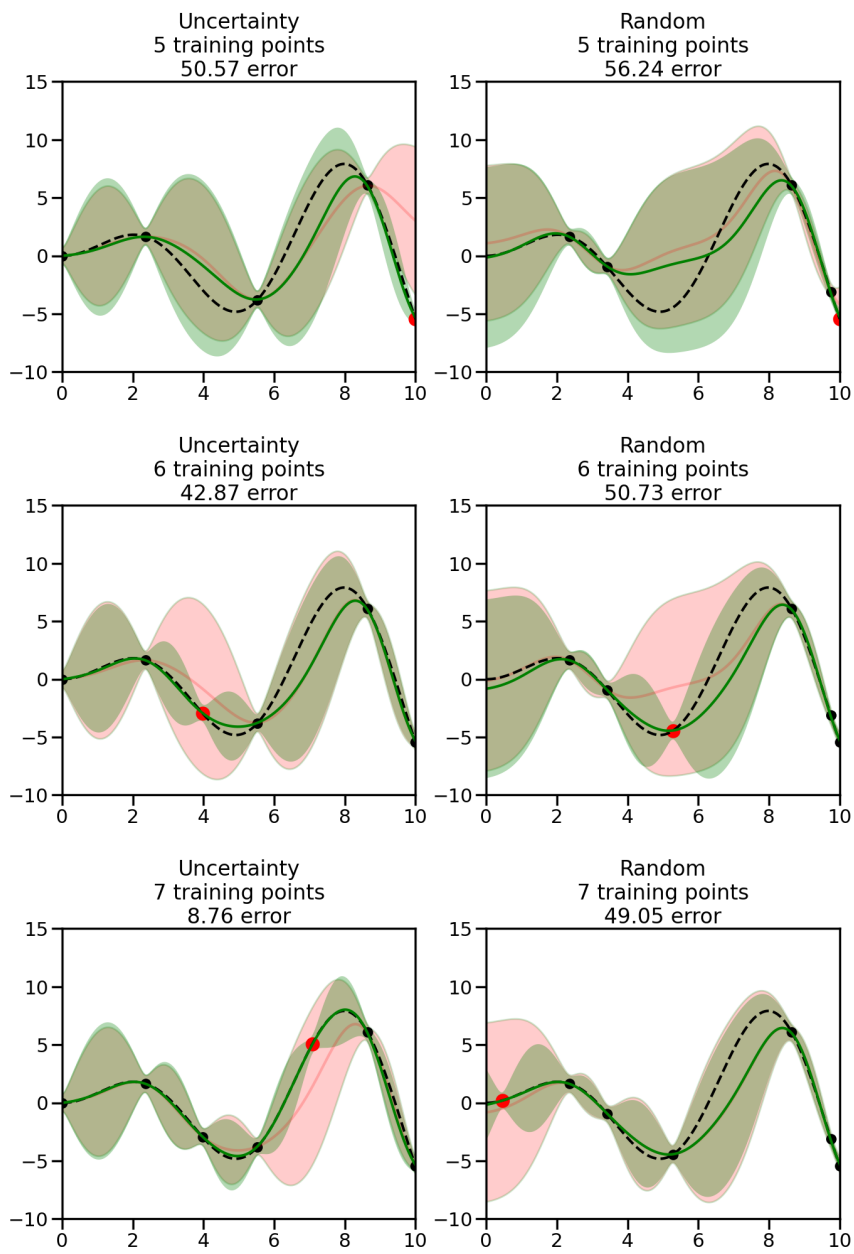
```

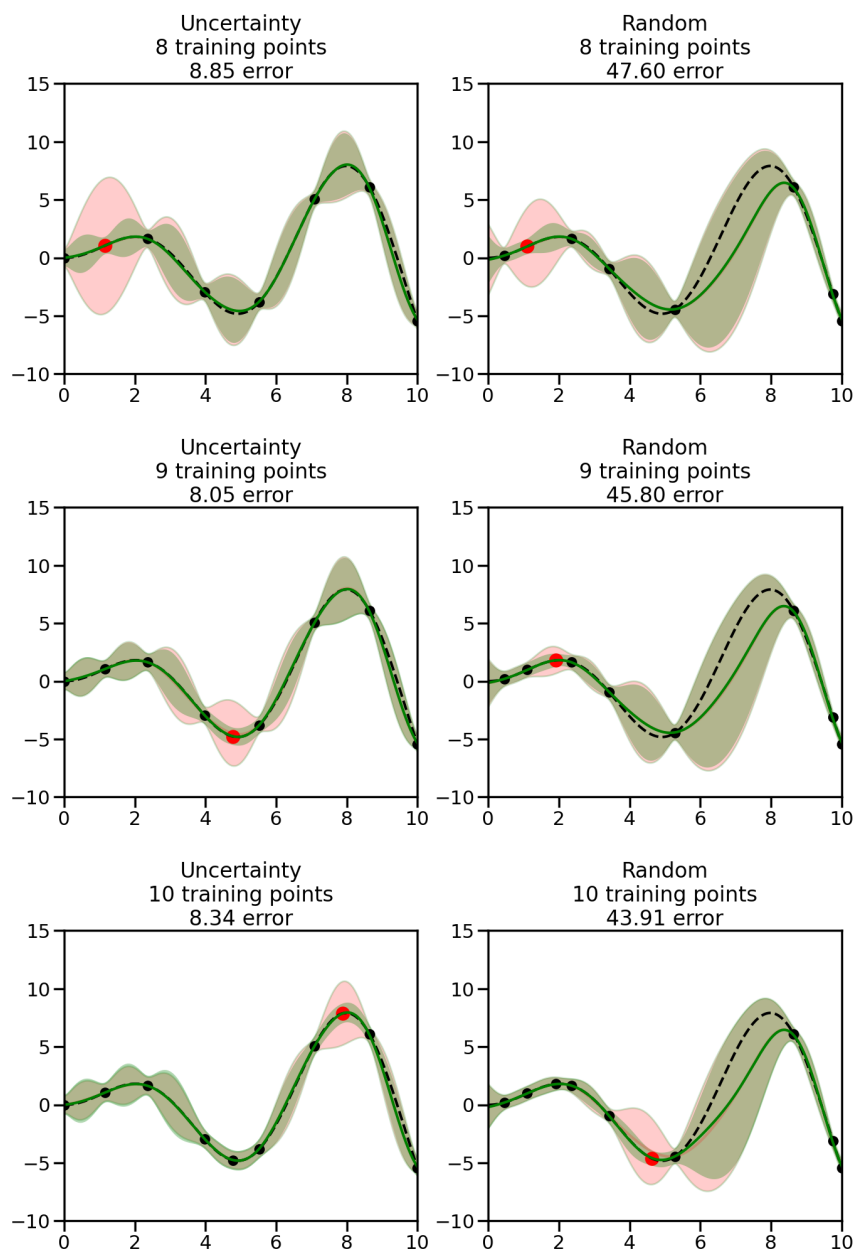
plt.scatter(X[train_ind[o],:],y[train_ind[o]],color='k',s=100)
# Highlight the point we just added
plt.scatter(X[next_ind,:].flatten(),y[next_ind].flatten(),color='r',s=200)
plt.ylim([-10,15]) # Set the y-axis limits
plt.xlim([0,10]) # Set the x-axis limits
# Label the figure title with the number of total training points
# and the total error between the predicted and actual curves
n_train = np.count_nonzero(train_ind[o])
error = np.linalg.norm(yp-f(x).flatten())
errors[o][i]=error
plt.title("%s\n%d training points\n%.2f error"%(o,n_train,error))
plt.show()
return errors

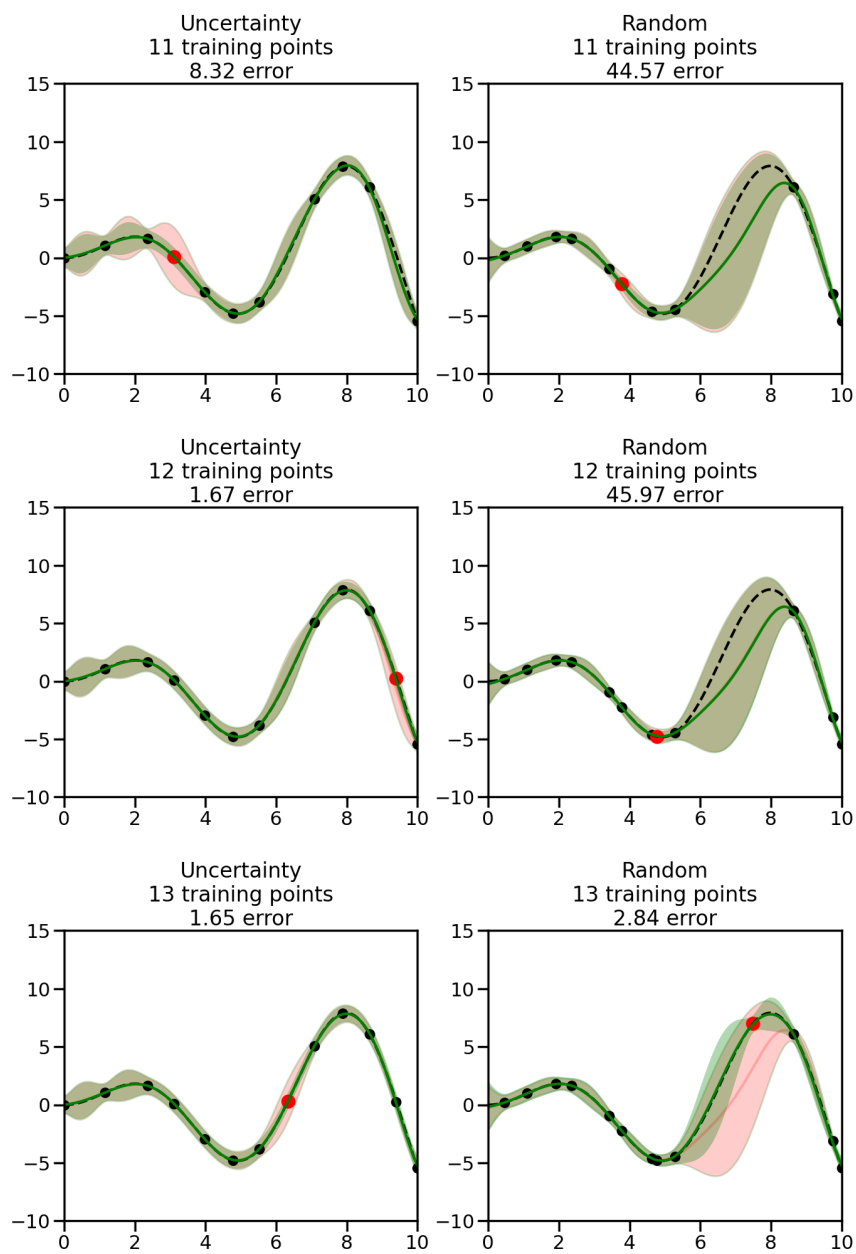
```

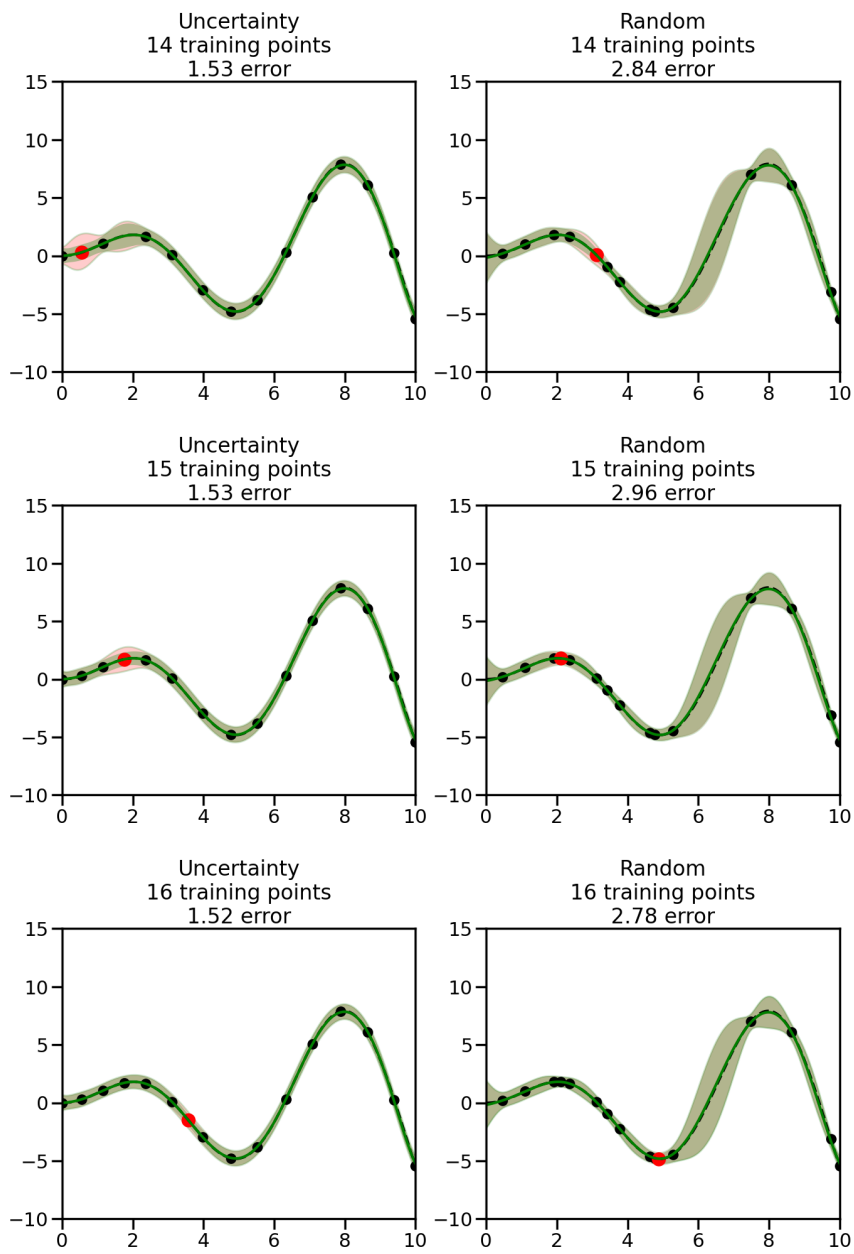
```
errors = sample_data_points(X, y, gp, num_points=35)
```

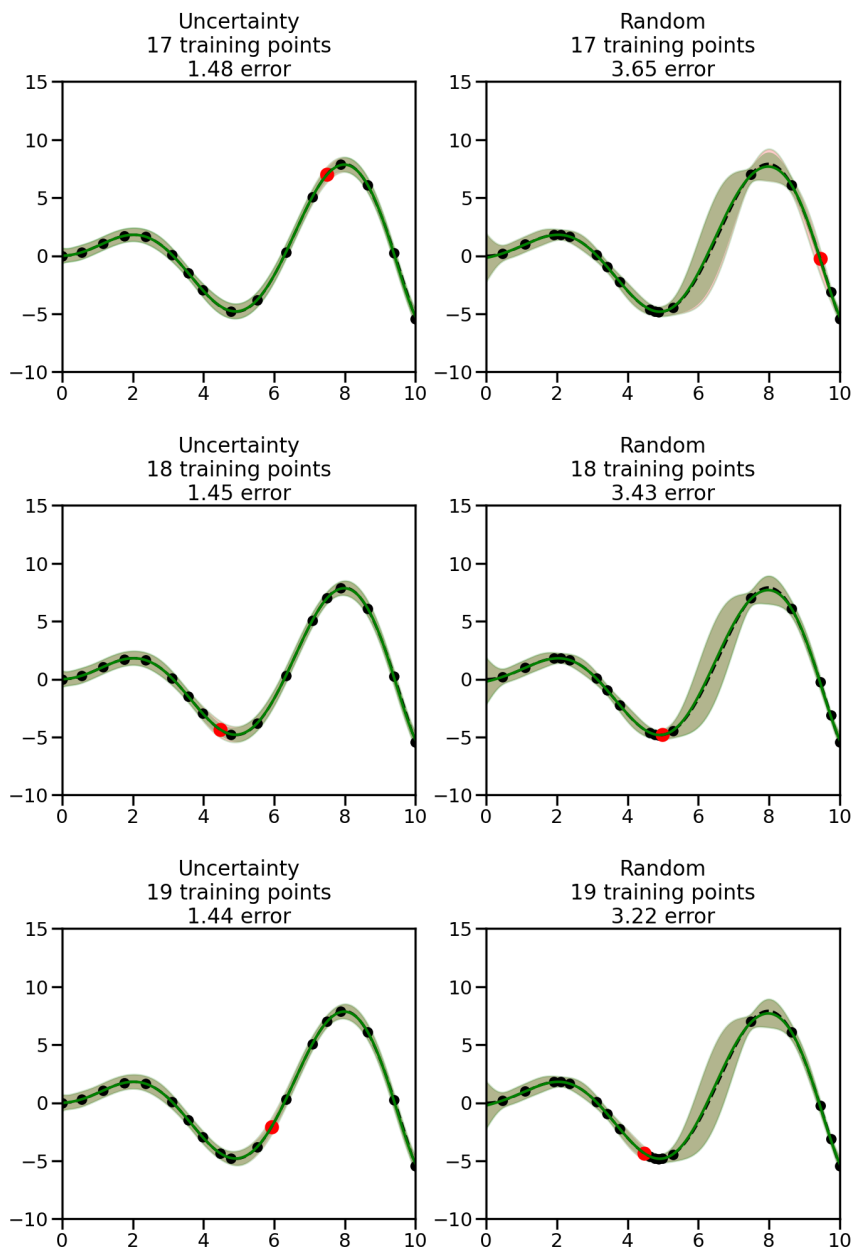


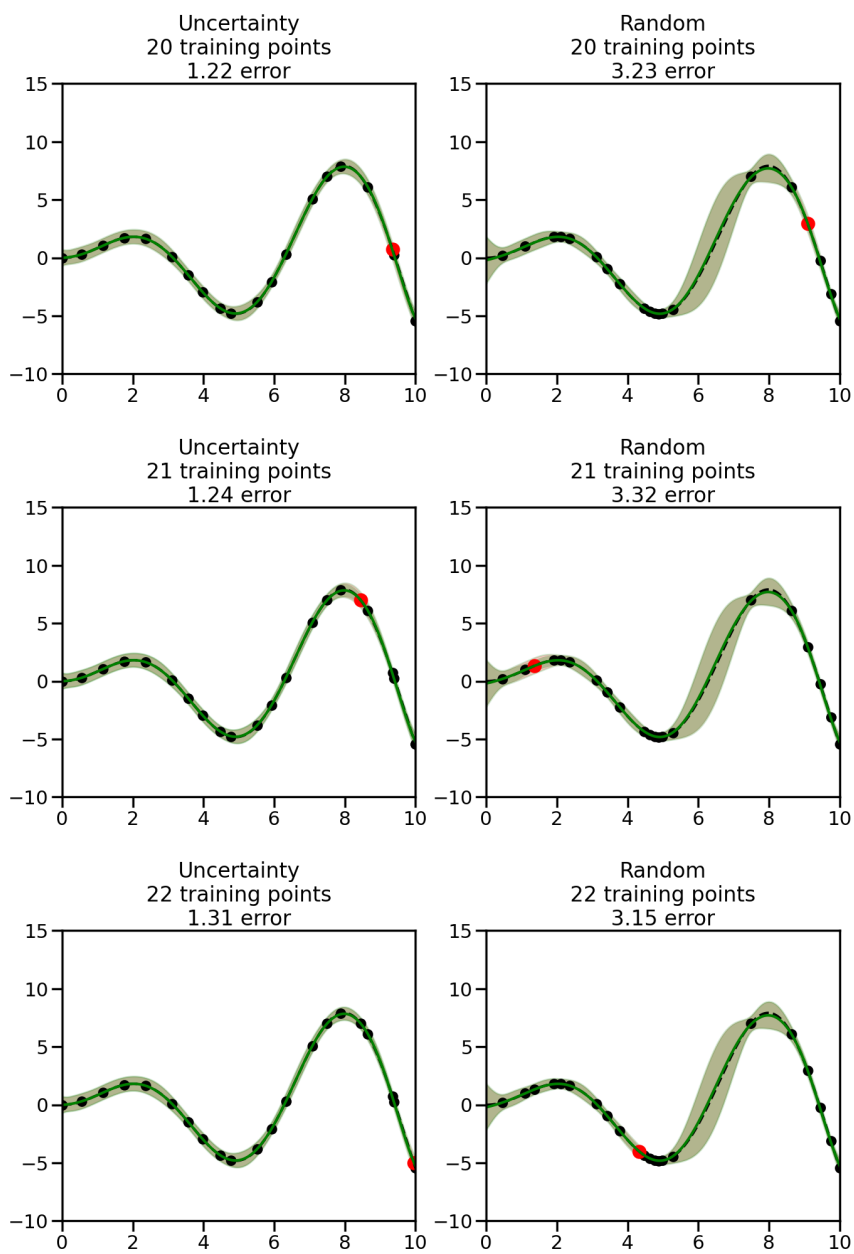


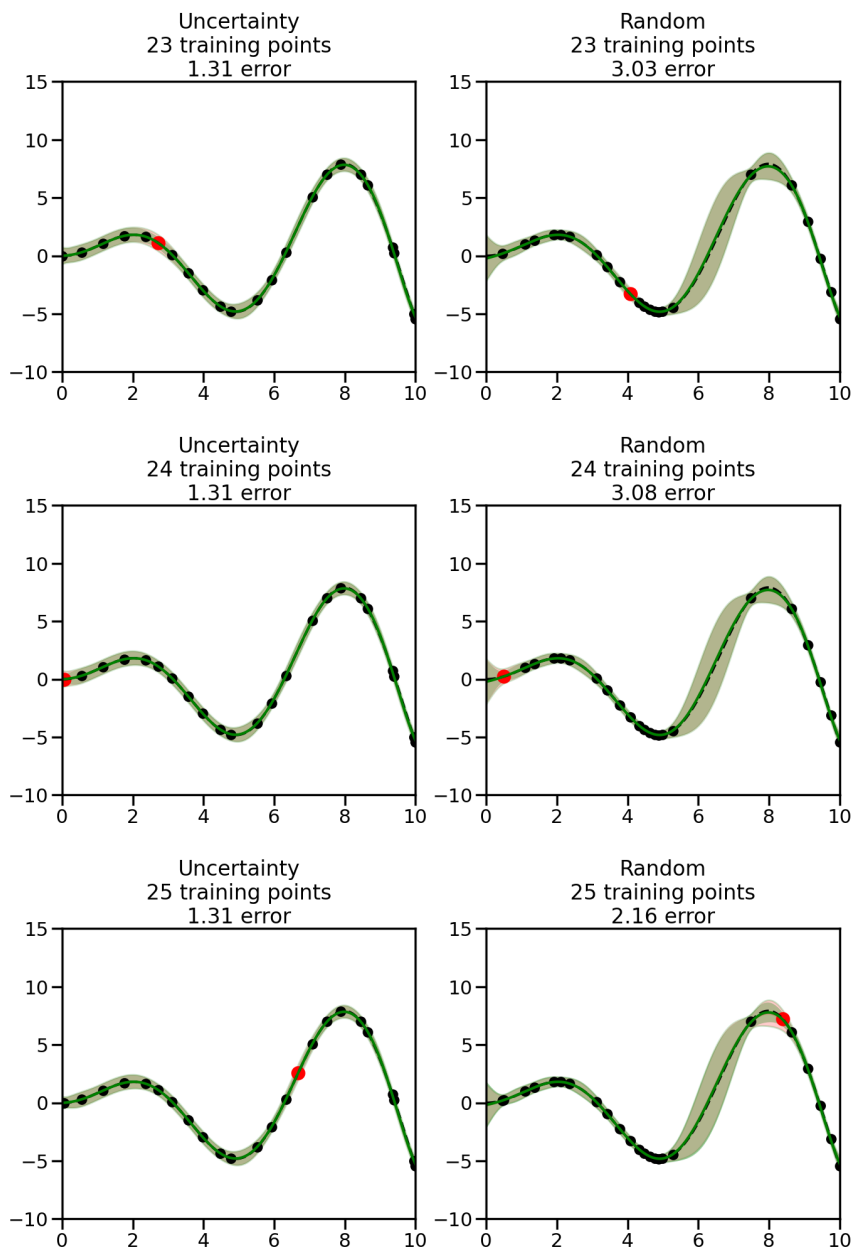


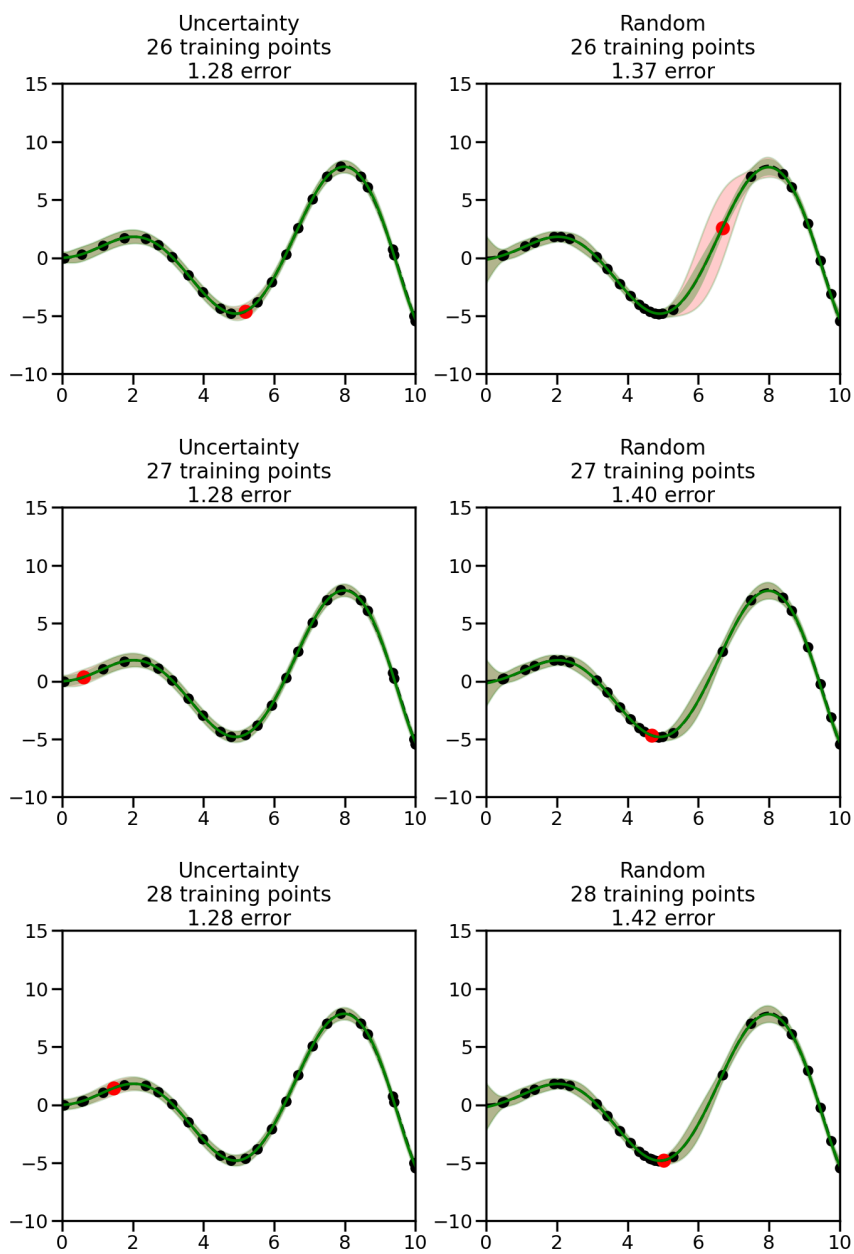


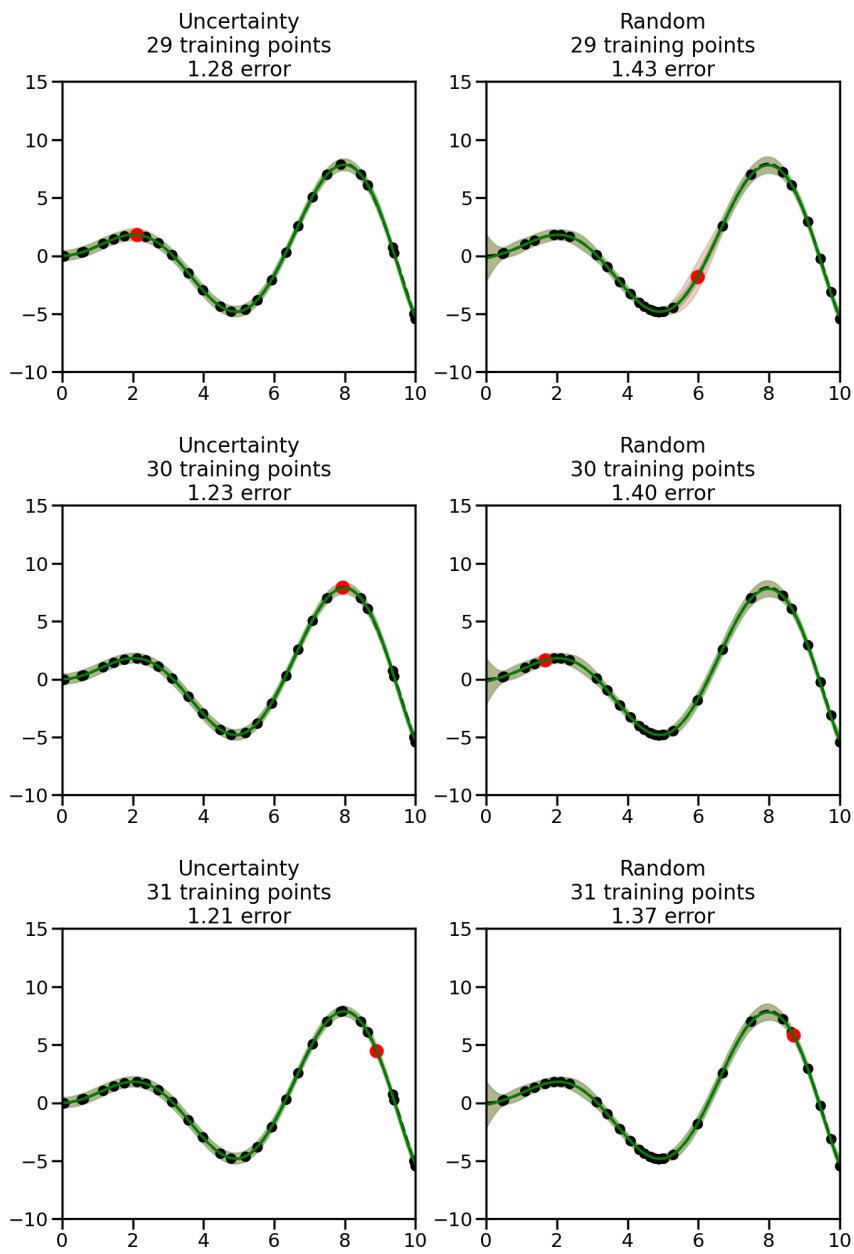


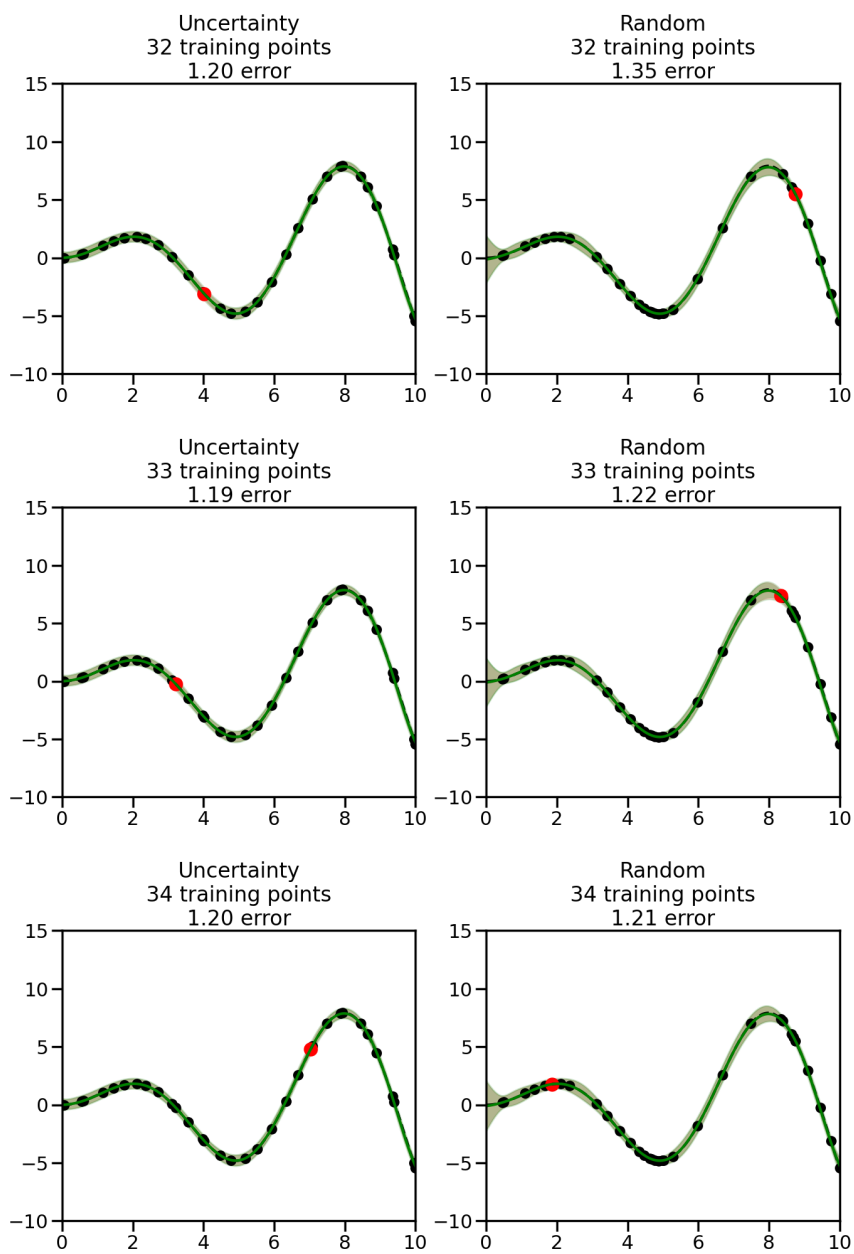


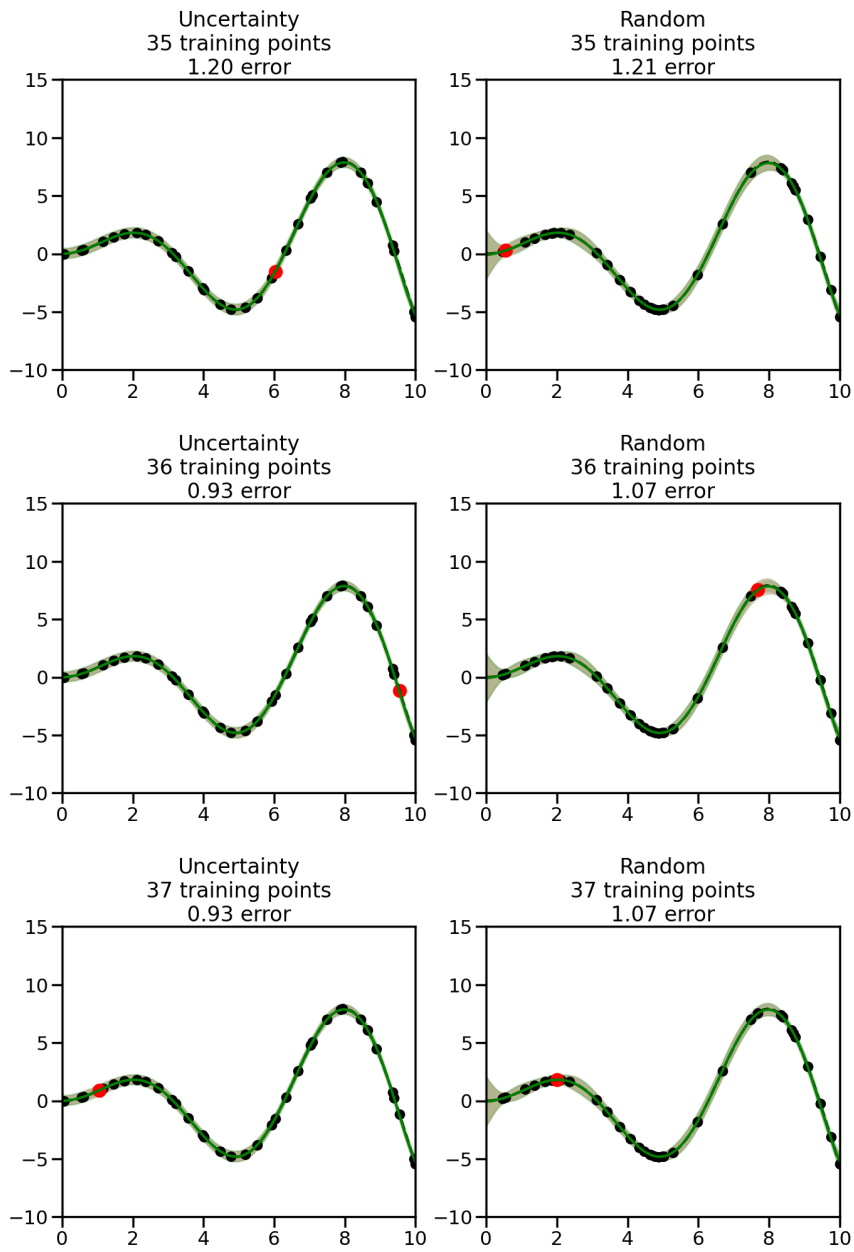




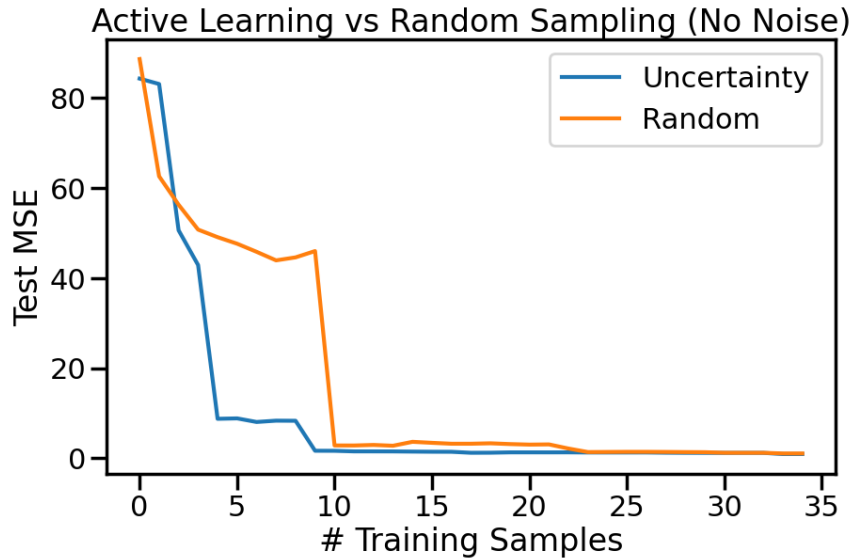






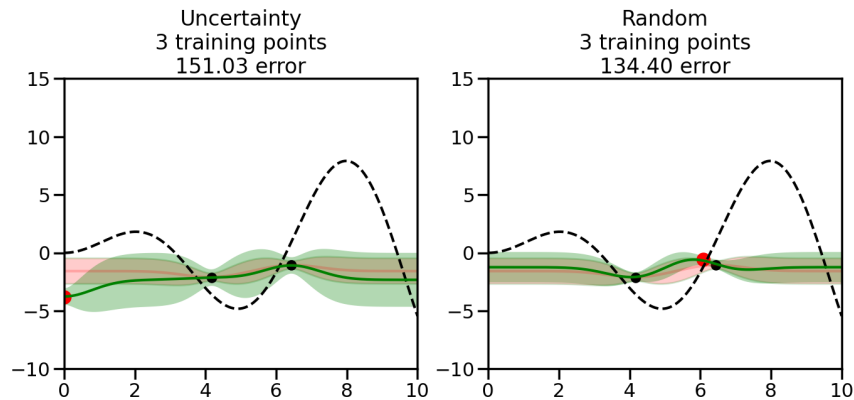


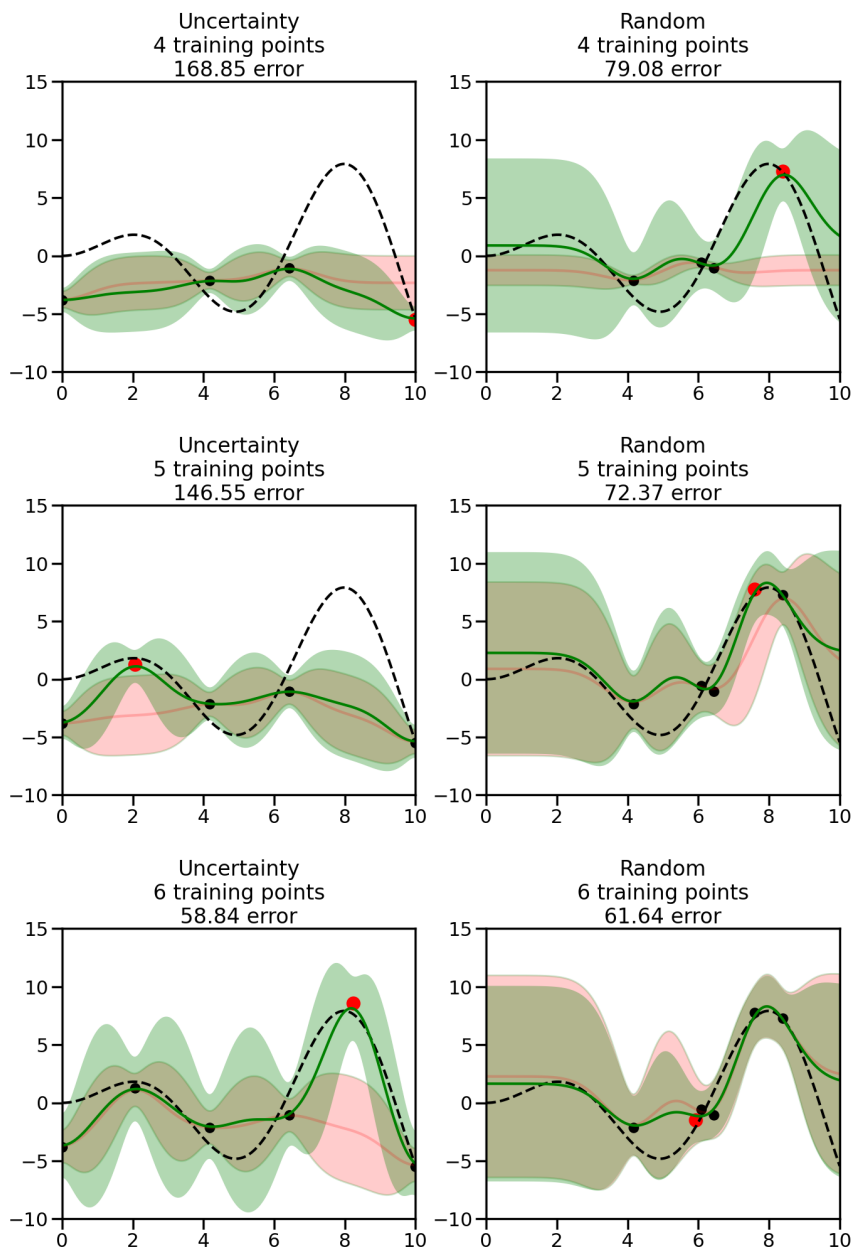
We can also plot the overall Test MSE as a function of the number of labeled datapoints added, to see how quickly each strategy improves the model's performance:

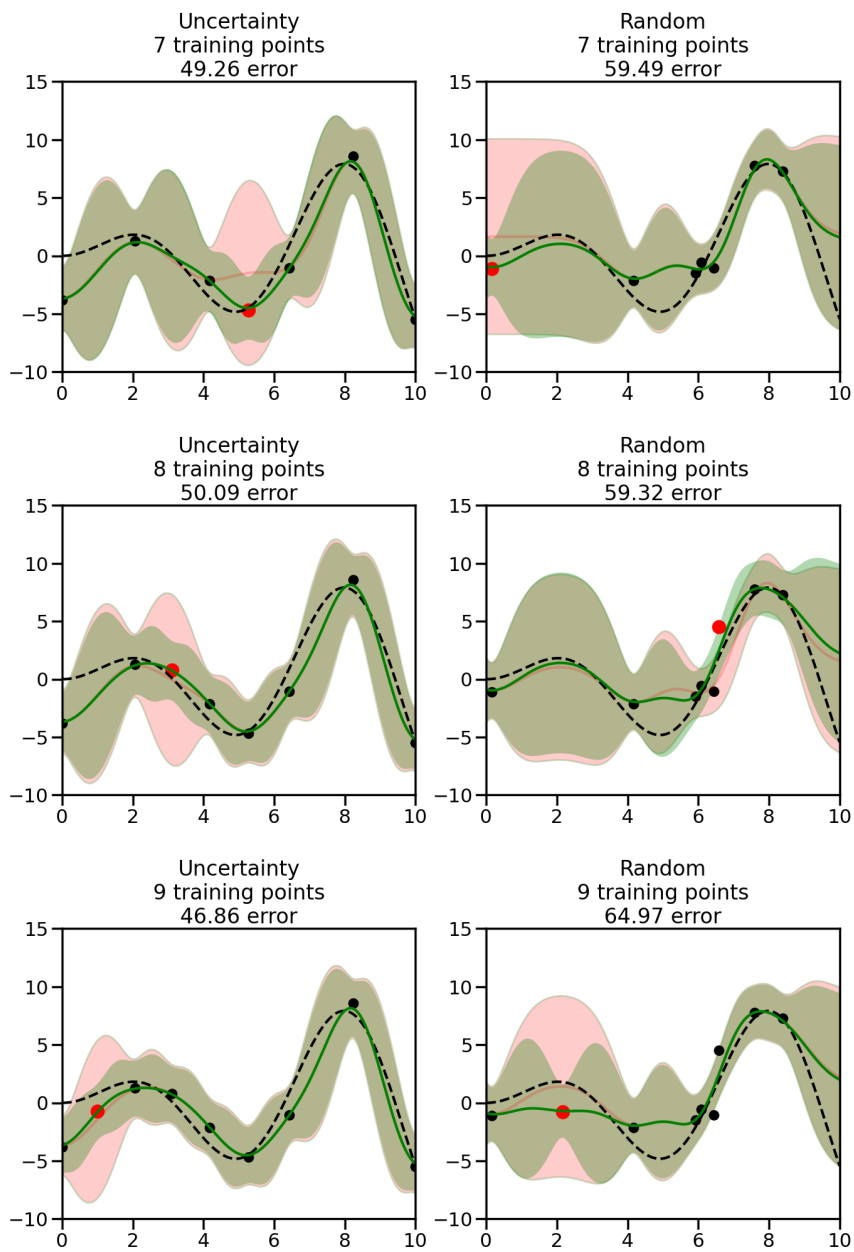


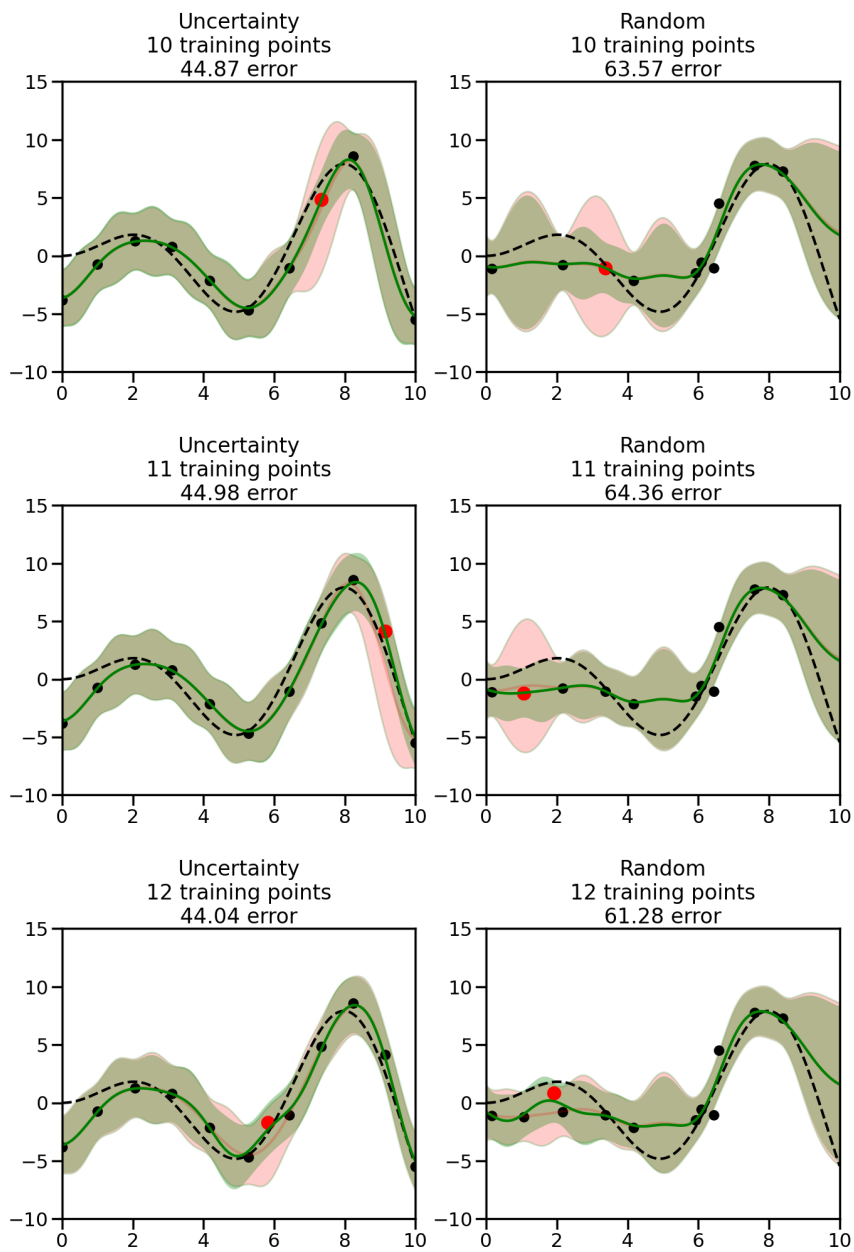
22.4.1 Uncertainty Sampling under Noise

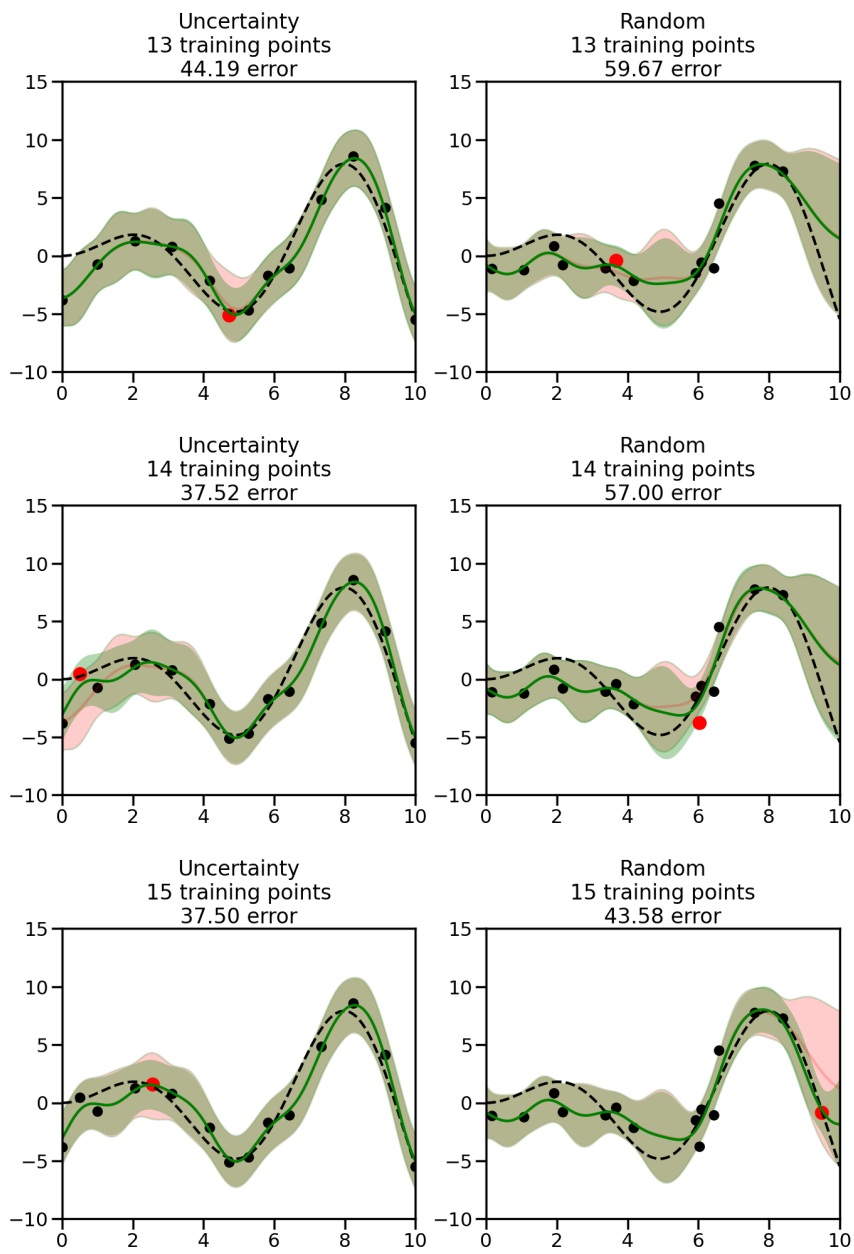
While the above example shows the effectiveness of uncertainty sampling in a noise-free scenario, real-world data is often noisy. To simulate this, we can add Gaussian noise to our observations. This will help us understand how uncertainty sampling performs when the data is not perfectly clean. Intuitively, we expect that uncertainty sampling will still be effective, but the presence of noise may lead to selecting points that are not as informative as in the noise-free case, and that as the noise increases the benefits we accrue from Active Learning may diminish. We can modify the previous example code to include Gaussian noise in the observations and observe how the model's performance changes with uncertainty sampling compared to random sampling.

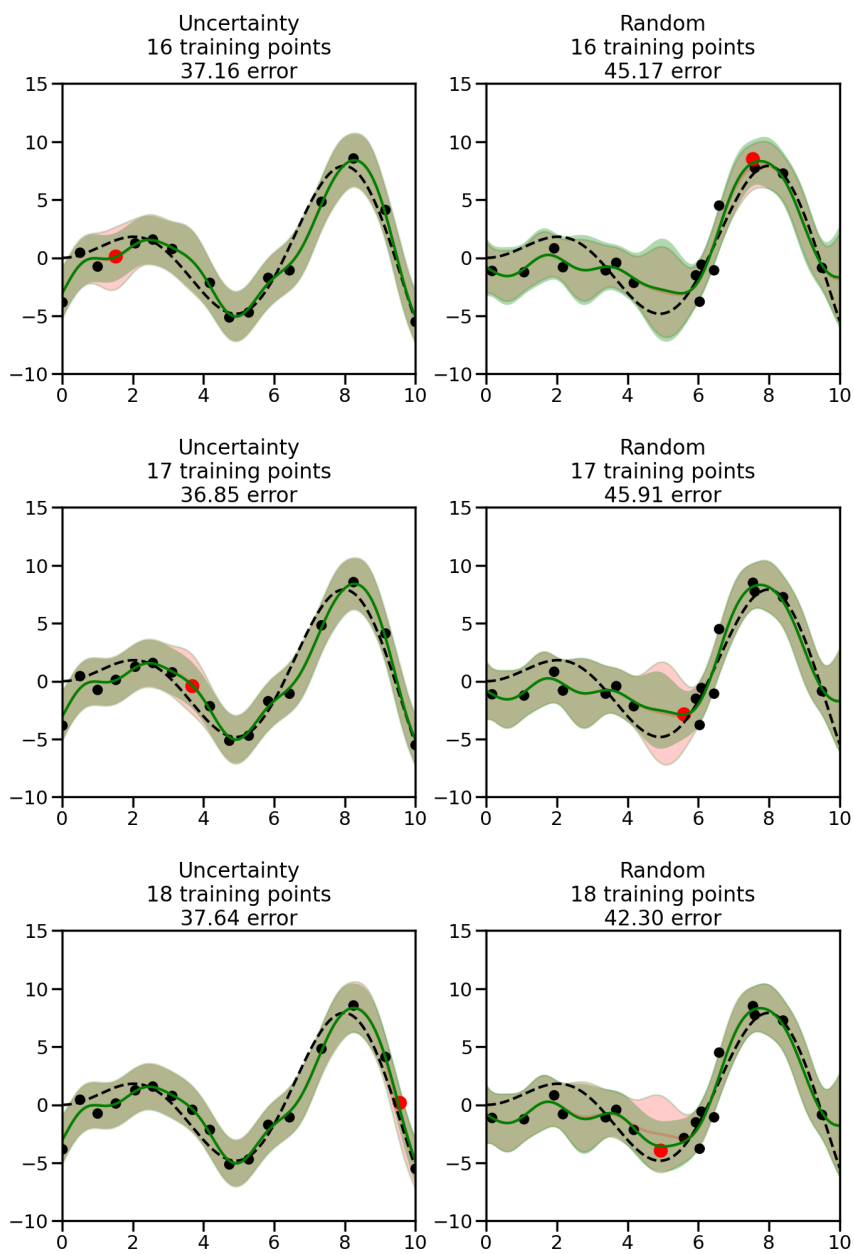


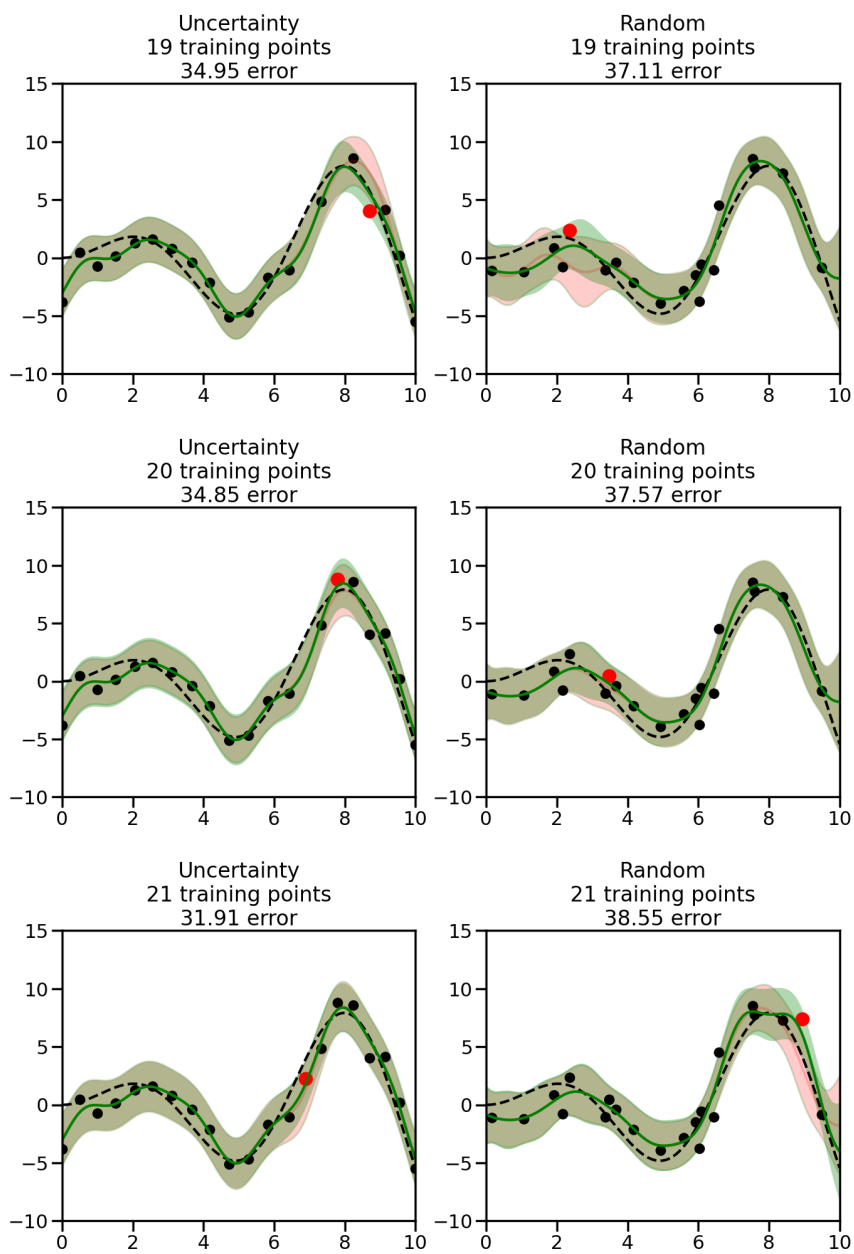


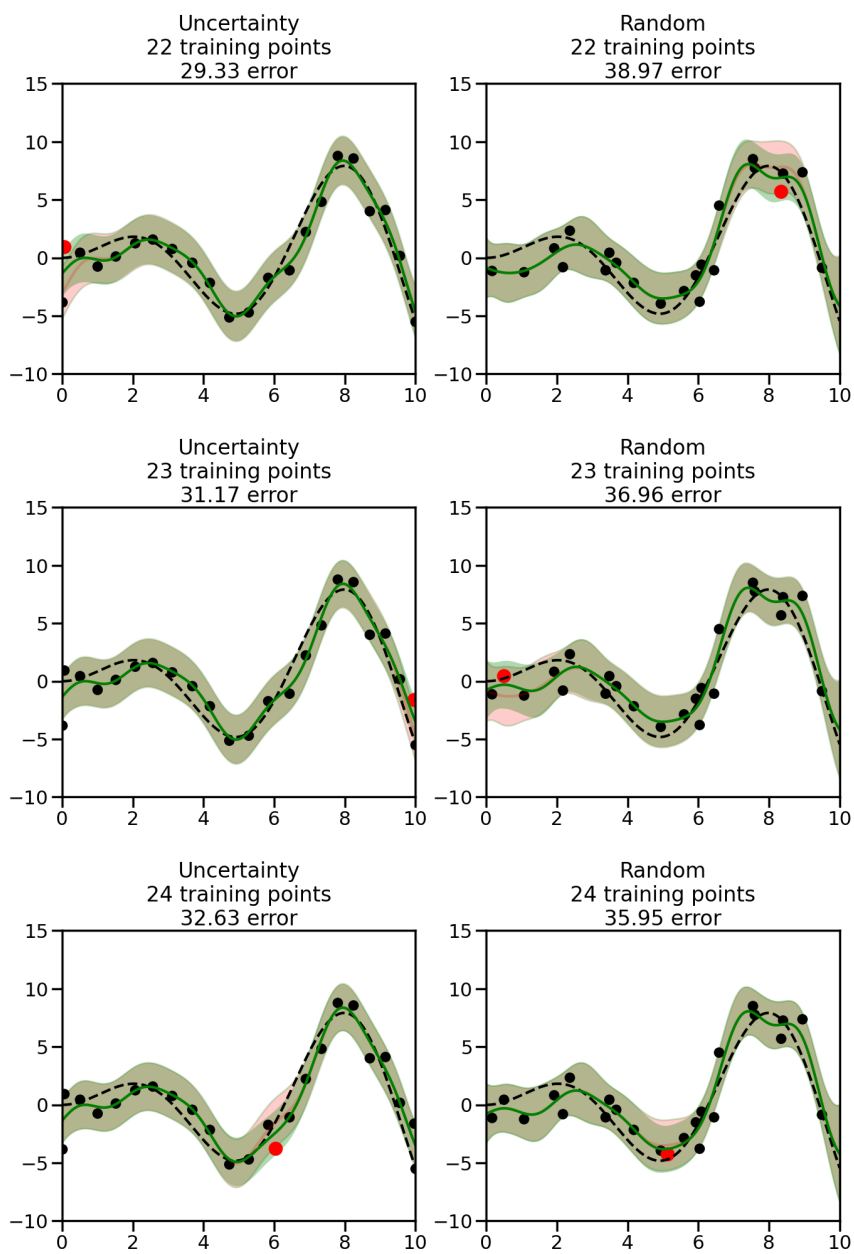


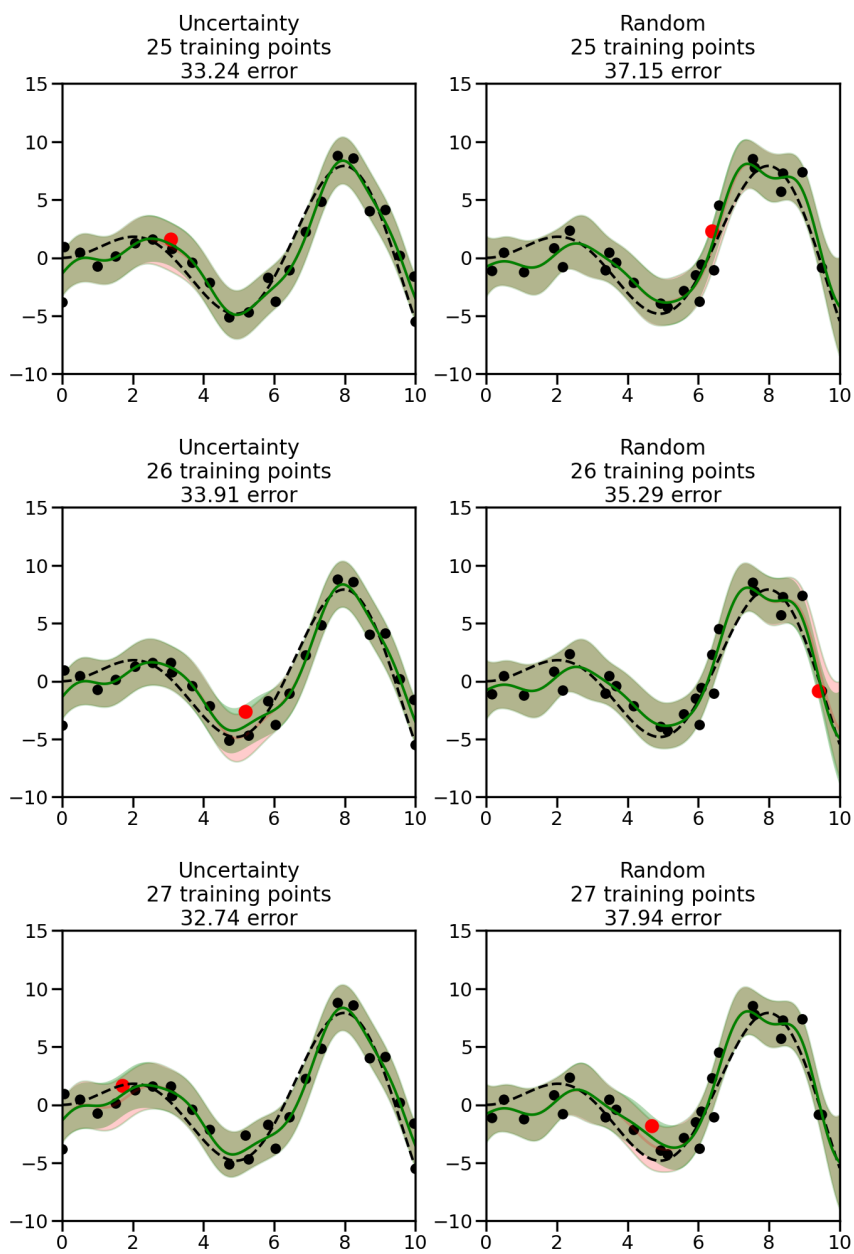


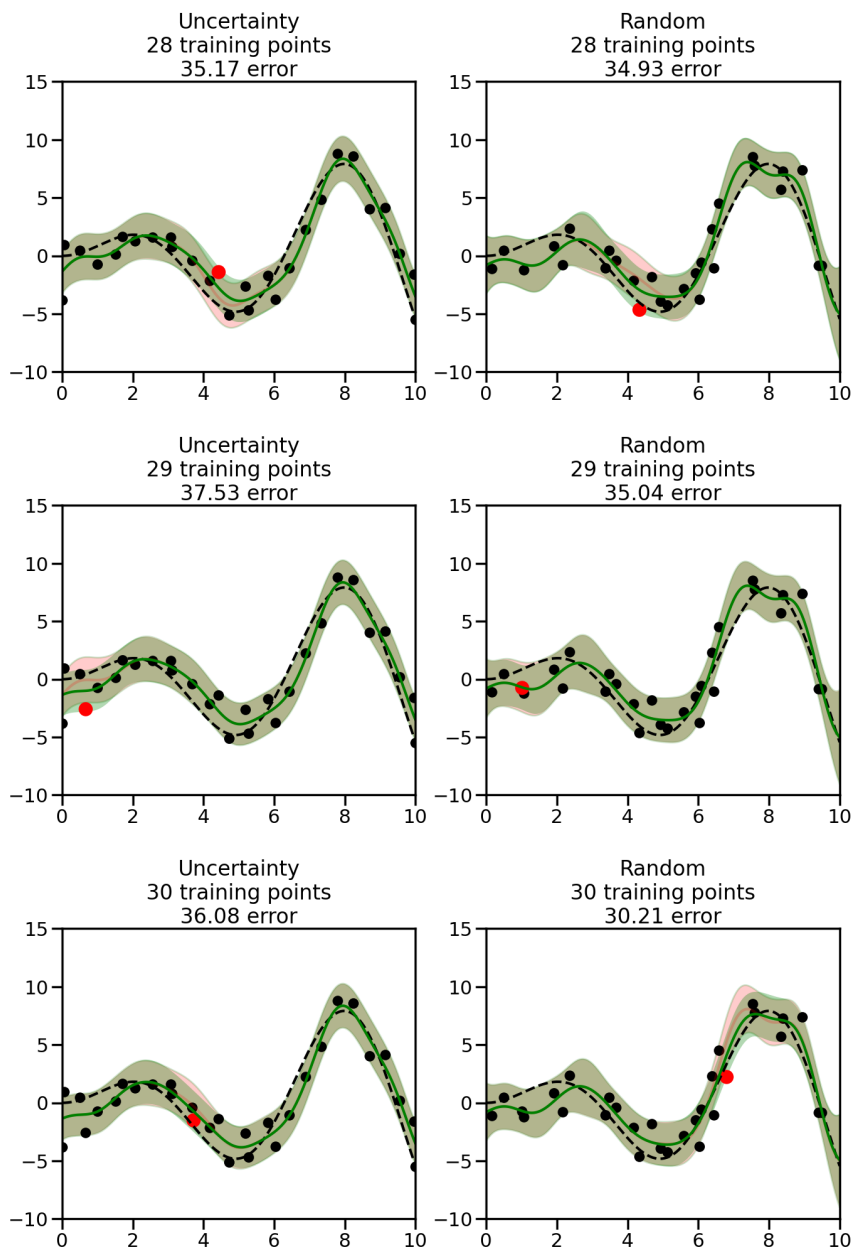


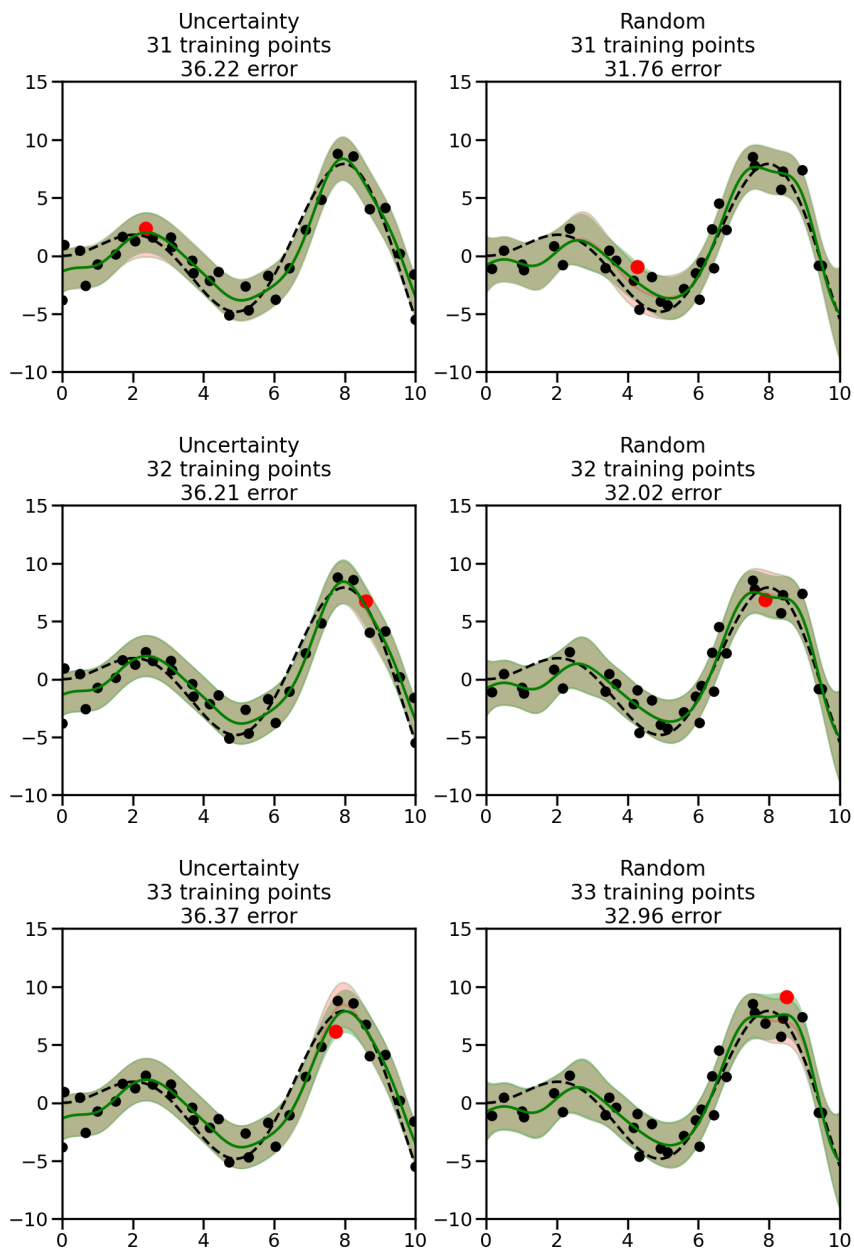


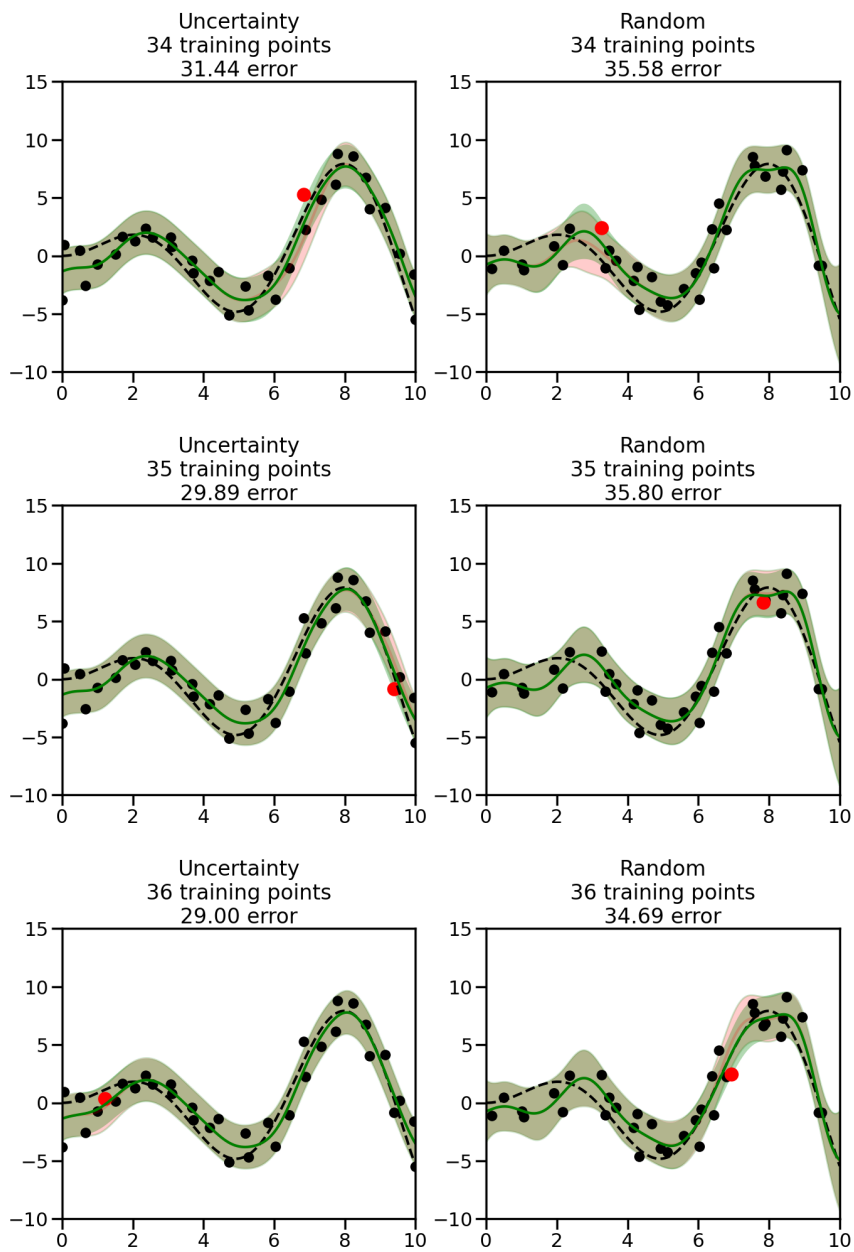


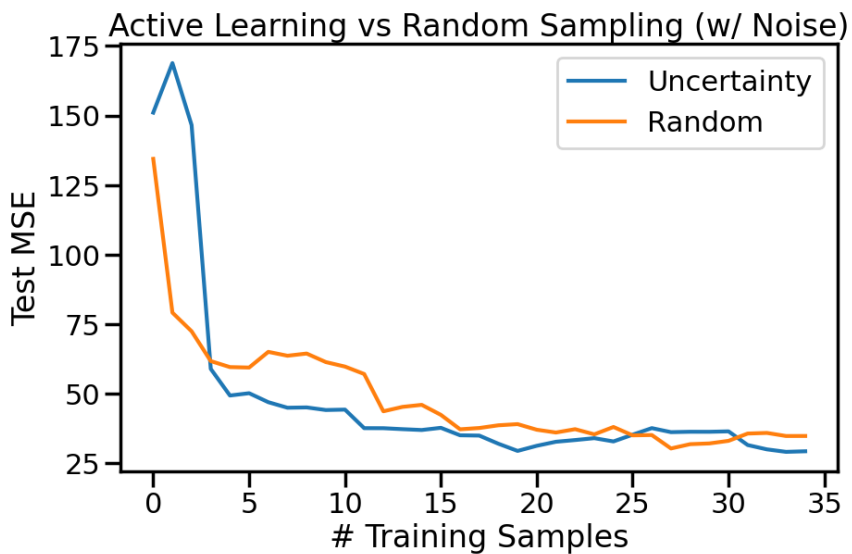
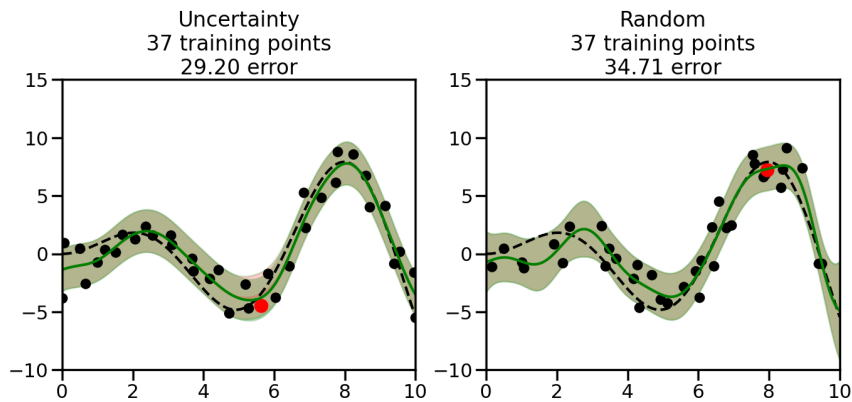












💡 Experiment: Comparison of Uncertainty and Random Sampling

Go to the above cells and re-run the model with different levels of noise and different RBF length scales. Observe how the performance of Uncertainty Sampling compares to Random Sampling in each case. Consider the following questions:

- What happens to Uncertainty Sampling compared to Random Sampling as we increase the noise in the data?
- What happens to Uncertainty Sampling compared to Random as we decrease the RBF length scale (i.e., allow more high frequency function behavior)?

22.5 Uncertainty Sampling of Classification Boundaries

We can apply the same ideas behind uncertainty sampling to classification problems as well. In this case, instead of using a Gaussian Process regression model, we can use a Gaussian Process classification model (or any other probabilistic classification model that provides uncertainty estimates). The key idea remains the same: we want to select data points for which the model is most uncertain about the predicted class label.

Compare the number of training samples and the Classification Accuracy across the Uncertainty Sampling and Random conditions. You can also try changing the noise parameter for the Moon's dataset and looking at how that affects things.

Pay special attention to which points the algorithms pick out for training, and how the resulting training data look different after 50-ish iterations.

```
# 2-D Non-Linear Classification
# Moons dataset
from sklearn.datasets import make_moons, make_circles
np.random.seed(3) # So we all see the same thing
n_samples = 1000 # Number of data samples

# ----- Try Uncommenting the Below -----
# Dataset 1
# Low Noise
X, y = make_moons(n_samples=n_samples, noise=0.1)

# Higher Noise
# X, y = make_moons(n_samples=n_samples, noise=0.25)

# Dataset 2
X,y = make_circles(n_samples = n_samples, factor=0.7, noise = .05)

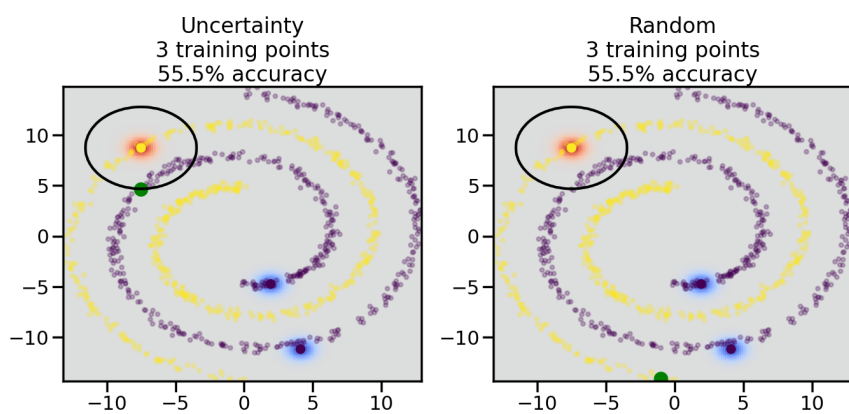
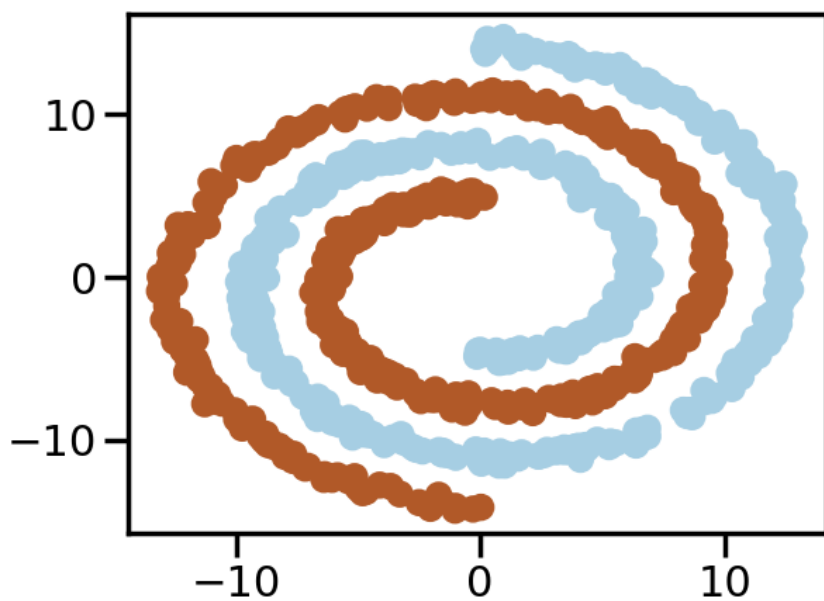
# Dataset 3
X,y = roll_it(n_samples,0.05*5)

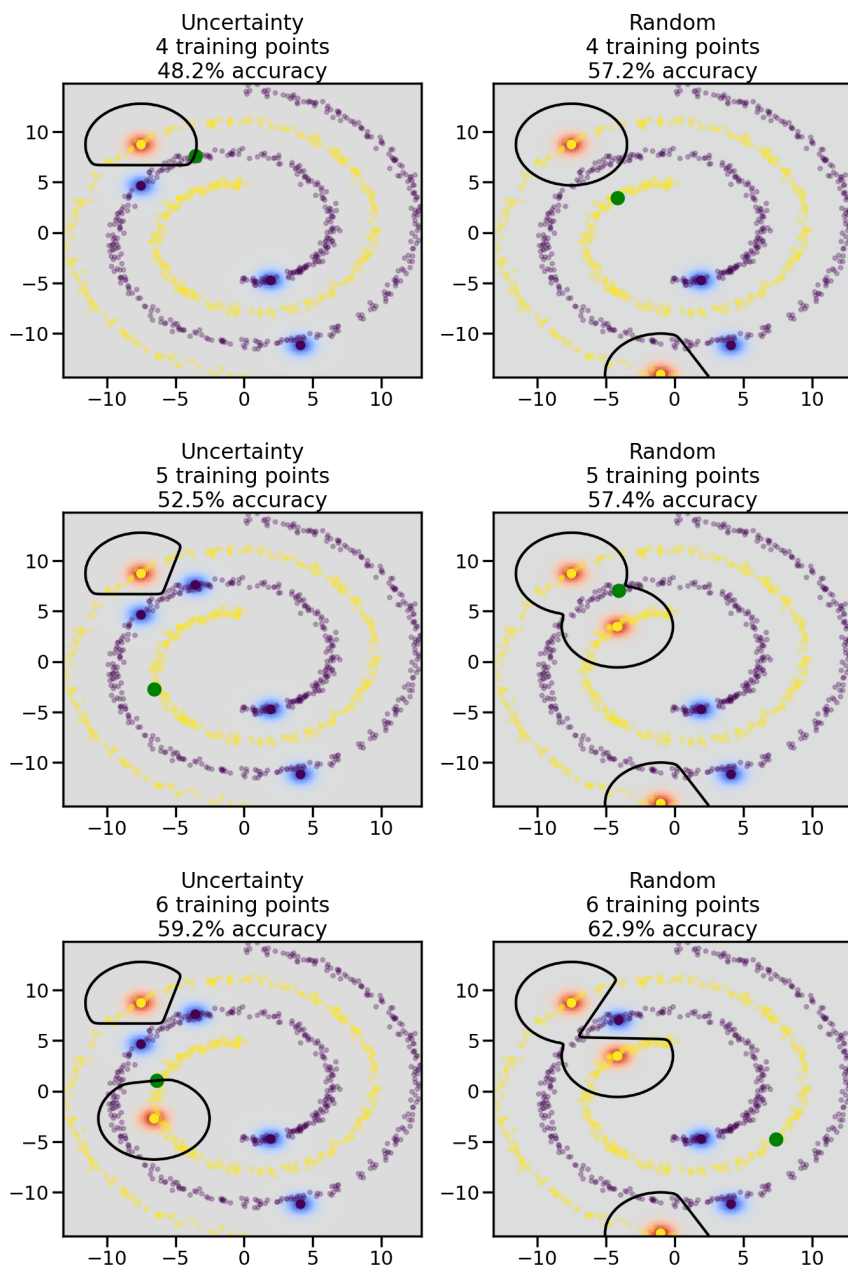
# Dataset 4
# X, y = make_moons(n_samples=n_samples, noise=0.1)
# Xt=np.random.multivariate_normal([-0.7,1.5],np.diag([0.01,0.01]),size=20)
# X = np.r_[X,Xt]
# y = np.r_[y,np.ones(len(Xt))]
```

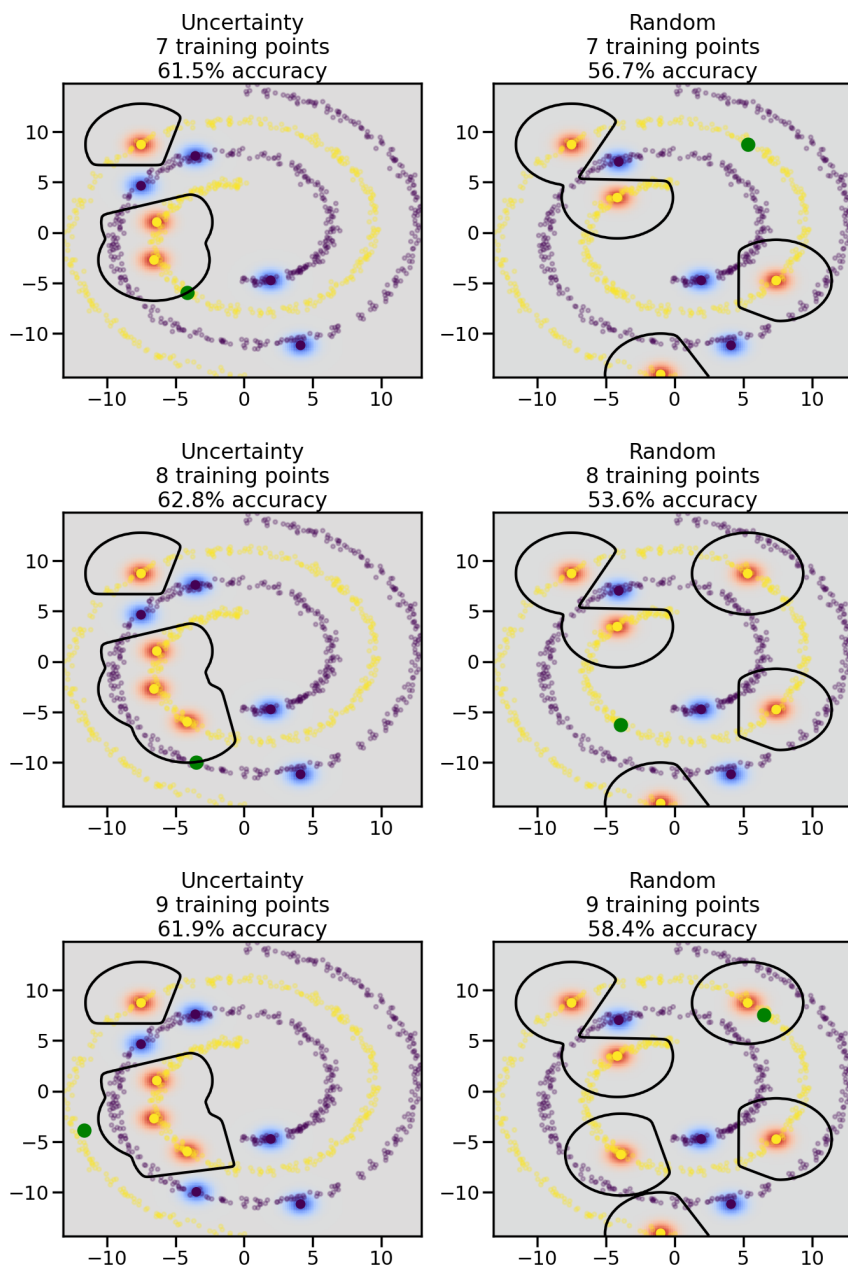
```
# -----
```

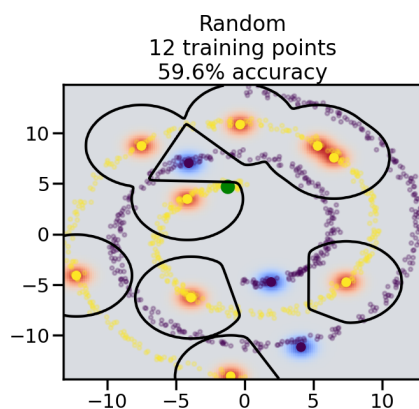
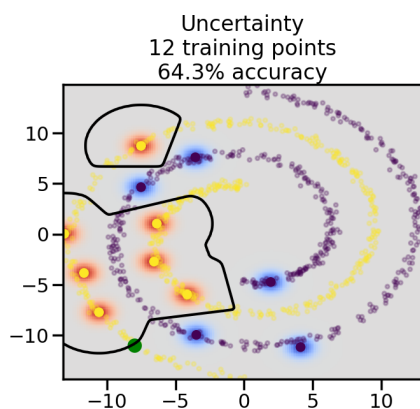
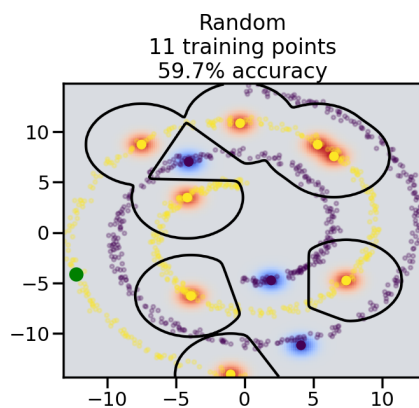
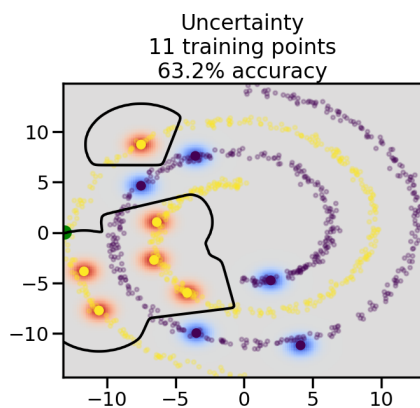
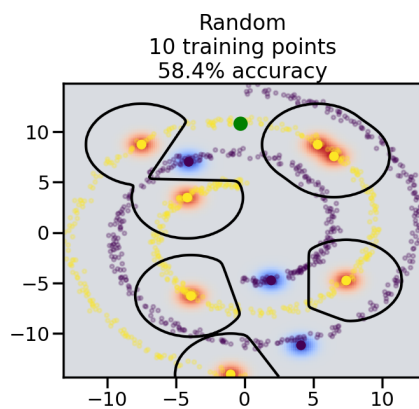
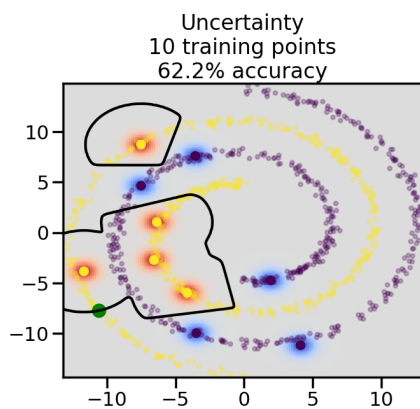


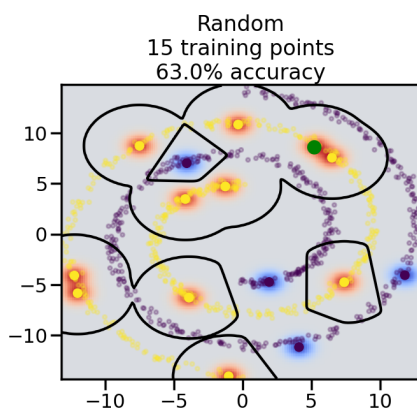
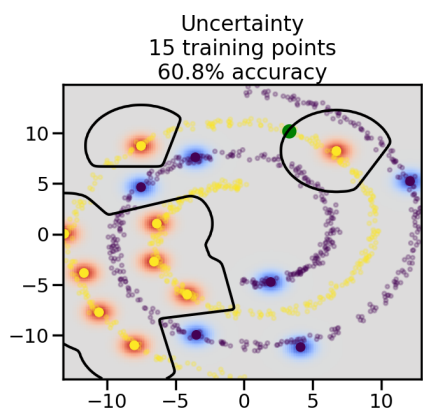
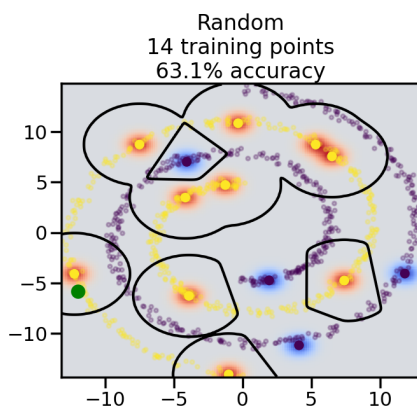
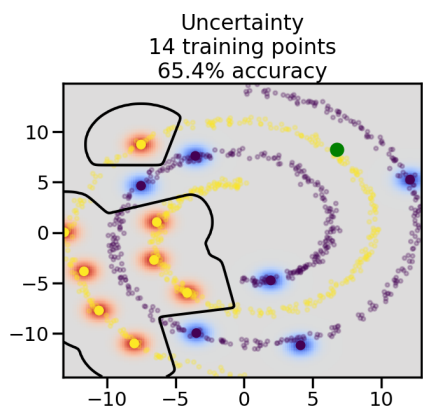
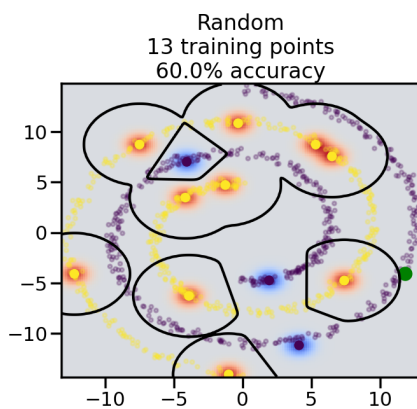
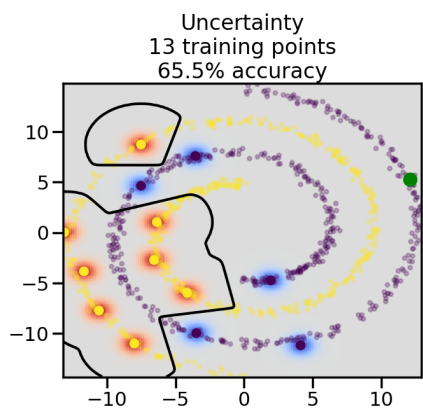
```
plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
plt.axis('tight')
plt.show()
```

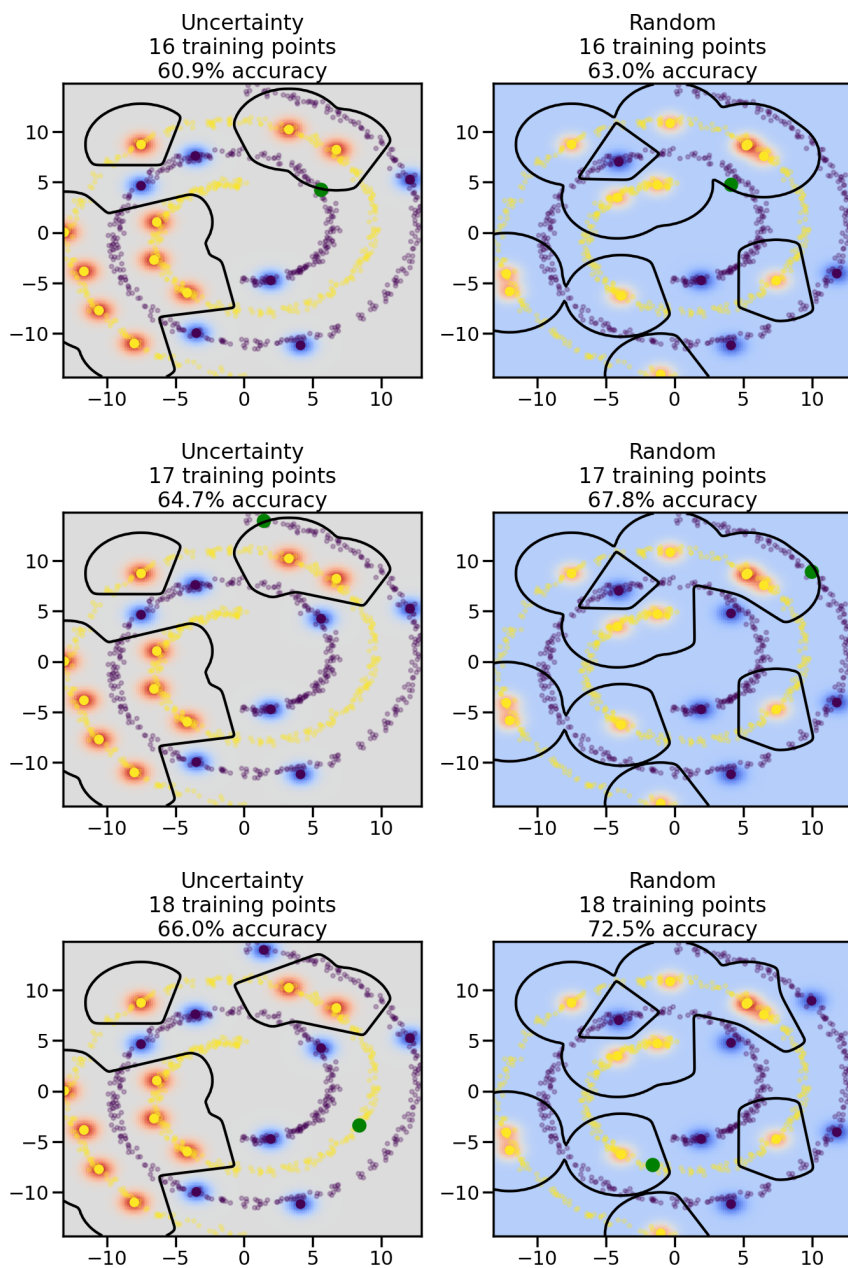


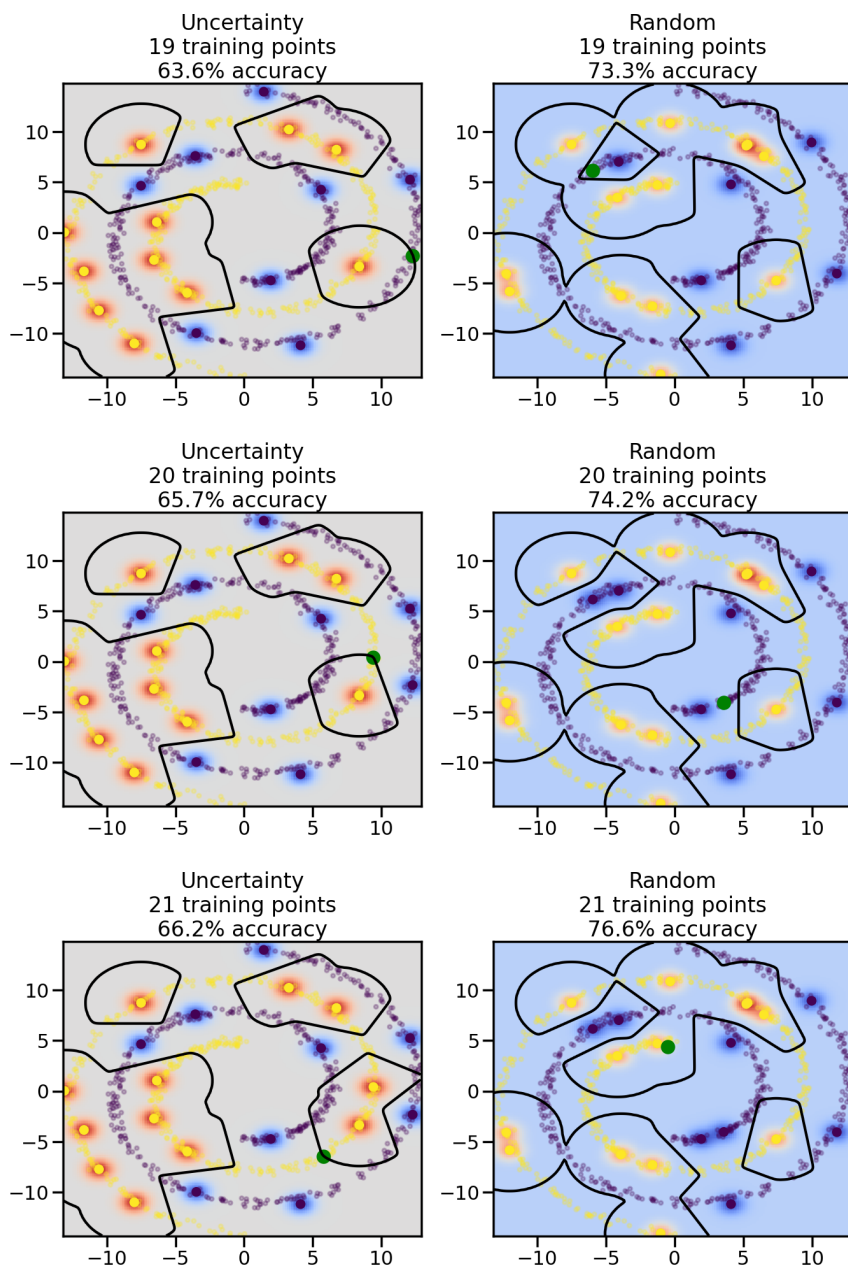


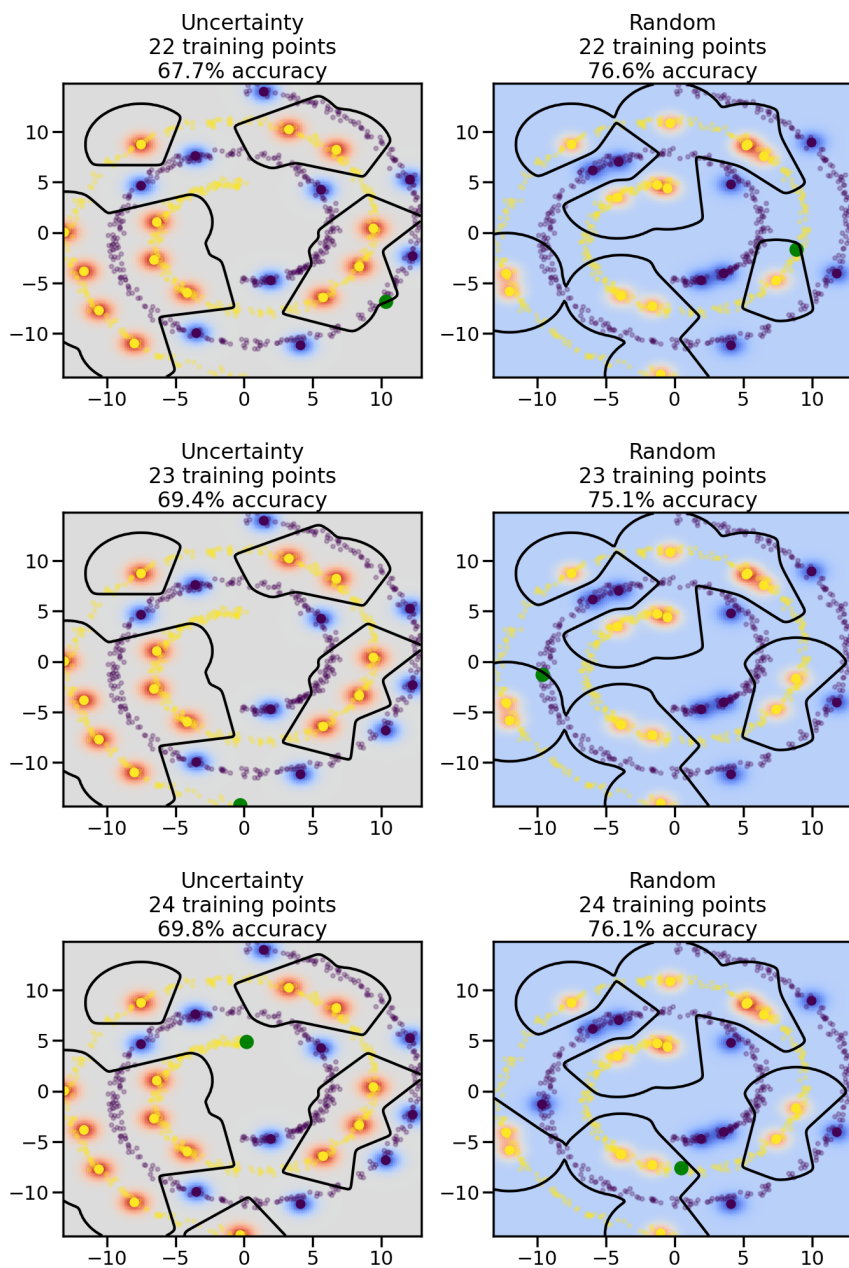


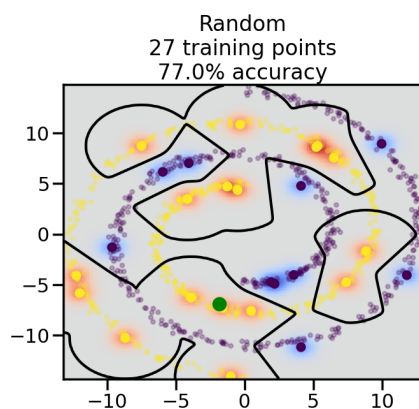
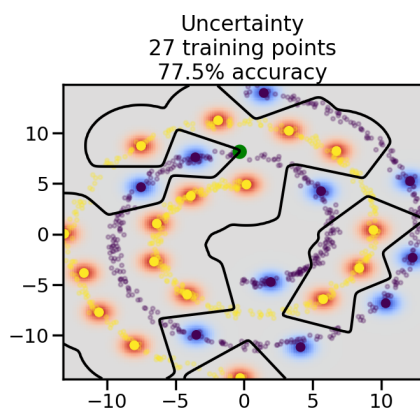
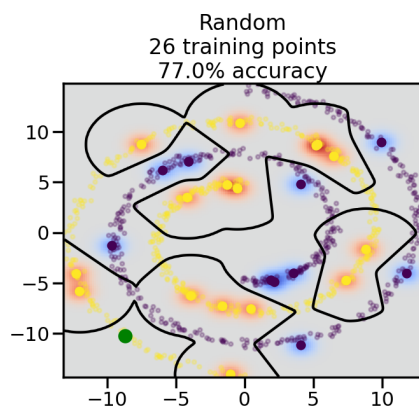
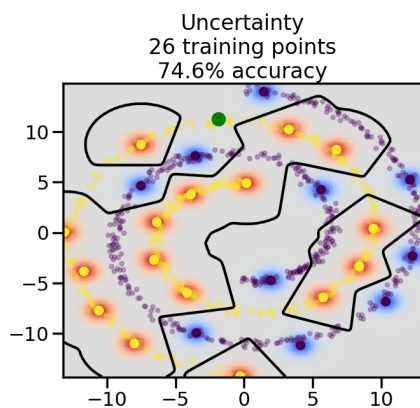
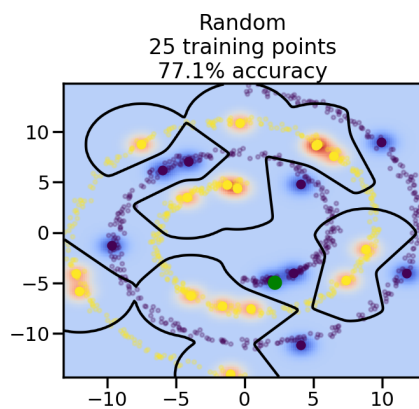
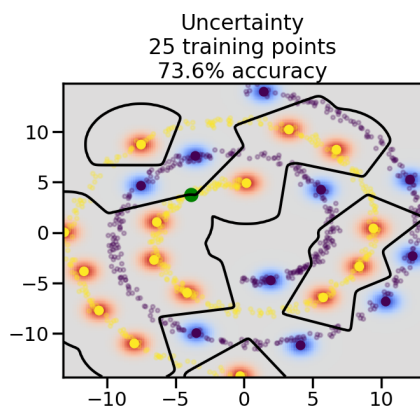


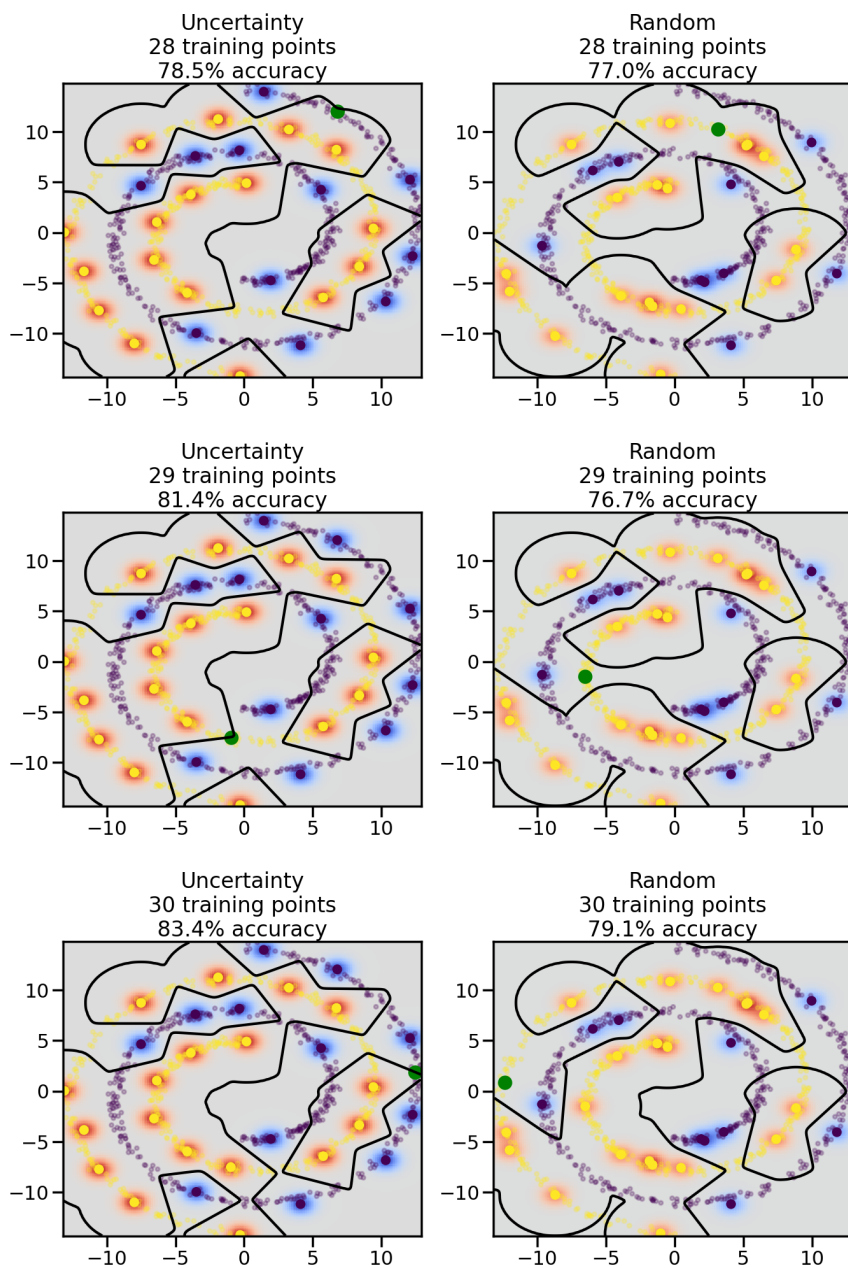


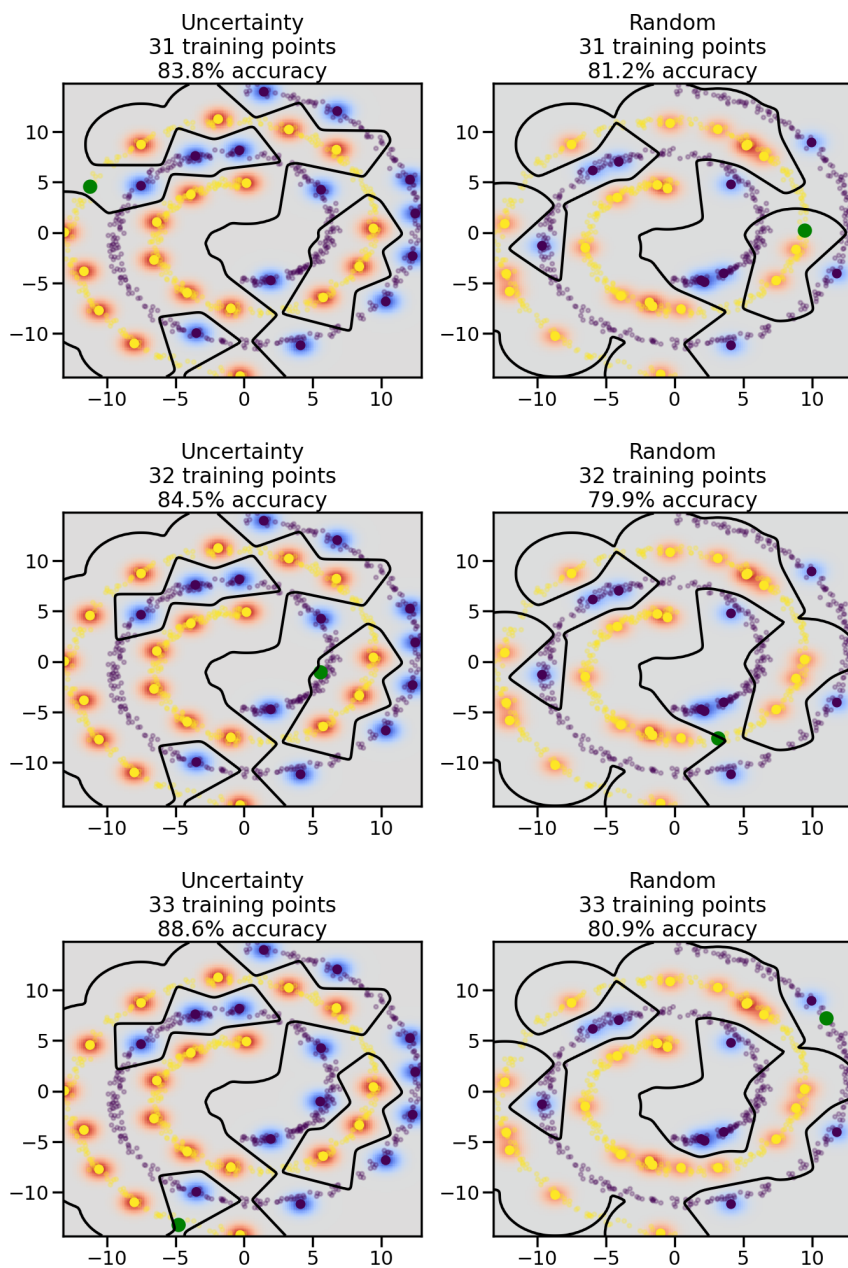


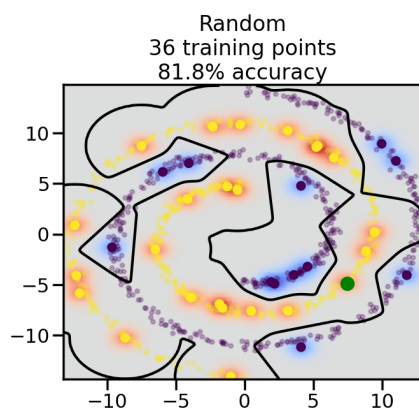
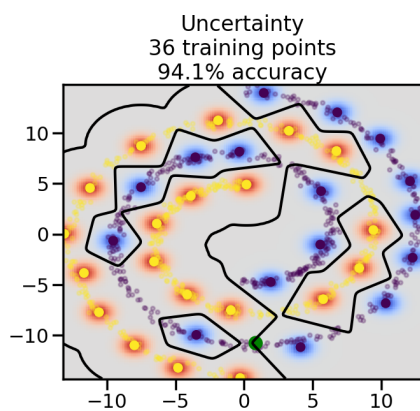
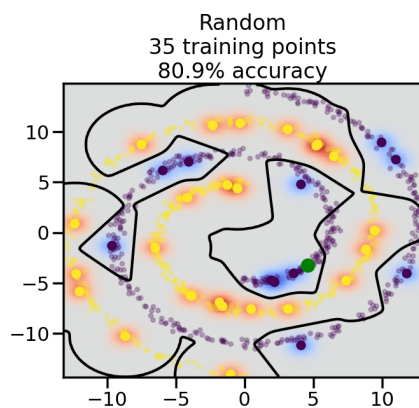
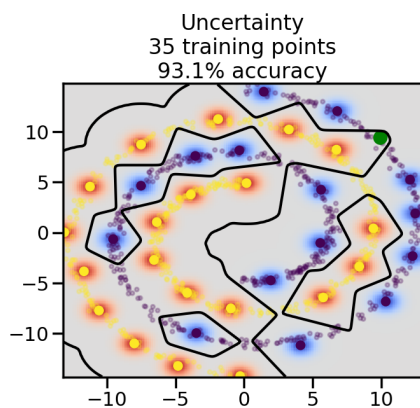
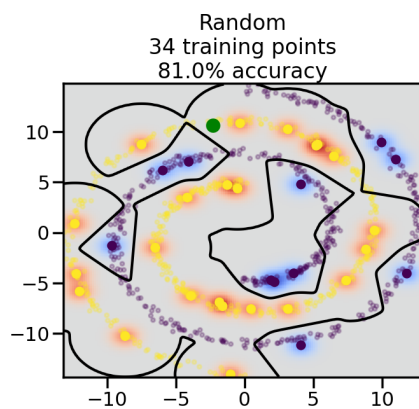
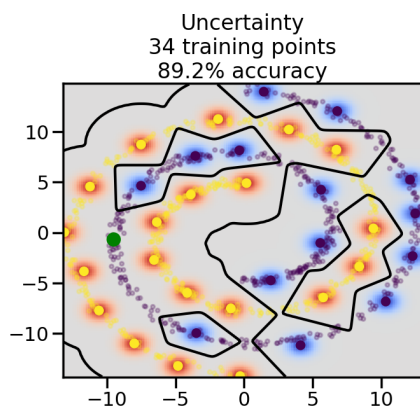


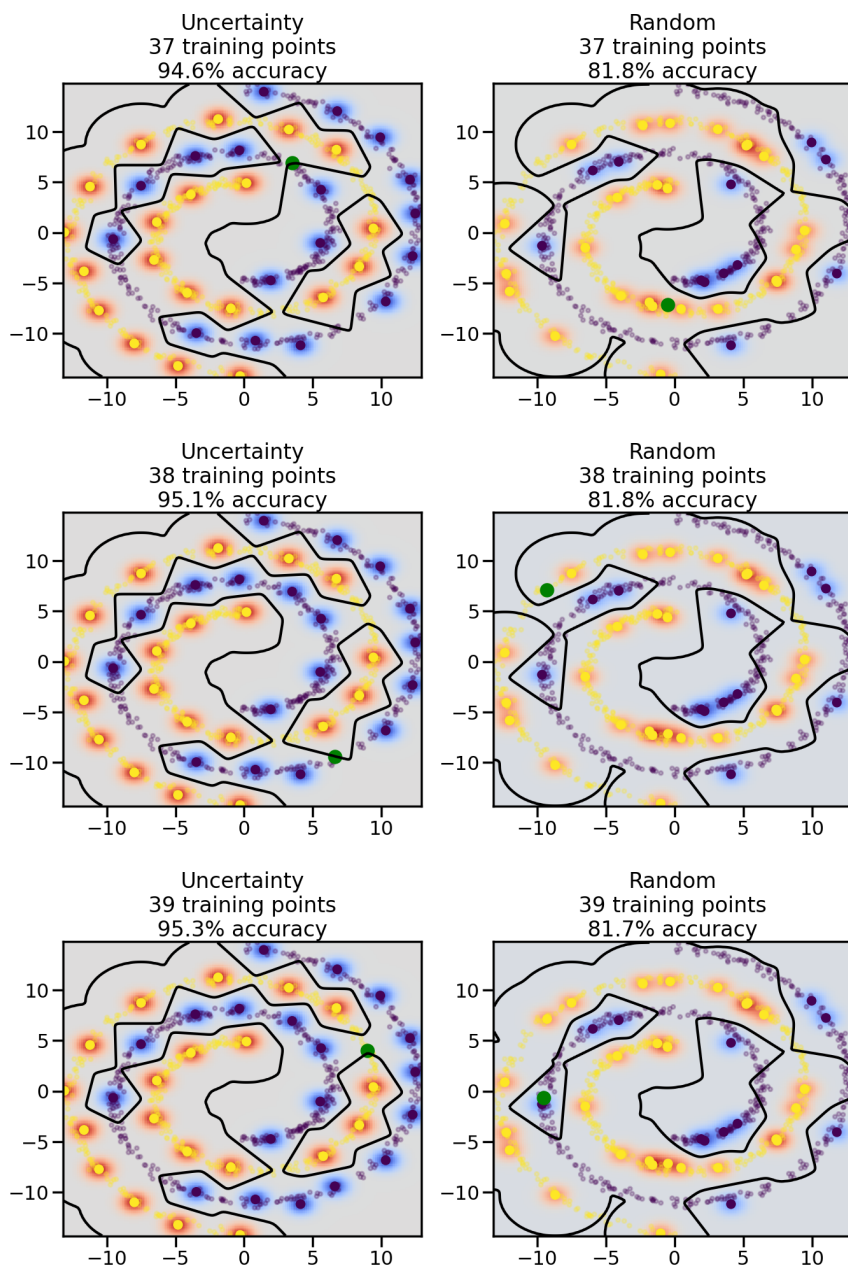


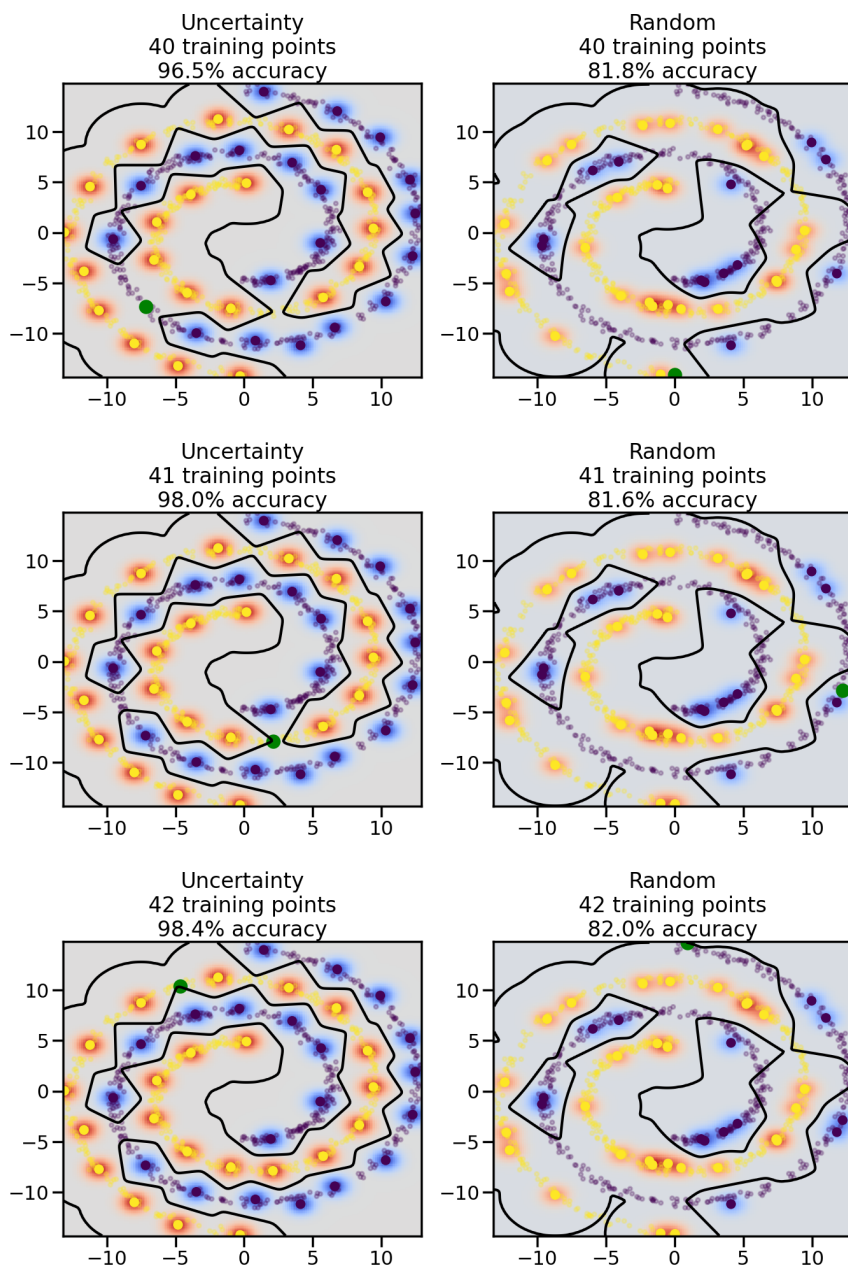


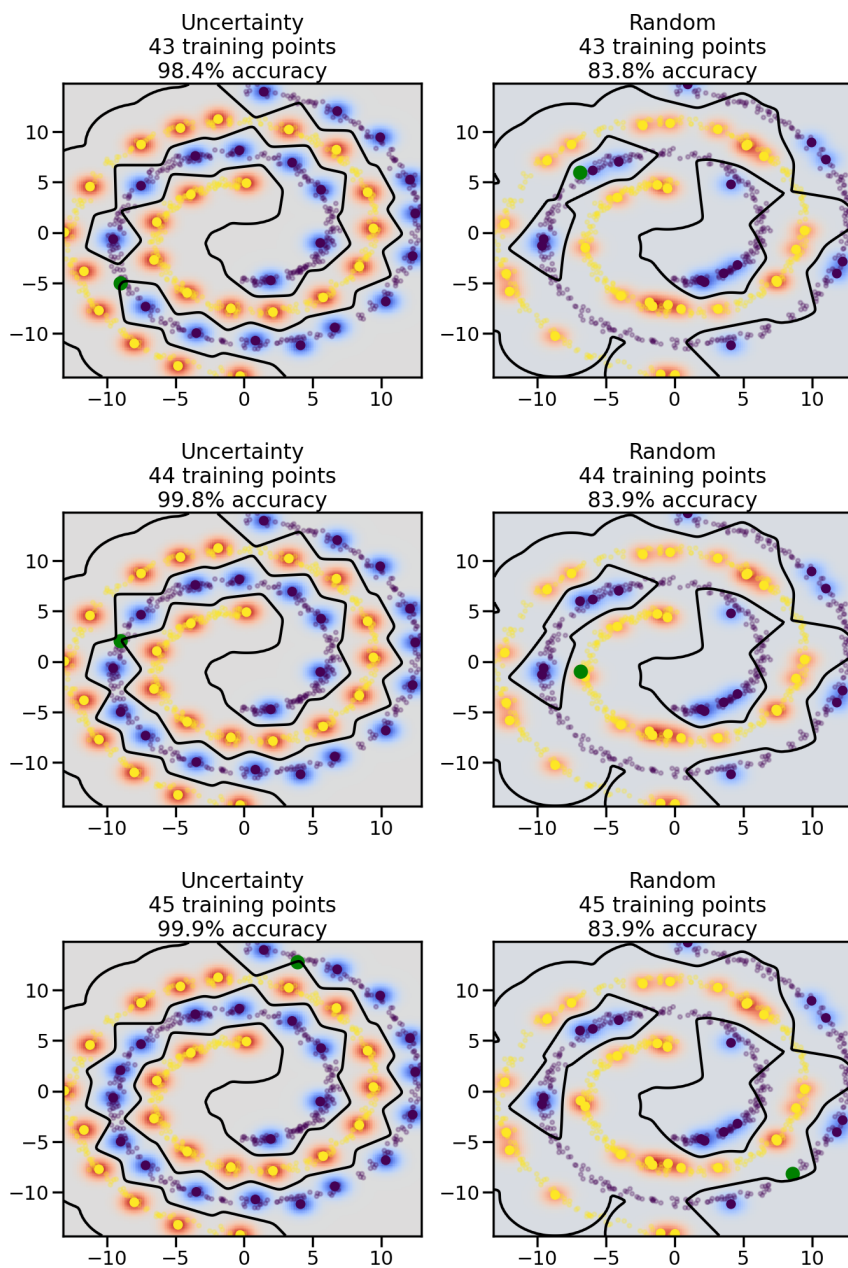


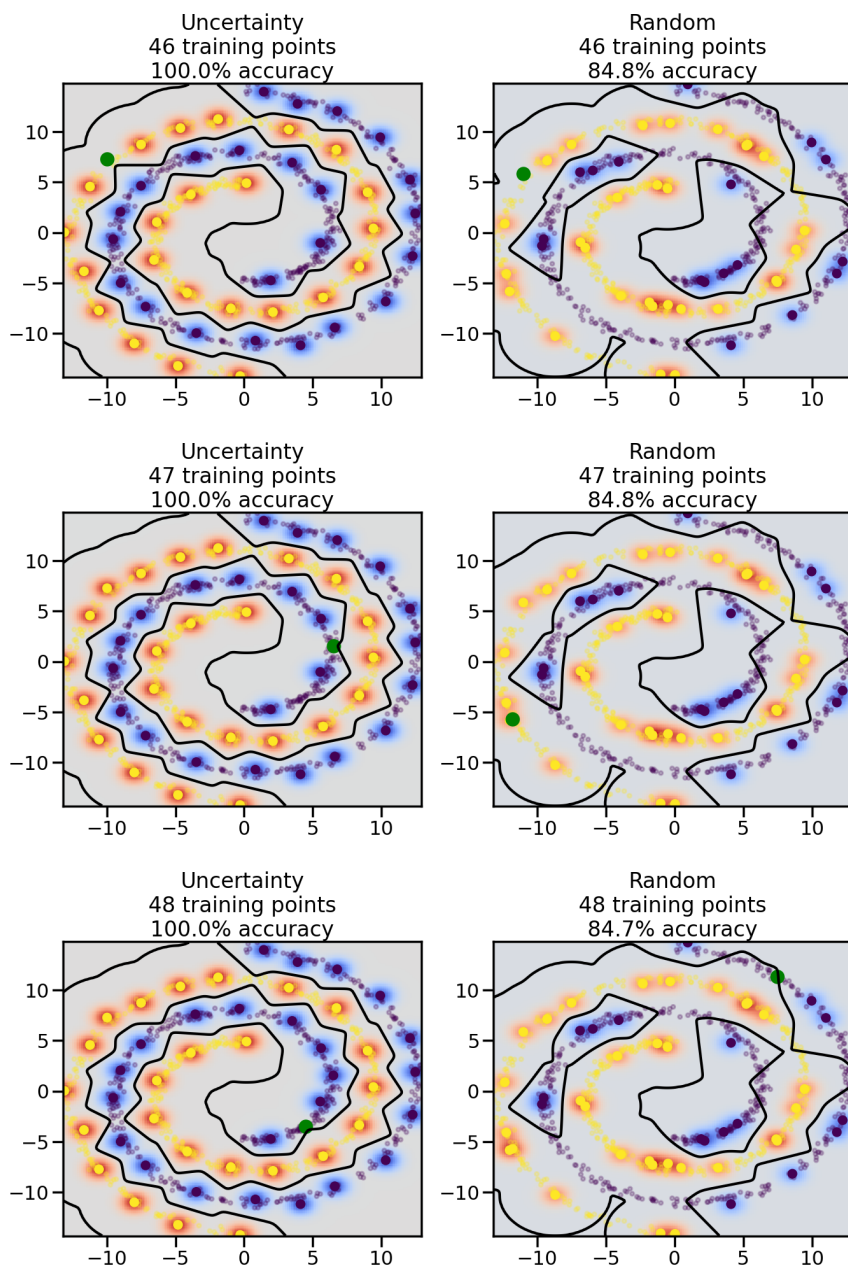


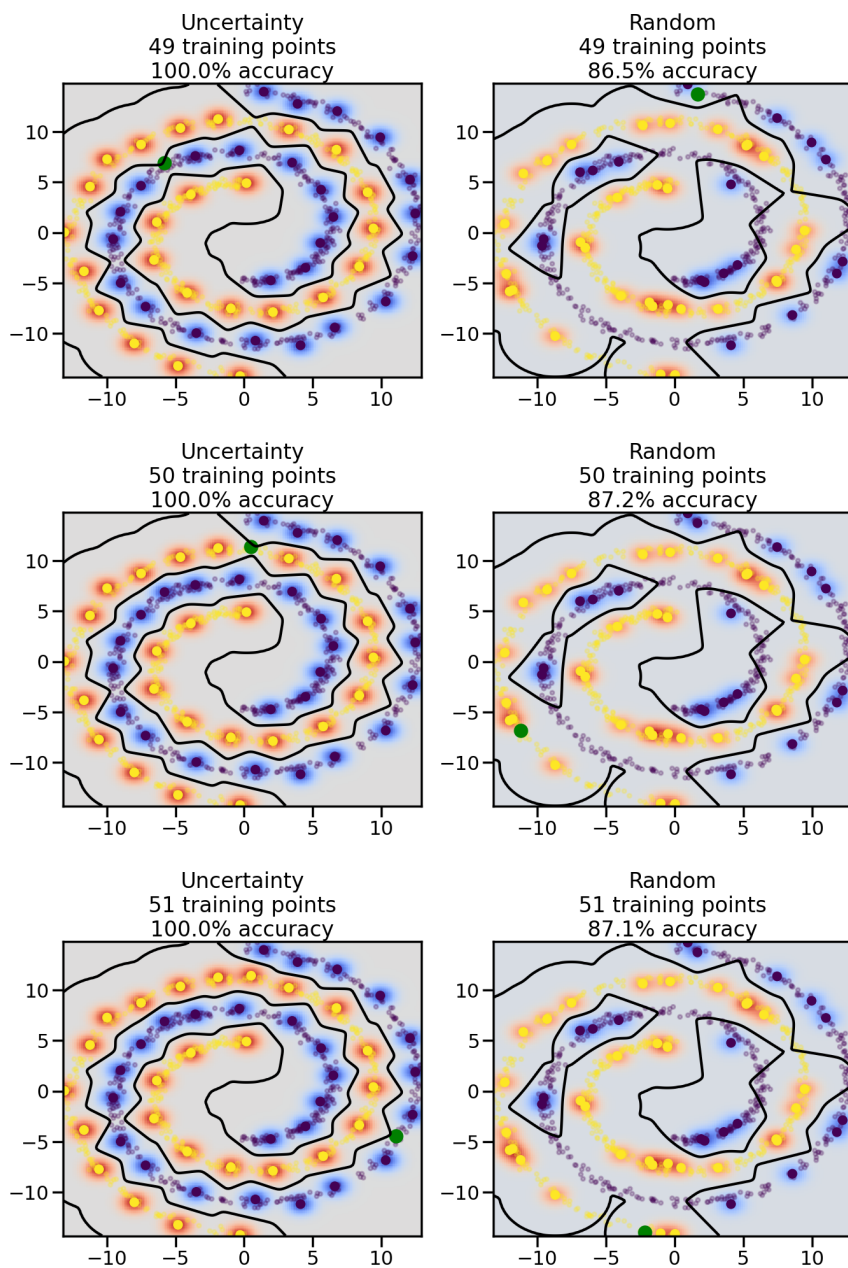


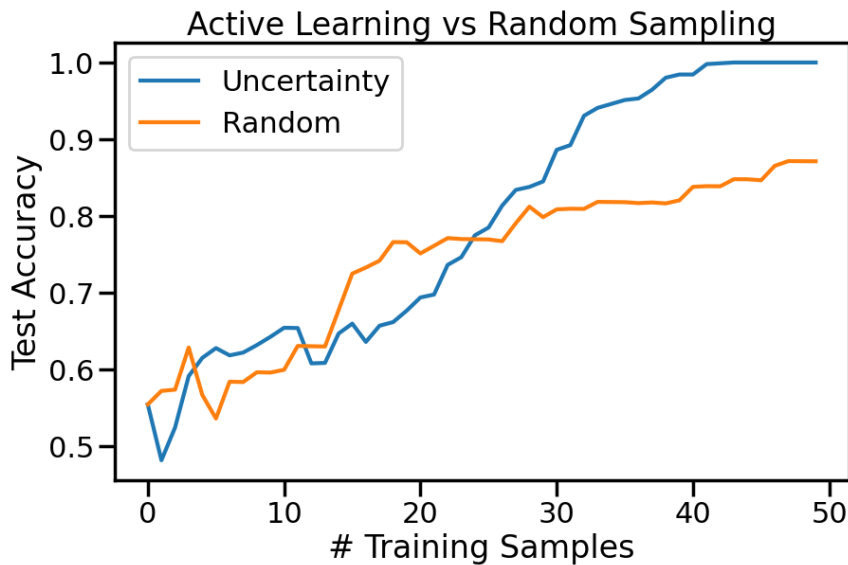
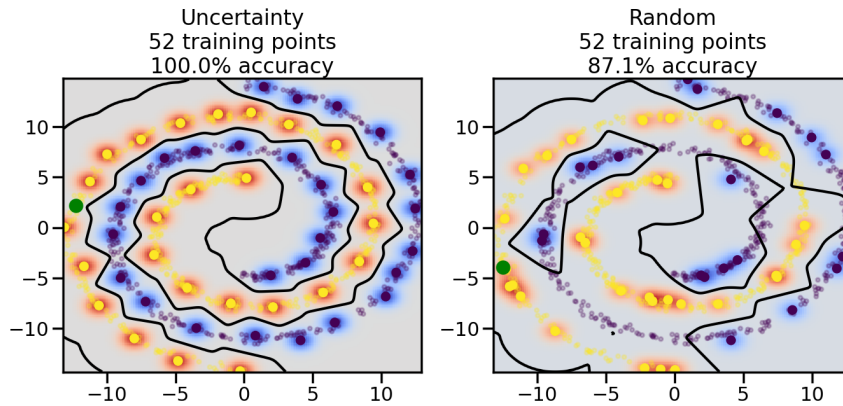












💡 Experiment: Comparison of Uncertainty Sampling in Classification

You can re-run the above classification experiments with different datasets (e.g., Moons, Circles, Swiss Roll) and varying noise levels. Observe how the performance of Uncertainty Sampling compares to Random Sampling in each case. Consider the following questions:

- How are the selected data points distributed using uncertainty sampling? How do they differ from those selected by Random Sampling?
- What happens when you increase the noise level on Dataset one (Moons)? How does the relative performance of Uncertainty sampling change as the noise increases?
- How does the difference between Uncertainty Sampling and Random Sampling change when we move to more complicated datasets (e.g., Dataset 3 - the swiss roll example?)
- How does the comparative behavior of Active Learning change when you change the

underlying model (say, in this case, by changing γ – the RBF kernel bandwidth)?

22.6 Semi-Supervised Learning

In addition to Active Learning, we can use [Semi-Supervised Learning](#) to leverage the availability of unlabeled data. Semi-Supervised learning works by assigning *unlabeled* points a label based on nearby supervised data points. Essentially, this uses a subset of labeled data to “imagine” or “propagate” labels to the unlabeled data points. To the extent that nearby points in the input space are likely to share the same label, this can be a powerful way to leverage unlabeled data to improve model performance.

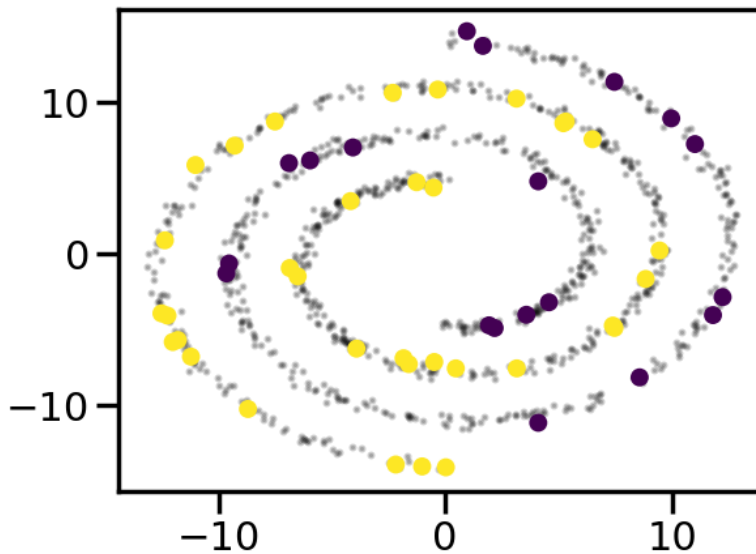
There are several common algorithms for semi-supervised learning, including:

- **Label Propagation:** This algorithm constructs a graph where nodes represent data points and edges represent similarities between points. Labels are then propagated through the graph from labeled to unlabeled nodes based on their connections.
- **Label Spreading:** Similar to label propagation, but it uses a different normalization technique for the graph, which can lead to different propagation dynamics.

In the below example, we will use Label Spreading to demonstrate semi-supervised learning on the same classification problem we saw above. However, despite whatever specific algorithm we might use, the key idea remains the same: we will use the unlabeled data to act as a kind of similarity measure between points, and then use that similarity measure to decide to what extent certain labeled points should influence the labels of nearby unlabeled points.

We will start by visualizing a largely unlabeled dataset, but then selecting a few points at random to label. As we can intuitively see below, even with only a few labeled points, we can tell just by looking at the data what labels are likely to occur in some of the unlabeled points:

Mix of Labeled and Unlabeled Data



22.6.1 Label Spreading Example

One common algorithm for semi-supervised learning is [Label Spreading](#). This algorithm constructs a graph where nodes represent data points and edges represent similarities between points. Labels are then propagated through the graph from labeled to unlabeled nodes based on what nodes they are connected to.

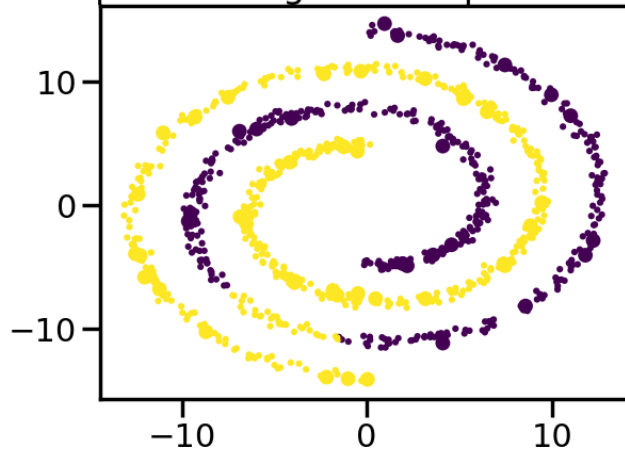
Specifically, Label Spreading works by first constructing a similarity graph from the data points. The similarity between two points is typically measured using a kernel function, such as the Radial Basis Function (RBF) kernel. Once the graph is constructed, the algorithm iteratively updates the labels of the unlabeled nodes based on the labels of their neighbors, which “closer” neighbors having greater influence. In this way, a label iteratively moves outward from the actual labels through unlabeled nodes. This process continues until convergence, at which point the labels of the unlabeled nodes are assigned based on the propagated labels.

```
label_prop_model = LabelSpreading(kernel='rbf', gamma=5.0, max_iter=400)
labels = -np.ones_like(y)
labels[train] = y[train]
label_prop_model.fit(X, labels)
ll = label_prop_model.transduction_

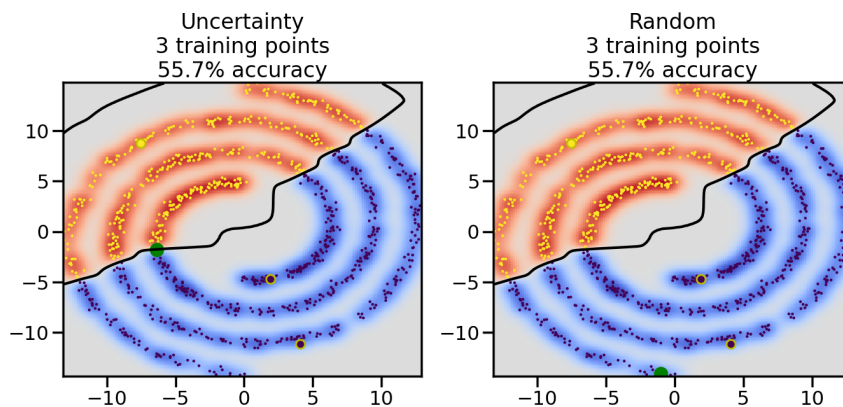
# Plot the results
plt.figure()
plt.scatter(X[:,0],X[:,1],c=ll,lw=0, s=20)
```

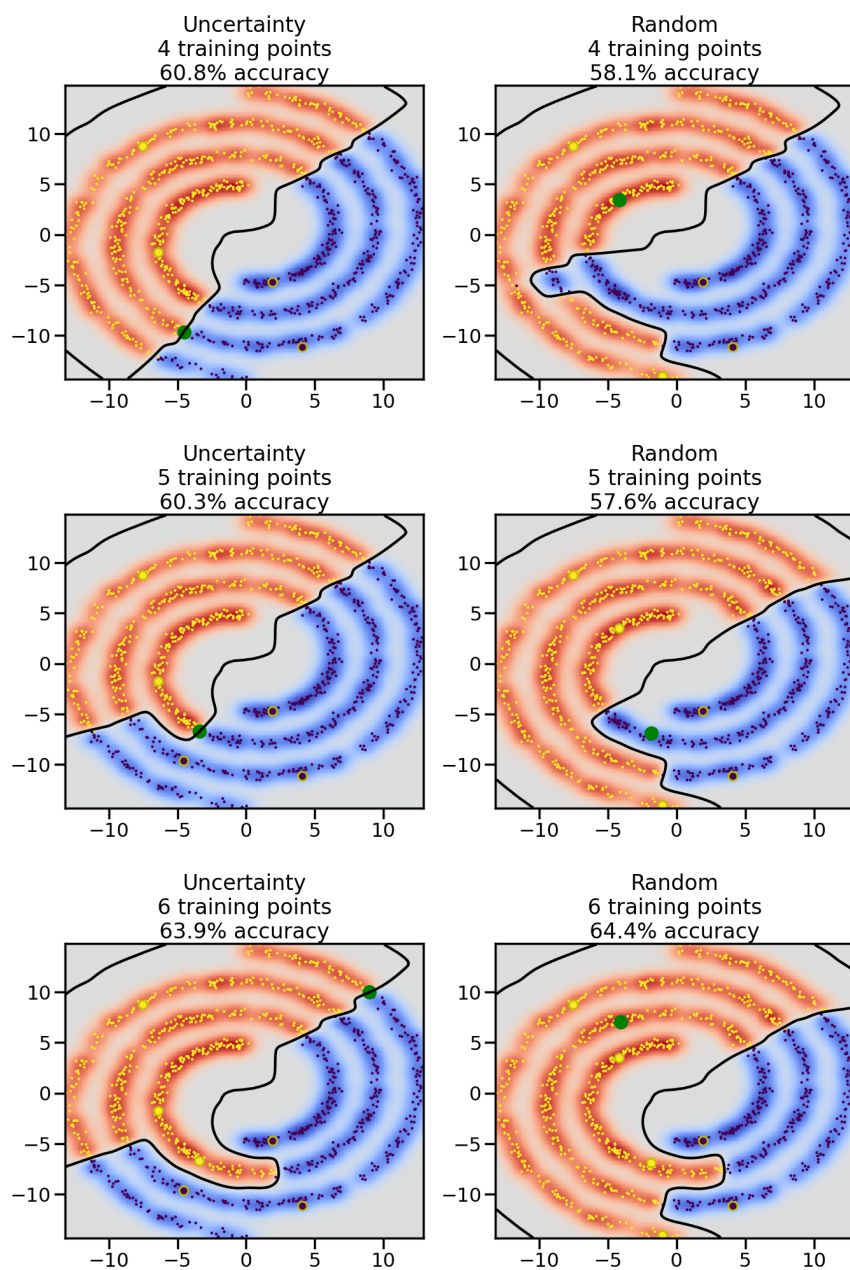
```
plt.scatter(X[train,0],X[train,1],c=y[train],s=100,lw=0)
plt.title("Labels Spread using Semi-Supervised Learning")
plt.show()
```

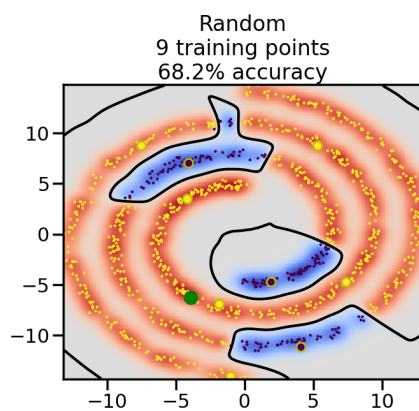
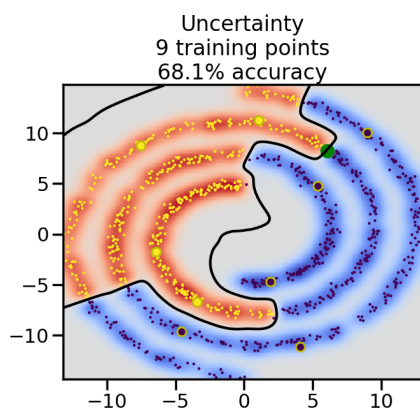
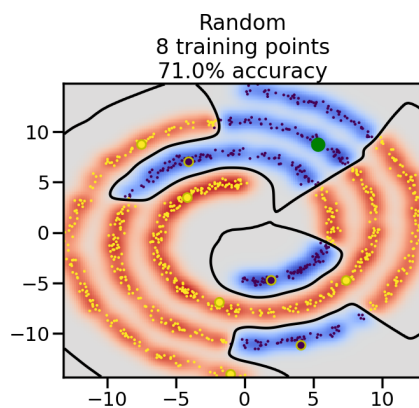
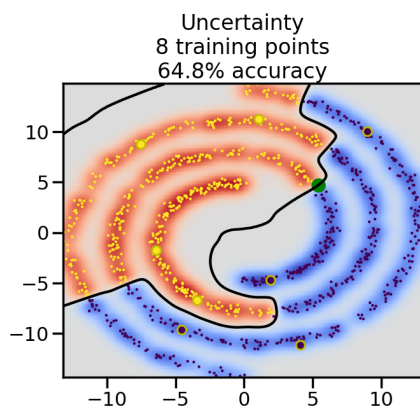
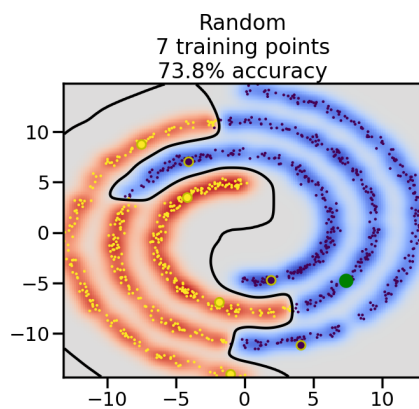
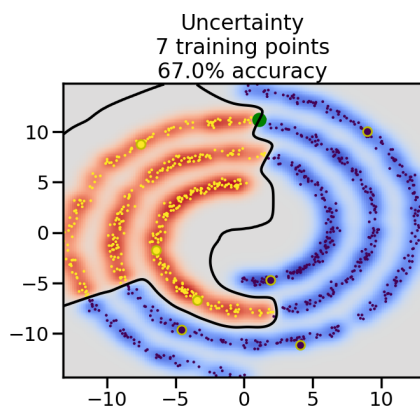
Labels Spread using Semi-Supervised Learning

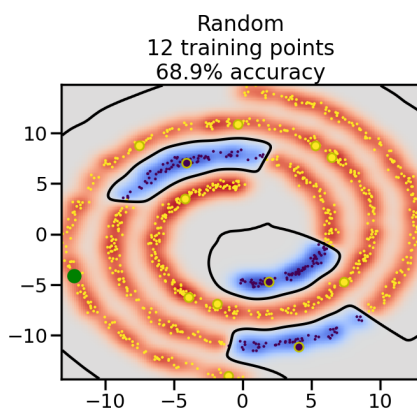
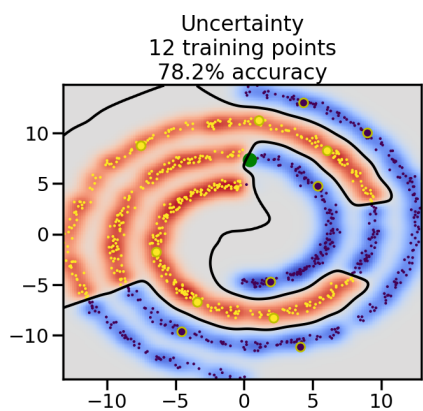
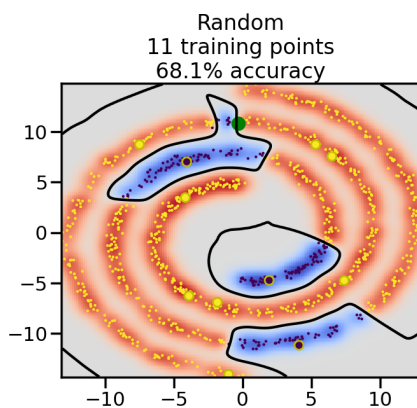
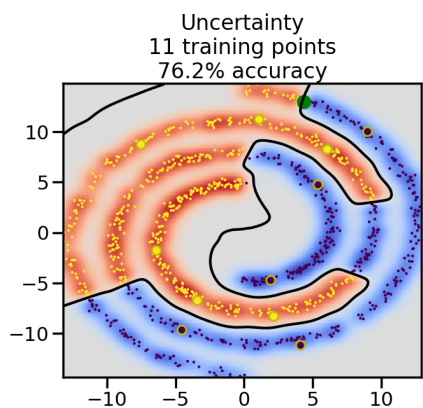
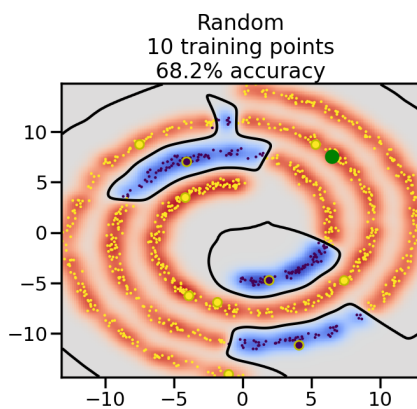
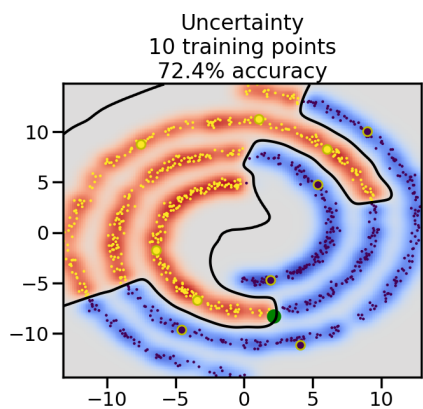


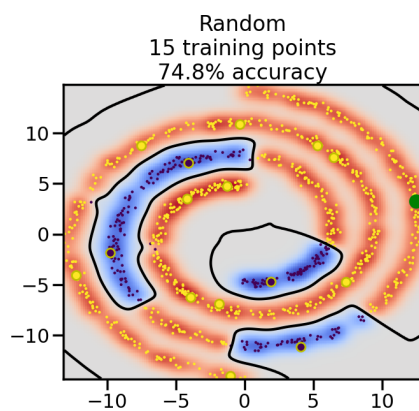
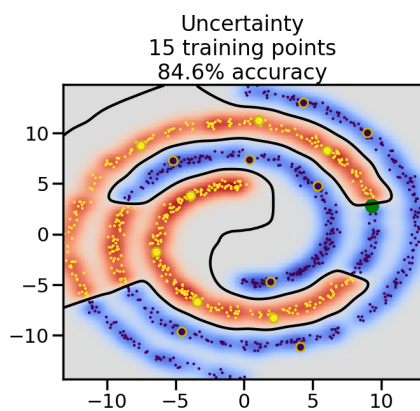
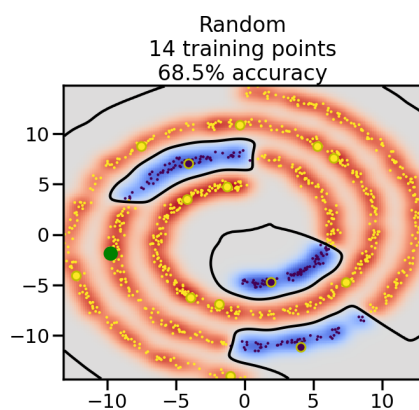
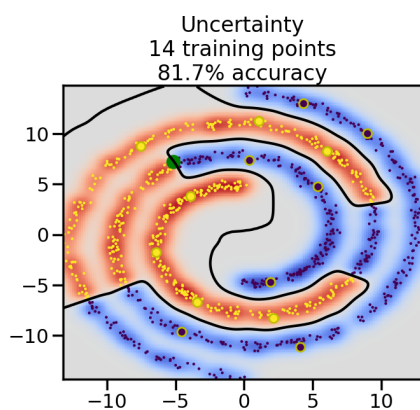
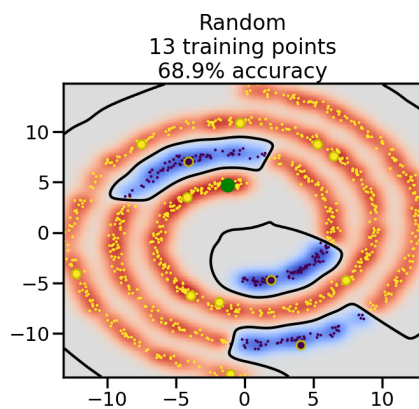
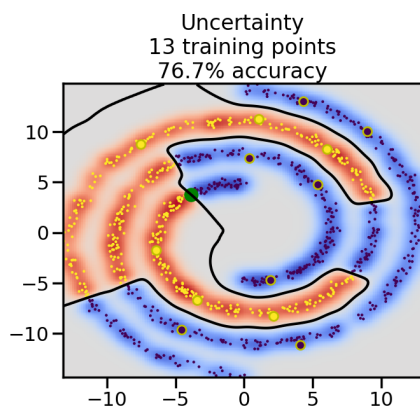
Now we can re-run our above Active Learning experiments, but this time we can also include Label Spreading to “extend the reach” of the samples we collect. Let’s take a look at how incorporating both Active Learning and Semi-Supervised Learning can improve our classification performance compared to using either method alone or neither method at all.

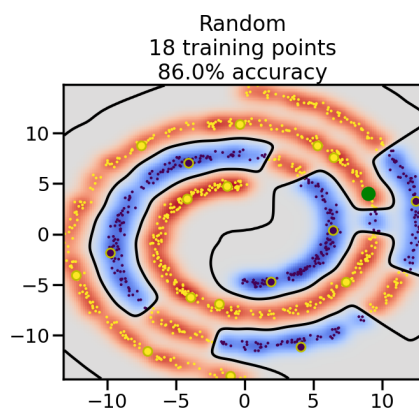
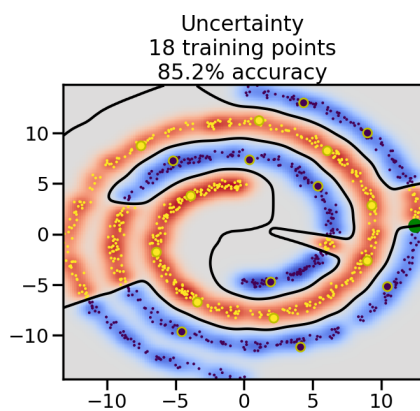
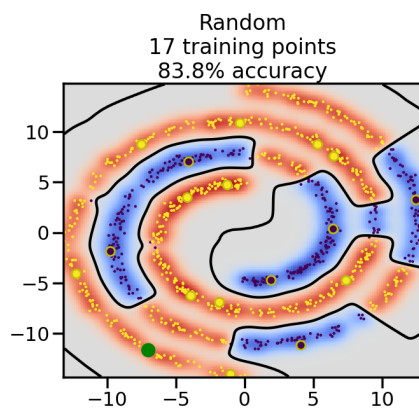
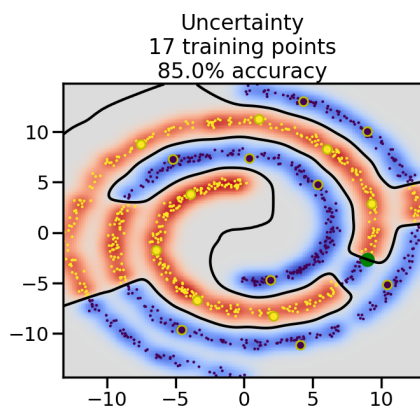
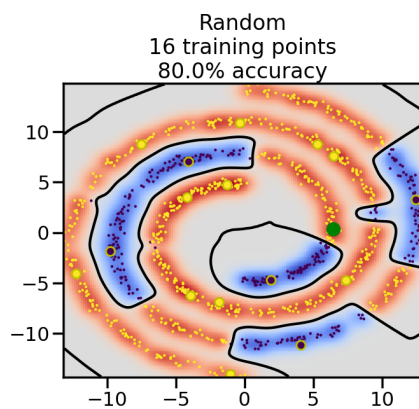
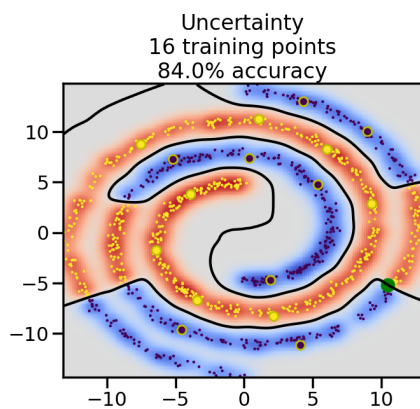


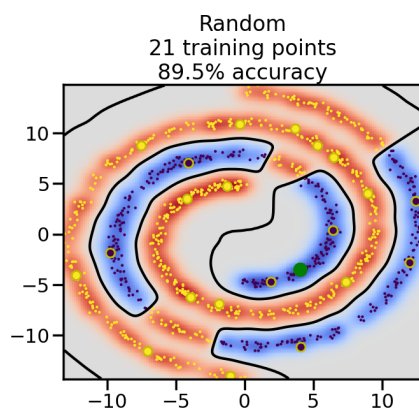
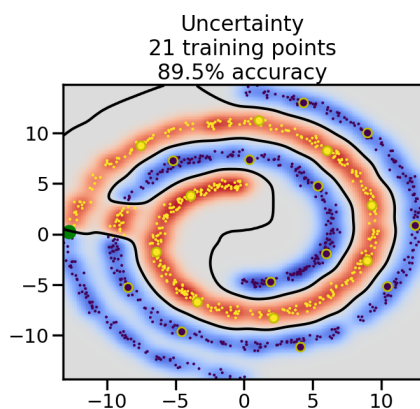
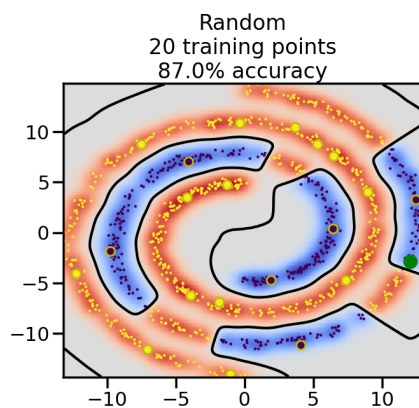
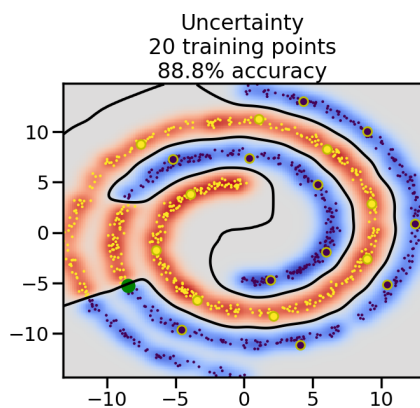
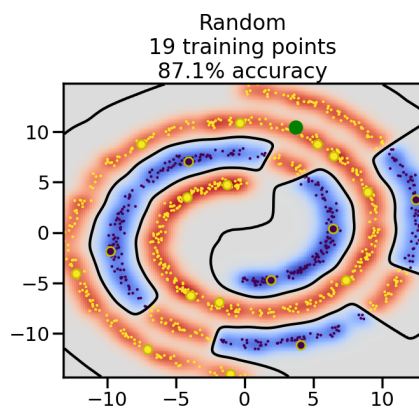
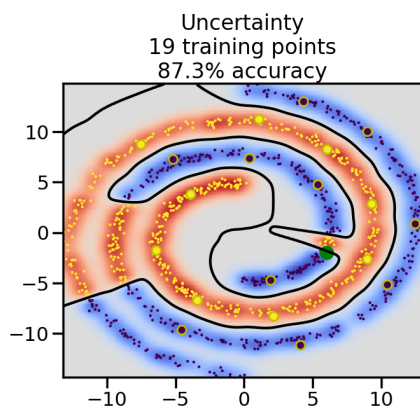


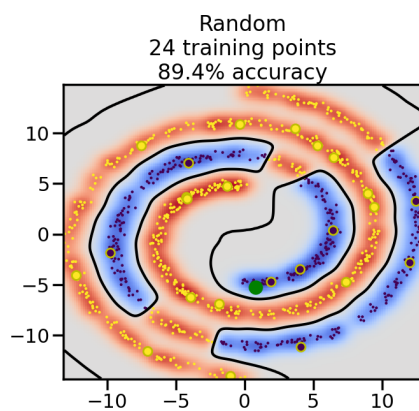
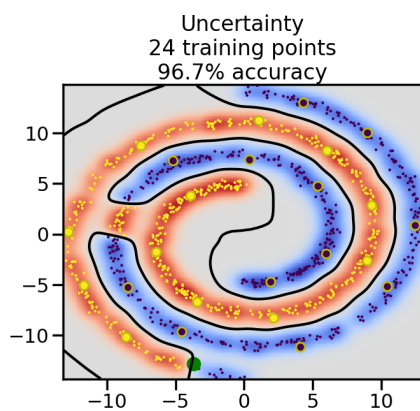
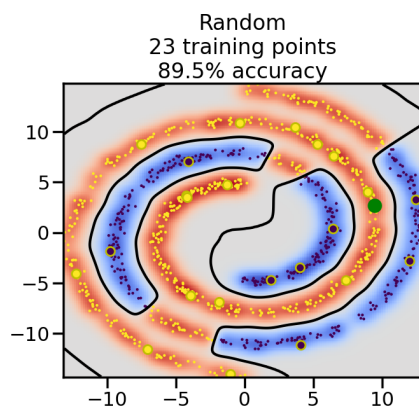
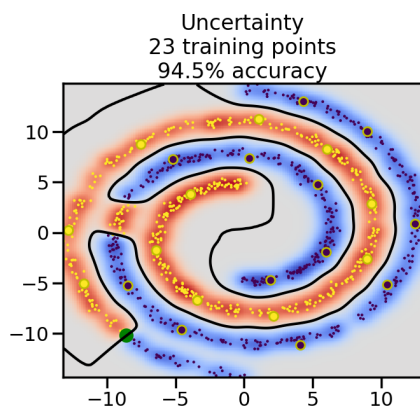
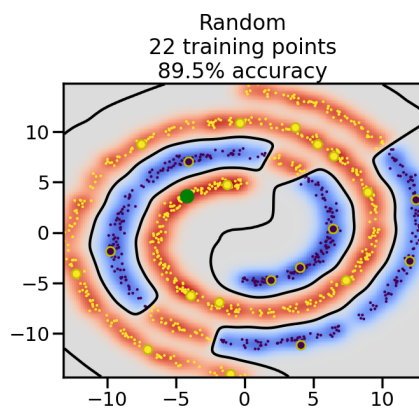
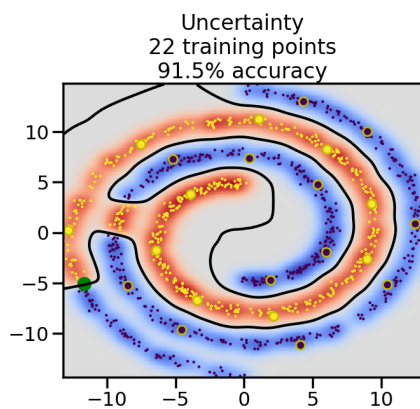


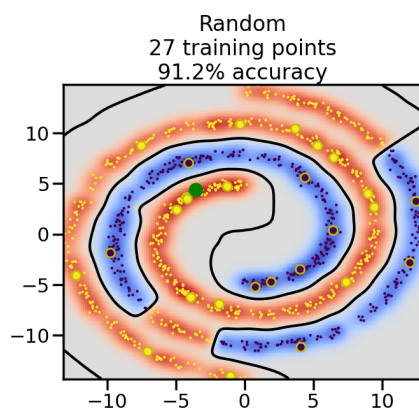
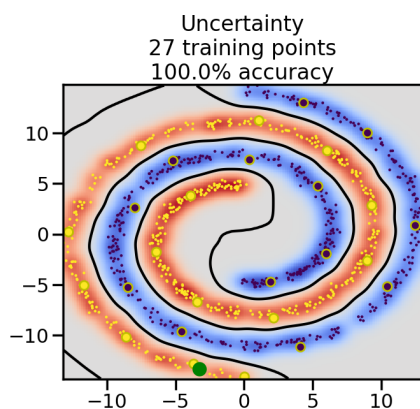
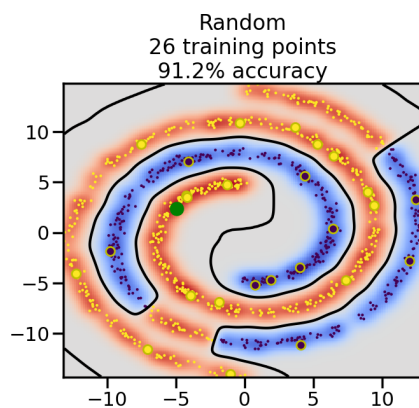
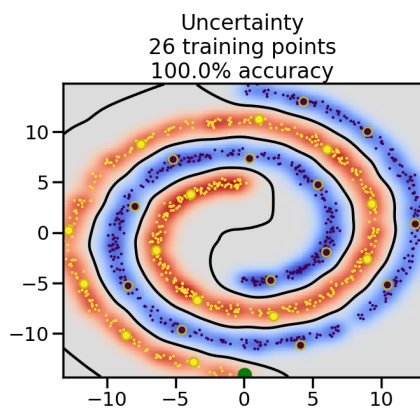
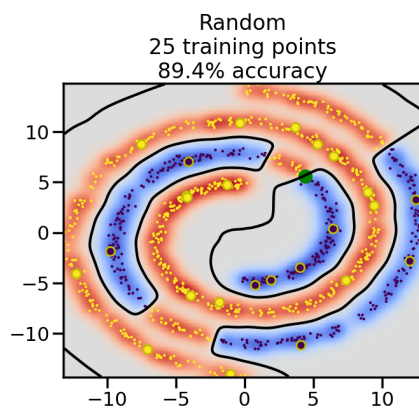
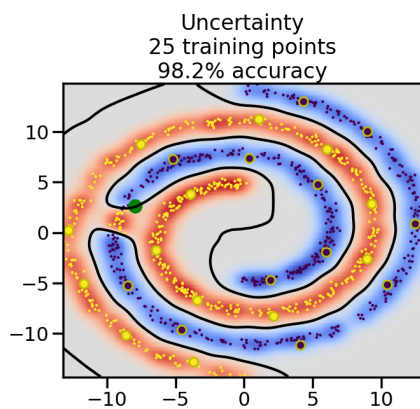


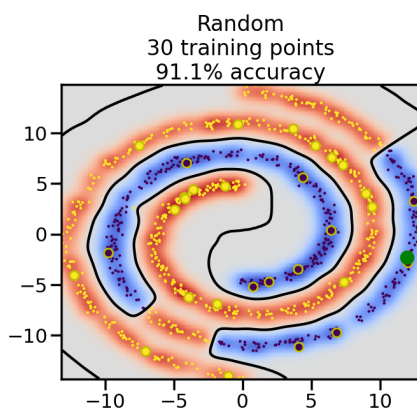
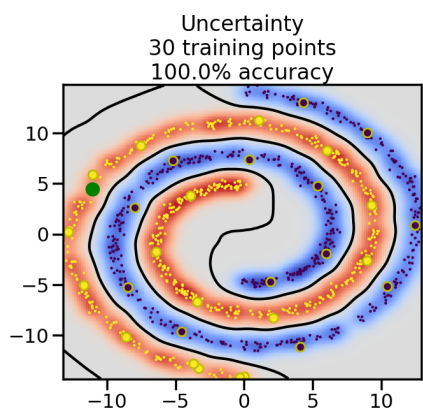
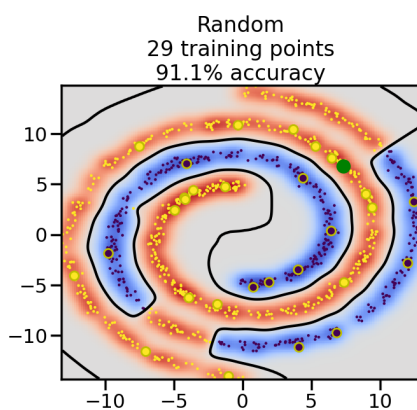
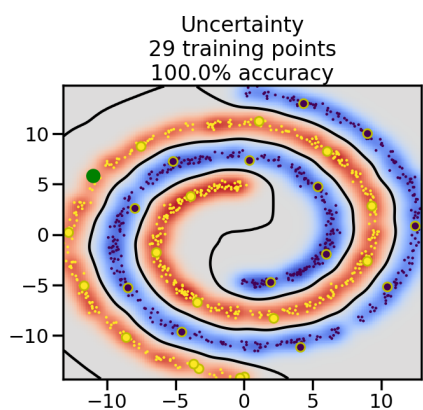
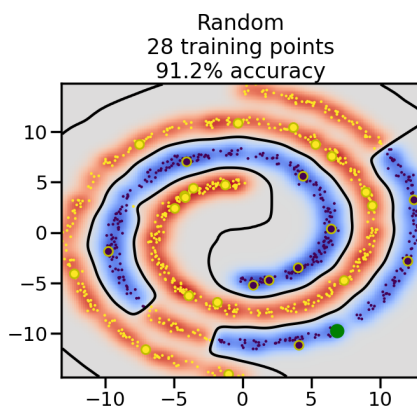
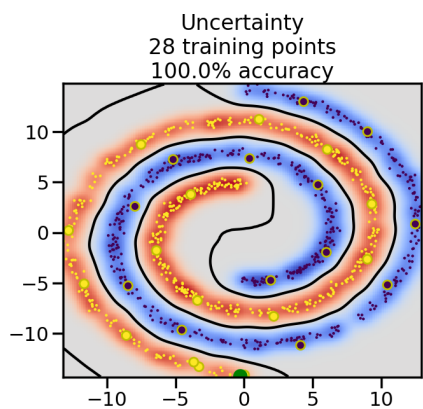


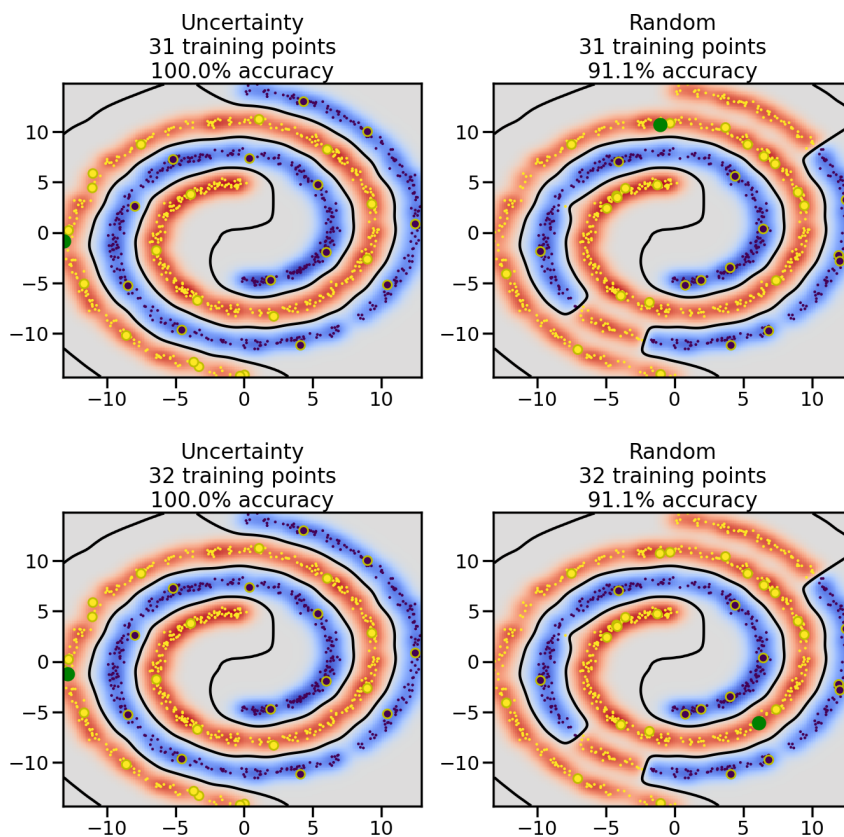




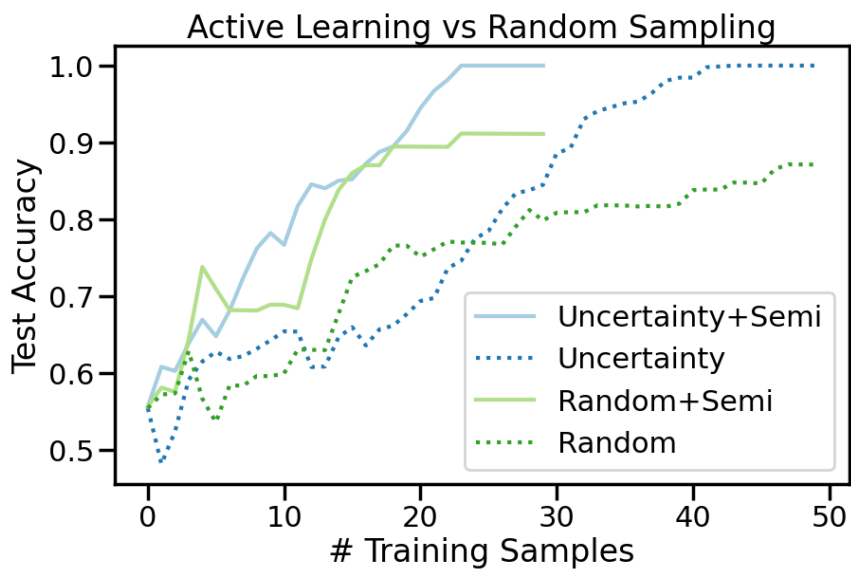








We can also compare the accuracy versus sample curve from before and after Semi-Supervised learning to see how much of a difference it makes:



💡 Experiment: When and Where does Semi-Supervised Learning Help?

Compare and contrast your results from the above Active Learning experiments with those using Semi-Supervised Learning and consider the following questions:

1. How does Semi-Supervised learning (using both Uncertainty and Random Sampling) compare to the non-semi-supervised cases you saw above?
2. In what ways does the use of Uncertainty Sampling + Semi-Supervised sampling change the location of sampled data points, compared to uncertainty sampling by itself?
3. How does Active + Semi-Supervised learning perform on Dataset 3, relative to just Active Learning by itself on Dataset 3? Why does it perform so differently?
4. Is Active + Semi-Supervised learning always better than either approach by itself? What happens in Dataset 4 (Moons + Gaussian)?

22.7 Summary

In this chapter, we explored the concepts of Active Learning and Semi-Supervised Learning, two powerful techniques for making the most of limited labeled data. We discussed various strategies for selecting informative data points in Active Learning, with a focus on Uncertainty Sampling due to its simplicity and effectiveness. We also examined how Semi-Supervised Learning can leverage unlabeled data to improve model performance through techniques like Label Spreading.

Overall, both Active Learning and Semi-Supervised Learning can significantly enhance model performance in scenarios where labeled data is scarce or expensive to obtain. By intelligently selecting data points to label and utilizing unlabeled data, these techniques allow us to build more accurate and robust models with fewer resources. However, we also saw a few downsides. Notably, both Active Learning and Semi-Supervised learning can become overly confident, leading the model to ignore sampling regions where it is particularly confident, even if those regions are actually incorrect. Thus, an over-reliance on only gathering samples using Active Learning could bias the model away from sampling regions that might, in the long run, be more informative.

References

A Helpful Tooling for Working with and Debugging Machine Learning Models

Machine learning projects are notoriously brittle: minor implementation details can cause major differences in outcomes. Good practices and tools make your implementation **reproducible** (and thus **debuggable**), and **portable** across different hardware environments, such as a teammate’s laptop or a high-performance computing (HPC) system.

Unlike traditional programming, debugging ML models by simply “running and fixing in a loop” is rarely effective. Instead, a structured set of practices and tools is needed to understand model behavior and reproduce results reliably.

A.1 TL;DR Checklist

Here is what we encourage doing, sorted by impact over effort.

1. **Environment & Dependencies**: Use a proper development environment, virtual environments and pin package versions. Document GPU/CUDA.
2. **Linting**: Use tools like `ruff` to help you write clean code from the start.
3. **Random Seeds**: Seed all libraries (`torch`, `numpy`, `random`) and enable deterministic operations.
4. **Logging & Checkpoints**: Log hyperparameters, save models, track experiments (e.g., `wandb`).
5. **Visualization**: Plot data and learning curves.
6. **Start Small Then Scale**: Debug on small datasets first.
7. **Version Control**: Track code with Git; use branches for experiments.
8. **Modular Code**: Split code into functions/classes and separate files.
9. **Testing & Type Hints**: Write `pytest` tests and use `mypy` for type checking.

A.2 Reproducible Runs

The first requirement for a robust ML project is to make the codebase **deterministic**. Because ML is inherently probabilistic, running the same code twice without precautions may yield different results. Determinism is therefore a prerequisite for debugging.

A.2.1 Python Project Setup

A good practice is to use a virtual environment (e.g., [venv](#)) to isolate dependencies. This prevents conflicts between projects (e.g., my linear regression project uses `sklearn 1.0.2`, and my generative design project uses `sklearn 2.5.3`) and avoids interfering with the base operating system.

Dependencies should be pinned to exact versions in a file such as `pyproject.toml` or `requirements.txt`. This enables others (including future you) to reproduce the same environment.

Note that not all dependencies are automatically captured in Python configuration files—for instance, the CUDA version used for GPU processing must be documented separately (typically in a `README.md`).

A.2.2 Seeded Runs

Most ML techniques involve randomness (e.g., parameter initialization, sampling, data shuffling). To ensure reproducibility, it is necessary to set a *random seed* so that random number generators produce a deterministic sequence.

In practice, several libraries must be seeded and it will look similar to:

```
import torch as th
import numpy as np
import random

my_seed = 42

th.manual_seed(my_seed) # PyTorch
th.backends.cudnn.deterministic = True
torch.cuda.benchmark = False
rng = np.random.default_rng(my_seed) # NumPy
random.seed(my_seed) # Python's built-in random
```

A.2.3 Hyperparameters

Hyperparameter values can influence results as strongly as changing the algorithm itself. It is essential to record which hyperparameters were used for each experiment.

Experiment-tracking platforms automate this process. For example, [Weights and Biases](#) (wandb), or [Trackio](#) can log hyperparameters, Python version, and hardware details, as well as visualize results such as learning curves. See, for example:

- [Run overview](#) with hyperparameter “Config.”
- [Learning curves and sampled designs](#)

Checkpoint

Once versions, seeds and hyperparameters are fixed, **running the model multiple times should yield identical results across runs** (look at the wandb curves). Inconsistent results usually indicate a missing seed or an unpinned dependency. Without determinism, debugging will be much more time-consuming. We strongly advise to pay attention to this.

Other Sources of Non-Determinism

This is unlikely to happen within the course but it is still worth mentioning. Even if you set seeds and pin library versions, some sources of non-determinism may persist due to the environment:

- **Multithreading or parallelism:** Operations may be executed in different orders on CPU threads.
- **GPU operations:** Certain GPU kernels are non-deterministic by design, even with fixed seeds.
- **Library versions or BLAS/CUDA backends:** Different versions of underlying math libraries may produce slightly different results.

To mitigate these issues:

- Enable deterministic operations where possible (e.g., `torch.backends.cudnn.deterministic = True` for PyTorch).
- Be aware that some operations may never be fully deterministic on GPU—document this for reproducibility.

A.3 Code Management

ML projects should be approached as software engineering projects. Code quality and management are especially critical: poorly organized or fragile code increases the likelihood of errors and makes debugging more difficult. In addition, Python’s permissiveness can hide subtle mistakes. For example, [automatic broadcasting of scalars to vectors](#) may not raise an exception when performing operations on vectors or matrices of mismatched sizes, yet it can still produce incorrect results. Such silent errors are often harder to detect than explicit crashes.

A.3.1 Code Organization

Notebooks are valuable for exploration and prototyping, but they are less suited for building robust and reproducible experiments. Relying on a single notebook or script often leads to unmanageable code as the project grows. By contrast, a modular codebase is easier to test, extend, and maintain. Organizing code into **smaller, modular components** simplifies both debugging and collaboration.

- Divide the project into functions, each with a single, well-defined purpose. A useful rule of thumb is: if you cannot clearly explain what a function does in one sentence, it should probably be split.
- Use classes when it is natural to group related data and behavior together.
- Split large projects across multiple files to make navigation easier. Avoid single files with 1000+ lines, as they are hard to read, debug, and extend.

A.3.2 Version Control

Version control ensures that specific states of a project can be identified and restored. We strongly recommend using [Git](#) (with GitHub or similar platforms) for ML projects. When working in teams, branches help manage changes and prevent conflicts.

A.3.3 Formatting and Linting

Code formatting conventions (e.g., number of spaces per indentation, placement of comments, naming conventions) do not affect program behavior but improve readability. There are formatters that can automatically fix the visual style of the code—things like indentation, line breaks, spacing around operators, and alignment.

Linting, goes beyond formatting: linters analyze your code for potential errors or risky patterns, such as unused variables, variables that may be undefined, or suspicious comparisons.

[ruff](#) integrates formatting, linting, and error detection in a single tool. It improves code quality, reduces stylistic disagreements, and allows developers to focus on the intent rather than the syntax.

A.3.4 Type Hints

In addition to formatting and linting, static type checking helps catch errors before running your code. Python is dynamically typed, which means you can easily pass the wrong type of object to a function without immediate errors. Tools like [mypy](#) analyze your code using type hints and report mismatches.

For example, in:

```
def add_numbers(a: int, b: int) -> int:
    return a + b

add_numbers(2, "3") # note the String here
```

Running `mypy` will output:

```
main.py:4: error: Argument 2 to "add_numbers" has incompatible type "str"; expected "int" [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Using type hints (`a: int, -> int`) together with `mypy` lets you detect bugs early, improves code readability, and helps IDEs provide better autocompletion and refactoring support.

A.3.5 Testing

Testing individual components—functions, classes, or modules—is an important way to ensure reliability. Well-written tests allow developers to:

- **Isolate potential error sources:** When a bug occurs, thoroughly tested components can be excluded from investigation, saving time.
- **Detect unintended side effects:** Tests help ensure that changes in one part of the codebase do not break other parts.

Testing and good code organization go hand in hand: modular code is naturally easier to test, and writing tests often encourages cleaner, more maintainable designs.

The most common tool for this is [pytest](#). For example:

If you define a function in your project:

```
# my_project/utils.py
def add_numbers(a, b):
    return a + b
```

You can define tests with:

```
# tests/test_math.py -- this is your test file
from my_project.utils import add_numbers

def test_add_numbers():
    assert add_numbers(2, 3) == 5
    assert add_numbers(-1, 1) == 0
```

Running pytest will automatically discover these tests and report any failures.

A.3.6 Integrated Development Environments (IDEs)

While you can write Python code in any text editor, using an IDE significantly improves productivity. **Visual Studio Code (VS Code)** is the most popular choice for Python and ML development. It supports:

- **Extensions:** Add functionality and friendly interface for linting `ruff`, type checking `mypy`, testing `pytest`, and `git` integration.
- **Handling of virtual environments:** VSCode can create and handle virtual environments for you.
- **Debugger:** Set breakpoints and inspect the current state of variables, run instruction by instruction. This is much easier than putting prints everywhere.
- **Notebooks inside VS Code:** You can run Jupyter notebooks directly within your IDE.
- **LLMs integration:** Students have access to GitHub education (and Copilot), VSCode has a direct LLM integration for code completion and agent.

A.3.7 Large Language Models (LLMs) for Coding

Tools like ChatGPT or GitHub Copilot can generate code quickly. While this can accelerate boilerplate writing, **it does not replace understanding**.

Machine learning code is particularly sensitive to details: a small mistake in data preprocessing, tensor dimensions, or random seeding can completely change results. Using LLMs without knowing what the code does may:

- Hide important assumptions.
- Lead to silent bugs that are hard to detect.
- Prevent you from learning how ML algorithms really work.

Guideline: LLMs are great for generating snippets (e.g., “write a function to convert my CSV data to JSON”), but always **read, run, and understand the code** before using it in experiments. For ML, correctness and reproducibility are more important than speed.

A.4 Debugging

If the codebase is well-structured and reproducible but issues persist, the problem is likely related to the maths, hyperparameters, or data.

A.4.1 Visualizing

Visualization is one of the most effective debugging tools in ML, particularly in engineering contexts where results can often be represented graphically.

Algorithm-level Visualizations

- **Loss curves:** Simple plots can reveal overfitting, underfitting, or learning failures.
- **Predictions:** Comparing model outputs with reference data at various training stages provides direct insight into progress.

For these, we often report metrics and outputs in wandb, see for instance [these lines](#).

Data-level Visualizations

- **Inspect dataset distributions:** Check whether features are on compatible scales, whether rescaling or normalization is needed, and whether outliers are present. Tools like `matplotlib` or `seaborn` can help.
- **Assess assumptions:** Determine whether the data distribution aligns with the model’s underlying assumptions, e.g., can the data distribution be captured by a Gaussian distribution.

A.4.2 Split Your Pipeline

It is good practice to split your training pipeline into distinct stages:

- **Data analysis:** [Visualize](#) your data. Look at the distributions, detect outliers, and gain insights into what preprocessing might be needed and which models may perform well.
- **Data pre-processing:** Massage your data before feeding it to the model. [Visualize](#) to ensure transformations are correct and consistent.
- **Training:** Train your model on the preprocessed data. Save trained models to disk after each run (`torch.save`, `pickle`, or similar). This allows you to avoid retraining from scratch every time you tweak evaluation code.
- **Evaluation:** Load the saved model and run your evaluation routines on validation or test datasets.

By separating these stages, you can debug each part independently, and validate progress.

A.4.3 Start Small, Then Scale

When debugging, it is inefficient to run large-scale experiments immediately. Instead:

- Begin with small, fast experiments (e.g., a reduced dataset or a lightweight simulator).
- Validate that the model can learn on trivial cases.
- Attempt to reproduce established results or baseline performance.

Scaling to larger, more complex runs should only occur once smaller experiments confirm that the model behaves as expected.

A.4.4 Performance Profiling with `timeit`

Sometimes the bug is actually that the code is too slow. When this happens, the first step is often to measure *where* the time goes. You can do that with Python's built-in [timeit](#).

`timeit` runs a snippet of code multiple times and reports the average execution time, helping you compare different implementations or detect bottlenecks.

Here is an example for normalizing data:

```

import numpy as np
import timeit

setup = """
import numpy as np
data = np.random.rand(10000, 100) # 10k samples, 100 features
"""

# Option 1: Pure Python loops
stmt1 = """
normalized = []
for row in data:
    mean = np.mean(row)
    std = np.std(row)
    normalized.append((row - mean) / std)
normalized = np.array(normalized)
"""

# Option 2: NumPy vectorization
stmt2 = """
means = np.mean(data, axis=1, keepdims=True)
stds = np.std(data, axis=1, keepdims=True)
normalized = (data - means) / stds
"""

print("Python loops:", timeit.timeit(stmt1, setup=setup, number=10))
print("NumPy vectorization:", timeit.timeit(stmt2, setup=setup, number=10))

```

Results:

```

Python loops: 0.8857301659882069
NumPy vectorization: 0.04489224997814745

```

Using NumPy vectorization is ~20x faster than Python loops.

In notebooks, you don't even need imports:

```
%timeit sum(range(1000))
```

A.5 A Practical Example

This section shows a practical example using the techniques explained above on an actual code.

A.5.1 Step 0: The Ugly Script

We start with some messy code that Ruff would flag:

```
# train.py
import numpy as np, torch, torch.nn as nn, torch.optim as optim, matplotlib.pyplot as plt, random
from sklearn.model_selection import train_test_split

X_all =np.linspace(0, 100, 500).reshape(-1,1)
y_all = 5* np.sin(0.1 * X_all)+np.random.randn(500,1)
X_train,X_test,y_train,y_test =train_test_split(X_all,y_all,test_size=0.2,random_state=42)

model =nn.Linear(1,1)
optimizer =optim.SGD(model.parameters(),lr=0.01)
loss_fn= nn.MSELoss()

for epoch in range(500):
    pred =model(torch.tensor(X_train))
    loss= loss_fn(pred, torch.tensor(y_train))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch} - Loss: {loss.item()}")
```

At first glance, it looks okay but it won't run. Try executing `python train.py` to see the errors.

A.5.2 Step 1: From Ugly to Bad

Run `ruff` to format and check. Fix the errors (or call `ruff check --fix train.py`).

Now the code is already cleaner and easier to debug. Still, running the file throws errors.

A.5.3 Step 2: Debugging

Running python train.py gives a cryptic type error at the loss computation: `RuntimeError: mat1 and mat2 must have the same dtype, but got Double and Float`.

The problem is that the model outputs `float32` predictions while `y` is `float64`. This causes a mismatch in the loss computation. The fix is to change the lines to:

```
- pred = model(torch.tensor(X_train))
- loss = loss_fn(pred, torch.tensor(y_train))
+ pred = model(torch.tensor(X_train, dtype=torch.float32))
+ loss = loss_fn(pred, torch.tensor(y_train, dtype=torch.float32))
```

A.5.4 Step 3: Make your Script as Deterministic as possible

It is important to remove sources of non determinism when debugging ML models, see [seeding](#).

```
# right after the imports
rng = np.random.default_rng(42) # seed NumPy random
torch.manual_seed(42) # seed PyTorch
torch.cuda.manual_seed(42) # see PyTorch CUDA (for NVIDIA GPUs)
torch.backends.cudnn.deterministic = True # tell PyTorch to use deterministic kernels
torch.backends.cudnn.benchmark = False # removes internal optimizations that can cause non-determinism
```

And when defining your outputs:

```
# replace np.random by the seeded RNG
- y = 5 * np.sin(0.1 * X_all) + np.random.randn(100, 1)
+ y = 5 * np.sin(0.1 * X_all) + rng.standard_normal(size=(100, 1))
```

A.5.5 Step 4: Visualizing

It is extremely important to [visualize](#) your data. For this, we can add this to the script:

Visualizing data

```
import matplotlib.pyplot as plt
```

and before the training loop:

```
plt.scatter(X_train, y_train, label="Train")
plt.scatter(X_test, y_test, color="red", label="Test")
plt.xlabel("X")
plt.ylabel("y")
plt.title("Data")
plt.legend()
plt.show()
```

Visualizing loss

```
# before training loop
losses = []

# in your training loop
losses.append(loss.item())

# after training loop
plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss over time")
plt.show()
```

Observations:

- The target is sinusoidal, a simple linear model cannot capture this (we've made bad assumptions for the model).
- The loss is producing nans and going to `inf`.
- The features are not normalized, making learning difficult.

A.5.6 Step 5: Normalizing Features

```

X_mean = X_train.mean(axis=0, keepdims=True)
X_std = X_train.std(axis=0, keepdims=True)
y_mean = y_train.mean(axis=0, keepdims=True)
y_std = y_train.std(axis=0, keepdims=True)
X_train_norm = (X_train - X_mean) / X_std
y_train_norm = (y_train - y_mean) / y_std
X_test_norm = (X_test - X_mean) / X_std
y_test_norm = (y_test - y_mean) / y_std

```

or using scikit-learn utils:

```

x_scaler = StandardScaler()
y_scaler = StandardScaler()
X_train_norm = x_scaler.fit_transform(X_train)
y_train_norm = y_scaler.fit_transform(y_train)
X_test_norm = x_scaler.transform(X_test)
y_test_norm = y_scaler.transform(y_test)

```

Use these normalized tensors in the training loop instead of the raw values.

A.5.7 Step 6: Visualizing Predictions

Now the loss seems to go down, the code runs. Let's look at the predictions. This code will help you visualize the predictions vs. the true values:

```

# after the training loop
with torch.no_grad():
    predictions = model(X_tensor)

plt.figure(figsize=(8, 5))
plt.scatter(X_tensor.numpy(), y_tensor.numpy(), label="True data", color="blue", alpha=0.5)
plt.scatter(
    X_tensor.numpy(), predictions.numpy(), label="Predictions", color="red", alpha=0.5
)
plt.xlabel("X")
plt.ylabel("y")
plt.title("True vs Predicted")
plt.legend()
plt.show()

```

Observation: It is pretty obvious that our model has not enough capacity to capture the data.

A.5.8 Step 7: Adjusting Hyperparameters

Let's try to increase the model size.

```
model = nn.Sequential(  
    nn.Linear(1, 16), nn.ReLU(), nn.Linear(16, 16), nn.ReLU(), nn.Linear(16, 1)  
)
```

And re-run. Now we see the loss is not optimal.

- Can you adjust the learning rate and model size?
- What about the optimizer?
- And how are you going to keep track of what combination of hyperparameter values you have tried?
- Also, you have several plots (prediction, loss) for each run which are helpful.

For this, we recommend using experiment trackers, such as [Weights and Biases](#) or [Trackio](#).

First, you start by defining your hyperparameters on top the file:

```
hyperparameters = {  
    "learning_rate": 0.01,  
    "model_layers": [16, 16],  
    "activation": "ReLU",  
}
```

and use them in your training script. For instance, your model definition becomes

```
model_layers: list[int] = hyperparameters["model_layers"]  
layers = []  
  
# Build all layers including input and hidden layers -- this allows to just change the model_layers in your h  
current_size = 1  
for layer_size in model_layers:  
    layers.append(nn.Linear(current_size, layer_size))  
    if hyperparameters["activation"] == "ReLU":  
        layers.append(nn.ReLU())  
    elif hyperparameters["activation"] == "Sigmoid":  
        layers.append(nn.Sigmoid())  
    current_size = layer_size  
  
# Add output layer  
layers.append(nn.Linear(current_size, 1))
```

```
model = nn.Sequential(*layers)
optimizer = optim.SGD(model.parameters(), lr=hyperparameters["learning_rate"]) # see hyperparameter here
```

Then, you log these hyperparameters for each experiment:

```
wandb.init(project="example", config=hyperparameters)
```

In your training loop:

```
wandb.log({"loss": loss.item()})
```

You can even log an image of prediction vs. true data at each training step. See below.

A.5.9 Final Code

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import trackio as wandb
from sklearn.model_selection import train_test_split

rng = np.random.default_rng(42) # seed NumPy random
torch.manual_seed(42) # seed PyTorch
torch.cuda.manual_seed(42) # see PyTorch CUDA (for NVIDIA GPUs)
torch.backends.cudnn.deterministic = True # tell PyTorch to use deterministic kernels
torch.backends.cudnn.benchmark = (
    False # removes internal optimizations that can cause non-determinism
)

hyperparameters = {
    "learning_rate": 0.01,
    "model_layers": [16, 16],
    "activation": "ReLU",
}

def visualize_data(
```



```

model: nn.Module,
X_test: np.ndarray,
y_test: np.ndarray,
epoch: int,
open_window: bool = False,
log: bool = False,
):
    """Visualize the data and the predictions.

    Args:
        model: The model to visualize the predictions of.
        X_test: The test data.
        y_test: The test labels.
        epoch: The epoch number.
        open_window: Whether to open a window to display the plot.
        log: Whether to log the plot to wandb.
    """
    with torch.no_grad():
        predictions = model(torch.tensor(X_test, dtype=torch.float32))
    plt.scatter(X_test, y_test, color="red", alpha=0.5, label="Data")
    plt.scatter(
        X_test, predictions.numpy(), color="blue", alpha=0.5, label="Predictions"
    )
    plt.xlabel("X")
    plt.ylabel("y")
    plt.title(f>Data - Epoch {epoch}")
    plt.legend()
    if log:
        plt.savefig("predictions.png")
        wandb.log({"predictions": wandb.Image("predictions.png")})
    if open_window:
        plt.show()
    plt.close()

if __name__ == "__main__":
    wandb.init(project="example", config=hyperparameters)

    X_all = np.linspace(0, 100, 500).reshape(-1, 1)
    y_all = 5 * np.sin(0.1 * X_all) + rng.standard_normal(size=(500, 1))

    X_train, X_test, y_train, y_test = train_test_split(

```

```

    X_all, y_all, test_size=0.2, random_state=42
)

plt.scatter(X_train, y_train, label="Train")
plt.scatter(X_test, y_test, color="red", label="Test")
plt.xlabel("X")
plt.ylabel("y")
plt.title("Data")
plt.legend()
plt.show()

# Normalize the data
X_mean = X_train.mean(axis=0, keepdims=True)
X_std = X_train.std(axis=0, keepdims=True)
y_mean = y_train.mean(axis=0, keepdims=True)
y_std = y_train.std(axis=0, keepdims=True)
X_train_norm = (X_train - X_mean) / X_std
y_train_norm = (y_train - y_mean) / y_std
X_test_norm = (X_test - X_mean) / X_std
y_test_norm = (y_test - y_mean) / y_std

model_layers: list[int] = hyperparameters["model_layers"]
layers = []

# Build all layers including input and hidden layers
current_size = 1
for layer_size in model_layers:
    layers.append(nn.Linear(current_size, layer_size))
    if hyperparameters["activation"] == "ReLU":
        layers.append(nn.ReLU())
    elif hyperparameters["activation"] == "Sigmoid":
        layers.append(nn.Sigmoid())
    current_size = layer_size

# Add output layer
layers.append(nn.Linear(current_size, 1))

model = nn.Sequential(*layers)
optimizer = optim.Adam(model.parameters(), lr=hyperparameters["learning_rate"])
loss_fn = nn.MSELoss()

losses = []

```

```

for epoch in range(500):
    pred = model(torch.tensor(X_train_norm, dtype=torch.float32))
    loss = loss_fn(pred, torch.tensor(y_train_norm, dtype=torch.float32))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch} - Loss: {loss.item()}")
    wandb.log({"loss": loss.item()})
    losses.append(loss.item())
    if epoch % 20 == 0:
        visualize_data(
            model,
            X_test_norm,
            y_test_norm,
            epoch=epoch,
            open_window=False,
            log=True,
        )

plt.plot(losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss over time")
plt.show()

# Show predictions plot at the end of training
visualize_data(
    model, X_test_norm, y_test_norm, epoch=500, open_window=True, log=True
)
wandb.finish()

```

B Exercise Solutions

Worked solutions for the end-of-chapter exercises are published as a separate, shorter book so that this one stays readable in print. Each solutions chapter follows the numbering used in the corresponding chapter here and reuses that chapter's data and code. Try the exercises before reading the solutions; most of the value is in the attempt.

The solutions are available online next to the book, in the `solutions/` folder, and as a separate PDF.

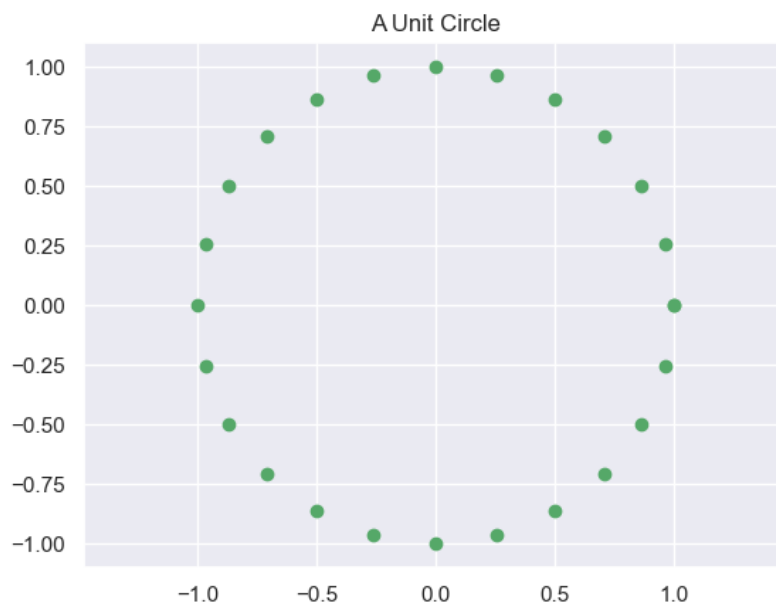
C Review of Matrices and the Singular Value Decomposition

For more info after class on fundamental matrix properties, see the [Matrix Cookbook](#)

Let's generate a simple circle of points, so that we can see what Matrices do to objects:

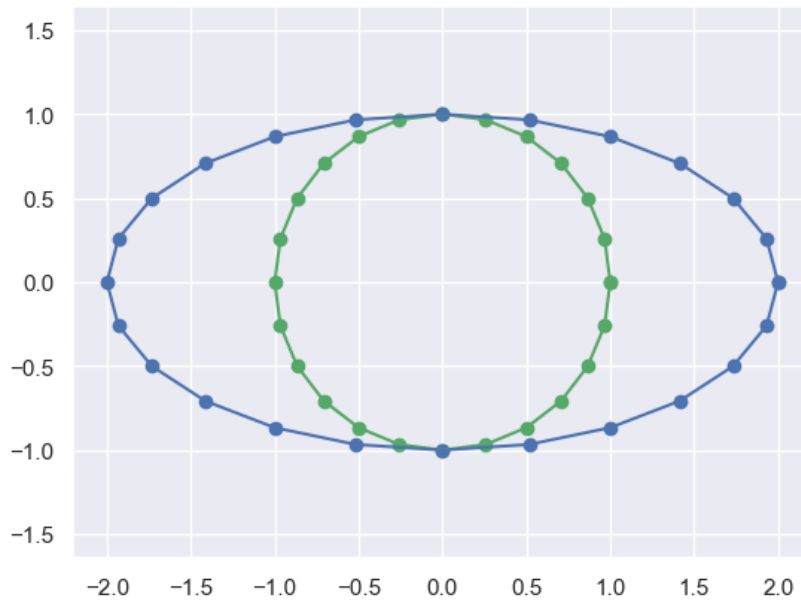
X: (2, 25)

```
[[ 1.    0.966 0.866 ... 0.866 0.966 1.    ]  
 [ 0.    0.259 0.5   ... -0.5  -0.259 -0.    ]]
```

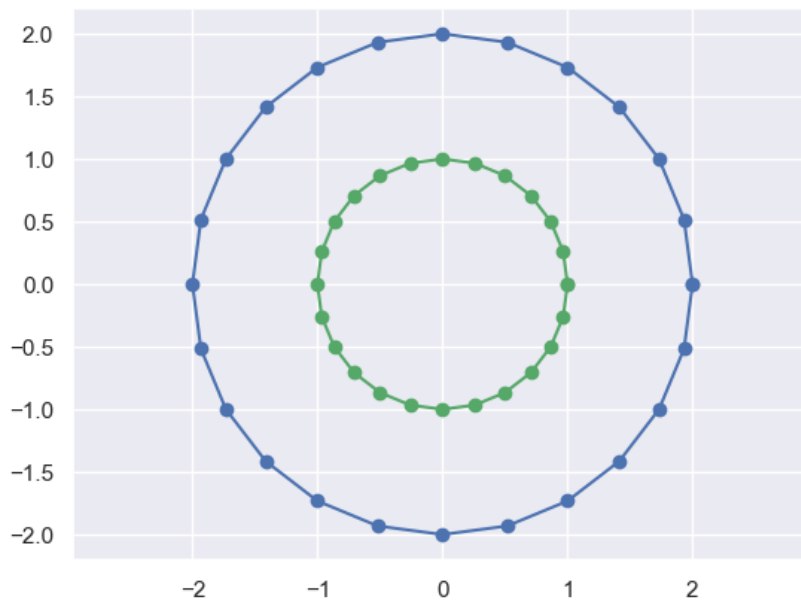


M

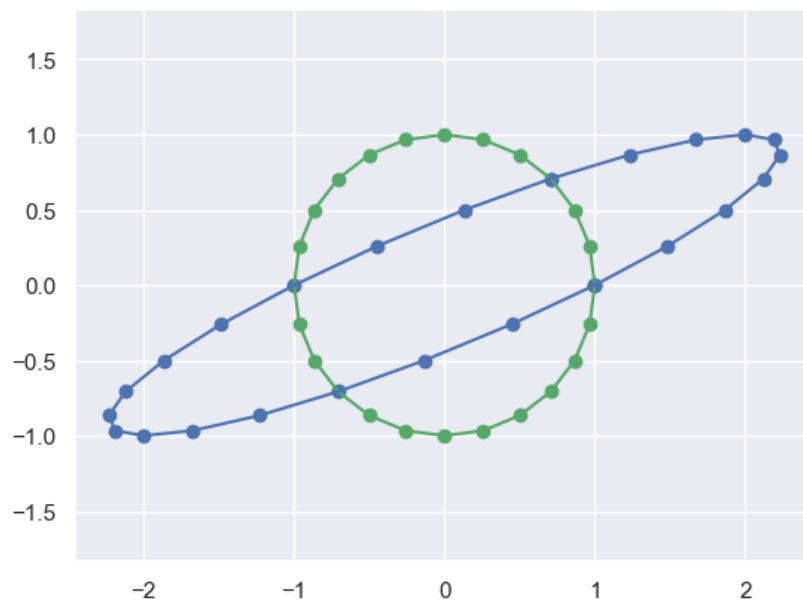
```
[[2 0]  
 [0 1]]
```

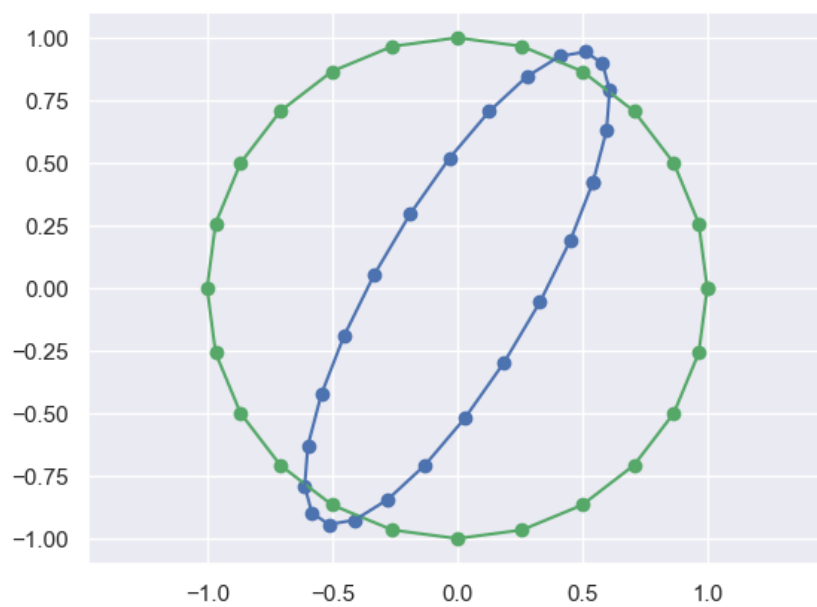


M
 $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$



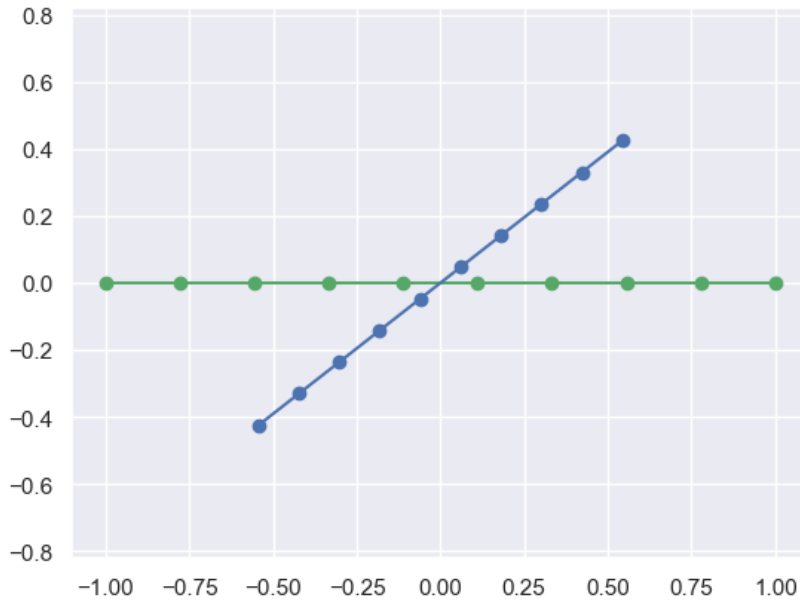
M

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$


$$\begin{bmatrix} 0.5 & 0.3 \\ 0.4 & 0.8 \end{bmatrix}$$


You can do this same thing to different dimensions of inputs, such as 1-D (lines):

```
X_line:
[-1. -0.8 -0.6 ... 0.6 0.8 1. ]
Transformed:
[[-0.5 -0.4 -0.3 ... 0.3 0.4 0.5]
 [-0.4 -0.3 -0.2 ... 0.2 0.3 0.4]]
```



C.1 The Singular Value Decomposition

For N data points of d dimensions:

$$X_{d \times N} = U_{d \times d} \Sigma_{d \times N} V_{N \times N}^*$$

Where U , V are [orthogonal](#) ($UU^T = I$), and Σ is diagonal.

```
matrix([[ 0.5,  0.6,  0.6, ...,  0.3,  0.5,  0.5],
        [ 0.4,  0.6,  0.8, ..., -0.1,  0.2,  0.4]])
```

U: (2, 2)

S: (2, 2)

V: (2, 25)


```

S =
[[3.8 0. ]
 [0.  1.1]]
U*U.T =
[[1. 0.]
 [0. 1.]]

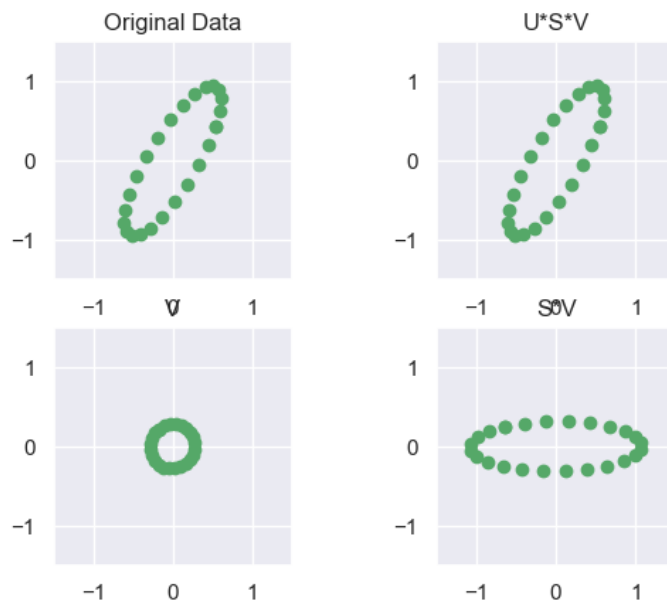
matrix([[-0.2, -0.2, -0.3, ..., -0. , -0.1, -0.2],
        [ 0.2,  0.2,  0.1, ...,  0.3,  0.3,  0.2]])

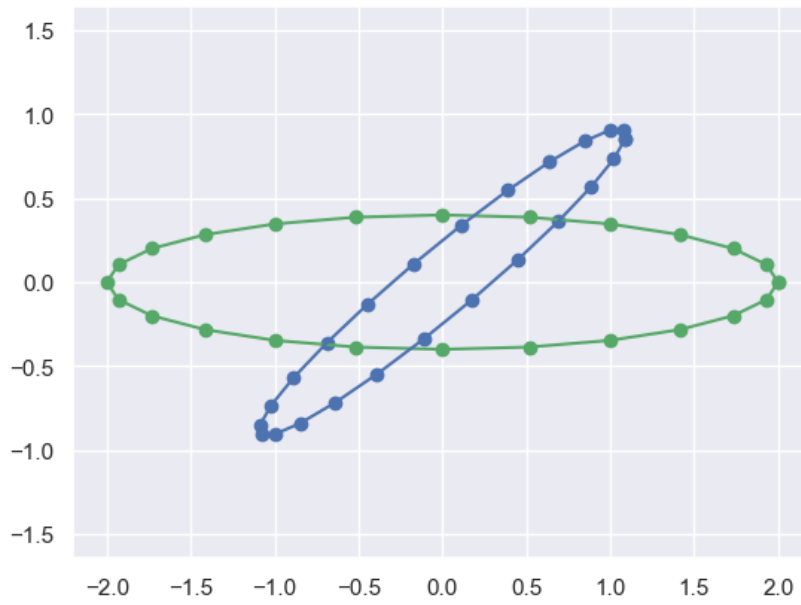
```

```

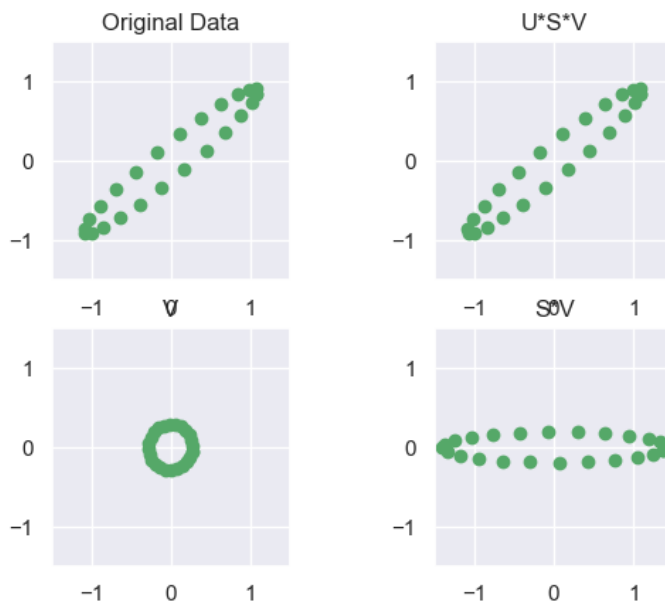
S =
[[3.8 0. ]
 [0.  1.1]]

```





S =
[[5.1 0.]
[0. 0.7]]



C.2 What about different dimensions?

```
M = [[1.  0. ]
      [0.  1. ]
      [2.  0.5]]
r = [[1]
      [1]]
M*r = [[1. ]
        [1. ]
        [2.5]]
```

[Interactive output: available in the online version of this chapter.]

```
matrix([[ 1. ,  1. ,  0.9, ...,  0.9,  1. ,  1. ],
        [ 0. ,  0.3,  0.5, ..., -0.5, -0.3, -0. ],
        [ 2. ,  2.1,  2. , ...,  1.5,  1.8,  2. ]])

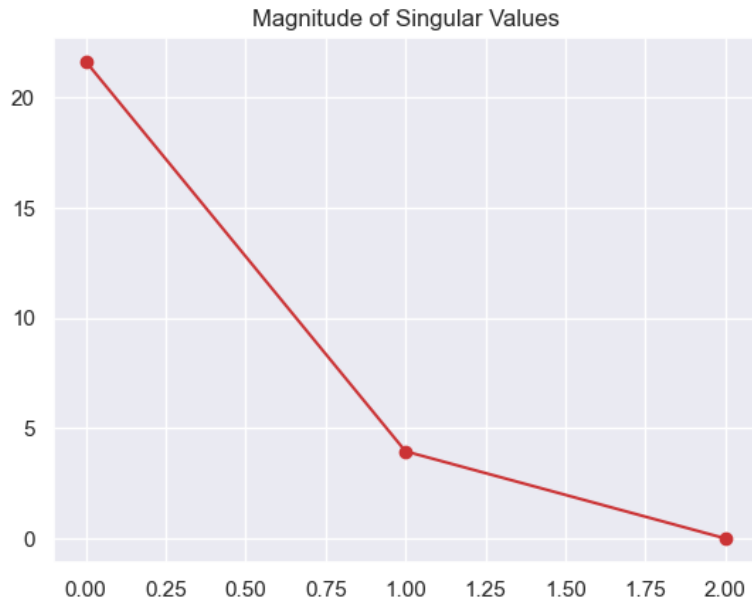
[[ 1.  2. ]
 [-2.  2. ]
 [ 2.  0.5]]
```

[Interactive output: available in the online version of this chapter.]

```
U: (3, 3)
S: (3, 3)
V: (3, 25)
S =
[[21.6  0.  0. ]
 [ 0.  4.  0. ]
 [ 0.  0.  0. ]]
```

```
matrix([[ -7.1,  2.9, -0. ],
        [ 14.5,  2.5,  0. ],
        [-14.4,  1. ,  0. ]])
```

```
matrix([[-0.3, -0.3, -0.2, ..., -0.2, -0.3, -0.3],
        [ 0. ,  0.1,  0.1, ..., -0.1, -0.1,  0. ],
        [-0.8,  0.5,  0.1, ..., -0. ,  0.1,  0. ]])
```



[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

[Interactive output: available in the online version of this chapter.]

```
S =
[[21.6  0.  0. ]
 [ 0.  4.  0. ]
 [ 0.  0.  0. ]]
V =
[[-0.3 -0.3 -0.2 ... -0.2 -0.3 -0.3]
 [ 0.  0.1 0.1 ... -0.1 -0.1 0. ]
 [-0.8 0.5 0.1 ... -0.  0.1 0. ]]
S*V =
[[-6.  -5.8 -5.1 ... -5.3 -5.8 -6. ]
 [ 0.  0.3 0.6 ... -0.5 -0.3 0. ]
 [-0.  0.  0.  ... -0.  0.  0. ]]
```

```
St =
[[21.6  0. ]
 [ 0.  4. ]
 [ 0.  0. ]]
Vt =
[[-0.3 -0.3 -0.2 ... -0.2 -0.3 -0.3]
```

```

[ 0.  0.1  0.1 ... -0.1 -0.1  0. ]]
St*Vt =
[[-6.  -5.8 -5.1 ... -5.3 -5.8 -6. ]
 [ 0.   0.3  0.6 ... -0.5 -0.3  0. ]
 [ 0.   0.   0.   ...  0.   0.   0. ]]

```

```

U =
[[-0.3  0.7 -0.6]
 [ 0.7  0.6  0.4]
 [-0.7  0.3  0.7]]

```

```

Ut =
[[-0.3  0.7]
 [ 0.7  0.6]
 [-0.7  0.3]]
St =
[[21.6  0. ]
 [ 0.   4. ]]
Vt =
[[-0.3 -0.3 -0.2 ... -0.2 -0.3 -0.3]
 [ 0.   0.1  0.1 ... -0.1 -0.1  0. ]]

```

```

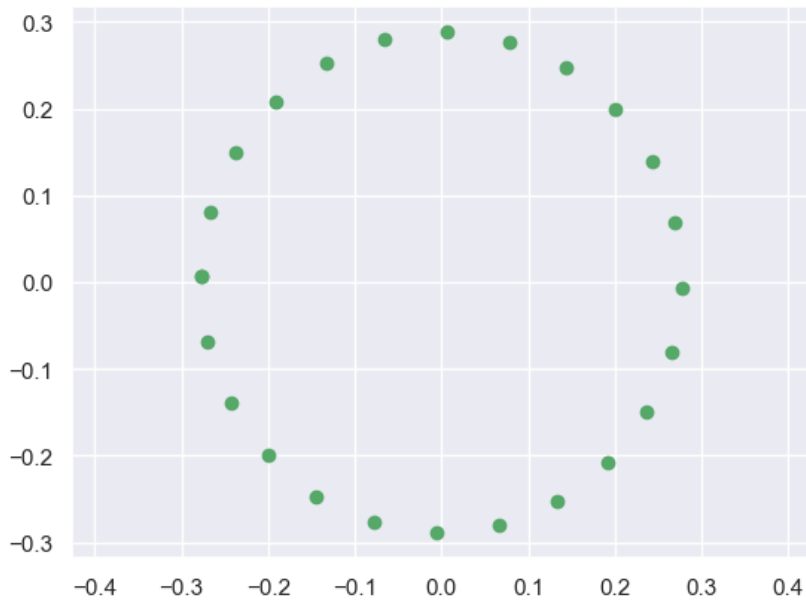
Ut*St*Vt =
[[ 2.   2.1  2.1 ...  1.3  1.7  2. ]
 [-4.  -3.7 -3.1 ... -3.9 -4.1 -4. ]
 [ 4.   3.9  3.6 ...  3.4  3.8  4. ]]
M*Y =
[[ 2.   2.1  2.1 ...  1.3  1.7  2. ]
 [-4.  -3.7 -3.1 ... -3.9 -4.1 -4. ]
 [ 4.   3.9  3.6 ...  3.4  3.8  4. ]]
Are they equal?: True

```

```

[[-0.3 -0.3 -0.2 ... -0.2 -0.3 -0.3]
 [ 0.   0.1  0.1 ... -0.1 -0.1  0. ]]

```



[Interactive output: available in the online version of this chapter.]

```
S =
[[11.5  0.  0. ]
 [ 0.  9.7  0. ]
 [ 0.  0.  2.8]]

[[ 1.5  1.7  2.  ...  0.2  1.1  0.8]
 [-2.  -1.3 -1.5 ... -3.  -2.9 -2.4]
 [ 2.1  2.3  1.6 ...  1.9  1.1  2.3]]

[[ 1.3  1.7  1.5 ...  0.2  0.5  0.9]
 [-1.9 -1.3 -1.2 ... -3.  -2.5 -2.5]
 [ 2.3  2.3  2.  ...  1.9  1.9  2.2]]
```

Are they equal?: False

[Interactive output: available in the online version of this chapter.]

The idea of uncovering structure, or reducing data-dimensions is one key goal of Unsupervised Learning. In particular, the SVD (among other methods) can be used for [Principal Component Analysis](#): reducing the number of dimensions of a data-set, by finding a linear transformation to a smaller orthogonal basis which minimizes reconstruction error to the original space.



D Reviewing Mathematical and Computational Foundations for Machine Learning

This notebook is designed to help you build (or refresh) the conceptual and computational foundations needed for understanding probabilistic generative modeling methods and other important topics in Machine Learning. You can look at each sub-section independently and in any order to refresh your knowledge of some foundations, though we may build upon certain common examples throughout.

Learning Objectives

By the end of this notebook, you will be able to:

1. Understand random variables and Monte Carlo sampling techniques
2. Apply change of variables transformations and compute Jacobians
3. Work with linear algebra concepts relevant to probabilistic models, such as spectral norms of matrices
4. Compute gradients and understand curvature in optimization landscapes
5. Perform marginalization operations in probabilistic settings

D.1 Random Variables, Monte Carlo, and Concentration Inequalities

D.1.1 Concepts:

- Random variables are functions that map outcomes to numerical values
- **Expectations** represent the average value of a function over a probability distribution
- **Monte Carlo methods** use random sampling to approximate expectations and integrals
- The **Law of Large Numbers** ensures that sample averages converge to the true expectation as sample size increases
- Convergence rate is typically $O(1/\sqrt{N})$, meaning we need 4x more samples to halve the error
- Finite samples introduce uncertainty that decreases with more data and **Concentration Inequalities** (Chebyshev, Hoeffding) bound the probability of large deviations, which can help us understand real-world machine learning behavior

D.1.2 Key Equations:

Expectation via Monte Carlo:

$$\mathbb{E}_{p(x)}[f(X)] = \int f(x)p(x)dx \approx \frac{1}{N} \sum_{i=1}^N f(x_i), \quad x_i \sim p(x)$$

Intuition: If we can sample from a distribution, we can approximate any expectation by averaging function values over those samples. The approximation improves as we collect more samples.

Law of Large Numbers:

$$\bar{X}_N = \frac{1}{N} \sum_{i=1}^N X_i \xrightarrow{N \rightarrow \infty} \mathbb{E}[X]$$

Chebyshev's Inequality:

$$P(|\bar{X}_N - \mu| \geq \epsilon) \leq \frac{\sigma^2}{N\epsilon^2}$$

Hoeffding's Inequality (for bounded r.v. in $[a, b]$):

$$P(|\bar{X}_N - \mu| \geq \epsilon) \leq 2 \exp\left(-\frac{2N\epsilon^2}{(b-a)^2}\right)$$

Understanding convergence of statistical estimates of various functions will end up being foundational to many aspects of Machine Learning, so it is useful for us to build intuition about it ahead of time.

Intuition: These inequalities tell us how confident we can be that our sample mean is close to the true mean. They guarantee exponential or polynomial concentration as N grows.

D.1.3 Monte Carlo Mean Estimation

We'll sample from a standard normal distribution $\mathcal{N}(0,1)$ and watch how the sample mean converges to the true mean (which is 0) as we increase the number of samples.

```
# Set seed for reproducibility
np.random.seed(42)

# True parameters
true_mean = 0.0
true_std = 1.0
```

```
# Sample sizes to test
sample_sizes = np.logspace(1, 4, 50).astype(int) # From 10 to 10,000 samples

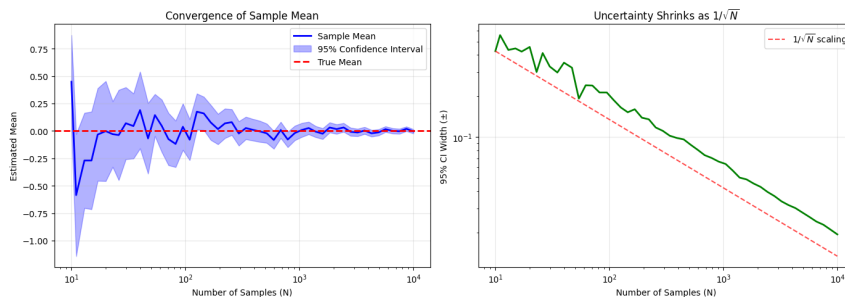
# Compute sample means for increasing sample sizes
sample_means = []
confidence_intervals = []

for N in sample_sizes:
    # Draw N samples from standard normal
    samples = np.random.normal(true_mean, true_std, N)

    # Compute sample mean
    sample_mean = np.mean(samples)
    sample_means.append(sample_mean)

    # Compute 95% confidence interval ( $\pm 1.96$  * standard error)
    standard_error = np.std(samples) / np.sqrt(N)
    ci = 1.96 * standard_error
    confidence_intervals.append(ci)

sample_means = np.array(sample_means)
confidence_intervals = np.array(confidence_intervals)
```



Notice how the sample mean oscillates around the true value but gradually settles down. The confidence interval shrinks, but not linearly—it follows a $1/\sqrt{N}$ pattern. This means:

- To halve your error, you need 4x more samples
- To reduce error by 10x, you need 100x more samples

Why do you think the convergence is so slow? What does this tell you about the computational cost of Monte Carlo methods, in general?

D.1.4 Expectation of a Nonlinear Function

So far we've estimated the mean of a simple identity function. But Monte Carlo really shines when computing expectations of complex, nonlinear functions. Let's estimate $\mathbb{E}[X^2]$ where $X \sim \mathcal{N}(0, 1)$.

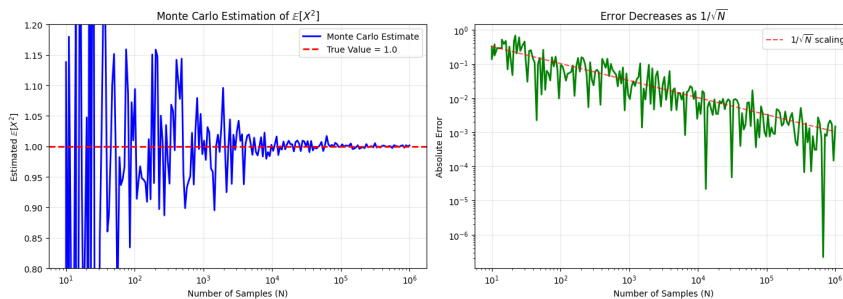
```
# For a standard normal,  $\mathbb{E}[X^2] = \text{Var}(X) + \mathbb{E}[X]^2 = 1 + 0 = 1$ 
true_expectation = 1.0

# Generate samples and compute  $\mathbb{E}[X^2]$ 
np.random.seed(42)
sample_sizes = np.logspace(1, 6, 200).astype(int)

# Function we want to compute the expectation of
def f(x):
    return x**2

# Monte Carlo estimates
mc_estimates = []
for N in sample_sizes:
    samples = np.random.normal(0, 1, N)
    estimate = np.mean(f(samples))
    mc_estimates.append(estimate)

mc_estimates = np.array(mc_estimates)
```



D.2 Change of Variables & Jacobians

D.2.1 Concepts:

- When you transform a random variable, the probability density changes

- The **Jacobian matrix** captures how a transformation stretches or compresses space
- The **determinant of the Jacobian** computes how the volume scales in the new space operated on by the Jacobian
- This principle underlies **normalizing flows** and other generative models
- In 1D: $|\frac{dy}{dx}|$ tells us how density scales; in higher dimensions we use $|\det J|$

D.2.2 Key Equations:

Change of Variables Formula (1D):

$$p_Y(y) = p_X(x) \left| \frac{dx}{dy} \right| = p_X(g^{-1}(y)) \left| \frac{dg^{-1}}{dy} \right|$$

where $y = g(x)$ is a transformation.

Change of Variables Formula (Multidimensional):

$$p_Y(y) = p_X(x) \left| \det \frac{\partial x}{\partial y} \right| = p_X(g^{-1}(y)) |\det J_{g^{-1}}(y)|$$

Intuition: When you transform a random variable, the probability density must be adjusted by how much the transformation stretches or compresses space. Think of it like this: if a transformation spreads points apart (stretches), the density must decrease to keep total probability = 1.

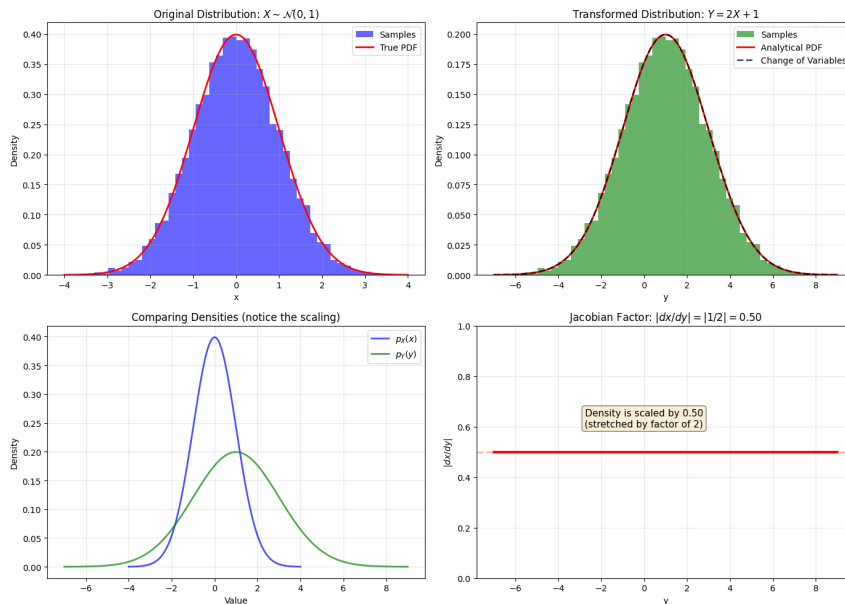
D.2.3 Example: Linear Transformation (1D)

Let's start with a simple linear transformation: $y = 2x + 1$. We'll sample from a standard normal and see how the distribution changes. Recall that for a normal distribution $\mathcal{N}(\mu, \sigma^2)$, a linear transformation $y = ax + b$ results in $y \sim \mathcal{N}(a\mu + b, (a\sigma)^2)$.

In this example, we should expect the expected mean of Y to be 1 and the standard deviation to be 2. Moreover, we should expect the density scaling factor from the Jacobian to be:

$$\left| \frac{dx}{dy} \right| = \frac{1}{a} = 0.5$$

which should be multiplied into the original density in the plots below.



Sample mean of Y: 0.9957 (expected: 1)
Sample std of Y: 2.0068 (expected: 2.0000)

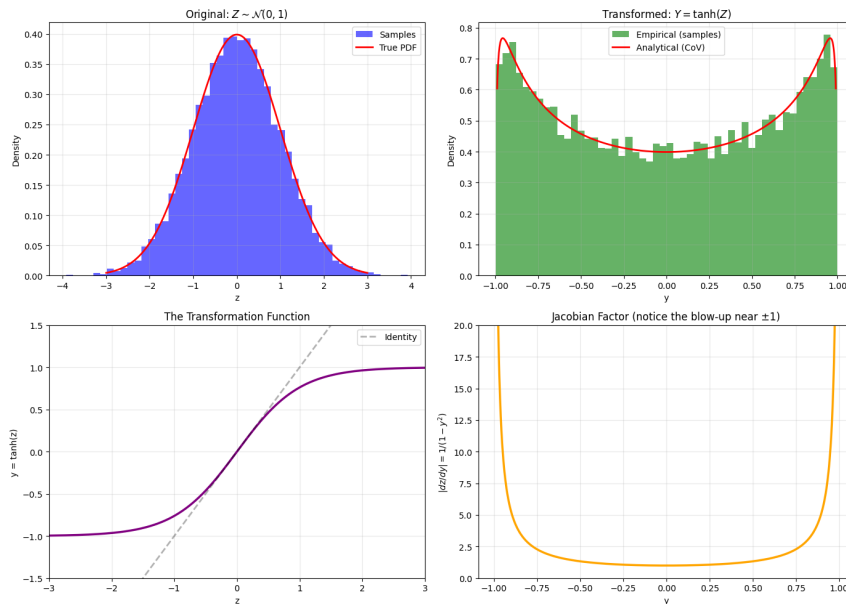
Jacobian factor $|dx/dy| = 0.5000$
Density is compressed by factor of 2

Notice how the transformation $y = 2x + 1$ shifts the mean (the $+1$ part) and stretches the distribution (the $2x$ part). The density gets compressed because the same probability mass is now spread over a wider range.

The Jacobian factor $|dx/dy| = 1/2$ tells us exactly how much to scale the density.

D.2.4 Example: Nonlinear Transformation

Linear transformations are straightforward, but what about nonlinear ones? Let's try $y = \tanh(x)$, which squashes values into the range $(-1, 1)$.



The tanh function compresses the tails of the Gaussian distribution into a bounded range. Notice how the Jacobian factor $|dz/dy| = 1/(1 - y^2)$ grows very large near $y = \pm 1$. This creates the characteristic “U-shaped” density—probability piles up where the transformation compresses most severely.

This type of transformation is common in neural networks (e.g., in activation functions). What do you think would happen if we used a transformation that wasn’t invertible?

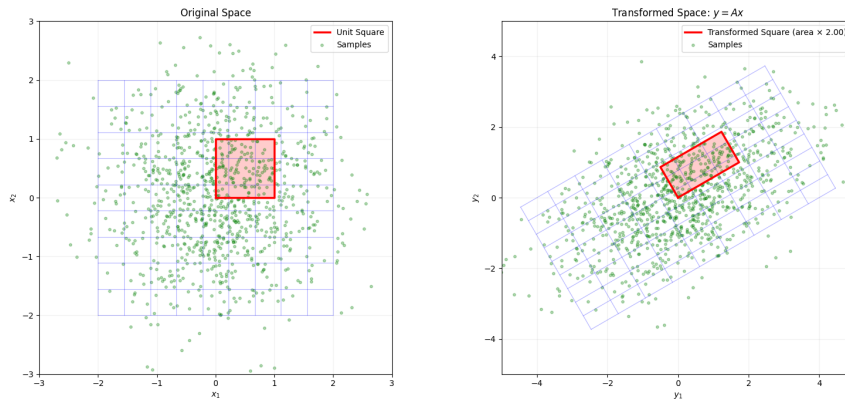
D.2.5 Example: 2D Transformation and Grid Deformation

In higher dimensions, the Jacobian is a matrix, and its determinant tells us about volume changes. Let’s visualize this with a 2D linear transformation.

Transformation matrix A:

```
[[ 1.73205081 -0.5      ]
 [ 1.         0.8660254 ]]
```

Determinant (area scaling factor): 2.0000

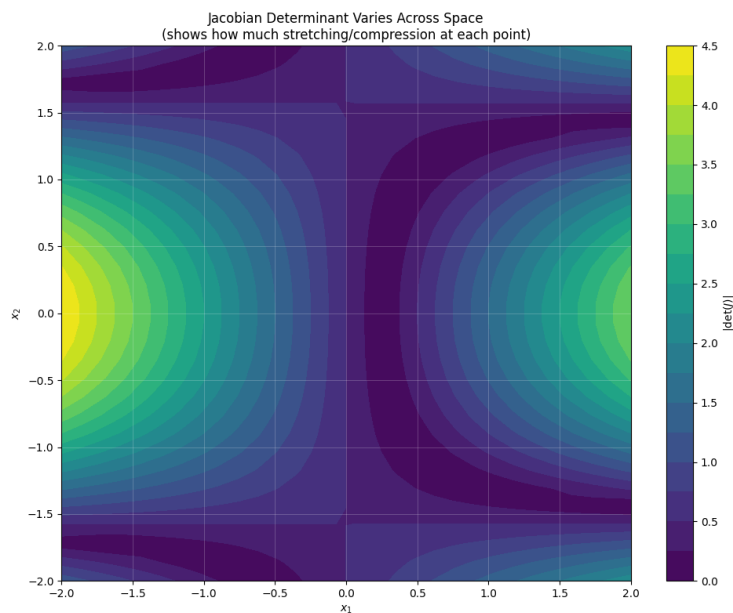


Original unit square area: 1.0000
 Transformed square area: 2.0000
 Area scaling factor: 2.0000

This means density must be scaled by 0.5000 to preserve probability mass

D.2.6 Example: Computing Jacobians with PyTorch

Let's use PyTorch's automatic differentiation to compute Jacobians for us. This is especially useful for complex non-linear functions for which writing out the analytical gradients might not be practical.



Understanding how to compute and interpret Jacobians is useful in many modern machine learning models. The Jacobian determinant tells us the local volume scaling at each point. For nonlinear transformations, this varies across space—some regions expand while others compress.

This is the core principle used in **normalizing flows**: by chaining multiple invertible transformations (each with computable Jacobians), we can transform a simple distribution (like a Gaussian) into arbitrarily complex distributions. The key requirement is that we can compute $|\det(J)|$ efficiently.

Can you think of why we need the transformation to be invertible for normalizing flows? How would you chain multiple transformations together? What happens to the overall Jacobian determinant? (We will revisit these in the relevant generative models chapter.)

D.3 Linear Algebra & Spectral Norms

D.3.1 Concepts:

- Matrices represent **linear transformations** that stretch, rotate, and compress space
- **Singular values** measure how much a matrix stretches along different directions
- The **spectral norm** (largest singular value) gives the maximum stretching factor
- **Matrix conditioning** (ratio of largest to smallest singular value) indicates numerical stability
- Understanding these geometric properties is essential for analyzing neural networks and optimization

D.3.2 Key Equations:

Singular Value Decomposition:

$$A = U\Sigma V^T$$

where U, V are orthogonal and Σ contains singular values $\sigma_1 \geq \sigma_2 \geq \dots \geq 0$

Spectral Norm (Matrix 2-norm):

$$\|A\|_2 = \sigma_{\max}(A) = \max_{\|x\|=1} \|Ax\|$$

Condition Number:

$$\kappa(A) = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)}$$

Intuition: The spectral norm tells you “how much can this transformation amplify a vector?” Large spectral norms mean the transformation can greatly magnify small changes in input—this is crucial for understanding gradient flow in deep networks.

D.3.3 Example: 2D Linear Transform Visualization

Let's see how a matrix transforms space by watching what it does to a regular grid. This gives us intuition for singular values and the spectral norm.

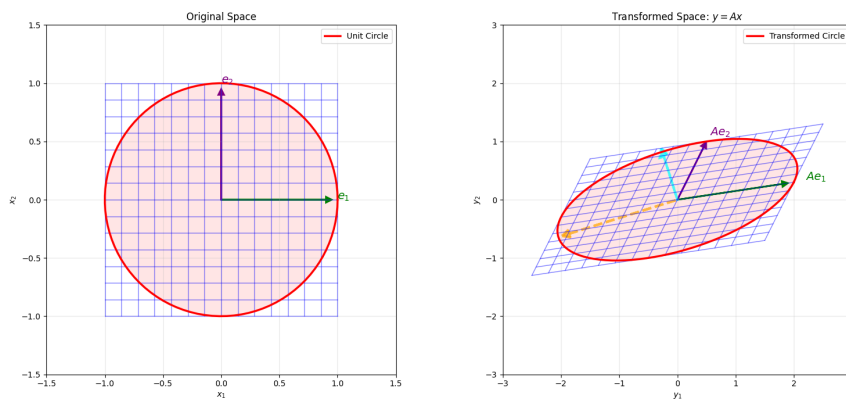
Transformation matrix A:

```
[[2.  0.5]
 [0.3 1. ]]
```

Singular values: [2.14364206 0.86301721]

Spectral norm (max singular value): 2.1436

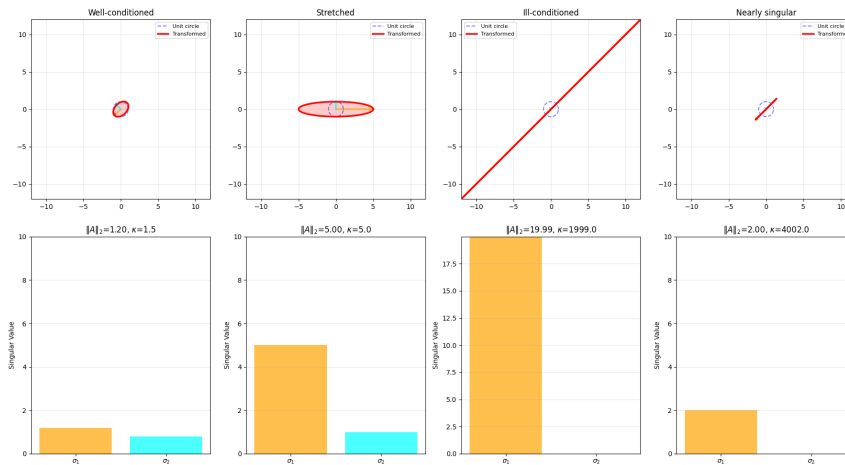
Condition number: 2.4839



The transformation turns a circle into an ellipse. The singular values tell us exactly how much the matrix stretches in its principal directions. The spectral norm is the maximum stretching factor, which in the above example is 2.1436 and the same as the ellipse's semi-major axis length, with 0.863 being the semi-minor axis length. This is crucial for understanding how errors propagate through linear transformations. For example, in deep learning, if weight matrices have large spectral norms, small input perturbations can get amplified dramatically. What problems do you think this might cause?

D.3.4 Example: Spectral Norms and Condition Numbers of Different Matrices

Let's explore how different matrices have different spectral properties and what that means for numerical stability.



Matrix Analysis Summary

Name	$\ A\ _2$	$\kappa(A)$	$\det(A)$
Well-conditioned	1.2000	1.50	0.960000
Stretched	5.0000	5.00	5.000000
Ill-conditioned	19.9900	1999.00	0.199900
Nearly singular	2.0005	4002.00	0.001000

We see that for well-conditioned matrices, they have a small condition number that barely amplifies or stretches the original data. As we increase the condition number, small perturbations in the input can cause large changes in the output matrix. However, even if the spectral norm is small, the matrix can still have a large condition number, which can make the matrix difficult to invert or compute stable gradients with respect to.

D.3.5 Example: Neural Network Jacobian and Local Geometry

Neural networks are compositions of nonlinear transformations. At any point, the local behavior is captured by the Jacobian matrix. Let's visualize how a simple neural network transforms space locally.

```
# Define a simple 2-layer neural network:  $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ 
class SimpleNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2, 8, bias=True)
        self.fc2 = nn.Linear(8, 2, bias=True)
```

```

def forward(self, x):
    x = torch.tanh(self.fc1(x))
    x = self.fc2(x)
    return x

# Initialize network with specific seed
torch.manual_seed(42)
net = SimpleNet()

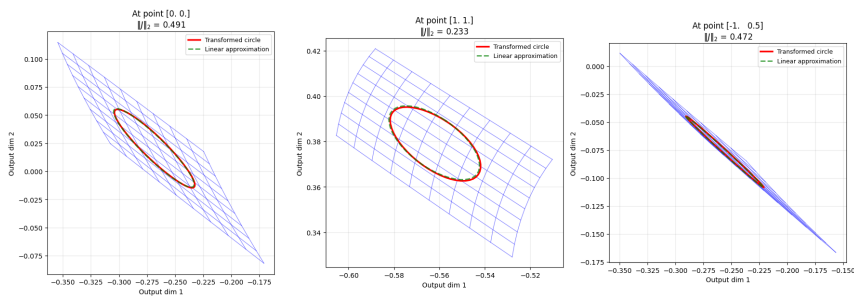
# Function to compute Jacobian at a point
def compute_jacobian(model, x):
    """Compute Jacobian matrix of model output w.r.t. input at point x"""
    x_tensor = torch.tensor(x, dtype=torch.float32, requires_grad=True)
    y = model(x_tensor.unsqueeze(0))

    jacobian = []
    for i in range(y.shape[1]):
        grad = torch.autograd.grad(y[0, i], x_tensor,
                                    create_graph=True, retain_graph=True)[0]
        jacobian.append(grad.detach().numpy())

    return np.array(jacobian)

# Visualize transformation at different points in space
test_points = [
    np.array([0.0, 0.0]),
    np.array([1.0, 1.0]),
    np.array([-1.0, 0.5]),
]

```



At point [0. 0.]:

```
Jacobian matrix:
[[-0.12561867 -0.3354176 ]
 [ 0.23783804  0.2557892 ]]
Singular values: [0.4907707  0.09707827]
Spectral norm: 0.4908
Condition number: 5.0554
```

```
At point [1. 1.]:
Jacobian matrix:
[[ 0.04538195 -0.19429947]
 [ 0.10090272  0.12735607]]
Singular values: [0.23314807 0.10887937]
Spectral norm: 0.2331
Condition number: 2.1413
```

```
At point [-1.  0.5]:
Jacobian matrix:
[[-0.20401104 -0.28298327]
 [ 0.20504908  0.24390957]]
Singular values: [0.47215527 0.01750529]
Spectral norm: 0.4722
Condition number: 26.9721
```

We can see that the neural network’s Jacobian is changing across the input space. The Jacobian tells us how the neural network behaves locally—like a linear approximation at each point. The red curve shows the actual transformation of a small circle, while the green dashed line shows what the Jacobian predicts. When they match well, the network is locally linear; when they diverge, the nonlinearity is strong.

D.3.6 Example: Spectral Normalization

Spectral normalization is a technique to control the Lipschitz constant of neural networks by constraining weight matrices to have spectral norm ≤ 1 . Let’s see how this affects the transformation.

```
# Function to apply spectral normalization to a weight matrix
def spectral_normalize(W, n_iterations=1):
    """
    Normalize a weight matrix so its spectral norm is 1.
    Uses power iteration to estimate the largest singular value.
    """
    U, s, Vt = np.linalg.svd(W, full_matrices=False)
```

```

sigma_max = s[0]
return W / sigma_max, sigma_max

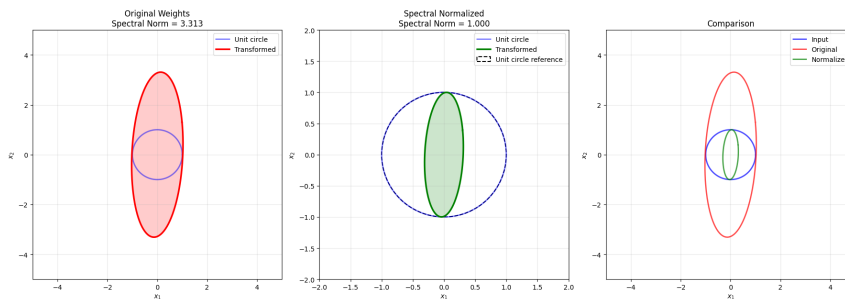
# Create two versions of a weight matrix
np.random.seed(42)
W_original = np.random.randn(2, 2) * 2 # Scaled up for effect

# Apply spectral normalization
W_normalized, original_spectral_norm = spectral_normalize(W_original)

```

Original weight matrix:
[[0.99342831 -0.2765286]
[1.29537708 3.04605971]]
Spectral norm: 3.3131

Spectral normalized weight matrix:
[[0.29985151 -0.08346603]
[0.39099024 0.91940767]]
Spectral norm: 1.0000



Spectral normalization ensures that no input vector gets amplified by more than a factor of 1. This is useful for training stability in deep networks since repeated multiplications by weight matrices with large spectral norms can cause gradients to explode. It can also have benefits in cases where we might wish to bound the amount that a given input perturbation can affect a network (e.g., for adversarial robustness).

D.4 Differentiation & Curvature

D.4.1 Concepts:

- **Gradients** point in the direction of steepest ascent and guide first-order optimization

- **Hessians** capture curvature (second-order information) about the function's shape
- **Jacobian-vector products (JVPs)** and **Hessian-vector products (HVPs)** allow efficient computation without forming full matrices
- Curvature determines optimization difficulty: flat regions are slow, steep valleys cause oscillation
- Understanding higher-order structure is useful for advanced optimization and model analysis

D.4.2 Key Equations:

Gradient (First derivative):

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

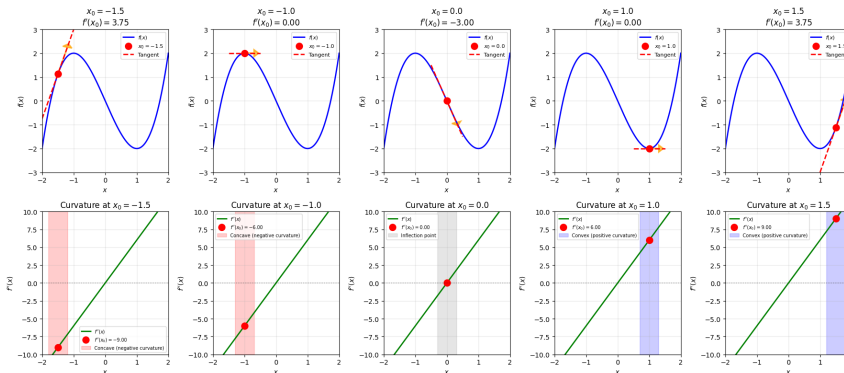
Hessian (Second derivative matrix):

$$H(f)(x) = \left[\frac{\partial^2 f}{\partial x_i \partial x_j} \right]$$

Intuition: The gradient tells you which way is “downhill,” but the Hessian tells you how quickly the slope is changing. Various optimizers like Newton’s method or L-BFGS can use curvature to take smarter steps than simple gradient descent.

D.4.3 Example: Gradient and Hessian Visualization

Let’s visualize how the gradient (slope) and Hessian (curvature) change at different points along a function. We’ll use $f(x) = x^3 - 3x$ and detect different gradient and curvature regions.



The gradient tells you the instantaneous rate of change (the slope of the tangent line), while the Hessian tells you how fast that slope is changing (the curvature). When the Hessian is positive,

the function curves upward like a bowl—this is where gradient descent naturally converges. When it's negative, the function curves downward—a local maximum. At the inflection point ($x=0$), the curvature is zero.

D.4.4 Example: Reducing Computational Cost using JVPs and HVPs

Jacobian-vector products (JVPs) and Hessian-vector products (HVPs) let us compute directional derivatives efficiently without forming the full Jacobian or Hessian matrix. This is crucial for large-scale problems. We will first see how this works on a simple 2D function, and then later move to a more realistic case of a Neural Network.

```
# A simple 2D function f(x, y)
def f(xy):
    x, y = xy[..., 0], xy[..., 1]
    return (x**2 + y**2) + 0.5 * x * y**3

xy = torch.tensor([1.0, 2.0], requires_grad=True)
f_val = f(xy)

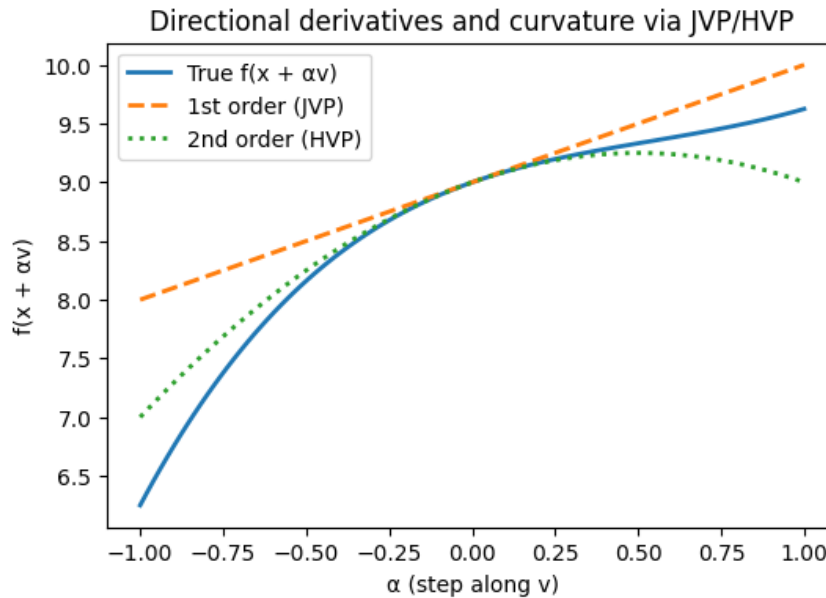
# Gradient (Jacobian of f wrt xy)
# create_graph=True so we can differentiate the gradient later
grad = torch.autograd.grad(f_val, xy, create_graph=True)[0]
print("Gradient:", grad)

# Choose a direction vector v (match dtype/device of xy)
v = torch.tensor([1.0, -0.5], dtype=xy.dtype, device=xy.device)

# Directional derivative (scalar):  $\nabla f(x) \cdot v$ 
jvp_scalar = (grad * v).sum()
print("Directional derivative ( $\nabla f \cdot v$ ):", jvp_scalar.item())

# Hessian-vector product:  $H \cdot v = \nabla_x (\nabla f \cdot v)$ 
hvp = torch.autograd.grad(jvp_scalar, xy)[0]
print("Hessian-Vector Product ( $H \cdot v$ ):", hvp)
```

```
Gradient: tensor([ 6., 10.], grad_fn=<AddBackward0>)
Directional derivative ( $\nabla f \cdot v$ ): 1.0
Hessian-Vector Product ( $H \cdot v$ ): tensor([-1., 2.])
```



Now let's show how this might be useful in a Neural Network context:

```
# Small network: 2 inputs → 4 hidden → 1 output
net = nn.Sequential(
    nn.Linear(2, 4),
    nn.Tanh(),
    nn.Linear(4, 1)
)

# Make output scalar to ensure gradients are well-defined for higher-order derivatives
x = torch.randn(1, 2)
y = net(x).sum() # sum to scalar

# Collect parameters
params = [p for p in net.parameters() if p.requires_grad]

# First-order gradients (create_graph=True so we can compute higher-order derivatives)
grads = torch.autograd.grad(y, params, create_graph=True)

# Random direction in parameter space
v = [torch.randn_like(p) for p in params]

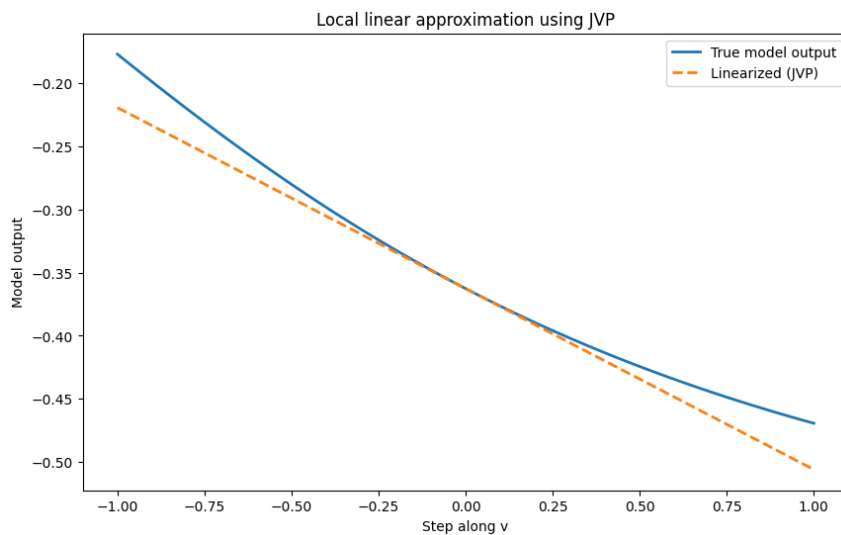
# Compute JVP contributions per-parameter (these are tensors shaped like each parameter)
jvp_per_param = [g * vi for g, vi in zip(grads, v)]
```



```
# Scalar directional derivative (sum over all parameters) =  $(\partial y / \partial \theta) \cdot v$ 
jvp_scalar = sum((jp).sum() for jp in jvp_per_param)

# Hessian-vector product  $H \cdot v$  computed as gradient of the scalar jvp_scalar w.r.t. params
# allow_unused=True may return None for parameters that did not contribute to jvp_scalar
hvp = torch.autograd.grad(jvp_scalar, params, retain_graph=True, allow_unused=True)
```

```
Layer 0 | Param shape (4, 2) | JVP contrib norm=0.246 | HVP norm=2.030
Layer 1 | Param shape (4,) | JVP contrib norm=0.176 | HVP norm=1.066
Layer 2 | Param shape (1, 4) | JVP contrib norm=1.220 | HVP norm=0.983
Layer 3 | Param shape (1,) | JVP contrib norm=0.027 | HVP norm=nan
```



The full Hessian of a neural network with p parameters is a $p \times p$ matrix, and for modern networks with millions of parameters, this is completely infeasible. JVPs and HVPs give us access to gradient and curvature information, respectively, along specific directions at a fraction of the cost.

D.5 Marginalization of Probability Distributions

D.5.1 Concepts:

- **Joint distributions** $p(x, z)$ describe the probability of multiple variables together
- **Marginal distributions** $p(x)$ are obtained by integrating (or summing) out other variables

- Marginalization is fundamental to probabilistic inference and latent variable models
- For continuous variables: $p(x) = \int p(x, z) dz$
- For discrete variables: $p(x) = \sum_z p(x, z)$
- This operation removes nuisance variables and reveals the distribution of the variable we care about

D.5.2 Key Equations:

Marginalization (continuous):

$$p(x) = \int p(x, z) dz$$

Law of Total Probability:

$$p(x) = \int p(x|z)p(z) dz = \mathbb{E}_Z[p(x|Z)]$$

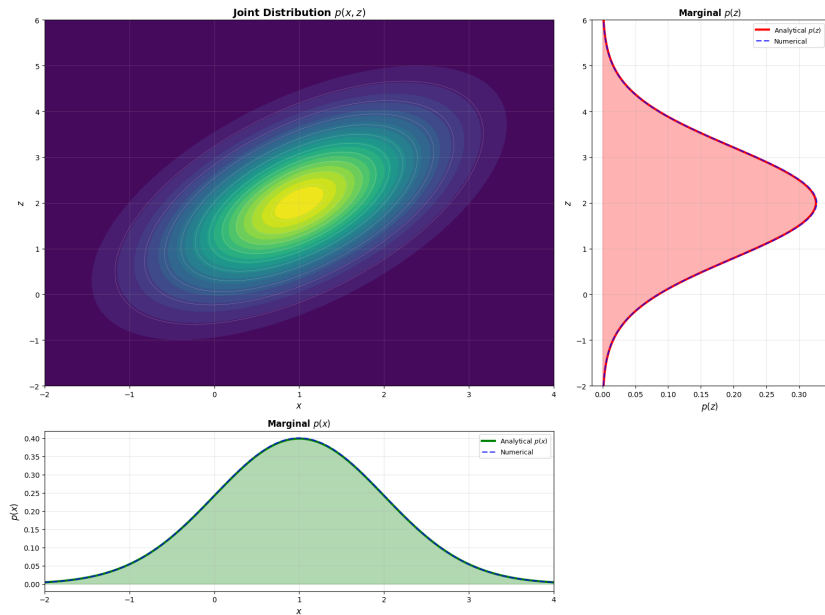
Conditional Distribution:

$$p(x|z) = \frac{p(x, z)}{p(z)}$$

Intuition: Marginalization asks “what’s the probability of x , averaging over all possible values of z ?” It’s like projecting a 2D distribution onto a 1D axis: you’re summing up all the probability mass along one direction.

D.5.3 Example: 2D Gaussian Marginalization

Let’s visualize a 2D Gaussian distribution and see what happens when we marginalize out one variable. This is the foundation for understanding latent variable models.

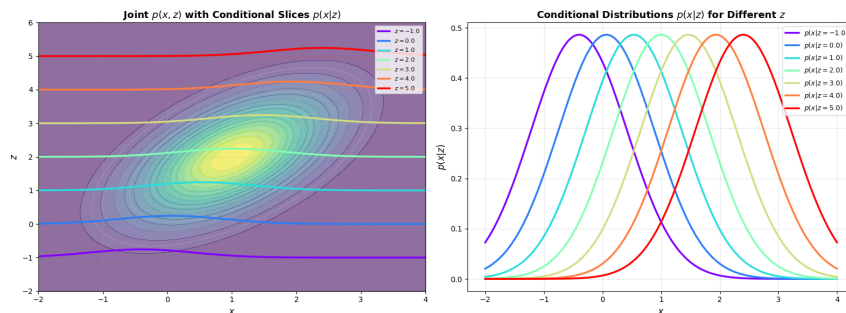


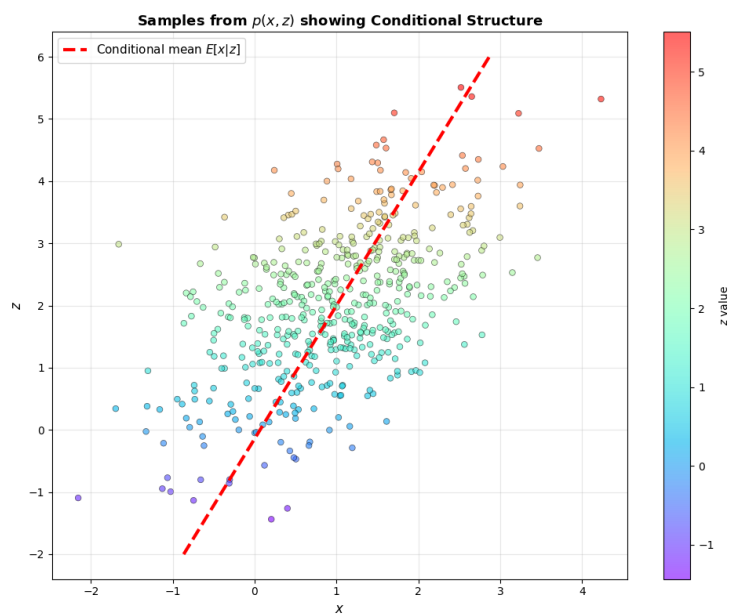
Marginalization “collapses” the joint distribution onto one axis by integrating out the other variable. Think of it as viewing the shadow of a 3D object—you lose information about one dimension but get a simpler view.

The marginal distributions don’t tell you anything about the correlation between x and z . If you only observe $p(x)$ and $p(z)$ separately, could you reconstruct $p(x, z)$?

D.5.4 Example: Conditional Sampling and the Joint Distribution

The joint distribution can be decomposed as $p(x, z) = p(x|z)p(z)$. Let’s visualize how conditionals $p(x|z)$ for different values of z shape the joint distribution.





The conditional distributions $p(x|z)$ show how knowing z affects our beliefs about x . When variables are correlated, observing one variable shifts the expected value of the other. The “ridge” of the joint distribution follows the conditional mean. In VAEs and other latent variable models, we often work with $p(x|z)$ where the decoder generates x given a latent code z .