Introduction to Neural Networks

Written by Rikab Gambhir (Center for Theoretical Physics, MIT)

In this tutorial, we will explore Neural Networks, the fundamental building block of deep learning. We will go into the very basics of the theory of Neural Networks and Universal Function Approximation. Then, we will explore practical immplementations of Neural Networks and deep learning that are widely used both in physics applications and also are widespread in industry.

This tutorial is divided into 4 parts:

1. **Neural Network Basics**: Constructing multi-layer perceptrons and studying universal function approximation.
2. **JAX**: An increasingly popular library used for machine learning. This library is extremeley similar to basic numpy, but has extra features like autodifferentiation and compilation that make it useful for machine learning.
3. **PyTorch**: A commonly used ML library. Developed by Meta. Especially nice for implementing fancy modern ML models, since they're mostly developed in PyTorch anyways!
4. **Tensorflow**: Less common in 2025, but many ML tools still use it.

Prerequisites

I will assume knowledge of the following:

1. Basic python and numpy. You should be comfortable with matrix operations within numpy, dealing with lists and loops, defining functions, and classes.
2. You are familiar with the previous tutorials on regression, classification, normalizing flows, and unsupervised learning. In particular you should appreciate the idea of finding parameters that minimize the log-likelihood (or other metrics) for function fitting, and the general importance of finding/optimizing for functions for statistical tasks.

Prelude: CPUs vs GPUs

The tutorials here can be relatively computationally intensive, especially when we start training neural networks. The tutorials you have seen thus far have been relatively light, and you could run them on your laptop or on default Google Colab settings.

However, for these tutorials, I recommend trying the GPU as well as the ordinary CPU. Google Colab provides free access to GPUs, which can significantly speed up the training of neural networks and other machine learning models. You can do this by going to the "Runtime" tab in Google Colab and selecting "Change runtime type" in the upper-right corner (next to the RAM icon). Then, select "GPU" as the hardware accelerator. It is possible to run these tutorials on a CPU, but it will be very slow, especially for the later tutorials.

Most ML is designed to take advantage of highly-parallelizable tasks and linear algebra, where operations are simple and can be done in parallel. GPUS are designed with many smaller cores that can do many simple operations in parallel, which is why they are so useful for ML. Whereas CPUs have only a few cores that are designed to do more complex operations, but not as many in parallel.

When running these tutorials, you may get an error telling you have run out of memory, especially on GPU. This is because Colab has a limited amount of memory available for each session. If you encounter this error, you can try restarting the runtime (Runtime -> Restart runtime) or clearing the output of the cells (Edit -> Clear all outputs), and then only re-running the cells of interest. If you still run into memory issues, you may need to reduce the size of the models or the batch size during training, or come up with some other clever solutions (like batching!). This is a very realistic issue that you will encounter in real ML applications, so consider running into this issue part of the tutorial!

Chapter 1: Neural Network Basics

```
# Standard Imports
import os
import sys
import numpy as np
import math
import scipy.stats as st
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

In previous tutorials, e.g. `regression.ipynb`, the goal was to model a fixed functional form $f(x)$ where f depended on some parameters θ. For example, a linear fit of the form $f(x) = \theta_0 + \theta_1 x$.

In Deep Learning, we want to be more ambitious. We do not want to assume a specific functional form: rather than only ``searching" over a fixed set of basis functions, we want to search over *all* functions, or at least a very large class of functions. Our strategy for doing this is to take a functional form with an extremeley large set of parameters, such that in the infinite parameter limit all functions of a particular class fit within the parameterization. For example. the set of functions:

$$f(x) = \sum_{i=0}^{N} \theta_i x^i$$

models all one-dimension analytic functions as $N \to \infty$. However, we would like a more general parameterization that can work for many dimensions and even model non-smooth (or even non-continuous) functions arbitrarily well continuous) functions arbitrarily well.

A **Neural Network (NN)** (also known as a **Multilayer Perceptron (MLP)** a **feedforward network**, or a **Dense Neural Network (DNN)** depending on the context) parameterizes *all* peicewise-continuous functions from $\mathbb{R}^n \to \mathbb{R}^m$ arbitrarily well with a very simple parameterization.

To define a neural network, we first specify $L-2$ integers $N_1, \ldots, N_{L-1}$. Just for notation, choose $N_0 = n$ as the input dimension, and $N_L = m$ as the output dimension. L is referred to as the *depth* of the network (or number of layers), and the N's are the *width* of each layer. Unless you are doing something fancy (e.g. autoencoders), it is typical to choose N to all be the same.

Then, we define a set of *layer functions*, which are maps $f^\ell : \mathbb{R}^{N_{\ell-1}} \to \mathbb{R}^{N_\ell}$, as:

$$f^\ell(x) = \sigma(W^{(\ell)}x + b^{(\ell)})$$

where $W^{(\ell)} \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ and $b^{(\ell)} \in \mathbb{R}^{N_\ell}$ are the parameters that define the layer, and σ is some pre-determined nonlinear transformation. This can differ between layers, but it is common to chose σ to be the same for every layer except the last, where σ is often instead chosen such that its image matches the desired output space. An extremeley common and simple chose for σ is the ReLU (Rectified Linear Unit) function, which we will use throuhout the rest of this tutorial:

$$\sigma(x) = \max(0, x)$$

Then, the full neural network is defined by:

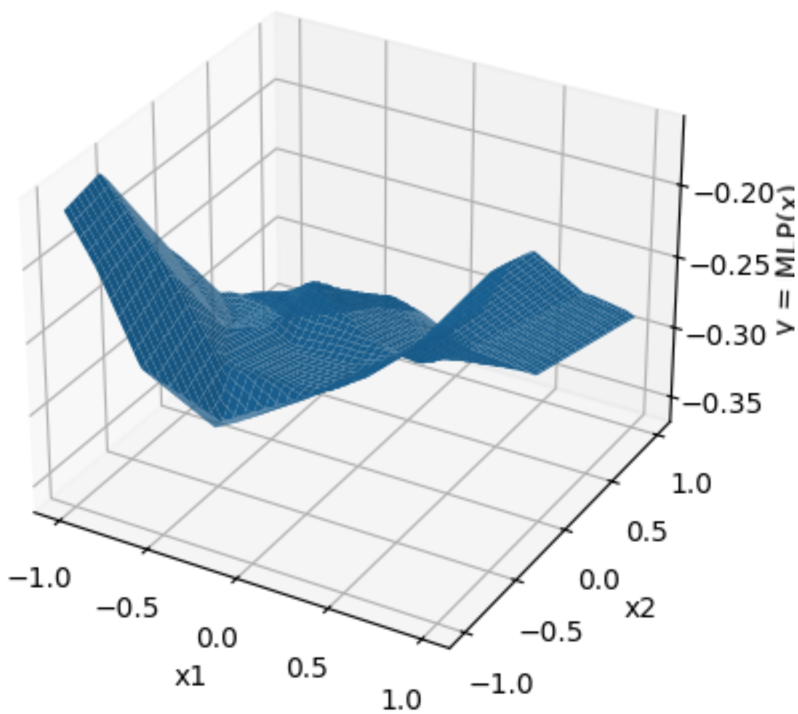
$$f = f^L \cdot f^{L-1} \cdot \ldots \cdot f^1$$

Let's make an MLP from scratch!

```
# Building a Neural Network from Scratch #

input_dim = 2
output_dim = 1


L = 3
N = 16  # We will use the same N throughout for simplicity


# Function to initialize the W's and b's
# For now, lets just pick random numbers!
def init_params(input_dim, output_dim, L, N):
    Ws = []
    bs = []

    for l in range(L):
        if l == 0:
            W = np.random.randn(N, input_dim) / np.sqrt(input_dim)
            b = np.random.randn(N) / np.sqrt(input_dim)
            # The sqrt(input_dim) normalization is not important for our toy examples,
```

```
        elif l == L - 1:
            W = np.random.randn(output_dim, N) / np.sqrt(N)
            b = np.random.randn(output_dim) / np.sqrt(N)

        else:
            W = np.random.randn(N, N) / np.sqrt(N)
            b = np.random.randn(N) / np.sqrt(N)

        Ws.append(W)
        bs.append(b)

    return Ws, bs


# Implement the ReLU function
def sigma(x):
    return np.maximum(0, x)


# Function to evaluate a neural network given x, the weights W, and the biases b


def MLP(x, Ws, bs):
    y = x.copy()

    for l in range(L):
        # Fun python fact: "@" implements matrix multiplication!
        y = Ws[l] @ y + bs[l]

        # Don't apply sigma to the final output so that our answer isn't forced positiv
        if l != L - 1:
            y = sigma(y)

    return y


# Test our MLP function by graphing the function f:R2 -> R1

# Define some test points in R2
xs1 = np.linspace(-1, 1, 100)
xs2 = np.linspace(-1, 1, 100)

xs1, xs2 = np.meshgrid(xs1, xs2)

# Initialize the weights and biases
Ws, bs = init_params(input_dim, output_dim, L, N)

ys = []
for x in zip(xs1.flatten(), xs2.flatten()):
    x = np.array(x)
    ys.append(MLP(x, Ws, bs))

ys = np.array(ys)
```

```
ys = ys.reshape(xs1.shape)

# 3d plot
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(xs1, xs2, ys)
ax.set_xlabel("x1")
ax.set_ylabel("x2")
ax.set_zlabel("y = MLP(x)")
plt.show()
```



A note on functional vs. object-oriented programming

In the above code, we defined our MLP purely using python functions. There is no neural network "object" with an internal state keeping track of the parameters. Instead, the parameters W and b are also treated as inputs to functions. This is *functional programming*, in which there are no objects with internal states that get modified. This is the approach to ML used by JAX.

It is also possible to define an MLP *class*, which is an object that contains the parameters as internal states that can potentially be modified, and methods that implement the model and evaluate $f(x)$. This is the approach to ML used by PyTorch and Tensorflow.

It is largely a matter of programming taste which you prefer. Below, we will see a brief example of the above code, but written in an object-oriented style rather than functional.

```
class My_MLP_Class:
    def __init__(self, input_dim, output_dim, L, N):
        # Initialize the network arguments
        self.input_dim = input_dim
```

```
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.L = L
        self.N = N

        # Initialize the network internal state using the same initi function
        self.Ws, self.bs = init_params(input_dim, output_dim, L, N)

    def evaluate(self, x):
        # Just use the same exact function as above
        return MLP(x, self.Ws, self.bs)

    # "Magic Method" that lets us call the class as if it were a function (just syntati
    def __call__(self, x):
        return self.evaluate(x)


my_MLP = My_MLP_Class(input_dim, output_dim, L, N)

# Access the weights
my_weights = my_MLP.Ws
print("The number of layers is ", len(my_weights), ",Expected 3")

# Evaluate the function
print("f(1,1) = ", my_MLP(np.array([1, 1])))
```

```
The number of layers is  3 ,Expected 3
f(1,1) =  [2.25652799]
```

Historical Notes and Semantics

The case where $L=2$ (no ``hidden layers" between the input and output) with the output dimensionality is 1 is called a perceptron historically. These were introduced with σ not as ReLU, but rather:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

(the sigmoid function, hence the notation), and were used back in the day as a model of a biological neuron. The neuron "activates" (produces 1) when x is large, and "deactivates" (produces 0 when x is small, where b is then a bias. For this reason, σ is called an activation function. This is also why our models are called "Neural Networks". The "network" is because the parameters of the weight matrix w_{ij} are drawn as lines connecting a node i in the previous layer to a node j in the next layer. It's important to remember though, that these are just affine transformations interleaved by some simple nonlinear functions, and there isn't really anything magic here, just slightly-nonlinear algebra.

The name "feedforward" network just refers to the function-compositional aspect of the model. It is to be contrasted with a "backwards pass", where derivatives with respect to the network are actually computed in reverse-order due to chain-rule simplifications. The name "dense" neural network is to emphasize that this is the simplest possible network one can build. There are many modern models with additional properties (such as gauranteeing symmetries, or working in spaces other than simple with additional properties (such as guaranteeing symmetries, or working in spaces other than simple vector spaces, or deliberately constraining the function space), but many of these can be reduced to very large MLPs with constrained weights. When we say "dense" or "fully-conenected" MLPs, we typically mean there are no constraints on the parameters.

Chapter 1.1: Universal Function Approximation

The power of MLPs is that they are an efficient way to parameterize a large class of functions. This is captured by the **Universal Function Approximation Theorem(s) (UFAT)** (there are lots of variants, but at the level of rigor we are working at, we won't worry about this).

Emotionally, the UFAT tells us that for sufficiently large N and L, an MLP can approximate any (reasonable) n-to-m dimensional function arbitrarily well.

Slightly more precisely, a version of UFAT says: For any piecewise-continuous function $f : \mathbb{R}^n \to \mathbb{R}^m$ defined on a compact domain $D \subset \mathbb{R}^n$, and for any "error tolerance" $\epsilon > 0$, there exists a large enough N and L such that one can define an MLP with specially-chosen parameters W and b such that:

$$\int_D dx |f(x) - MLP(x)| < \epsilon$$

i.e. that we have approximated the function to within the specified error.

[Side note: It is actually always possible to do this with just L = 3 (meaning just one hidden layer with chosen N in our defined L counting), but typically this requires an exponentially large N and isn't of practical use for what we will be doing].

We will not prove the UFAT. However, we will explore a weaker-version of it that is easier to understand: If instead we explore continuous-and-piecewise-once-differentiable functions rather than just piecewise-continuous, then there is an easy construction using ReLU networks. If a function is piecewise-once-differentiable, then it can be well-approximated by a piecewise-linear function. We will see below (as exercises) how ReLU networks can exactly reproduce piecewise linear functions.

Exercise: Modeling |x|

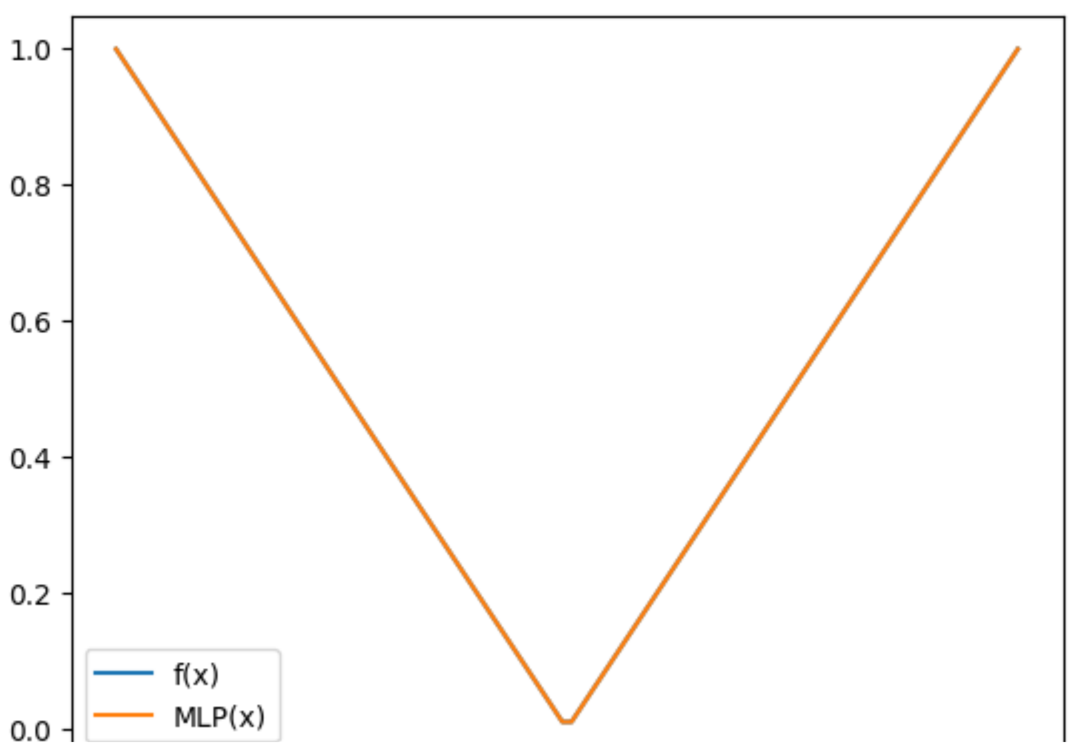
Given $f(x) = |x|$ in 1 dimension, design an MLP with a choice of N, L, weights W, and biases b that *exactly* match $f(x)$.

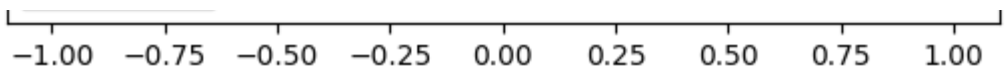
HINT: It is possible to do this with $L = 2$ (one hidden layer) and $N = 2$.

HINT 2: It is possible to do this with $b = 0$.

```
def f(x):
    return np.abs(x)
```

```python
L = 2
N = 2

W0 = np.array([[1.0], [-1.0]])  # hidden unit 1:  +x  # hidden unit 2:  −x
b0 = np.array([0.0, 0.0])  # no shift

W1 = np.array([[1.0, 1.0]])  # add the two ReLU outputs
b1 = np.array(
    [
        0.0,
    ]
)  # no shift

Ws = [W0, W1]
bs = [b0, b1]

xs = np.linspace(-1, 1, 100)

# Evaluate the solution
ys = []
for x in xs:
    x = np.array([x])
    ys.append(MLP(x, Ws, bs))

ys = np.array(ys)

# Plot
plt.plot(xs, f(xs), label="f(x)")
plt.plot(xs, ys, label="MLP(x)")
plt.legend()
plt.show()
```





Exercise: Approximating a smooth 1D function.

Given $f(x) = \sin(10x)\exp(-2x^2)$ on the interval $[-1, 1]$, design an MLP with ReLU-activations that approximates the function to within an error of $\epsilon < 0.01$ (where error is the mean-absolute error, as defined above). As a bonus, your implementation should be systematically improvable, e.g. it should be straightforward to make the MLP bigger to reduce the error further. Don't cheat and use minimization to get the parameters, explicitly construct them!

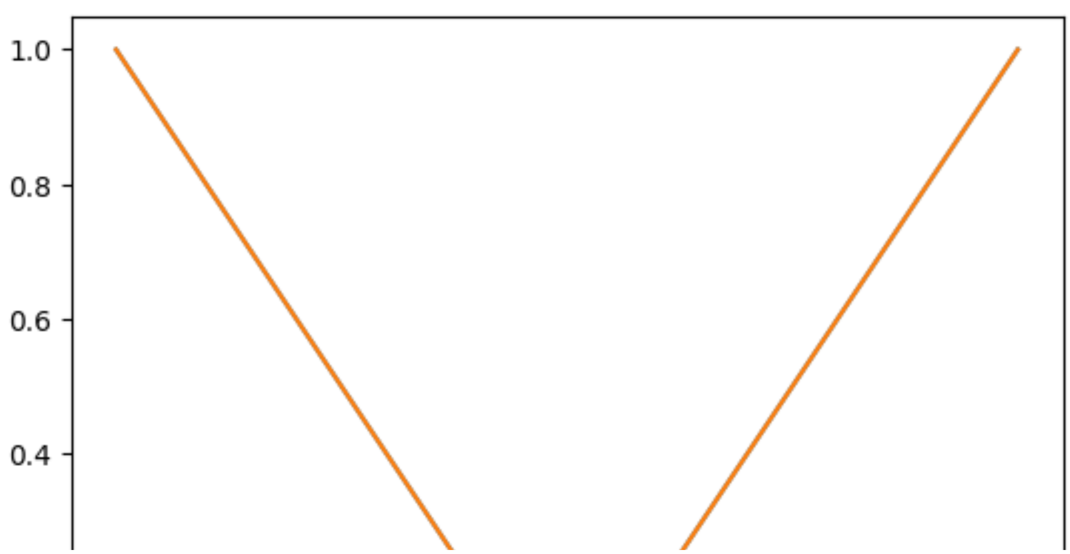
HINT: First construct a continuous piecewise linear appoximation to the function, then implement this piecewise linear function as an MLP. It is possible to do this without knowledge of the actual form of f.

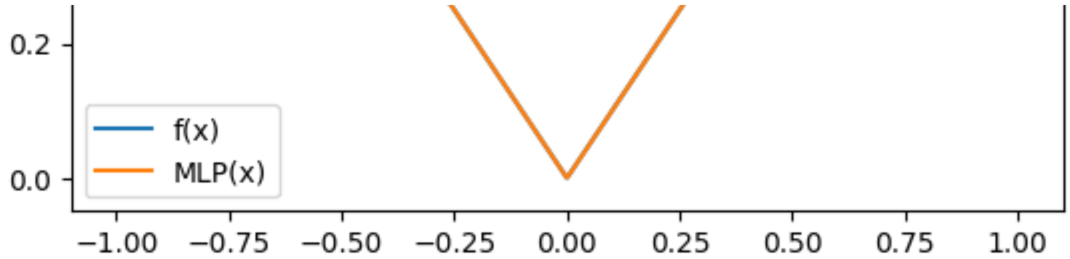
HINT 2: This is possible to do systematically with $L = 2$ as before, but with a very large N. My personal solution requires N between 100 and 150.

HINT 3: A piecewise-linear continuous function can be written as $f(x) = c_0 + m_0 x + + \sum_{j=1}^{n-1}(m_j - m_{j-1})\sigma(x - x_j)$, where σ is ReLU, $x_1 \ldots x_{n-1}$ are the internal breakpoints, m_j are the slopes to the right of each breakpoint, and c_0 is the y-coordinate at the leftmost point.

```
# For any function f, define a piecewise-linear approximation with n breakpoints.
def piecewise_linear_params(n, f):
    # Breakpoints of the piecewise linear approximation
    xk = np.linspace(-1.0, 1.0, n + 1)  # x0, …, xn
    yk = f(xk)

    # Approximate the slopes numerically (technically, possible exactly)
    mk = np.diff(yk) / np.diff(xk)  # m0 … m_{n-1}

    # slope jumps delta_m_j at each breakpoint
    dm = mk[1:] - mk[:-1]

    # constant term that glues the first segment to y(-1)
    c0 = yk[0] - mk[0] * xk[0]

    # Initialize weights
    N = n + 1  # annoying indexing
    W0 = np.ones((N, 1))
    b0 = np.zeros(N)

    # Add and subtract a ReLU, like the |x| example
    W0[0, 0] = 1.0
    b0[0] = 0.0
    W0[1, 0] = -1.0
    b0[1] = 0.0
```

```
    # interior break-points
    for j, t_j in enumerate(xk[1:-1], start=2):  # j = 2 … n
        W0[j, 0] = 1.0
        b0[j] = -t_j  # shift to next break point

    # Second layer where everything is just 1 and -1 to add and subtract
    W1 = np.zeros((1, N))
    W1[0, 0] = mk[0]  #  +m0·ReLU(x)
    W1[0, 1] = -mk[0]  #  −m0·ReLU(−x)

    for j, d_m in enumerate(dm, start=2):
        W1[0, j] = d_m

    b1 = np.array([c0])  # constant offset for leftmost point

    Ws = [W0, W1]
    bs = [b0, b1]
    return Ws, bs


num_breakpoints = 100  # Increase for more accuracy!
Ws, bs = piecewise_linear_params(num_breakpoints, f)
xs = np.linspace(-1, 1, 1000)

ys = []
for x in xs:
    x = np.array([x])
    ys.append(MLP(x, Ws, bs))
ys = np.array(ys)

plt.plot(xs, f(xs), label="f(x)")
plt.plot(xs, ys, label="MLP(x)")
plt.legend()
plt.show()


print("Mean absolute error: ", np.mean(np.abs(ys[:, 0] - f(xs))))
```






```
Mean absolute error:  0.0
```

Chapter 1.2: Functional Optimization

We now have the ability to approximate function spaces with MLP's! The fun part of Machine Learning (the Learning) comes in when we can phrase problems as *functional optimization* problems:

"Out of all the (reasonably nice) functions from $\mathbb{R}^n \to \mathbb{R}^m$, which function f minimizes the loss functional $L[f]$?"

Almost every interesting problem in life, statistics, and physics can be phrased this way. In fact, this is completely identical to Lagrangian mechanics, in the case that $L[f]$ can be written as the integral of a local Lagrangian. In simple cases (ordinary classical mechanics) this functional optimization can be performed using the Euler-Lagrange equations. But in many cases (e.g. where $L[f]$ is written as a sum rather than an integral so EL does not apply, or we can't solve the EL equations, etc), we must settle for numerics.

You have seen in previous tutorials how many statistics problems (e.g. regression, classification, and density estimation) can be seen as functional optimization. In those examples, there were only a few parameters defining the function space: now there are *many* parameters and our function space is as close to the space of all possible functions as possible. We can no longer just use a simple parameter minimizer in this case.

The strategy will be **gradient descent**. If we have an estimate of:

$$\nabla_\theta L[f]$$

then by simply moving θ in the opposite direction of the gradient, we will move towards a local minimum. The process of iterating this is called **training**, and each iteration is called an **epoch**. In statistical settings, where $L[f]$ is some statistical measure (like in the regression examples), this training requires data to obtain statistical estimates of $\nabla_\theta L[f]$, hence the need for **training data**. There are many variants of gradient descent that work on the same principle but have varying numerical properties, like stochastic gradient descent and ADAM, but we will not dive deeper into these here.

In principle, if we know L (which we usually do, because it is typically part of the problem specification), we can explictly construct $\nabla_\theta L[f]$ exactly, since we know how f depends on our parameters $\theta = (W, b)$. However, it is still painful to manually construct. This is where libraries like **JAX**, **PyTorch**, and **Tensorflow** come in. These libraries are capable of **autodifferentiation**:

computations are kept track of in a graph structure, so that gradients can be easily and exactly computed alongside the execution of the function. Exploring this further will be the subject of another tutorial, for now we wil take it for granted that these libraries can perform autodifferentiation.

We have reached the limit of what we can practically do without the use of libraries in a reasonable amount of time. Now, we will explore how to use these libraries.

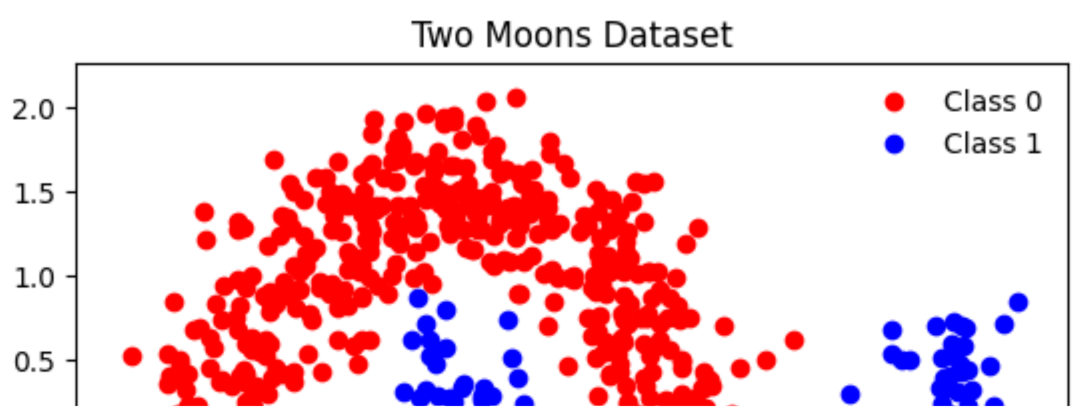
Interlude: The problems we will solve:

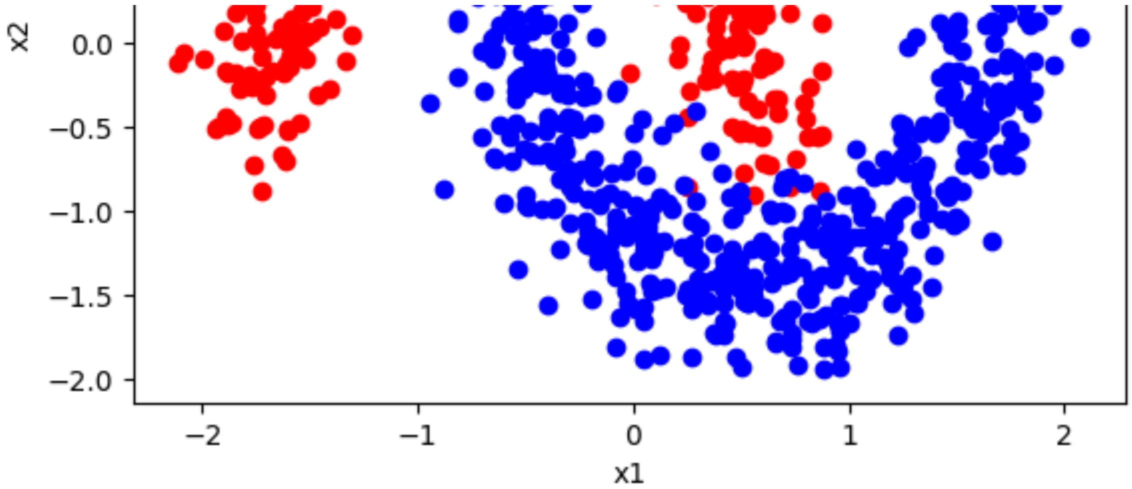
We will be interested in using Neural Networks to solve classification problems. We have previously seen how to do this with logistic regression and cross-entropies in the classification tutorial. We will now see how to do this with Neural Networks.

We will have two problems: an easy problem, and a hard problem. The easy problem is "Two Moons", a classic ML test case in 2 dimensions. The hard problem is the "MNIST" dataset, a dataset of handwritten digits that is commonly used to test classification algorithms. This is a 28x28 pixel image dataset, so the input dimension is 784, so MLPs can really shine compared to classical fixed-form regressors.

```
from sklearn.datasets import make_moons, fetch_openml
from sklearn.preprocessing import StandardScaler


# Two Moons Dataset

X_moons, y_moons = make_moons(1024, noise=0.15, random_state=0)
X_moons = StandardScaler().fit_transform(X_moons)  # Just normalizing the data

plt.plot(X_moons[y_moons == 0, 0], X_moons[y_moons == 0, 1], "ro", label="Class 0")
plt.plot(X_moons[y_moons == 1, 0], X_moons[y_moons == 1, 1], "bo", label="Class 1")
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Two Moons Dataset")
plt.legend(frameon=False)
```

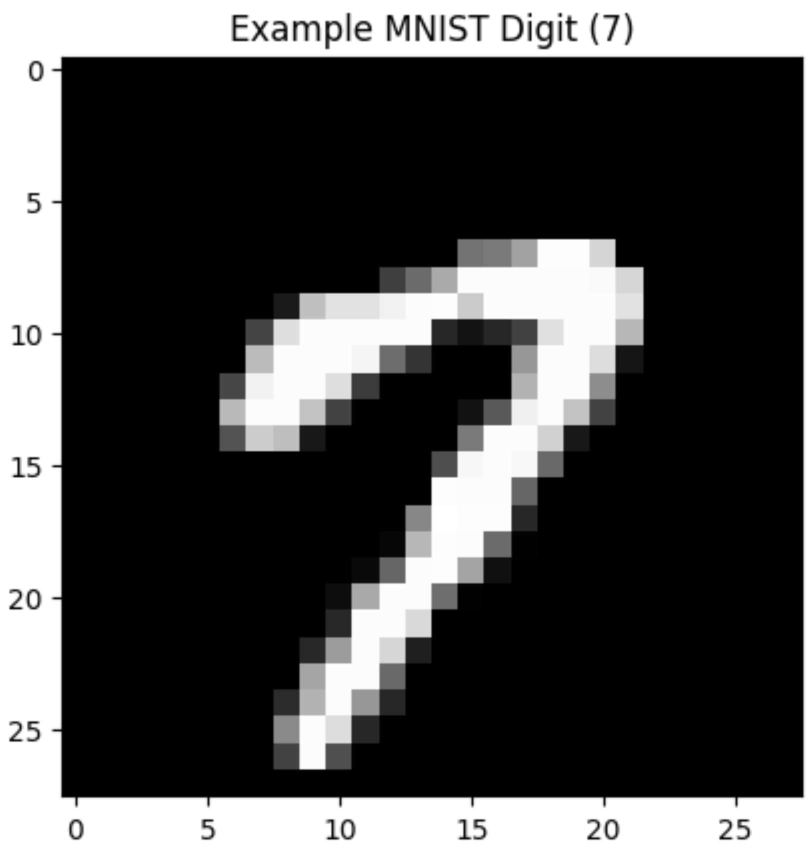
```
<matplotlib.legend.Legend at 0x7b2c1d30ce90>
```






```
mnist = fetch_openml("mnist_784", version=1, as_frame=False)
X_mnist = mnist.data / 255.0  # Normalize pixel values to [0, 1]
y_mnist = mnist.target.astype(int)  # Convert target to integers

# The MNIST input dimension is 784, but we can visualize it as 28x28 images
plt.imshow(X_mnist[y_mnist == 7][0].reshape(28, 28), cmap="gray")
plt.title("Example MNIST Digit (7)")
```

```
Text(0.5, 1.0, 'Example MNIST Digit (7)')
```



Chapter 2: JAX

JAX is a library developed by Google that is very similar to numpy, but with some extra features that

JAX is a library developed by Google that is very similar to numpy, but with some extra features that make it useful for machine learning. It is based on the idea of functional programming, where functions are first-class citizens and can be passed around like any other object. JAX is particularly useful for machine learning because it has built-in support for autodifferentiation, which allows us to compute gradients of functions with respect to their inputs.

Compared to PyTorch and Tensorflow, JAX is more low-level and requires more manual work to set up. However, it is also more flexible and allows for more control over computations. JAX is particularly useful for research and experimentation, where you want to try out new ideas quickly without having to worry about the details of the implementation.

```
import jax
import jax.numpy as jnp
from jax import grad, jit, vmap
from jax import random

import time
```

Chapter 2.1: JAX basics; vmapping, autodifferentiation, and compilation.

JAX has three useful features that we should be aquainted with:

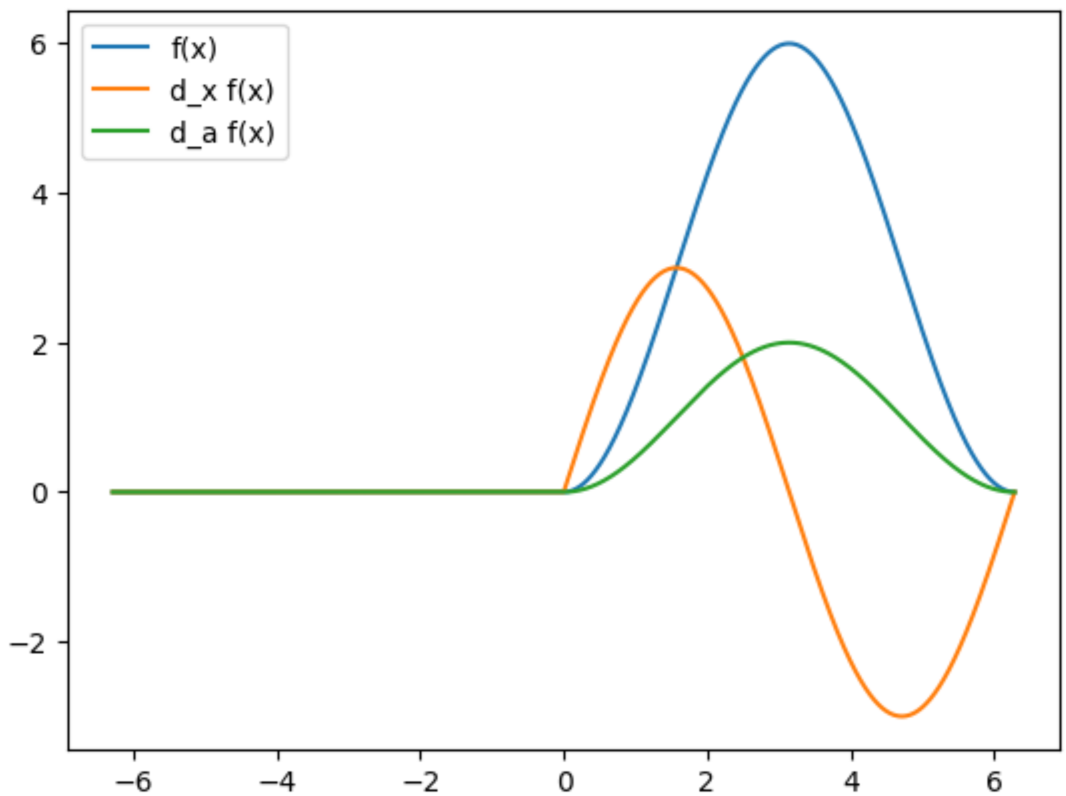
1. Vmapping: We can write a function acting on a single variable, and then execute that function on an entire list at once without using loops. In fact, this is much faster than looping (in Python), since Python loops must wait for the previous iteration to finish. Note that numpy can technically do this too, but it becomes especially important in JAX
2. Just-In-Time compilation (JIT): Python is a scripted language, meaning lines of code are carried out as your computer sees them. In compiled languages, the computer looks at the entire program, translates to machine code (compilation), then executes. You pay an up-front time cost for the initial compilation, but every subsequent execution is much faster since the machine code is typically highly optimized. JIT allows us to pre-compile functions in Python. The cost is that we have to be a little be conscious of things like memory, and we cannot use things like ordinary if-statements or for-loops.
3. Autodifferentiation: If we write a function in JAX, we can automatically compute its exact derivative. We don't have to manually compute it ourselves! This even works with multi-variate functions, functions that are highly-composed and require lots of chain-ruling, etc.

```
def relu(x):
    return jnp.maximum(0, x)
```

```
def theta(x):
    return x > 0


def f(x, a):
    return a * (1 - jnp.cos(x)) * theta(x)


a = 3.0
xs = jnp.linspace(-2 * jnp.pi, 2 * jnp.pi, 10000)

# Vmapping time save test:
start_time = time.time()
ys = []
for x in xs:
    ys.append(f(x, a))
ys = jnp.array(ys)
end_time = time.time()
print("Loop time: ", end_time - start_time)

start_time = time.time()

# vmap(f) is a new function with the same signature as f.
# vmap(f, in_axes = (0, None)) means we only want to vectorize over the first argument (

ys = vmap(f, in_axes=(0, None))(xs, a)
end_time = time.time()
print("Vmap time: ", end_time - start_time)


# Compile the function
start_time = time.time()
f_jit = jit(f)
f_jit(
    0, a
).block_until_ready()  # Need to run the compiled function once to compile "just in time
end_time = time.time()
print("Compilation time: ", end_time - start_time)
# Note that compilation is NOT always faster, especially for only simple functions.
# Also machine-dependent!


start_time = time.time()
ys = []
for x in xs:
    ys.append(f_jit(x, a))
ys = jnp.array(ys)
end_time = time.time()
print("JIT loop time: ", end_time - start_time)

start_time = time.time()
ys = vmap(f_jit, in_axes=(0, None))(xs, a)
end_time = time.time()
```

```
print("JIT vmap time: ", end_time - start_time)


# Get the exact gradient with respect to x
f_prime = jax.grad(f, argnums=0)  # Argnums is the argument we want the gradient of.
f_prime_jit = jit(f_prime)
f_prime_jit(
    0.0, a
).block_until_ready()  # Need to run the compiled function once to compile "just in time

# Get the exact gradient with respect to a
f_prime_a = jax.grad(f, argnums=1)  # Argnums is the argument we want the gradient of.
f_prime_a_jit = jit(f_prime_a)


plt.plot(xs, vmap(f_jit, in_axes=(0, None))(xs, a), label="f(x)")
plt.plot(xs, vmap(f_prime_jit, in_axes=(0, None))(xs, a), label="d_x f(x)")
plt.plot(xs, vmap(f_prime_a_jit, in_axes=(0, None))(xs, a), label="d_a f(x)")
plt.legend()
plt.show()
```

```
Loop time:  2.355942487716675
Vmap time:  0.1668710708618164
Compilation time:  0.024274826049804688
JIT loop time:  1.0622775554656982
JIT vmap time:  0.02984023094177246
```



Exercise: Autodifferentiation Practice

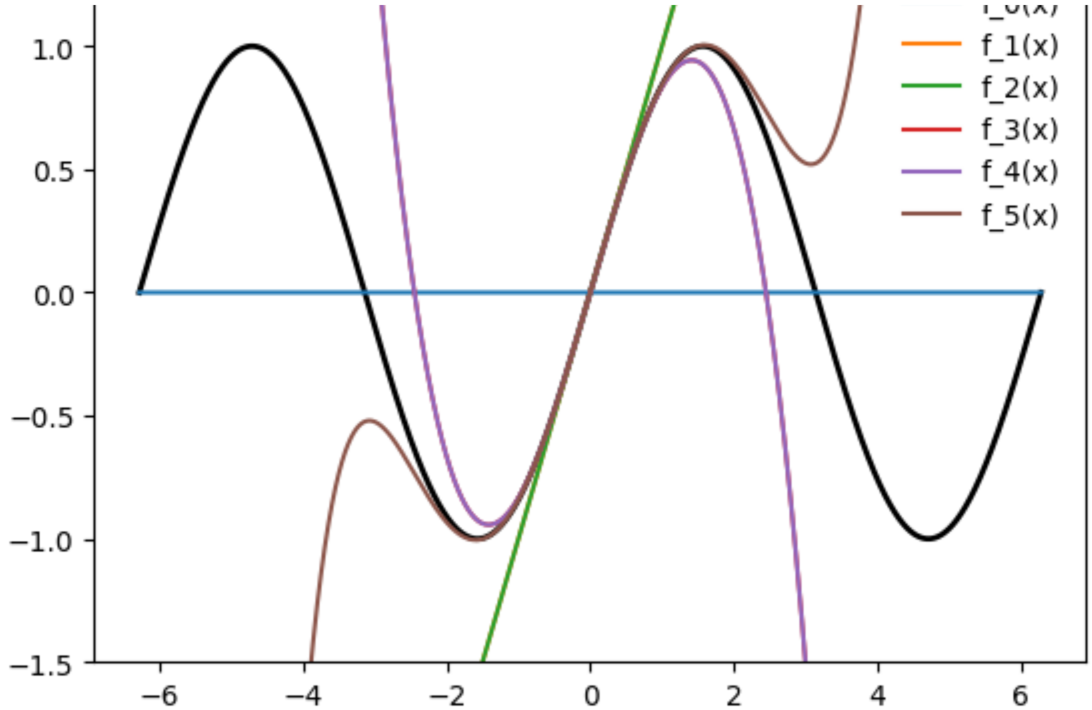
Given an *arbitrary* scalar-valued function $f(x)$, which you know is smooth, write a function that computes the Taylor expansion of f around a point x_0 to order n.

Specifically, your function should take as input f, x_0, and n, and return a new function, f_n, which is the Taylor expansion of f around x_0 to order n. The function f_n should take as input a single variable x, and return the value of the Taylor expansion at that point.

HINT: Construct a list of functions $f_0, f_1, \ldots, f_n$ where f_i is the i-th derivative of f, or equivalently, f_i is the single derivative of f_{i-1}. Then evaluate this list of functions at x_0 to obtain the coefficients of the Taylor expansion.

```
def f(x):
    # EXAMPLE FUNCTION
    return jnp.sin(x)


def build_taylor_series(f, x0, n):
    # Compute the derivatives at x0
    derivative_functions = [f]
    derivatives_at_x0 = [f(x0)]

    for i in range(1, n + 1):
        derivative_functions.append(grad(derivative_functions[-1], argnums=0))
        derivatives_at_x0.append(derivative_functions[-1](x0))

    # Define the Taylor series expansion
    def taylor_series(x):
        series = 0.0
        for i in range(n + 1):
            series += derivatives_at_x0[i] * (x - x0) ** i / math.factorial(i)
        return series

    return taylor_series


# Test the solution
x0 = 0.0
n = 5
xs = jnp.linspace(-2 * jnp.pi, 2 * jnp.pi, 10000)
plt.plot(xs, f(xs), label=r"f(x)", color="black", lw=2)

for i in range(n + 1):
    taylor_series = build_taylor_series(f, x0, i)
    plt.plot(xs, vmap(taylor_series, in_axes=0)(xs), label="f_{}(x)".format(i))
plt.ylim(-1.5, 1.5)

plt.legend(frameon=False)
```

```
<matplotlib.legend.Legend at 0x7b2bd708d050>
```



Chapter 2.2: End-to-End MLP and Training from Scratch

First, let's reproduce the MLP we defined above, but now using JAX. We will use the functional programming style, so we will define our MLP as a function that takes in the parameters W and b as inputs.

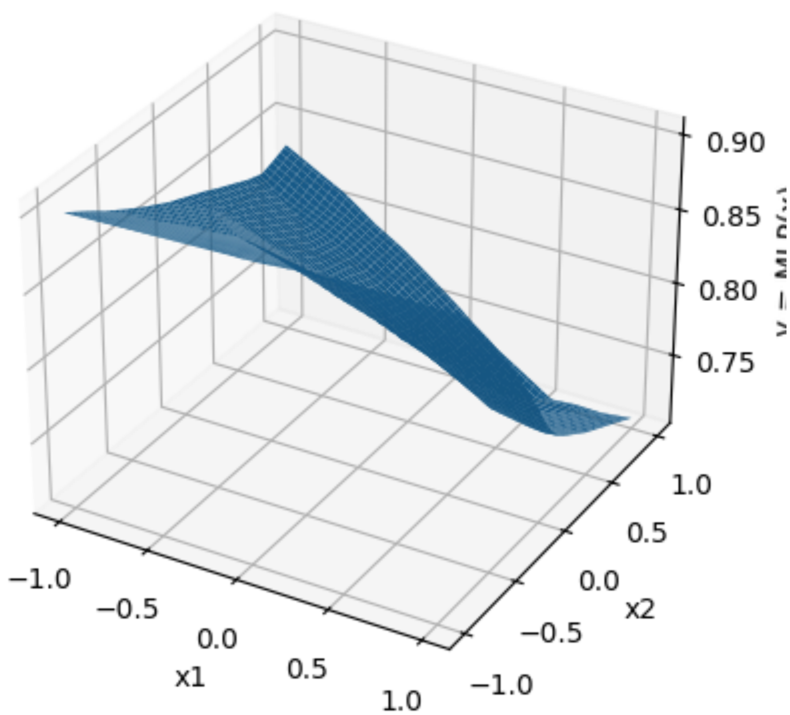
A small difference is that rather than having W and b as separate inputs, we will combine them into a single input called `params`. This is just for convenience to make taking gradients cleaner. Also, since we are interested in classification, we will use a sigmoid activation for the last layer in the binary classification case to ensure the output is between 0 and 1, which is useful for classification tasks, and use a softmax for the multiclass case.

```
# JAX MLP. Basically identical to the above numpy code.
def MLP_jax(x, params):
    y = x
    Ws, bs = params

    for l in range(len(Ws) - 1):
        y = jnp.dot(Ws[l], y) + bs[l]
        y = relu(y)

    y = jnp.dot(Ws[-1], y) + bs[-1]  # No activation on the last layer

    return y


# Our classifier will be an MLP with a sigmoid at the end. Change to softmax for multi-
def classifier(x, params):
```

```python
    # Its common to call the input to the sigmoid the "logits". But really its the log-
    logits = MLP_jax(x, params)
    return 1 / (1 + jnp.exp(-logits))


# Gradient with respect to the parameters
classifier_grad = grad(classifier, argnums=1)


# Initialize the parameters for the MLP
def init_params_jax(input_dim, output_dim, L, N):
    Ws = []
    bs = []

    for l in range(L):
        if l == 0:
            # Basically the same as the numpy version, but using JAX's random module.
            # Unlike numpy, JAX's random module is functional and requires a PRNG key.
            W = random.normal(random.PRNGKey(l), (N, input_dim)) / jnp.sqrt(input_dim)
            b = random.normal(random.PRNGKey(l + 1), (N,)) / jnp.sqrt(input_dim)
        elif l == L - 1:
            W = random.normal(random.PRNGKey(l), (output_dim, N)) / jnp.sqrt(N)
            b = random.normal(random.PRNGKey(l + 1), (output_dim,))
        else:
            W = random.normal(random.PRNGKey(l), (N, N)) / jnp.sqrt(N)
            b = random.normal(random.PRNGKey(l + 1), (N,))

        Ws.append(W)
        bs.append(b)

    return Ws, bs


# Test just to make sure it works

N = 16
L = 3
input_dim = 2
output_dim = 1

xs1 = jnp.linspace(-1, 1, 100)
xs2 = jnp.linspace(-1, 1, 100)

xs1, xs2 = jnp.meshgrid(xs1, xs2)
xs = jnp.array(list(zip(xs1.flatten(), xs2.flatten())))
print("Shape of xs: ", xs.shape)  # Should be (10000, 2)

# Initialize the weights and biases
Ws, bs = init_params_jax(input_dim, output_dim, L, N)

# VMAP our model. We can also JIT it, but its actually better to JIT the entire trainin
vmapped_classifier = vmap(classifier, in_axes=(0, None))
vmapped_classifier_grads = vmap(classifier_grad, in_axes=(0, None))
```

```
ys = vmapped_classifier(xs, (Ws, bs))
ys = ys.reshape(xs1.shape)

# 3d plot
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
ax.plot_surface(xs1, xs2, ys)
ax.set_xlabel("x1")
ax.set_ylabel("x2")
ax.set_zlabel("y = MLP(x)")
plt.show()
```

```
Shape of xs:  (10000, 2)
```



Now we set up the training. We want to minimize the cross-entropy loss function, which is defined as:

$$L[f] = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(f(x_i)) + (1-y_i)\log(1-f(x_i))$$

where y_i is the true label for the i-th data point, and $f(x_i)$ is the model score for the i-th data point.

We will use JAX's autodifferentiation to compute the gradient of the loss function with respect to the parameters, and then use gradient descent to update the parameters.

```
# Define the BCE loss
def bce_loss(x, y_true, params):
    y_pred = vmapped_classifier(x, params).squeeze()
    epsilon = 1e-10  # Small value to avoid log(0), just in case
    return -jnp.mean(
        y_true * jnp.log(y_pred + epsilon)
```

```
        + (1 - y_true) * jnp.log(1 - y_pred + epsilon)
    )


grad_bce_loss = grad(bce_loss, argnums=2)  # Gradient with respect to the parameters


# Gradient step function to update the parameters
@jit  # <- Inline way to JIT the function
def gradient_step(x, y_true, params, learning_rate=0.01):
    # Compute the loss gradients (the loss itself is not needed, just the gradients)
    loss = bce_loss(x, y_true, params)  # But we will compute it anyways for logging
    grads = grad_bce_loss(x, y_true, params)

    # Update the parameters using gradient descent
    Ws, bs = params
    dWs, dbs = grads
    new_Ws = [W - learning_rate * dW for W, dW in zip(Ws, dWs)]
    new_bs = [b - learning_rate * db for b, db in zip(bs, dbs)]
    new_params = (new_Ws, new_bs)

    return new_params, loss


# ##### Train the model on the Two Moons dataset #####

# Set the random seed for reproducibility
train_fraction = 0.8
test_fraction = 1 - train_fraction
epochs = 2000
learning_rate = 0.1
N = 8  # Its nice to choose N ~ 2^n, since differences in width are only really importa
L = 3

# Split the dataset into training and testing sets
train_size = int(train_fraction * len(X_moons))

# Ensure the dataset is shuffled before splitting
# [Just for fun, try removing this part and see how badly things get ruined]
indices = np.random.permutation(len(X_moons))
X_moons = X_moons[indices]
y_moons = y_moons[indices]
X_train, y_train = X_moons[:train_size], y_moons[:train_size]
X_test, y_test = X_moons[train_size:], y_moons[train_size:]

# Initialize the parameters
params = init_params_jax(input_dim, output_dim, L, N)

# Arrays for saving training info
train_losses = []
test_losses = []

# Train the model
```

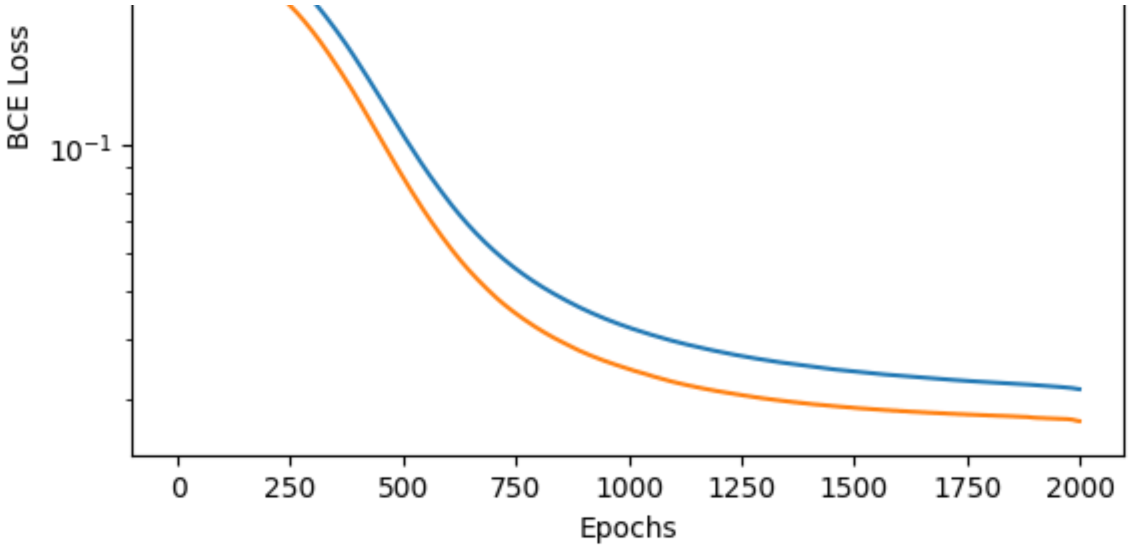
```python
for epoch in range(epochs):
    # Perform a gradient step
    params, loss = gradient_step(X_train, y_train, params, learning_rate)
    train_losses.append(loss)
    # Evaluate on the test set
    test_loss = bce_loss(X_test, y_test, params)
    test_losses.append(test_loss)

    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}, Test Loss: {test_loss:.4f}")
```

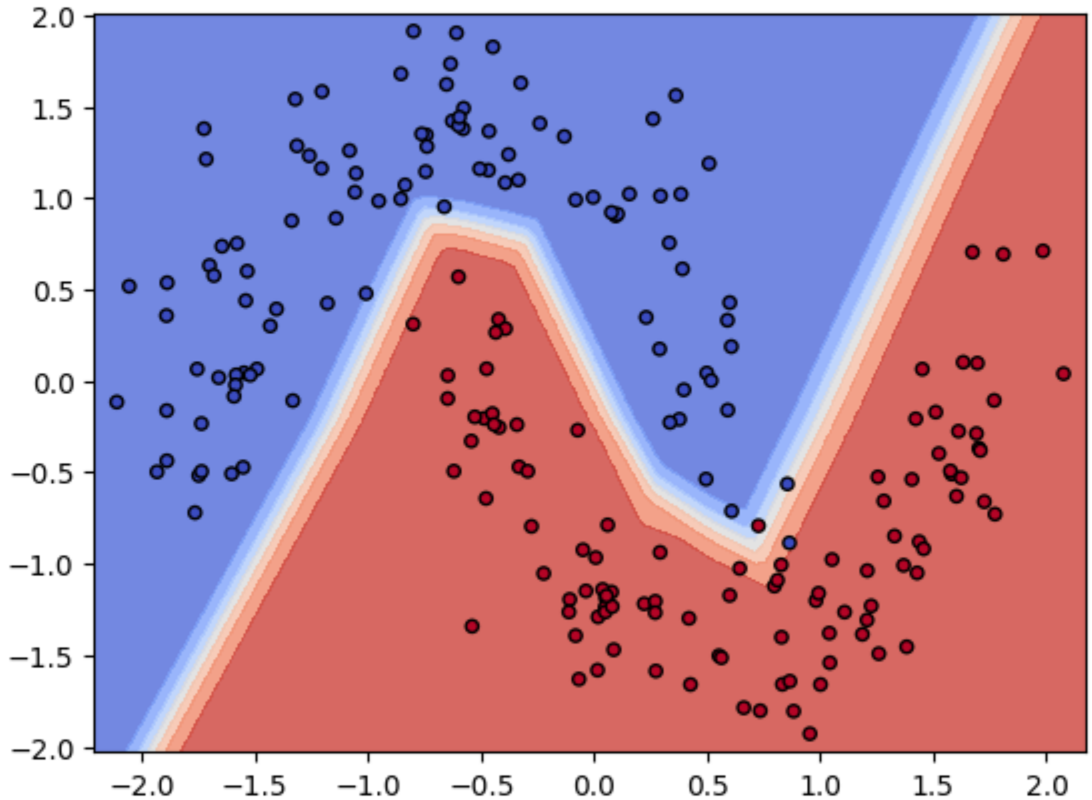
```
Epoch 0, Loss: 0.6577, Test Loss: 0.6362
Epoch 10, Loss: 0.4976, Test Loss: 0.4780
Epoch 20, Loss: 0.4118, Test Loss: 0.3897
Epoch 30, Loss: 0.3578, Test Loss: 0.3343
Epoch 40, Loss: 0.3266, Test Loss: 0.3018
Epoch 50, Loss: 0.3095, Test Loss: 0.2833
Epoch 60, Loss: 0.2997, Test Loss: 0.2725
Epoch 70, Loss: 0.2933, Test Loss: 0.2654
Epoch 80, Loss: 0.2884, Test Loss: 0.2600
Epoch 90, Loss: 0.2843, Test Loss: 0.2555
Epoch 100, Loss: 0.2804, Test Loss: 0.2514
Epoch 110, Loss: 0.2767, Test Loss: 0.2474
Epoch 120, Loss: 0.2732, Test Loss: 0.2436
Epoch 130, Loss: 0.2697, Test Loss: 0.2399
Epoch 140, Loss: 0.2661, Test Loss: 0.2364
Epoch 150, Loss: 0.2625, Test Loss: 0.2328
Epoch 160, Loss: 0.2589, Test Loss: 0.2292
Epoch 170, Loss: 0.2553, Test Loss: 0.2255
Epoch 180, Loss: 0.2515, Test Loss: 0.2219
Epoch 190, Loss: 0.2476, Test Loss: 0.2182
Epoch 200, Loss: 0.2435, Test Loss: 0.2144
Epoch 210, Loss: 0.2394, Test Loss: 0.2105
Epoch 220, Loss: 0.2352, Test Loss: 0.2065
Epoch 230, Loss: 0.2310, Test Loss: 0.2024
Epoch 240, Loss: 0.2267, Test Loss: 0.1982
Epoch 250, Loss: 0.2224, Test Loss: 0.1940
Epoch 260, Loss: 0.2180, Test Loss: 0.1897
Epoch 270, Loss: 0.2135, Test Loss: 0.1853
Epoch 280, Loss: 0.2088, Test Loss: 0.1808
Epoch 290, Loss: 0.2039, Test Loss: 0.1762
Epoch 300, Loss: 0.1989, Test Loss: 0.1715
Epoch 310, Loss: 0.1939, Test Loss: 0.1667
Epoch 320, Loss: 0.1888, Test Loss: 0.1619
Epoch 330, Loss: 0.1837, Test Loss: 0.1570
Epoch 340, Loss: 0.1786, Test Loss: 0.1521
Epoch 350, Loss: 0.1735, Test Loss: 0.1473
Epoch 360, Loss: 0.1684, Test Loss: 0.1426
Epoch 370, Loss: 0.1634, Test Loss: 0.1379
Epoch 380, Loss: 0.1584, Test Loss: 0.1333
Epoch 390, Loss: 0.1533, Test Loss: 0.1288
Epoch 400, Loss: 0.1484, Test Loss: 0.1243
Epoch 410, Loss: 0.1435, Test Loss: 0.1198
Epoch 420, Loss: 0.1387, Test Loss: 0.1155
Epoch 430, Loss: 0.1340, Test Loss: 0.1113
Epoch 440, Loss: 0.1295, Test Loss: 0.1072
Epoch 450, Loss: 0.1251, Test Loss: 0.1033
```

```
Epoch 460, Loss: 0.1209, Test Loss: 0.0996
Epoch 470, Loss: 0.1167, Test Loss: 0.0959
Epoch 480, Loss: 0.1127, Test Loss: 0.0924
Epoch 490, Loss: 0.1088, Test Loss: 0.0891
Epoch 500, Loss: 0.1052, Test Loss: 0.0860
Epoch 510, Loss: 0.1017, Test Loss: 0.0830
Epoch 520, Loss: 0.0983, Test Loss: 0.0802
Epoch 530, Loss: 0.0951, Test Loss: 0.0775
Epoch 540, Loss: 0.0921, Test Loss: 0.0749
Epoch 550, Loss: 0.0892, Test Loss: 0.0725
Epoch 560, Loss: 0.0865, Test Loss: 0.0703
Epoch 570, Loss: 0.0839, Test Loss: 0.0681
```

```python
# Plot our answers!

# Plot the training and test losses
fig, ax = plt.subplots()
plt.plot(train_losses, label="Train Loss")
plt.plot(test_losses, label="Test Loss")
plt.yscale("log")
plt.xlabel("Epochs")
plt.ylabel("BCE Loss")
plt.legend()

# Plot the decision boundary
fig, ax = plt.subplots()
x_min, x_max = X_test[:, 0].min() - 0.1, X_test[:, 0].max() + 0.1
y_min, y_max = X_test[:, 1].min() - 0.1, X_test[:, 1].max() + 0.1
xx, yy = jnp.meshgrid(jnp.linspace(x_min, x_max, 100), jnp.linspace(y_min, y_max, 100))
xs = jnp.array(list(zip(xx.ravel(), yy.ravel())))
print("Shape of xs for contour plot: ", xs.shape)  # Should be (10000, 2)
Z = vmap(classifier, in_axes=(0, None))(xs, params)
Z = Z.reshape(xx.shape)
ax.contourf(xx, yy, Z, alpha=0.8, cmap="coolwarm")
ax.scatter(
    X_test[:, 0],
    X_test[:, 1],
    c=y_test,
    edgecolors="k",
    marker="o",
    s=20,
    cmap="coolwarm",
)
```

```
Shape of xs for contour plot:  (10000, 2)
<matplotlib.collections.PathCollection at 0x7b2bc49bbc10>
```





Congrats! You have coded an MLP from scratch and trained it!

Exercise 2.1: Ruining the model

In this exercise, we will deliberate try to break the model. The goal here is to get a little experience with failure modes.

Try to deliberately overfit the model. You have a few knobs to play with: The depth L, the width N, the learning rate, the number of epochs, and the amount of training data. Try to find a combination of these that overfits the training data, i.e. the training loss is very low but the validation loss is high.

Try to see what happens if you forget to shuffle the data! This is a very common mistake. I made this mistake when writing this tutorial, and it took me about 30 minutes to realize why the model was not learning anything.

Try extreme learning rates, like 10^{-6} or 10^{2}.

When we loaded in the data a few cells ago, we normalized the data. What happens if the data is not O(1)? Try arranging the data so that the inputs are all $O(10^6)$ or $O(10^{-6})$. Technically, the optimal classifier should be coordinate-invariant, but in practice there can be issues! Try some other coordinate transforms of the data too.

Everything else unchanged, choosing $N \gtrapprox 32$ and $L \gtrapprox 8$ will overfit the training data. There are many other options though!

Exercise 2.2: MNIST with our JAX MLP

Modify the above code to classify MNIST. You will need to change the loss function and the output activation to account for the fact that MNIST is a multi-class classification problem.

```
# Prepare the MNIST Dataset
train_fraction = 0.8
test_fraction = 1 - train_fraction

# Shuffle the dataset before splitting
indices = np.random.permutation(len(X_mnist))
X_mnist = X_mnist[indices]
y_mnist = y_mnist[indices]
train_size = int(train_fraction * len(X_mnist))
X_train_mnist, y_train_mnist = X_mnist[:train_size], y_mnist[:train_size]
X_test_mnist, y_test_mnist = X_mnist[train_size:], y_mnist[train_size:]


def one_hot_encode(y, num_classes):
    return jax.nn.one_hot(y, num_classes)


input_dim = 784  # MNIST images are 28x28 pixels, flattened to 784
output_dim = 10  # 10 classes for digits 0-9


# YOUR PARAMETERS HERE
epochs = 1000
learning_rate = 0.1
N = 16
L = 3

# YOUR MODIFIED CLASSIFIER HERE
classifier_mnist = lambda x, params: jax.nn.softmax(MLP_jax(x, params))
```

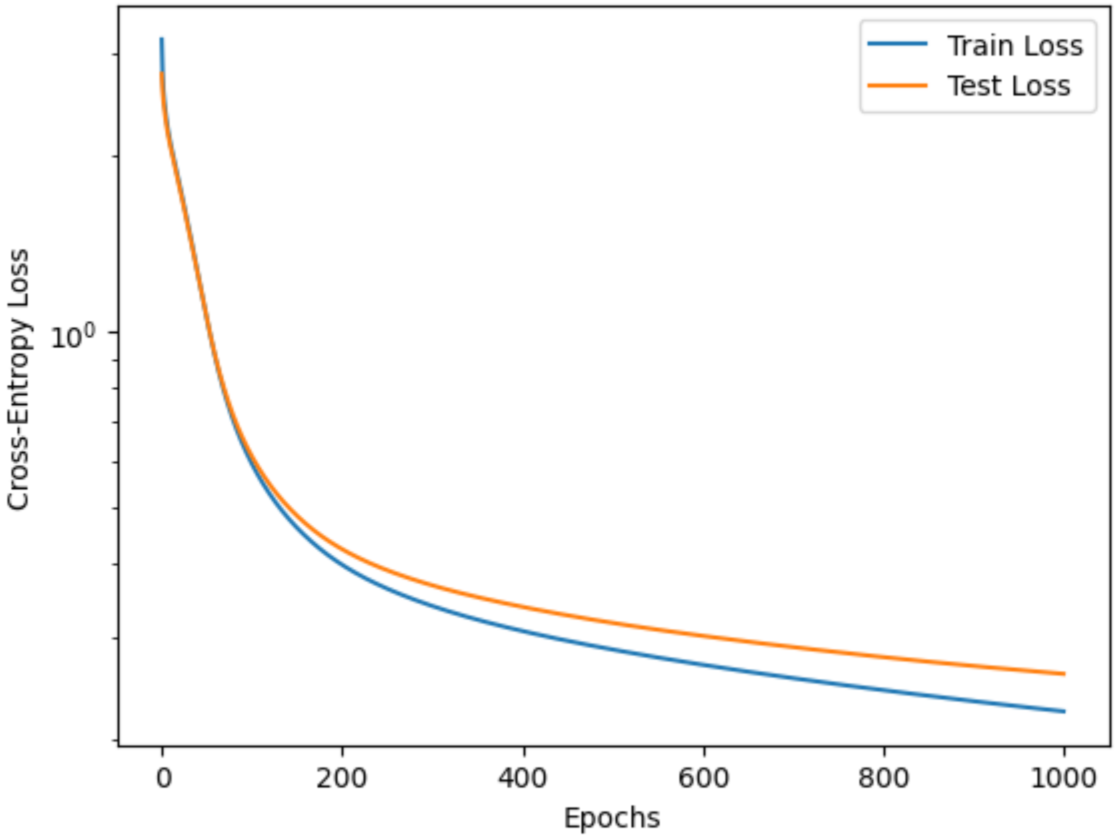
```
vmapped_classifier = vmap(classifier_mnist, in_axes=(0, None))


# YOUR MULTICLASS LOSS FUNCTION HERE
def multiclass_cross_entropy_loss(x, y_true, params):
    y_pred = vmapped_classifier(x, params)
    epsilon = 1e-10  # Small value to avoid log(0)
    return -jnp.mean(jnp.sum(y_true * jnp.log(y_pred + epsilon), axis=1))


# YOUR TRAINING LOOP HERE
params_mnist = init_params_jax(input_dim, output_dim, L, N)
train_losses_mnist = []
test_losses_mnist = []


@jit
def gradient_step_mnist(x, y_true, params, learning_rate=0.01):
    loss = multiclass_cross_entropy_loss(x, y_true, params)
    grads = grad(multiclass_cross_entropy_loss, argnums=2)(x, y_true, params)

    Ws, bs = params
    dWs, dbs = grads
    new_Ws = [W - learning_rate * dW for W, dW in zip(Ws, dWs)]
    new_bs = [b - learning_rate * db for b, db in zip(bs, dbs)]
    new_params = (new_Ws, new_bs)

    return new_params, loss


# Train the model
for epoch in range(epochs):
    params_mnist, loss = gradient_step_mnist(
        X_train_mnist,
        one_hot_encode(y_train_mnist, output_dim),
        params_mnist,
        learning_rate,
    )
    train_losses_mnist.append(loss)
    test_loss = multiclass_cross_entropy_loss(
        X_test_mnist, one_hot_encode(y_test_mnist, output_dim), params_mnist
    )
    test_losses_mnist.append(test_loss)

    # compute the accuracy
    y_pred = jnp.argmax(
        vmap(classifier_mnist, in_axes=(0, None))(X_test_mnist, params_mnist), axis=1
    )
    accuracy = jnp.mean(y_pred == y_test_mnist)
    if epoch % 10 == 0:
        print(
            f"Epoch {epoch}, Loss: {loss:.4f}, Test Loss: {test_loss:.4f}, Accuracy: {a
        )
```

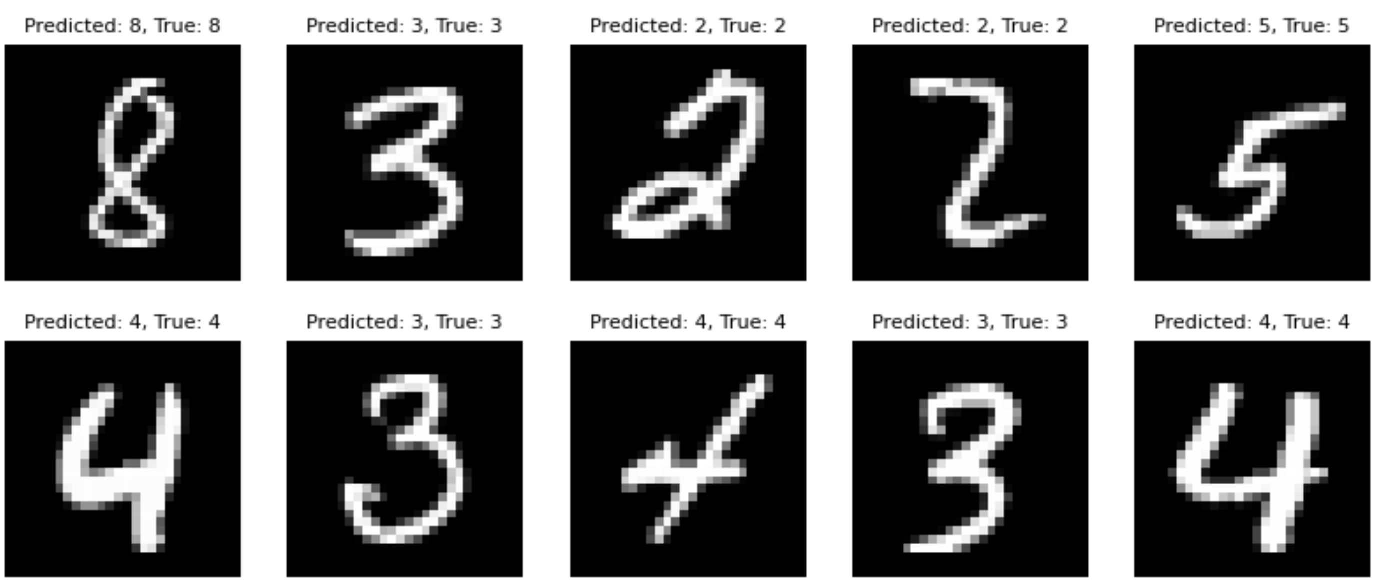
```
# Plot the training and test losses
fig, ax = plt.subplots()
plt.plot(train_losses_mnist, label="Train Loss")
plt.plot(test_losses_mnist, label="Test Loss")
plt.yscale("log")
plt.xlabel("Epochs")
plt.ylabel("Cross-Entropy Loss")
plt.legend()

# Plot some predictions
fig, ax = plt.subplots(2, 5, figsize=(10, 4))
for i in range(10):
    ax[i // 5, i % 5].imshow(X_test_mnist[i].reshape(28, 28), cmap="gray")
    ax[i // 5, i % 5].set_title(
        f"Predicted: {jnp.argmax(classifier_mnist(X_test_mnist[i], params_mnist))}, Tru
        fontsize=8,
    )
    ax[i // 5, i % 5].axis("off")
```

```
Epoch 0, Loss: 3.1699, Test Loss: 2.7661, Accuracy: 0.1026
Epoch 10, Loss: 2.0925, Test Loss: 2.0642, Accuracy: 0.3466
Epoch 20, Loss: 1.7817, Test Loss: 1.7619, Accuracy: 0.5511
Epoch 30, Loss: 1.5062, Test Loss: 1.4926, Accuracy: 0.6420
Epoch 40, Loss: 1.2603, Test Loss: 1.2527, Accuracy: 0.7009
Epoch 50, Loss: 1.0533, Test Loss: 1.0522, Accuracy: 0.7372
Epoch 60, Loss: 0.8946, Test Loss: 0.8997, Accuracy: 0.7644
Epoch 70, Loss: 0.7810, Test Loss: 0.7912, Accuracy: 0.7842
Epoch 80, Loss: 0.6999, Test Loss: 0.7137, Accuracy: 0.7979
Epoch 90, Loss: 0.6401, Test Loss: 0.6561, Accuracy: 0.8103
Epoch 100, Loss: 0.5938, Test Loss: 0.6116, Accuracy: 0.8208
Epoch 110, Loss: 0.5568, Test Loss: 0.5760, Accuracy: 0.8313
Epoch 120, Loss: 0.5266, Test Loss: 0.5468, Accuracy: 0.8409
Epoch 130, Loss: 0.5013, Test Loss: 0.5223, Accuracy: 0.8476
Epoch 140, Loss: 0.4799, Test Loss: 0.5017, Accuracy: 0.8544
Epoch 150, Loss: 0.4616, Test Loss: 0.4840, Accuracy: 0.8605
Epoch 160, Loss: 0.4457, Test Loss: 0.4687, Accuracy: 0.8654
Epoch 170, Loss: 0.4317, Test Loss: 0.4553, Accuracy: 0.8691
Epoch 180, Loss: 0.4194, Test Loss: 0.4436, Accuracy: 0.8726
Epoch 190, Loss: 0.4086, Test Loss: 0.4333, Accuracy: 0.8756
Epoch 200, Loss: 0.3989, Test Loss: 0.4242, Accuracy: 0.8782
Epoch 210, Loss: 0.3903, Test Loss: 0.4161, Accuracy: 0.8794
Epoch 220, Loss: 0.3825, Test Loss: 0.4087, Accuracy: 0.8817
Epoch 230, Loss: 0.3755, Test Loss: 0.4021, Accuracy: 0.8839
Epoch 240, Loss: 0.3690, Test Loss: 0.3960, Accuracy: 0.8857
Epoch 250, Loss: 0.3631, Test Loss: 0.3904, Accuracy: 0.8874
Epoch 260, Loss: 0.3576, Test Loss: 0.3853, Accuracy: 0.8893
Epoch 270, Loss: 0.3525, Test Loss: 0.3805, Accuracy: 0.8904
Epoch 280, Loss: 0.3478, Test Loss: 0.3760, Accuracy: 0.8926
Epoch 290, Loss: 0.3433, Test Loss: 0.3718, Accuracy: 0.8940
Epoch 300, Loss: 0.3391, Test Loss: 0.3679, Accuracy: 0.8953
Epoch 310, Loss: 0.3351, Test Loss: 0.3642, Accuracy: 0.8964
Epoch 320, Loss: 0.3314, Test Loss: 0.3607, Accuracy: 0.8974
Epoch 330, Loss: 0.3279, Test Loss: 0.3574, Accuracy: 0.8981
Epoch 340, Loss: 0.3245, Test Loss: 0.3542, Accuracy: 0.8988
Epoch 350, Loss: 0.3213, Test Loss: 0.3512, Accuracy: 0.8993
```

```
Epoch 350, Loss: 0.3213, Test Loss: 0.3512, Accuracy: 0.8993
Epoch 360, Loss: 0.3182, Test Loss: 0.3484, Accuracy: 0.8999
Epoch 370, Loss: 0.3153, Test Loss: 0.3456, Accuracy: 0.9006
Epoch 380, Loss: 0.3125, Test Loss: 0.3430, Accuracy: 0.9019
Epoch 390, Loss: 0.3098, Test Loss: 0.3404, Accuracy: 0.9029
Epoch 400, Loss: 0.3072, Test Loss: 0.3380, Accuracy: 0.9034
Epoch 410, Loss: 0.3047, Test Loss: 0.3356, Accuracy: 0.9039
Epoch 420, Loss: 0.3023, Test Loss: 0.3333, Accuracy: 0.9042
Epoch 430, Loss: 0.3000, Test Loss: 0.3311, Accuracy: 0.9049
Epoch 440, Loss: 0.2978, Test Loss: 0.3289, Accuracy: 0.9055
Epoch 450, Loss: 0.2956, Test Loss: 0.3268, Accuracy: 0.9064
Epoch 460, Loss: 0.2935, Test Loss: 0.3248, Accuracy: 0.9071
Epoch 470, Loss: 0.2914, Test Loss: 0.3228, Accuracy: 0.9078
Epoch 480, Loss: 0.2894, Test Loss: 0.3209, Accuracy: 0.9081
Epoch 490, Loss: 0.2875, Test Loss: 0.3190, Accuracy: 0.9083
Epoch 500, Loss: 0.2856, Test Loss: 0.3172, Accuracy: 0.9086
Epoch 510, Loss: 0.2838, Test Loss: 0.3155, Accuracy: 0.9091
Epoch 520, Loss: 0.2820, Test Loss: 0.3137, Accuracy: 0.9093
Epoch 530, Loss: 0.2802, Test Loss: 0.3121, Accuracy: 0.9099
Epoch 540, Loss: 0.2785, Test Loss: 0.3104, Accuracy: 0.9104
Epoch 550, Loss: 0.2768, Test Loss: 0.3088, Accuracy: 0.9109
Epoch 560, Loss: 0.2752, Test Loss: 0.3073, Accuracy: 0.9113
Epoch 570, Loss: 0.2736, Test Loss: 0.3058, Accuracy: 0.9118
Epoch 580, Loss: 0.2720, Test Loss: 0.3043, Accuracy: 0.9121
Epoch 590, Loss: 0.2704, Test Loss: 0.3028, Accuracy: 0.9123
Epoch 600, Loss: 0.2689, Test Loss: 0.3014, Accuracy: 0.9129
Epoch 610, Loss: 0.2674, Test Loss: 0.3000, Accuracy: 0.9137
Epoch 620, Loss: 0.2660, Test Loss: 0.2986, Accuracy: 0.9141
Epoch 630, Loss: 0.2645, Test Loss: 0.2973, Accuracy: 0.9144
Epoch 640, Loss: 0.2631, Test Loss: 0.2959, Accuracy: 0.9145
Epoch 650, Loss: 0.2617, Test Loss: 0.2946, Accuracy: 0.9149
Epoch 660, Loss: 0.2604, Test Loss: 0.2933, Accuracy: 0.9152
Epoch 670, Loss: 0.2590, Test Loss: 0.2920, Accuracy: 0.9157
Epoch 680, Loss: 0.2577, Test Loss: 0.2908, Accuracy: 0.9161
Epoch 690, Loss: 0.2564, Test Loss: 0.2896, Accuracy: 0.9163
Epoch 700, Loss: 0.2551, Test Loss: 0.2884, Accuracy: 0.9166
Epoch 710, Loss: 0.2539, Test Loss: 0.2872, Accuracy: 0.9171
Epoch 720, Loss: 0.2526, Test Loss: 0.2860, Accuracy: 0.9176
Epoch 730, Loss: 0.2514, Test Loss: 0.2848, Accuracy: 0.9180
Epoch 740, Loss: 0.2502, Test Loss: 0.2837, Accuracy: 0.9181
Epoch 750, Loss: 0.2490, Test Loss: 0.2826, Accuracy: 0.9186
Epoch 760, Loss: 0.2478, Test Loss: 0.2815, Accuracy: 0.9192
Epoch 770, Loss: 0.2467, Test Loss: 0.2804, Accuracy: 0.9196
Epoch 780, Loss: 0.2456, Test Loss: 0.2793, Accuracy: 0.9199
Epoch 790, Loss: 0.2444, Test Loss: 0.2783, Accuracy: 0.9203
Epoch 800, Loss: 0.2433, Test Loss: 0.2772, Accuracy: 0.9206
Epoch 810, Loss: 0.2422, Test Loss: 0.2762, Accuracy: 0.9210
Epoch 820, Loss: 0.2412, Test Loss: 0.2752, Accuracy: 0.9214
Epoch 830, Loss: 0.2401, Test Loss: 0.2742, Accuracy: 0.9219
Epoch 840, Loss: 0.2391, Test Loss: 0.2732, Accuracy: 0.9223
Epoch 850, Loss: 0.2380, Test Loss: 0.2723, Accuracy: 0.9224
Epoch 860, Loss: 0.2370, Test Loss: 0.2713, Accuracy: 0.9226
Epoch 870, Loss: 0.2360, Test Loss: 0.2704, Accuracy: 0.9229
Epoch 880, Loss: 0.2350, Test Loss: 0.2695, Accuracy: 0.9234
Epoch 890, Loss: 0.2340, Test Loss: 0.2686, Accuracy: 0.9236
Epoch 900, Loss: 0.2330, Test Loss: 0.2677, Accuracy: 0.9241
Epoch 910, Loss: 0.2320, Test Loss: 0.2668, Accuracy: 0.9243
Epoch 920, Loss: 0.2311, Test Loss: 0.2660, Accuracy: 0.9246
Epoch 930, Loss: 0.2301, Test Loss: 0.2651, Accuracy: 0.9253
```

```
Epoch 930, Loss: 0.2301, Test Loss: 0.2651, Accuracy: 0.9253
Epoch 940, Loss: 0.2292, Test Loss: 0.2643, Accuracy: 0.9257
Epoch 950, Loss: 0.2283, Test Loss: 0.2635, Accuracy: 0.9261
Epoch 960, Loss: 0.2274, Test Loss: 0.2627, Accuracy: 0.9265
Epoch 970, Loss: 0.2265, Test Loss: 0.2619, Accuracy: 0.9268
Epoch 980, Loss: 0.2256, Test Loss: 0.2611, Accuracy: 0.9270
Epoch 990, Loss: 0.2247, Test Loss: 0.2603, Accuracy: 0.9272
```





Chapter 2.3 (BONUS): Solving ODE's with MLPs and Autodiff

We'll explore how we can use MLPs to solve ODEs. The idea is to represent our solution $f(x)$ as an

MLP, and write the ODE as the minimum of a loss function. We can use autodiff to compute the derivatives of $f(x)$, and then use gradient descent to minimize the loss function. Unlike what we did above, there is no training (this is not a statistical problem). Also, we are now dealing with gradients of the function within respect to the input AND with respect to the model parameters.

Suppose we have an ODE of the form $F(f, f', x) = 0$, where f is the function we want to solve for, f' is the derivative of f with respect to x, and F is some function that defines the ODE. We can define a loss function as:

$$L[f] = \int dx |F(f, f', x)|^2$$

where the integral is over the domain of x we are interested in. The goal is to minimize this loss function with respect to the parameters of the MLP that defines $f(x)$.

HINT: If $f(x)$ is just an MLP, then your solution will likely just collapse to just $f(x) = 0$. Try to find a way to write $f(x) =$ something involving an MLP but also manifestly satisfies the initial condition.

BONUS: If you try to do a second-order ODE using our MLP, the solution will fail miserably. Why? Hint: this relates to piecewise-linearity. Can you fix this?

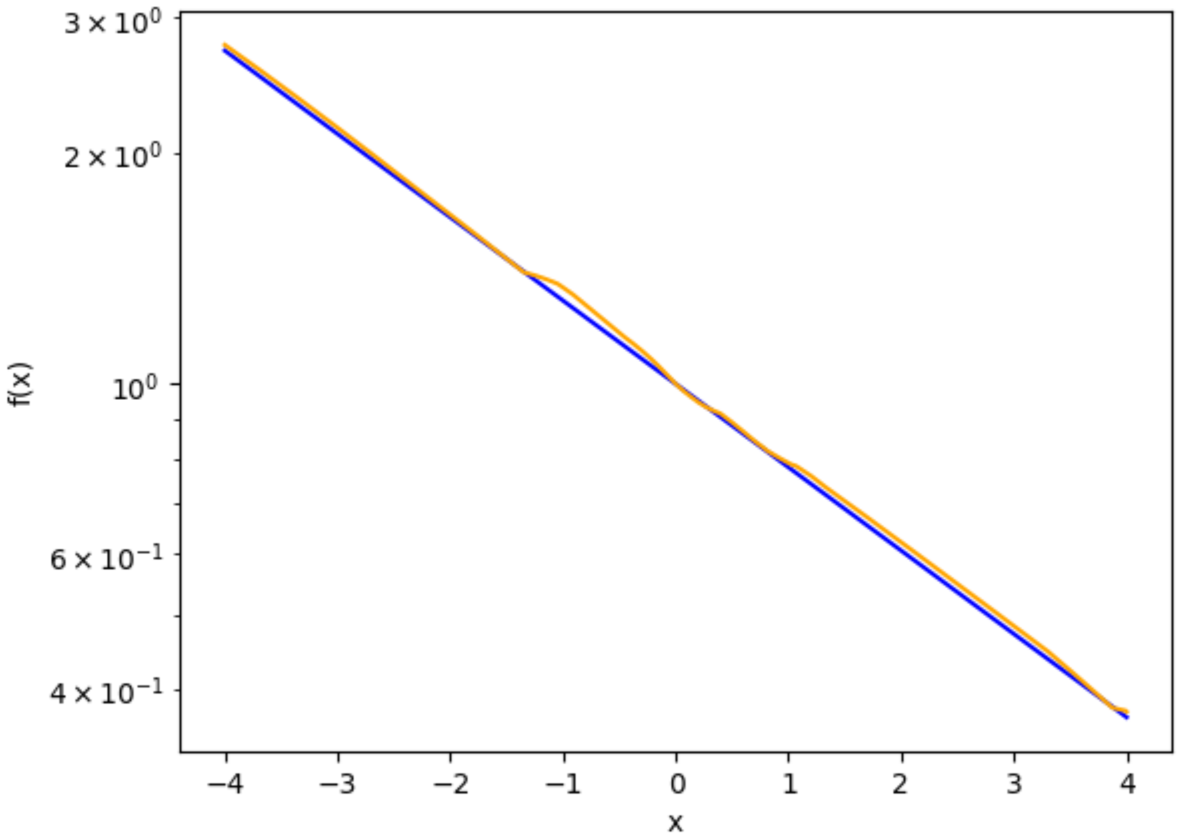
Fill out the rest of this code to build our approximate ODE solver! This requires only minor modifications to the code we have already written above.

```
# We will only solve the ODE on a compact domain
x_domain = jnp.linspace(-4, 4, 1000)
x_domain = x_domain.reshape(-1, 1)  # Reshape to (1000, 1) for compatibility with MLP

x_0 = 0
f_0 = 1  # Initial condition: f(0) = 1


def my_solution(x, params):
    g_x = MLP_jax(x, params)  # Get the output of the MLP

    # This solution will automatically satisfy the initial condition
    f_x = f_0 + (x - x_0) * g_x

    return f_x


vmapped_ODE_solution = vmap(
    my_solution, in_axes=(0, None)
)  # Vectorized solution function

# lambda function to make the output of my_solution a scalar so that we can compute the
vmapped_grad_ODE = vmap(
    grad(lambda x, params: my_solution(x, params).squeeze(), argnums=0),
    in_axes=(0, None),
)  # Vectorized gradient of MLP
```

```
)  # Vectorized gradient of MLP


def F(f_x, grad_f_x, x):
    # Example ODE: f = exp(-0.25 * x)
    ODE_term = 0.25 * f_x + grad_f_x

    return ODE_term


vmapped_grad_MLP = vmap(
    grad(lambda x, params: MLP_jax(x, params).squeeze(), argnums=0), in_axes=(0, None)
)  # Vectorized gradient of MLP


def ODE_loss(params):
    # Compute the loss as the mean squared error between the network output and the ODE
    f_x = vmapped_ODE_solution(x_domain, params)  # params is a tuple of (Ws, bs)
    grad_f_x = vmapped_grad_ODE(x_domain, params)

    # Compute the ODE residual
    residual = F(f_x, grad_f_x, x_domain)

    # Mean squared error loss
    return jnp.mean(residual**2)


def gradient_step_ODE(params, learning_rate=0.01):
    # Compute the loss and its gradient
    loss = ODE_loss(params)
    grads = grad(ODE_loss)(params)

    # Update the network parameters using gradient descent
    Ws, bs = params
    dWs, dbs = grads
    new_Ws = [W - learning_rate * dW for W, dW in zip(Ws, dWs)]
    new_bs = [b - learning_rate * db for b, db in zip(bs, dbs)]
    new_params = (new_Ws, new_bs)

    return new_params, loss


# Initialize the parameters for the ODE solver
input_dim = 1
output_dim = 1
L = 3
N = 17
params = init_params_jax(input_dim, output_dim, L, N)


# Train the ODE solver
epochs = 1000
for epoch in range(epochs):
    params, loss = gradient_step_ODE(params, learning_rate=0.01)
```

```
    params, loss = gradient_step_ODE(params, learning_rate=0.01)
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Plot the solution
y_vals = vmapped_ODE_solution(x_domain, params)
plt.plot(
    x_domain, jnp.exp(-0.25 * x_domain), label="True Solution: exp(-x)", color="blue"
)
plt.plot(x_domain, y_vals, label="MLP Solution", color="orange")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.yscale("log")
```

```
Epoch 0, Loss: 1.0988
Epoch 100, Loss: 0.0113
Epoch 200, Loss: 0.0052
Epoch 300, Loss: 0.0038
Epoch 400, Loss: 0.0037
Epoch 500, Loss: 0.0036
Epoch 600, Loss: 0.0034
Epoch 700, Loss: 0.0027
Epoch 800, Loss: 0.0021
Epoch 900, Loss: 0.0020
```



Chapter 2.4: Some notes on JAX Prebuilt Libraries

JAX has a number of prebuilt libraries that can be useful for machine learning. Some of the most popular ones are:

1. **Flax**: A neural network library for JAX that provides a high-level interface for building and training neural networks. It is similar to PyTorch in terms of functionality, but uses JAX's functional programming style.
2. **stax**: A library for probabilistic programming in JAX. It provides a high-level interface for building and training probabilistic models, and is similar to PyMC3 or TensorFlow Probability.

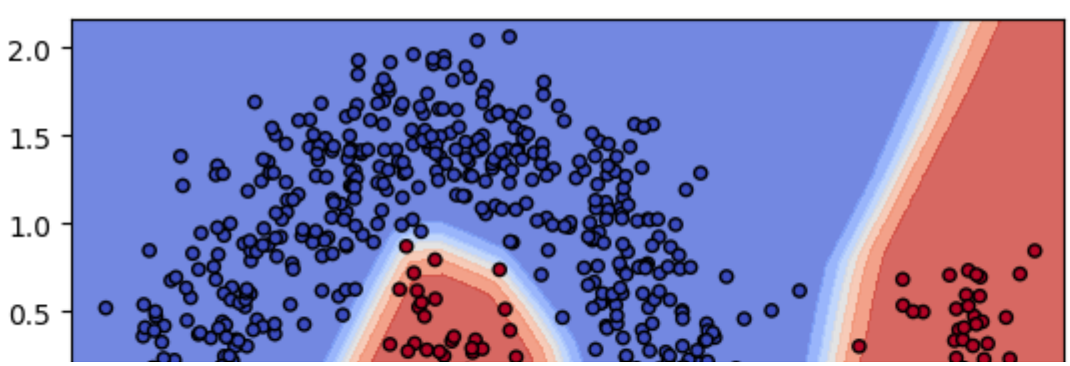
However, we will not cover these in any great depth This is because if you are going to use prebuilt libraries, you are probably better off using PyTorch or TensorFlow, which have more mature libraries and a larger community. JAX is more useful for experimentation with explicitly defined models and getting into the guts of it all.

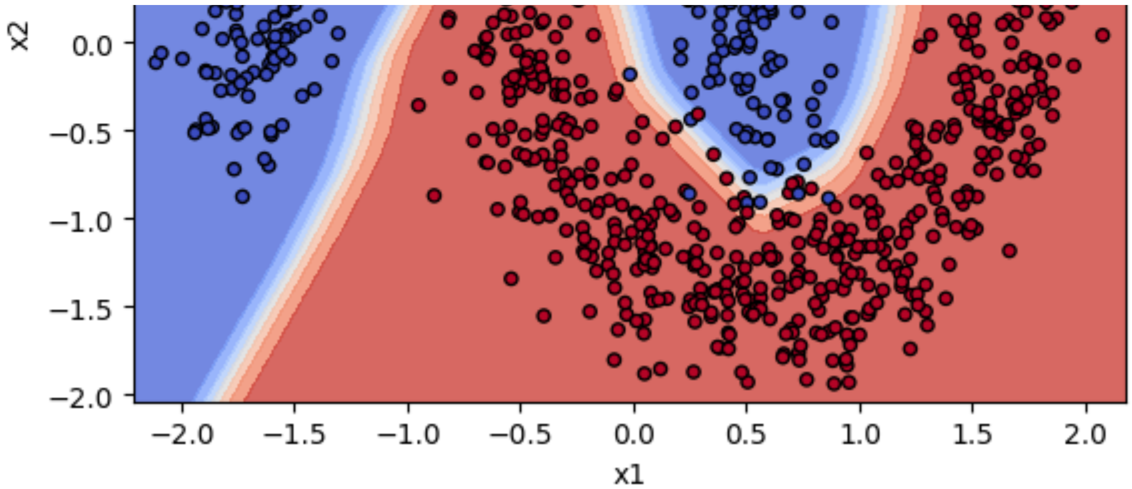
Just for completeness, we will show how to use stax to build a simple MLP. This is not meant to be a comprehensive tutorial on these libraries, but rather a quick introduction to their usage so you can see the syntax. The syntax between stax and flax is virtually identical. Both are also directly meant to mimic PyTorch anyways, so after this example, we will go into an in-depth PyTorch tutorial.

```
from jax.example_libraries import stax

# prepare data
indices = np.random.permutation(len(X_moons))
X_moons = X_moons[indices]
y_moons = y_moons[indices]
X_moons = X_moons.astype(np.float32)  # Convert to float32 for JAX compatibility
y_moons = y_moons.astype(np.float32)  # Convert to float32 for JAX compatibility


# A model is a sequential list of layers
init, apply = stax.serial(
    stax.Dense(32),
    stax.Relu,  # Dense(N) is a fully connected layer with output dimension N. The inpu
    stax.Dense(32),
    stax.Relu,
    stax.Dense(32),
    stax.Relu,
    stax.Dense(1),
)


# init is the function to initialize the parameters of the model.
# apply is the function to apply the model to the input data.

key = random.PRNGKey(0)
key1, key2 = random.split(key)
_, params = init(key2, (-1, 2))


# Define a loss function for binary classification
```

```
def loss(params, x, y):
    logits = jnp.squeeze(apply(params, x))
    return -jnp.mean(
        y * jax.nn.log_sigmoid(logits) + (1 - y) * jax.nn.log_sigmoid(-logits)
    )


@jit  # Same type of gradient step as before, but now using stax
def step(params, x, y, lr=0.01):
    grads = grad(loss)(params, x, y)

    # tree_map is a JAX function that applies a function to each leaf of a pytree (like
    # Makes it easy to update the parameters all at once
    return jax.tree_util.tree_map(lambda a, b: a - lr * b, params, grads)


for _ in range(10000):
    params = step(params, X_moons, y_moons)

print("stax loss:", loss(params, X_moons, y_moons))

# Decision boundary
x_min, x_max = X_moons[:, 0].min() - 0.1, X_moons[:, 0].max() + 0.1
y_min, y_max = X_moons[:, 1].min() - 0.1, X_moons[:, 1].max() + 0.1
xx, yy = jnp.meshgrid(jnp.linspace(x_min, x_max, 100), jnp.linspace(y_min, y_max, 100))
xs = jnp.array(list(zip(xx.ravel(), yy.ravel())))
Z = jnp.squeeze(apply(params, xs))
Z = Z.reshape(xx.shape)
fig, ax = plt.subplots()
ax.contourf(xx, yy, jax.nn.sigmoid(Z), alpha=0.8, cmap="coolwarm")
ax.scatter(
    X_moons[:, 0],
    X_moons[:, 1],
    c=y_moons,
    edgecolors="k",
    marker="o",
    s=20,
    cmap="coolwarm",
)
plt.xlabel("x1")
plt.ylabel("x2")
```

```
stax loss: 0.025776248
Text(0, 0.5, 'x2')
```





Chapter 3: PyTorch

PyTorch is a popular ML library developed by Meta (formerly Facebook). It is widely used in industry and research. It is typically more high-level than JAX, and has a wider range of prebuilt modules and utilities for common ML tasks. It is also older and more widely supported with documentation and tutorials online.

Unlike JAX, PyTorch is object-oriented, meaning that we will define an MLP as a class with an internal state that keeps track of the parameters.

```
import torch
import torch.nn as nn

device = "cuda" if torch.cuda.is_available() else "cpu"
```

Chapter 3.1: Primer on PyTorch tensors and autodiff

PyTorch has its own tensor class, which are similar to (but not the same as) numpy arrays. They can live on a "device" (e.g. `cuda` or `cpu`), and can be used to perform computations on that device. PyTorch tensors have many of the same methods as numpy arrays, but also have some additional methods for ML tasks, such as `backward()` for computing gradients.

Autodiff in PyTorch is different than in JAX. In PyTorch, we define a computation graph by performing operations on tensors, and then call `backward()` on the output tensor to compute the gradients. This is different from JAX, where we define a function and then call `grad()` on that function to compute the gradients. Once a tensor is in the graph, it cannot be converted back to numpy without first `detach`ing it.

To emphasize, in JAX, we take derivatives of functions, and the derivative of a function is another function. In PyTorch, we tell the graph to keep track of a tensor, we pass the tensor through a function, and then we ``backpropagate'' the resultant tensor. Derivatives are computed on the output tensor and the result is a tensor.

```
# A pytorch tensor defined from our numpy moons dataset
x_pytorch = torch.tensor(X_moons, dtype=torch.float32, device=device)

# Converting the pytorch tensor back to a numpy tensor
x_numpy = x_pytorch.detach().cpu().numpy()
# .detach() is used to remove the tensor from the computation graph, so it can be conve
# .cpu() is used to move the tensor to the CPU, since numpy only works with CPU tensors


##### DEFINING GRADIENTS AND COMPUTATION GRAPHS ######


# Define a function and compute gradients
def f_torch(x):
    return torch.sin(10 * x) * torch.exp(-2 * x**2)


# requires_grad=True to compute gradients later! Try setting to False and see what happ

x_single = torch.tensor([0.0], device=device, requires_grad=True)  # Single value tenso
y_single = f_torch(x_single)  # Compute the function value


# We want to compute the gradient of the function and plot it.
# First, we call .backward on the output of the function (not the function itself!).
# This tells Pytorch its time to compute the gradients in the computational graph.

y_single.backward()  # Compute the gradient for the single value

# Now the gradients are computed. To access it, we use .grad() on the input tensor.
print("Gradient at x=0.0: ", x_single.grad.item())  # Should print the gradient at x=0.


##### MULTIPLE INPUTS ######

# The same as above, but now on a vectorized input
xs = torch.linspace(-1, 1, 1000, device=device, requires_grad=True)
ys = f_torch(xs)

# We cannot just use ys.backward() because ys is a vector, not a scalar.
# ys has 1000 elements, and xs has 1000 elements, so PyTorch thinks there is a 1000 x 1
# We need to specify a gradient direction to sum the gradients over the output dimensio
gradient_direction = torch.ones_like(ys, device=device)  #
_ = ys.backward(gradient=gradient_direction)  # Compute the gradients
# Now we can access the gradients
xs_grad = (
    xs.grad
)  # This will be a tensor of the same shape as xs in the specified direction.
```

```
)  # This will be a tensor of the same shape as xs in the specified direction.


# Plot the function and its gradient
plt.plot(xs.detach().cpu().numpy(), ys.detach().cpu().numpy(), label="f(x)")
plt.plot(xs.detach().cpu().numpy(), xs_grad.detach().cpu().numpy(), label="f'(x)")



# Zero-ing out gradients
# In PyTorch, gradients accumulate by default, so we need to zero them out before the n
x_single.grad.zero_()  # Zero out the gradients
xs.grad.zero_()  # Zero out the gradients
```

```
    Gradient at x=0.0:  10.0
    tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
            0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
    0., 0., 0., 0., 0.,
```

```
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
         0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.,
```

Chapter 3.2: Pre-built PyTorch Modules

The `Sequential` class

There are many possible layers of abstraction in PyTorch. The highest-level, "I don't care, I just want a neural network, don't bother me with details" way to define an MLP is to use the `Sequential` class.

Most models are simply a sequence of layers, where each layer is a function that takes in the output of the previous layer and produces an output. The `Sequential` class allows us to define a model as a sequence of layers, where each layer is applied in order. The `nn` module provides many prebuilt layers, such as `Linear`, `ReLU`, and `Softmax`, that can be used to define a model. Note that `Linear` actually means "Affine", i.e. it implements the affine transformation $Wx + b$.

Let's see how to implement an MLP. Compared to the above implementations, this will be much shorter and cleaner, since most of this is already implemented for us in PyTorch (but the cost is that we have less control over the details of the implementation, like if we wanted to mess with layer weights). Note that we don't even have to worry about the initialization of the weights, since PyTorch does this for us automatically.

We also don't have to bother defining a loss function or gradient descent, sice this also already exists.

```
input_dim = 2
output_dim = 1


model = nn.Sequential(
```

```
    nn.Linear(input_dim, 32),
    nn.ReLU(),
    nn.Linear(32, 32),
    nn.ReLU(),  # We just stack a bunch of layers
    nn.Linear(
        32, output_dim
    ),  # No activation on the last layer. Instead, we'll put the sigmoid or softmax in
).to(
    device
)  # .to(device) moves the model to the GPU if available


# Lots of optimizers to choose from! SGD is ordinary (stochastic) gradient descent, Ada
opt_sgd = torch.optim.SGD(model.parameters(), lr=1e-2)
opt_adam = torch.optim.Adam(model.parameters(), lr=1e-2)


# Prebuilt BCE. Already includes the sigmoid, so we don't need to apply it in the model
loss_fn = nn.BCEWithLogitsLoss()


# Train the model
epochs = 300
train_fraction = 0.8
opt = opt_adam  # Choose the optimizer

X_torch = torch.tensor(X_moons, dtype=torch.float32, device=device)
y_torch = torch.tensor(y_moons, dtype=torch.float32, device=device).unsqueeze(
    1
)  # Unsqueeze to make it a column vector

# Training/test split, shuffle the dataset
train_size = int(train_fraction * len(X_torch))
indices = torch.randperm(len(X_torch))  # Shuffle the dataset
X_torch = X_torch[indices]
y_torch = y_torch[indices]
X_train, y_train = X_torch[:train_size], y_torch[:train_size]
X_test, y_test = X_torch[train_size:], y_torch[train_size:]


train_losses = []
test_losses = []

for _ in range(300):
    opt.zero_grad()  # Zero out the gradients before the backward pass. VITAL!
    loss = loss_fn(model(X_train), y_train)
    loss.backward()  # Tell the graph its time to compute the gradients
    opt.step()  # "step" automatically updates the parameters using the gradients compu
    train_losses.append(loss.item())  # Save the training loss
    test_loss = loss_fn(model(X_test), y_test)
    test_losses.append(test_loss.item())  # Save the test loss
    if _ % 10 == 0:
        print(
            f"Epoch {_}, Train Loss: {loss.item():.4f}, Test Loss: {test_loss.item():.4
```

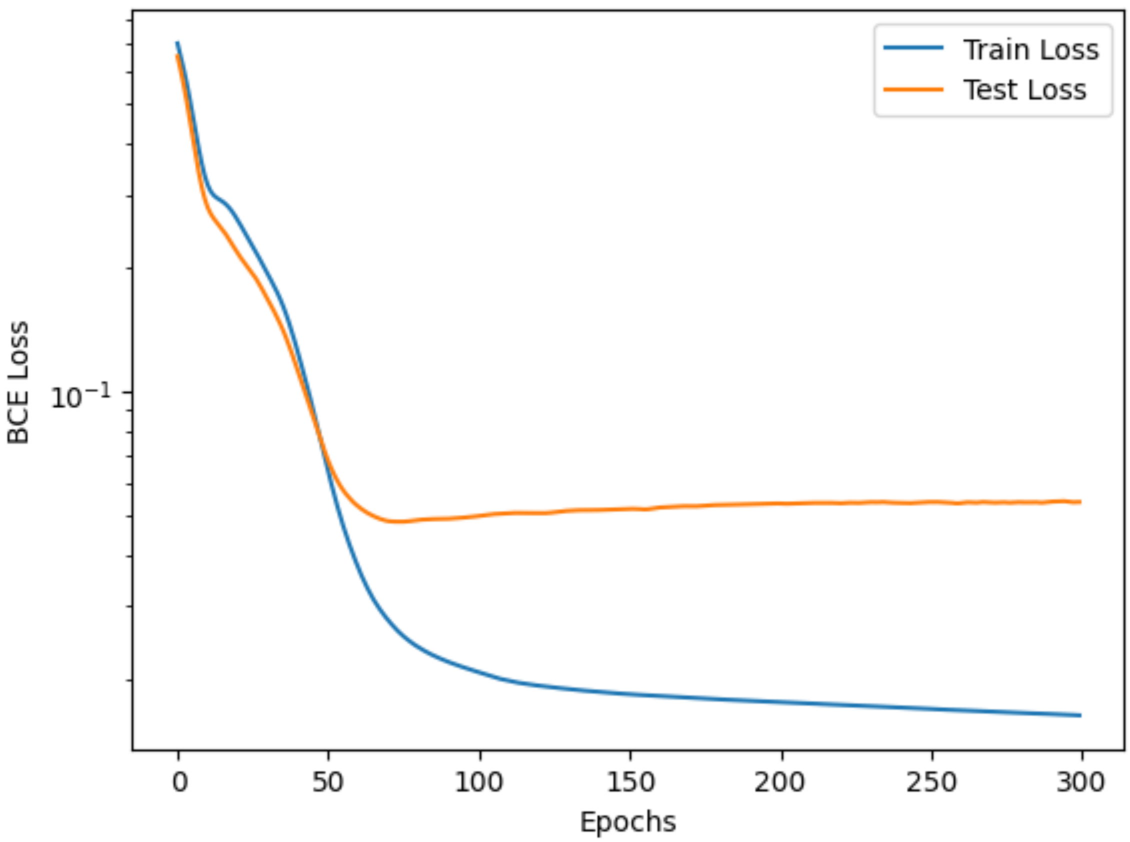
```
            f"Epoch {_}, Train Loss: {loss.item():.4f}, Test Loss: {test_loss.item():.4f}"
        )

print("PyTorch loss:", loss.item())
```

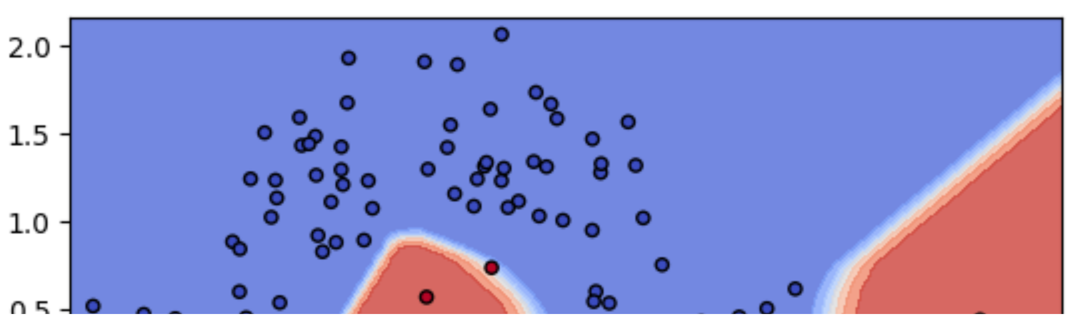
```
     Epoch 0, Train Loss: 0.7024, Test Loss: 0.6533
     Epoch 10, Train Loss: 0.3179, Test Loss: 0.2803
     Epoch 20, Train Loss: 0.2606, Test Loss: 0.2166
     Epoch 30, Train Loss: 0.1924, Test Loss: 0.1668
     Epoch 40, Train Loss: 0.1238, Test Loss: 0.1116
     Epoch 50, Train Loss: 0.0637, Test Loss: 0.0677
     Epoch 60, Train Loss: 0.0371, Test Loss: 0.0524
     Epoch 70, Train Loss: 0.0277, Test Loss: 0.0483
     Epoch 80, Train Loss: 0.0238, Test Loss: 0.0487
     Epoch 90, Train Loss: 0.0219, Test Loss: 0.0490
     Epoch 100, Train Loss: 0.0207, Test Loss: 0.0498
     Epoch 110, Train Loss: 0.0198, Test Loss: 0.0506
     Epoch 120, Train Loss: 0.0192, Test Loss: 0.0506
     Epoch 130, Train Loss: 0.0189, Test Loss: 0.0513
     Epoch 140, Train Loss: 0.0186, Test Loss: 0.0515
     Epoch 150, Train Loss: 0.0183, Test Loss: 0.0518
     Epoch 160, Train Loss: 0.0181, Test Loss: 0.0522
     Epoch 170, Train Loss: 0.0180, Test Loss: 0.0525
     Epoch 180, Train Loss: 0.0178, Test Loss: 0.0530
     Epoch 190, Train Loss: 0.0177, Test Loss: 0.0532
     Epoch 200, Train Loss: 0.0175, Test Loss: 0.0534
     Epoch 210, Train Loss: 0.0174, Test Loss: 0.0535
     Epoch 220, Train Loss: 0.0173, Test Loss: 0.0534
     Epoch 230, Train Loss: 0.0171, Test Loss: 0.0538
     Epoch 240, Train Loss: 0.0170, Test Loss: 0.0535
     Epoch 250, Train Loss: 0.0169, Test Loss: 0.0538
     Epoch 260, Train Loss: 0.0168, Test Loss: 0.0535
     Epoch 270, Train Loss: 0.0166, Test Loss: 0.0536
     Epoch 280, Train Loss: 0.0165, Test Loss: 0.0537
     Epoch 290, Train Loss: 0.0164, Test Loss: 0.0539
     PyTorch loss: 0.01629592292010784
```

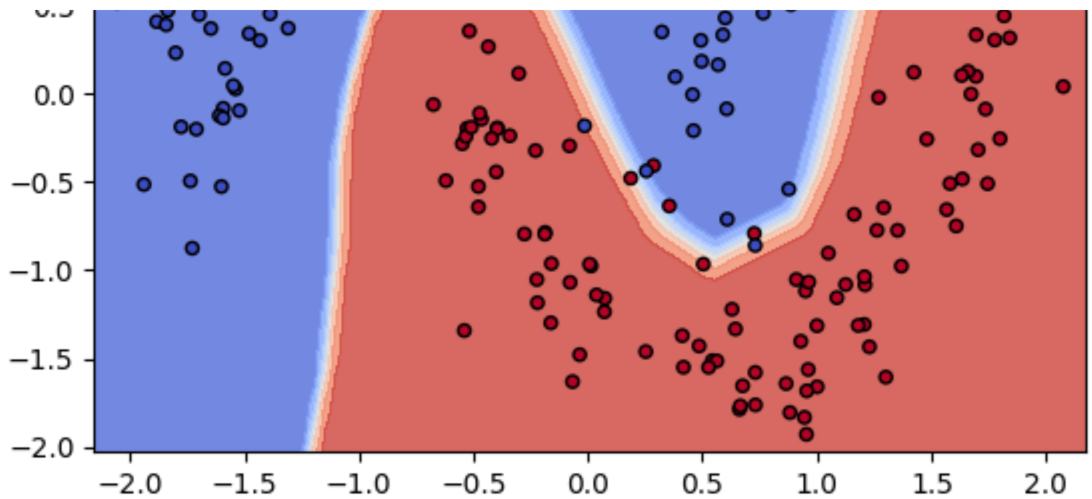
```
# Plot the training and test losses
fig, ax = plt.subplots()
plt.plot(train_losses, label="Train Loss")
plt.plot(test_losses, label="Test Loss")
plt.yscale("log")
plt.xlabel("Epochs")
plt.ylabel("BCE Loss")
plt.legend()

# Plot the decision boundary
fig, ax = plt.subplots()
x_min, x_max = X_test[:, 0].min() - 0.1, X_test[:, 0].max() + 0.1
y_min, y_max = X_test[:, 1].min() - 0.1, X_test[:, 1].max() + 0.1
xx, yy = torch.meshgrid(
    torch.linspace(x_min, x_max, 100), torch.linspace(y_min, y_max, 100)
)
xs = torch.stack([xx.ravel(), yy.ravel()], dim=1).to(
    device
)  # Stack to create a grid of points
```

```
Z = model(xs)  # Get the model predictions


# Dont forget to apply the sigmoid to the logits!
Z = torch.sigmoid(Z)  # Apply sigmoid to the logits

Z = Z.detach().cpu().numpy()
Z = Z.reshape(xx.shape)  # Reshape to match the grid shape
ax.contourf(xx.cpu().numpy(), yy.cpu().numpy(), Z, alpha=0.8, cmap="coolwarm")
ax.scatter(
    X_test[:, 0].cpu().numpy(),
    X_test[:, 1].cpu().numpy(),
    c=y_test.cpu().numpy(),
    edgecolors="k",
    marker="o",
    s=20,
    cmap="coolwarm",
)
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/torch/functional.py:539: UserWarning: torch
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```







Exercise 3.1:

Now that we don't have to code anything, lets try some more advanced things. 1) Compare the ADAM optimizer to the SGD optimizer. 2) Trade out the ReLU activation for sigmoid (used in the original MLP's back in the day), selu (a variant of ReLU that doesn't go to 0), and others. How do they compare? The modern lore in 2025 is to use "Swish" (aka "SiLU") activations, which are supposedly better than ReLU. There are many more at https://docs.pytorch.org/docs/main/nn.functional.html#non-linear-activation-functions. Try them out! 3) PyTorch MLP layers are called "Linear" despite being affine transformations. Let's see what happens if they are literally linear: the bias term can be removed with `bias=False`, e.g. `nn.Linear(32,32, bias = False)`. What happens to the decision boundary? As you do this, reflect on the exercise we did earlier where we constructed MLPs to exactly match piecewise linear functions, and the role the bias played there. 4) It should be straightforward to change the model to work on MNIST.

Chapter 3.3: Convolutional Neural Networks (CNNs)

MLP's are already universal function approximators. But we can do better!

Often, and especially in physics, our function has some structure we can exploit (such as symmetries, or locality). Perhaps we can use this symmetry to constrain the weights of the MLP, or even better (and equivalently), design a new MLP-like-model that has this symmetry built in and is a universal function approximator with respect to this symmetry.

Image classification has an approximate translational symmetry: If an image of a handwritten "5" is shifted to the right, it is still a "5". We can exploit this symmetry by using a **Convolutional Neural Network (CNN)**, which is a type of neural network that is specifically designed for image classification tasks that has this translational symmetry built in. A CNN can achieve significantly better performance than an MLP with the same number or far fewer parameters.

A 2D translation on our 784-dimensional input space is a violent operation, and a generic MLP will not be translationlly invariant (or covariant). One can attempt to solve for a special class of W's that are covariant (related to Toeplitz matrices), but this will end up being related to convolutions anyways.

Instead, if we represent an image as a distribution $I(x)$, where $I(x)$ is the pixel value at position x (and there can be multiple channels, e.g. RGB labelled by $I^a(x)$), then the following operation (**convolution**) is equivalent to a translation-invariant operation:

$$(I * K)^b(x) = \int dy \sum_a I^a(y) K^b(x-y)$$

where $K^b(x)$ is a kernel (or filter) that is applied to the image (there can be several kernels, labeled by b and summed over a). The convolution operation is equivalent to sliding the kernel over the image and computing the dot product at each position.

Then, we can construct a universal function approximator for translation-invariant functions:

$$F(I) = \Psi(\int dx [\text{Equivariant operations on I}])$$

where Ψ is any function that is a universal function approximator (e.g. an MLP). This is the basic recipe for a CNN. We use (discrete) convolutions to extract features from an image in an equivariant way aggregate them (represented by the integral, but it could also any other invariant aggregation like max-pooling), then feed them into an MLP. In practice, we will interleave pooling with the convolutions (since it also helps introduce nonlinearity) and use max-pooling rather than integration (mean-pooling) for more nonlinearity.

Images are not exactly translation invariant, but they are approximately so. In particular, they are discrete, and they have boundaries. So we will use a discrete convolution, and use pading on boundaries. We will also assume that our convolutions are local (the support of $K(x-y)$ is dominated by $x \sim y$), so that we can write our kernels as small 3×3 or (5×5) matrices.

We will also make use of the `nn.Module` class. This is the base class for all neural network modules in PyTorch. It is one level of abstraction above the `nn.Sequential` class, and allows us to define the model with more contrl (e.g. we can define the parameters of the model directory (the layers in most models), and we can define how those paramrameters are used to define the function (the `forward` method).)

We will also see our first example of a nontrivial layer beyond just the linear layer. The `nn.Conv2d` layer is a convolutional layer that applies a 2D convolution to the input. It takes as input the number of input channels, the number of output channels, and the kernel size (the size of the filter). It also has a stride and padding parameter, which control how the convolution is applied to the input.

```
# Constructing a PyTorch CNN


class SimpleCNN(nn.Module):
```

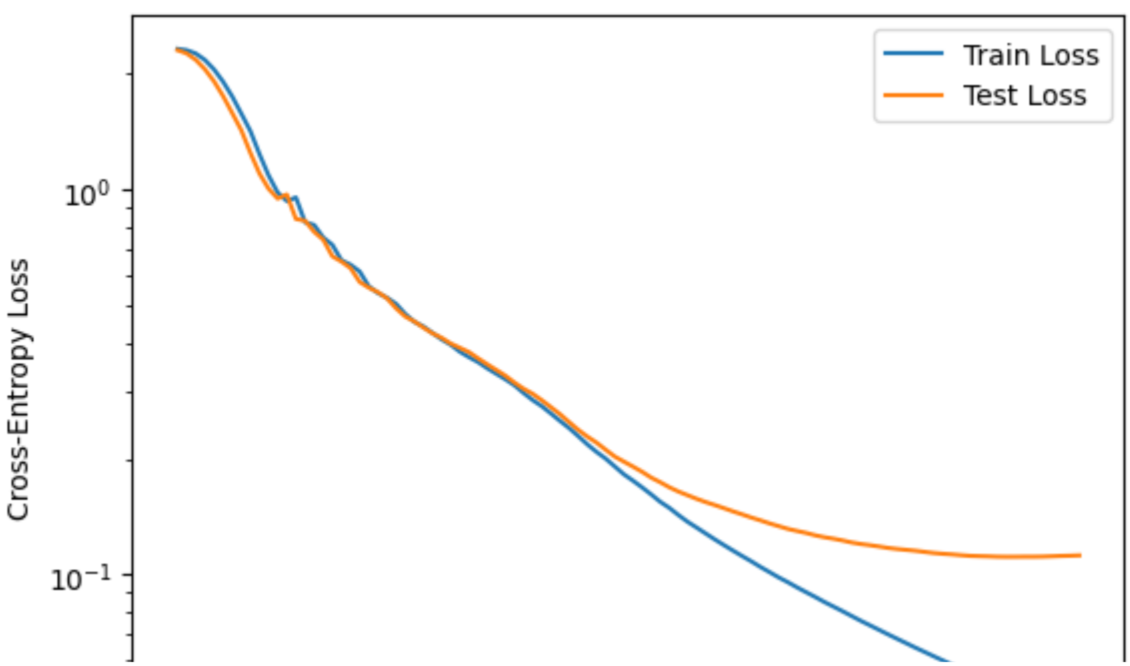
```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(
            1, 16, kernel_size=3, padding=1
        )  # Input channels = 1 for grayscale images
        self.conv2 = nn.Conv2d(16, 16, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(
            16 * 7 * 7, 16
        )  # Assuming input images are 28x28, and we aggregate down to 7x7.
        self.fc2 = nn.Linear(16, num_classes)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(
            x, kernel_size=2
        )  # aggregation: Take every 2x2 block and take the maximum value
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(
            x, kernel_size=2
        )  # After 2 aggregations, we go from 28x28 to 14x14 to 7x7.

        # We havent aggregated all the way down to a single value, so we arent perfectl
        # But we are invariant to small translations, which is good enough!

        x = x.view(x.size(0), -1)  # Flatten the tensor:

        # MLP part:
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x


# Initialize the CNN model
model_cnn = SimpleCNN(num_classes=10).to(
    device
)  # Move the model to the GPU if available

# Define the optimizer and loss function (IDENTICAL TO BEFORE)
opt_cnn = torch.optim.Adam(model_cnn.parameters(), lr=1e-2)  # Adam optimizer
loss_fn_cnn = nn.CrossEntropyLoss()  # Cross-entropy loss for multi-class classificatio

# Data. We can treat the images as genuine 28x28x1 (1 = no color) images, rather than f
train_fraction = 0.8
train_size = (
    int(train_fraction * len(X_mnist)) // 5
)  # Use a smaller training set for faster training, just for tutorial's sake
indices = torch.randperm(len(X_mnist))  # Shuffle the dataset
X_mnist_torch = (
    torch.tensor(X_mnist, dtype=torch.float32, device=device)
    .unsqueeze(1)
    .reshape(-1, 1, 28, 28)
)  # Reshape to (N, C, H, W)
y_mnist_torch = torch.tensor(
```

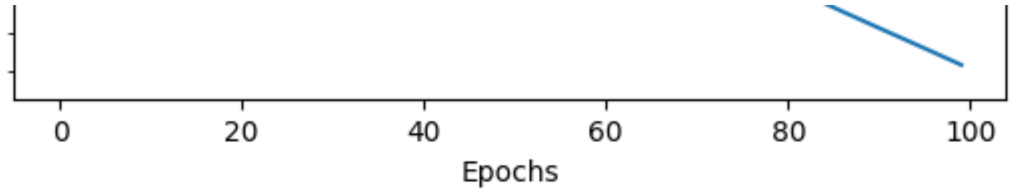
```
y_mnist_torch = torch.tensor(
    y_mnist, dtype=torch.long, device=device
)  # Long tensor for class labels
X_train_mnist_torch, y_train_mnist_torch = (
    X_mnist_torch[indices[:train_size]],
    y_mnist_torch[indices[:train_size]],
)
X_test_mnist_torch, y_test_mnist_torch = (
    X_mnist_torch[indices[train_size:]],
    y_mnist_torch[indices[train_size:]],
)


# Train the CNN model
epochs = 100  # Far fewer epochs needed!
train_losses_cnn = []
test_losses_cnn = []

for epoch in range(epochs):
    opt_cnn.zero_grad()  # Zero out the gradients before the backward pass
    outputs = model_cnn(X_train_mnist_torch)  # Forward pass
    loss = loss_fn_cnn(outputs, y_train_mnist_torch)  # Compute the loss
    loss.backward()  # Backward pass to compute gradients
    opt_cnn.step()  # Update the model parameters
    train_losses_cnn.append(loss.item())  # Save the training loss

    # Evaluate on the test set
    with torch.no_grad():  # No need to compute gradients for evaluation
        test_outputs = model_cnn(X_test_mnist_torch)
        test_loss = loss_fn_cnn(test_outputs, y_test_mnist_torch)
        test_losses_cnn.append(test_loss.item())  # Save the test loss

        # Compute accuracy
        _, predicted = torch.max(test_outputs, 1)
        accuracy = (predicted == y_test_mnist_torch).float().mean().item()

    if epoch % 1 == 0:  # Print every epoch
        print(
            f"Epoch {epoch}, Train Loss: {loss.item():.4f}, Test Loss: {test_loss.item(
        )

# Plot the training and test losses
fig, ax = plt.subplots()
plt.plot(train_losses_cnn, label="Train Loss")
plt.plot(test_losses_cnn, label="Test Loss")
plt.yscale("log")
plt.xlabel("Epochs")
plt.ylabel("Cross-Entropy Loss")
plt.legend()
plt.show()
```

```
Epoch 0, Train Loss: 2.3116, Test Loss: 2.2950, Accuracy: 0.0985
Epoch 1, Train Loss: 2.2944, Test Loss: 2.2505, Accuracy: 0.2753
Epoch 2, Train Loss: 2.2496, Test Loss: 2.1662, Accuracy: 0.2508
```

```
Epoch 3, Train Loss: 2.1653, Test Loss: 2.0483, Accuracy: 0.2465
Epoch 4, Train Loss: 2.0466, Test Loss: 1.9061, Accuracy: 0.3422
Epoch 5, Train Loss: 1.9015, Test Loss: 1.7448, Accuracy: 0.4024
Epoch 6, Train Loss: 1.7416, Test Loss: 1.5767, Accuracy: 0.4728
Epoch 7, Train Loss: 1.5718, Test Loss: 1.4229, Accuracy: 0.5270
Epoch 8, Train Loss: 1.4155, Test Loss: 1.2429, Accuracy: 0.5959
Epoch 9, Train Loss: 1.2332, Test Loss: 1.0984, Accuracy: 0.6404
Epoch 10, Train Loss: 1.0858, Test Loss: 1.0030, Accuracy: 0.6648
Epoch 11, Train Loss: 0.9801, Test Loss: 0.9458, Accuracy: 0.6790
Epoch 12, Train Loss: 0.9320, Test Loss: 0.9693, Accuracy: 0.6776
Epoch 13, Train Loss: 0.9520, Test Loss: 0.8373, Accuracy: 0.7312
Epoch 14, Train Loss: 0.8213, Test Loss: 0.8284, Accuracy: 0.7350
Epoch 15, Train Loss: 0.8077, Test Loss: 0.7755, Accuracy: 0.7524
Epoch 16, Train Loss: 0.7494, Test Loss: 0.7398, Accuracy: 0.7706
Epoch 17, Train Loss: 0.7175, Test Loss: 0.6700, Accuracy: 0.7891
Epoch 18, Train Loss: 0.6541, Test Loss: 0.6493, Accuracy: 0.7978
Epoch 19, Train Loss: 0.6369, Test Loss: 0.6242, Accuracy: 0.8095
Epoch 20, Train Loss: 0.6118, Test Loss: 0.5748, Accuracy: 0.8272
Epoch 21, Train Loss: 0.5600, Test Loss: 0.5556, Accuracy: 0.8316
Epoch 22, Train Loss: 0.5382, Test Loss: 0.5407, Accuracy: 0.8388
Epoch 23, Train Loss: 0.5232, Test Loss: 0.5213, Accuracy: 0.8472
Epoch 24, Train Loss: 0.5042, Test Loss: 0.4910, Accuracy: 0.8589
Epoch 25, Train Loss: 0.4748, Test Loss: 0.4677, Accuracy: 0.8655
Epoch 26, Train Loss: 0.4531, Test Loss: 0.4541, Accuracy: 0.8647
Epoch 27, Train Loss: 0.4405, Test Loss: 0.4376, Accuracy: 0.8678
Epoch 28, Train Loss: 0.4237, Test Loss: 0.4229, Accuracy: 0.8734
Epoch 29, Train Loss: 0.4079, Test Loss: 0.4117, Accuracy: 0.8772
Epoch 30, Train Loss: 0.3950, Test Loss: 0.3978, Accuracy: 0.8820
Epoch 31, Train Loss: 0.3788, Test Loss: 0.3879, Accuracy: 0.8859
Epoch 32, Train Loss: 0.3668, Test Loss: 0.3780, Accuracy: 0.8888
Epoch 33, Train Loss: 0.3561, Test Loss: 0.3642, Accuracy: 0.8933
Epoch 34, Train Loss: 0.3433, Test Loss: 0.3515, Accuracy: 0.8968
Epoch 35, Train Loss: 0.3321, Test Loss: 0.3398, Accuracy: 0.8992
Epoch 36, Train Loss: 0.3214, Test Loss: 0.3283, Accuracy: 0.9019
Epoch 37, Train Loss: 0.3100, Test Loss: 0.3150, Accuracy: 0.9060
Epoch 38, Train Loss: 0.2961, Test Loss: 0.3037, Accuracy: 0.9096
Epoch 39, Train Loss: 0.2838, Test Loss: 0.2946, Accuracy: 0.9123
Epoch 40, Train Loss: 0.2736, Test Loss: 0.2835, Accuracy: 0.9160
Epoch 41, Train Loss: 0.2619, Test Loss: 0.2720, Accuracy: 0.9191
Epoch 42, Train Loss: 0.2506, Test Loss: 0.2608, Accuracy: 0.9219
Epoch 43, Train Loss: 0.2401, Test Loss: 0.2488, Accuracy: 0.9254
Epoch 44, Train Loss: 0.2288, Test Loss: 0.2375, Accuracy: 0.9290
Epoch 45, Train Loss: 0.2178, Test Loss: 0.2282, Accuracy: 0.9325
Epoch 46, Train Loss: 0.2083, Test Loss: 0.2202, Accuracy: 0.9352
Epoch 47, Train Loss: 0.2002, Test Loss: 0.2111, Accuracy: 0.9376
Epoch 48, Train Loss: 0.1910, Test Loss: 0.2026, Accuracy: 0.9399
Epoch 49, Train Loss: 0.1822, Test Loss: 0.1966, Accuracy: 0.9415
Epoch 50, Train Loss: 0.1755, Test Loss: 0.1908, Accuracy: 0.9432
Epoch 51, Train Loss: 0.1686, Test Loss: 0.1850, Accuracy: 0.9449
Epoch 52, Train Loss: 0.1616, Test Loss: 0.1787, Accuracy: 0.9462
Epoch 53, Train Loss: 0.1546, Test Loss: 0.1738, Accuracy: 0.9476
Epoch 54, Train Loss: 0.1491, Test Loss: 0.1686, Accuracy: 0.9491
Epoch 55, Train Loss: 0.1428, Test Loss: 0.1644, Accuracy: 0.9506
Epoch 56, Train Loss: 0.1372, Test Loss: 0.1608, Accuracy: 0.9515
Epoch 57, Train Loss: 0.1325, Test Loss: 0.1574, Accuracy: 0.9530
Epoch 58, Train Loss: 0.1279, Test Loss: 0.1543, Accuracy: 0.9539
Epoch 59, Train Loss: 0.1235, Test Loss: 0.1516, Accuracy: 0.9545
Epoch 60, Train Loss: 0.1194, Test Loss: 0.1488, Accuracy: 0.9553
```

```
Epoch 61, Train Loss: 0.1155, Test Loss: 0.1460, Accuracy: 0.9562
Epoch 62, Train Loss: 0.1118, Test Loss: 0.1434, Accuracy: 0.9571
Epoch 63, Train Loss: 0.1084, Test Loss: 0.1408, Accuracy: 0.9580
Epoch 64, Train Loss: 0.1048, Test Loss: 0.1384, Accuracy: 0.9588
Epoch 65, Train Loss: 0.1016, Test Loss: 0.1359, Accuracy: 0.9595
Epoch 66, Train Loss: 0.0985, Test Loss: 0.1337, Accuracy: 0.9601
Epoch 67, Train Loss: 0.0957, Test Loss: 0.1316, Accuracy: 0.9606
Epoch 68, Train Loss: 0.0928, Test Loss: 0.1299, Accuracy: 0.9611
Epoch 69, Train Loss: 0.0902, Test Loss: 0.1284, Accuracy: 0.9615
Epoch 70, Train Loss: 0.0876, Test Loss: 0.1266, Accuracy: 0.9623
Epoch 71, Train Loss: 0.0851, Test Loss: 0.1252, Accuracy: 0.9626
Epoch 72, Train Loss: 0.0827, Test Loss: 0.1241, Accuracy: 0.9631
Epoch 73, Train Loss: 0.0805, Test Loss: 0.1228, Accuracy: 0.9632
Epoch 74, Train Loss: 0.0782, Test Loss: 0.1213, Accuracy: 0.9634
Epoch 75, Train Loss: 0.0760, Test Loss: 0.1202, Accuracy: 0.9637
Epoch 76, Train Loss: 0.0740, Test Loss: 0.1194, Accuracy: 0.9642
Epoch 77, Train Loss: 0.0720, Test Loss: 0.1184, Accuracy: 0.9643
Epoch 78, Train Loss: 0.0701, Test Loss: 0.1174, Accuracy: 0.9647
Epoch 79, Train Loss: 0.0682, Test Loss: 0.1167, Accuracy: 0.9650
Epoch 80, Train Loss: 0.0665, Test Loss: 0.1161, Accuracy: 0.9651
Epoch 81, Train Loss: 0.0647, Test Loss: 0.1155, Accuracy: 0.9651
Epoch 82, Train Loss: 0.0631, Test Loss: 0.1147, Accuracy: 0.9658
Epoch 83, Train Loss: 0.0615, Test Loss: 0.1139, Accuracy: 0.9660
Epoch 84, Train Loss: 0.0600, Test Loss: 0.1134, Accuracy: 0.9662
Epoch 85, Train Loss: 0.0585, Test Loss: 0.1130, Accuracy: 0.9664
Epoch 86, Train Loss: 0.0571, Test Loss: 0.1124, Accuracy: 0.9665
Epoch 87, Train Loss: 0.0557, Test Loss: 0.1120, Accuracy: 0.9665
Epoch 88, Train Loss: 0.0543, Test Loss: 0.1118, Accuracy: 0.9667
Epoch 89, Train Loss: 0.0530, Test Loss: 0.1117, Accuracy: 0.9667
Epoch 90, Train Loss: 0.0517, Test Loss: 0.1114, Accuracy: 0.9668
Epoch 91, Train Loss: 0.0505, Test Loss: 0.1114, Accuracy: 0.9667
Epoch 92, Train Loss: 0.0493, Test Loss: 0.1114, Accuracy: 0.9668
Epoch 93, Train Loss: 0.0481, Test Loss: 0.1114, Accuracy: 0.9669
Epoch 94, Train Loss: 0.0470, Test Loss: 0.1114, Accuracy: 0.9673
Epoch 95, Train Loss: 0.0458, Test Loss: 0.1115, Accuracy: 0.9675
Epoch 96, Train Loss: 0.0447, Test Loss: 0.1117, Accuracy: 0.9677
Epoch 97, Train Loss: 0.0436, Test Loss: 0.1120, Accuracy: 0.9676
Epoch 98, Train Loss: 0.0426, Test Loss: 0.1121, Accuracy: 0.9678
Epoch 99, Train Loss: 0.0415, Test Loss: 0.1123, Accuracy: 0.9678
```





Chapter 3.4 (BONUS): Permutation Invariance and Equivariance with Deep Sets and Transformers

We saw above how we can use convolutions to exploit translational symmetry. In this section, we will see how we can exploit permutation symmetry of inputs. Here, we are interested in functions of SETS to real numbers, rather than functions of vectors to real numbers. Sets are invariant to the order of the elements, meaning that the function should produce the same output regardless of the order of the points in the set. One immediate upside of this is that our model can allow for an arbitrary input size, since it's still a set regardless of how many points are in it!

In physics, we often deal with sets of particles or measurements, where the particle labeling is completely arbitrary, so this symmetry is important. We also often do not have a fixed number of particles, so we want our model to be able to handle an arbitrary number of particles.

Two models that are designed to exploit permutation symmetry are **Deep Sets** and **Transformers**.

A Deep Set is a type of network that is a universal function approximator for functions of sets. It is defined as:

$$f(S) = \Psi\left(\sum_{x\in S}\phi(x)\right)$$

where ϕ is a function that maps each element of the set to a vector, and Ψ is a function that aggregates the vectors into a single vector. Both are MLPs! It is manifestly invariant to the order of the points in the set. Note that even though ϕ only acts on a single element at a time, inter-point correlations are still captured by the aggregation function Ψ (though not necessarily efficiently).

Transformers are a more general class of networks that include Deep Sets as a special case. Transformers are extremely common and have exploded in popularity in the last few years, especially in the context of natural language processing (NLP). Most of the most powerful models in high energy physics (such as Particle Transformer), as well as models outside physics (such as ChatGPT) are Transformers.

Transformers are based on permutation equivariant layers:

$$f(x_i) = \sigma(C_{ij}x_j + \phi(x_i, x_j)D_{jk}x_k)$$

where C and D are matrices that are shared across all points in the set, and ϕ is some symmetric kernel. This is the simplest non-linear layer one can construct that is permutation invariant. Typically, $\phi = \mathrm{softmax}(\langle Qx_i, Kx_j\rangle)$.

A lot of literature about transformers is based on natural language processing, so it's worth getting used to. Just for terminology's sake, it is common to call Q_i the "query vector", K_j the "key vector", and $V_j = D_{jk}x_k$ the "value vector", but these are just names and don't have any special meaning outside the very specific context of NLP. Emotionally, Q is supposed to represent the "question" we are asking about the input x_i, K is supposed to represent the "context" of the input x_j, and V is supposed to represent the "value" of the input x_k. This construction is called the **self-attention** layer, because the kernal ϕ is a function from 0 to 1 that tells the model at x_i how much to "pay attention" to x_j, but these are just non-rigorous words invented before the math was understood and don't get too caught up in them. The elements of the sets are called "tokens", because words are often encoded as tokens. Note that since self-attention is permutation invariant, the order of words in a sentence is usually encoded into the token itself to break the symmetry, or the kernel is forced to be 0 if the words are out of order.

Both the deep-sets and the transformer were written as a scalar output. One can easily extend this to a vector of outputs by appropriately appending extra indcies to everything.

TOY MODEL:

We will construct the following problem: Every point is a list of 10 random vectors. If the sum of all pairwise dot-products of the vectors in the set is positive, then the output is 1, otherwise it is 0. E.g. we are testing if the set of vectors are roughly pointing in the same direction or not. This problem is permnutation invariant (so Deep Sets or Transformers will be useful), but explicitly involves pairwise nonlinear correlations (so a small Deep Sets will struggle).

```
NUM_SAMPLES = 25000
SET_SIZE = 10  # number of vectors in each set
FEATURES = 16  # dimension of vectors

# Generate random sets
X_full = torch.randn(NUM_SAMPLES, SET_SIZE, FEATURES, device=device)

# Label is 1 if the sum of all pairwise dot products is positive


def upper_triangle_dot_sum(x_set):  # x_set: (n, d)
    total = 0.0
    for i in range(SET_SIZE):
        for j in range(i + 1, SET_SIZE):
            total += (x_set[i] * x_set[j]).sum()  # simple dot-product
    return total


y_full = torch.tensor(
    [upper_triangle_dot_sum(x) > 0 for x in X_full], dtype=torch.float32, device=device
).unsqueeze(
    1
```

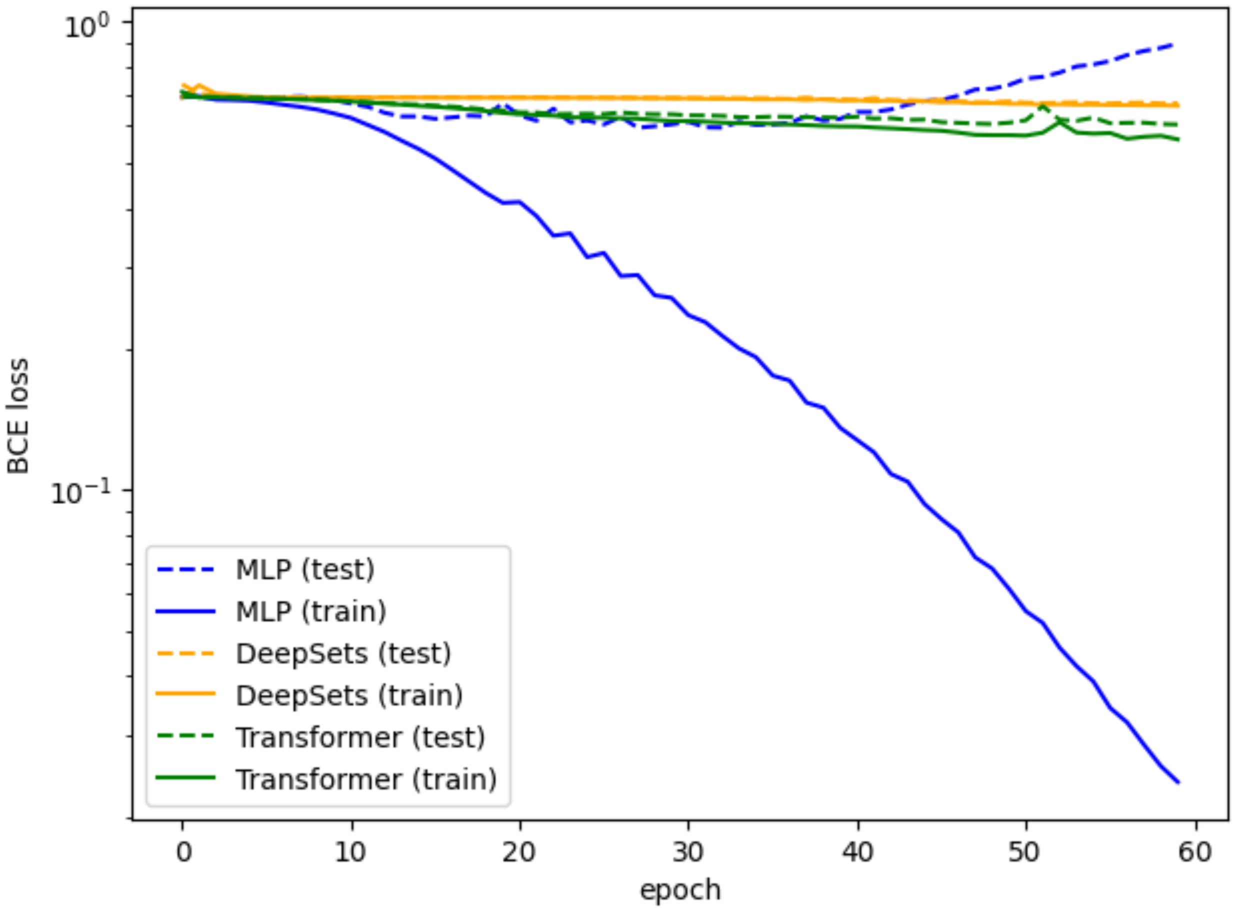
```
    _
)  # shape (N, 1)

# Train / test split
train_size = 20000
X_train, y_train = X_full[:train_size], y_full[:train_size]
X_test, y_test = X_full[train_size:], y_full[train_size:]


# ##### DEFINE MODELS #####


class MLP_torch(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Flatten(),  # (B, n*d)
            nn.Linear(SET_SIZE * FEATURES, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 1),
        )

    def forward(self, x):
        return self.layers(x)


class DeepSets(nn.Module):
    def __init__(self):
        super().__init__()
        self.phi = nn.Sequential(
            nn.Linear(FEATURES, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
        )
        self.rho = nn.Sequential(
            nn.Linear(64, 64), nn.ReLU(), nn.Linear(64, 64), nn.ReLU(), nn.Linear(64, 1
        )

    def forward(self, x):  # x: (B, n, d)
        h = self.phi(x)  # (B, n, 64)
        s = h.sum(dim=1)  # permutation-invariant
        return self.rho(s)


class Transformer(nn.Module):
    def __init__(self):
        super().__init__()
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=FEATURES, nhead=4, dim_feedforward=64, batch_first=True
```

```
            d_model=FEATURES, nhead=4, dim_feedforward=64, batch_first=True
        )

        # Encoder layer: contains a stack of self-attention and feedforward layers as d
        # Can code this yourself if you really want, but PyTorch has a built-in impleme
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=2)
        self.head = nn.Sequential(nn.Linear(FEATURES, 64), nn.ReLU(), nn.Linear(64, 1))

    def forward(self, x):
        h = self.encoder(x)  # equivariant map
        g = h.mean(dim=1)  # invariant aggregation
        return self.head(g)


models = {
    "MLP": MLP_torch().to(device),
    "DeepSets": DeepSets().to(device),
    "Transformer": Transformer().to(device),
}

# ##### TRAINING #####


def train_model(model, X_tr, y_tr, X_te, y_te, epochs=60, lr=1e-2, print_every=5):
    optimiser = torch.optim.Adam(model.parameters(), lr=lr)
    train_losses = []
    test_losses = []
    bce_loss = nn.BCEWithLogitsLoss()

    for epoch in range(epochs):
        # forward / backward on the entire training set
        model.train()
        optimiser.zero_grad()
        logits = model(X_tr)
        loss = bce_loss(logits, y_tr)
        loss.backward()
        optimiser.step()
        train_losses.append(loss.item())

        # Evaluate on test set
        model.eval()
        with torch.no_grad():
            test_logits = model(X_te)
            test_loss = bce_loss(test_logits, y_te)
            test_losses.append(test_loss.item())

            preds = (torch.sigmoid(test_logits) > 0.5).float()
            accuracy = (preds == y_te).float().mean().item()

        if epoch % print_every == 0:
            print(
                f"epoch {epoch:02d}  "
                f"train-loss {loss.item():.4f}  "
                f"test-loss {test_loss.item():.4f}  "
```

```
            f"acc {accuracy:.3f}"
        )

    return train_losses, test_losses


# ##### RESULTS #####
plt.figure()

colors = {"MLP": "blue", "DeepSets": "orange", "Transformer": "green"}
for name, net in models.items():
    print(f"\n{name}")
    tr_loss, te_loss = train_model(net, X_train, y_train, X_test, y_test, epochs=60)
    plt.plot(te_loss, label=f"{name} (test)", ls="--", color=colors[name])
    plt.plot(tr_loss, label=f"{name} (train)", color=colors[name])

plt.yscale("log")
plt.xlabel("epoch")
plt.ylabel("BCE loss")
plt.legend()
plt.tight_layout()
plt.show()
```

```
MLP
epoch 00  train-loss 0.6905  test-loss 0.7003  acc 0.539
epoch 05  train-loss 0.6729  test-loss 0.6884  acc 0.550
epoch 10  train-loss 0.6238  test-loss 0.6695  acc 0.589
epoch 15  train-loss 0.5109  test-loss 0.6211  acc 0.660
epoch 20  train-loss 0.4124  test-loss 0.6312  acc 0.685
epoch 25  train-loss 0.3210  test-loss 0.6041  acc 0.712
epoch 30  train-loss 0.2364  test-loss 0.6121  acc 0.729
epoch 35  train-loss 0.1756  test-loss 0.6038  acc 0.749
epoch 40  train-loss 0.1279  test-loss 0.6422  acc 0.754
epoch 45  train-loss 0.0867  test-loss 0.6811  acc 0.759
epoch 50  train-loss 0.0551  test-loss 0.7572  acc 0.766
epoch 55  train-loss 0.0342  test-loss 0.8254  acc 0.769

DeepSets
epoch 00  train-loss 0.6900  test-loss 0.7362  acc 0.539
epoch 05  train-loss 0.6907  test-loss 0.6902  acc 0.539
epoch 10  train-loss 0.6896  test-loss 0.6904  acc 0.539
epoch 15  train-loss 0.6892  test-loss 0.6901  acc 0.539
epoch 20  train-loss 0.6890  test-loss 0.6900  acc 0.539
epoch 25  train-loss 0.6883  test-loss 0.6893  acc 0.539
epoch 30  train-loss 0.6868  test-loss 0.6880  acc 0.561
epoch 35  train-loss 0.6836  test-loss 0.6869  acc 0.559
epoch 40  train-loss 0.6808  test-loss 0.6812  acc 0.565
epoch 45  train-loss 0.6733  test-loss 0.6825  acc 0.554
epoch 50  train-loss 0.6685  test-loss 0.6725  acc 0.588
epoch 55  train-loss 0.6648  test-loss 0.6699  acc 0.587

Transformer
epoch 00  train-loss 0.7090  test-loss 0.6920  acc 0.541
epoch 05  train-loss 0.6863  test-loss 0.6869  acc 0.544
epoch 10  train-loss 0.6761  test-loss 0.6769  acc 0.589
```

```
epoch 15  train-loss 0.6592  test-loss 0.6624  acc 0.599
epoch 20  train-loss 0.6352  test-loss 0.6410  acc 0.645
epoch 25  train-loss 0.6238  test-loss 0.6356  acc 0.646
epoch 30  train-loss 0.6143  test-loss 0.6316  acc 0.648
epoch 35  train-loss 0.6061  test-loss 0.6278  acc 0.651
epoch 40  train-loss 0.5980  test-loss 0.6274  acc 0.652
epoch 45  train-loss 0.5856  test-loss 0.6107  acc 0.668
epoch 50  train-loss 0.5717  test-loss 0.6164  acc 0.674
epoch 55  train-loss 0.5790  test-loss 0.6076  acc 0.669
```



Exercise: Linear Permutation Invariance

Can you design a problem for which DeepSets will outperform a transformer? That is, a problem for which inter-particle correlations are expected to be less important, so that the problem can be more easily written as a sum of single-particle functions?

Chapter 4: Tensorflow

Tensorflow (and the associated API, Keras) is another popular ML library developed by Google. It is similar to PyTorch in many ways. It used to be the most popular ML library years ago before being overtaken by PyTorch, but it is still widely used in industry and research. It is typically more high-level than JAX, and has a wider range of prebuilt modules and utilities for common ML tasks. It is also older and more widely supported with documentation and tutorials online.

Keras is a high-level API for Tensorflow that allows us to define models in an intuitive way. It is similar to the `nn.Sequential` class in PyTorch, but with more features and flexibility. Keras allows us to define models as a sequence of layers, where each layer is applied in order. It also has a wide range of prebuilt layers and utilities for common ML tasks.

We will not go into detail about Tensorflow and Keras, since they are similar to PyTorch and JAX, but we will provide a brief example of how to define an MLP in Keras, so that you are familiar with the syntax and can use it if you prefer.

Tensorflow has even more pre-built modules that are abstracted away from the user than PyTorch, so it is even easier to define models. You do not even have to write your own traning loop. One difference is that Tensorflow uses a "static graph" approach, meaning that the computation graph is defined before the model is run. The model must be specified and then "compiled", not entirely unlike JAX's JIT compilation. This means that the model is optimized for performance before it is run, which can lead to better performance in some cases. However, it also means that the model is less flexible and harder to debug, since you cannot change the model on the fly like you can in PyTorch or JAX.

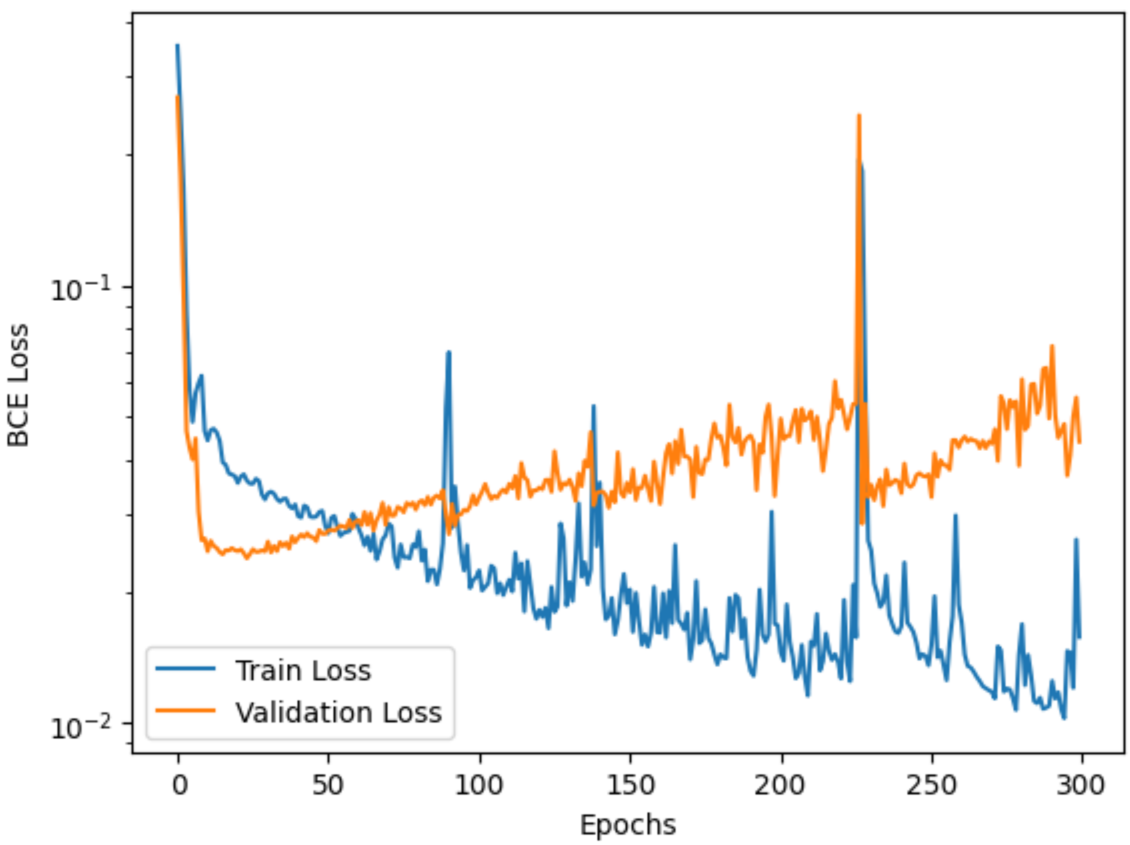
A historical note: Tensorflow 1.0 is extremely different from Tensorflow 2.0 and Keras. We will only be using Tensorflow 2.0 and Keras, which is much more user-friendly and similar to PyTorch. It is rare to see Tensorflow 1.0 code these days, especially in physics, but it is worth being aware that it exists if you find yourself being confused by some old code.

Chapter 4.1: Defining an MLP and a CNN in Keras

```
import tensorflow as tf

tf.random.set_seed(0)


# Something nice about tensorflow: You only have to specify the output dimension of a la
mlp = tf.keras.Sequential(
    [
        tf.keras.layers.Dense(32, activation="relu", input_shape=(2,)),
        tf.keras.layers.Dense(32, activation="relu"),
        tf.keras.layers.Dense(1),  # logits
    ]
)

mlp.compile(
    optimizer=tf.keras.optimizers.Adam(1e-2),
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    metrics=[
        "accuracy"
    ],  # We can tell tensorflow what metrics we want to track on top of the loss!
```

```
)


# Tensorflow automatically has a training loop. We dont even need to split the dataset!
history = mlp.fit(X_moons, y_moons, epochs=300, verbose=0, validation_split=0.2)

# Training gives us a history object with the training and validation losses and accurac
# Very convenient!

# Plot the training and validation losses
plt.plot(history.history["loss"], label="Train Loss")
plt.plot(history.history["val_loss"], label="Validation Loss")
plt.yscale("log")
plt.xlabel("Epochs")
plt.ylabel("BCE Loss")
plt.legend()

print("final val-acc:", history.history["val_accuracy"][-1])
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarn
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
final val-acc: 0.9853658676147461
```



CNNS and Callbacks

A very cute feature of Keras/TF are "Callbacks", which are functions you can execute during the training loop. In JAX or PyTorch, you manually write training loops anyways and so can decorate them however you like, but Keras has a specialized interface for this.

For example, you can define a callback that prints the training loss every 10 epochs, or a callback that saves the model every 100 epochs. This is very useful for debugging and monitoring the training process.

The most useful callback is the `EarlyStopping` callback, which stops the training process if the validation loss does not improve for a certain number of epochs. This is useful to prevent overfitting and save time during training. This can also be done for PyTorch and JAX, of course, but we are introducing it here since it is especially convenient.

Many callbacks are built-in, but for the sake of this tutorial, we will also define a custom callback to print the mean and spread of the weights of each layer. This is useful to monitor the training process and see if the weights are exploding or vanishing.

```
# Prepare data
X_mnist_np = X_mnist.astype("float32").reshape(-1, 28, 28, 1) / 255.0
y_mnist_np = y_mnist.astype("int32")  # integer labels (0–9)

idx = np.random.permutation(len(X_mnist_np))
train_size = int(0.8 * len(idx))

X_train = X_mnist_np[idx[:train_size]]
y_train = y_mnist_np[idx[:train_size]]
X_test = X_mnist_np[idx[train_size:]]
y_test = y_mnist_np[idx[train_size:]]

# Basic CNN model in Keras. Basicly the same as in PyTorch! Slightly more convient sinc
cnn = tf.keras.Sequential(
    [
        tf.keras.layers.Conv2D(
            16, 3, padding="same", activation="relu", input_shape=(28, 28, 1)
        ),
        tf.keras.layers.MaxPool2D(),
        tf.keras.layers.Conv2D(16, 3, padding="same", activation="relu"),
        tf.keras.layers.MaxPool2D(),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(16, activation="relu"),
        tf.keras.layers.Dense(10),  # logits
    ]
)


cnn.compile(
    optimizer=tf.keras.optimizers.Adam(1e-3),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=["accuracy"],
)

# ########## CALLBACKS ##########

# Pre-built callback: Stop training if the validation loss does not improve for 5 epoch
```

```python
# Pre-built callback: Stop training if the validation loss does not improve for 5 epoch
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor="val_loss", patience=5, restore_best_weights=True, verbose=1
)


# Define a custom callback
class LayerWeights(tf.keras.callbacks.Callback):
    # Override the on_epoch_end method so the function executed at the end of each epoc
    def on_epoch_end(self, epoch, logs=None):
        # For each layer, print the mean and standard deviation of the weights
        for layer in self.model.layers:
            if hasattr(layer, "kernel"):
                weights = layer.kernel.numpy()
                print(
                    f"Layer {layer.name} - Mean: {np.mean(weights):.4f}, Std: {np.std(w
                )


hist = cnn.fit(
    X_train,
    y_train,
    epochs=50,
    batch_size=256,
    validation_split=0.1,
    callbacks=[early_stop, LayerWeights()],
    verbose=1,
)

# ##### RESULTS #####
test_loss, test_acc = cnn.evaluate(X_test, y_test, verbose=0)
print(f"MNIST  test-accuracy = {test_acc:.3f}")

plt.plot(hist.history["loss"], label="train")
plt.plot(hist.history["val_loss"], label="val")
plt.yscale("log")
plt.xlabel("epoch")
plt.ylabel("cross-entropy")
plt.legend()
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - accuracy: 0.1179 - loss: 2.2978Layer co
Layer conv2d_1 - Mean: 0.0117, Std: 0.1210
Layer dense_3 - Mean: 0.0010, Std: 0.0639
Layer dense_4 - Mean: 0.0251, Std: 0.3139
197/197 ━━━━━━━━━━━━━━━━━━━━ 23s 105ms/step - accuracy: 0.1181 - loss: 2.2976 - val
Epoch 2/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - accuracy: 0.4495 - loss: 1.6887Layer co
Layer conv2d_1 - Mean: 0.0316, Std: 0.2033
Layer dense_3 - Mean: 0.0069, Std: 0.1082
Layer dense_4 - Mean: 0.0133, Std: 0.3981
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 102ms/step - accuracy: 0.4500 - loss: 1.6870 - val
Epoch 3/50
```

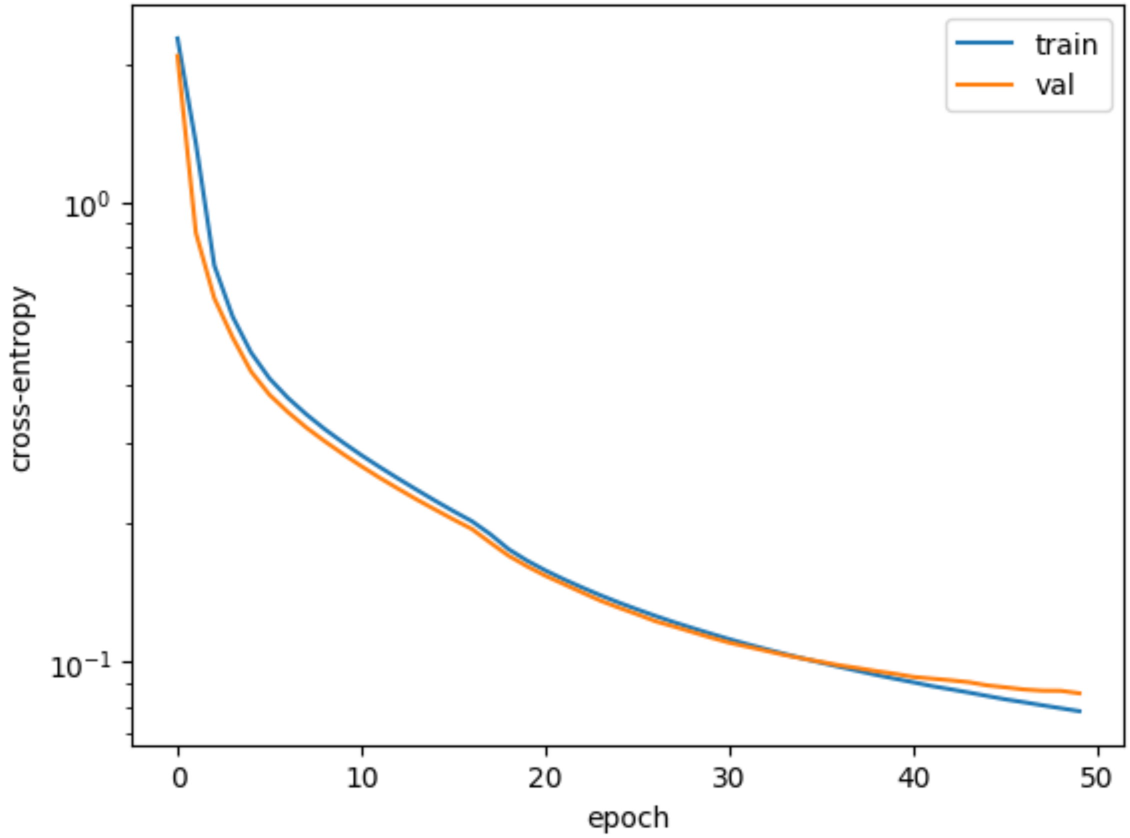
```
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 93ms/step - accuracy: 0.7386 - loss: 0.7930Layer co
Layer conv2d_1 - Mean: 0.0368, Std: 0.2374
Layer dense_3 - Mean: 0.0079, Std: 0.1192
Layer dense_4 - Mean: 0.0114, Std: 0.4189
197/197 ━━━━━━━━━━━━━━━━━━━━ 19s 96ms/step - accuracy: 0.7387 - loss: 0.7927 - val_
Epoch 4/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 99ms/step - accuracy: 0.8178 - loss: 0.5970Layer co
Layer conv2d_1 - Mean: 0.0392, Std: 0.2591
Layer dense_3 - Mean: 0.0085, Std: 0.1255
Layer dense_4 - Mean: 0.0103, Std: 0.4304
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 102ms/step - accuracy: 0.8178 - loss: 0.5968 - val
Epoch 5/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 93ms/step - accuracy: 0.8495 - loss: 0.4922Layer co
Layer conv2d_1 - Mean: 0.0394, Std: 0.2754
Layer dense_3 - Mean: 0.0089, Std: 0.1325
Layer dense_4 - Mean: 0.0095, Std: 0.4380
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 99ms/step - accuracy: 0.8495 - loss: 0.4921 - val_
Epoch 6/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - accuracy: 0.8696 - loss: 0.4266Layer co
Layer conv2d_1 - Mean: 0.0382, Std: 0.2862
Layer dense_3 - Mean: 0.0084, Std: 0.1382
Layer dense_4 - Mean: 0.0090, Std: 0.4434
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 104ms/step - accuracy: 0.8696 - loss: 0.4265 - val
Epoch 7/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 93ms/step - accuracy: 0.8836 - loss: 0.3847Layer co
Layer conv2d_1 - Mean: 0.0364, Std: 0.2939
Layer dense_3 - Mean: 0.0082, Std: 0.1428
Layer dense_4 - Mean: 0.0083, Std: 0.4478
197/197 ━━━━━━━━━━━━━━━━━━━━ 40s 99ms/step - accuracy: 0.8837 - loss: 0.3847 - val_
Epoch 8/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 99ms/step - accuracy: 0.8932 - loss: 0.3533Layer co
Layer conv2d_1 - Mean: 0.0343, Std: 0.3003
Layer dense_3 - Mean: 0.0065, Std: 0.1471
Layer dense_4 - Mean: 0.0082, Std: 0.4512
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 105ms/step - accuracy: 0.8932 - loss: 0.3533 - val
Epoch 9/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 93ms/step - accuracy: 0.9012 - loss: 0.3272Layer co
Layer conv2d_1 - Mean: 0.0324, Std: 0.3056
Layer dense_3 - Mean: 0.0068, Std: 0.1505
Layer dense_4 - Mean: 0.0078, Std: 0.4547
197/197 ━━━━━━━━━━━━━━━━━━━━ 40s 100ms/step - accuracy: 0.9012 - loss: 0.3272 - val
Epoch 10/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9076 - loss: 0.3055Layer c
Layer conv2d_1 - Mean: 0.0305, Std: 0.3104
Layer dense_3 - Mean: 0.0070, Std: 0.1537
Layer dense_4 - Mean: 0.0074, Std: 0.4580
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 107ms/step - accuracy: 0.9076 - loss: 0.3055 - val
Epoch 11/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 109ms/step - accuracy: 0.9122 - loss: 0.2862Layer c
Layer conv2d_1 - Mean: 0.0286, Std: 0.3148
Layer dense_3 - Mean: 0.0071, Std: 0.1565
Layer dense_4 - Mean: 0.0071, Std: 0.4611
197/197 ━━━━━━━━━━━━━━━━━━━━ 42s 112ms/step - accuracy: 0.9122 - loss: 0.2862 - val
Epoch 12/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9176 - loss: 0.2693Layer c
Layer conv2d_1 - Mean: 0.0270, Std: 0.3191
Layer dense_3 - Mean: 0.0060, Std: 0.1593
Layer dense_4 - Mean: 0.0069, Std: 0.4636
```

```
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 108ms/step - accuracy: 0.9177 - loss: 0.2693 - val
Epoch 13/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - accuracy: 0.9224 - loss: 0.2540Layer co
Layer conv2d_1 - Mean: 0.0254, Std: 0.3233
Layer dense_3 - Mean: 0.0060, Std: 0.1618
Layer dense_4 - Mean: 0.0065, Std: 0.4666
197/197 ━━━━━━━━━━━━━━━━━━━━ 40s 103ms/step - accuracy: 0.9224 - loss: 0.2540 - val
Epoch 14/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9264 - loss: 0.2401Layer c
Layer conv2d_1 - Mean: 0.0238, Std: 0.3275
Layer dense_3 - Mean: 0.0060, Std: 0.1641
Layer dense_4 - Mean: 0.0061, Std: 0.4696
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 106ms/step - accuracy: 0.9264 - loss: 0.2400 - val
Epoch 15/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9302 - loss: 0.2272Layer c
Layer conv2d_1 - Mean: 0.0225, Std: 0.3318
Layer dense_3 - Mean: 0.0060, Std: 0.1663
Layer dense_4 - Mean: 0.0057, Std: 0.4725
197/197 ━━━━━━━━━━━━━━━━━━━━ 40s 103ms/step - accuracy: 0.9302 - loss: 0.2272 - val
Epoch 16/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step - accuracy: 0.9341 - loss: 0.2154Layer co
Layer conv2d_1 - Mean: 0.0213, Std: 0.3358
Layer dense_3 - Mean: 0.0060, Std: 0.1683
Layer dense_4 - Mean: 0.0053, Std: 0.4754
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 100ms/step - accuracy: 0.9341 - loss: 0.2153 - val
Epoch 17/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step - accuracy: 0.9373 - loss: 0.2046Layer co
Layer conv2d_1 - Mean: 0.0197, Std: 0.3399
Layer dense_3 - Mean: 0.0060, Std: 0.1703
Layer dense_4 - Mean: 0.0049, Std: 0.4782
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 101ms/step - accuracy: 0.9373 - loss: 0.2046 - val
Epoch 18/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9413 - loss: 0.1936Layer c
Layer conv2d_1 - Mean: 0.0189, Std: 0.3435
Layer dense_3 - Mean: 0.0067, Std: 0.1728
Layer dense_4 - Mean: 0.0045, Std: 0.4845
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 107ms/step - accuracy: 0.9413 - loss: 0.1936 - val
Epoch 19/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9464 - loss: 0.1783Layer c
Layer conv2d_1 - Mean: 0.0183, Std: 0.3468
Layer dense_3 - Mean: 0.0068, Std: 0.1750
Layer dense_4 - Mean: 0.0040, Std: 0.4903
197/197 ━━━━━━━━━━━━━━━━━━━━ 41s 107ms/step - accuracy: 0.9465 - loss: 0.1783 - val
Epoch 20/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step - accuracy: 0.9492 - loss: 0.1679Layer co
Layer conv2d_1 - Mean: 0.0178, Std: 0.3497
Layer dense_3 - Mean: 0.0068, Std: 0.1769
Layer dense_4 - Mean: 0.0036, Std: 0.4950
197/197 ━━━━━━━━━━━━━━━━━━━━ 39s 98ms/step - accuracy: 0.9492 - loss: 0.1679 - val_
Epoch 21/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9519 - loss: 0.1597Layer c
Layer conv2d_1 - Mean: 0.0174, Std: 0.3525
Layer dense_3 - Mean: 0.0069, Std: 0.1787
Layer dense_4 - Mean: 0.0032, Std: 0.4989
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 107ms/step - accuracy: 0.9519 - loss: 0.1597 - val
Epoch 22/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step - accuracy: 0.9536 - loss: 0.1526Layer co
Layer conv2d_1 - Mean: 0.0170, Std: 0.3550
```

```
Layer dense_3 - Mean: 0.0070, Std: 0.1803
Layer dense_4 - Mean: 0.0029, Std: 0.5023
197/197 ━━━━━━━━━━━━━━━━━━━━ 39s 98ms/step - accuracy: 0.9536 - loss: 0.1526 - val_
Epoch 23/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9561 - loss: 0.1464Layer c
Layer conv2d_1 - Mean: 0.0166, Std: 0.3574
Layer dense_3 - Mean: 0.0071, Std: 0.1818
Layer dense_4 - Mean: 0.0026, Std: 0.5055
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 107ms/step - accuracy: 0.9561 - loss: 0.1464 - val
Epoch 24/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step - accuracy: 0.9577 - loss: 0.1407Layer co
Layer conv2d_1 - Mean: 0.0163, Std: 0.3597
Layer dense_3 - Mean: 0.0071, Std: 0.1833
Layer dense_4 - Mean: 0.0023, Std: 0.5084
197/197 ━━━━━━━━━━━━━━━━━━━━ 40s 101ms/step - accuracy: 0.9577 - loss: 0.1407 - val
Epoch 25/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 100ms/step - accuracy: 0.9593 - loss: 0.1355Layer c
Layer conv2d_1 - Mean: 0.0161, Std: 0.3618
Layer dense_3 - Mean: 0.0072, Std: 0.1846
Layer dense_4 - Mean: 0.0020, Std: 0.5111
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 107ms/step - accuracy: 0.9593 - loss: 0.1355 - val
Epoch 26/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step - accuracy: 0.9603 - loss: 0.1309Layer co
Layer conv2d_1 - Mean: 0.0159, Std: 0.3637
Layer dense_3 - Mean: 0.0073, Std: 0.1859
Layer dense_4 - Mean: 0.0017, Std: 0.5137
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 101ms/step - accuracy: 0.9603 - loss: 0.1309 - val
Epoch 27/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - accuracy: 0.9612 - loss: 0.1266Layer co
Layer conv2d_1 - Mean: 0.0156, Std: 0.3656
Layer dense_3 - Mean: 0.0074, Std: 0.1871
Layer dense_4 - Mean: 0.0014, Std: 0.5162
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 104ms/step - accuracy: 0.9612 - loss: 0.1266 - val
Epoch 28/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9621 - loss: 0.1226Layer c
Layer conv2d_1 - Mean: 0.0154, Std: 0.3673
Layer dense_3 - Mean: 0.0075, Std: 0.1882
Layer dense_4 - Mean: 0.0012, Std: 0.5185
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 107ms/step - accuracy: 0.9621 - loss: 0.1226 - val
Epoch 29/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9628 - loss: 0.1191Layer c
Layer conv2d_1 - Mean: 0.0153, Std: 0.3689
Layer dense_3 - Mean: 0.0075, Std: 0.1893
Layer dense_4 - Mean: 0.0009, Std: 0.5208
197/197 ━━━━━━━━━━━━━━━━━━━━ 41s 107ms/step - accuracy: 0.9628 - loss: 0.1191 - val
Epoch 30/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.9640 - loss: 0.1157Layer co
Layer conv2d_1 - Mean: 0.0151, Std: 0.3705
Layer dense_3 - Mean: 0.0076, Std: 0.1903
Layer dense_4 - Mean: 0.0006, Std: 0.5230
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 102ms/step - accuracy: 0.9640 - loss: 0.1157 - val
Epoch 31/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.9645 - loss: 0.1125Layer co
Layer conv2d_1 - Mean: 0.0148, Std: 0.3720
Layer dense_3 - Mean: 0.0077, Std: 0.1913
Layer dense_4 - Mean: 0.0004, Std: 0.5251
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 98ms/step - accuracy: 0.9646 - loss: 0.1125 - val_
Epoch 32/50
```

```
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9655 - loss: 0.1095Layer c
Layer conv2d_1 - Mean: 0.0146, Std: 0.3735
Layer dense_3 - Mean: 0.0078, Std: 0.1922
Layer dense_4 - Mean: 0.0001, Std: 0.5271
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 107ms/step - accuracy: 0.9655 - loss: 0.1095 - val
Epoch 33/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 113ms/step - accuracy: 0.9664 - loss: 0.1068Layer c
Layer conv2d_1 - Mean: 0.0143, Std: 0.3749
Layer dense_3 - Mean: 0.0078, Std: 0.1932
Layer dense_4 - Mean: -0.0001, Std: 0.5291
197/197 ━━━━━━━━━━━━━━━━━━━━ 24s 119ms/step - accuracy: 0.9664 - loss: 0.1068 - val
Epoch 34/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step - accuracy: 0.9674 - loss: 0.1042Layer co
Layer conv2d_1 - Mean: 0.0141, Std: 0.3763
Layer dense_3 - Mean: 0.0079, Std: 0.1941
Layer dense_4 - Mean: -0.0003, Std: 0.5310
197/197 ━━━━━━━━━━━━━━━━━━━━ 37s 101ms/step - accuracy: 0.9674 - loss: 0.1042 - val
Epoch 35/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9682 - loss: 0.1018Layer c
Layer conv2d_1 - Mean: 0.0137, Std: 0.3776
Layer dense_3 - Mean: 0.0080, Std: 0.1949
Layer dense_4 - Mean: -0.0005, Std: 0.5329
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 108ms/step - accuracy: 0.9682 - loss: 0.1018 - val
Epoch 36/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.9691 - loss: 0.0995Layer co
Layer conv2d_1 - Mean: 0.0135, Std: 0.3789
Layer dense_3 - Mean: 0.0080, Std: 0.1958
Layer dense_4 - Mean: -0.0008, Std: 0.5347
197/197 ━━━━━━━━━━━━━━━━━━━━ 40s 101ms/step - accuracy: 0.9691 - loss: 0.0995 - val
Epoch 37/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9699 - loss: 0.0973Layer c
Layer conv2d_1 - Mean: 0.0133, Std: 0.3800
Layer dense_3 - Mean: 0.0081, Std: 0.1966
Layer dense_4 - Mean: -0.0010, Std: 0.5365
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 107ms/step - accuracy: 0.9699 - loss: 0.0973 - val
Epoch 38/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.9703 - loss: 0.0953Layer co
Layer conv2d_1 - Mean: 0.0131, Std: 0.3811
Layer dense_3 - Mean: 0.0081, Std: 0.1974
Layer dense_4 - Mean: -0.0012, Std: 0.5382
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 102ms/step - accuracy: 0.9703 - loss: 0.0953 - val
Epoch 39/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 96ms/step - accuracy: 0.9710 - loss: 0.0933Layer co
Layer conv2d_1 - Mean: 0.0129, Std: 0.3822
Layer dense_3 - Mean: 0.0082, Std: 0.1982
Layer dense_4 - Mean: -0.0014, Std: 0.5398
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 103ms/step - accuracy: 0.9710 - loss: 0.0933 - val
Epoch 40/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9715 - loss: 0.0914Layer c
Layer conv2d_1 - Mean: 0.0127, Std: 0.3832
Layer dense_3 - Mean: 0.0082, Std: 0.1989
Layer dense_4 - Mean: -0.0016, Std: 0.5414
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 108ms/step - accuracy: 0.9715 - loss: 0.0914 - val
Epoch 41/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.9723 - loss: 0.0897Layer co
Layer conv2d_1 - Mean: 0.0125, Std: 0.3843
Layer dense_3 - Mean: 0.0076, Std: 0.1998
Layer dense_4 - Mean: -0.0017, Std: 0.5429
```

```
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 102ms/step - accuracy: 0.9723 - loss: 0.0897 - val
Epoch 42/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9730 - loss: 0.0879Layer c
Layer conv2d_1 - Mean: 0.0123, Std: 0.3852
Layer dense_3 - Mean: 0.0077, Std: 0.2005
Layer dense_4 - Mean: -0.0019, Std: 0.5444
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 108ms/step - accuracy: 0.9730 - loss: 0.0879 - val
Epoch 43/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.9737 - loss: 0.0864Layer co
Layer conv2d_1 - Mean: 0.0121, Std: 0.3860
Layer dense_3 - Mean: 0.0077, Std: 0.2012
Layer dense_4 - Mean: -0.0021, Std: 0.5459
197/197 ━━━━━━━━━━━━━━━━━━━━ 39s 98ms/step - accuracy: 0.9737 - loss: 0.0864 - val_
Epoch 44/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9739 - loss: 0.0850Layer c
Layer conv2d_1 - Mean: 0.0121, Std: 0.3869
Layer dense_3 - Mean: 0.0077, Std: 0.2019
Layer dense_4 - Mean: -0.0023, Std: 0.5473
197/197 ━━━━━━━━━━━━━━━━━━━━ 22s 108ms/step - accuracy: 0.9739 - loss: 0.0851 - val
Epoch 45/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - accuracy: 0.9745 - loss: 0.0836Layer co
Layer conv2d_1 - Mean: 0.0120, Std: 0.3877
Layer dense_3 - Mean: 0.0077, Std: 0.2026
Layer dense_4 - Mean: -0.0024, Std: 0.5487
197/197 ━━━━━━━━━━━━━━━━━━━━ 20s 102ms/step - accuracy: 0.9745 - loss: 0.0836 - val
Epoch 46/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 97ms/step - accuracy: 0.9752 - loss: 0.0819Layer co
Layer conv2d_1 - Mean: 0.0114, Std: 0.3885
Layer dense_3 - Mean: 0.0077, Std: 0.2033
Layer dense_4 - Mean: -0.0026, Std: 0.5500
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 103ms/step - accuracy: 0.9752 - loss: 0.0820 - val
Epoch 47/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9753 - loss: 0.0808Layer c
Layer conv2d_1 - Mean: 0.0113, Std: 0.3892
Layer dense_3 - Mean: 0.0077, Std: 0.2039
Layer dense_4 - Mean: -0.0027, Std: 0.5513
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 108ms/step - accuracy: 0.9753 - loss: 0.0808 - val
Epoch 48/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9755 - loss: 0.0794Layer c
Layer conv2d_1 - Mean: 0.0112, Std: 0.3900
Layer dense_3 - Mean: 0.0077, Std: 0.2046
Layer dense_4 - Mean: -0.0028, Std: 0.5525
197/197 ━━━━━━━━━━━━━━━━━━━━ 41s 108ms/step - accuracy: 0.9755 - loss: 0.0795 - val
Epoch 49/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 98ms/step - accuracy: 0.9756 - loss: 0.0782Layer co
Layer conv2d_1 - Mean: 0.0110, Std: 0.3907
Layer dense_3 - Mean: 0.0078, Std: 0.2053
Layer dense_4 - Mean: -0.0030, Std: 0.5538
197/197 ━━━━━━━━━━━━━━━━━━━━ 21s 104ms/step - accuracy: 0.9756 - loss: 0.0782 - val
Epoch 50/50
197/197 ━━━━━━━━━━━━━━━━━━━━ 0s 101ms/step - accuracy: 0.9763 - loss: 0.0772Layer c
Layer conv2d_1 - Mean: 0.0110, Std: 0.3913
Layer dense_3 - Mean: 0.0078, Std: 0.2059
Layer dense_4 - Mean: -0.0031, Std: 0.5549
197/197 ━━━━━━━━━━━━━━━━━━━━ 41s 105ms/step - accuracy: 0.9763 - loss: 0.0772 - val
Restoring model weights from the end of the best epoch: 50.
MNIST  test-accuracy = 0.970
<matplotlib.legend.Legend at 0x7b2a84fa7f50>
```



Chapter 4.2: Custom Training Loops and GradientTape

Gradients in TF/Keras are a little unusual. Most operations in TF/Keras do not record gradients, unlike in JAX or PyTorch. Instead, you have to explicitly tell TF/Keras that it's time to record gradients using the `tf.GradientTape` context manager. Inside a GradientTape, all operations will record gradients, and you can then call `tape.gradient()` to compute the gradients of the output with respect to the inputs. This is not terribly dissimilar to PyTorch's `backward()` method, but it is more explicit and requires you to manage the context yourself.

```
# Setting up a toy problem to show off the manual training loop in TensorFlow

# Toy data: 1 000 points, y = 1 if x₁ + x₂ > 0 else 0
x_np = np.random.randn(1_000, 2).astype("float32")
y_np = ((x_np[:, 0] + x_np[:, 1]) > 0).astype("int32")
train_ds = (
    tf.data.Dataset.from_tensor_slices((x_np, y_np))  # batches of 128
    .shuffle(1_000)
    .batch(128)
)

# Toy MLP
model = tf.keras.Sequential(
    [
        tf.keras.layers.Dense(8, activation="relu", input_shape=(2,)),
        tf.keras.layers.Dense(1)  # logits
```

```
        tf.keras.layers.Dense(1),  # logits
    ]
)

opt = tf.keras.optimizers.Adam(1e-3)
bce = tf.keras.losses.BinaryCrossentropy(from_logits=True)


# ##### MANUAL TRAINING LOOP #####

# This is more similar to the Pytorch and Jax Training Loops!
for epoch in range(10):
    running = 0.0
    for xb, yb in train_ds:
        # Open a "Gradient Tape" to tell Tensorflow its time to start recording the gra
        with tf.GradientTape() as tape:
            logits = model(xb, training=True)
            loss = bce(yb, logits)

        # Use the gradients
        grads = tape.gradient(loss, model.trainable_variables)
        opt.apply_gradients(zip(grads, model.trainable_variables))

        running += loss.numpy()

    print(f"epoch {epoch:02d}  mean-loss {running / len(train_ds):.4f}")
```

```
epoch 00  mean-loss 0.8189
epoch 01  mean-loss 0.7955
epoch 02  mean-loss 0.7716
epoch 03  mean-loss 0.7502
epoch 04  mean-loss 0.7281
epoch 05  mean-loss 0.7072
epoch 06  mean-loss 0.6863
epoch 07  mean-loss 0.6681
epoch 08  mean-loss 0.6504
epoch 09  mean-loss 0.6318
```

Exercise: Regularization

One way to prevent overfitting is to add a regularization term to the loss function. This can be done by adding a term that penalizes large weights, such as L1 or L2 regularization.

There are two ways to do this in Keras:

1. Add a regularization term to the loss function manually, e.g. `loss = loss + lambda * tf.reduce_sum(tf.square(model.trainable_weights))`, where `lambda` is the regularization strength. Do this in the custom training loop.
2. Use a built-in regularization layer, such as `tf.keras.regularizers.l1` or `tf.keras.regularizers.l2`, and add it to the model when defining the layers, e.g. `tf.keras.layers.Dense(32,`

`kernel_regularizer=tf.keras.regularizers.l2(0.01))`. This can be done without needing a custom loop.

Method 2 is easier, but method 1 lets you control the form of the regularization directly. Try both methods and see how they affect the training process. Note how the spread of the weights (as recorded by our callback!) changes as a result of the regularization.

A second type of regularization is **dropout**, which randomly sets a fraction of the inputs to zero during training. This can be done by adding a `tf.keras.layers.Dropout` layer to the model, e.g. `tf.keras.layers.Dropout(0.5)`. This is a very common regularization technique in deep learning, and can be used in conjunction with L1 or L2 regularization. It also has some nice interpretations in terms of Bayesian inference, but we will not go into that here.

Try overfitting the model, but then adding dropout and/or L1/L2 regularization.

CHAPTER 4.3 (BONUS): GANS and Generative Models

We will explore the simplest model of generative models, the **Generative Adversarial Network (GAN)**. In a previous tutorial, you learned about Normalizing Flows, which are a more sophisticated type of generative model that can learn complex distributions. GANs are a simpler type of generative model that can be used to generate new data that is similar to the training data. GANs make for a nice tutorial because they involve the interplay of *two* neural networks in a fun way, and show how MLPs can be combined to create new types of functions.

The basic premise is that we have two neural networks: a **generator** and a **discriminator**. Both of these can be MLPs, or any other type of model suited to the data. The generator takes in a random noise vector and generates a new data point, while the discriminator takes in a data point and outputs a probability that the data point is real (i.e. from the training data) or fake (i.e. generated by the generator). They are "adversarial" because they are trained in opposition to each other: the generator tries to generate data that is similar to the training data, while the discriminator tries to distinguish between real and fake data. The generator wins if it can fool the discriminator, and the discriminator wins if it can correctly classify the data. The loss functions are:

$$L_D = -\mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] - \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$$
$$L_G = -\mathbb{E}_{z\sim p_z}[\log D(G(z))]$$

(Technically, one can integrate out the discriminator and form the KL-divergence between the data distribution and the generator distribution, but this is harder to train. Normalizing flows achieve this though.)

The generator and discriminator are trained in alternating steps. This is a "minimax" game, where the generator tries to minimize its loss while the discriminator tries to maximize its loss.

In the end, we will have a generator that can generate new data points that are similar to the training

In the end, we will have a generator that can generate new data points that are similar to the training data -— or at least similar enough that the discriminator cannot tell the difference!

```
# Generator: G(z) to image, where z is a random noise vector

noise_dim = 64
generator = tf.keras.Sequential(
    [
        tf.keras.layers.Dense(128, activation="relu", input_shape=(noise_dim,)),
        tf.keras.layers.Dense(28 * 28, activation="sigmoid"),
        tf.keras.layers.Reshape((28, 28, 1)),
    ]
)

# Discriminator: D(x) to logits, where x is an image
discriminator = tf.keras.Sequential(
    [
        tf.keras.layers.Flatten(input_shape=(28, 28, 1)),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dense(1),  # logits
    ]
)

# Losses and optimiziers
bce = tf.keras.losses.BinaryCrossentropy(from_logits=True)
g_opt = tf.keras.optimizers.Adam(1e-4)
d_opt = tf.keras.optimizers.Adam(1e-4)

# helper for label tensors
ones = lambda n: tf.ones((n, 1))
zeros = lambda n: tf.zeros((n, 1))

# Training Loop
batch = 256
epochs = 50
steps = len(X_train) // batch
d_losses = []
g_losses = []

for epoch in range(epochs):
    # shuffle once per epoch
    idx = np.random.permutation(len(X_train))
    X_shuffled = X_train[idx]

    d_epoch, g_epoch = 0.0, 0.0

    for i in range(steps):
        # Get a batch of real images
        real = X_shuffled[i * batch : (i + 1) * batch]

        # Generate a batch of fake images
        z = tf.random.normal((batch, noise_dim))
        fake = generator(z, training=True)
```

```
        fake = generator(z, training=True)

        # Update discriminator
        with tf.GradientTape() as tape:
            d_real = discriminator(real, training=True)
            d_fake = discriminator(fake, training=True)
            d_loss = bce(ones(batch), d_real) + bce(zeros(batch), d_fake)
        grads = tape.gradient(d_loss, discriminator.trainable_variables)
        d_opt.apply_gradients(zip(grads, discriminator.trainable_variables))

        # Update generator
        z = tf.random.normal((batch, noise_dim))
        with tf.GradientTape() as tape:
            fake = generator(z, training=True)
            d_fake = discriminator(fake, training=True)
            g_loss = bce(ones(batch), d_fake)
        grads = tape.gradient(g_loss, generator.trainable_variables)
        g_opt.apply_gradients(zip(grads, generator.trainable_variables))

        d_epoch += d_loss.numpy()
        g_epoch += g_loss.numpy()

    d_losses.append(d_epoch / steps)
    g_losses.append(g_epoch / steps)
    print(
        f"epoch {epoch:02d}: D_loss = {d_epoch/steps:.3f}  G_loss = {g_epoch/steps:.3
    )

# Generate some samples from the generator
z = tf.random.normal((16, noise_dim))
samples = generator(z, training=False).numpy()

fig, ax = plt.subplots(4, 4, figsize=(4, 4))
for i, axi in enumerate(ax.flat):
    axi.imshow(samples[i, ..., 0], cmap="gray")
    axi.axis("off")
plt.tight_layout()
plt.show()


# plot the losses
fig = plt.figure(figsize=(8, 4))
plt.plot(d_losses, label="Discriminator Loss")
plt.plot(g_losses, label="Generator Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.yscale("log")
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/reshaping/flatten.py:37: U
  super().__init__(**kwargs)
epoch 00: D_loss =  0.676  G_loss =  4.256
epoch 01: D_loss =  0.525  G_loss =  5.752
epoch 02: D_loss =  0.409  G_loss =  6.673
epoch 03: D_loss =  0.314  G_loss =  7.409
```

```
epoch 04: D_loss =  0.228  G_loss =  6.740
epoch 05: D_loss =  0.339  G_loss =  3.165
epoch 06: D_loss =  0.735  G_loss =  1.531
epoch 07: D_loss =  1.074  G_loss =  1.046
epoch 08: D_loss =  1.285  G_loss =  0.825
epoch 09: D_loss =  1.382  G_loss =  0.730
epoch 10: D_loss =  1.408  G_loss =  0.696
epoch 11: D_loss =  1.398  G_loss =  0.689
epoch 12: D_loss =  1.377  G_loss =  0.693
epoch 13: D_loss =  1.362  G_loss =  0.702
epoch 14: D_loss =  1.340  G_loss =  0.718
epoch 15: D_loss =  1.337  G_loss =  0.723
epoch 16: D_loss =  1.343  G_loss =  0.724
epoch 17: D_loss =  1.347  G_loss =  0.719
epoch 18: D_loss =  1.359  G_loss =  0.712
epoch 19: D_loss =  1.381  G_loss =  0.698
epoch 20: D_loss =  1.411  G_loss =  0.683
epoch 21: D_loss =  1.423  G_loss =  0.678
epoch 22: D_loss =  1.385  G_loss =  0.696
epoch 23: D_loss =  1.399  G_loss =  0.687
epoch 24: D_loss =  1.375  G_loss =  0.705
epoch 25: D_loss =  1.340  G_loss =  0.725
epoch 26: D_loss =  1.319  G_loss =  0.733
epoch 27: D_loss =  1.310  G_loss =  0.740
epoch 28: D_loss =  1.323  G_loss =  0.735
epoch 29: D_loss =  1.340  G_loss =  0.727
epoch 30: D_loss =  1.374  G_loss =  0.707
```

Exercise: Upgrade to a Convolutional GAN

Your images probably aren't that great. Adding some convolutional structure might help. let's upgrade it to a Convolutional GAN (CGAN). This will allow us to generate images that are more realistic and similar to the training data.

You already know how to make a CNN discriminator. You can also make a CNN generator, but it is a little more complicated. The generator will take in a random noise vector and output an image, so it will need to upsample the noise vector to the size of the image. This can be done using transposed convolutions (also known as deconvolutions) or upsampling layers. You can use the `tf.keras.layers.Conv2DTranspose` layer to do this. The generator will also need to use a non-linear activation function, such as ReLU or LeakyReLU, to introduce non-linearity into the model. It is also recommended to use "batch normalization", which is a technique that normalizes the inputs to each layer to have zero mean and unit variance. This can help stabilize the training process and improve the performance of the model. You can use the `tf.keras.layers.BatchNormalization` layer to do this.

```
noise_dim = 64

# -- de-convolutional generator (lighter) --------------------
generator = tf.keras.Sequential(
```

```
    [
        tf.keras.layers.Dense(7 * 7 * 16, input_shape=(noise_dim,)),
        tf.keras.layers.Reshape((7, 7, 16)),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.ReLU(),
        tf.keras.layers.Conv2DTranspose(16, 4, 2, "same", use_bias=False),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.ReLU(),
        tf.keras.layers.Conv2DTranspose(
            1, 4, 2, "same", activation="sigmoid", use_bias=False
        ),
    ]
)

# -- convolutional discriminator (lighter) -------------------
discriminator = tf.keras.Sequential(
    [
        tf.keras.layers.Conv2D(16, 4, 2, "same", input_shape=(28, 28, 1)),
        tf.keras.layers.LeakyReLU(0.2),
        tf.keras.layers.Conv2D(16, 4, 2, "same"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.LeakyReLU(0.2),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(1),
    ]
)

# -- losses / optimisers ------------------------------------
bce = tf.keras.losses.BinaryCrossentropy(from_logits=False)
g_opt = tf.keras.optimizers.Adam(2e-4)
d_opt = tf.keras.optimizers.Adam(2e-4)

ones = lambda n: tf.ones((n, 1))
zeros = lambda n: tf.zeros((n, 1))


batch = 128
epochs = 10
steps = len(X_train) // batch
d_losses = []
g_losses = []

for epoch in range(epochs):
    idx = np.random.permutation(len(X_train))
    X_shuffled = X_train[idx]
    d_epoch = g_epoch = 0.0

    for i in range(steps):
        real = X_shuffled[i * batch : (i + 1) * batch]

        z = tf.random.normal((batch, noise_dim))
        fake = generator(z, training=True)
```

```
        with tf.GradientTape() as tape:
            d_real = discriminator(real, training=True)
            d_fake = discriminator(fake, training=True)
            d_loss = bce(ones(batch), tf.sigmoid(d_real)) + bce(
                zeros(batch), tf.sigmoid(d_fake)
            )
        grads = tape.gradient(d_loss, discriminator.trainable_variables)
        d_opt.apply_gradients(zip(grads, discriminator.trainable_variables))

        z = tf.random.normal((batch, noise_dim))
        with tf.GradientTape() as tape:
            fake = generator(z, training=True)
            d_fake = discriminator(fake, training=True)
            g_loss = bce(ones(batch), tf.sigmoid(d_fake))
        grads = tape.gradient(g_loss, generator.trainable_variables)
        g_opt.apply_gradients(zip(grads, generator.trainable_variables))

        d_epoch += d_loss.numpy()
        g_epoch += g_loss.numpy()

    d_losses.append(d_epoch / steps)
    g_losses.append(g_epoch / steps)
    print(
        f"epoch {epoch:02d}: D_loss = {d_epoch/steps:.3f}  G_loss = {g_epoch/steps:.3f}
    )

# generate & plot samples
z = tf.random.normal((16, noise_dim))
samples = generator(z, training=False).numpy()

fig, ax = plt.subplots(4, 4, figsize=(4, 4))
for i, axi in enumerate(ax.flat):
    axi.imshow(samples[i, ..., 0], cmap="gray")
    axi.axis("off")
plt.tight_layout()
plt.show()

# loss curves
plt.figure(figsize=(8, 4))
plt.plot(d_losses, label="Discriminator")
plt.plot(g_losses, label="Generator")
plt.yscale("log")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.show()
```