In [156…
```
import matplotlib.pyplot as plt
%matplotlib inline
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('png')
import seaborn as sns
sns.set_context("paper")
sns.set_style("ticks");
```

Homework 1

References

- Module 1: Introduction
- Module 2: Modern Machine Learning Software

Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

Student details

- **First Name: Shaunak**
- **Last Name:Mukherjee**
- **Email:mukher86@purdue.edu**
- **Used generative AI to complete this assignment (Yes/No): No**
- **Which generative AI tool did you use (if applicable)?: N/A**

Problem 1 - Recursion vs Iteration

This problem adjusted from the Structure and Interpretation of Computer Programs book. In particular from this section.

Imagine you are working with a programming language that does not have loops. This is how you have to think when writing code in `Jax`. Let's say we want to write a function that calculates the factorial of a number:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

The standard recursive definition of the factorial function is:

In [157…
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Here is how it can be used:

In [158…
```
factorial(5)
```

Out[158… 120

Let's unroll what actually happens behind the scenes:

```
factorial(5)
5 * factorial(4)
5 * (4 * factorial(3))
5 * (4 * (3 * factorial(2)))
5 * (4 * (3 * (2 * factorial(1))))
5 * (4 * (3 * (2 * 1)))
5 * (4 * (3 * 2))
5 * (4 * 6)
5 * 24
120
```

You quickly notice, that the amount of intermediate results that are stored in memory grows exponentially with the input. This won't work for large inputs, because you will run out of memory. But, there is another way to achieve the same result without exploding memory usage. We could start by multiplying 1 by 2, then the result with 3, then the result with 4, and so on. So, we keep track of a running product that we update. We don't need a loop to do this kind of iteration. We can do it with recursion:

In [159…
```
def fact_iter(product, counter, max_iter):
    if counter > max_iter:
        return product
    else:
        return fact_iter(counter * product, counter + 1, max_iter)

def good_factorial(n):
    return fact_iter(1, 1, n)
```

Check that this works as before:

In [160…
```
good_factorial(5)
```

Out[160… 120

Here is how this unrolls:

```
factorial(5)
fact_iter(1, 1, 5)
fact_iter(1, 2, 5)
fact_iter(2, 3, 5)
fact_iter(6, 4, 5)
fact_iter(24, 5, 5)
fact_iter(120, 6, 5)
120
```

We say that the second approach is *iterative* and the first approach is *recursive*. We want to be writing iterative code, because it is more efficient.

Write iterative code that, given n, computes the fibonacci number:

$$f_n = f_{n-1} + f_{n-2}$$

where $f_0 = 0$ and $f_1 = 1$. You should not use a loop!

Answer:

In [161…
```
# Your code here - Demonstrate that it works

# Define main function (no loop or recusion)
def fibonacci_iterative(n):

  if n < 0:

      print("n must be a non-negative integer.")

  if n == 0:
      return 0
  if n == 1:
      return 1

  def fibo_step(a, b, counter, max_iter):
      if counter == max_iter:
          return b
      return fibo_step(b, a + b, counter + 1, max_iter)

  # Initial state need to be defined: (f0 = 0, f1 = 1, current step = 1, max
  return fibo_step(0, 1, 1, n)
```

Here show how your code works for $n=5$ like I did above with the factorial example.

In [162…
```
# Demonstration
fibonacci_iterative(5)
```

Out[162… 5

This is how it should compute

```
fibonacci_iterative(5)
fibo_step(0, 1, 1, 5)
fibo_step(1, 1, 2, 5)
fibo_step(1, 2, 3, 5)
fibo_step(2, 3, 4, 5)
fibo_step(3, 5, 5, 5)
5
```

Problem 2 - The `foldl` function

The `foldl` function is a higher order function that is used to implement iteration. It is defined as follows:

$$\text{foldl}(f, z, [x_1, x_2, \ldots, x_n]) = f(f(\ldots f(f(z, x_1), x_2), \ldots), x_n)$$

where f is a function that takes two arguments and z is the initial value. In words, `foldl` takes a function f, an initial value z, and a list $[x_1, x_2, \ldots, x_n]$. It then applies f to z and the first element of the list, then applies f to the result of the previous application and the second element of the list, and so on.

Implement `foldl` in `Python`. Pay attention to create an iterative implementation.

Answer:

In [163…

```python
# Your code here - Demonstrate that it works

# Iterative
def foldl(f, z, lst):
  iterator = iter(lst)
  result = z

  while True:
      try:
          result = f(result, next(iterator))
      except StopIteration:
          return result
```

Use your `foldl` function to implement the `sum` function and the `product` function.

Answer:

In [164…

```python
# Defining sum and product functions using foldl function

def sum_list(lst):
    return foldl(lambda x, y: x + y, 0, lst) #The initial value is 0

def product_list(lst):
  return foldl(lambda x, y: x * y, 1, lst) # The initial value is 1
```

In [165…

```
# Your code here - Demonstrate that it works
lst = [3, 1, 2, 5, 4]

print(f" Sum:{sum_list(lst)}")

print(f" Product:{product_list(lst)}")
```

```
Sum:15
Product:120
```

Problem 3 - No Loops in Jax

Use `Jax`'s `jax.lax.scan` to implement and `jit` a function that returns the Fibonacci sequence up to a given number. Don't bother using integer types, just use `float32` for everything.

Answer:

In [166…

```
pip install -U "jax[cpu]"
```

```
/home/alkai/miniconda3/envs/mypatience/lib/python3.12/pty.py:95: RuntimeWarn
ing: os.fork() was called. os.fork() is incompatible with multithreaded cod
e, and JAX is multithreaded, so this will likely lead to a deadlock.
  pid, fd = os.forkpty()
Requirement already satisfied: jax[cpu] in /home/alkai/miniconda3/envs/mypat
ience/lib/python3.12/site-packages (0.5.0)
Requirement already satisfied: jaxlib<=0.5.0,>=0.5.0 in /home/alkai/minicond
a3/envs/mypatience/lib/python3.12/site-packages (from jax[cpu]) (0.5.0)
Requirement already satisfied: ml_dtypes>=0.4.0 in /home/alkai/miniconda3/en
vs/mypatience/lib/python3.12/site-packages (from jax[cpu]) (0.5.1)
Requirement already satisfied: numpy>=1.25 in /home/alkai/miniconda3/envs/my
patience/lib/python3.12/site-packages (from jax[cpu]) (2.2.2)
Requirement already satisfied: opt_einsum in /home/alkai/miniconda3/envs/myp
atience/lib/python3.12/site-packages (from jax[cpu]) (3.4.0)
Requirement already satisfied: scipy>=1.11.1 in /home/alkai/miniconda3/envs/
mypatience/lib/python3.12/site-packages (from jax[cpu]) (1.15.1)
Note: you may need to restart the kernel to use updated packages.
```

In [167…

```
import jax
import jax.numpy as jnp
from jax import lax
import functools as ft

# Function definition with jit, making n static
@ft.partial(jax.jit, static_argnums=(0,))
def fibonacci_sequence(n):
  def f(carry, _):
      a, b = carry
      return (b, a + b), b

  # Used jax.lax.scan here calculate the Fibonacci sequence
  _, fibo_sequence = lax.scan(f, (jnp.float32(0), jnp.float32(1)), None, ler
  return jnp.concatenate([jnp.array([0.], dtype=jnp.float32), fibo_sequence]
```

```
n = 10  # Length of the Fibonacci sequence
result = fibonacci_sequence(n)
print(f" For n = {n} Fibonacci sequence is: {jnp.int32(jnp.round(result))}")
```

```
For n = 10 Fibonacci sequence is: [ 0  1  1  2  3  5  8 13 21 34]
```

In [168…

```
# More demonstration below
n = 10

for i in range(1, n+1):
  result = fibonacci_sequence(i)
  int_result = jnp.int32(jnp.round(result))
  print(f"For n = {i}, Fibonacci sequence is: {int_result}")
```

```
For n = 1, Fibonacci sequence is: [0]
For n = 2, Fibonacci sequence is: [0 1]
For n = 3, Fibonacci sequence is: [0 1 1]
For n = 4, Fibonacci sequence is: [0 1 1 2]
For n = 5, Fibonacci sequence is: [0 1 1 2 3]
For n = 6, Fibonacci sequence is: [0 1 1 2 3 5]
For n = 7, Fibonacci sequence is: [0 1 1 2 3 5 8]
For n = 8, Fibonacci sequence is: [ 0  1  1  2  3  5  8 13]
For n = 9, Fibonacci sequence is: [ 0  1  1  2  3  5  8 13 21]
For n = 10, Fibonacci sequence is: [ 0  1  1  2  3  5  8 13 21 34]
```

Problem 4 - Feigenbaum Map

Consider the function:

$$f(x; r) = rx(1 - x)$$

where r is a parameter. One can define dynamics on the real line by iterating this function:

$$x_{n+1} = f(x_n; r)$$

where x_n is the state at time n.

This map exhibits a period doubling cascade as r increases.

Write a function in `jax`, call it `logistic_map`, that takes a lot of r's and x_0's as inputs and returns the first n states of the system. You should independently vectorize for the r's and the x_0's. And you should `jit`. Use `jax.lax.scan` to implement the iteration.

Answer:

I found a decent article https://arpita95b.medium.com/feigenbaum-constant-60fe5e5b4c72

In [169…

```
# Your code here - Demonstrate that it works
import jax
import jax.numpy as jnp
```

```
from jax import lax, vmap
import functools as ft

gpu_device = jax.devices('gpu')[0]
cpu_device = jax.devices('cpu')[0]

@ft.partial(jax.jit, static_argnums=(2,), device=cpu_device)
def logistic_map(rs, x0s, n):
  def f(x, r):
      result = r * x * (1 - x)
      # jax.debug.print("f(x, r) -> r: {r}, x: {x}, result: {result}", r=r,
      return result

  def scan(carry, _):
      x, r = carry
      next_x = f(x, r)
      # jax.debug.print("step -> x: {x}, r: {r}, next_x: {next_x}", x=x, r=r
      return (next_x, r), next_x

  @vmap
  def outer_vmap(r):
    @vmap
    def inner_vmap(x0):
        _, track = jax.lax.scan(scan, (x0, r), None, length=n)
        # jax.debug.print("inner_vmap -> r: {r}, x0: {x0}, track: {track}",

        return track

    return inner_vmap(x0s)
    print(inner_vmap(x0s))

  return outer_vmap(rs)
  print(outer_vmap(rs))
```

Test your code here:

In [170…

```
x0s = jnp.linspace(0, 1, 100)
rs = jnp.linspace(0, 4, 1_000)
n = 10_000
data = logistic_map(rs, x0s, n)
```

Your shape should be `(1000, 100, 10000)`:

In [171…

```
data.shape
```

Out[171…

```
(1000, 100, 10000)
```

Discard all but the last iteration:

In [172…

```
data = data[:, :, -1:]
data.shape
```

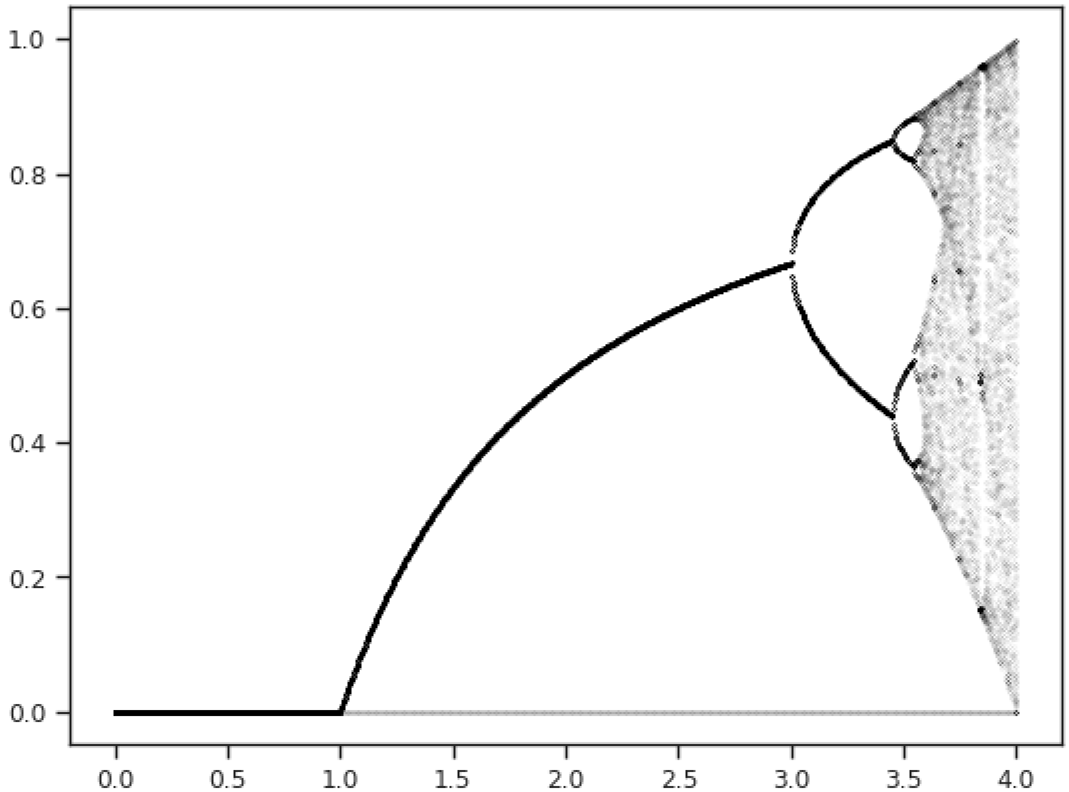
Out[172…

```
(1000, 100, 1)
```

Make the famous period doubling plot. The plot will take a while and it will take a lot of memory. I suggest you restart your kernel before moving to the next problem.

In [173…

```
fig, ax = plt.subplots()
ax.plot(rs,
        data.reshape(data.shape[0], data.shape[1] * data.shape[2]),
        '.k',
        ms=0.1,
        alpha=0.5
);
```



Problem 5 - Implement autoencoders in `jax`, `equinox`, and `optax`

Implement autoencoders in `jax` and train it on the MNIST dataset. Autoencoders, consist of two neural networks, an encoder and a decoder. The encoder maps the input to a latent space (typically of a much smaller dimension than the input), and the decoder maps the latent space back to the input space. You can think of the encoder as a compression algorithm and the decoder as a decompression algorithm. Alternatively, you can think of the encoder as the projection of the input data onto a lower-dimensional manifold, and the decoder as the reconstruction operator.

Part A

Follow these directions:

- Pick the dimension of the latent space to be 2. This means that the encoder will map the input to a 2-dimensional space, and the decoder will map the 2-dimensional space back to the input space.
- Your encoder should work on a flattened version of the input image. This means that the input to the encoder is a vector of 784 elements (28x28).
- Start by picking your encoder $z = f(x; \theta_f)$ to be a neural network with 2 hidden layers, each with 128 units and ReLU activations. Increase the number of units and layers if you think it is necessary.
- Start by picking your decoder $x' = g(z; \theta_g)$ to be a neural network with 2 hidden layers, each with 128 units and ReLU activations. Increase the number of units and layers if you think it is necessary.
- Make all your neural networks in `equinox`.
- The loss function is the mean squared error between the input and the output of the decoder:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} ||x_i - g(f(x_i; \theta_f); \theta_g)||^2.$$

where N is the number of samples in the dataset.

- Split the MNIST dataset into a training and a test set.
- Use `optax` for the optimization.
- Train the autoencoder using the Adam optimizer with a learning rate of 0.001 for 1 epoch to debug. Use a batch size of 32. Feel free to play with the learning rate and batch size.
- Monitor the loss function on the training and test set. Increase the number of epochs up to the point where the loss function on the test set stops decreasing.

Here is the dataset:

In [174…

```python
# Download the MNIST dataset
from sklearn.datasets import fetch_openml
import torch

mnist = fetch_openml('mnist_784', version=1, parser='auto')

# Split the dataset into training and test sets
from sklearn.model_selection import train_test_split

X_train_val, X_test, y_train_val, y_test = train_test_split(
    mnist.data, mnist.target, test_size=10000, random_state=42)

X_train, X_val, y_train, y_val = train_test_split(
    X_train_val, y_train_val, test_size=10000, random_state=42)
```

Answer:

Put your answer here. Use as many markdown and code blocks as you want.

In [175…

```
# Import libraries
import jax
import jax.numpy as jnp
from jax import lax, vmap
import equinox as eqx
import optax
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
from tqdm import tqdm
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import jax.random as jrandom
from sklearn.mixture import GaussianMixture
```

Defined hyperparameters

In [176…

```
# Define Hyperparameters
LEARNING_RATE = 1e-3
NUM_EPOCHS = 200 # epoch to debug
SEED = 42
BATCH_SIZE = 64
key = jrandom.PRNGKey(SEED)
```

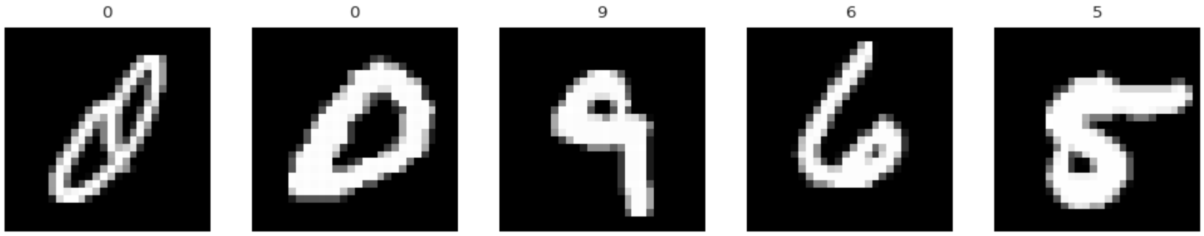
Convert to Dataloader, transform and normalize (sources https://docs.kidger.site/equinox/examples/mnist/ and https://predictivesciencelab.github.io/advanced-scientific-machine-learning/ml-software/optimization/09_gpu_training.html)

In [177…

```
# Convert data to torch tensors and normalize
X_train = torch.tensor(X_train.values, dtype=torch.float32) / 255.0
X_val = torch.tensor(X_val.values, dtype=torch.float32) / 255.0
X_test = torch.tensor(X_test.values, dtype=torch.float32) / 255.0
y_train = torch.tensor(y_train.astype(int).values, dtype=torch.long)
y_val = torch.tensor(y_val.astype(int).values, dtype=torch.long)
y_test = torch.tensor(y_test.astype(int).values, dtype=torch.long)

# Create a custom Dataset class for MNIST
class MNISTDataset(Dataset):
    def __init__(self, X, y, transform=None):
        self.X = X
        self.y = y
        self.transform = transform

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
```

```
        image = self.X[idx].view(28, 28)
        label = self.y[idx]

        if self.transform:
            image = self.transform(image)

        return image, label

# Define  transformations
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((28, 28)),
    transforms.ToTensor(),
])

# Create DataLoader instances for training, validation, and testing
train_dataset = MNISTDataset(X_train, y_train, transform=transform)
val_dataset = MNISTDataset(X_val, y_val, transform=transform)
test_dataset = MNISTDataset(X_test, y_test, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)


for data, labels in train_dataset:
    print(data.shape)  # Shape of the data in the first batch
    break

# Plot and vizualize dataset
def show_images(images, labels):
    ncols = len(labels)
    nrows = int(np.ceil(len(images) / ncols))
    fig, axes = plt.subplots(nrows, ncols, figsize=(ncols * 2, nrows * 2))
    for ax in axes.ravel():
        ax.axis("off")
    for ax, image, label in zip(axes.ravel(), images, labels):
        ax.imshow(image.squeeze(), cmap="gray")
        ax.set_title(int(label))
    plt.tight_layout()

# Select 5 random images and vizualize
k = 5
images, labels = next(iter(train_loader))
images = images[:k]
labels = labels[:k]

# Visualize the images
show_images(images, labels)
```

```
torch.Size([1, 28, 28])
```



Create the model

In [178…

```
# Define the encoder z=f(x;θf)  to be a neural network with 2 hidden layers,
class Encoder(eqx.Module):
    layers: list

    def __init__(self, key):
        key1, key2, key3 = jax.random.split(key, 3)
        self.layers = [
            eqx.nn.Linear(784, 128, key=key1),
            eqx.nn.Linear(128, 128, key=key2),
            eqx.nn.Linear(128, 2, key=key3),
        ]

    def __call__(self, x):
        x = self.layers[0](x)
        for layer in self.layers[1:]:
            x = jax.nn.relu(layer(x))
        return x

# Define the decoder x′=g(z;θg)  to be a neural network with 2 hidden layers
class Decoder(eqx.Module):
    layers: list

    def __init__(self, key):
        key1, key2, key3 = jax.random.split(key, 3)
        self.layers = [
            eqx.nn.Linear(2, 128, key=key1),
            eqx.nn.Linear(128, 128, key=key2),
            eqx.nn.Linear(128, 784, key=key3),
        ]

    def __call__(self, z):
        for layer in self.layers[:-1]:
            z = jax.nn.relu(layer(z))
        return self.layers[-1](z)

## Main model function
class Autoencoder(eqx.Module):
    encoder: Encoder
    decoder: Decoder

    def __init__(self, x):
        key1, key2 = jax.random.split(key, 2)
        self.encoder = Encoder(key)
        self.decoder = Decoder(key)
```

```
    def __call__(self, x):
        z = self.encoder(x)
        x_recon = self.decoder(z)
        return x_recon

# Initialize the model
model = Autoencoder(key)
```

Define MSE loss and Adam optimizer

In [179…
```
# Loss function is the mean squared error between the input and the output c
@eqx.filter_jit
def mse_loss(model, x):
    z = vmap(model.encoder)(x)
    x_recon = vmap(model.decoder)(z)
    return jnp.mean(jnp.sum((x - x_recon) ** 2, axis=1))


# optimizer using optax done!
optim = optax.adamw(LEARNING_RATE)
```

Evaluate loss

In [180…
```
# Define function to update the model losses
def evaluate(model,test_data):
    avg_loss = 0

    for x, y in test_loader:
        x = x.numpy()
        x = x.reshape(x.shape[0], -1)
        avg_loss += mse_loss(model, x)
    return avg_loss / len(test_data)
```

Define Training loop

In [181…
```
# Define main training loop
import time

def train(model, train_loader, test_loader, optim, num_epochs, batch_size):
    opt_state = optim.init(eqx.filter(model, eqx.is_array))

    # Lists to store the losses
    epoch_train_losses = []
    epoch_test_losses = []

    @eqx.filter_jit
    def make_step(model, opt_state, x):
        loss_value, grads = eqx.filter_value_and_grad(mse_loss)(model, x)
        updates, opt_state = optim.update(grads, opt_state, model)
        model = eqx.apply_updates(model, updates)
        return model, opt_state, loss_value

    # Overall training time
    start_time = time.time()
```

```
    # Training loop with tqdm
    for epoch in range(num_epochs):
        epoch_start_time = time.time()
        epoch_train_loss = 0
        with tqdm(total=len(train_loader), desc=f"Epoch {epoch+1}/{num_epoch
            for step, (x, y) in enumerate(train_loader):

                x = x.numpy()
                x = x.reshape(x.shape[0], -1)  # Flatten the images

                model = eqx.nn.inference_mode(model, value=False)
                model, opt_state, batch_loss = make_step(model, opt_state, x

                epoch_train_loss += batch_loss

                pbar.set_postfix(batch_loss=f"{batch_loss:.4f}")
                pbar.update(1)

        # Compute average loss for the epoch
        epoch_train_loss /= len(train_loader)
        epoch_train_losses.append(epoch_train_loss)

        # Evaluate on the test set
        model = eqx.nn.inference_mode(model, value=True)
        test_loss = evaluate(model, test_loader)
        epoch_test_losses.append(test_loss)


        epoch_end_time = time.time()
        epoch_duration = epoch_end_time - epoch_start_time

        # Print epoch losses
        print(f"Epoch {epoch+1}/{num_epochs}: Train Loss = {epoch_train_loss
    # Overall training time
    total_training_time = time.time() - start_time
    print(f"\nTotal Training Time: {total_training_time:.2f} seconds")

    # Plot the loss curve after training
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, num_epochs+1), epoch_train_losses, label="Training Los
    plt.plot(range(1, num_epochs+1), epoch_test_losses, label="Test Loss", c
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Epoch vs. Loss')
    plt.legend()
    plt.grid(True)
    plt.show()
    return model
```

Moving model to either CPU or GPU (https://predictivesciencelab.github.io/advanced-scientific-machine-learning/ml-software/optimization/09_gpu_training.html).

In [182…

```
# Check if GPU is available
is_gpu_avail = len(jax.devices('gpu')) > 0
```

```
# Helper functions for committing JAX arrays to a GPU or CPU
trained_model_cpu = lambda x: jax.device_put(x, jax.devices('cpu')[0]) if is
trained_model_gpu = lambda x: jax.device_put(x, jax.devices('gpu')[0]) if is

# Define functions to transfer model to CPU or GPU
def commit_model_to_device(model, is_gpu_avail):
    if is_gpu_avail:
        model = jax.tree.map(trained_model_gpu, model)
    else:
        model = jax.tree.map(trained_model_cpu, model)
    return model

# Commit model to appropriate device (CPU or GPU)
model = commit_model_to_device(model, is_gpu_avail)

# Print where the model is located
if is_gpu_avail:
    print('The model is on GPU:', jax.devices('gpu')[0])
else:
    print('The model is on CPU:', jax.devices('cpu')[0])
```

```
The model is on GPU: cuda:0
```

Let me training begin! Plot to monitor loss per epoch!

In [183…

```
print("Begin training")
trained_model = train(model, train_loader, test_loader, optim, NUM_EPOCHS, E
```

```
Begin training
Epoch 1/200: 100%|██████████| 782/782 [00:09<00:00, 83.91batch/s, batch_loss
=40.3885]
Epoch 1/200: Train Loss = 43.0973, Test Loss = 37.9553
Epoch 2/200: 100%|██████████| 782/782 [00:05<00:00, 133.12batch/s, batch_los
s=30.3252]
Epoch 2/200: Train Loss = 36.6069, Test Loss = 35.6513
Epoch 3/200: 100%|██████████| 782/782 [00:05<00:00, 136.54batch/s, batch_los
s=36.7825]
Epoch 3/200: Train Loss = 35.0125, Test Loss = 34.7013
Epoch 4/200: 100%|██████████| 782/782 [00:05<00:00, 134.86batch/s, batch_los
s=33.2397]
Epoch 4/200: Train Loss = 34.1403, Test Loss = 33.7027
Epoch 5/200: 100%|██████████| 782/782 [00:05<00:00, 141.92batch/s, batch_los
s=36.8677]
Epoch 5/200: Train Loss = 33.5810, Test Loss = 33.4150
Epoch 6/200: 100%|██████████| 782/782 [00:05<00:00, 143.01batch/s, batch_los
s=33.4016]
Epoch 6/200: Train Loss = 33.1065, Test Loss = 33.1408
Epoch 7/200: 100%|██████████| 782/782 [00:05<00:00, 141.86batch/s, batch_los
s=37.4913]
Epoch 7/200: Train Loss = 32.7732, Test Loss = 32.9709
Epoch 8/200: 100%|██████████| 782/782 [00:05<00:00, 143.96batch/s, batch_los
s=29.1926]
Epoch 8/200: Train Loss = 32.4915, Test Loss = 32.2802
```

```
Epoch 9/200: 100%|██████████| 782/782 [00:05<00:00, 145.30batch/s, batch_los
s=29.7216]
Epoch 9/200: Train Loss = 32.2092, Test Loss = 32.0618
Epoch 10/200: 100%|██████████| 782/782 [00:05<00:00, 143.31batch/s, batch_lo
ss=33.0313]
Epoch 10/200: Train Loss = 31.9579, Test Loss = 32.0986
Epoch 11/200: 100%|██████████| 782/782 [00:05<00:00, 140.28batch/s, batch_lo
ss=28.7470]
Epoch 11/200: Train Loss = 31.8132, Test Loss = 31.8850
Epoch 12/200: 100%|██████████| 782/782 [00:05<00:00, 139.42batch/s, batch_lo
ss=29.1856]
Epoch 12/200: Train Loss = 31.6744, Test Loss = 31.7299
Epoch 13/200: 100%|██████████| 782/782 [00:05<00:00, 144.35batch/s, batch_lo
ss=27.9213]
Epoch 13/200: Train Loss = 31.4322, Test Loss = 31.3597
Epoch 14/200: 100%|██████████| 782/782 [00:05<00:00, 141.62batch/s, batch_lo
ss=29.3949]
Epoch 14/200: Train Loss = 31.3172, Test Loss = 31.4237
Epoch 15/200: 100%|██████████| 782/782 [00:05<00:00, 142.67batch/s, batch_lo
ss=27.7985]
Epoch 15/200: Train Loss = 31.2275, Test Loss = 31.3197
Epoch 16/200: 100%|██████████| 782/782 [00:05<00:00, 144.89batch/s, batch_lo
ss=28.9488]
Epoch 16/200: Train Loss = 31.1003, Test Loss = 31.3596
Epoch 17/200: 100%|██████████| 782/782 [00:05<00:00, 141.25batch/s, batch_lo
ss=29.3057]
Epoch 17/200: Train Loss = 30.9836, Test Loss = 31.3197
Epoch 18/200: 100%|██████████| 782/782 [00:05<00:00, 141.21batch/s, batch_lo
ss=25.3147]
Epoch 18/200: Train Loss = 30.8826, Test Loss = 31.3027
Epoch 19/200: 100%|██████████| 782/782 [00:05<00:00, 146.46batch/s, batch_lo
ss=29.3270]
Epoch 19/200: Train Loss = 30.8157, Test Loss = 31.3755
Epoch 20/200: 100%|██████████| 782/782 [00:05<00:00, 141.23batch/s, batch_lo
ss=29.0835]
Epoch 20/200: Train Loss = 30.8417, Test Loss = 30.9562
Epoch 21/200: 100%|██████████| 782/782 [00:05<00:00, 142.84batch/s, batch_lo
ss=32.4169]
Epoch 21/200: Train Loss = 30.6869, Test Loss = 30.9446
Epoch 22/200: 100%|██████████| 782/782 [00:05<00:00, 144.81batch/s, batch_lo
ss=31.8159]
Epoch 22/200: Train Loss = 30.6253, Test Loss = 30.9703
Epoch 23/200: 100%|██████████| 782/782 [00:05<00:00, 140.83batch/s, batch_lo
ss=28.1272]
Epoch 23/200: Train Loss = 30.5408, Test Loss = 30.7403
Epoch 24/200: 100%|██████████| 782/782 [00:05<00:00, 141.47batch/s, batch_lo
ss=40.2268]
Epoch 24/200: Train Loss = 30.4825, Test Loss = 30.6818
Epoch 25/200: 100%|██████████| 782/782 [00:05<00:00, 143.89batch/s, batch_lo
ss=32.8621]
Epoch 25/200: Train Loss = 30.3941, Test Loss = 30.6036
```

```
Epoch 26/200: 100%|██████████| 782/782 [00:05<00:00, 144.86batch/s, batch_loss=28.7079]
Epoch 26/200: Train Loss = 30.4595, Test Loss = 30.5018
Epoch 27/200: 100%|██████████| 782/782 [00:05<00:00, 141.30batch/s, batch_loss=33.3729]
Epoch 27/200: Train Loss = 30.3344, Test Loss = 30.6859
Epoch 28/200: 100%|██████████| 782/782 [00:05<00:00, 145.23batch/s, batch_loss=31.1628]
Epoch 28/200: Train Loss = 30.3596, Test Loss = 30.7349
Epoch 29/200: 100%|██████████| 782/782 [00:05<00:00, 146.12batch/s, batch_loss=32.3935]
Epoch 29/200: Train Loss = 30.3290, Test Loss = 31.0594
Epoch 30/200: 100%|██████████| 782/782 [00:05<00:00, 141.78batch/s, batch_loss=30.0424]
Epoch 30/200: Train Loss = 30.1864, Test Loss = 30.6200
Epoch 31/200: 100%|██████████| 782/782 [00:05<00:00, 141.88batch/s, batch_loss=28.2656]
Epoch 31/200: Train Loss = 30.2300, Test Loss = 30.2167
Epoch 32/200: 100%|██████████| 782/782 [00:05<00:00, 137.49batch/s, batch_loss=28.2378]
Epoch 32/200: Train Loss = 30.1555, Test Loss = 30.6724
Epoch 33/200: 100%|██████████| 782/782 [00:05<00:00, 141.77batch/s, batch_loss=23.7123]
Epoch 33/200: Train Loss = 30.0888, Test Loss = 30.3979
Epoch 34/200: 100%|██████████| 782/782 [00:05<00:00, 140.02batch/s, batch_loss=27.3675]
Epoch 34/200: Train Loss = 30.0422, Test Loss = 30.3231
Epoch 35/200: 100%|██████████| 782/782 [00:05<00:00, 143.22batch/s, batch_loss=30.3943]
Epoch 35/200: Train Loss = 29.9580, Test Loss = 30.3064
Epoch 36/200: 100%|██████████| 782/782 [00:05<00:00, 144.56batch/s, batch_loss=38.0775]
Epoch 36/200: Train Loss = 29.9551, Test Loss = 30.6592
Epoch 37/200: 100%|██████████| 782/782 [00:05<00:00, 145.26batch/s, batch_loss=32.6704]
Epoch 37/200: Train Loss = 29.9642, Test Loss = 30.4152
Epoch 38/200: 100%|██████████| 782/782 [00:05<00:00, 145.06batch/s, batch_loss=30.3393]
Epoch 38/200: Train Loss = 29.9406, Test Loss = 30.2569
Epoch 39/200: 100%|██████████| 782/782 [00:05<00:00, 144.70batch/s, batch_loss=31.2125]
Epoch 39/200: Train Loss = 29.9168, Test Loss = 30.3572
Epoch 40/200: 100%|██████████| 782/782 [00:05<00:00, 144.56batch/s, batch_loss=31.1600]
Epoch 40/200: Train Loss = 29.8967, Test Loss = 30.3431
Epoch 41/200: 100%|██████████| 782/782 [00:05<00:00, 144.68batch/s, batch_loss=25.5277]
Epoch 41/200: Train Loss = 29.8091, Test Loss = 30.0529
Epoch 42/200: 100%|██████████| 782/782 [00:05<00:00, 144.09batch/s, batch_loss=34.2762]
Epoch 42/200: Train Loss = 29.7864, Test Loss = 30.6195
```

```
Epoch 43/200: 100%|██████████| 782/782 [00:05<00:00, 145.71batch/s, batch_loss=31.9540]
Epoch 43/200: Train Loss = 29.8621, Test Loss = 30.1404
Epoch 44/200: 100%|██████████| 782/782 [00:05<00:00, 143.13batch/s, batch_loss=31.3466]
Epoch 44/200: Train Loss = 29.7848, Test Loss = 30.2929
Epoch 45/200: 100%|██████████| 782/782 [00:05<00:00, 139.24batch/s, batch_loss=27.0588]
Epoch 45/200: Train Loss = 29.7122, Test Loss = 30.3004
Epoch 46/200: 100%|██████████| 782/782 [00:05<00:00, 142.69batch/s, batch_loss=36.1972]
Epoch 46/200: Train Loss = 29.7019, Test Loss = 30.3263
Epoch 47/200: 100%|██████████| 782/782 [00:05<00:00, 140.74batch/s, batch_loss=28.3988]
Epoch 47/200: Train Loss = 29.7163, Test Loss = 30.4308
Epoch 48/200: 100%|██████████| 782/782 [00:05<00:00, 146.79batch/s, batch_loss=30.2881]
Epoch 48/200: Train Loss = 29.6488, Test Loss = 30.4034
Epoch 49/200: 100%|██████████| 782/782 [00:05<00:00, 143.03batch/s, batch_loss=31.5329]
Epoch 49/200: Train Loss = 29.5678, Test Loss = 30.4391
Epoch 50/200: 100%|██████████| 782/782 [00:05<00:00, 143.68batch/s, batch_loss=37.7039]
Epoch 50/200: Train Loss = 29.6613, Test Loss = 30.1656
Epoch 51/200: 100%|██████████| 782/782 [00:05<00:00, 145.37batch/s, batch_loss=31.4037]
Epoch 51/200: Train Loss = 29.6177, Test Loss = 29.9529
Epoch 52/200: 100%|██████████| 782/782 [00:05<00:00, 143.24batch/s, batch_loss=24.8788]
Epoch 52/200: Train Loss = 29.5184, Test Loss = 30.0719
Epoch 53/200: 100%|██████████| 782/782 [00:05<00:00, 141.68batch/s, batch_loss=32.6587]
Epoch 53/200: Train Loss = 29.5925, Test Loss = 30.4770
Epoch 54/200: 100%|██████████| 782/782 [00:05<00:00, 143.73batch/s, batch_loss=28.2593]
Epoch 54/200: Train Loss = 29.6018, Test Loss = 30.0303
Epoch 55/200: 100%|██████████| 782/782 [00:05<00:00, 143.83batch/s, batch_loss=33.6065]
Epoch 55/200: Train Loss = 29.4969, Test Loss = 29.9502
Epoch 56/200: 100%|██████████| 782/782 [00:05<00:00, 144.10batch/s, batch_loss=30.0400]
Epoch 56/200: Train Loss = 29.4536, Test Loss = 29.8563
Epoch 57/200: 100%|██████████| 782/782 [00:05<00:00, 145.66batch/s, batch_loss=33.3499]
Epoch 57/200: Train Loss = 29.3897, Test Loss = 29.8888
Epoch 58/200: 100%|██████████| 782/782 [00:05<00:00, 142.58batch/s, batch_loss=28.8269]
Epoch 58/200: Train Loss = 29.4978, Test Loss = 29.8606
Epoch 59/200: 100%|██████████| 782/782 [00:05<00:00, 144.81batch/s, batch_loss=25.4529]
Epoch 59/200: Train Loss = 29.3289, Test Loss = 30.1883
```

```
Epoch 60/200: 100%|██████████| 782/782 [00:05<00:00, 145.50batch/s, batch_lo
ss=21.3011]
Epoch 60/200: Train Loss = 29.3536, Test Loss = 29.8584
Epoch 61/200: 100%|██████████| 782/782 [00:05<00:00, 143.68batch/s, batch_lo
ss=29.2312]
Epoch 61/200: Train Loss = 29.3801, Test Loss = 30.3353
Epoch 62/200: 100%|██████████| 782/782 [00:05<00:00, 143.27batch/s, batch_lo
ss=30.4168]
Epoch 62/200: Train Loss = 29.3414, Test Loss = 30.2481
Epoch 63/200: 100%|██████████| 782/782 [00:05<00:00, 145.85batch/s, batch_lo
ss=31.5441]
Epoch 63/200: Train Loss = 29.3324, Test Loss = 30.0404
Epoch 64/200: 100%|██████████| 782/782 [00:05<00:00, 144.26batch/s, batch_lo
ss=29.1349]
Epoch 64/200: Train Loss = 29.3337, Test Loss = 29.9714
Epoch 65/200: 100%|██████████| 782/782 [00:05<00:00, 142.61batch/s, batch_lo
ss=25.4286]
Epoch 65/200: Train Loss = 29.2887, Test Loss = 29.8365
Epoch 66/200: 100%|██████████| 782/782 [00:05<00:00, 142.27batch/s, batch_lo
ss=34.1014]
Epoch 66/200: Train Loss = 29.2356, Test Loss = 29.9825
Epoch 67/200: 100%|██████████| 782/782 [00:05<00:00, 144.47batch/s, batch_lo
ss=25.7142]
Epoch 67/200: Train Loss = 29.2604, Test Loss = 30.1682
Epoch 68/200: 100%|██████████| 782/782 [00:05<00:00, 142.47batch/s, batch_lo
ss=22.2962]
Epoch 68/200: Train Loss = 29.1732, Test Loss = 30.3228
Epoch 69/200: 100%|██████████| 782/782 [00:05<00:00, 143.17batch/s, batch_lo
ss=23.7042]
Epoch 69/200: Train Loss = 29.2170, Test Loss = 29.9655
Epoch 70/200: 100%|██████████| 782/782 [00:05<00:00, 135.99batch/s, batch_lo
ss=26.8767]
Epoch 70/200: Train Loss = 29.1978, Test Loss = 30.0682
Epoch 71/200: 100%|██████████| 782/782 [00:05<00:00, 139.22batch/s, batch_lo
ss=31.0411]
Epoch 71/200: Train Loss = 29.2671, Test Loss = 30.0326
Epoch 72/200: 100%|██████████| 782/782 [00:05<00:00, 135.58batch/s, batch_lo
ss=28.0235]
Epoch 72/200: Train Loss = 29.1772, Test Loss = 29.6206
Epoch 73/200: 100%|██████████| 782/782 [00:05<00:00, 135.19batch/s, batch_lo
ss=25.4176]
Epoch 73/200: Train Loss = 29.1737, Test Loss = 30.2311
Epoch 74/200: 100%|██████████| 782/782 [00:05<00:00, 134.89batch/s, batch_lo
ss=28.7899]
Epoch 74/200: Train Loss = 29.0628, Test Loss = 29.8089
Epoch 75/200: 100%|██████████| 782/782 [00:05<00:00, 137.39batch/s, batch_lo
ss=28.3059]
Epoch 75/200: Train Loss = 29.0828, Test Loss = 29.9714
Epoch 76/200: 100%|██████████| 782/782 [00:06<00:00, 116.23batch/s, batch_lo
ss=28.8893]
Epoch 76/200: Train Loss = 29.1236, Test Loss = 29.9689
```

```
Epoch 77/200: 100%|██████████| 782/782 [00:06<00:00, 123.56batch/s, batch_lo
ss=30.2061]
Epoch 77/200: Train Loss = 29.1174, Test Loss = 29.8316
Epoch 78/200: 100%|██████████| 782/782 [00:06<00:00, 124.92batch/s, batch_lo
ss=27.8868]
Epoch 78/200: Train Loss = 29.0268, Test Loss = 29.5620
Epoch 79/200: 100%|██████████| 782/782 [00:06<00:00, 116.94batch/s, batch_lo
ss=33.5580]
Epoch 79/200: Train Loss = 29.0163, Test Loss = 29.8766
Epoch 80/200: 100%|██████████| 782/782 [00:06<00:00, 120.62batch/s, batch_lo
ss=27.2248]
Epoch 80/200: Train Loss = 28.9687, Test Loss = 30.0372
Epoch 81/200: 100%|██████████| 782/782 [00:06<00:00, 121.15batch/s, batch_lo
ss=21.4224]
Epoch 81/200: Train Loss = 29.0950, Test Loss = 29.6473
Epoch 82/200: 100%|██████████| 782/782 [00:06<00:00, 120.03batch/s, batch_lo
ss=29.1212]
Epoch 82/200: Train Loss = 28.9487, Test Loss = 29.5958
Epoch 83/200: 100%|██████████| 782/782 [00:06<00:00, 119.86batch/s, batch_lo
ss=29.8707]
Epoch 83/200: Train Loss = 29.0034, Test Loss = 29.8029
Epoch 84/200: 100%|██████████| 782/782 [00:06<00:00, 113.52batch/s, batch_lo
ss=28.6191]
Epoch 84/200: Train Loss = 28.9196, Test Loss = 29.5851
Epoch 85/200: 100%|██████████| 782/782 [00:06<00:00, 118.63batch/s, batch_lo
ss=32.5657]
Epoch 85/200: Train Loss = 28.9256, Test Loss = 29.9539
Epoch 86/200: 100%|██████████| 782/782 [00:06<00:00, 120.58batch/s, batch_lo
ss=31.6965]
Epoch 86/200: Train Loss = 29.0193, Test Loss = 29.5354
Epoch 87/200: 100%|██████████| 782/782 [00:06<00:00, 124.03batch/s, batch_lo
ss=28.2009]
Epoch 87/200: Train Loss = 28.9533, Test Loss = 29.4713
Epoch 88/200: 100%|██████████| 782/782 [00:06<00:00, 122.54batch/s, batch_lo
ss=30.3608]
Epoch 88/200: Train Loss = 28.9361, Test Loss = 29.6787
Epoch 89/200: 100%|██████████| 782/782 [00:06<00:00, 119.88batch/s, batch_lo
ss=34.5151]
Epoch 89/200: Train Loss = 28.8866, Test Loss = 29.5725
Epoch 90/200: 100%|██████████| 782/782 [00:06<00:00, 124.10batch/s, batch_lo
ss=23.9858]
Epoch 90/200: Train Loss = 28.8578, Test Loss = 29.5407
Epoch 91/200: 100%|██████████| 782/782 [00:06<00:00, 118.91batch/s, batch_lo
ss=40.8126]
Epoch 91/200: Train Loss = 28.8513, Test Loss = 29.4829
Epoch 92/200: 100%|██████████| 782/782 [00:06<00:00, 116.54batch/s, batch_lo
ss=29.2464]
Epoch 92/200: Train Loss = 28.8095, Test Loss = 29.6520
Epoch 93/200: 100%|██████████| 782/782 [00:06<00:00, 126.76batch/s, batch_lo
ss=28.8718]
Epoch 93/200: Train Loss = 28.7617, Test Loss = 29.5287
```

```
Epoch 94/200: 100%|██████████| 782/782 [00:06<00:00, 129.95batch/s, batch_lo
ss=24.4867]
Epoch 94/200: Train Loss = 28.8324, Test Loss = 29.7328
Epoch 95/200: 100%|██████████| 782/782 [00:05<00:00, 134.63batch/s, batch_lo
ss=32.5319]
Epoch 95/200: Train Loss = 29.4191, Test Loss = 30.4555
Epoch 96/200: 100%|██████████| 782/782 [00:06<00:00, 121.39batch/s, batch_lo
ss=30.8111]
Epoch 96/200: Train Loss = 29.4995, Test Loss = 29.8100
Epoch 97/200: 100%|██████████| 782/782 [00:05<00:00, 130.77batch/s, batch_lo
ss=25.0860]
Epoch 97/200: Train Loss = 29.3378, Test Loss = 29.6893
Epoch 98/200: 100%|██████████| 782/782 [00:06<00:00, 129.15batch/s, batch_lo
ss=31.3611]
Epoch 98/200: Train Loss = 29.0981, Test Loss = 29.5729
Epoch 99/200: 100%|██████████| 782/782 [00:06<00:00, 128.03batch/s, batch_lo
ss=30.1908]
Epoch 99/200: Train Loss = 28.9644, Test Loss = 29.6353
Epoch 100/200: 100%|██████████| 782/782 [00:06<00:00, 126.81batch/s, batch_l
oss=27.9115]
Epoch 100/200: Train Loss = 28.8095, Test Loss = 29.5858
Epoch 101/200: 100%|██████████| 782/782 [00:06<00:00, 126.76batch/s, batch_l
oss=27.9974]
Epoch 101/200: Train Loss = 28.8527, Test Loss = 29.5576
Epoch 102/200: 100%|██████████| 782/782 [00:06<00:00, 128.79batch/s, batch_l
oss=27.7476]
Epoch 102/200: Train Loss = 28.9534, Test Loss = 29.4473
Epoch 103/200: 100%|██████████| 782/782 [00:06<00:00, 122.91batch/s, batch_l
oss=29.4723]
Epoch 103/200: Train Loss = 28.7934, Test Loss = 29.8050
Epoch 104/200: 100%|██████████| 782/782 [00:06<00:00, 124.49batch/s, batch_l
oss=28.7752]
Epoch 104/200: Train Loss = 28.8226, Test Loss = 29.4036
Epoch 105/200: 100%|██████████| 782/782 [00:06<00:00, 128.37batch/s, batch_l
oss=29.6002]
Epoch 105/200: Train Loss = 28.7820, Test Loss = 29.7762
Epoch 106/200: 100%|██████████| 782/782 [00:06<00:00, 123.96batch/s, batch_l
oss=25.0442]
Epoch 106/200: Train Loss = 28.7930, Test Loss = 29.5273
Epoch 107/200: 100%|██████████| 782/782 [00:06<00:00, 119.28batch/s, batch_l
oss=27.7659]
Epoch 107/200: Train Loss = 28.7127, Test Loss = 29.6170
Epoch 108/200: 100%|██████████| 782/782 [00:06<00:00, 124.95batch/s, batch_l
oss=25.4546]
Epoch 108/200: Train Loss = 28.7073, Test Loss = 29.4876
Epoch 109/200: 100%|██████████| 782/782 [00:06<00:00, 120.66batch/s, batch_l
oss=27.3065]
Epoch 109/200: Train Loss = 28.7615, Test Loss = 29.5585
Epoch 110/200: 100%|██████████| 782/782 [00:06<00:00, 123.40batch/s, batch_l
oss=26.6985]
Epoch 110/200: Train Loss = 28.7297, Test Loss = 29.7152
```

```
Epoch 111/200: 100%|██████████| 782/782 [00:06<00:00, 124.22batch/s, batch_loss=30.6399]
Epoch 111/200: Train Loss = 28.7016, Test Loss = 29.3584
Epoch 112/200: 100%|██████████| 782/782 [00:06<00:00, 120.02batch/s, batch_loss=28.3477]
Epoch 112/200: Train Loss = 28.7877, Test Loss = 29.5341
Epoch 113/200: 100%|██████████| 782/782 [00:06<00:00, 124.68batch/s, batch_loss=28.0380]
Epoch 113/200: Train Loss = 28.7384, Test Loss = 29.4302
Epoch 114/200: 100%|██████████| 782/782 [00:06<00:00, 122.50batch/s, batch_loss=31.4307]
Epoch 114/200: Train Loss = 28.7660, Test Loss = 29.4533
Epoch 115/200: 100%|██████████| 782/782 [00:06<00:00, 122.83batch/s, batch_loss=25.3053]
Epoch 115/200: Train Loss = 28.6824, Test Loss = 29.3128
Epoch 116/200: 100%|██████████| 782/782 [00:06<00:00, 125.41batch/s, batch_loss=32.3190]
Epoch 116/200: Train Loss = 28.6231, Test Loss = 29.2500
Epoch 117/200: 100%|██████████| 782/782 [00:06<00:00, 119.75batch/s, batch_loss=24.3133]
Epoch 117/200: Train Loss = 28.6769, Test Loss = 29.6832
Epoch 118/200: 100%|██████████| 782/782 [00:06<00:00, 123.76batch/s, batch_loss=30.9258]
Epoch 118/200: Train Loss = 28.7352, Test Loss = 29.4646
Epoch 119/200: 100%|██████████| 782/782 [00:06<00:00, 123.19batch/s, batch_loss=31.7804]
Epoch 119/200: Train Loss = 28.6853, Test Loss = 29.4028
Epoch 120/200: 100%|██████████| 782/782 [00:06<00:00, 118.62batch/s, batch_loss=29.4043]
Epoch 120/200: Train Loss = 28.6078, Test Loss = 29.6238
Epoch 121/200: 100%|██████████| 782/782 [00:06<00:00, 122.91batch/s, batch_loss=28.6820]
Epoch 121/200: Train Loss = 28.5892, Test Loss = 29.4300
Epoch 122/200: 100%|██████████| 782/782 [00:06<00:00, 116.48batch/s, batch_loss=31.1910]
Epoch 122/200: Train Loss = 28.5980, Test Loss = 29.5505
Epoch 123/200: 100%|██████████| 782/782 [00:06<00:00, 118.81batch/s, batch_loss=34.3658]
Epoch 123/200: Train Loss = 28.6194, Test Loss = 29.7028
Epoch 124/200: 100%|██████████| 782/782 [00:06<00:00, 116.86batch/s, batch_loss=27.3717]
Epoch 124/200: Train Loss = 28.5678, Test Loss = 29.5350
Epoch 125/200: 100%|██████████| 782/782 [00:07<00:00, 110.97batch/s, batch_loss=30.5045]
Epoch 125/200: Train Loss = 28.6171, Test Loss = 29.5907
Epoch 126/200: 100%|██████████| 782/782 [00:06<00:00, 118.90batch/s, batch_loss=26.4443]
Epoch 126/200: Train Loss = 28.5961, Test Loss = 29.4439
Epoch 127/200: 100%|██████████| 782/782 [00:06<00:00, 121.41batch/s, batch_loss=24.1591]
Epoch 127/200: Train Loss = 28.7077, Test Loss = 29.3619
```

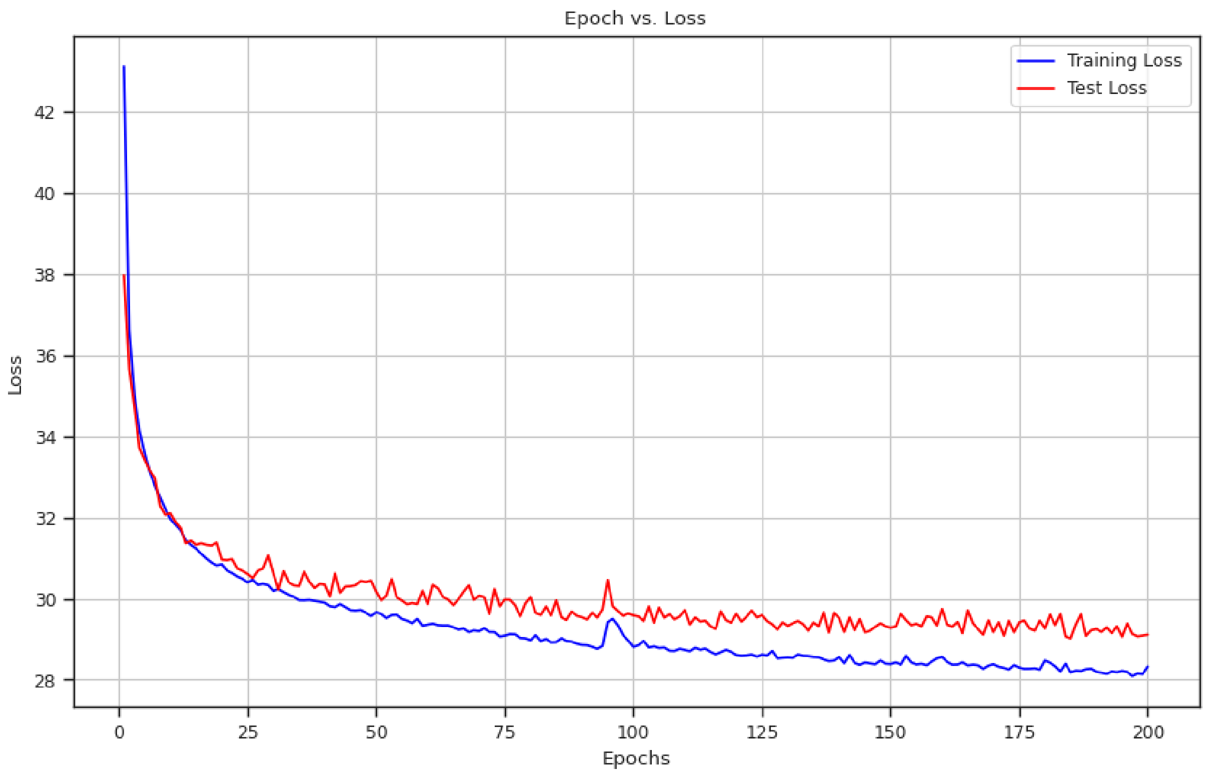
```
Epoch 128/200: 100%|██████████| 782/782 [00:05<00:00, 131.01batch/s, batch_loss=25.0038]
Epoch 128/200: Train Loss = 28.5261, Test Loss = 29.2334
Epoch 129/200: 100%|██████████| 782/782 [00:06<00:00, 122.19batch/s, batch_loss=29.2787]
Epoch 129/200: Train Loss = 28.5402, Test Loss = 29.4095
Epoch 130/200: 100%|██████████| 782/782 [00:06<00:00, 121.87batch/s, batch_loss=24.2326]
Epoch 130/200: Train Loss = 28.5496, Test Loss = 29.3239
Epoch 131/200: 100%|██████████| 782/782 [00:06<00:00, 128.69batch/s, batch_loss=26.6542]
Epoch 131/200: Train Loss = 28.5386, Test Loss = 29.3909
Epoch 132/200: 100%|██████████| 782/782 [00:06<00:00, 127.10batch/s, batch_loss=28.7576]
Epoch 132/200: Train Loss = 28.6203, Test Loss = 29.4445
Epoch 133/200: 100%|██████████| 782/782 [00:06<00:00, 124.51batch/s, batch_loss=28.3870]
Epoch 133/200: Train Loss = 28.5882, Test Loss = 29.3640
Epoch 134/200: 100%|██████████| 782/782 [00:06<00:00, 127.90batch/s, batch_loss=33.9039]
Epoch 134/200: Train Loss = 28.5825, Test Loss = 29.2032
Epoch 135/200: 100%|██████████| 782/782 [00:06<00:00, 127.19batch/s, batch_loss=26.0939]
Epoch 135/200: Train Loss = 28.5605, Test Loss = 29.4058
Epoch 136/200: 100%|██████████| 782/782 [00:06<00:00, 130.16batch/s, batch_loss=27.9382]
Epoch 136/200: Train Loss = 28.5527, Test Loss = 29.3160
Epoch 137/200: 100%|██████████| 782/782 [00:06<00:00, 127.19batch/s, batch_loss=29.1416]
Epoch 137/200: Train Loss = 28.4990, Test Loss = 29.6610
Epoch 138/200: 100%|██████████| 782/782 [00:06<00:00, 124.23batch/s, batch_loss=28.5246]
Epoch 138/200: Train Loss = 28.4581, Test Loss = 29.1621
Epoch 139/200: 100%|██████████| 782/782 [00:06<00:00, 119.73batch/s, batch_loss=23.8353]
Epoch 139/200: Train Loss = 28.4715, Test Loss = 29.6456
Epoch 140/200: 100%|██████████| 782/782 [00:06<00:00, 127.52batch/s, batch_loss=27.7401]
Epoch 140/200: Train Loss = 28.5604, Test Loss = 29.5201
Epoch 141/200: 100%|██████████| 782/782 [00:06<00:00, 126.49batch/s, batch_loss=31.4110]
Epoch 141/200: Train Loss = 28.4064, Test Loss = 29.1767
Epoch 142/200: 100%|██████████| 782/782 [00:05<00:00, 130.47batch/s, batch_loss=30.1551]
Epoch 142/200: Train Loss = 28.6104, Test Loss = 29.5360
Epoch 143/200: 100%|██████████| 782/782 [00:05<00:00, 130.56batch/s, batch_loss=18.6283]
Epoch 143/200: Train Loss = 28.4150, Test Loss = 29.2179
Epoch 144/200: 100%|██████████| 782/782 [00:06<00:00, 127.23batch/s, batch_loss=25.6871]
Epoch 144/200: Train Loss = 28.3662, Test Loss = 29.4947
```

```
Epoch 145/200: 100%|██████████| 782/782 [00:05<00:00, 130.66batch/s, batch_loss=24.8993]
Epoch 145/200: Train Loss = 28.4247, Test Loss = 29.1686
Epoch 146/200: 100%|██████████| 782/782 [00:05<00:00, 130.68batch/s, batch_loss=30.5568]
Epoch 146/200: Train Loss = 28.4019, Test Loss = 29.2054
Epoch 147/200: 100%|██████████| 782/782 [00:05<00:00, 130.46batch/s, batch_loss=27.2354]
Epoch 147/200: Train Loss = 28.3811, Test Loss = 29.3002
Epoch 148/200: 100%|██████████| 782/782 [00:06<00:00, 125.10batch/s, batch_loss=35.9784]
Epoch 148/200: Train Loss = 28.4729, Test Loss = 29.3878
Epoch 149/200: 100%|██████████| 782/782 [00:06<00:00, 127.65batch/s, batch_loss=31.2953]
Epoch 149/200: Train Loss = 28.3967, Test Loss = 29.3164
Epoch 150/200: 100%|██████████| 782/782 [00:06<00:00, 124.71batch/s, batch_loss=31.8915]
Epoch 150/200: Train Loss = 28.3857, Test Loss = 29.2798
Epoch 151/200: 100%|██████████| 782/782 [00:05<00:00, 131.49batch/s, batch_loss=30.2205]
Epoch 151/200: Train Loss = 28.4267, Test Loss = 29.3106
Epoch 152/200: 100%|██████████| 782/782 [00:06<00:00, 125.19batch/s, batch_loss=24.3238]
Epoch 152/200: Train Loss = 28.3729, Test Loss = 29.6210
Epoch 153/200: 100%|██████████| 782/782 [00:05<00:00, 130.42batch/s, batch_loss=29.8840]
Epoch 153/200: Train Loss = 28.5868, Test Loss = 29.4658
Epoch 154/200: 100%|██████████| 782/782 [00:05<00:00, 131.39batch/s, batch_loss=30.9089]
Epoch 154/200: Train Loss = 28.4263, Test Loss = 29.3407
Epoch 155/200: 100%|██████████| 782/782 [00:06<00:00, 126.57batch/s, batch_loss=28.9048]
Epoch 155/200: Train Loss = 28.3761, Test Loss = 29.3771
Epoch 156/200: 100%|██████████| 782/782 [00:06<00:00, 127.84batch/s, batch_loss=27.6915]
Epoch 156/200: Train Loss = 28.3936, Test Loss = 29.3121
Epoch 157/200: 100%|██████████| 782/782 [00:06<00:00, 127.05batch/s, batch_loss=24.3568]
Epoch 157/200: Train Loss = 28.3555, Test Loss = 29.5593
Epoch 158/200: 100%|██████████| 782/782 [00:06<00:00, 128.53batch/s, batch_loss=22.0800]
Epoch 158/200: Train Loss = 28.4448, Test Loss = 29.5359
Epoch 159/200: 100%|██████████| 782/782 [00:06<00:00, 128.72batch/s, batch_loss=30.0427]
Epoch 159/200: Train Loss = 28.5270, Test Loss = 29.3347
Epoch 160/200: 100%|██████████| 782/782 [00:06<00:00, 126.43batch/s, batch_loss=25.2454]
Epoch 160/200: Train Loss = 28.5631, Test Loss = 29.7421
Epoch 161/200: 100%|██████████| 782/782 [00:06<00:00, 126.65batch/s, batch_loss=28.3461]
Epoch 161/200: Train Loss = 28.4397, Test Loss = 29.3556
```

```
Epoch 162/200: 100%|██████████| 782/782 [00:06<00:00, 129.52batch/s, batch_l
oss=25.1726]
Epoch 162/200: Train Loss = 28.3704, Test Loss = 29.3075
Epoch 163/200: 100%|██████████| 782/782 [00:06<00:00, 124.89batch/s, batch_l
oss=25.6764]
Epoch 163/200: Train Loss = 28.3748, Test Loss = 29.4218
Epoch 164/200: 100%|██████████| 782/782 [00:06<00:00, 126.95batch/s, batch_l
oss=30.0406]
Epoch 164/200: Train Loss = 28.4291, Test Loss = 29.1440
Epoch 165/200: 100%|██████████| 782/782 [00:06<00:00, 122.67batch/s, batch_l
oss=30.1017]
Epoch 165/200: Train Loss = 28.3516, Test Loss = 29.7084
Epoch 166/200: 100%|██████████| 782/782 [00:06<00:00, 121.40batch/s, batch_l
oss=32.1146]
Epoch 166/200: Train Loss = 28.3711, Test Loss = 29.3877
Epoch 167/200: 100%|██████████| 782/782 [00:06<00:00, 122.63batch/s, batch_l
oss=29.8067]
Epoch 167/200: Train Loss = 28.3544, Test Loss = 29.2497
Epoch 168/200: 100%|██████████| 782/782 [00:06<00:00, 120.79batch/s, batch_l
oss=29.5556]
Epoch 168/200: Train Loss = 28.2683, Test Loss = 29.1069
Epoch 169/200: 100%|██████████| 782/782 [00:06<00:00, 117.90batch/s, batch_l
oss=26.2129]
Epoch 169/200: Train Loss = 28.3471, Test Loss = 29.4617
Epoch 170/200: 100%|██████████| 782/782 [00:06<00:00, 124.61batch/s, batch_l
oss=28.8677]
Epoch 170/200: Train Loss = 28.3816, Test Loss = 29.1732
Epoch 171/200: 100%|██████████| 782/782 [00:06<00:00, 120.77batch/s, batch_l
oss=32.1267]
Epoch 171/200: Train Loss = 28.3196, Test Loss = 29.4141
Epoch 172/200: 100%|██████████| 782/782 [00:06<00:00, 123.08batch/s, batch_l
oss=26.6752]
Epoch 172/200: Train Loss = 28.2896, Test Loss = 29.0845
Epoch 173/200: 100%|██████████| 782/782 [00:06<00:00, 118.36batch/s, batch_l
oss=26.6290]
Epoch 173/200: Train Loss = 28.2493, Test Loss = 29.4498
Epoch 174/200: 100%|██████████| 782/782 [00:06<00:00, 116.91batch/s, batch_l
oss=34.0339]
Epoch 174/200: Train Loss = 28.3594, Test Loss = 29.1689
Epoch 175/200: 100%|██████████| 782/782 [00:06<00:00, 120.89batch/s, batch_l
oss=24.5987]
Epoch 175/200: Train Loss = 28.2973, Test Loss = 29.4153
Epoch 176/200: 100%|██████████| 782/782 [00:06<00:00, 121.73batch/s, batch_l
oss=23.7926]
Epoch 176/200: Train Loss = 28.2675, Test Loss = 29.4581
Epoch 177/200: 100%|██████████| 782/782 [00:06<00:00, 116.46batch/s, batch_l
oss=27.5193]
Epoch 177/200: Train Loss = 28.2689, Test Loss = 29.2619
Epoch 178/200: 100%|██████████| 782/782 [00:06<00:00, 121.16batch/s, batch_l
oss=27.2733]
Epoch 178/200: Train Loss = 28.2788, Test Loss = 29.2089
```

```
Epoch 179/200: 100%|██████████| 782/782 [00:06<00:00, 122.55batch/s, batch_l
oss=26.6250]
Epoch 179/200: Train Loss = 28.2472, Test Loss = 29.4531
Epoch 180/200: 100%|██████████| 782/782 [00:06<00:00, 118.44batch/s, batch_l
oss=29.3072]
Epoch 180/200: Train Loss = 28.4744, Test Loss = 29.2646
Epoch 181/200: 100%|██████████| 782/782 [00:06<00:00, 125.29batch/s, batch_l
oss=27.3703]
Epoch 181/200: Train Loss = 28.4158, Test Loss = 29.6074
Epoch 182/200: 100%|██████████| 782/782 [00:06<00:00, 115.37batch/s, batch_l
oss=32.7744]
Epoch 182/200: Train Loss = 28.3314, Test Loss = 29.3464
Epoch 183/200: 100%|██████████| 782/782 [00:06<00:00, 115.57batch/s, batch_l
oss=33.6347]
Epoch 183/200: Train Loss = 28.2022, Test Loss = 29.6181
Epoch 184/200: 100%|██████████| 782/782 [00:06<00:00, 120.15batch/s, batch_l
oss=34.4144]
Epoch 184/200: Train Loss = 28.3950, Test Loss = 29.0629
Epoch 185/200: 100%|██████████| 782/782 [00:06<00:00, 124.26batch/s, batch_l
oss=25.7013]
Epoch 185/200: Train Loss = 28.1801, Test Loss = 29.0170
Epoch 186/200: 100%|██████████| 782/782 [00:06<00:00, 123.38batch/s, batch_l
oss=32.6357]
Epoch 186/200: Train Loss = 28.2221, Test Loss = 29.3602
Epoch 187/200: 100%|██████████| 782/782 [00:06<00:00, 119.12batch/s, batch_l
oss=30.0997]
Epoch 187/200: Train Loss = 28.2139, Test Loss = 29.6152
Epoch 188/200: 100%|██████████| 782/782 [00:06<00:00, 112.97batch/s, batch_l
oss=30.4883]
Epoch 188/200: Train Loss = 28.2581, Test Loss = 29.0809
Epoch 189/200: 100%|██████████| 782/782 [00:06<00:00, 121.53batch/s, batch_l
oss=34.4133]
Epoch 189/200: Train Loss = 28.2707, Test Loss = 29.2200
Epoch 190/200: 100%|██████████| 782/782 [00:06<00:00, 117.04batch/s, batch_l
oss=25.2518]
Epoch 190/200: Train Loss = 28.1934, Test Loss = 29.2507
Epoch 191/200: 100%|██████████| 782/782 [00:06<00:00, 128.21batch/s, batch_l
oss=27.9287]
Epoch 191/200: Train Loss = 28.1737, Test Loss = 29.1782
Epoch 192/200: 100%|██████████| 782/782 [00:06<00:00, 125.69batch/s, batch_l
oss=31.1485]
Epoch 192/200: Train Loss = 28.1445, Test Loss = 29.2897
Epoch 193/200: 100%|██████████| 782/782 [00:06<00:00, 121.45batch/s, batch_l
oss=27.0875]
Epoch 193/200: Train Loss = 28.2044, Test Loss = 29.1467
Epoch 194/200: 100%|██████████| 782/782 [00:06<00:00, 122.05batch/s, batch_l
oss=30.0447]
Epoch 194/200: Train Loss = 28.1835, Test Loss = 29.3156
Epoch 195/200: 100%|██████████| 782/782 [00:06<00:00, 117.09batch/s, batch_l
oss=28.4235]
Epoch 195/200: Train Loss = 28.2168, Test Loss = 29.0605
```

```
Epoch 196/200: 100%|██████████| 782/782 [00:06<00:00, 112.50batch/s, batch_l
oss=27.7394]
Epoch 196/200: Train Loss = 28.1893, Test Loss = 29.3848
Epoch 197/200: 100%|██████████| 782/782 [00:06<00:00, 121.65batch/s, batch_l
oss=28.7884]
Epoch 197/200: Train Loss = 28.0918, Test Loss = 29.1283
Epoch 198/200: 100%|██████████| 782/782 [00:06<00:00, 125.15batch/s, batch_l
oss=29.1305]
Epoch 198/200: Train Loss = 28.1542, Test Loss = 29.0672
Epoch 199/200: 100%|██████████| 782/782 [00:05<00:00, 133.32batch/s, batch_l
oss=25.5031]
Epoch 199/200: Train Loss = 28.1392, Test Loss = 29.0869
Epoch 200/200: 100%|██████████| 782/782 [00:06<00:00, 128.71batch/s, batch_l
oss=23.4082]
Epoch 200/200: Train Loss = 28.3234, Test Loss = 29.1086

Total Training Time: 1364.88 seconds
```



Part B

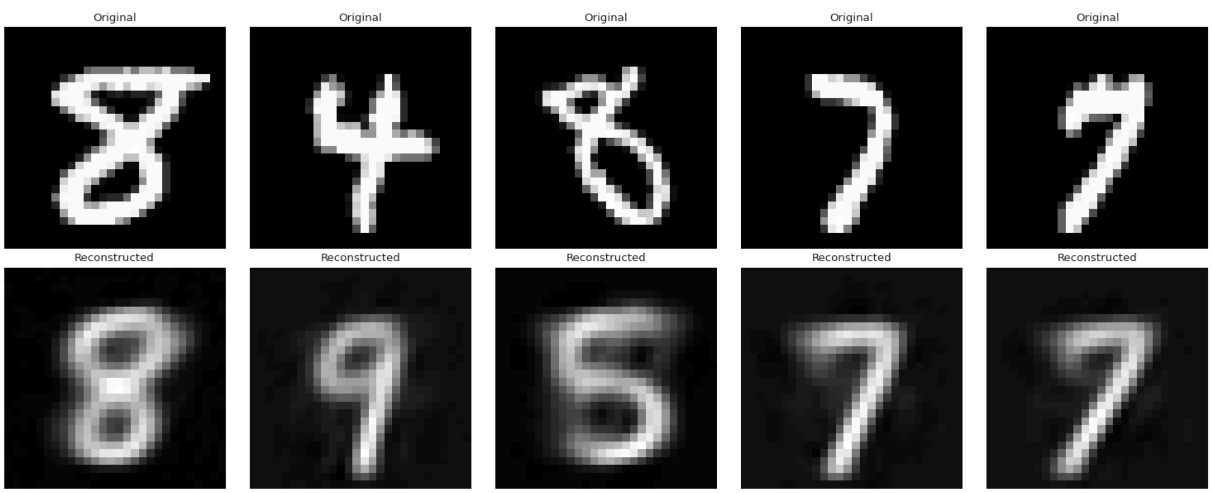
Pick the first five digits in the test set and plot the original and reconstructed images.

Answer:

In [184…

```python
# your code here

# Define a function to plot the original and reconstructed images
def plot_original_and_reconstructed(model, test_loader):
    test_iter = iter(test_loader)
    x, _ = next(test_iter)
```

```
    x = x[:5].numpy()
    x_flat = x.reshape(x.shape[0], -1)

    # Reconstruct the images
    model = eqx.nn.inference_mode(model, value=True)
    x_recon = vmap(model)(x_flat).reshape(-1, 28, 28)


    fig, axes = plt.subplots(2, 5, figsize=(15, 6))

    for i in range(5):
        # Original images
        axes[0, i].imshow(x[i, 0], cmap="gray")
        axes[0, i].axis("off")
        axes[0, i].set_title("Original")

        # Reconstructed images
        axes[1, i].imshow(x_recon[i], cmap="gray")
        axes[1, i].axis("off")
        axes[1, i].set_title("Reconstructed")
    plt.tight_layout()
    plt.show()

# Call the function
plot_original_and_reconstructed(trained_model, test_loader)
```



Part C

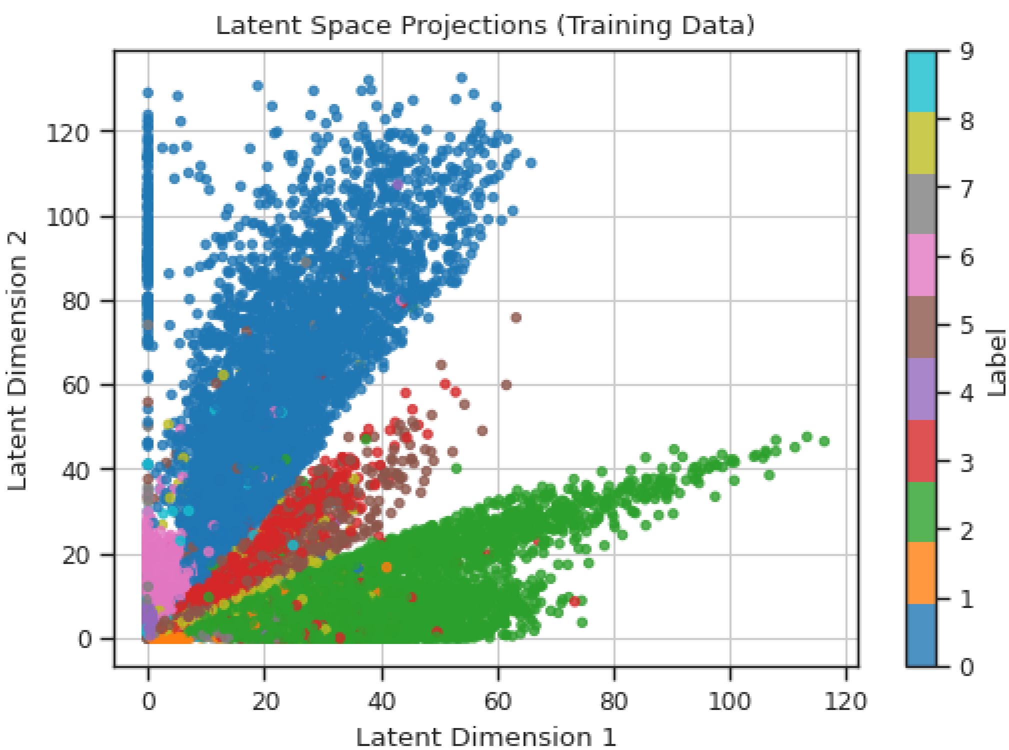
Plot the projections of the digits in the latent space (training and test).

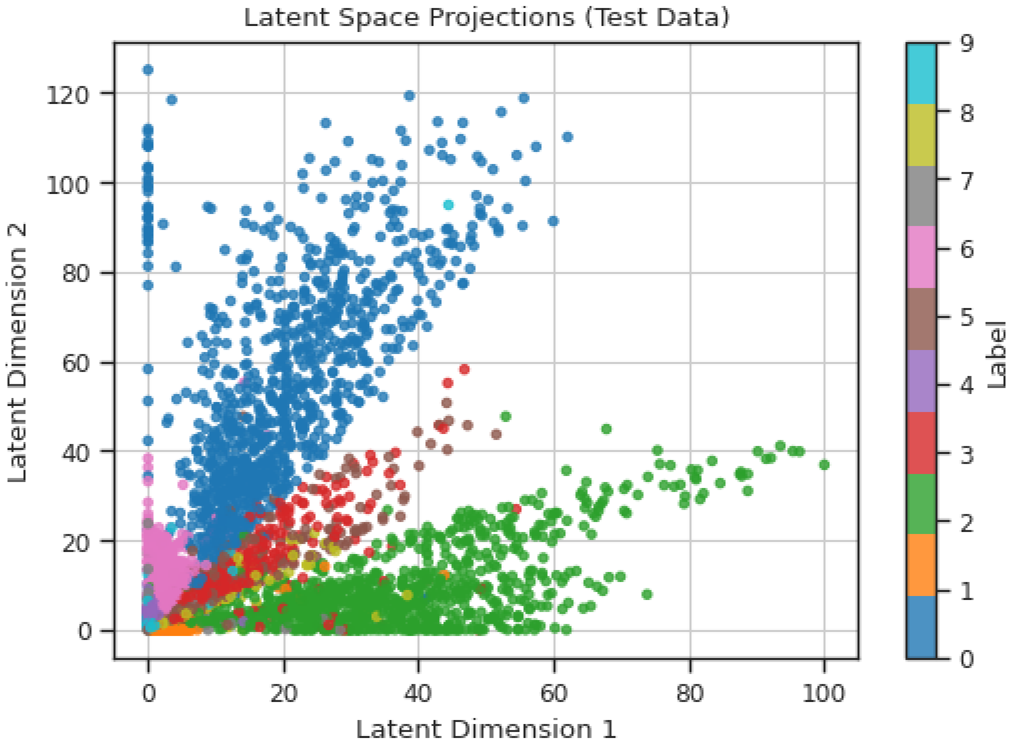
Answer:

In [191…

```
# your code here

def plot_latent_space_projections(model, loader, title):
    # Encode all images in the loader into the latent space
    latent_space = []
    labels = []
```

```
    for x, y in loader:
        x = x.numpy().reshape(x.shape[0], -1)
        latent_space.append(jax.vmap(model.encoder)(x))
        labels.append(y.numpy())

    # labels = jnp.hstack(labels)
    # latent_space = jnp.vstack(latent_space)
    latent_space = jnp.concatenate(latent_space, axis=0)
    labels = jnp.concatenate(labels, axis=0)

    plt.figure(figsize=(6, 4))
    scatter = plt.scatter(latent_space[:, 0], latent_space[:, 1], c=labels,
    plt.colorbar(scatter, label="Label")
    plt.title(f"Latent Space Projections ({title})")
    plt.xlabel("Latent Dimension 1")
    plt.ylabel("Latent Dimension 2")
    plt.grid(True)
    plt.show()

# Plot projections for training and test data
plot_latent_space_projections(trained_model, train_loader, title="Training D
plot_latent_space_projections(trained_model, test_loader, title="Test Data")
```





Part D

Use `scikitlearn` to fit a mixture of Gaussians to the latent space. Use 10 components. Then sample five times from the fitted mixture of Gaussians, reconstruct the samples, and plot the reconstructed images.

Answer:

In [201…

```
# Function to fit a Gaussian Mixture Model (GMM) to the latent space and rec
def fit_gmm_and_reconstruct(loader, model, num_components=10, num_samples=5,

    # same code snippet from previous block
    latent_space = []
    labels = []

    for x, y in loader:
        x = x.numpy().reshape(x.shape[0], -1)
        latent_space.append(jax.vmap(model.encoder)(x))
        labels.append(y.numpy())

    labels = jnp.hstack(labels)
    latent_space = jnp.vstack(latent_space)

    # Fit a Gaussian Mixture Model (GMM) with specified components
    gmm = GaussianMixture(n_components=num_components, random_state=random_s
    gmm.fit(np.array(latent_space))

    # Sample from the GMM and reconstruct the images
    samples, _ = gmm.sample(num_samples)
```

```
    # Decode and plot
    reconstructed_images = jax.vmap(model.decoder)(jnp.array(samples))
    plt.figure(figsize=(10, 2))
    for i, img in enumerate(reconstructed_images):
        plt.subplot(1, num_samples, i + 1)
        plt.imshow(img.reshape(28, 28), cmap='gray')
        plt.axis('off')
    plt.suptitle(f"Reconstructed Images from {num_samples} GMM Samples")
    plt.show()

# Call the function
fit_gmm_and_reconstruct(train_loader, trained_model, num_components=10, num_
```

Reconstructed Images from 5 GMM Samples