ECE-GY 7143 Introduction to Deep Learning

Homework 1

Ali Hamza (ah7072)

February 20, 2025

Question 1

Given that a neuron is defined as follows:

$$y = \sigma(w^T x + b) \quad \text{where} \quad \sigma(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Part (a)

We want to approximate the box function given by:

$$f(x) = \begin{cases} h & \text{if } 0 < x < \delta, \\ 0 & \text{otherwise,} \end{cases}$$

using a neural network with two hidden neurons (defined above) and one linear output neuron. The neural network should output h only when $0 < x < \delta$, and 0 otherwise. We can achieve this by setting the weights and biases of the neurons as follows:

1. Input Layer: The input to the network is x.
2. Hidden Layer:
3. (a) Neuron 1: Set weight $w_1 = 1$ and bias $b_1 = 0$ so that its pre-activation is

$$z_1 = x,$$

and its activation is

$$\sigma(z_1) = \sigma(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}.$$

(b) Neuron 2: Set weight $w_2 = 1$ and bias $b_2 = -\delta$ so that its pre-activation is

$$z_2 = x - \delta,$$

and its activation is

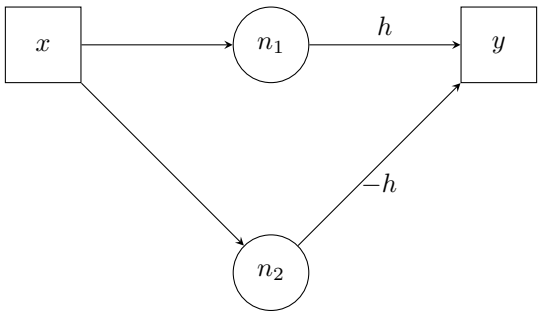
$$\sigma(z_2) = \sigma(x - \delta) = \begin{cases} 1, & x > \delta \\ 0, & x \leq \delta \end{cases}.$$

4. Output Neuron: This neuron simply sums the weighted outputs of the hidden neurons without a nonlinearity. Set its weights as follows:

$$\text{Weight on } \sigma(x): \quad h, \quad \text{and on } \sigma(x - \delta): \quad -h.$$

Thus, the output is given by

$$y = h \cdot \sigma(x) - h \cdot \sigma(x - \delta) = h\big[\sigma(x) - \sigma(x - \delta)\big].$$



Cases:

1. For $x < 0$: $\sigma(x) = 0$ and $\sigma(x - \delta) = 0$ (since $x - \delta < 0$), hence
$$y = h\,(0 - 0) = 0.$$
2. For $0 < x < \delta$: $\sigma(x) = 1$ (since $x > 0$) and $\sigma(x - \delta) = 0$ (since $x - \delta < 0$), hence
$$y = h\,(1 - 0) = h.$$
3. For $x > \delta$: $\sigma(x) = 1$ and $\sigma(x - \delta) = 1$ (since $x - \delta > 0$), hence
$$y = h\,(1 - 1) = 0.$$

Thus, the network outputs h only when $0 < x < \delta$, perfectly recreating the desired box function.

Part (b)

We wish to approximate a smooth, bounded function
$$f : [-B, B] \to \mathbb{R},$$
by representing it as a sum of simple "box functions" that are easy to implement in a neural network. The idea is analogous to forming a Riemann sum.

Step 1: Partitioning the Domain

Divide the interval $[-B, B]$ into N small subintervals of equal width
$$\delta = \frac{2B}{N}.$$
Label the endpoints as
$$x_i = -B + i\delta, \quad \text{for } i = 0, 1, \ldots, N.$$
Over each interval $[x_i, x_i + \delta]$, the function $f(x)$ does not vary much, so we approximate it by a constant value
$$c_i = f\Big(x_i + \frac{\delta}{2}\Big),$$
which is the function value at the midpoint.

Step 2: Representing f as a Sum of Box Functions

Define the indicator (or box) function for the ith subinterval as:
$$\chi_{[x_i, x_i+\delta]}(x) = \begin{cases} 1, & \text{if } x \in [x_i, x_i + \delta], \\ 0, & \text{otherwise.} \end{cases}$$
Then f is approximated by the piecewise constant function:
$$f_{\text{approx}}(x) = \sum_{i=0}^{N-1} c_i\, \chi_{[x_i, x_i+\delta]}(x).$$
This is very similar to the idea behind Riemann sums, where you sum the contributions of small "rectangles" to approximate a function.

Step 3: Realizing Box Functions in a Neural Network

From Part (a) we saw a single box function on an interval $[a, a + \delta]$ can be implemented using two neurons with step activation functions:

- **Neuron 1:** Computes $\sigma(x - a)$. It outputs 1 when $x > a$ (thus "turning on" the box at the left edge).

- **Neuron 2:** Computes $\sigma(x - (a + \delta))$. It outputs 1 when $x > a + \delta$ (thus "turning off" the box at the right edge).

Taking the difference,

$$\sigma(x - a) - \sigma(x - (a + \delta)),$$

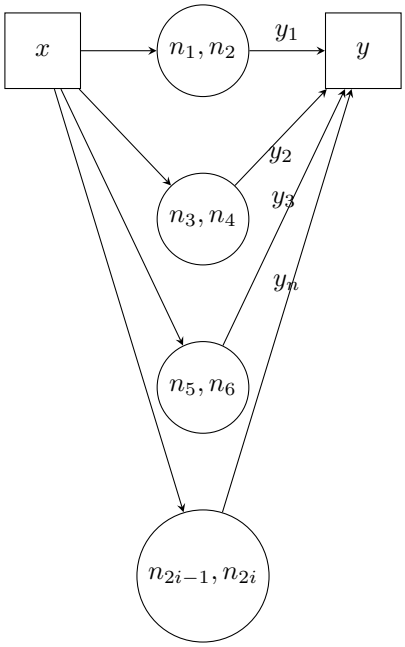
produces 1 exactly when $x \in (a, a + \delta)$ and 0 otherwise. Multiplying by c_i scales the box function to the appropriate height.

Step 4: Combining the Boxes in the Network

The overall network approximates $f(x)$ by summing over the outputs of all these box detectors:

$$y(x) = \sum_{i=0}^{N-1} c_i \left[\sigma(x - x_i) - \sigma(x - (x_i + \delta))\right].$$

As we make δ smaller (i.e., increasing N), the sum becomes a closer approximation to $f(x)$. We can show the neural network as follows:



Where each pair of neurons (n_{2i-1}, n_{2i}) implements a box function over the interval $[x_i, x_i + \delta]$ as described in Part (a) and the output neuron sums the contributions from all these pairs.

Part (c)

The same construction extends to functions $f : \mathbb{R}^d \to \mathbb{R}$ by partitioning a bounded region in $\mathbb{R}^d$ into many small d-dimensional boxes (hyperrectangles).

- Partition each dimension into N segments, creating N^d hyperrectangles.
- For each hyperrectangle R, define an indicator function

$$\chi_R(x) = \begin{cases} 1, & \text{if } x \in R, \\ 0, & \text{otherwise.} \end{cases}$$

- Approximate $f(x)$ by summing over these boxes:

$$f_{\text{approx}}(x) = \sum_{R} c_R \, \chi_R(x),$$

  where c_R is chosen to approximate f in that region.

The number of boxes grows exponentially with d (i.e., N^d). This means that while the method is theoretically sound, the neural network would require an impractically large number of neurons in high-dimensional settings.

Question 2

Given a single fully-connected layer with n_{in} inputs and n_{out} outputs. Each weight W_{ji} is drawn from a Gaussian distribution $\mathcal{N}(0, \sigma^2)$, and the biases are zero. The inputs x_i are i.i.d. with mean 0 and variance v^2. We want to find constraints on σ^2 so that the variance of the neuron outputs does not vanish or explode, and similarly for the backward pass gradients.

Part (a)

Consider a single neuron's output:

$$a_j = \sum_{i=1}^{n_{\text{in}}} W_{ji}\, x_i.$$

Since $W_{ji} \sim \mathcal{N}(0, \sigma^2)$ and x_i are i.i.d. with $\text{Var}(x_i) = v^2$,

$$\mathbb{E}[a_j] = 0,$$

and

$$\text{Var}(a_j) \;=\; \mathbb{E}\Big[a_j^2\Big] \;=\; \sum_{i=1}^{n_{\text{in}}} \mathbb{E}\big[W_{ji}^2\, x_i^2\big] \;=\; \sum_{i=1}^{n_{\text{in}}} \big(\mathbb{E}[W_{ji}^2]\big)\,\big(\mathbb{E}[x_i^2]\big) \;=\; n_{\text{in}}\,\sigma^2\,v^2.$$

Part (b)

If σ^2 is too large, $\text{Var}(a_j)$ grows with n_{in}, leading to exploding activations. If σ^2 is too small, the variance goes to zero with increasing n_{in}. A rough rule to keep the variance at a stable scale is:

$$n_{\text{in}}\,\sigma^2 \approx 1 \quad\Longrightarrow\quad \sigma^2 \approx \frac{1}{n_{\text{in}}}.$$

Part (c)

If the neuron has a ReLU activation, only about half of the inputs (on average) will be positive (assuming a symmetric distribution around 0). This halves the effective variance. Hence, to preserve variance across a ReLU, we need:

$$\frac{1}{2}\, n_{\text{in}}\,\sigma^2 \approx 1 \quad\Longrightarrow\quad \sigma^2 \approx \frac{2}{n_{\text{in}}}.$$

Thus, compared to the linear case, the initialization variance is scaled by a factor of 2.

Part (d)

During backpropagation, gradients to the input layer are computed via

$$\delta_{\text{in}} = W^T\,\delta_{\text{out}},$$

where δ_{out} has variance g^2. A similar derivation shows that

$$\text{Var}(\delta_{\text{in}}) \;=\; n_{\text{out}}\,\sigma^2\,g^2.$$

To avoid exploding or vanishing gradients, we also need

$$n_{\text{out}}\,\sigma^2 \approx 1 \quad\Longrightarrow\quad \sigma^2 \approx \frac{1}{n_{\text{out}}}.$$

Part (e)

To balance both forward and backward stability, we want σ^2 to be on the order of both $1/n_{\text{in}}$ and $1/n_{\text{out}}$. A reasonable choice is the harmonic mean of the two:

$$\sigma^2 \approx \frac{2}{n_{\text{in}} + n_{\text{out}}}.$$

To account for the ReLU activation, we scale this by a factor of 2:

$$\sigma^2 \approx \frac{4}{n_{\text{in}} + n_{\text{out}}}.$$

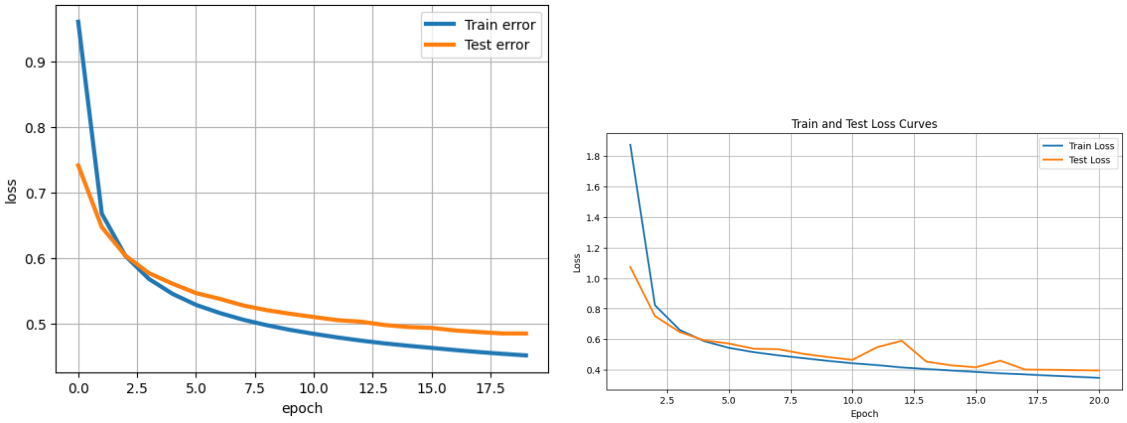
Question 3

We improved the FashionMNIST classifier by replacing a simple logistic regression (a single linear layer) with a deeper network. The new architecture comprises three fully connected hidden layers with 256, 128, and 64 neurons, each followed by a ReLU activation. The details are as follows:

1. **Input Layer:** 784 neurons (28×28 image pixels).
2. **Hidden Layer 1:**
   (a) Fully Connected layer with 256 neurons.
   (b) ReLU activation.
3. **Hidden Layer 2:**
   (a) Fully Connected layer with 128 neurons.
   (b) ReLU activation.
4. **Hidden Layer 3:**
   (a) Fully Connected layer with 64 neurons.
   (b) ReLU activation.
5. **Output Layer:** Fully connected layer with 10 neurons (one per class) followed by softmax activation.

The network was trained using the SGD optimizer (learning rate = 0.001) and CrossEntropyLoss for 20 epochs. The training loss decreased steadily, while the test loss decreased initially and then slightly increased after epoch 10, suggesting some overfitting. The deep network achieved a minimum test loss of 0.3954 and a test accuracy of 85.82%, compared to the logistic regression model's training loss of 0.4515 and test loss of 0.4849.

Figures 1a and 1b display the loss curves for the logistic regression and deep network models, respectively.



(a) Logistic Regression (b) Deep Network

Figure 1: Training and Test Loss Curves

Additionally, predictions on random test images show that the model generally assigns a high probability to the correct class. However, for more complex images with multiple patterns, the confidence is reduced.

Overall, the deep network more effectively captures complex patterns in the FashionMNIST dataset, leading to better generalization and performance compared to logistic regression.

demo01-basics

February 20, 2025

1 PyTorch Basics

I'll assume that everyone is familiar with python. Training neural nets in bare python is somewhat painful, but fortunately there are several well-established libraries which can help. I like pytorch, which is built upon an earlier library called torch. There are many others, including TensorFlow and Jax.

[1]:
```
# We start by importing the libraries we'll use today
import numpy as np
import torch
import torchvision
```

[2]:
```
a = np.random.rand(2,3)
b = torch.from_numpy(a)

print(a)
print(b)
```

```
[[0.33019079 0.79115599 0.53016433]
 [0.19352589 0.52861492 0.95683112]]
tensor([[0.3302, 0.7912, 0.5302],
        [0.1935, 0.5286, 0.9568]], dtype=torch.float64)
```

[3]:
```
print(b + 10.0)
print()
print(torch.sin(b))
print()
print(b.sum())
print()
print(b.mean())
print()
print(b.shape)
```

```
tensor([[10.3302, 10.7912, 10.5302],
        [10.1935, 10.5286, 10.9568]], dtype=torch.float64)

tensor([[0.3242, 0.7112, 0.5057],
        [0.1923, 0.5043, 0.8174]], dtype=torch.float64)
```

```
tensor(3.3305, dtype=torch.float64)

tensor(0.5551, dtype=torch.float64)

torch.Size([2, 3])
```

Torch believes everything is a *tensor.*

The main intuition is that tensors allow for intuitive and efficient matrix multiplication across different indexing dimensions. Soon, we will see that training neural nets basically consits of *forward* and *backward* passes, both of which are essentially matrix multiplies.

The other thing about torch variables is that they (natively) can be differentiated. Again, we'll see why this is important when we learn about backpropagation.

Suppose we want dy/da in the following expression: - $y = a + b$

[4]:
```
a = torch.rand(1,1, requires_grad=True)
b = torch.rand(1,1)
y = a + b
print("a:", a)
print("b:", b)
print("y:", y)
```

```
a: tensor([[0.3312]], requires_grad=True)
b: tensor([[0.0790]])
y: tensor([[0.4102]], grad_fn=<AddBackward0>)
```

Here, y is a function of the input a so we can use PyTorch to compute dy/da

[5]:
```
y.backward()
print("dy/da:", a.grad)
```

```
dy/da: tensor([[1.]])
```

Let's try this again with a more complex function: - $y = a^2 \cdot b$

[6]:
```
a = torch.rand(1,1, requires_grad=True)
b = torch.rand(1,1)
y = (a**2)*b
print("a:", a)
print("b:", b)
print("y:", y)
y.backward()
print("dy/da:", a.grad)
print("dy/da:", 2 * a * b)
```

```
a: tensor([[0.0382]], requires_grad=True)
b: tensor([[0.9591]])
y: tensor([[0.0014]], grad_fn=<MulBackward0>)
dy/da: tensor([[0.0732]])
dy/da: tensor([[0.0732]], grad_fn=<MulBackward0>)
```

Torch has calculated dy/da using backpropagation which is in agreement with our answer calculated using standard differentiation rules.

Here is an example with matrices and vectors:

[7]:
```
A = torch.rand(2,2)
b = torch.rand(2,1)
x = torch.rand(2,1, requires_grad = True)

y = torch.matmul(A, x) + b
z = y.sum()
```

Here, z is a function of the input x. Let us now compute the derivative of z with respect to x using backpropagation.

[8]:
```
z.backward()
print(x)
print(x.grad)
```

```
tensor([[0.7036],
        [0.5930]], requires_grad=True)
tensor([[1.1756],
        [0.7826]])
```

1.1 Training simple models

Let's jump in with our first, simple model. We will train a logistic classifier (equivalent to using a single-layer neural network) on a popular image dataset called *Fashion-MNIST*. Torchvision also has several other image datasets which we can directly load as variables.

[9]:
```
trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/
  ↪',train=True,download=True,transform=torchvision.transforms.ToTensor())
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/
  ↪',train=False,download=True,transform=torchvision.transforms.ToTensor())
```

Let's check that everything has been downloaded.

[10]:
```
print(len(trainingdata))
print(len(testdata))
```

```
60000
10000
```

Let's investigate to see what's inside the dataset.

[11]:
```
image, label = trainingdata[0]
print(image.shape, label)
```

```
torch.Size([1, 28, 28]) 9
```

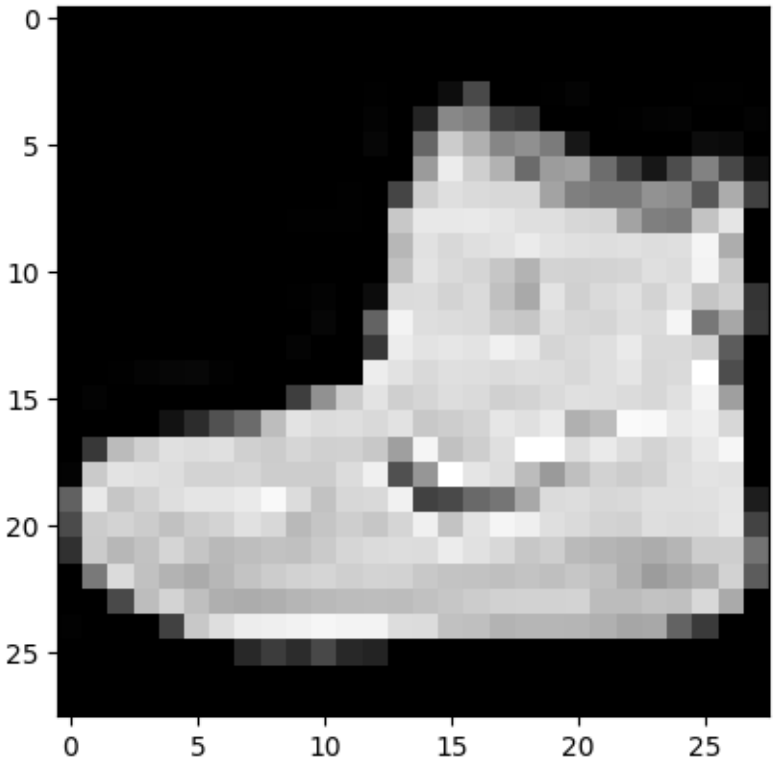
We cannot directly plot the image object given that its first dimension has a size of 1. So we will use the `squeeze` function to get rid of the first dimension.

[12]:
```
print(image.squeeze().shape)
```

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

[13]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

[13]:
```
<matplotlib.image.AxesImage at 0x137b6c1d0>
```



Looks like a shoe? Fashion-MNIST is a bunch of different black and white images of clothing with a corresponding label identifying the category the clothing belongs to. It looks like label 9 corresponds to shoes.

In order to nicely wrap the process of iterating through the dataset, we'll use a dataloader.

[14]:
```
trainDataLoader = torch.utils.data.
 ↪DataLoader(trainingdata,batch_size=64,shuffle=True)
testDataLoader = torch.utils.data.
 ↪DataLoader(testdata,batch_size=64,shuffle=False)
```

Let's also check the length of the train and test dataloader

[15]:
```
print(len(trainDataLoader))
print(len(testDataLoader))
```

```
938
157
```

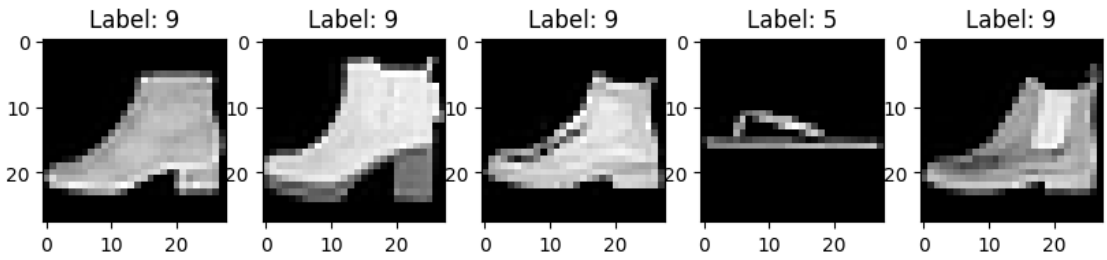
The length here depends upon the batch size defined above. Multiplying the length of our dataloader by the batch size should give us back the number of samples in each set.

[16]:
```
print(len(trainDataLoader) * 64) # batch_size from above
print(len(testDataLoader) * 64)
```

```
60032
10048
```

Now let's use it to look at a few images.

[17]:
```
images, labels = next(iter(trainDataLoader))

plt.figure(figsize=(10,4))
for index in np.arange(0,5):
  plt.subplot(1,5,index+1)
  plt.title(f'Label: {labels[index].item()}')
  plt.imshow(images[index].squeeze(),cmap=plt.cm.gray)
```



Now let's set up our model.

[18]:
```
gpu = "cuda" if torch.cuda.is_available() else "cpu"

class LinearReg(torch.nn.Module):
  def __init__(self):
    super(LinearReg, self).__init__()
    self.linear = torch.nn.Linear(28*28, 10)

  def forward(self, x):
    x = x.view(-1, 28*28) # change so 784 vector instead of 28x28 matrix
    return self.linear(x)

model = LinearReg().to(gpu)
```

```
loss = torch.nn.CrossEntropyLoss() # Step 2: loss
optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # Step 3: training␣
↪method
```

Now let's train our model!

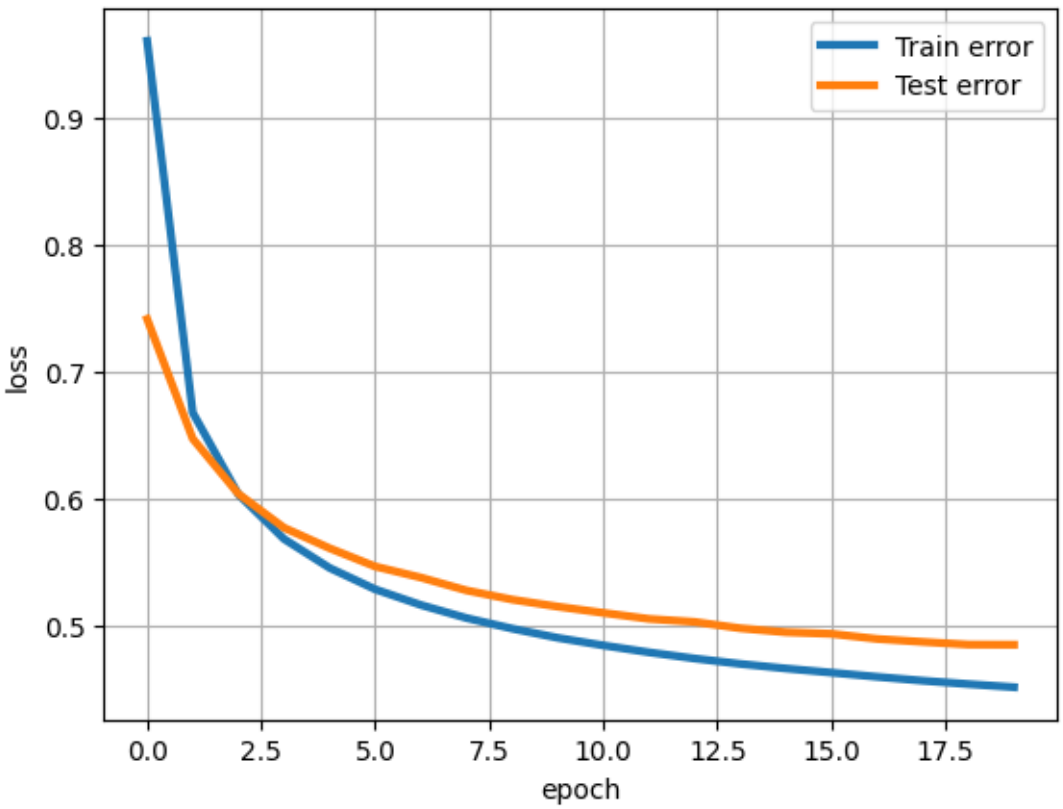
[19]:
```
train_loss_history = []
test_loss_history = []

for epoch in range(20):
  train_loss = 0.0
  test_loss = 0.0

  model.train()
  for i, data in enumerate(trainDataLoader):
    images, labels = data
    images = images.to(gpu)
    labels = labels.to(gpu)
    optimizer.zero_grad() # zero out any gradient values from the previous␣
↪iteration
    predicted_output = model(images) # forward propagation
    fit = loss(predicted_output, labels)  # calculate our measure of goodness
    fit.backward() # backpropagation
    optimizer.step() # update the weights of our trainable parameters
    train_loss += fit.item()

  model.eval()
  for i, data in enumerate(testDataLoader):
    with torch.no_grad():
      images, labels = data
      images = images.to(gpu)
      labels = labels.to(gpu)
      predicted_output = model(images)
      fit = loss(predicted_output, labels)
      test_loss += fit.item()
  train_loss = train_loss / len(trainDataLoader)
  test_loss = test_loss / len(testDataLoader)
  train_loss_history += [train_loss]
  test_loss_history += [test_loss]
  print(f'Epoch {epoch}, Train loss {train_loss}, Test loss {test_loss}')
```

```
Epoch 0, Train loss 0.9601187540142775, Test loss 0.7412131815958934
Epoch 1, Train loss 0.6675267960153409, Test loss 0.6470010595716489
Epoch 2, Train loss 0.6032319129275869, Test loss 0.6036657882723838
Epoch 3, Train loss 0.5681219312554991, Test loss 0.5769736187852872
Epoch 4, Train loss 0.5453312239095346, Test loss 0.5608135051788039
Epoch 5, Train loss 0.5284961068998776, Test loss 0.5464734791950056
Epoch 6, Train loss 0.5161677314274347, Test loss 0.5377659698960128
```

```
Epoch 7, Train loss 0.5058915300378158, Test loss 0.5276417339303691
Epoch 8, Train loss 0.49758261766260875, Test loss 0.5204305840525657
Epoch 9, Train loss 0.49031835348049463, Test loss 0.5148852697223615
Epoch 10, Train loss 0.48436400486525694, Test loss 0.509982881462498
Epoch 11, Train loss 0.4789103496271664, Test loss 0.5052185162996791
Epoch 12, Train loss 0.4740513692151255, Test loss 0.5028605480102977
Epoch 13, Train loss 0.46980230416506846, Test loss 0.49784401647604193
Epoch 14, Train loss 0.4662030571972383, Test loss 0.4947179732428994
Epoch 15, Train loss 0.4629362891795539, Test loss 0.49346096177769316
Epoch 16, Train loss 0.4596298555893176, Test loss 0.4894542811782497
Epoch 17, Train loss 0.45658500818237824, Test loss 0.487122900546736
Epoch 18, Train loss 0.45390113378003205, Test loss 0.48499564380402777
Epoch 19, Train loss 0.4514781088748975, Test loss 0.4848912764506735
```

Let's plot our loss by training epoch to see how we did.

[20]:
```
plt.plot(range(20),train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(range(20),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
plt.show()
```



Why is test loss larger than training loss?

We definitely see some improvement. Let's look at the images, the predictions our model makes and the true label.

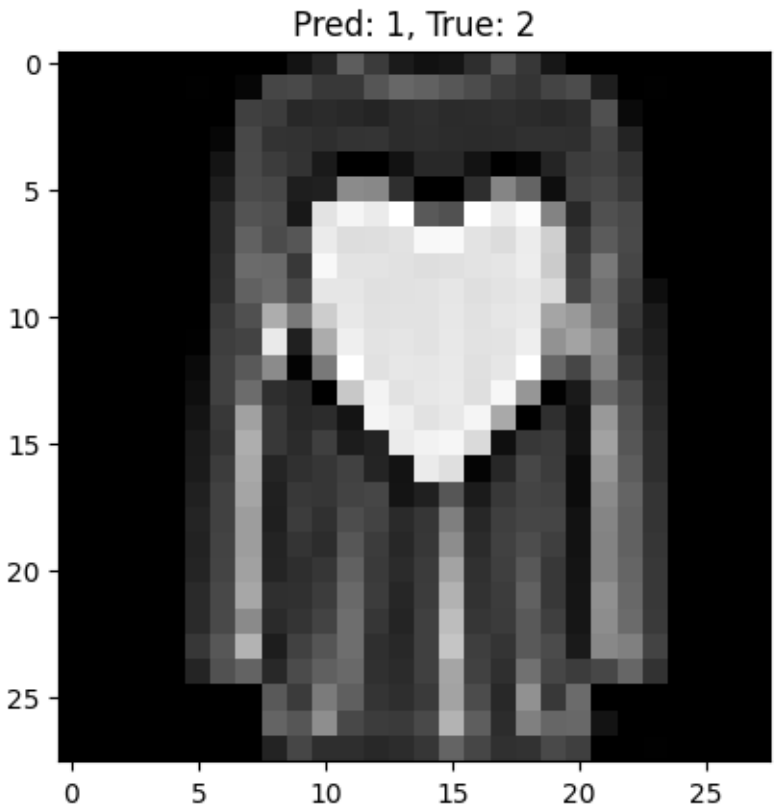
Now for the labels and predicted labels.

[21]:
```
predicted_outputs = model(images)
predicted_classes = torch.max(predicted_outputs, 1)[1]
print('Predicted:', predicted_classes)
fit = loss(predicted_output, labels)
print('True labels:', labels)
print(fit.item())
```

```
Predicted: tensor([3, 1, 7, 5, 8, 2, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5])
True labels: tensor([3, 2, 7, 5, 8, 4, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5])
0.317776620388031
```

[22]:
```
plt.imshow(images[1].squeeze().cpu(), cmap=plt.cm.gray)
plt.title(f'Pred: {predicted_classes[1].item()}, True: {labels[1].item()}')
```

[22]:
```
Text(0.5, 1.0, 'Pred: 1, True: 2')
```



1.2 Training a more complex model

Start with importing the necessary libraries, dataset, split the data into train and test sets, create a dataloade

[36]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import random
# Load FashionMNIST datasets
train_data = torchvision.datasets.FashionMNIST(
    './FashionMNIST/', train=True, download=True, transform=transforms.
 ↪ToTensor()
)
test_data = torchvision.datasets.FashionMNIST(
    './FashionMNIST/', train=False, download=True, transform=transforms.
 ↪ToTensor()
)

# Create DataLoaders
batch_size = 64
gpu = "cuda" if torch.cuda.is_available() else "cpu"
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,␣
 ↪shuffle=True)
test_loader  = torch.utils.data.DataLoader(test_data, batch_size=batch_size,␣
 ↪shuffle=False)
```

1.2.1 Define the model:

- We will use a simple deep neural network with 3 hidden layers with 784 -> 256 -> 128 -> 64 -> 10 neurons.
- We will use ReLU activation function for each layer.
- We will use SGD optimizer with a learning rate of 0.01
- We will use CrossEntropyLoss as our loss function

[37]:
```
class DenseNet(torch.nn.Module):
    def __init__(self):
        super(DenseNet, self).__init__()
        self.flatten = torch.nn.Flatten()  # Flatten 28x28 images into 784-dim␣
 ↪vectors
        self.fc1 = torch.nn.Linear(28*28, 256)
        self.fc2 = torch.nn.Linear(256, 128)
```

```
        self.fc3 = torch.nn.Linear(128, 64)
        self.fc4 = torch.nn.Linear(64, 10)  # 10 output classes
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.relu(self.fc3(x))
        x = self.fc4(x)
        return x

model = DenseNet().to(gpu)

# Loss function and optimizer
criterion = torch.nn.CrossEntropyLoss()  # Loss function
optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # SGD optimizer
```

1.2.2 Training the model:

We will train the model for 20 epochs and evaluate the model on the test set. All the parameters are the same as the previous model, such as batch size, learning rate, optimizer, loss function, etc. The only difference is the model architecture.

- The model is trained for 20 epochs and the loss is calculated for each epoch.
- The model is evaluated on the test set and the accuracy is calculated.

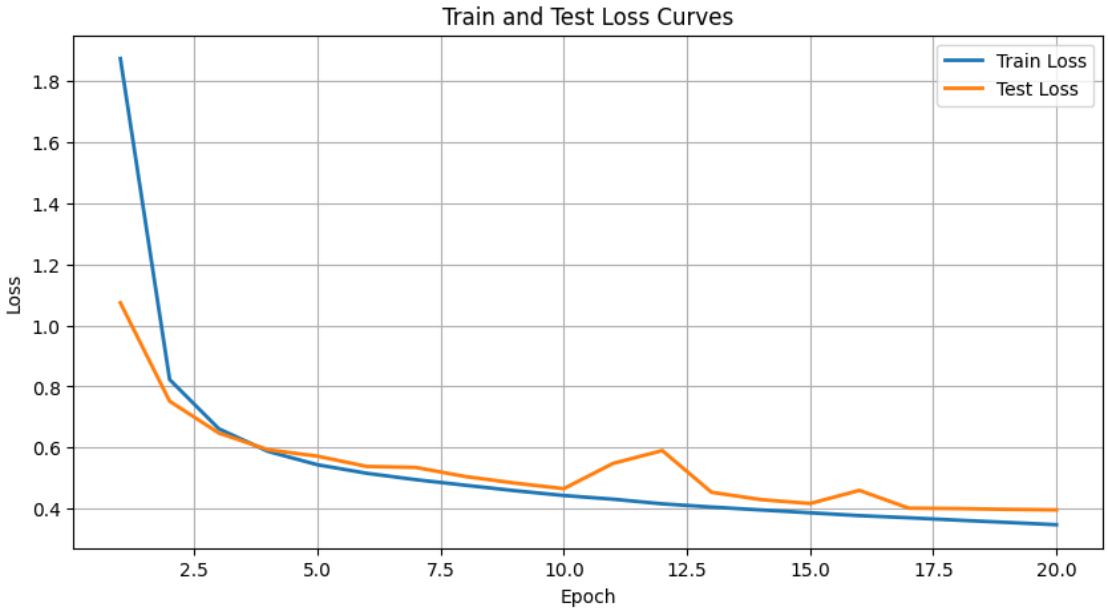
[ ]:
```
# Training loop
num_epochs = 20
train_loss_history = []
test_loss_history = []
test_accuracy_history = []

for epoch in range(num_epochs):
    model.train()
    running_train_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(gpu), labels.to(gpu)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item() * images.size(0)

    train_loss = running_train_loss / len(train_loader.dataset)
    train_loss_history.append(train_loss)
```

```
    # Evaluate on test set
    model.eval()
    running_test_loss = 0.0
    correct = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(gpu), labels.to(gpu)
            outputs = model(images)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item() * images.size(0)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
    test_loss = running_test_loss / len(test_loader.dataset)
    test_loss_history.append(test_loss)
    accuracy = 100 * correct / len(test_loader.dataset)
    test_accuracy_history.append(accuracy)

    print(f"Epoch {epoch} Train Loss = {train_loss:.4f}, Test Loss = {test_loss:
↪.4f}, Test Accuracy = {accuracy:.2f}%")
```

```
Epoch 1/20: Train Loss = 1.8749, Test Loss = 1.0741, Test Accuracy = 58.30%
Epoch 2/20: Train Loss = 0.8227, Test Loss = 0.7516, Test Accuracy = 72.70%
Epoch 3/20: Train Loss = 0.6611, Test Loss = 0.6472, Test Accuracy = 77.30%
Epoch 4/20: Train Loss = 0.5869, Test Loss = 0.5924, Test Accuracy = 79.30%
Epoch 5/20: Train Loss = 0.5434, Test Loss = 0.5716, Test Accuracy = 79.76%
Epoch 6/20: Train Loss = 0.5154, Test Loss = 0.5378, Test Accuracy = 80.87%
Epoch 7/20: Train Loss = 0.4944, Test Loss = 0.5345, Test Accuracy = 81.41%
Epoch 8/20: Train Loss = 0.4762, Test Loss = 0.5048, Test Accuracy = 81.83%
Epoch 9/20: Train Loss = 0.4585, Test Loss = 0.4834, Test Accuracy = 82.62%
Epoch 10/20: Train Loss = 0.4425, Test Loss = 0.4649, Test Accuracy = 83.65%
Epoch 11/20: Train Loss = 0.4304, Test Loss = 0.5478, Test Accuracy = 79.21%
Epoch 12/20: Train Loss = 0.4152, Test Loss = 0.5898, Test Accuracy = 77.75%
Epoch 13/20: Train Loss = 0.4047, Test Loss = 0.4532, Test Accuracy = 83.86%
Epoch 14/20: Train Loss = 0.3951, Test Loss = 0.4291, Test Accuracy = 84.60%
Epoch 15/20: Train Loss = 0.3859, Test Loss = 0.4163, Test Accuracy = 85.28%
Epoch 16/20: Train Loss = 0.3766, Test Loss = 0.4593, Test Accuracy = 83.96%
Epoch 17/20: Train Loss = 0.3696, Test Loss = 0.4012, Test Accuracy = 85.76%
Epoch 18/20: Train Loss = 0.3617, Test Loss = 0.3997, Test Accuracy = 85.92%
Epoch 19/20: Train Loss = 0.3544, Test Loss = 0.3971, Test Accuracy = 85.73%
Epoch 20/20: Train Loss = 0.3468, Test Loss = 0.3954, Test Accuracy = 85.82%
```

[41]:
```
# Plot loss curves
plt.figure(figsize=(10,5))
plt.plot(range(1, num_epochs+1), train_loss_history, label='Train Loss',␣
↪linewidth=2)
```

```
plt.plot(range(1, num_epochs+1), test_loss_history, label='Test Loss',␣
 ↪linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Train and Test Loss Curves")
plt.legend()
plt.grid(True)
plt.show()

# Report final test accuracy
print(f"Final Test Accuracy: {test_accuracy_history[-1]:.2f}%")
```




```
Final Test Accuracy: 85.82%
```

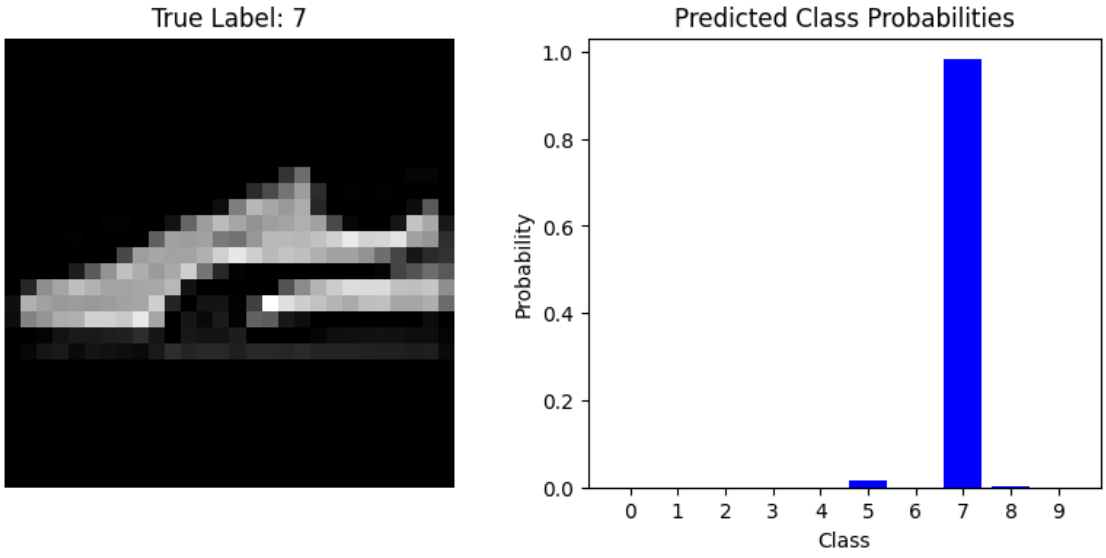
1.2.3 Model Evaluation:

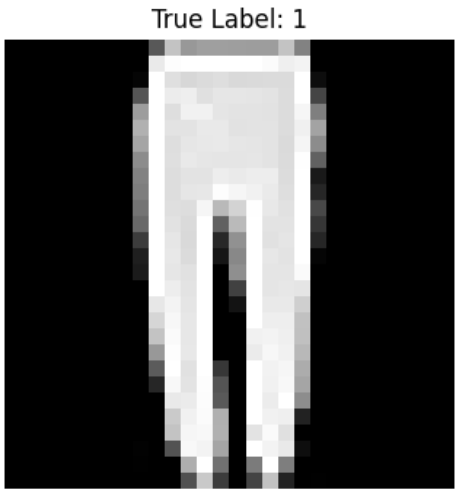
We see that the model is able to achieve an accuracy of 85% on the test set. This is a significant improvement over the previous model. The model is able to learn the features of the images and make accurate predictions.

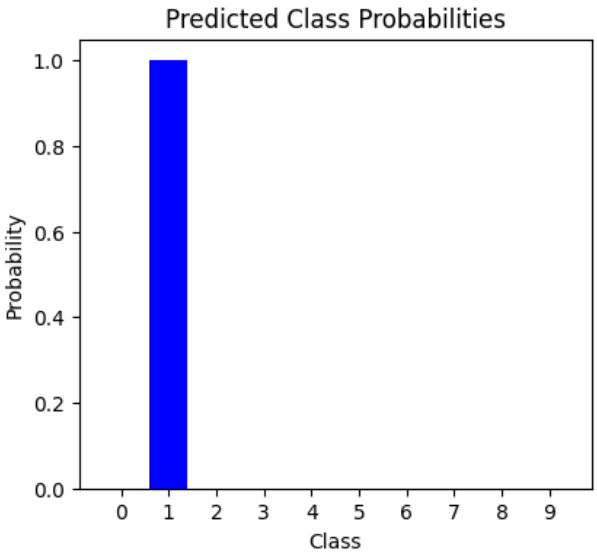
We can further test the model by passing a random image from the test set and checking the prediction made by the model and the true label of the image.

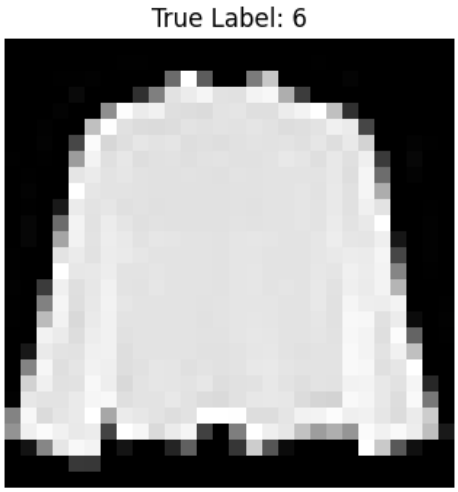
[42]:
```
import random
# Visualize predictions for 3 random test images
model.eval()
for _ in range(3):
    idx = random.randint(0, len(test_data) - 1)
```

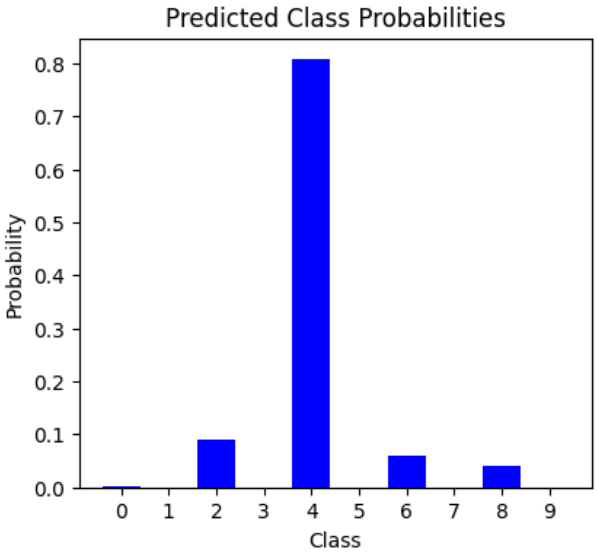
```
image, true_label = test_data[idx]

# Prepare image for the model: add batch dimension and send to device
input_img = image.unsqueeze(0).to(gpu)
output = model(input_img)

# Convert logits to probabilities
probabilities = torch.softmax(output, dim=1).cpu().detach().numpy()[0]

# Plot the image and a bar chart of predicted class probabilities
plt.figure(figsize=(10,4))

# Image subplot
plt.subplot(1,2,1)
plt.imshow(image.squeeze(), cmap='gray')
plt.title(f"True Label: {true_label}")
plt.axis("off")

# Bar chart subplot
plt.subplot(1,2,2)
plt.bar(np.arange(10), probabilities, color='blue')
plt.xlabel("Class")
plt.ylabel("Probability")
plt.title("Predicted Class Probabilities")
plt.xticks(np.arange(10))

plt.show()
```











1.2.4 Conclusion

We can see that changing the model architecture from a simple linear model to a deep neural network with multiple hidden layers has significantly improved the performance of the model. The model is able to learn the features of the images and make accurate predictions.

Some images, such as the last one we tested, are still not predicted with 100% confidence given the complexity of the shapes and patterns in the image. However, the model is able to make accurate predictions on most of the images in the test set.

Question 4

The general structure of the Python Notebook for this question is as follows:

1. Load MNIST dataset using using TensorFlow, and show some sample images.
2. Define `sigmoid, dsigmoid, softmax, cross_entropy_loss` functions.
3. Network setup
    - Two layer neural network with 784 input neurons, 128 hidden neurons, and 10 output neurons.
    - Global list of weights and biases for each layer initialized using gaussian distribution.
4. Feedforward and backpropagation functions.
    - `feed_forward_sample` computes the activations and returns the loss and one-hot prediction for a single sample.
    - `feed_forward_dataset` applies the sample function over all data and reports average loss and accuracy.
    - `train_one_sample`performs a forward pass, computes the loss, then backpropagates errors to compute gradients.
    - Gradients are computed using the chain rule, and the parameters are updated accordingly.
5. Training the network
    - `train_one_epoch` loops through every training sample
    - `test_and_train` runs an epoch then evaluates on the test set.

hw1_s25_p4

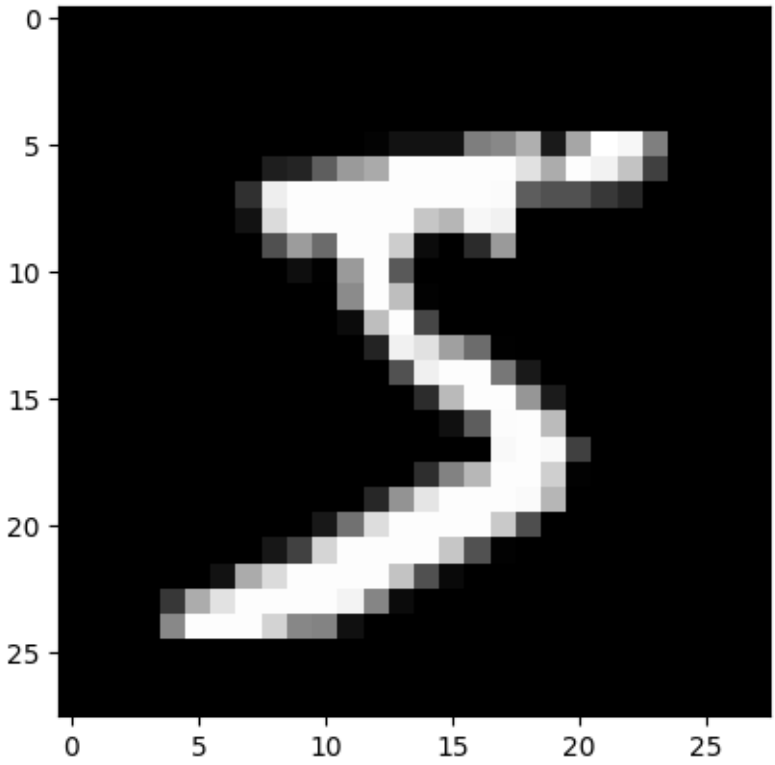
February 20, 2025

In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

[43]:
```
import tensorflow as tf
import matplotlib.pyplot as plt

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.
 ↪load_data(path="mnist.npz")

plt.imshow(x_train[0],cmap='gray');
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

[44]:
```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

  x = np.clip(x, -500, 500) # We get an overflow warning without this

  return np.where(
    x >= 0,
    1 / (1 + np.exp(-x)),
    np.exp(x) / (1 + np.exp(x))
  )

def dsigmoid(x): # Derivative of sigmoid
  return sigmoid(x) * (1 - sigmoid(x))
```

```
def softmax(x):
  # Numerically stable softmax based on (same source as sigmoid)
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/
  b = x.max()
  y = np.exp(x - b)
  return y / y.sum()

def cross_entropy_loss(y, yHat):
  return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max):
  # x: integer to convert to one hot encoding
  # max: the size of the one hot encoded array
  result = np.zeros(10)
  result[x] = 1
  return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance $1/\max(n_{in}, n_{out})$.

0.0.1 Weight and Bias Initialization

- The bias term for each layer is initialized to zero.
- The weights for each layer are initialized by sampling from a normal distribution with mean zero and variance that depends on the size of the layer. We can use various strategies to define the standard deviation of the normal distribution. A common one is to use $1/\sqrt{max}(n_{in}, n_{out})$, where n_{in} and n_{out} are the number of input and output units for the layer. Although, there are a few others that can also be used and may affect model performance. $1/\max n_{in}, n_{out}$ had lower performance in my experiments.

[49]:
```
# # Initialize weights of each layer with a normal distribution of mean 0 and
# # standard deviation 1/sqrt(n), where n is the number of inputs.
# # This means the weighted input will be a random variable itself with mean
# # 0 and standard deviation close to 1 (if biases are initialized as 0,␣
  ↪standard
# # deviation will be exactly 1)

from numpy.random import default_rng

rng = default_rng(80085)
```

```python
import numpy as np

def standard_deviation(n_in, n_out, method='xavier'):
    if method == 'sqrtmax':
        return 1 / np.sqrt(max(n_in, n_out))
    if method == 'max':
        return 1 / max(n_in, n_out)
    if method == 'xavier':
        return 1 / np.sqrt(n_in)
    if method == 'he':
        return 2 / np.sqrt(n_in)
    if method == 'he2':
        return 2 / np.sqrt(n_in + n_out)
    raise ValueError('Invalid method')

# Network architecture: 784 -> 32 -> 32 -> 10
input_size = 28 * 28  # 784
hidden_size = 32
output_size = 10

weights = []
biases = []

# Layer 1: Input (784) -> Hidden1 (32)
n_in, n_out = input_size, hidden_size
std = standard_deviation(n_in, n_out, 'sqrtmax')
weights.append(np.random.randn(n_out, n_in) * std)  # shape: (32,784)
biases.append(np.zeros(n_out))

# Layer 2: Hidden1 (32) -> Hidden2 (32)
n_in, n_out = hidden_size, hidden_size
std = standard_deviation(n_in, n_out, 'sqrtmax')
weights.append(np.random.randn(n_out, n_in) * std)  # shape: (32,32)
biases.append(np.zeros(n_out))

# Layer 3: Hidden2 (32) -> Output (10)
n_in, n_out = hidden_size, output_size
std = standard_deviation(n_in, n_out, 'sqrtmax')
weights.append(np.random.randn(n_out, n_in) * std)  # shape: (10,32)
biases.append(np.zeros(n_out))
```

Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test

accuracy, it should be somewhere around 1/10 (i.e., a random guess).

0.0.2 Forward Pass

In this step, we will implement the forward pass of the neural network for a single sample and for the entire dataset.

The forward pass for the single step is given by the following equations:

$$z_1 = xW_1 + b_1 \tag{1}$$
$$a_1 = \sigma(z_1) \tag{2}$$
$$z_2 = a_1W_2 + b_2 \tag{3}$$
$$a_2 = \sigma(z_2) \tag{4}$$
$$z_3 = a_2W_3 + b_3 \tag{5}$$
$$a_3 = \text{softmax}(z_3) \tag{6}$$

where x is the input, W_i and b_i are the weights and biases for the i-th layer, σ is the sigmoid activation function, and softmax is the softmax activation function.

The softmax function simply applies the exponential function to each element of the input vector and then normalizes the output vector to have a sum of 1

This step simply calculates the inference of the neural network on the test and training data based on the weights and biases that we have initialized.

[50]:
```
def feed_forward_sample(sample, y):
    """
    Forward pass for a single sample.
    Inputs:
      sample: a 2D numpy array (28x28) for the MNIST digit.
      y: integer label (0-9).
    Returns:
      loss: cross entropy loss for this sample.
      one_hot_guess: one-hot encoded prediction (vector of length 10).
    """
    # Flatten the sample into a vector of size 784
    x = sample.flatten()  # shape (784,)

    # Layer 1: hidden layer
    z1 = np.dot(weights[0], x) + biases[0]      # shape (128,)
    a1 = sigmoid(z1)                            # shape (128,)

    # Layer 2: output layer
    z2 = np.dot(weights[1], a1) + biases[1]       # shape (10,)
    a2 = sigmoid(z2)                            # shape (10,)

     # Layer 3 forward
```

```
    z3 = np.dot(weights[2], a2) + biases[2]
    a3 = softmax(z3)

    # Compute loss using cross-entropy; convert y to one-hot
    y_one_hot = integer_to_one_hot(y, output_size)
    loss = cross_entropy_loss(y_one_hot, a3)

    # Get prediction as one-hot vector
    pred_label = np.argmax(a3)
    one_hot_guess = np.zeros(output_size)
    one_hot_guess[pred_label] = 1

    return loss, one_hot_guess


def feed_forward_dataset(x, y):
  losses = np.empty(x.shape[0])
  one_hot_guesses = np.empty((x.shape[0], output_size))

  for i in range(x.shape[0]):
          loss, one_hot_guess = feed_forward_sample(x[i], y[i])
          losses[i] = loss
          one_hot_guesses[i] = one_hot_guess

  y_one_hot = np.zeros((y.size, 10))
  y_one_hot[np.arange(y.size), y] = 1

  correct_guesses = np.sum(y_one_hot * one_hot_guesses)
  correct_guess_percent = format((correct_guesses / y.shape[0]) * 100, ".2f")

  print("\nAverage loss:", np.round(np.average(losses), decimals=2))
  print("Accuracy (# of correct guesses):", correct_guesses, "/", y.shape[0],␣
 ↪"(", correct_guess_percent, "%)")

def feed_forward_training_data():
  print("Feeding forward all training data...")
  feed_forward_dataset(x_train, y_train)
  print("")

def feed_forward_test_data():
  print("Feeding forward all test data...")
  feed_forward_dataset(x_test, y_test)
  print("")

feed_forward_test_data()
```

```
Feeding forward all test data…
```

```
Average loss: 2.43
Accuracy (# of correct guesses): 989.0 / 10000 ( 9.89 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

0.0.3 Model Training: Forward and Backward Pass

- The forward pass computes the predicted output for a given input. – The forward pass implemented here is the same as the one we implemented in the previous step
- The backward pass computes the gradient of the loss with respect to the weights and biases. – The backward pass is implemented here by computing the gradients of the loss with respect to the weights and biases of each layer. The gradients are computed by backpropagating the derivatives from the output layer to the input layer. The gradients are then used to update the weights and biases of the network using the gradient descent algorithm. The equations for the gradients are derived using the chain rule. The equation is as follows:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial W}$$

[51]:
```python
def train_one_sample(sample, y, learning_rate=0.003):
    x = sample.flatten()  # shape: (784,)

    # Forward pass
    z1 = np.dot(weights[0], x) + biases[0]  # (32,)
    a1 = sigmoid(z1)

    z2 = np.dot(weights[1], a1) + biases[1]   # (32,)
    a2 = sigmoid(z2)

    z3 = np.dot(weights[2], a2) + biases[2]   # (10,)
    a3 = softmax(z3)

    y_one_hot = integer_to_one_hot(y, output_size)
    loss = cross_entropy_loss(y_one_hot, a3)

    # Backward pass:
    # Output layer: derivative of softmax-crossentropy
    delta3 = a3 - y_one_hot                  # (10,)
    dW3 = np.outer(delta3, a2)               # (10,32)
    db3 = delta3

    # Backpropagate to second hidden layer:
    delta2 = np.dot(weights[2].T, delta3) * dsigmoid(z2)  # (32,)
    dW2 = np.outer(delta2, a1)                            # (32,32)
```

```
    db2 = delta2

    # Backpropagate to first hidden layer:
    delta1 = np.dot(weights[1].T, delta2) * dsigmoid(z1)  # (32,)
    dW1 = np.outer(delta1, x)                             # (32,784)
    db1 = delta1

    # Update weights and biases
    weights[2] -= learning_rate * dW3
    biases[2] -= learning_rate * db3
    weights[1] -= learning_rate * dW2
    biases[1] -= learning_rate * db2
    weights[0] -= learning_rate * dW1
    biases[0] -= learning_rate * db1

    return loss
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

0.0.4 Model Training: Looping Over the Dataset

The model is trained by looping over the entire training dataset for a fixed number of epochs. The forward and backward passes are computed for each sample in the dataset. The gradients are then used to update the weights and biases of the network using the gradient descent algorithm. The loss is computed for each sample and the average loss is computed for the entire dataset.

[52]:
```
def train_one_epoch(learning_rate=0.003):
    print("Training for one epoch over the training dataset...")
    total_loss = 0.0
    num_samples = x_train.shape[0]

    # Loop through every sample in the training set
    for i in range(num_samples):
        loss = train_one_sample(x_train[i], y_train[i], learning_rate)
        total_loss += loss
        if (i+1) % 10000 == 0:
            print(f"Processed {i+1}/{num_samples} samples")

    avg_loss = total_loss / num_samples
    print("Finished training epoch. Average Loss:", np.round(avg_loss,␣
 ↪decimals=4), "\n")

feed_forward_test_data()

def test_and_train():
  train_one_epoch()
```

```
  feed_forward_test_data()

for i in range(3):
  test_and_train()
```

```
Feeding forward all test data…

Average loss: 2.43
Accuracy (# of correct guesses): 989.0 / 10000 ( 9.89 %)

Training for one epoch over the training dataset…
Processed 10000/60000 samples
Processed 20000/60000 samples
Processed 30000/60000 samples
Processed 40000/60000 samples
Processed 50000/60000 samples
Processed 60000/60000 samples
Finished training epoch. Average Loss: 1.2322

Feeding forward all test data…

Average loss: 1.05
Accuracy (# of correct guesses): 6302.0 / 10000 ( 63.02 %)

Training for one epoch over the training dataset…
Processed 10000/60000 samples
Processed 20000/60000 samples
Processed 30000/60000 samples
Processed 40000/60000 samples
Processed 50000/60000 samples
Processed 60000/60000 samples
Finished training epoch. Average Loss: 0.9951

Feeding forward all test data…

Average loss: 0.96
Accuracy (# of correct guesses): 6773.0 / 10000 ( 67.73 %)

Training for one epoch over the training dataset…
Processed 10000/60000 samples
Processed 20000/60000 samples
Processed 30000/60000 samples
Processed 40000/60000 samples
Processed 50000/60000 samples
Processed 60000/60000 samples
Finished training epoch. Average Loss: 0.9748

Feeding forward all test data…
```

```
Average loss: 0.86
Accuracy (# of correct guesses): 7026.0 / 10000 ( 70.26 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~70% accuracy after 3 epochs.

References

1. 4. Linear Neural Networks for Classification
2. 5. Multilayer Perceptrons
3. GPT O3 used for syntax referencing, report formatting, and general guidance backpropagation.
4. Discussed solutions, concepts, and ideas with Saad Zubairi