Homework 1

Xiaoyan Ouyang

1.(a)

A box function with height h and width δ is defined as:

$$f(x) = \begin{cases} h, & 0 < x < \delta \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

To approximate this using a neural network with two hidden neurons and step activation $\sigma(u)$:

- The first neuron detects when $x > 0$.
- The second neuron detects when $x > \delta$.
- The output neuron computes the difference of these activations, scaled by h.

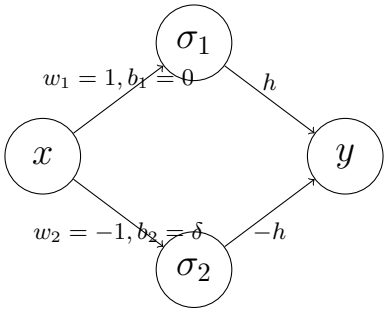
Define the neurons as:

$$z_1 = \sigma(x - 0) = \sigma(x),$$
$$z_2 = \sigma(x - \delta).$$

Then, the output neuron computes:

$$y = h(z_1 - z_2) = h(\sigma(x) - \sigma(x - \delta)). \tag{2}$$

This correctly produces h for $0 < x < \delta$ and 0 otherwise. The network in the following shows all the weights and biases.



1.(b)

For any smooth, bounded function $f(x)$ over $[-B, B]$, we approximate it by summing multiple box functions. We partition $[-B, B]$ into small intervals $[x_i, x_{i+1}]$ of width δ and set the height of each box as $f(x_i)$.

The neural network consists of many neurons, each representing a box function:

$$y = \sum_i h_i(\sigma(x - x_i) - \sigma(x - x_{i+1})). \tag{3}$$

For small δ, the sum of these box functions approximates $f(x)$ closely.

1.(c)

This argument can be extended to d-dimensional inputs by constructing small box functions in $\mathbb{R}^d$ using multiple neurons per dimension. However, practical issues arise:

- The number of neurons grows exponentially with dimension d (curse of dimensionality).
- Efficiently selecting neurons and weights becomes computationally difficult.

In the 1-dimensional case, we only need 2 neurons to construct a Box function. However, in the d-dimensional case, if we want to model a high-dimensional hyperrectangle $[0, \delta_1] \times [0, \delta_2] \times \cdots \times [0, \delta_d]$, each dimension requires 2 neurons (similar to the "start detection" and "end detection" in the 1-dimensional case). Therefore, the total number of neurons is approximately 2^d (as each dimension requires 2 neurons). In the 1-dimensional case, we only need to manually adjust a few weights, but in higher-dimensional space, we need to precisely set weights and biases for each dimension to ensure that the Box function is correctly activated. The total number of connections (weights) also grows exponentially, leading to a significant computational overhead.

2.(a)

Given a fully-connected layer with n_{in} inputs and n_{out} outputs, let the weights be initialized as

$$W_{ij} \sim \mathcal{N}(0, \sigma^2) \tag{4}$$

and assume the inputs x_i are independent, zero-mean random variables with variance

$$\mathrm{Var}(x_i) = v^2. \tag{5}$$

The pre-activation a_j for neuron j is given by:

$$a_j = \sum_{i=1}^{n_{in}} W_{ij} x_i. \tag{6}$$

Since W_{ij} and x_i are independent, so:

$$\mathrm{Var}(a_j) = \sum_{i=1}^{n_{in}} \mathrm{Var}(W_{ij} x_i). \tag{7}$$

Using the fact that $\mathrm{Var}(AB) = \mathrm{Var}(A)\,\mathrm{Var}(B)$ for independent A and B, we get:

$$\mathrm{Var}(a_j) = \sum_{i=1}^{n_{in}} \sigma^2 v^2 = n_{in} \sigma^2 v^2. \tag{8}$$

2.(b)

For stability, we require that $\mathrm{Var}(a_j)$ remains of the same order as v^2. That is,

$$n_{in} \sigma^2 v^2 \approx v^2. \tag{9}$$

Solving for σ^2:

$$\sigma^2 \approx \frac{1}{n_{in}}. \tag{10}$$

If σ^2 is too small, the variance of activations will shrink layer by layer, leading to vanishing activations. If σ^2 is too large, the activations will explode, leading to unstable gradients. Thus, we would better to set:

$$\sigma^2 = \frac{1}{n_{in}}. \tag{11}$$

2.(c)

In the case of the ReLU activation function ($\text{ReLU}(x) = \max(0, x)$), the output distribution of the network changes. A key property of ReLU is that it outputs zero for negative inputs while retaining positive inputs unchanged. When the input follows a symmetric distribution (such as a zero-mean Gaussian distribution), approximately 50% of the neurons' outputs become zero, thereby reducing the variance of the entire layer by half. To maintain the variance of activation values during forward propagation, we have:

$$\sigma^2_{\text{ReLU}} = \frac{2}{n_{\text{in}}}$$

Compared to the case without an activation function (where fully connected layers typically use $\frac{1}{n_{\text{in}}}$), we need to multiply by 2 to compensate for the loss of half of the activation values due to ReLU. Otherwise, the variance signal would decay layer by layer.

2.(d)

The gradients are multiplied by the transpose of W:

$$g_i = \sum_{j=1}^{n_{\text{out}}} W_{ij} g_j$$

Since g_j are independent and $W_{ij} \sim \mathcal{N}(0, \sigma^2)$, using the variance addition formula:

$$\text{Var}(g_i) = \sum_{j=1}^{n_{\text{out}}} \text{Var}(W_{ij} g_j)$$

Since weights and gradients are independent:

$$\text{Var}(W_{ij} g_j) = \text{Var}(W_{ij}) \, \text{Var}(g_j) = \sigma^2 g^2$$

Thus, we obtain:

$$\text{Var}(g_i) = n_{\text{out}} \sigma^2 g^2$$

To prevent gradient vanishing or explosion, we require:

$$\text{Var}(g_i) \approx g^2$$

Substituting into the equation:

$$n_{\text{out}} \sigma^2 g^2 \approx g^2$$

Canceling g^2 (assuming it is nonzero):

$$\sigma^2 \approx \frac{1}{n_{\text{out}}}$$

Therefore, a reasonable initialization strategy is:

$$\sigma^2 = \frac{1}{n_{\text{out}}}$$

This ensures that the variance of gradients remains stable during propagation.

2.(e)

To balance both forward and backward signal propagation, we take the average of the constraints found in parts (b) and (d):

$$\sigma^2 = \frac{2}{n_{in} + n_{out}}. \tag{12}$$

This is the Xavier (Glorot) initialization, ensuring neither vanishing nor exploding signals during training.

Problem_3

February 21, 2025

[ ]:
```
# import the libraries
import numpy as np
import torch
import torchvision
```

[ ]:
```
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())
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/train-images-
idx3-ubyte.gz
```

```
100%|      | 26.4M/26.4M [00:02<00:00, 12.6MB/s]
```

```
Extracting ./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/train-labels-
idx1-ubyte.gz
```

```
100%|      | 29.5k/29.5k [00:00<00:00, 198kB/s]
```

```
Extracting ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz
```

```
100%|      | 4.42M/4.42M [00:01<00:00, 3.72MB/s]
```

```
Extracting ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

100%|      | 5.15k/5.15k [00:00<00:00, 11.9MB/s]

Extracting ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw
```

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

```
60000
10000
```

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

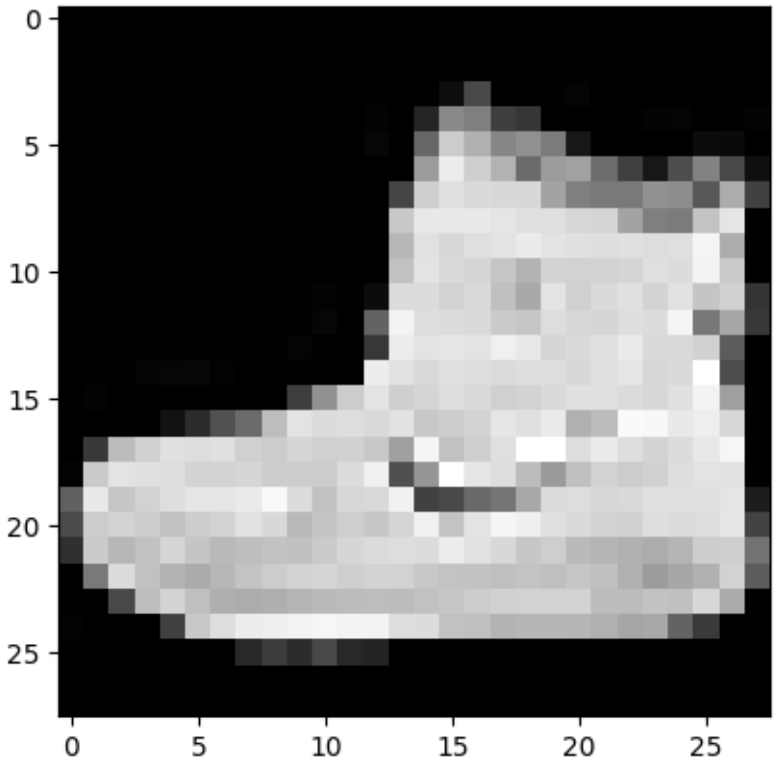
```
torch.Size([1, 28, 28]) 9
```

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

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

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

[ ]:
```
<matplotlib.image.AxesImage at 0x7bcceaa85250>
```



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

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

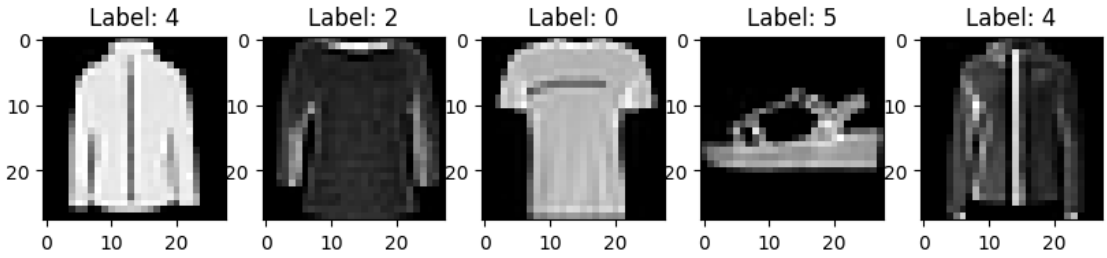
```
938
157
```

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

```
60032
10048
```

[ ]:
```
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)
```



[ ]:
```
class FashionMNISTModel(torch.nn.Module):
  def __init__(self):
    super(FashionMNISTModel,self).__init__()
    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)
    self.relu = torch.nn.ReLU()

  def forward(self, x):
    x = x.view(-1, 28*28)
    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 = FashionMNISTModel().cuda() # Step 1: architecture
loss = torch.nn.CrossEntropyLoss() # Step 2: loss
optimizer = torch.optim.SGD(model.parameters(), lr=0.01) # Step 3: training␣
 ↪method
```

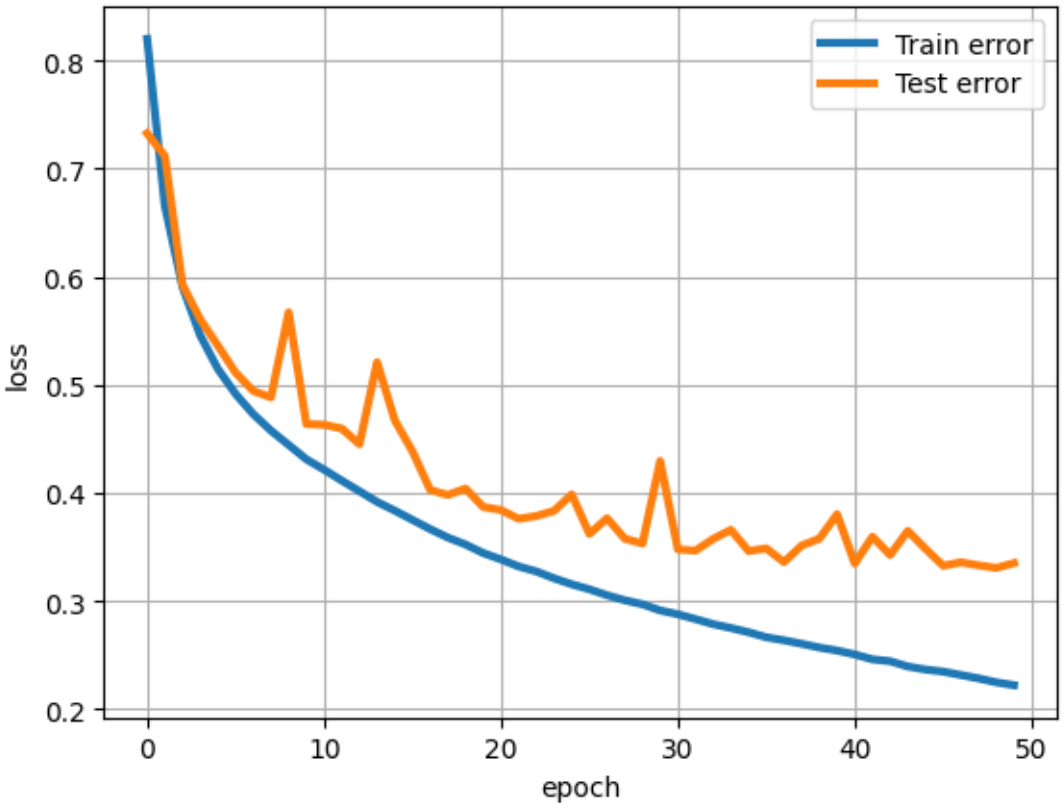
[ ]:
```
train_loss_history = []
test_loss_history = []

for epoch in range(50):
  train_loss = 0.0
  test_loss = 0.0

  model.train()
  for i, data in enumerate(trainDataLoader):
    images, labels = data
    images = images.cuda()
```

```
    labels = labels.cuda()
    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.cuda()
      labels = labels.cuda()
      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 0.8200335381572434, Test loss 0.7326804498198686
Epoch 1, Train loss 0.665718307690834, Test loss 0.7116923011412286
Epoch 2, Train loss 0.5906475426228062, Test loss 0.5926861100515742
Epoch 3, Train loss 0.5457974640863028, Test loss 0.5606379247015449
Epoch 4, Train loss 0.5148079536362752, Test loss 0.5363499848705948
Epoch 5, Train loss 0.49167311339299563, Test loss 0.5114035538047742
Epoch 6, Train loss 0.4729746097663064, Test loss 0.4943873890835768
Epoch 7, Train loss 0.4577874707927836, Test loss 0.48849217717055304
Epoch 8, Train loss 0.4445181418297642, Test loss 0.5671002291570044
Epoch 9, Train loss 0.43125458921133075, Test loss 0.4638637961096065
Epoch 10, Train loss 0.4216652001176816, Test loss 0.46319339448099683
Epoch 11, Train loss 0.411558430713377, Test loss 0.4597173428079884
Epoch 12, Train loss 0.401729068665235, Test loss 0.44516734247374684
Epoch 13, Train loss 0.3918211419445111, Test loss 0.5210342504036655
Epoch 14, Train loss 0.383749128746262, Test loss 0.46723415251750094
Epoch 15, Train loss 0.37521459705539856, Test loss 0.4385564943217927
Epoch 16, Train loss 0.3666893032822273, Test loss 0.4028862781205754
Epoch 17, Train loss 0.35885209242291033, Test loss 0.3982689765038764
Epoch 18, Train loss 0.35241075597210986, Test loss 0.40421634219634306
Epoch 19, Train loss 0.3445079951668218, Test loss 0.3870985234618946
Epoch 20, Train loss 0.3386262595764737, Test loss 0.38436001814474724
Epoch 21, Train loss 0.33214848486980647, Test loss 0.3761763641978525
Epoch 22, Train loss 0.32754341848909474, Test loss 0.3787363739150345
```

```
Epoch 23, Train loss 0.32119926396431697, Test loss 0.3835079055872692
Epoch 24, Train loss 0.3156572524259594, Test loss 0.39882386765282624
Epoch 25, Train loss 0.31103204473503615, Test loss 0.36221614185791867
Epoch 26, Train loss 0.3055283787217476, Test loss 0.3767634172728107
Epoch 27, Train loss 0.3008508968597917, Test loss 0.3578552864729219
Epoch 28, Train loss 0.29715881352898665, Test loss 0.3533400070325584
Epoch 29, Train loss 0.2914532613573171, Test loss 0.42966997471584634
Epoch 30, Train loss 0.2879214670135777, Test loss 0.3479153208292214
Epoch 31, Train loss 0.28345096788839746, Test loss 0.34679541428377675
Epoch 32, Train loss 0.27875598672547064, Test loss 0.3576511265176117
Epoch 33, Train loss 0.2752353979437463, Test loss 0.36603504855921315
Epoch 34, Train loss 0.27129416232869064, Test loss 0.34637003463165017
Epoch 35, Train loss 0.2666461776727552, Test loss 0.3488483873142558
Epoch 36, Train loss 0.2638548987347688, Test loss 0.33612170483276343
Epoch 37, Train loss 0.2606392344956332, Test loss 0.35152637569388007
Epoch 38, Train loss 0.25712381808488355, Test loss 0.35781701440644115
Epoch 39, Train loss 0.25437884713445647, Test loss 0.3804061690903014
Epoch 40, Train loss 0.2507424075593318, Test loss 0.33481416794335006
Epoch 41, Train loss 0.24622550944307212, Test loss 0.35968598581043776
Epoch 42, Train loss 0.2445218058457888, Test loss 0.3427320975501826
Epoch 43, Train loss 0.2396058153662918, Test loss 0.365043097668013
Epoch 44, Train loss 0.23681550292270398, Test loss 0.34848266802016337
Epoch 45, Train loss 0.23487135837835543, Test loss 0.33279177323458303
Epoch 46, Train loss 0.2317561384941787, Test loss 0.3359520827319212
Epoch 47, Train loss 0.2286240428702028, Test loss 0.3331539783697979
Epoch 48, Train loss 0.22498356889703, Test loss 0.3307615016011675
Epoch 49, Train loss 0.2223383083081703, Test loss 0.3352103267031111
```

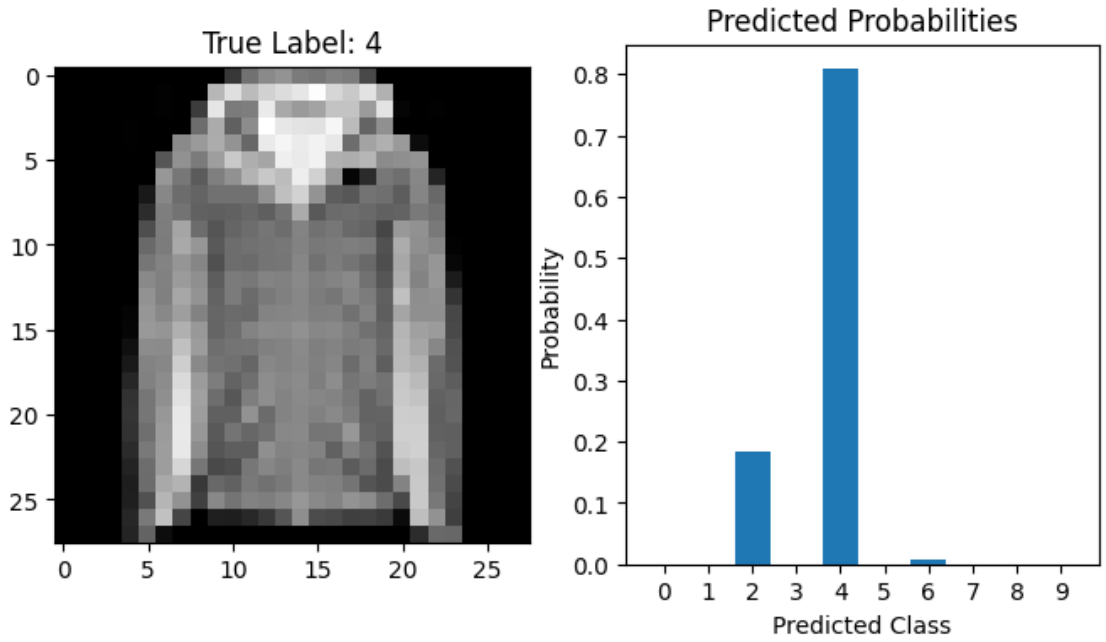
[ ]:
```
plt.plot(range(50),train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(range(50),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
plt.show()
```



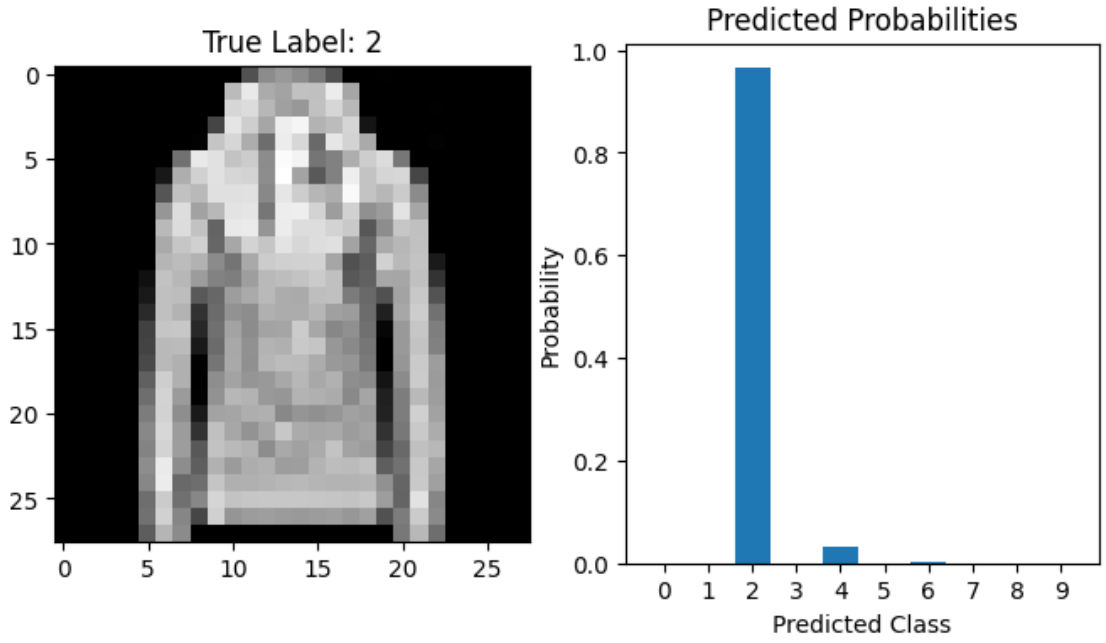
[ ]:
```
# draw any 3 random image samples from the test dataset, visualize the
# predicted class probabilities for each sample, and comment on what you can␣
 ↪observe from these plots.

import matplotlib.pyplot as plt
import random

# Select 3 random indices from the test dataset
random_indices = random.sample(range(len(testdata)), 3)

# Get the images and labels for the selected samples
images = [testdata[i][0] for i in random_indices]
labels = [testdata[i][1] for i in random_indices]

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  #
print(f"Using device: {device}")
model.to(device)

# Get predicted probabilities
model.eval()
with torch.no_grad():
```

```
    predicted_probabilities = [
        torch.softmax(model(image.unsqueeze(0).to(device)), dim=1).squeeze()
         for image in images]

# Visualize and comment on predicted probabilities
for i in range(3):
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.imshow(images[i].squeeze(), cmap=plt.cm.gray)
    plt.title(f"True Label: {labels[i]}")

    plt.subplot(1, 2, 2)
    plt.bar(range(10), predicted_probabilities[i].cpu().numpy())
    plt.xticks(range(10))
    plt.xlabel("Predicted Class")
    plt.ylabel("Probability")
    plt.title(f"Predicted Probabilities")
    plt.show()

    print(f"Observation for sample {i+1}:")
    print(f"The model predicts class {torch.argmax(predicted_probabilities[i])}␣
 ↪with the highest probability.")
    print(f"The true label is {labels[i]}.")
    print("-"*20)
```

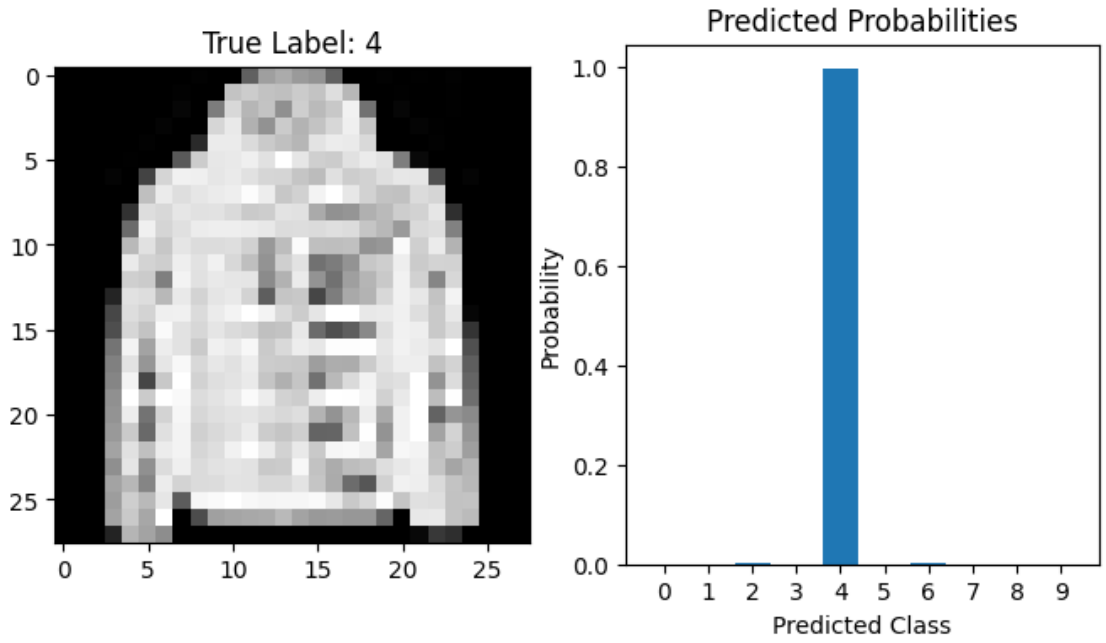
```
Using device: cuda
```



```
Observation for sample 1:
The model predicts class 4 with the highest probability.
The true label is 4.
--------------------
```




```
Observation for sample 2:
The model predicts class 2 with the highest probability.
The true label is 2.
--------------------
```




```
Observation for sample 3:
The model predicts class 4 with the highest probability.
The true label is 4.
--------------------
```

Comment on these plots:

1. The training loss decreases gradually as the epoch increases.
2. From the loss curves, the training loss continues to decrease while the testing loss tends to level off, indicating that the model generalizes well on the test set.
3. On 3 randomly selected test images, we visualized the model's category probability distribution, showing its strong categorization ability.

Problem_4

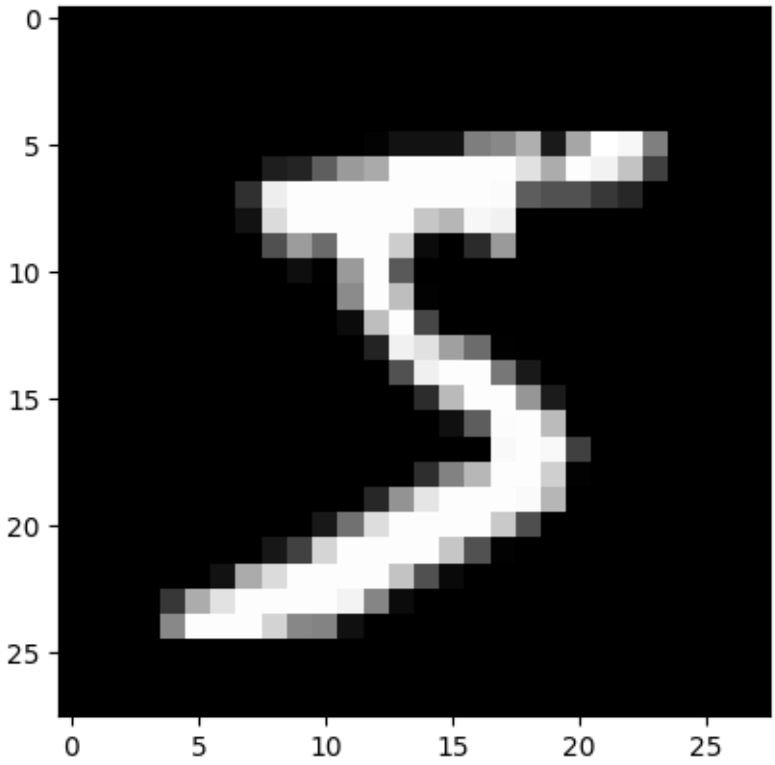
February 21, 2025

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.

[16]:
```
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.

[17]:
```
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 $1/\max(n_{in}, n_{out})$.

[18]:
```
import math

# 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)

# Q1. Fill initialization code here.

# the number of neurons
n_input = 784
n_hidden1 = 32
n_hidden2 = 32
n_output = 10

# weight
```

```
weights = [
    rng.normal(0, 1/math.sqrt(n_input), (n_input, n_hidden1)),
    rng.normal(0, 1/math.sqrt(n_hidden1), (n_hidden1, n_hidden2)),
    rng.normal(0, 1/math.sqrt(n_hidden2), (n_hidden2, n_output))
]
# bias
biases = [
    np.zeros(n_hidden1),
    np.zeros(n_hidden2),
    np.zeros(n_output)
]
```

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).

[19]:
```
def feed_forward_sample(sample, y):
  """ Forward pass through the neural network.
    Inputs:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: the cross entropy loss, most likely class
  """
  # Q2. Fill code here.

  z1 = np.dot(sample, weights[0]) + biases[0]
  a1 = sigmoid(z1)

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

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

  # compute the cross entropy loss, most likely class
  loss = cross_entropy_loss(integer_to_one_hot(y, 10), a3)
  one_hot_guess = integer_to_one_hot(np.argmax(a3), 10)


  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], 10))

  # Q2. Fill code here to calculate losses, one_hot_guesses
  # iterate each sample
  for i in range(x.shape[0]):
    sample = x[i].flatten()
    losses[i], one_hot_guesses[i] = feed_forward_sample(sample, y[i])

  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: 2.37
Accuracy (# of correct guesses): 880.0 / 10000 ( 8.80 %)
```

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.

[20]:
```
def train_one_sample(sample, y, learning_rate=0.003):
  a = sample.flatten()

  # We will store each layer's activations to calculate gradient
```

```
activations = []
zs = []

# Forward pass

# Q3. This should be the same as what you did in feed_forward_sample above.
for i in range(len(weights)):
  z = np.dot(a, weights[i]) + biases[i]
  a = sigmoid(z)
  zs.append(z)
  activations.append(a)

# compute the loss
y_one_hot = integer_to_one_hot(y, 10)
loss = cross_entropy_loss(y_one_hot, a)

# Backward pass

# Q3. Implement backpropagation by backward-stepping gradients through each␣
↪layer.
# You may need to be careful to make sure your Jacobian matrices are the␣
↪right shape.
# At the end, you should get two vectors: weight_gradients and bias_gradients.
weight_gradients = []
bias_gradients = []

# the error in the output layer
delta = activations[-1] - y_one_hot

for i in reversed(range(len(weights))):
  if i > 0:
    prev_activation = activations[i-1] # the activation of the previous layer
  else:
    prev_activation = sample

  prev_activation = prev_activation.reshape(-1, 1)

  weight_grad = np.dot(prev_activation, delta.reshape(1, -1))
  bias_grad = delta

  weight_gradients.insert(0, weight_grad)
  bias_gradients.insert(0, bias_grad)

  # Error delta for current layer i
  if i > 0:
    delta = np.dot(delta, weights[i].T) * dsigmoid(zs[i-1]).reshape(1, -1)
```

```
  # Update weights & biases based on your calculated gradient
  for i in range(len(weights)):
    weights[i] -= weight_gradients[i] * learning_rate
    biases[i] -= bias_gradients[i].flatten() * learning_rate
```

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

Q4. Train your model for 3 epochs.

[21]:
```
def train_one_epoch(learning_rate=0.003):
  print("Training for one epoch over the training dataset...")

  # Q4. Write the training loop over the epoch here.
  for i in range(x_train.shape[0]):
    sample = x_train[i]
    y = y_train[i]
    train_one_sample(sample, y, learning_rate)

  print("Finished training.\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: 2.37
Accuracy (# of correct guesses): 880.0 / 10000 ( 8.80 %)

Training for one epoch over the training dataset…
Finished training.

Feeding forward all test data…

Average loss: 1.02
Accuracy (# of correct guesses): 6541.0 / 10000 ( 65.41 %)

Training for one epoch over the training dataset…
Finished training.

Feeding forward all test data…
```

```
Average loss: 1.0
Accuracy (# of correct guesses): 6565.0 / 10000 ( 65.65 %)

Training for one epoch over the training dataset…
Finished training.

Feeding forward all test data…

Average loss: 1.06
Accuracy (# of correct guesses): 6455.0 / 10000 ( 64.55 %)
```

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.

Attributes & Values for Satyrium spini (Spine Hairstreak):

- **Name**: Satyrium spini (Spine Hairstreak)
- **Wingspan**: 1.5 to 2 inches (3.8 to 5.1 cm)
- **Color**: Dark brown or black with blue and orange markings
- **Underside of Wings**: Mottled brown with orange and white markings
- **Distinctive Features**: Spine-like projection at the end of the hind wing
- **Habitat**: Forests, fields, and gardens
- **Nectar Sources**: Variety of flowers
- **Attracted to**: Bright, open areas