Assignment – 1

Submitted By – Shambhavi Seth(ss17936)

1. **(4 points)** *Expressivity of neural networks*. Recall that the functional form for a single neuron is given by $y = \sigma(\langle w, x \rangle + b, 0)$, where x is the input and y is the output. In this exercise, assume that x and y are 1-dimensional (i.e., they are both just real-valued scalars) and σ is the unit step activation. We will use multiple layers of such neurons to approximate pretty much any function f. There is no learning/training required for this problem; you should be able to guess/derive the weights and biases of the networks by hand.

    a. (1pt) A *box* function with height h and width δ is the function $f(x) = h$ for $0 < x < \delta$ and 0 otherwise. Show that a simple neural network with 2 hidden neurons with step activations can realize this function. Draw this network and identify all the weights and biases. (Assume that the output neuron only sums up inputs and does not have a nonlinearity.)

    b. (2pts) Now suppose that f is *any arbitrary, smooth, bounded* function defined over an interval $[-B, B]$. (You can ignore what happens to the function outside this interval, or just assume it is zero). Use part a to show that this function can be closely approximated by a neural network with a hidden layer of neurons. You don't need a rigorous mathematical proof here; a handwavy argument or even a figure is okay here, as long as you convey the right intuition.

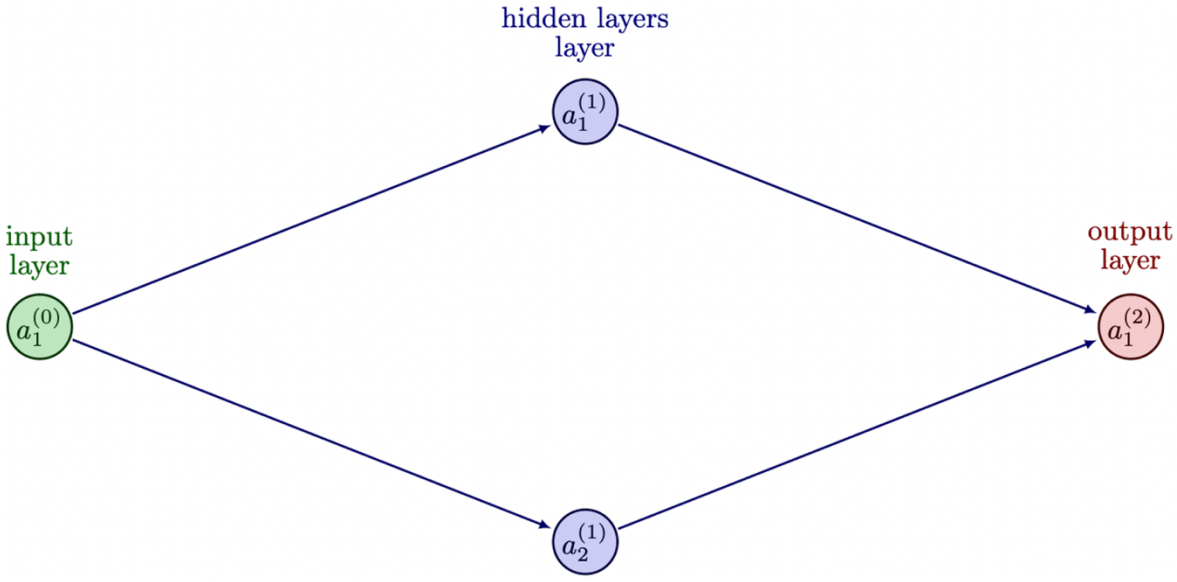
    c. (1pt) Do you think the argument in part b can be extended to the case of d-dimensional inputs? (i.e., where the input x is a vector – think of it as an image, or text query, etc). If yes, comment on potential practical issues involved in defining such networks. If not, explain why not.

Solution 1 -

Given a box function with height h and width δ, where $f(x) = h$ for $0 < x < \delta$, we can represent this function using a neural network with 2 hidden neurons with step activations.

- The first hidden node $a_1^{(1)}$ takes $a_1^{(0)}$ as input and uses a weight (w_1) and bias (b_1) and applies a step function to the same.
- The second hidden node $a_2^{(1)}$ takes $a_1^{(0)}$ as input and uses a weight (w_2) and bias (b_2) and applies a step function to the same.
- The final output layer $a_1^{(2)}$ aggregates output from the two hidden nodes $a_1^{(1)}$ and $a_2^{(1)}$ as input with weights w_3 and w_4 respectively and a bias b_3 and applies a step function to the same.

The neural network represented below can accurately represent the above function:



Suppose b1 = 0, w1 = 1 and $b2 = \delta$, w2 = -1, w3 = 1 and w4 = 1

Activation function for the hidden nodes is $f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

Activation function for the output node is $f(x) = \begin{cases} h & \text{if } x > 1.5 \\ 0 & \text{otherwise} \end{cases}$

The outputs from each node would be -
$y_1 = \sigma(w_1 x_1 + b_1)$
$y_2 = \sigma(w_2 x_2 + b_2)$
$y = \sigma(w_3 y_1 + w_4 y_2)$

For $x < 0$ condition, let $x = -1$:

$$\begin{aligned} y_1 &= \sigma(-1 + 0) = \sigma(-1) = 0 \\ y_2 &= \sigma(1 + \delta) \quad \text{as } \delta > 0, \text{ therefore } y_2 = 1 \\ y &= \sigma(1 + 0) = \sigma(1) = 0 \quad \text{as } 1 < 1.5 \end{aligned}$$

For $x > \delta$ condition, let $x = 2\delta$:

$$\begin{aligned} y_1 &= \sigma(2\delta + 0) \quad \text{as } \delta > 0, \text{ therefore } y_1 = 1 \\ y_2 &= \sigma(-2\delta + \delta) = \sigma(-\delta) = 0 \\ y &= \sigma(1 + 0) = \sigma(1) = 0 \quad \text{as } 1 < 1.5 \end{aligned}$$

For $0 < x < \delta$ condition, let $x = \frac{\delta}{2}$:

$$\begin{aligned} y_1 &= \sigma\left(\frac{\delta}{2} + 0\right) = \sigma\left(\frac{\delta}{2}\right) \quad \text{as } \frac{\delta}{2} > 0, \text{ therefore } y_1 = 1 \\ y_2 &= \sigma\left(-\frac{\delta}{2} + \delta\right) = \sigma\left(\frac{\delta}{2}\right) \quad \text{as } \frac{\delta}{2} > 0, \text{ therefore } y_2 = 1 \\ y &= \sigma(1 + 1) = \sigma(2) = h \quad \text{as } 2 > 1.5 \end{aligned}$$

Hence $f(x) = h$ for all $0 < x < \delta$ and 0 otherwise.

(b) The given function f, which is a smooth, bounded, and arbitrary function, defined over an interval $[-B, B]$, can be divided into sub-intervals. These sub-intervals can be approximated using the box function, as given in part (a). Assuming the interval $[-B, B]$ is divided into n sub-intervals of width $\delta = \frac{2B}{n}$, then the function can be approximated over each sub-interval using an n-neuron neural network with step activations and two hidden neurons, similar to what was done in part (a).

We adjust weights and biases to obtain the best approximation. These sub-networks can then be connected together to approximate the entire function. The resulting neural network will have n sub-networks, with each sub-network having two hidden neurons and an output neuron to sum up the output of each sub-network.

(c) Yes, we can extend the argument in part (b) for the case of d-dimensional inputs. We need to follow the same approach as we did in part (b) where we can create sub-sections of the d-dimensional input to approximate the given function using the neural network defined in part (a). Finally, we can aggregate all such individual models on subsections of the d-dimensional input to produce the final output. The potential practical issues associated with doing this is that it can become computationally too expensive as the number of dimensions increase thereby increasing the number of sub-sections and hence the number of individual neural networks with 2 hidden neurons. Also, if the dimensionality increases the number of training samples could increase exponentially therefore increasing the chances of overfitting on the data. This might further lead to poor results on the test data.

2. **(3 points) Algebraic exercises with gradients**. Suppose that z is a vector with n elements. We would like to compute the gradient of $y = \text{softmax}(z)$. Show that the Jacobian of y with respect to z, J, is given by the expression:

$$J_{ij} = \frac{\partial y_i}{\partial z_j} = y_i(\delta_{ij} - y_j)$$

where δ_{ij} is the Dirac delta, i.e., 1 if $i = j$ and 0 else. *Hint: Your algebra could be simplified if you try computing the log derivative,* $\frac{\partial \log y_i}{\partial z_j}$.

Solution 2 –

The softmax function is given by:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

where $\mathbf{z}$ is a vector of real numbers, and N is the number of elements in the vector.

The Jacobian matrix for the softmax function is given by:

$$\text{J} = \begin{bmatrix} \frac{\partial(\mathbf{s_1})}{\partial z_1} & \frac{\partial(\mathbf{s_1})}{\partial z_2} & \cdots & \frac{\partial(\mathbf{s_1})}{\partial z_N} \\ \frac{\partial(\mathbf{s_2})}{\partial z_1} & \frac{\partial(\mathbf{s_2})}{\partial z_2} & \cdots & \frac{\partial(\mathbf{s_2})}{\partial z_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial(\mathbf{s_N})}{\partial z_1} & \frac{\partial(\mathbf{s_N})}{\partial z_2} & \cdots & \frac{\partial(\mathbf{s_n})}{\partial z_N} \end{bmatrix}$$

where

$$s_i = \frac{e^{z_i}}{\sum_{l=1}^{n} e^{z_l}}$$

Since the outputs of the softmax function are strictly positive values, we can take the partial derivative of the log of the output:

$$\frac{\partial \log s_i}{\partial z_j} = \frac{1}{s_i} \frac{\partial s_i}{\partial z_j}$$

$$\frac{\partial s_i}{\partial z_j} = s_i \frac{\partial \log s_i}{\partial z_j}$$

As

$$s_i = \frac{e^{z_i}}{\sum_{l=1}^{n} e^{z_l}}$$

Therefore,

$$\log(s_i) = \log \frac{e^{z_i}}{\sum_{l=1}^{n} e^{z_l}}$$
$$= z_i - \log(\sum_{l=1}^{n} e^{z_l})$$

The partial derivative of the above expression would be -

$$\frac{\partial \log s_i}{\partial z_j} = \frac{\partial z_i}{\partial z_j} - \frac{\partial \log(\sum_{l=1}^{n} e^{z_l})}{\partial z_j}$$

$\frac{\partial z_i}{\partial z_j}$ can be represented as the Dirac delta δ_{ij} and $\frac{\partial z_i}{\partial z_j}$ is 1 if i=j and 0 otherwise

For the second term of the above equation,

$$\frac{\partial \log s_i}{\partial z_j} = \delta_{ij} - \frac{1}{\sum_{l=1}^{n} e^{z_l}} \frac{\partial(\sum_{l=1}^{n} e^{z_l})}{\partial z_j}$$

By the property of natural logarithm,

$$\frac{\partial \log s_i}{\partial z_j} = \frac{1}{s_i}$$

$$\frac{\partial(\sum_{l=1}^{n} e^{z_l})}{\partial z_j} = \frac{\partial(e^{z_1} + e^{z_2} + e^{z_j} + e^{z_n})}{\partial z_j} = \frac{\partial(e^{z_j})}{\partial z_j} = e^{z_j}$$

$$= \frac{\partial log s_i}{\partial z_j} = \delta_{ij} - s_j$$

Multiplying this equation by s,

$$= \frac{\partial s_i}{\partial z_j} = s_i(\delta_{ij} - s_j)$$

We thus have both the diagonal and off diagonal formula for Jacobian.

3. (3 points) Improving the FashionMNIST classifier. In the first demo, we trained a simple logistic regression model to classify MNIST digits. Repeat the same experiment, but now use a (dense) neural network with three (3) hidden layers with 256, 128, and 64 neurons respectively, all with ReLU activations. Display train- and test- loss curves, and report test accuracies of your final model. You may have to tweak the total number of training epochs to get reasonable accuracy. Finally, draw any 3 image samples from the test dataset, visualize the predicted class probabilities for each sample, and comment on what you can observe from these plots.

Solution-

[1]:
```
import numpy as np
import torch
import torchvision
```

[2]:
```
#Getting the FashionMNIST dataset
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%|      | 26421880/26421880 [00:01<00:00, 17350500.90it/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%|      | 29515/29515 [00:00<00:00, 302463.25it/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%|      | 4422102/4422102 [00:00<00:00, 5523979.69it/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%|      | 5148/5148 [00:00<00:00, 13614298.23it/s]

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

[3]:
```
print(len(trainingdata))
print(len(testdata))
```

```
60000
10000
```

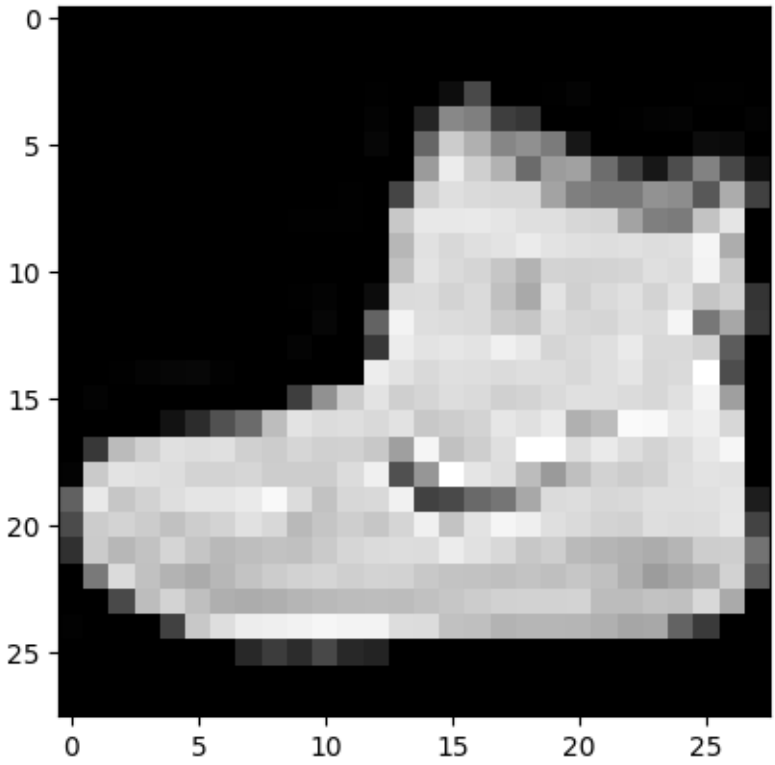
[4]:
```
image, label = trainingdata[0]
print(image.shape, label)
```

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

[5]:
```
#Checking the data
print(image.squeeze().shape)
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

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

[5]:
```
<matplotlib.image.AxesImage at 0x7dac5cdfc520>
```

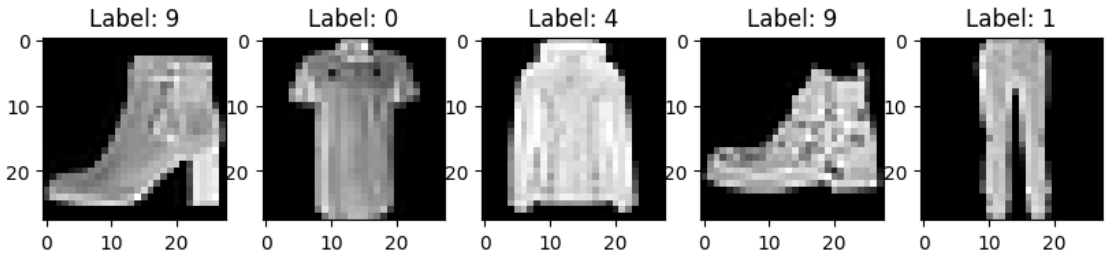


[6]:
```
#Loading the data using DataLoader in order to make it efficient
trainDataLoader = torch.utils.data.
  ↪DataLoader(trainingdata,batch_size=64,shuffle=True)
testDataLoader = torch.utils.data.
  ↪DataLoader(testdata,batch_size=64,shuffle=False)
```

[7]:
```
#Verifying the length of the loaded data
print(len(trainDataLoader) * 64) # batch_size from above
print(len(testDataLoader) * 64)
```

```
60032
10048
```

[9]:
```
#Checking how the data looks like
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)
```



[11]:
```
# Checking for availability of CPU or CUDA
device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Using device:', device)
torch.cuda.is_available()
image=image.to(device)
print("image",image.is_cuda)
```

```
Using device: cpu
image False
```

The above print statement shows that we have CPU available and would hence run on that.

We have to create a neural network with 256, 128, and 64 neurons respectively, all with ReLU activations. Therefore ur network has different layers - first hidden layer has 256 nodes and is connected to the second hidden layer which has 128 nodes which is further connected to a hidden layer with 64 nodes. Relu is used as the activation function for the hidden layers. The last layer is the output layer which has 10 nodes that represent the 10 class labels.

[12]:
```
#Defining the neural network
class NeuralNetwork(torch.nn.Module):
    def __init__(self):
        super().__init__()
        #Flattening the 28*28 image into tensor of length 784
        self.flatten = torch.nn.Flatten()
        # Creating the layers of the neural network
        self.linear_relu_stack = torch.nn.Sequential(
            torch.nn.Linear(784, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
```

```
        return logits
```

The 3 step recipe is - 1. Representation - it is a neural network 2. Measure of goodness - we are using the cross entropy loss function as we have categorical variables 3. Optimizing the model - we are using SGD optimizer with a learning rate of 0.01

[14]:
```
model = NeuralNetwork().to(device) # Representation
loss=torch.nn.CrossEntropyLoss() #2. Measure of goodness
optimizer=torch.optim.SGD(model.parameters(),lr=0.01) #3.Optimizing the model
print(model)
print(loss)
```

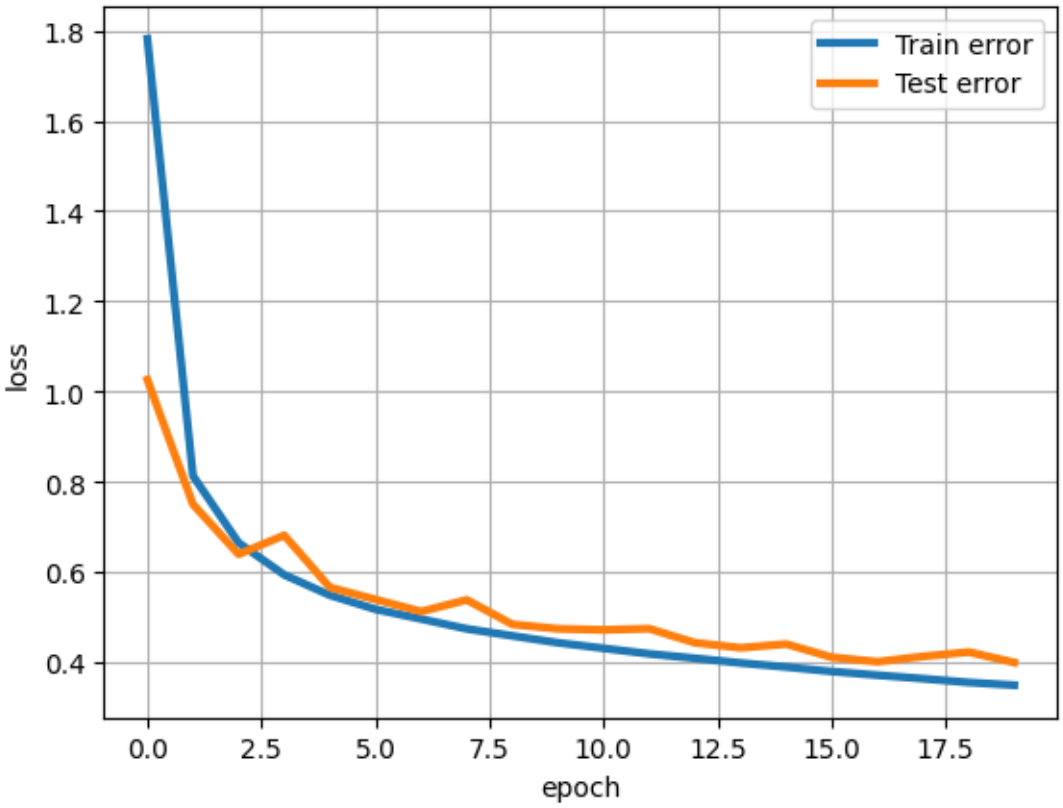
```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=128, bias=True)
    (3): ReLU()
    (4): Linear(in_features=128, out_features=64, bias=True)
    (5): ReLU()
    (6): Linear(in_features=64, out_features=10, bias=True)
  )
)
CrossEntropyLoss()
```

[17]:
```
train_loss_history = []
test_loss_history = []

# training the model for 20 epochs
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.cpu()
    labels=labels.cpu()
    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.cpu()
      labels=labels.cpu()
      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 1.7819251995096836, Test loss 1.0261874035665184
Epoch 1, Train loss 0.8129061895138674, Test loss 0.7497557033399108
Epoch 2, Train loss 0.6646442640501299, Test loss 0.6391484593130221
Epoch 3, Train loss 0.5935777638639722, Test loss 0.6811564689988543
Epoch 4, Train loss 0.5482979146148096, Test loss 0.565602210107123
Epoch 5, Train loss 0.5170701298632348, Test loss 0.5388425007747237
Epoch 6, Train loss 0.49565904925881166, Test loss 0.5121295626755733
Epoch 7, Train loss 0.4738062710237147, Test loss 0.5379782387405444
Epoch 8, Train loss 0.4582294791714469, Test loss 0.48369779565911386
Epoch 9, Train loss 0.4429047410485587, Test loss 0.4738432527727382
Epoch 10, Train loss 0.4305858701499286, Test loss 0.4714879361307545
Epoch 11, Train loss 0.418352536031051, Test loss 0.4740946687710513
Epoch 12, Train loss 0.40847969174321525, Test loss 0.4429735034514385
Epoch 13, Train loss 0.3981113229860375, Test loss 0.4316815262197689
Epoch 14, Train loss 0.38911121696043116, Test loss 0.4400212998223153
Epoch 15, Train loss 0.3791961253706072, Test loss 0.4107440999548906
Epoch 16, Train loss 0.3710171215887517, Test loss 0.40075994458547826
Epoch 17, Train loss 0.36329754233868644, Test loss 0.4125317935920825
Epoch 18, Train loss 0.3549458852359481, Test loss 0.4223640187151113
Epoch 19, Train loss 0.34882848755097084, Test loss 0.3991674111717066
```

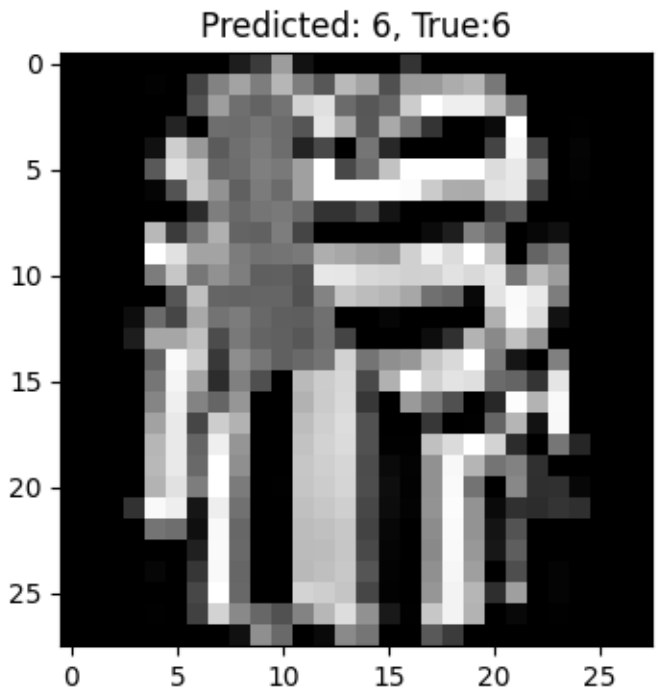
[18]:
```
# plot for loss vs epoch for train and test data
plt.plot(range(20),train_loss_history,'-',linewidth=3,label='Train error')
plt.plot(range(20),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
plt.show()
```

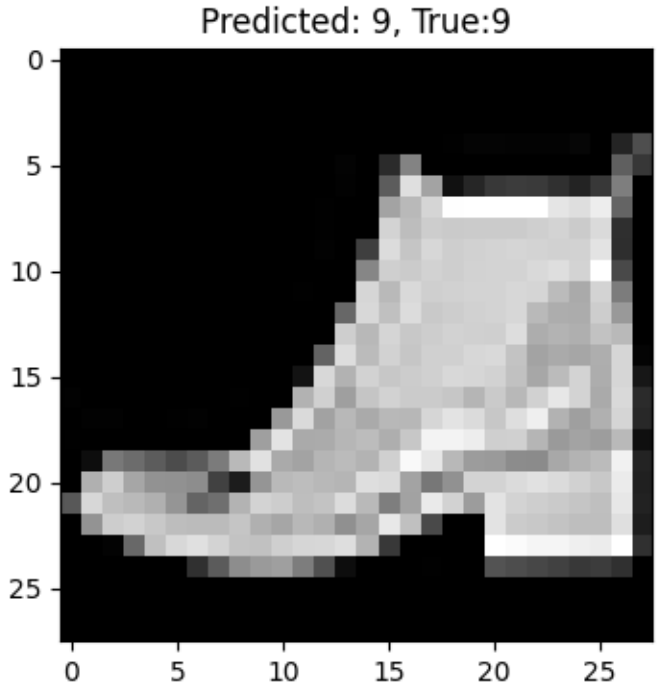


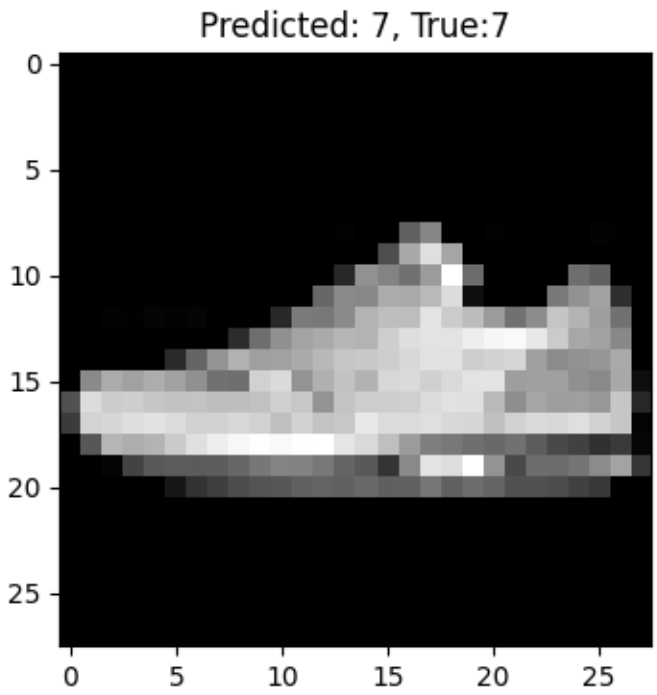
[19]:
```
#Establishing comparison for a small sample dataset
predicted_outputs = model(images)
predicted_classes = torch.max(predicted_outputs, 1)[1]
print('Predicted:', predicted_classes)
fit = loss(predicted_output, labels)
print('True labels:', labels)
print(fit.item())
```

```
Predicted: tensor([3, 1, 7, 5, 8, 4, 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])
0.1585426926612854
```

[23]:
```
for i in range(3):
  plt.figure(figsize=(10,4))
  index = np.random.randint(1, 16)
  plt.title(f'Predicted: {predicted_classes[index].item()}, True:{labels[index].
 ↪item()}')
  plt.imshow(images[index].squeeze().cpu(), cmap = plt.cm.gray)
```


Predicted: 6, True:6
0
5
10
15
20
25
0
5
10
15
20
25


Predicted: 9, True:9
0
5
10
15
20
25
0
5
10
15
20
25



The above image samples show that the model is predicting correct labels as the predicted and true labels are same.

[24]:
```
#COunting the correct predicted samples
correct=0
correct+=(predicted_classes==labels).float().sum()
print(correct)
```

```
tensor(15.)
```

[25]:
```
# Calculating model accuracy
accuracy=correct/images.shape[0]
```

[26]:
```
print(accuracy)
```

```
tensor(0.9375)
```

Model has an accuracy of 93.75% which means it is making correct predictions for most of the samples.

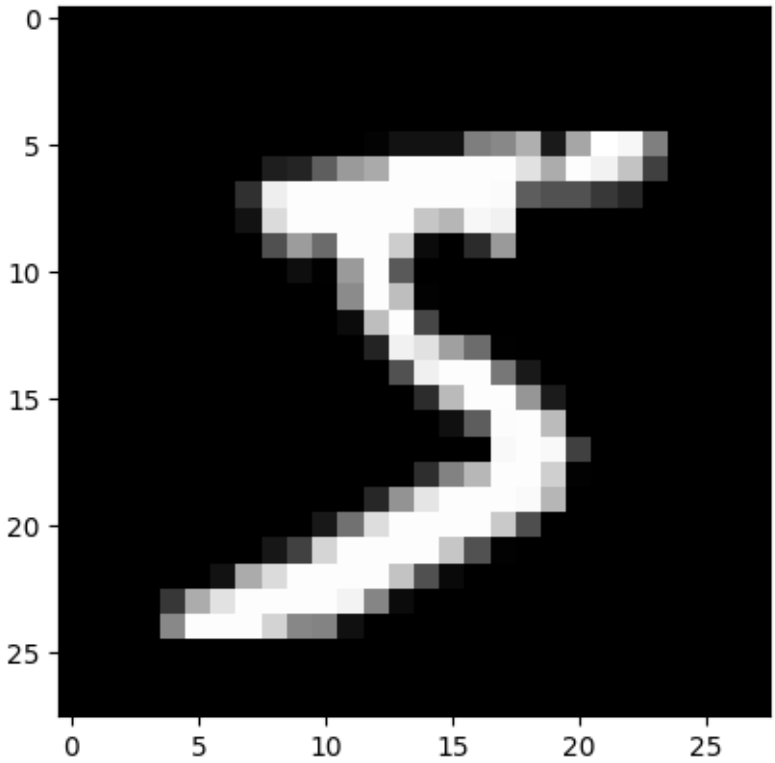
4. (5 points) Implementing back-propagation in Python from scratch. Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other Python IDE of your choice) and complete the missing items. In the second demo, we worked with autodiff. Autodiff enables us to 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.

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.

[14]:
```
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.

[15]:
```
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 $1/\max(n_{in}, n_{out})$.

[16]:
```
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.
# ...
input_size = 784 #layer 1
hidden_size = 32 #hidden layer
output_size = 10 #number of labels

w1=np.random.normal(0, np.sqrt(1/input_size), size=(input_size, hidden_size))
#print(w1.shape[1])
w2=np.random.normal(0, np.sqrt(1/hidden_size), size=(w1.shape[1], hidden_size))
```

```
#print(w2.shape[1])
w3=np.random.normal(0, np.sqrt(1/hidden_size), size=(w2.shape[1], output_size))
weights =[w1,w2,w3] #weights calculated

b1= np.zeros((1,w1.shape[1]))
b2= np.zeros((1,w2.shape[1]))
b3= np.zeros((1,w3.shape[1]))
biases = [b1,b2,b3] #biases calculated
```

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).

[17]:
```
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.
  # ...
  sample1=sample.reshape(1,784) #flattening the image
  #layer 1
  Z1=np.matmul(sample1,weights[0])+biases[0]
  a1=sigmoid(Z1)
  #layer 2, output from previous layer is passed to the next layer
  Z2=np.matmul(a1,weights[1])+biases[1]
  a2=sigmoid(Z2)
  #output from previous hidden layer is sent to the output layer
  Z3=np.matmul(a2,weights[2])+biases[2]
  #applying activation function to get the final output
  ypred=softmax(Z3)
  #encoding the actual labels
  ylabel=integer_to_one_hot(y,10)
  #calculating loss between actual and predicted labels
  loss=cross_entropy_loss(ylabel,ypred)
  one_hot_guess=integer_to_one_hot(np.argmax(ypred),10)#one hot endcoded value␣
 ↪for the final output

  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
  # ...
  for index in range(x.shape[0]):
    losses[index], one_hot_guesses[index] = feed_forward_sample(x[index],␣
 ↪y[index])

  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): 864.0 / 10000 ( 8.64 %)
```

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.

[18]:
```
def train_one_sample(sample, y, learning_rate=0.003):
  a = sample.flatten()

  # We will store each layer's activations to calculate gradient
  activations = []

  # Forward pass

  # Q3. This should be the same as what you did in feed_forward_sample above.
  # ...

  #flattening the image
  sample1=sample.reshape(1,784)
  #Calculating output from layer 1
  Z1=np.matmul(sample1,weights[0])+biases[0]
  a1=sigmoid(Z1)
  #output from previous layer is passed to the next layer
  Z2=np.matmul(a1,weights[1])+biases[1]
  a2=sigmoid(Z2)
  #output from previous layer is sent to the output layer
  Z3=np.matmul(a2,weights[2])+biases[2]
  #applying activation function to get the final output
  ypred=softmax(Z3)
  #encoding the actual labels
  ylabel=integer_to_one_hot(y,10)
  #calculating loss between actual and predicted labels
  loss=cross_entropy_loss(ylabel,ypred)

  one_hot_guess=integer_to_one_hot(np.argmax(ypred),10)#one hot endcoded value␣
↪for the final output

  #Backpropogation traverses backwards to compute the new weights and biases
  # Output layer, it goes backwards and recalculates the values of weights and␣
↪biases for dw3,db3
  dZ3=ypred-ylabel
  dw3=np.dot(a2.T,dZ3)
  db3=dZ3
  #layer 2, it goes backwards and recalculates the values of weightsand biases␣
↪for dw2,db2
  dZ2= np.multiply(np.dot(dZ3,w3.T),dsigmoid(Z2))
  dw2=np.dot(a1.T,dZ2)
  db2=dZ2
  #layer 1, it goes backwards and recalculates the values of weightsand biases␣
↪for dw1,db1
  dZ1=np.multiply(np.dot(dZ2,w2.T),dsigmoid(Z1))
  dw1=np.dot(sample1.T,dZ1)
  db1=dZ1
```

```
  #updating the new weights
  weight_gradients=[dw1,dw2,dw3]
  #updating the new biases
  bias_gradients=[db1,db2,db3]


  # 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.
  # ...
  for i in range(3):
    # Update weights & biases based on your calculated gradient
    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.

[19]:
```
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]):
    train_one_sample(x_train[i], y_train[i], 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: 2.43
Accuracy (# of correct guesses): 864.0 / 10000 ( 8.64 %)
```

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

Feeding forward all test data…

Average loss: 0.86
Accuracy (# of correct guesses): 7092.0 / 10000 ( 70.92 %)

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

Feeding forward all test data…

Average loss: 0.89
Accuracy (# of correct guesses): 7006.0 / 10000 ( 70.06 %)

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

Feeding forward all test data…

Average loss: 0.89
Accuracy (# of correct guesses): 7105.0 / 10000 ( 71.05 %)
```

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

Citations

- https://github.com/kvgarimella/dl-demos/blob/main/demo01-basics.ipynb
- https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1
- https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html
- https://ashki23.github.io/markdown-latex.html
- Class notes
- Chat GPT