Yinhong Qin
yq2021@nyu.edu

Homework 1

ECE 7123
September 30, 2022

Problem 1

Assume a single data point is in $\mathbb{R}^d$.

1. X is a matrix contains n training data points. Suppose each row represents a data point, the size of X is $n \times d$.

   Since $y = Xw$ is the vector of corresponding labels, then the size of y is $n \times 1$.

   Assume the predict label is $\hat{y}$ and the ground truth label is y, the loss function l can be written in:

   $$l = \frac{1}{n}|\hat{y} - y| = \frac{1}{n}|Xw - y| = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

   The size of loss function is also $n \times 1$.

2. We cannot simply write down the optimal linear model in closed form.

   Because the closed form solution is to solve the equation:

   $$\nabla_w l = 0$$
   $$\nabla_w(\frac{1}{n}|Xw - y|) = 0$$
   $$\frac{1}{n}X = 0$$

   We could notice that in the final expression, there is no term related to w. Besides, it is obvious that we cannot set the data points matrix X to zero to solve this.

Problem 2

a. The structure of neural network with 2 hidden neurons is shown below:

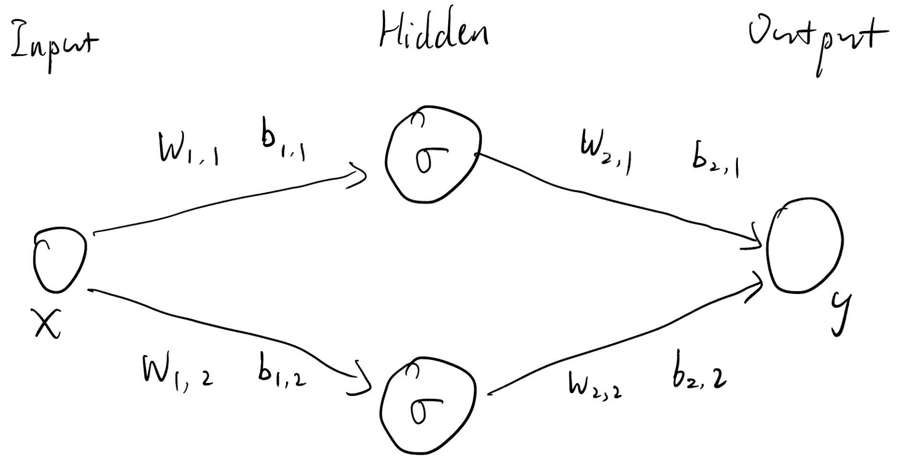


Figure 1. The strcture of neural network.

Thus, we can know the output can be written as:

$$y = (W_{2,1} * \sigma(W_{1,1}x + b_{1,1}) + b_{2,1}) + (W_{2,2} * \sigma(W_{1,2}x + b_{1,2}) + b_{2,2})$$

To realize the box function shown below:

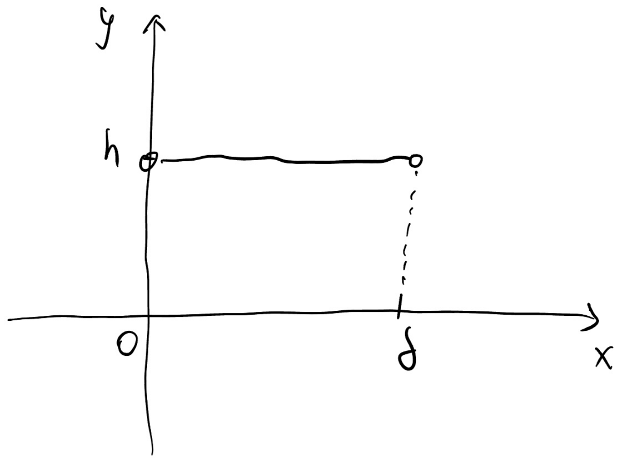


Figure 2. The target box function.

We could first use the hidden layer to select the "interested input region" and then use the weight of the output to reach the height h. Thus, the bias in the output layer should be 0.

$$b_{2,1} = 0, b_{2,2} = 0$$

Since the unit step activation function has a infinite non-zero part, we could utilize the difference between 2 unit step function to realize a "finite length non-zero" box function. Thus, we can set

$$W_{2,1} = h, W_{2,2} = -h$$

For the positive part, we need it to appear in $(0, +\infty)$ and the negative part need to appear in $(\delta, +\infty)$ to neutralize the positive part.

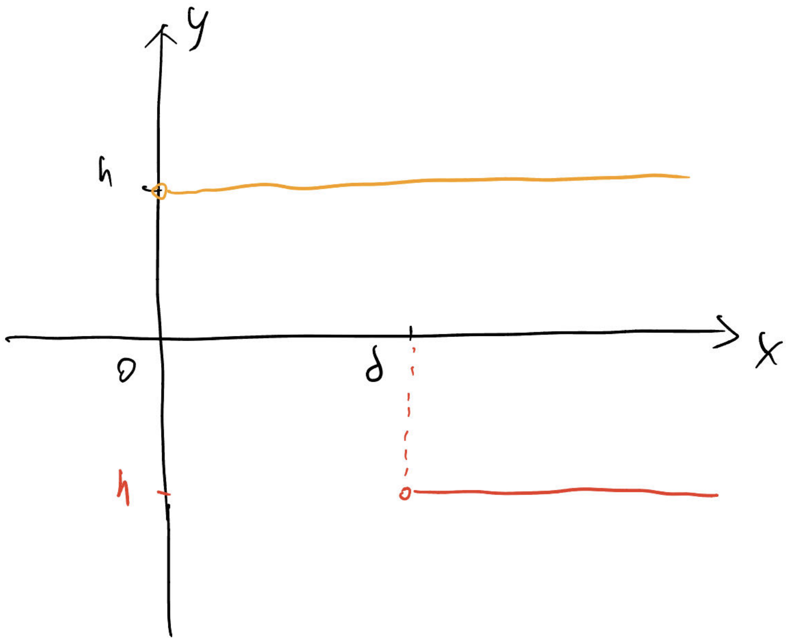


Figure 3. The positive part and negative part.

Thus, we can solve that:

$$W_{1,1} = 1, b_{1,1} = 0, W_{1,2} = 1, b_{1,2} = -\delta$$

In conclusion, the parameters for the neural network shown in Figure 1. are:

$$W_{1,1} = 1, W_{1,2} = 1, b_{1,1} = 0, b_{1,2} = -\delta$$
$$W_{2,1} = h, W_{2,2} = -h, b_{2,1} = 0, b_{2,2} = 0$$

b. For any arbitrary, smooth, bounded function defined over an interval $[-B, B]$, similar to the definite integration, we could divide the function into infinite small intervals, where each interval represents a box function, like the picture below: Based on problem a).,

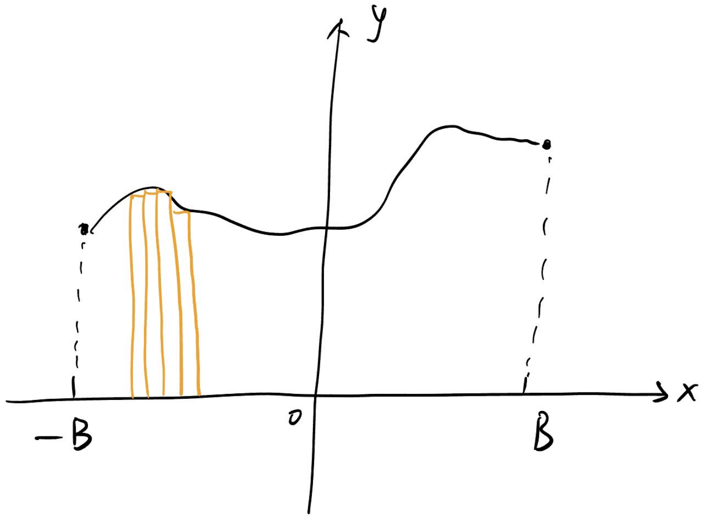


Figure 4. The divide of the function.

we could know a box function with any kinds of heights and width can be realized by changing the weight and bias of a pair of neurons. By keep adding pairs of neurons, we can gradually approximate this function with less errors.

Therefore, any arbitrary, smooth, bounded function defined over an interval $[-B, B]$ can be closely approximated by a neural network with a hidden layer of neurons.

c. Based on problem b.) and *"Universal approximation theorem"* [1], we can know that a single hidden layer of neurons can approximate any function for 1-d inputs. Similarly, we can extrapolate this theorem to $d-$dimensional inputs by applying it to each component. However, there are some practical tradeoffs.

First, the more complex the target function is, the more neurons we need. That will make the layer becomes too complicated to be interpreted. Another issue is when the number of neurons increases, a tiny change on a parameter can significantly influence the output function. It will make the neural network more vulnerable to noisy input and more possible to be overfitting to training data. Thus the generalization of this neural network is degrading too.

[1] https://en.wikipedia.org/wiki/Universal_approximation_theorem

Problem 3

The relationship between y and z is

$$y = softmax(z), y_i = \frac{\exp z_i}{\sum \exp(z_i)}$$

The partial derivative can be written in:

$$\frac{\partial y_i}{\partial z_j} = \frac{\frac{\partial \exp(z_i)}{\partial z_j} \cdot \sum \exp(z_i) - \exp(z_i) \cdot \frac{\partial \sum \exp(z_i)}{\partial z_j}}{(\sum \exp(z_i))^2}$$

1. if $i = j$, then we can have:

$$\begin{aligned}\frac{\partial y_i}{\partial z_j} &= \frac{\frac{\partial \exp(z_i)}{\partial z_j} \cdot \sum \exp(z_i) - \exp(z_i) \cdot \frac{\partial \sum \exp(z_i)}{\partial z_j}}{(\sum \exp(z_i))^2} \\ &= \frac{\exp(z_i) \cdot \sum \exp(z_i) - \exp(z_i) \cdot \exp(z_i)}{(\sum \exp(z_i))^2} \\ &= \frac{\exp(z_i)}{\sum \exp(z_i)} - \frac{\exp(z_i)}{\sum \exp(z_i)} \cdot \frac{\exp(z_i)}{\sum \exp(z_i)} \\ &= y_i - y_i \cdot y_i \\ &= y_i(1 - y_i)\end{aligned}$$

2. if $i \neq j$, then we can have:

$$\begin{aligned}\frac{\partial y_i}{\partial z_j} &= \frac{\frac{\partial \exp(z_i)}{\partial z_j} \cdot \sum \exp(z_i) - \exp(z_i) \cdot \frac{\partial \sum \exp(z_i)}{\partial z_j}}{(\sum \exp(z_i))^2} \\ &= \frac{0 \cdot \sum \exp(z_i) - \exp(z_i) \cdot \exp(z_j)}{(\sum \exp(z_i))^2} \\ &= -\frac{\exp(z_i)}{\sum \exp(z_i)} \cdot \frac{\exp(z_j)}{\sum \exp(z_i)} \\ &= -y_i \cdot y_j\end{aligned}$$

Thus, the entry in the Jacobian matrix can be written as:

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

Yinhong Qin
yq2021@nyu.edu

Homework 1

ECE 7123
September 30, 2022

Problem 4

Please check the sub-report below:

HW1_p4

September 30, 2022

[6]:
```
# Import PyTorch package
import numpy as np
import torch
import torchvision
```

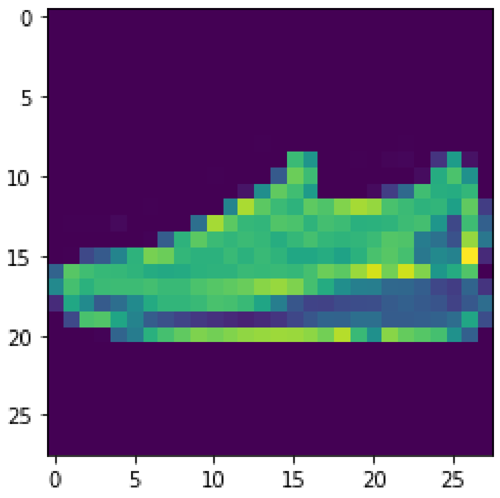
[7]:
```
# Import Dataset
trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/
 ↪',train=True,download=True,transform=torchvision.transforms.ToTensor())
testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/
 ↪',train=False,download=True,transform=torchvision.transforms.ToTensor())
```

```
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

  0%|          | 0/26421880 [00:00<?, ?it/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

  0%|          | 0/29515 [00:00<?, ?it/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
```

```
  0%|          | 0/4422102 [00:00<?, ?it/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
  0%|          | 0/5148 [00:00<?, ?it/s]


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

[9]:
```
# Test Downloaded Dataset
image, label = trainingdata[1893]
print(image.shape, label)

import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(image.squeeze().numpy())
plt.show()
```

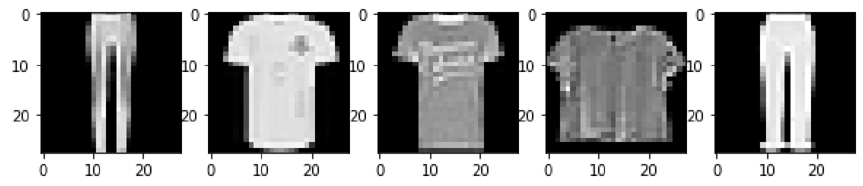
```
torch.Size([1, 28, 28]) 7
```



```
[10]: # Create data loader for training and testing
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
```

```
[11]: # Test Dataloader
images, labels = iter(trainDataLoader).next()

plt.figure(figsize=(10,4))
for index in np.arange(0,5):
  plt.subplot(1,5,index+1)
  plt.imshow(images[index].squeeze().numpy(),cmap=plt.cm.gray)
```



```
[12]: # Create Network Structure and initialize the parameter
from torch import nn

net_3_layers_perceptron = nn.Sequential(nn.Flatten(),
                    nn.Linear(28 * 28, 256),
                    nn.ReLU(),
                    nn.Linear(256, 128),
                    nn.ReLU(),
                    nn.Linear(128, 64),
                    nn.ReLU(),
                    nn.Linear(64, 10)).cuda()

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)
```

```
net_3_layers_perceptron.apply(init_weights)
```

```
[12]: Sequential(
        (0): Flatten(start_dim=1, end_dim=-1)
        (1): Linear(in_features=784, out_features=256, bias=True)
        (2): ReLU()
        (3): Linear(in_features=256, out_features=128, bias=True)
        (4): ReLU()
        (5): Linear(in_features=128, out_features=64, bias=True)
        (6): ReLU()
        (7): Linear(in_features=64, out_features=10, bias=True)
      )
```

```
[13]: # Define the learning rate, epoch nums, optimizer, loss function...
lr = 0.02
epoch_nums = 50
Loss = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net_3_layers_perceptron.parameters(), lr)
```

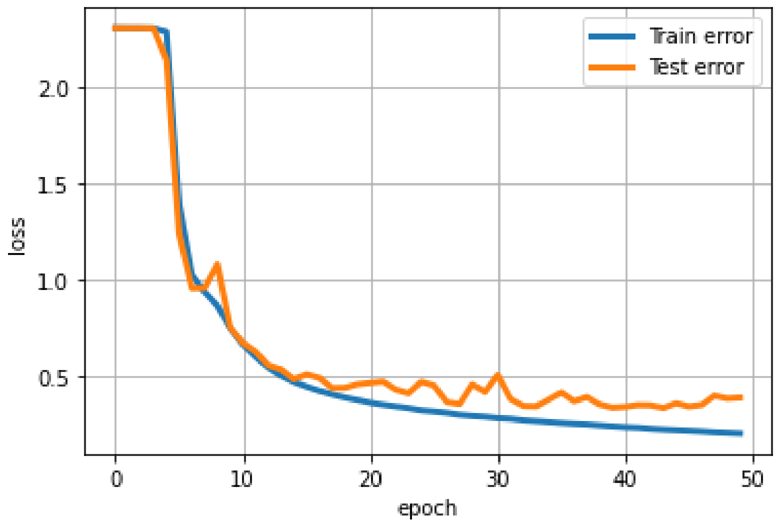
```
[14]: # Begin training and testing
train_loss_history = []
test_loss_history = []
test_accuracy_history = []

for epoch in range(epoch_nums):
  train_loss = 0.0
  test_loss = 0.0
  test_accuracy = 0.0
  for i, data in enumerate(trainDataLoader):
    images, labels = data
    images = images.cuda()
    labels = labels.cuda()
    optimizer.zero_grad()
    predicted_output = net_3_layers_perceptron(images)
    fit = Loss(predicted_output,labels)
    fit.backward()
    optimizer.step()
    train_loss += fit.item()

  correct_predicted_label_num = 0
  total_num_of_labels = len(testdata)
  for i, data in enumerate(testDataLoader):
    with torch.no_grad():
      images, labels = data
      images = images.cuda()
      labels = labels.cuda()
      predicted_output = net_3_layers_perceptron(images)
```

```python
        fit = Loss(predicted_output,labels)
        test_loss += fit.item()

        # Calculate testing accuracy
        predicted_labels = torch.max(predicted_output, 1).indices
        correct_predicted_label_num += torch.eq(predicted_labels, labels).sum()


    test_accuracy = correct_predicted_label_num / total_num_of_labels
    train_loss = train_loss/len(trainDataLoader)
    test_loss = test_loss/len(testDataLoader)
    train_loss_history.append(train_loss)
    test_loss_history.append(test_loss)
    test_accuracy_history.append(test_accuracy)
    print('Epoch %s, Train loss %s, Test loss %s'%(epoch, train_loss, test_loss))
```

```
Epoch 0, Train loss 2.303288939410944, Test loss 2.3026329453583734
Epoch 1, Train loss 2.3026656313999885, Test loss 2.3025010634379783
Epoch 2, Train loss 2.302520084736952, Test loss 2.3023078153087835
Epoch 3, Train loss 2.3020962522482313, Test loss 2.3013176371337503
Epoch 4, Train loss 2.2838296633539423, Test loss 2.1378464182471015
Epoch 5, Train loss 1.3927732301292135, Test loss 1.239315010939434
Epoch 6, Train loss 1.0284173071130251, Test loss 0.957804045100121
Epoch 7, Train loss 0.937849393912724, Test loss 0.9602925390194935
Epoch 8, Train loss 0.8683912374063342, Test loss 1.081169197513799
Epoch 9, Train loss 0.7525679993985305, Test loss 0.7556559291614848
Epoch 10, Train loss 0.6679288638171865, Test loss 0.6756025622984406
Epoch 11, Train loss 0.6047675323003391, Test loss 0.6313826830903436
Epoch 12, Train loss 0.5482314621239329, Test loss 0.5598531329328087
Epoch 13, Train loss 0.5056541453260602, Test loss 0.5387791215803972
Epoch 14, Train loss 0.4740880811169966, Test loss 0.4881954356363625
Epoch 15, Train loss 0.4492566147084429, Test loss 0.5135117249124369
Epoch 16, Train loss 0.42821390169070983, Test loss 0.49530001962260833
Epoch 17, Train loss 0.4102137350101969, Test loss 0.4424129799482929
Epoch 18, Train loss 0.3940752685260671, Test loss 0.4436657093702608
Epoch 19, Train loss 0.3812355605809927, Test loss 0.46166111775644264
Epoch 20, Train loss 0.3669788326535906, Test loss 0.4690746577682009
Epoch 21, Train loss 0.356240924145939, Test loss 0.47580185779340706
Epoch 22, Train loss 0.34739060913607767, Test loss 0.4345349000327906
Epoch 23, Train loss 0.3385332088305879, Test loss 0.41566411685791743
Epoch 24, Train loss 0.3271602399901413, Test loss 0.4743002156732948
Epoch 25, Train loss 0.3214676023593971, Test loss 0.4558868312342152
Epoch 26, Train loss 0.3145685161529446, Test loss 0.37010603250971263
Epoch 27, Train loss 0.3044803959410836, Test loss 0.36022600465139765
Epoch 28, Train loss 0.2988965465331764, Test loss 0.4618303416071424
Epoch 29, Train loss 0.2946172474480387, Test loss 0.4225139598937551
Epoch 30, Train loss 0.28770788703391803, Test loss 0.5094753506646794
Epoch 31, Train loss 0.2840642210350298, Test loss 0.38700716482226255
Epoch 32, Train loss 0.27597395026448696, Test loss 0.34951994553872734
Epoch 33, Train loss 0.27154296161984204, Test loss 0.3474722394518032
Epoch 34, Train loss 0.266581248738237, Test loss 0.3835802800526285
Epoch 35, Train loss 0.26148684940008976, Test loss 0.41975149445852655
Epoch 36, Train loss 0.25747979201995996, Test loss 0.37533049750479924
Epoch 37, Train loss 0.25434230673891395, Test loss 0.3972062335652151
Epoch 38, Train loss 0.24917279453928282, Test loss 0.35734037884101744
Epoch 39, Train loss 0.24438226079222744, Test loss 0.34130788437879767
Epoch 40, Train loss 0.2400833210870147, Test loss 0.3452069767436404
Epoch 41, Train loss 0.2374104316165643, Test loss 0.3544638219532693
Epoch 42, Train loss 0.23199086100149002, Test loss 0.35311419331723715
Epoch 43, Train loss 0.22812451036579445, Test loss 0.3392550607870339
Epoch 44, Train loss 0.22525270053668062, Test loss 0.36403391344152436
Epoch 45, Train loss 0.22210550966706358, Test loss 0.3467123775155681
Epoch 46, Train loss 0.2187631985526095, Test loss 0.35545669277762154
Epoch 47, Train loss 0.21435141906952426, Test loss 0.40347034934979337
Epoch 48, Train loss 0.21154671208833709, Test loss 0.3895075054495198
Epoch 49, Train loss 0.20860832766381535, Test loss 0.3934898465207428
```

[16]:
```
# Plotting tarining and testingloss
plt.plot(range(epoch_nums),train_loss_history,'-',linewidth=3,label='Train␣
 ↪error')
plt.plot(range(epoch_nums),test_loss_history,'-',linewidth=3,label='Test error')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.grid(True)
plt.legend()
```

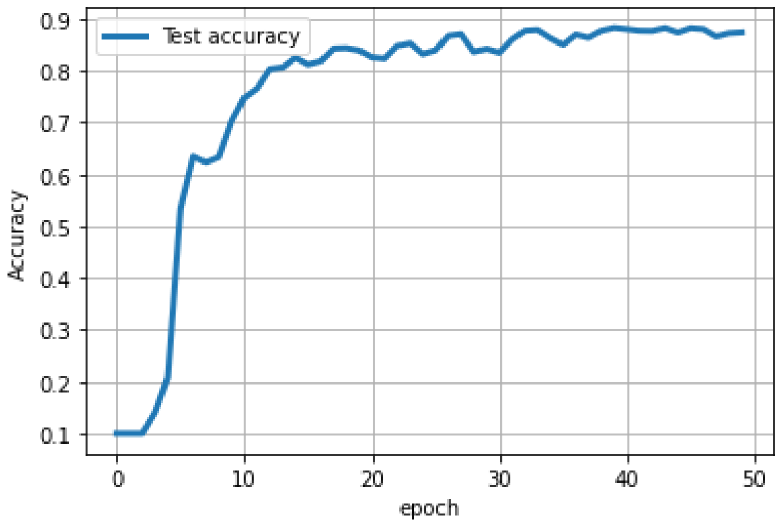
[16]:
```
<matplotlib.legend.Legend at 0x7fe1dd737990>
```



```
[32]: for i in range(epoch_nums):
        item = test_accuracy_history[i].cpu().numpy()
        test_accuracy_history[i] = item
```

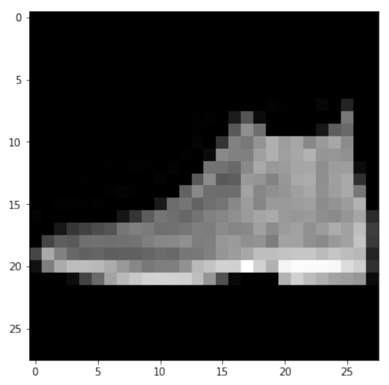
```
[34]: # Plotting testing accuracy
      plt.plot(range(epoch_nums),test_accuracy_history,'-',linewidth=3,label='Test␣
       ↪accuracy')
      plt.xlabel('epoch')
      plt.ylabel('Accuracy')
      plt.grid(True)
      plt.legend()
```

```
[34]: <matplotlib.legend.Legend at 0x7fe1dc05e150>
```

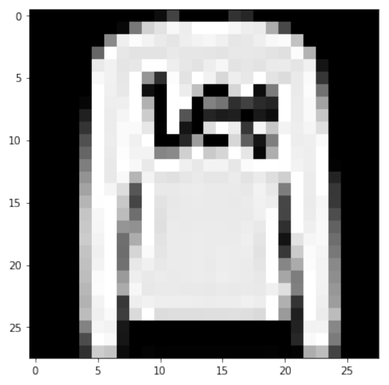


[51]:
```
# Select random 3 images from test dataset and get the predicted labels using␣
↪trained model
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',␣
↪'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
random_3_dataLoader = torch.utils.data.
↪DataLoader(testdata,batch_size=3,shuffle=False)

random_3_images, random_3_labels = iter(random_3_dataLoader).next()
predicted_output = net_3_layers_perceptron(random_3_images.cuda())
predicted_labels = torch.max(predicted_output, 1)

# Print the prediction probabilities using softmax
softmax_layer = nn.Softmax(dim=1)
predicted_prob = softmax_layer(predicted_output)

# Plot selected images and its label, predicted label and prediction probability
plt.subplots_adjust(top=5)
for index in range(3):
  plt.subplot(3, 1, index+1)
  plt.imshow(random_3_images[index].cpu().squeeze().numpy(),cmap=plt.cm.gray)
  predicted_label = predicted_labels.indices[index]
  plt.title('The correct label is: %s.\n The correct label is: %s, its accuracy␣
↪is: %.5lf.\n' % (class_names[random_3_labels[index]],␣
↪class_names[predicted_label], predicted_prob[index][predicted_label]))
```

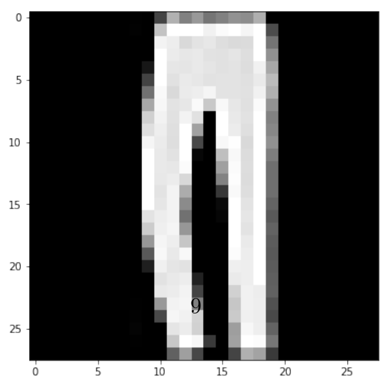
The correct label is: Ankle boot.
The correct label is: Ankle boot, its accuracy is: 0.99305.



The correct label is: Pullover.
The correct label is: Pullover, its accuracy is: 0.99924.



The correct label is: Trouser.
The correct label is: Trouser, its accuracy is: 1.00000.



Based on the training and testing results we can notice that:

1. Compare to the single layer linear regression model in demo 1, this 3 hidden layer perceptron has a higher training and testing loss at the beginning. After several about 20 epochs, the training loss decrease to a relatively lower level.
2. The accuracy curve start from around 10%, which is almost the accuracy by giving a random guess on a 10 category dataset. After 10 epochs, the testing accuracy becomes greater than 80%. After around 35 epochs, the accuracy reaches around 85% and finally stay around 87%.

Problem 5

Please check the sub-report below:

dl_hw1_prob5

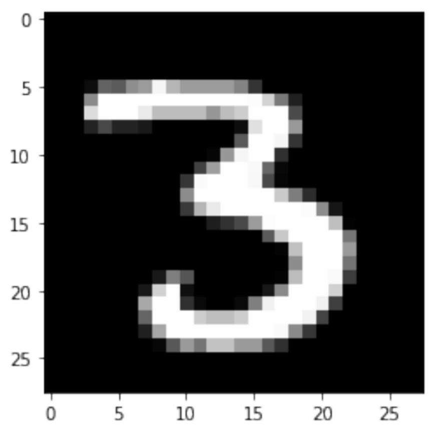
September 30, 2022

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.

[ ]:
```
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[12],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.

[67]:
```python
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(max)
  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 $1/\max(n_{in}, n_{out})$.

```
[90]: 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(44017)

      # Q1. Fill initialization code here.
      # ...

      # Generate initial weight and biases parameters

      weights = [rng.normal(0, 1 / (784), (32, 784)), rng.normal(0, 1 / (32), (32,␣
       ↪32)), rng.normal(0, 1 / (32), (10, 32))]
      biases = [np.zeros(32), np.zeros(32), np.zeros(10)]
```

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).

```
[69]: def feed_forward_sample(sample, ground_truth_label):
        """ 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.

        # First we need to flatten the input image to a 1-D vector
        layer_output = sample.flatten() # The flattened image size is (1, 28*28) =␣
       ↪(1, 784)

        # Then we pass the input forward to each layer
        for index in range(3):
```

```
    neurons_before_activation = np.matmul(weights[index], layer_output) +␣
↪biases[index] # forward to next layer
    if index == 2:
      layer_output = softmax(neurons_before_activation) # If we reach the␣
↪output layer, we use softmax to compute the probabilty
    else:
      layer_output = sigmoid(neurons_before_activation)# Otherwise, we use␣
↪sigmoid as activation function

  # Calculate ground truth distribution
  ground_truth_label_distribution = integer_to_one_hot(ground_truth_label, 10)
  # Calculate the crossentropy loss
  loss = cross_entropy_loss(ground_truth_label_distribution, layer_output)

  # Get the predicted label
  predicted_label = np.argmax(layer_output)

  # Set the predicted output distribution
  one_hot_guess = np.zeros_like(layer_output)
  one_hot_guess[predicted_label] = 1

  return loss, one_hot_guess


def feed_forward_dataset(x, y):
  num_of_dataset = x.shape[0]
  losses = np.empty(num_of_dataset)
  one_hot_guesses = np.empty((num_of_dataset, 10))


  # ...
  # Q2. Fill code here to calculate losses, one_hot_guesses
  # ...
  for i in range(num_of_dataset):
    losses[i], one_hot_guesses[i] = feed_forward_sample(x[i], 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: 2.31
Accuracy (# of correct guesses): 1009.0 / 10000 ( 10.09 %)
```

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.

[97]:
```
def train_one_sample(sample, ground_truth_label, learning_rate=0.003):
  layer_output = 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.
  for index in range(3):
    neurons_before_activation = np.matmul(weights[index], layer_output) +␣
↪biases[index] # forward to next layer
    if index == 2:
      layer_output = softmax(neurons_before_activation) # If we reach the␣
↪output layer, we use softmax to compute the probabilty
    else:
      layer_output = sigmoid(neurons_before_activation) # Otherwise, we use␣
↪sigmoid as activation function

    # After each layer, we need to store the activations
    activations.append(layer_output)

  # Calculate ground truth distribution and the loss
  ground_truth_label_distribution = integer_to_one_hot(ground_truth_label, 10)
  loss = cross_entropy_loss(ground_truth_label_distribution, layer_output)
```

```
# Get the predicted label
predicted_label = np.argmax(layer_output)
# Set the predicted output distribution
one_hot_guess = np.zeros_like(layer_output)
one_hot_guess[predicted_label] = 1

# Check if we got the correct prediction
corrected_prediction = (predicted_label == ground_truth_label)


# 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.

# Declare variables
num_of_layers = 3
weight_gradients = [None] * num_of_layers
bias_gradients = [None] * num_of_layers
activation_gradients = [None] * (num_of_layers - 1)

for i in range(len(weights) - 1, -1, -1): # Traverse layers in reverse
  if i == num_of_layers - 1:
    # If it is the last layer
    output_y = ground_truth_label_distribution[:, np.newaxis]
    current_activation = activations[i][:, np.newaxis]
    previous_activation = activations[i - 1][:, np.newaxis]

    weight_gradients[i] = np.matmul((current_activation - output_y),␣
↪previous_activation.T)
    bias_gradients[i] = current_activation - output_y

  else:
    next_layer_weights = weights[i + 1]
    next_layer_activation = activations[i + 1][:, np.newaxis]
    output_y = ground_truth_label_distribution[:, np.newaxis]
    current_activation = activations[i][:, np.newaxis]

    # Calculate the activation gradients
    if i == num_of_layers - 2:
      activation_gradient = np.matmul(next_layer_weights.T,␣
↪(next_layer_activation - output_y))
      activation_gradients[i] = activation_gradient
      else:
        activation_gradient_next = activation_gradients[i+1]
        activation_gradient = np.matmul(next_layer_weights.T,␣
↪(dsigmoid(next_layer_activation) * activation_gradient_next))
        activation_gradients[i] = activation_gradient

      # Get the previous activation
      if i > 0:
        previous_activation = activations[i - 1][:, np.newaxis]
      else:
        previous_activation = sample.flatten()[:, np.newaxis]


      # Calculate the gradients with respect to weights and biases
      x = dsigmoid(current_activation) * activation_gradient
      weight_gradients[i] = np.matmul(x, previous_activation.T)
      bias_gradients[i] = x

    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.

[98]:
```
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.
  # ...
  num_of_training_set = x_train.shape[0]
  for i in range(num_of_training_set):
    if i == 0 or ((i + 1) % 10000 == 0):
      completion_percent = format(((i + 1) / x_train.shape[0]) * 100, ".2f")
      print(i + 1, "/", x_train.shape[0], "(", completion_percent, "%)")
    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: 2.31
Accuracy (# of correct guesses): 1014.0 / 10000 ( 10.14 %)

Training for one epoch over the training dataset…
1 / 60000 ( 0.00 %)
10000 / 60000 ( 16.67 %)
20000 / 60000 ( 33.33 %)
30000 / 60000 ( 50.00 %)
40000 / 60000 ( 66.67 %)
50000 / 60000 ( 83.33 %)
60000 / 60000 ( 100.00 %)
Finished training.

Feeding forward all test data…

Average loss: 0.99
Accuracy (# of correct guesses): 6978.0 / 10000 ( 69.78 %)

Training for one epoch over the training dataset…
1 / 60000 ( 0.00 %)
10000 / 60000 ( 16.67 %)
20000 / 60000 ( 33.33 %)
30000 / 60000 ( 50.00 %)
40000 / 60000 ( 66.67 %)
50000 / 60000 ( 83.33 %)
60000 / 60000 ( 100.00 %)
Finished training.

Feeding forward all test data…

Average loss: 0.8
Accuracy (# of correct guesses): 7515.0 / 10000 ( 75.15 %)

Training for one epoch over the training dataset…
1 / 60000 ( 0.00 %)
10000 / 60000 ( 16.67 %)
20000 / 60000 ( 33.33 %)
30000 / 60000 ( 50.00 %)
40000 / 60000 ( 66.67 %)
50000 / 60000 ( 83.33 %)
60000 / 60000 ( 100.00 %)
Finished training.

Feeding forward all test data…

Average loss: 0.72
```

```
Accuracy (# of correct guesses): 7623.0 / 10000 ( 76.23 %)
```

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.