Assignment

Your tasks are as follows. All algorithms should be your own code. No Tensorflow/Pytorch.

1. (2 pts) Apply the normalization on the training and test data.

```
def normalize(data):
  # get mean and std.
  mean = np.mean(data, axis=0)
  std_dev = np.std(data, axis=0)

  # reenforce the division not 0
  std_dev[std_dev == 0] = 1

  #  x = (x-mean) / std
  data = (data - mean) / std_dev
  return data

def preprocess_data(data):
  X = []  # (60000, 784)
  Y = []  # (60000,)
  for image_tensor, label in data:
    # Reshape each image to a flat vector of 784 elements and convert to a numpy array
    X.append(image_tensor.numpy().reshape(784))

    # rewrite the label to (0 if label between 0-4) or (1 if 5-9)
    if label <= 4:
      Y.append(0)
    else:
      Y.append(1)

  X = np.array(X, dtype=np.float32) # without enforece the dtype, may unable to process
  Y = np.array(Y, dtype=np.int64)
  X = normalize(X)      # normalize feature
```

```
X_train,Y_train = preprocess_data(train_data)
X_test, Y_test = preprocess_data(test_data)
print("X_train:", X_train.shape)
print("y_train:",Y_train.shape)
print("X_test:", X_test.shape)
print("Y_test:", Y_test.shape)

print(np.std(X_train))
```

```
X_train: (60000, 785)
y_train: (60000,)
X_test: (10000, 785)
Y_test: (10000,)
0.9563967519997209
```

2. (2 pts) As a baseline, train a linear classifier ˆy = vT x and quadratic loss. Report its test accuracy.

```
# How to implement Linear Regression from scratch with Python
class LinearRegression:
  def __init__(self, lr=0.01, epochs=10):
    self.lr = lr
    self.epochs = epochs
    self.weights = None

  def train(self, X, y):
    n_samples, n_features = X.shape
    self.weights = np.zeros(n_features)

    for epoch in range(self.epochs):
      y_pred = np.dot(X, self.weights)  # f(x) = wx as linaer

      loss = y - y_pred # compute loss

      dw = np.dot(X.T, loss) / n_samples
      self.weights -= self.lr * dw

      if epoch % 10 == 0:  # track losss as mean square error
        print(epoch, " \niter, Loss: ", np.mean(loss**2))
        print(" iter, accuracy: ",self.accuracy(X, y))

  def predict(self, X):
    return (np.dot(X, self.weights) > 0.5) #Return if predict

  def accuracy(self, X, y):
    y_pred = self.predict(X)
    return np.mean(y_pred == y)
```

```python
#problem 1, linear classfication
linear_model = LinearRegression(lr=0.001, epochs=100)
linear_model.train(X_train,Y_train)
#linear_model.train(XO,YO)
print("Accuracy:", linear_model.accuracy(X_test, Y_test))
```

```
0
iter, Loss:  0.49006666666666665
 iter, accuracy:  0.5099333333333333
10
iter, Loss:  0.540939285159809
 iter, accuracy:  0.5099333333333333
20
iter, Loss:  0.6215125870061101
 iter, accuracy:  0.5066833333333334
30
iter, Loss:  0.7564380981161849
 iter, accuracy:  0.44506666666666667
40
iter, Loss:  0.9938405528110311
 iter, accuracy:  0.38526666666666665
50
iter, Loss:  1.4296833821470212
 iter, accuracy:  0.3573
60
iter, Loss:  2.258927872976632
 iter, accuracy:  0.34826666666666667
70
iter, Loss:  3.884077457454416
 iter, accuracy:  0.3480666666666667
80
iter, Loss:  7.147460769903205
```

Observation:

I have trained with learning rate from 0.1 to 0.00001. The smaller learning rate helps reduce the loss and increase accuracy. However, it has the highest accuracy at 51%. Which kind make sense, since linear regression is not designed for classification problem, especially we labeled the target with numerical 0 and 1, model may interpret it as mathematical values.

3. (7 pts) Train a neural network classifier with quadratic loss ℓ(y, f (x)) = (y − f (x))2. Plot the progress
of the test and training accuracy (y-axis) as a function of the iteration counter t (x-axis)2. Report the
final test accuracy for the following choices

- k=5
- k=50
- k=200
- Comment on the role of hidden units k on the ease of optimization and accuracy

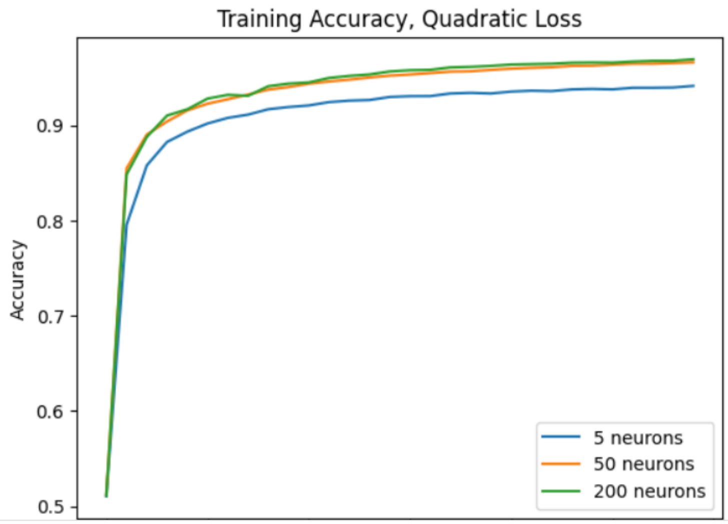
```python
# this structure of this neural network browesd https://github.com/OriYarden/Binary-
class NeuralNetwork:
  def __init__(self, n_neurons=10, lr=0.0001, epochs=10, loss_fun="quadratic", batch
    self.n_neurons = n_neurons
    self.lr = lr
    self.epochs = epochs

    self.loss_fun = loss_fun

    self.W = None
    self.v = None
    self.batch = batch

  #  Xaiver initialization for neural network
  # https://machinelearningmastery.com/weight-initialization-for-deep-learning-neura
  @staticmethod
  def xavier_initialization(n_in, n_out):
    limit = np.sqrt(1 / (n_in + n_out))
    return np.random.uniform(-limit, limit, size=(n_in, n_out))

  def initialize_weights(self, n_features):
    # for the layers, input 785 feature with 60k datapoints, after layer1 become 5,5
    np.random.seed(42)
    self.W = self.xavier_initialization(self.n_neurons, n_features)
    # self.v = self.xavier_initialization(self.n_neurons, 1)
    self.v = self.xavier_initialization(self.n_neurons, 1).reshape(-1) # make it vec
    # print(self.W.shape, "Wshape")
```

```python
  def train(self, X, y):
    n_samples, n_features = X.shape #get number of sa
    # print(n_samples.shape)
    # print(n_features.shape)
    self.initialize_weights(n_features) #initialize w

    # print(self.initialize_weights.shape)
    # test = np.dot(X, self.W.T)
    # print(test, "forward propa")
    # print(test.shape, "forward propa")

    batch_size = self.batch
    accuracies = []

    for epoch in range(self.epochs):
      for i in range(0, X.shape[0], batch_size):
        # choose the batch datapoints
        X_batch = X[i:i + batch_size]
        y_batch = y[i:i + batch_size]

        #forward fass
        h1_input, h1_output, h1_pred = self.forward_p

        # compute loss as quadratic fun
        if self.loss_fun == "quadratic":
          loss = y_batch - h1_pred
          # print(loss.shape)

          h1 loss = -loss  # compute the h1 layers' l
```

```python
k = [5,50,200]
kth_quadTrain_Accuracies = []
for neurons in k:
    quadraticNN = NeuralNetwork(n_neurons=neurons, lr=0.001, epochs=10, batch=10, loss_fun="quadratic")
    train_accuracies = quadraticNN.train(X_train,Y_train)
    kth_quadTrain_Accuracies.append(train_accuracies)

    final_test_accuracy = quadraticNN.accuracy(X_test, Y_test)
    print("Quadratic nn for # neuron = ", neurons, " final test Accuracy: ", final_test_accuracy)
```

```
Quadratic nn for # neuron =  5  final test Accuracy:  0.9413
Quadratic nn for # neuron =  50  final test Accuracy:  0.9595
Quadratic nn for # neuron =  200  final test Accuracy:  0.9629
```

```
plot_accuracies(kth_quadTrain_Accuracies, k, title="Training Accuracy, Quadratic Loss")
```



Obs:

With a common epoch =10, batch_size=10, learning rate = 0.001

For quradatic loss function, the accuracy increase as number of neurons increases.

Quadratic nn for # neuron = 5 final test Accuracy: 0.9413

Quadratic nn for # neuron = 50 final test Accuracy: 0.9595

Quadratic nn for # neuron = 200 final test Accuracy: 0.9629

The total number of neuron at 50 and 200 shares a very similar accuracy curve. 200 neuron doing slightly better. However, with 200 neurons the computation duration is a lot longer.

With a learning rate = 0.01, the accuracy stuck at 50%

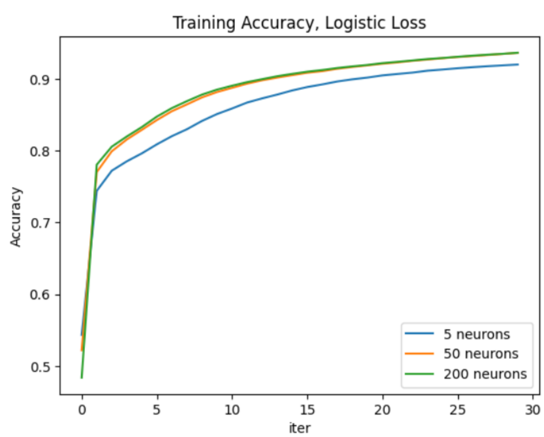
4. (7 pts) Train a neural network classifier with logistic loss, namely ℓ(y, f (x)) = −y log(σ(f (x))) − (1 −
y) log(1 − σ(f (x))) where σ(x) = 1/(1 + e−x) is the sigmoid function. In this case, the hard-thresholding
is applied on top of the sigmoid function, i.e., ˆy = 1σ(f (x))>0.5. Repeat step 3.

```
y_batch = y[i:i + batch_size]

#forward fass
h1_input, h1_output, h1_pred = self.forward_pass(X_batch)

# compute loss as quadratic fun
if self.loss_fun == "quadratic":
  loss = y_batch - h1_pred
  # print(loss.shape)

  h1_loss = -loss  # compute the h1 layers' loss as quardati

# compute loss as logsitc fun
elif self.loss_fun == "logistic":
  # use sigmoid to compute the probability first
  h1_pred_sigmoid = self.sigmoid(h1_pred)
  # then compute the loss based some sigmoid with y_batch th
  loss = y_batch - h1_pred_sigmoid
  # print(loss.shape)

  # compute the h1 layers' loss as sigmod
  #print("logsic before: -loss, h1Pred_Sigmoid, sigDerivaitv
  h1_loss = -loss * h1_pred_sigmoid *(1-h1_pred_sigmoid)  #

  # logsic: h1_loss, -loss, h1Pred_Sigmoid, sigDerivaitve (1
  #print("logsic after: h1_loss, -loss, h1Pred_Sigmoid, sigD

# Backprop for v
dv = np.dot(h1_loss, h1_output) / batch_size
# print(self.v.shape)
```

```
k = [5,50,200]
kth_logTrain_Accuracies = []
for neurons in k:
    logisticNN = NeuralNetwork(n_neurons=neurons, lr=0.001, epochs=10, batch=10, loss_fun="logistic")
    train_accuracies = logisticNN.train(X_train,Y_train)
    kth_logTrain_Accuracies.append(train_accuracies)

    # Print final test accuracy
    final_test_accuracy = logisticNN.accuracy(X_test, Y_test)
    print("Logisctic nn for # neuron = ", neurons, " final test Accuracy: ", final_test_accuracy)
```

```
Logisctic nn for # neuron =  5  final test Accuracy:  0.9226
Logisctic nn for # neuron =  50  final test Accuracy:  0.9377
Logisctic nn for # neuron =  200  final test Accuracy:  0.9372
```

```
plot_accuracies(kth_logTrain_Accuracies, k, title="Training Accuracy, Logistic Loss")
```



For logistic loss function, the accuracy increase as number of neurons increases.

Logisctic nn for # neuron = 5 final test Accuracy: 0.9226

Logisctic nn for # neuron = 50 final test Accuracy: 0.9377

Logisctic nn for # neuron = 200 final test Accuracy: 0.9372

For the logistic neural network, the total number of neuron at 50 and 200 shares a very similar accuracy curve. 200 neurons doing slightly better. However, with 200 neurons the computation duration is a lot longer.

5. (2 pts) Comment on the difference between linear model and neural net. Comment on the differences
between logistic and quadratic loss in terms of optimization and test/train accuracy.

The simple linear classifier has significant lower accuracy compared neural network, however, the computation is very robust. The reason Neural network overcome the linear classifier could be the increasing in complexity. The extra neurons not only helps contribute weight, they also can help contribute weights to strange images. At worst, they may not efficiently contribute the accuracy, with gradient update their corresponding weight to nearly 0 so that they would not hurt the accuracy badly.

The Logistics' accuracy is lower than the Quadratic one, and by observing the plot, at mid of the total epoch, number of neuron at 50 and 200 has no significant difference. I suspect the cause is logistic loss is more complex than quadratic loss because of sigmoid function. By choosing a smaller learning rate = 0.0001, the result is slightly better. I believe it may hit its limitation

```
k = [5,50,200]
kth_logTrain_Accuracies2 = []
for neurons in k:
    nn_model = NeuralNetwork(n_neurons=neurons, lr=0.0001, epochs=100, batch=10, lo
    train_accuracies = nn_model.train(X_train,Y_train)
    kth_logTrain_Accuracies2.append(train_accuracies)

    # Print final test accuracy
    final_test_accuracy = nn_model.accuracy(X_test, Y_test)
    print("Logisctic nn for # neuron = ", neurons, " final test Accuracy: ", final_

# # Plot training accuracy over epochs
# plot_accuracies(train_accuracies, title="Neural Network Training Accuracy - Quadr
```

```
Logisctic nn for # neuron =  5  final test Accuracy:  0.923
Logisctic nn for # neuron =  50  final test Accuracy:  0.9376
Logisctic nn for # neuron =  200  final test Accuracy:  0.9378
```

```
plot_accuracies(kth_logTrain_Accuracies2, k, title="Training Accuracy, Logistic Los
```