Assignment 2

Zahidur Talukder
Student ID- 1001771748

Problem 1

The main requirement of this problem is to implement logistic regression. Logistic regression is a statistical model that in its basic form uses a logistic function to model a binary dependent variable, although many more complex extensions exist. In regression analysis, logistic regression (or logit regression) is estimating the parameters of a logistic model (a form of binary regression).

In this problem, I have used sigmoid function as my activation function and cross entropy for my objective function and finally i performed batch training of my data. Here I will not go for the details theory of the things rather I will discuss about the result of my evaluation.

Here I am attaching the code of my logistic Regression class. Full code is available in my code section.

```
## Code begins here
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

class LogisticRegression:

    def __init__(self):
        pass

    def sigmoid(self, a):
        return 1 / (1 + np.exp(-a))

    def train(self, X, y_true, n_iters, learning_rate):
        n_samples, n_features = X.shape
        self.weights = np.zeros((n_features, 1))
        self.bias = 0
        costs = []
        for i in range(n_iters):
            y_predict = self.sigmoid(np.dot(X, self.weights) + self.bias)
            cost = (- 1 / n_samples) * np.sum(y_true * np.log(y_predict) + (1 - y_true) * (np.log(1 - y_predict)))
            dw = (1 / n_samples) * np.dot(X.T, (y_predict - y_true))
            db = (1 / n_samples) * np.sum(y_predict - y_true)
            self.weights = self.weights - learning_rate * dw
            self.bias = self.bias - learning_rate * db
            costs.append(cost)
            if i % 100 == 0:
                print(f"Cost after iteration {i}: {cost}")

        return self.weights, self.bias, costs

    def predict(self, X):
        y_predict = self.sigmoid(np.dot(X, self.weights) + self.bias)
        y_predict_labels = [1 if elem > 0.5 else 0 for elem in y_predict]
        return np.array(y_predict_labels)[:, np.newaxis]

    def confusion_metrics(self,labels,predictions,threshold):
        true_positive=0;
```

```
        false_positive=0;
        true_negative=0;
        false_negative=0;
        for i in range(len(labels)):
            if labels[i]==1:
                if predictions[i]>=threshold:
                    true_positive+=1;
                else:
                    false_negative+=1;
            else:
                if predictions[i]>=threshold:
                    false_positive+=1;
                else:
                    true_negative+=1;
        tpf=true_positive/(true_positive + false_negative);
        fpf=false_positive/(false_positive + true_negative);
        return tpf,fpf;

    def results(self,labels,predictions):
        TPF=[];
        FPF=[];
        THRESHOLD=[];
        i=0;
        #increemental step size for threshold
        dx_step=0.0002;
        while(i<=1):
            threshold=i;
            tpf,fpf=confusion_metrics(labels,predictions,threshold);
            TPF.append(tpf);
            FPF.append(fpf);
            THRESHOLD.append(threshold);
            i+=dx_step;

        plt.plot(FPF,TPF);
        plt.plot(THRESHOLD,THRESHOLD,'--')
        plt.xlabel("False Positive fraction (FPF)--->")
        plt.ylabel("True Positive fraction (TPF)--->")
        plt.title("ROC Curve")
        plt.show()
        area = np.trapz(TPF, dx=dx_step)
        print("AUC:Area under the ROC curve is", area)
        return;

```

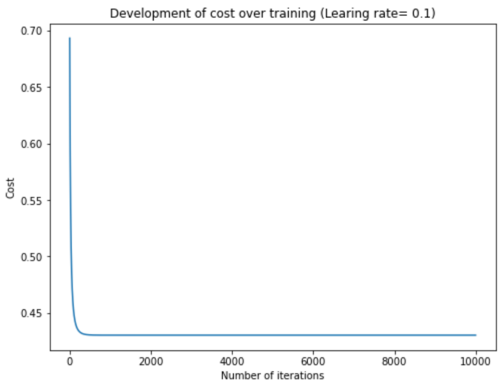
Result Evaluation

Here I have a train function for training the model and predict function for predicting the model based on training in the LogisticRegression Class. During the training period I can control the learning rate. For different learning rate the cost function graph is given below:

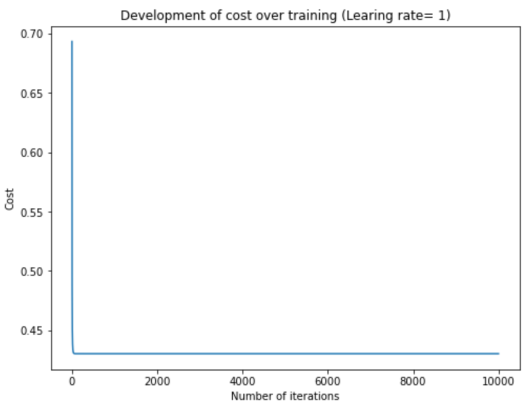
(a) Cost function for learning rate .0001

(b) Cost function for learning rate .001

(c) Cost function for learning rate .01



(d) Cost function for learning rate .1



(e) Cost function for learning rate 1

Figure 1: Cost function for different learning rate

Here from the graph we can see that when the learning rate is small, model needs long time to converge and this is obvious. When the learning rate is 0.0001 system doesn't even converge with 10000 iterations, when its 0.001, its about to converge, when it is 0.01, it converges after about 3000 iterations, when 0,1, it converges after 500 iterations and finally when 1, it converges after 100 iterations.

Here blue line indicate the ROC graph and from the figure we can see that the Area Under Curve (AUC) is 0.810019. From the graph we can also see that the accuracy that we get from our training and testing data set is 82.2 and 83.2 percent respectively. This is the most accuracy that we have got from our data.

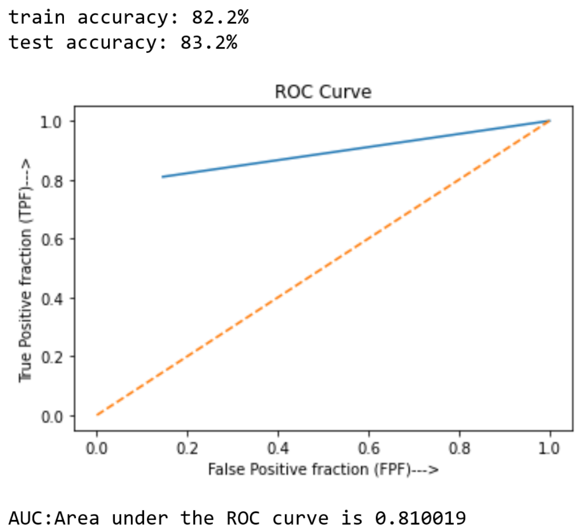


Figure 2: ROC Curve with AUC and Accuracy value

Since logistic regression is a binary regression it performs well for classification problem. Since our data set is from two multivariate normal distribution which has many common merging area so our model could not perform more than 83 percent accurate. But it performs very good for other types of binary data.

Problem 1.2

The main challenge in this problem was to Implement multi-class logistic regression using a soft-max function and cross entropy in MNIST dataset. MNIST-dataset is s 60000 training and 10000 test samples that contain images of hand-written numbers in 28x28 pixels. But since we need to take only first 5(0 to 4) classes, the number of training and testing samples was reduced about half. In this experiment I used hot encoding of my target data and train the model. Here I attached the soft-max regression class and complete code is available in my code section.

```
## Code begins here

from tensorflow.keras.datasets import mnist
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

class SoftmaxRegression:
    def __init__(self):
        pass

    def initialize_param(self,d):
        np.random.seed(1)
        params = {}
        params['w'] = np.random.randn(d,5)/np.sqrt(d)
        params['b'] = np.zeros((5,1))
        return params

    def softmax(self,Z):
        expZ = np.exp(Z - np.max(Z))
        return expZ / expZ.sum(axis=0, keepdims=True)

    def forward(self, params, X):
        w = params['w']
        b = params['b']
        Z = np.dot(w.T,X) + b
        A = self.softmax(Z)
        return A

    def compute_cost(self,A,Y):
        m = Y.shape[1]
        cost = (-1/m)*np.sum(Y * np.log(A + 1e-8))
        return cost

    def backprop(self,X, Y, A):
        m = Y.shape[1]
        dw = (1/m) * np.dot(X, (A - Y).T)
        db = (1/m) * np.sum(A - Y)
        return dw, db

    def optimise(self, params, X, Y, num_iterations, l_rate):
        costs = []
        for i in range(num_iterations):
            A = self.forward(params, X)
            cost = self.compute_cost(A, Y)
            dw, db = self.backprop(X, Y, A)

            params['w'] = params['w'] - l_rate * dw
            params['b'] = params['b'] - l_rate * db

```

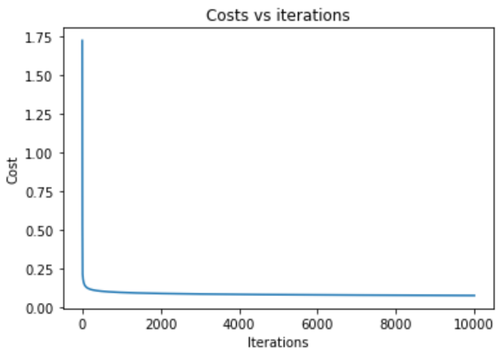
```
            if i % 100 == 0:
                print("Cost after iteration %i : %f " %(i, cost))

            costs.append(cost)
        return params, costs

    def predict(self,params, X, Y):
        w = params['w']
        #print(w.shape,X.shape)
        probs = self.forward(params,X)
        y_hat = np.argmax(probs, axis=0)
        Y = np.argmax(Y, axis=0)
        conf_matrix=self.compute_confusion_matrix(Y, y_hat)
        print("label precision recall")
        for label in range(5):
            print(f"{label:5d} {self.precision(label, conf_matrix):9.3f} {self.recall(label, conf_matrix):6.3f}")

        accuracy = self.accuracy(conf_matrix)
        print("Test Accuracy: ",accuracy)
        return conf_matrix

    def model(self,d, X_train, Y_train, num_interation, l_rate):
        params = self.initialize_param(d)
        params, costs = self.optimise(params, X_train, Y_train, num_interation, l_rate)
        return params, costs

    def compute_confusion_matrix(self,true, pred):
        K = len(np.unique(true)) # Number of classes
        result = np.zeros((K, K))
        for i in range(len(true)):
            result[true[i]][pred[i]] += 1
        return result

    def precision(self,label, confusion_matrix):
        col = confusion_matrix[:, label]
        return confusion_matrix[label, label] / col.sum()

    def recall(self,label, confusion_matrix):
        row = confusion_matrix[label, :]
        return confusion_matrix[label, label] / row.sum()

    def accuracy(self,confusion_matrix):
        diagonal_sum = confusion_matrix.trace()
        sum_of_all_elements = confusion_matrix.sum()
        return diagonal_sum / sum_of_all_elements

```

Result evaluation

The main challenge here was to get the first 5 class of data from MNIST data set. There are 60000 training data and there are 10000 testing data in the dataset for 10 classes. So when I take only 5 classes the number of dataset was about half as we can see in the figure. Again the MNIST dataset is 28*28 pixesls imaage. I convert them to a linear array of 28*28= 784. Again I have done hot encoding to my target data as you can see the shape of the data.

```
Train Input Shape:  (784, 30596)
Train Target Shape:  (5, 30596)
Test Input Shape:  (784, 5139)
Test Target Shape:  (5, 5139)
```

Figure 3: For 5 class of MNIST dataset



(a) Cost Vs Iteration

```
label precision recall
    0     0.988  0.994
    1     0.989  0.991
    2     0.965  0.945
    3     0.963  0.972
    4     0.978  0.981
Test Accuracy:  0.9766491535318156
```

(b) Accuracy, Precision and Recall

Figure 4: Result of the Model for Training and Testing

In this problem I have used the softmax function for activation function as you can see in the code and cross entropy for objective function. The model is trained with train data set and you can see the cost vs iteration of my training data. Although the model converges much earlier I iterated it 10000 times as requirement. Then by checking the test data, The accuracy of the model is 97.66 percent.

For finding the precision and recall of my model I defined the function compute confusion matrix for getting confusion matrix .And from that confusion matrix using the function precision and recall I found out the Precision and Recall for each class level as you can see in the figure, the average precision was about 97 percent and recall was 99 percent.

Problem 3

The main challenge here was to implement an Artificial Neural Network(ANN). I have write the code for making the ANN. Here I have used the same MNIST dataset as required. Here I have used both the Sigmod and ReLu function as activation function. The class for ANN is given here full code is availbale in the code section.

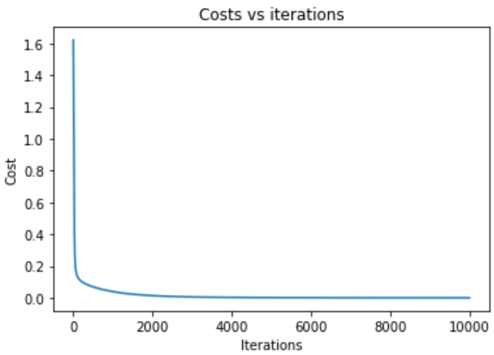
```
## Code begins here

from tensorflow.keras.datasets import mnist
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

class ANN_perceptron:
    def __init__(self):
        pass


    def sigmoid(self, Z):
        A = 1/(1+np.exp(-Z))
        return A

    def relu(self, Z):
        A = np.maximum(0,Z)
        cache = Z
        return A

    def deep_initialize_parameters(self, layer_dims):
        np.random.seed(1)
        parameters = {}
        L = len(layer_dims)            # number of layers in the network

        for l in range(1, L):
            parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1])/ np.sqrt(layer_dims[l-1])# *0.01
            parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))
        return parameters

    def linear_activation_forward(self, A_prev, W, b, activation):
        Z = np.dot(W,A_prev) + b
        linear_cache = (A_prev, W, b)

        if activation == "sigmoid":
            A = sigmoid(Z)
        elif activation == "relu":
            A = relu(Z)

        cache = (linear_cache, Z)
        return A, cache

    def deep_model_forward(self, X, parameters):
        caches = []
        A = X
        L = len(parameters) // 2
        for l in range(1, L):
            A_prev = A
            A, cache = self.linear_activation_forward(A_prev, parameters['W' + str(l)], parameters['b' + str(l)], activation = "relu")
```

```python
            caches.append(cache)
        AL, cache = self.linear_activation_forward(A, parameters['W' + str(L)], parameters['b' + str(L)], activation = "sigmoid")
        caches.append(cache)
        return AL, caches

    def compute_cost(self, AL, Y):
        m = Y.shape[1]
        cost = (1./m) * (-np.dot(Y,np.log(AL).T) - np.dot(1-Y, np.log(1-AL).T))
        cost = np.squeeze(cost)
        assert(cost.shape == ())
        return cost

    def linear_backward(self, dZ, cache):
        A_prev, W, b = cache
        m = A_prev.shape[1]

        dW = 1./m * np.dot(dZ,A_prev.T)
        db = 1./m * np.sum(dZ, axis = 1, keepdims = True)
        dA_prev = np.dot(W.T,dZ)
        return dA_prev, dW, db

    def relu_backward(self, dA, cache):
        Z = cache
        dZ = np.array(dA, copy=True)
        dZ[Z <= 0] = 0
        assert (dZ.shape == Z.shape)
        return dZ

    def sigmoid_backward(self, dA, cache):
        Z = cache
        s = 1/(1+np.exp(-Z))
        dZ = dA * s * (1-s)
        assert (dZ.shape == Z.shape)
        return dZ

    def linear_activation_backward(self, dA, cache, activation):
        linear_cache, activation_cache = cache
        if activation == "relu":
            dZ = self.relu_backward(dA, activation_cache)
            dA_prev, dW, db = linear_backward(dZ, linear_cache)

        elif activation == "sigmoid":
            dZ = self.sigmoid_backward(dA, activation_cache)
            dA_prev, dW, db = linear_backward(dZ, linear_cache)

        return dA_prev, dW, db

    def deep_model_backward(self,AL, Y, caches):
        grads = {}
        L = len(caches) # the number of layers
        m = AL.shape[1]
        Y = Y.reshape(AL.shape)

        dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

        current_cache = caches[L-1]
        grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = self.linear_activation_backward(dAL, current_cache, activation = "
```

```
sigmoid")
        for l in reversed(range(L-1)):
            # lth layer: (RELU -> LINEAR) gradients.
            current_cache = caches[l]
            dA_prev_temp, dW_temp, db_temp = self.linear_activation_backward(grads["dA" + str(l + 2)], current_cache, activation = "relu")
            grads["dA" + str(l + 1)] = dA_prev_temp
            grads["dW" + str(l + 1)] = dW_temp
            grads["db" + str(l + 1)] = db_temp

        return grads

    def update_parameters(self,parameters, grads, learning_rate):
        L = len(parameters) // 2
        for l in range(L):
            parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_rate * grads["dW" + str(l+1)]
            parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate * grads["db" + str(l+1)]

        return parameters

    def deep_layer_model(self,X, Y, layers_dims, learning_rate=0.0075, num_iterations=3000): #lr was 0.009
        costs = []
        parameters = self.deep_initialize_parameters(layers_dims)
        for i in range(0, num_iterations):
            AL, caches = self.deep_model_forward(X, parameters)
            cost = self.compute_cost(AL, Y)
            grads = self.deep_model_backward(AL, Y, caches)
            parameters = self.update_parameters(parameters, grads, learning_rate)
            costs.append(cost)
            if i % 100 == 0:
                print ("Cost after iteration %i: %f" % (i, cost))
        return parameters, costs


    def predict(self,params, X, Y):
        probs, caches = self.deep_model_forward(X, parameters)
        y_hat = np.argmax(probs, axis=0)
        Y = np.argmax(Y, axis=0)
        conf_matrix=self.compute_confusion_matrix(Y, y_hat)
        print("label precision recall")
        for label in range(5):
            print(f"{label:5d} {self.precision(label, conf_matrix):9.3f} {self.recall(label, conf_matrix):6.3f}")

        accuracy = self.accuracy(conf_matrix)
        print("Test Accuracy: ",accuracy)
        return conf_matrix

    def compute_confusion_matrix(self,true, pred):
        K = len(np.unique(true)) # Number of classes
        result = np.zeros((K, K))
        for i in range(len(true)):
            result[true[i]][pred[i]] += 1
        return result

```

```
    def precision(self,label, confusion_matrix):
        col = confusion_matrix[:, label]
        return confusion_matrix[label, label] / col.sum()

    def recall(self,label, confusion_matrix):
        row = confusion_matrix[label, :]
        return confusion_matrix[label, label] / row.sum()

    def accuracy(self,confusion_matrix):
        diagonal_sum = confusion_matrix.trace()
        sum_of_all_elements = confusion_matrix.sum()
        return diagonal_sum / sum_of_all_elements

```

Result evaluation



(a) Cost Vs Iteration

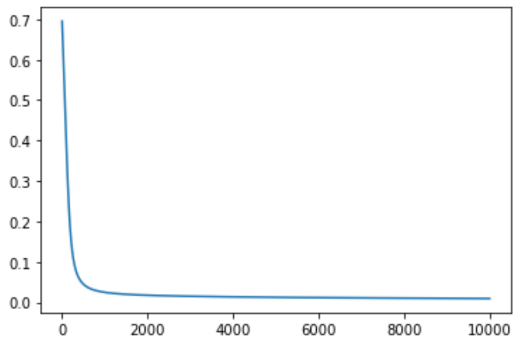
label	precision	recall
0	0.992	0.996
1	0.996	0.994
2	0.979	0.972
3	0.988	0.989
4	0.987	0.991

Test Accuracy: 0.9883245767659078

(b) Accuracy, Precision and Recall

Figure 5: Result of the Model for Training and Testing for ReLU activation function

Here I have used 3 hidden layer with nodes 256, 64,32. These are dense layer and for the first case I have used ReLU function as my activation function. For this case as we can see from the figure I have done 10000 iteration and the precision and recall for each level is given above. Again the model has accuracy of 98.83 percent for test data set. So in this case the system performs better than Multi-class logistic regression.



(a) Cost Vs Iteration

label	precision	recall
0	0.997	0.995
1	0.996	0.996
2	0.984	0.986
3	0.995	0.993
4	0.994	0.994

Test Accuracy: 0.9929947460595446

(b) Accuracy, Precision and Recall

Figure 6: Result of the Model for Training and Testing for Sigmoid activation function

As you can see from the figure, for the Second case I have used Sigmoid function as my activation function. For this case as we can see from the figure I have done 10000 iteration and the precision and recall for each level is given above. Again the model has accuracy of 99.29 percent for test data set. So in this case the system performs better than Multi-class logistic regression as well.

So for my case the Artificial Neural Network(ANN) performs better than Multi-class logistic regression. The main advantage of ANNs over logistic regression models lies in their hidden layers of nodes. In fact, a special ANN with no hidden node has been shown to be identical to a logistic regression model. That was obvious for my case also. And since the data set was large enough, there was no overfitting. Thus in both the cases the model performs really well.