EEL 6814: Homework 2

As part of this assignment, we have used 13-dimensional wine data to classify wines into 3 classes. In this implementation, we have re-used the code from assignment 1 to create a neural network with 13 input neurons, 8 neurons in the hidden layer and 3 neurons in the output layer. For this implementation I have used the k-fold validation technique, splitting the data into 3 equal parts, using 1/3rd of the sample to train the model using the steepest descent algorithm. Now I share the results for various experiments that were conducted over this wine dataset.

1. **Experiment 1** – For this experiment used the exact same dataset and fed it to a neural network with a 13 input neuron layer, a single hidden layer of 8 neurons and 3 output neurons. The most optimal learning rate for this network is 0.19, for this learning rate, we achieve the average error of 57.67% which is bad as seen from the confusion matrix. This model is predicting label 1 for all the samples, thus this is incorrect. The confusion matrix for this experiment is given as follows.

[[0 20 0]

[0 21 0]

[0 18 0]]

2. **Experiment 2** – For this experiment used the linear normalization which is given by the formula

$$x' = (x - x_{min})/(x_{max} - x_{min})$$

For each dimension, we have calculated the minimum and maximum value and then normalized it before training the model. The result was an improvement over the previous experiment, we observe that the average error for this experiment was reduced to 25% from 57.67% which is not bad considering we have not performed too much computation. It is also seen that this model can make some predictions however it still is not higher in terms of accuracy predicted for class 0 and class 2. The confusion matrix and the average error

Average Error : 0.2521547715446509 at epoch 9995

Average Error : 0.2521547651934047 at epoch 9996

Average Error : 0.2521547588435537 at epoch 9997

Average Error : 0.25215475249509767 at epoch 9998

Average Error : 0.25215474614803607 at epoch 9999

[[7 13 0]

[3 16 2]

[12 2 4]]

3. **Experiment 3** – For this experiment, we have used the z-score normalization which is given by the formula

$$x' = (x - \mu)/\sigma$$

In this approach, we have created a mean array and a standard deviation array which holds the mean and standard deviation for all thirteen dimensions, then we have used the above function to create normalized inputs and then used it to train the network. It performed slightly better than the linear normalization and achieved higher classification accuracy.

Average Error : 0.2022480022963275 at epoch 49996

Average Error : 0.20224800227984926 at epoch 49997

Average Error : 0.2022480022633717 at epoch 49998

Average Error : 0.2022480022468947 at epoch 49999

[[19 3 5]

[18 6 8]

[12 1 17]]

4. **Experiment 4** – For this experiment, we have used the one-hot encoding with the categorical cross-entropy function and Adam optimizer. This was built with TensorFlow and Keras. I have used 13 input neurons, 13 neurons in the hidden layer and 3 output neurons. This performed the best and converged to over 95 percent accuracy. Also, for the previous 3 experiments we have used the sigmoid activation function, however with this we have used the ReLU activation function on the input and hidden layer and used the Softmax activation function for the output layer.

Epoch 9996/10000

89/89 [==============================] - 0s 430us/step - loss: 1.5096e-05 - accuracy: 1.0000 - val_loss: 0.4824 - val_accuracy: 0.9551

Epoch 9997/10000

89/89 [==============================] - 0s 415us/step - loss: 1.3285e-05 - accuracy: 1.0000 - val_loss: 0.4979 - val_accuracy: 0.9551

Epoch 9998/10000

89/89 [==============================] - 0s 504us/step - loss: 1.2609e-05 - accuracy: 1.0000 - val_loss: 0.4917 - val_accuracy: 0.9551

Epoch 9999/10000

89/89 [==============================] - 0s 416us/step - loss: 1.3086e-05 - accuracy: 1.0000 - val_loss: 0.4824 - val_accuracy: 0.9551

Epoch 10000/10000

89/89 [==============================] - 0s 501us/step - loss: 1.1723e-05 - accuracy: 1.0000 - val_loss: 0.4959 - val_accuracy: 0.9551

Further, we observed the correctly predicted labels for true positives and true negatives, we observe that in the final iteration, 67 true positives and 139 true negatives were detected with 5 false positives and 5 false negatives which is more than 95 percent accurate.

Epoch 1297/1300

106/106 [==============================] - 0s 436us/step - loss: 0.0180 - true_positives_9: 106.0000 - true_negatives_9: 212.0000 - false_positives_9: 0.0000e+00 - false_negatives_9: 0.0000e+00 - val_loss: 0.4007 - val_true_positives_9: 67.0000 - val_true_negatives_9: 139.0000 - val_false_positives_9: 5.0000 - val_false_negatives_9: 5.0000

Epoch 1298/1300

106/106 [==============================] - 0s 494us/step - loss: 0.0199 - true_positives_9: 105.0000 - true_negatives_9: 211.0000 - false_positives_9: 1.0000 - false_negatives_9: 1.0000 - val_loss: 0.3831 - val_true_positives_9: 68.0000 - val_true_negatives_9: 140.0000 - val_false_positives_9: 4.0000 - val_false_negatives_9: 4.0000

Epoch 1299/1300

106/106 [==============================] - 0s 424us/step - loss: 0.0155 - true_positives_9: 105.0000 - true_negatives_9: 211.0000 - false_positives_9: 1.0000 - false_negatives_9: 1.0000 - val_loss: 0.3418 - val_true_positives_9: 67.0000 - val_true_negatives_9: 139.0000 - val_false_positives_9: 5.0000 - val_false_negatives_9: 5.0000

Epoch 1300/1300

106/106 [==============================] - 0s 484us/step - loss: 0.0135 - true_positives_9: 106.0000 - true_negatives_9: 212.0000 - false_positives_9: 0.0000e+00 - false_negatives_9: 0.0000e+00 - val_loss: 0.3932 - **val_true_positives_9: 67.0000 - val_true_negatives_9: 139.0000 - val_false_positives_9: 5.0000 - val_false_negatives_9: 5.0000**

Enlisted below is the code for this implementation -

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Oct  7 00:30:59 2020

@author: Abhishek Jha

"""

from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import keras
from keras.models import Sequential
from keras.layers import Dense
import numpy as np



class MLP:

    def __init__(self, ip=13, hidden=[8], op=3):

        self.ip = ip
```

```
        self.hidden = hidden
        self.op = op

        layers = [self.ip] + hidden + [self.op]

        self.weights = []
        for i in range(len(layers) - 1):
            w = np.random.rand(layers[i], layers[i+1])
            self.weights.append(w)

        activations = []
        for i in range(len(layers)):
            a=np.zeros(layers[i])
            activations.append(a)
        self.activations = activations

        derivatives = []
        for i in range(len(layers) - 1):
            d=np.zeros((layers[i], layers[i+1]))
            derivatives.append(d)
        self.derivatives = derivatives


    def forward_propogation(self, inputs):

        activations = inputs
        self.activations[0] = inputs

        for i, w in enumerate(self.weights):
            net_inputs = np.dot(activations, w)
```

```
            activations = self._sigmoid(net_inputs)
            self.activations[i+1] = activations
        return activations

    def back_propagate(self, error, flag=False):

        for i in reversed(range(len(self.derivatives))):
            activations = self.activations[i+1]
            delta = error * self._sigmoid_derivative(activations)
            delta_reshaped = delta.reshape(delta.shape[0], -1).T
            current_activations = self.activations[i]
            current_activations_reshaped = current_activations.reshape(current_activations.shape[0],
-1)
            self.derivatives[i] = np.dot(current_activations_reshaped, delta_reshaped)
            error = np.dot(delta, self.weights[i].T)

            if flag:
                print("Derivatives for W{} : {}".format(i, self.derivatives[i]))

        return error

    def steepest_descent(self, learning_rate):
        for i in range(len(self.weights)):
            weights = self.weights[i]
#             print("Original W{} {}".format(i, weights))
            derivatives = self.derivatives[i]
            weights += derivatives * learning_rate
#             print("Updated W{} {}".format(i, weights))
```

```
def _sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def _sigmoid_derivative(self, x):
    return (x * (1.0 - x))

def _mse(self, target, output):
    return np.average((target - output)**2)



def train(self, inputs, targets, epochs, learning_rate):

    for i in range(epochs):
        sum_error = 0
        for inp, target in (zip(inputs, targets)):

            output = self.forward_propogation(inp)

            error = target - output

            self.back_propagate(error)

            self.steepest_descent(learning_rate)

            sum_error += self._mse(target, output)

        av_error = sum_error/len(inputs)
        print("Average Error : {} at epoch {}".format(av_error, i))
```

```
if __name__ == "__main__":



    training = load_wine()
    train_data = np.array(training['data'])
    labels = np.array(training['target'])


    x_train, x_test, y_train, y_test = train_test_split(train_data, labels, test_size=0.33)


    """
    Experiment 1 : Training the model without normalization, splitting 1/3rd of the dataset for training and the rest
    for testing, 13 inputs
    Uuncomment the below section and comment the other 3 sections
    """


    """
    mlp = MLP()
    learning_rate=0.19  #hyper-parameter
    epochs=10000
    mlp.train(x_train, y_train, epochs, learning_rate)


    # Test The Model


    output_test = []
#    print(input_test.shape)
#    t = np.array([2])
    for i in x_test:
```

```
        output = mlp.forward_propogation(i)
#         print(output)
        if output[0] > output[1] and output[0] > output[2]:
            output_test.append(0)
        elif output[1] > output[0] and output[1] > output[2]:
            output_test.append(1)
        elif output[2] > output[1] and output[2] > output[0]:
            output_test.append(2)
        else:
            output_test.append(2)

    print(confusion_matrix(y_test, output_test))

    """

    """
```

Experiment 2 :Training the model after linear normalization, splitting 1/3rd of the dataset for training and the rest

for testing

Uncomment the below section and comment the other 3 sections

```
    """

    """

    min_data = np.array([11.0,0.74,1.36,10.6,70.0,0.98,0.34,0.13,0.41,1.3,0.48,1.27,278])
    max_data = np.array([14.8,5.80,3.23,30.0,162.0,3.88,5.08,0.66,3.58,13.0,1.71,4.00,1680])
```

```
data_r = np.array(max_data-min_data)
normalized_data = np.array((train_data-min_data)/data_r)

xn_train, xn_test, yn_train, yn_test = train_test_split(normalized_data, labels, test_size=0.33)

mlp = MLP()
learning_rate=0.2  #hyper-parameter
epochs=10000
mlp.train(xn_train, yn_train, epochs, learning_rate)

outputn_test = []

for i in xn_test:
    output = mlp.forward_propogation(i)
    if output[0] > output[1] and output[0] > output[2]:
        outputn_test.append(0)
    elif output[1] > output[0] and output[1] > output[2]:
        outputn_test.append(1)
    elif output[2] > output[1] and output[2] > output[0]:
        outputn_test.append(2)
    else:
        outputn_test.append(2)

print(confusion_matrix(yn_test, outputn_test))



"""
```

```
"""

Experiment 3 : Training the model after z-score normalization, splitting 1/3rd of the dataset for training and the rest

for testing

Uncomment the below section and comment the other 3 sections


"""


"""

mean_array = np.array([13.0,2.34,2.36,19.5,99.7,2.29,2.03,0.36,1.59,5.1,0.96,2.61,746])

sd_array = np.array([0.8,1.12,0.27,3.3,14.3,0.63,1.00,0.12,0.57,2.3,0.23,0.71,315])


znormalized_data = np.array((train_data-mean_array)/sd_array)


xz_train, xz_test, yz_train, yz_test = train_test_split(znormalized_data, labels, test_size=0.33)


mlp = MLP()

learning_rate=0.22  #hyper-parameter

epochs=10000

mlp.train(xz_train, yz_train, epochs, learning_rate)


outputz_test = []


for i in xz_test:

    output = mlp.forward_propogation(i)

    if output[0] > output[1] and output[0] > output[2]:

        outputz_test.append(0)

    elif output[1] > output[0] and output[1] > output[2]:
```

```
        outputz_test.append(1)
    elif output[2] > output[1] and output[2] > output[0]:
        outputz_test.append(2)
    else:
        outputz_test.append(2)

print(confusion_matrix(yz_test, outputz_test))



"""

"""

    Experiment 4: Using Adam with categorical crossentropy loss function using Tensorflow and Keras

    Uuncomment the below section and comment the other 3 sections

"""

np.random.seed(2)

classifications = 3

training = load_wine()
train_data = np.array(training['data'])
labels = np.array(training['target'])

x_train, x_test, y_train, y_test = train_test_split(train_data, labels, test_size=0.4)

# one hot encoding
```

```
    y_train = keras.utils.to_categorical(y_train-1, classifications)
    y_test = keras.utils.to_categorical(y_test-1, classifications)



    model = Sequential()
    model.add(Dense(13, input_dim = 13, activation='relu'))
    model.add(Dense(13, activation='relu'))
    model.add(Dense(classifications, activation='softmax'))



#    ac = keras.metrics.Accuracy(name="accuracy", dtype=None)
    tp = keras.metrics.TruePositives(thresholds=None, name=None, dtype=None)
    tn = keras.metrics.TrueNegatives(thresholds=None, name=None, dtype=None)
    fp = keras.metrics.FalsePositives(thresholds=None, name=None, dtype=None)
    fn = keras.metrics.FalseNegatives(thresholds=None, name=None, dtype=None)


    model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=['accuracy',tp,tn,fp,fn])
    model.fit(x_train, y_train, batch_size=5, epochs=1300, validation_data=(x_test, y_test))
```