Yusuf Taha Sarı
22003405

EEE 443 - Neural Networks

Mini Project 3 – HAR

Introduction

This mini project is aimed to create a small neural network dedicated to recognize 6 different human activities from the data collected by the Samsung Galaxy S2 model phone. Human activity data are collected by using the phone's gyroscope and accelerometer. 3rd data is created by subtracting gravity effect from these two data and certain specialties of these 3 data are used as input such as mean, standard deviation, variance, skewness, kurtosis. In total, 561 data for each pattern is collected and inputted. There are 7352 train data and 2947 test data available in the dataset. Neural network contains one input layer, two hidden layers and one output layer. First hidden layer has 300 neurons and second hidden layer has 100 or 200 neurons for different cases. There are 2 different learning coefficients (0.01, 0.001), two different momentum values (0, 0.9) and two different mini batch sizes (0, 50). With this different scenarios, 16 different cases are created in total. Then for the last part, 17th case is created for best result case with dropout rate p = 0.5.

Setup

Needed libraries such as Numpy, time and math are imported. Data file is downloaded and it is read in the program by the open and readlines functions.

```
import numpy as np
from scipy.special import expit
import time
import math

with open("X_train.txt", 'r') as train_data:
    lines_train_data = train_data.readlines()
with open("y_train.txt", 'r') as train_label:
    lines_train_label = train_label.readlines()

with open("X_test.txt", 'r') as test_data:
    lines_test_data = test_data.readlines()
with open("y_test.txt", 'r') as test_label:
    lines_test_label = test_label.readlines()
```

Figure 1: Import - Getting Data

Then, the data is written in a numpy array form for matrix multiplication.

```
train_data_array = np.zeros((len(lines_train_data), 561))

for i, line in enumerate(lines_train_data):

    values = [float(val) for val in line.strip().split()]
    train_data_array[i, :] = values

#print(train_data_array)

train_label_array = np.zeros((len(lines_train_label), 1))

for i in range(len(lines_train_label)):

    train_label_array[i] = lines_train_label[i]


test_data_array = np.zeros((len(lines_test_data), 561))

for k, linee in enumerate(lines_test_data):

    values = [float(val) for val in linee.strip().split()]
    test_data_array[k, :] = values


test_label_array = np.zeros((len(lines_test_label), 1))

for g in range(len(lines_test_label)):

    test_label_array[g] = lines_test_label[g]
```

Figure 2: Creating Numpy Matrix Array

Then the matrix form of the train and test data are controlled.

```
print(train_data_array.shape)
print(train_label_array.shape)
print(test_data_array.shape)
print(test_label_array.shape)
```

```
(7352, 561)
(7352, 1)
(2947, 561)
(2947, 1)
```

Figure 2: Data Matrix Shapes

After checking the data, I created a class called HAR. First function of this class was the initialize function that randomly assign weights to network between (-0.01, 0.01). This function also saved the neuron counts of hidden layers, learning coefficient, mini batch size, and the momentum of the particular case to the class.

```
class HAR:

    def initilaze(self, hidden1_size, hidden2_size, learning_rate, batch_size, epochs, momentum):

        self.IH1_weight = np.random.uniform(-0.01, 0.01, size=(561 ,hidden1_size))
        self.H1H2_weight = np.random.uniform(-0.01, 0.01, size=(hidden1_size, hidden2_size))
        self.H2O_weight = np.random.uniform(-0.01, 0.01, size=(hidden2_size, 6))

        self.learning_rate = learning_rate
        self.hidden1_size = hidden1_size
        self.hidden2_size = hidden2_size
        self.batch_size = batch_size
        self.epochs = epochs
        self.momentum = momentum
        return
```

Figure 3: Initializing Weights and Variables

After these parts, the forward propagation is done with the same logic we used in the previous projects. A function is written which does the forward propagation for one data. Matrix multiplication calculations are done for each multiplication and comments are used to keep track of it. In hidden layers, RELU function is used and for the output layer, SOFTMAX function is used since it creates the probability distribution which is the suitable output value for multiclass recognition neural networks. Lastly, the error is calculated by using the close entropy loss function. A small coefficient is added to each value to avoid the error in the code. Error and output of the propagation is returned from the function.

```
    def forward_one_data(self,one_row_data, label):

        self.label = int(label)
        self.reshaped_data = one_row_data.reshape(1,561)                #(1,561)
        #print(self.reshaped_data.shape)
        self.v_1 = np.dot(self.reshaped_data, self.IH1_weight)      #(1,hidden1_size)
        #print(self.v_1.shape)
        self.y_1 = self.RELU(self.v_1)                              #(1,hidden1_size)
        self.v_2 = np.dot(self.y_1, self.H1H2_weight)               #(1,hidden2_size)
        self.y_2 = self.RELU(self.v_2)                              #(1,hidden2_size)
        #print(self.y_2.shape)
        self.v_3 = np.dot(self.y_2, self.H2O_weight)                #(1,6)
        #print(self.v_3.shape)
        self.y_3 = self.SOFTMAX(self.v_3)                           #(1,6)
        #print(self.y_3.shape)
        self.output = self.y_3                                      #(1,6)
        #print(self.output.shape)
        self.error = np.log(self.output[0, int(label) - 1] + 0.000000000000001)

        return self.error, self.output
```

Figure 4: Forward Propagation Function

Activation functions used in the forward propagation are:

```
def SOFTMAX(self,v_1):

    e_v_1 = np.exp(v_1)
    all_prob = e_v_1.sum()
    result_matrix = np.round(e_v_1/all_prob, 5)

    return result_matrix


def RELU_derivative(self,x):
    return np.where(x < 0, 0, 1)


def RELU(self,x):
    return np.maximum(0,x)
```

Figure 5: Activation Functions

Then the backward propagation function is written after the gradient calculations. For output layer, the calculation is easier due to the SOFTMAX activation function. In the output layer, W_{H20} is calculated by using the formula for both hot coded 1 output class and the other 0 output classes.

$$\frac{\partial E}{\partial w_{1j}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial v_1}\frac{\partial v_1}{\partial w_{1j}}$$
$$\frac{\partial E}{\partial y_i} = -\frac{1}{y_i}$$
$$\frac{\partial y_i}{\partial v_1} = -\frac{e^{v_1}e^{v_i}}{(\sum_j e^{v_y})^2} = -y_i y_1$$
$$\frac{\partial E}{\partial w_{1j}} = \frac{1}{y_i} y_i y_1 x_j = y_1 x_j$$
$$w_{1j} \leftarrow w_{1j} \leftarrow w_{1j} - \eta y_1 x_j$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial v_i}\frac{\partial v_i}{\partial w_{ij}}$$
$$\frac{\partial E}{\partial y_i} = -\frac{1}{y_i}$$
$$\frac{\partial y_i}{\partial v_i} = \frac{e^{v_i}(\sum_j e^{v_y}) - e^{v_i}e^{v_i}}{(\sum_j e^{v_y})^2} = y_i - y_i^2$$
$$\frac{\partial E}{\partial w_{1j}} = -\frac{1}{y_i} y_i (1 - y_i) x_j = -(1 - y_i) x_j$$
$$w_{1j} \leftarrow w_{1j} \leftarrow w_{1j} + \eta(1 - y_i) x_j$$

Figure 6: Gradient Calculation for Wrong Classes (left) - Gradient Calculation for Right Class (right)

The gradient matrix (1, 6) is created according to the given formulas and the matrix is multipiccated with the output of second hidden layer. Comments are also added to calculate matrix multiplication.

```
row = - self.output
row[0, self.label-1] = 1 - self.output[0, self.label-1]     #(1,6)
#print(row.shape)
update_H2O = self.learning_rate * np.dot(self.y_2.T, row)   # (hidden2_size, 1) @ (1,6) = (hidden2_size, 6)
```

Figure 7: Output Layer Gradient

Then for the second hidden layer gradient equation is;

$$grad_{WH2} = \frac{dE}{dW_{H2}} = \frac{dE}{dy_3} * \frac{dy_3}{dv_3} * \frac{dv_3}{dy_2} * \frac{dy_2}{dv_2} * \frac{dv_2}{dW_{H2}}$$

when;

$$\frac{dE}{dy_3} * \frac{dy_3}{dv_3} = grad_O \ , \ \frac{dv_3}{dy_2} = W_O \ , \ \frac{dy_2}{dv_2} = RELU_{derivative(v_2)} \ , \ \frac{dv_2}{dW_{H2}} = y_1$$

Therefore, the code multiplies all the parts.

```
der2 = self.RELU_derivative(self.v_2)                                  # (1,hidden2_size)
diag_der2 = np.diag(der2[0])                                           # (hidden2_size,hidden2_size)
row_2 = np.dot(np.dot(row, self.H2O_weight.T), diag_der2)              # (1, 6) @ (6, h2) @ (h2, h2) = (1, h2)
update_H1H2 = self.learning_rate * np.dot(self.y_1.T, row_2)          # (h1, 1) @ (1, h2) = (h1, h2)
```

Figure 8: Second Hidden Layer Gradient

Lastly, the first hidden layer gradients are calculated in the same way.

$$grad_{WH1} = \frac{dE}{dW_{H1}} = \frac{dE}{dy_3} * \frac{dy_3}{dv_3} * \frac{dv_3}{dy_2} * \frac{dy_2}{dv_2} * \frac{dv_2}{dy_1} * \frac{dy_1}{dv_1} * \frac{dv_1}{dW_{H1}}$$

when;

$$\frac{dE}{dy_3} * \frac{dy_3}{dv_3} * \frac{dv_3}{dy_2} * \frac{dy_2}{dv_2} = \frac{grad_{WH2}}{y_1},$$

and;

$$\frac{dv_2}{dy_1} = W_{H2} \ , \ \frac{dy_1}{dv_1} = RELU_{derivative(v_1)} \ , \ \frac{dv_1}{dW_{H1}} = data$$

Therefore, the code multiplies all the parts.

```
der3 = self.RELU_derivative(self.v_1)                                  # (1,hidden1_size)
diag_der3 = np.diag(der3[0])                                           # (hidden1_size, hidden1_size)
row_3 = np.dot(np.dot(row_2,self.H1H2_weight.T), diag_der3)            # (1, h2) @ (h2, h1) @ (h1, h1) = (1, h1)
update_IH1 = self.learning_rate * np.dot(self.reshaped_data.T, row_3)# (561, 1) @ (1, h1) = (561, h1)
```

Figure 9: First Hidden Layer Gradient

In the backpropagation function, these gradients are calculated. Then, they are multiplied with the learning rate and returned. The full backpropagation function is given as;

```
def backward_propagation(self):

    row = - self.output
    row[0, self.label-1] = 1 - self.output[0, self.label-1]     #(1,6)
    #print(row.shape)
    update_H2O = self.learning_rate * np.dot(self.y_2.T, row)   # (hidden2_size, 1) @ (1,6) = (hidden2_size, 6)

    der2 = self.RELU_derivative(self.v_2)                          # (1,hidden2_size)
    diag_der2 = np.diag(der2[0])                                   # (hidden2_size,hidden2_size)
    row_2 = np.dot(np.dot(row, self.H2O_weight.T), diag_der2)      # (1, 6) @ (6, h2) @ (h2, h2) = (1, h2)
    update_H1H2 = self.learning_rate * np.dot(self.y_1.T, row_2)   # (h1, 1) @ (1, h2) = (h1, h2)

    der3 = self.RELU_derivative(self.v_1)                                 # (1,hidden1_size)
    diag_der3 = np.diag(der3[0])                                          # (hidden1_size, hidden1_size)
    row_3 = np.dot(np.dot(row_2,self.H1H2_weight.T), diag_der3)           # (1, h2) @ (h2, h1) @ (h1, h1) = (1, h1)
    update_IH1 = self.learning_rate * np.dot(self.reshaped_data.T, row_3)# (561, 1) @ (1, h1) = (561, h1)

    return update_H2O, update_H1H2, update_IH1
```

Figure 10: Backpropagation Function.

Then, a small update weights function created. It is used in the train function to update weights with the found deltaweight values.

```
def update_weights(self,c_up_H2O, c_up_H1H2, c_up_IH1):

    self.H2O_weight += c_up_H2O
    self.H1H2_weight += c_up_H1H2
    self.IH1_weight += c_up_IH1
```

Figure 11: Weight Update Function.

Then the whole train algorithm is written which is suitable for all the 16 different cases.

```
def train(self,full_train_data, full_train_label):

    self.prev2_val_error = 9999999999
    self.prev1_val_error = 9999999999
    self.val_error = 0

    for epoch in range(self.epochs):
        batch_counter = 0
        self.one_epoch_error = 0
        cum_up_H2O = 0
        cum_up_H1H2 = 0
        cum_up_IH1 = 0
        for data_index in range(int(full_train_data.shape[0]*0.9)):

            for_error, out = self.forward_one_data(full_train_data[data_index], full_train_label[data_index])
            self.one_epoch_error += for_error
            up_H2O, up_H1H2, up_IH1  = self.backward_propagation()

            cum_up_H2O += up_H2O
            cum_up_H1H2 += up_H1H2
            cum_up_IH1 += up_IH1

            batch_counter +=1

            if (self.batch_size > 0 and batch_counter == self.batch_size ) or self.batch_size == 0:
                self.update_weights(cum_up_H2O, cum_up_H1H2, cum_up_IH1)

                batch_counter = 0

                if self.momentum == 0.09:

                    cum_up_H2O = cum_up_H2O * 0.09
                    cum_up_H1H2 = cum_up_H1H2 * 0.09
                    cum_up_IH1 = cum_up_IH1 * 0.09
                else:
                    cum_up_H2O = cum_up_H2O * 0
                    cum_up_H1H2 = cum_up_H1H2 * 0
                    cum_up_IH1 = cum_up_IH1 * 0


        print(f"Epoch {epoch+1} Train Error: {(-self.one_epoch_error):.7f}", end=" ")

        self.val_error = 0

        for data_index in range(int(full_train_data.shape[0]*0.9),full_train_data.shape[0]):

            for_error, out = self.forward_one_data(full_train_data[data_index], full_train_label[data_index])
            self.val_error += for_error

        print(f" Validation Error: {(-self.val_error):.7f}")


        if (-(self.val_error) > self.prev1_val_error) and (self.prev1_val_error > self.prev2_val_error):
            print("\nDue to Validation Error Data: Early Stoping is happened at", str(epoch+1)+". Epoch.")
            break

        self.prev2_val_error = self.prev1_val_error
        self.prev1_val_error = -(self.val_error)
        self.val_error = 0
```

Figure 12: Train Algorithm

The train function takes whole dataset and labels, trains the network. Finds training error and validation error. It early stops if the validation error increase twice in a row to reduce overfitting.

```
for data_index in range(int(full_train_data.shape[0]*0.9),full_train_data.shape[0]):

    for_error, out = self.forward_one_data(full_train_data[data_index], full_train_label[data_index])
    self.val_error += for_error

print(f" Validation Error: {(-self.val_error):.7f}")


if (-(self.val_error) > self.prev1_val_error) and (self.prev1_val_error > self.prev2_val_error):
    print("\nDue to Validation Error Data: Early Stoping is happened at", str(epoch+1)+". Epoch.")
    break

self.prev2_val_error = self.prev1_val_error
self.prev1_val_error = -(self.val_error)
self.val_error = 0
```

Figure 4: Validation Error Calculation and Early Stopping

The test function is created to find the accuracy percentage and misclassification function is find to see the number of misclassifications.

```
def test(self,full_test_data, full_test_label):
    acc = 0
    total_test = 0
    for data_index in range(full_test_data.shape[0]):
        total_test += 1
        for_error, out = self.forward_one_data(full_test_data[data_index], full_test_label[data_index])
        result = int(np.argmax(out))

        if result == (int(full_test_label[data_index] - 1)):
            acc += 1

    print("\nTest Results are:\n In", total_test, "number of test data\n Accurarcy = %"+ str((acc/total_test)*100))

def missclas_counter(self,full_data,full_label):
    self.class_count_list = [0, 0, 0, 0, 0, 0]

    for data_index in range(full_data.shape[0]):

        for_error, out = self.forward_one_data(full_data[data_index], full_label[data_index])
        result = int(np.argmax(out))

        if result == (int(full_label[data_index] - 1)):
            a = 1
        else:
            self.class_count_list[int(full_label[data_index] - 1)] += 1

    print("Miss clasification stats are:\nClass 1:"+str(self.class_count_list[0]))
    print("Class 2:  "+str(self.class_count_list[1]))
    print("Class 3:  "+str(self.class_count_list[2]))
    print("Class 4:  "+str(self.class_count_list[3]))
    print("Class 5:  "+str(self.class_count_list[4]))
    print("Class 6:  "+str(self.class_count_list[5]))
```

Figure 13: Test and Miss Class Counter Functions

Lastly, the data was in order. Many same class data were next to each other. Therefore, the data matrix is shuffled to have better training experience.

```
per = np.random.permutation(train_data_array.shape[0])
shuffled_train_data_array = train_data_array
shuffled_train_label_array = train_label_array
for old, new in enumerate(per):
    shuffled_train_data_array[new,:] = train_data_array[old,:]
    shuffled_train_label_array[new,:] = train_label_array[old,:]
```

Figure 14: Shuffle Code

Training and Testing

For 16 different cases, 16 different classes are created. Creating different classes help to get weights later to check again.

```
case1 = HAR()
case2 = HAR()
case3 = HAR()
case4 = HAR()
case5 = HAR()
case6 = HAR()
case7 = HAR()
case8 = HAR()
case9 = HAR()
case10 = HAR()
case11 = HAR()
case12 = HAR()
case13 = HAR()
case14 = HAR()
case15 = HAR()
case16 = HAR()

"""
case1 = mini_batch = 0 / lear_rate = 0.001 / N2 = 100 / Momentum = 0
case2 = mini_batch = 0 / lear_rate = 0.001 / N2 = 100 Momentum = 0.9
case3 = mini_batch = 0 / lear_rate = 0.001 / N2 = 200 Momentum = 0
case4 = mini_batch = 0 / lear_rate = 0.001 / N2 = 200 Momentum = 0.9
case5 = mini_batch = 0 / lear_rate = 0.01 / N2 = 100 Momentum = 0
case6 = mini_batch = 0 / lear_rate = 0.01 / N2 = 100 Momentum = 0.9
case7 = mini_batch = 0 / lear_rate = 0.01 / N2 = 200 Momentum = 0
case8 = mini_batch = 0 / lear_rate = 0.01 / N2 = 200 Momentum = 0.9
case9 = mini_batch = 0 / lear_rate = 0.001 / N2 = 100 / Momentum = 0
case10 = mini_batch = 50 / lear_rate = 0.001 / N2 = 100 Momentum = 0.9
case11 = mini_batch = 50 / lear_rate = 0.001 / N2 = 200 Momentum = 0
case12 = mini_batch = 50 / lear_rate = 0.001 / N2 = 200 Momentum = 0.9
case13 = mini_batch = 50 / lear_rate = 0.01 / N2 = 100 Momentum = 0
case14 = mini_batch = 50 / lear_rate = 0.01 / N2 = 100 Momentum = 0.9
case15 = mini_batch = 50 / lear_rate = 0.01 / N2 = 200 Momentum = 0
case16 = mini_batch = 50 / lear_rate = 0.01 / N2 = 200 Momentum = 0.9
"""
```

Figure 15: Creation of Classes

Same training algorithm is run to train networks.

```
case1.initilaze(300, 100, 0.001, 0, 100, 0)
case1.train(shuffled_train_data_array, shuffled_train_label_array)
case1.test(test_data_array, test_label_array)
case1.missclas_counter(train_data_array, train_label_array)
case1.missclas_counter(test_data_array, test_label_array)
```

```
case2.initilaze(300, 100, 0.001, 0, 100, 0.9)
case2.train(shuffled_train_data_array, shuffled_train_label_array)
case2.test(test_data_array, test_label_array)
case2.missclas_counter(train_data_array, train_label_array)
case2.missclas_counter(test_data_array, test_label_array)
```

⋮

```
case16.initilaze(300, 200, 0.01, 0, 100, 0.9)
case16.train(shuffled_train_data_array, shuffled_train_label_array)
case16.test(test_data_array, test_label_array)
case16.missclas_counter(train_data_array, train_label_array)
case16.missclas_counter(test_data_array, test_label_array)
```

Figure 16: Train and Test Code

After training one early stopping example is;

```
Epoch 1 Train Error: 11841.3889342  Validation Error: 1309.1802399
Epoch 2 Train Error: 9836.4876841  Validation Error: 819.3826753
Epoch 3 Train Error: 6199.8036952  Validation Error: 516.5009716
Epoch 4 Train Error: 4525.2020879  Validation Error: 413.1786433
Epoch 5 Train Error: 3831.7790099  Validation Error: 348.7590071
Epoch 6 Train Error: 3376.6312377  Validation Error: 304.9189845
Epoch 7 Train Error: 3067.1795700  Validation Error: 272.0750236
Epoch 8 Train Error: 2789.4172218  Validation Error: 239.9743715
Epoch 9 Train Error: 2332.1099201  Validation Error: 180.6723479
Epoch 10 Train Error: 1607.9861048  Validation Error: 120.1586384
Epoch 11 Train Error: 1242.2379371  Validation Error: 95.4158166
Epoch 12 Train Error: 1079.5137500  Validation Error: 83.2584895
Epoch 13 Train Error: 978.2508895  Validation Error: 75.7677750
Epoch 14 Train Error: 902.4594767  Validation Error: 70.6381676
```

⋮

```
Epoch 59 Train Error: 278.0273310  Validation Error: 28.4856531
Epoch 60 Train Error: 273.1717129  Validation Error: 28.4164277
Epoch 61 Train Error: 270.3646307  Validation Error: 28.0766664
Epoch 62 Train Error: 265.5204379  Validation Error: 27.6640764
Epoch 63 Train Error: 261.8170347  Validation Error: 27.8062943
Epoch 64 Train Error: 258.1046237  Validation Error: 27.8986448

Due to Validation Error Data: Early Stoping is happened at 64. Epoch.
```

Figure 17: Case 1 Result with Early Stopping

Results of the same example is;

```
Test Results are:
 In 2947 number of test data
 Accurarcy = %93.95995928062436
Miss clasification stats are:
Class 1:4
Class 2:  45
Class 3:  17
Class 4:  75
Class 5:  14
Class 6:  23
```

Figure 18: Results of Case 1

Full data training is done for 8 different cases for online learning. Results are;

Case Number	Cases	Last Train Error	Last Val. Error	Early Stopping	Most Misclass	Accuracy
1	LR = 0.001 N2 = 100 Momentum = 0	258.1046237	27.8986448	64. Epoch	Class 4	%93.959959
2	LR = 0.001 N2 = 100 Momentum = 0.9	238.2325619	28.7267594	75. Epoch	Class 4	%93.417034
3	LR = 0.001 N2 = 200 Momentum = 0	232.7826873	27.6387598	75. Epoch	Class 4	%93.272141
4	LR = 0.001 N2 = 200 Momentum = 0.9	231.7675724	26.5585860	71. Epoch	Class 4	%93.688496
5	LR = 0.01 N2 = 100 Momentum = 0	326.7726553	30.0432158	34. Epoch	Class 5	%94.502884
6	LR = 0.01 N2 = 100 Momentum = 0.9	402.3202117	40.2340801	31. Epoch	Class 5	%94.299287
7	LR = 0.01 N2 = 200 Momentum = 0	404.5020003	45.6987614	20. Epoch	Class 5	%92.908042
8	LR = 0.01 N2 = 200 Momentum = 0.9	9857.3427152	1175.4815378	3. Epoch	Class 1	%15.405497

Yusuf Taha Sarı
22003405

Number of misclassifications for each class result is;

Class	Misclass Amount Class 1	Misclass Amount Class 2	Misclass Amount Class 3	Misclass Amount Class 4	Misclass Amount Class 5	Misclass Amount Class 6
1	4	45	17	75	14	23
2	5	47	19	84	13	26
3	4	52	17	83	12	26
4	3	45	17	83	12	26
5	5	21	49	56	31	0
6	9	17	52	46	41	3
7	17	31	43	56	38	24
8	496	437	0	491	532	537

Full data training is also done for mini batch learning with mini batch size of 50. Results are;

Case Number	Cases	Last Train Error	Last Val. Error	Early Stopping	Most Misclass	Accuracy
9	LR = 0.001 N2 = 100 Momentum = 0	498.3859089	50.0147770	39. Epoch	Class 5	%94.502884
10	LR = 0.001 N2 = 100 Momentum = 0.9	351.4810063	45.8595293	54. Epoch	Class 5	%94.774346
11	LR = 0.001 N2 = 200 Momentum = 0	742.2557735	73.6336124	27. Epoch	Class 5	%94.163556
12	LR = 0.001 N2 = 200 Momentum = 0.9	297.1348250	32.0567964	65. Epoch	Class 5	%94.672548
13	LR = 0.01 N2 = 100 Momentum = 0	10143.0167329	1196.2393255	10. Epoch	Class 6	%24.329826
14	LR = 0.01 N2 = 100 Momentum = 0.9	9936.7422448	1133.9824526	6. Epoch	Class 6	%17.848659
15	LR = 0.01 N2 = 200 Momentum = 0	662.3178510	62.2945800	8. Epoch	Class 4	%92.263318
16	LR = 0.01 N2 = 200 Momentum = 0.9	354.2881131	42.6257974	27. Epoch	Class 5	%94.061757

Number of misclassifications for each class result is;

Case	Misclass Amount Class 1	Misclass Amount Class 2	Misclass Amount Class 3	Misclass Amount Class 4	Misclass Amount Class 5	Misclass Amount Class 6
9	13	26	36	34	52	0
10	14	12	47	29	52	0
11	13	25	35	22	77	0
12	9	29	34	45	40	0
13	225	429	16	491	532	537
14	0	471	390	491	532	537
15	26	14	39	131	17	1
16	11	21	48	38	53	4

Due to the strictness of the early stopping condition, Early stopping is happened at the every one of the cases but at different epochs. In several cases, early stopping is happened too early and network couldn't train itself enough to pass %30 accuracy rate. Best case is found as the Case 10. Then, the dropout code is added to the code to train with it.

```
self.dropout_mask = np.random.binomial(1, 1 - 0.5, self.y_2.shape)
self.y_2 *= self.dropout_mask
```

Figure 17: Forward Propagation Dropout Code

```
row_2 *= self.dropout_mask
```

Figure 17: Backward Propagation Dropout Code

With just a small adjustment to the forward and backward codes, dropout is added to the system.

Case 17 training code is;

```
case17 = HAR()
```

```
case17.initilaze(300, 100, 0.001, 50, 100, 0.9)
case17.train_dropout(shuffled_train_data_array, shuffled_train_label_array)
case17.test(test_data_array, test_label_array)
case17.missclas_counter(train_data_array, train_label_array)
case17.missclas_counter(test_data_array, test_label_array)
```

Figure 19: Case 17 Training Code

The results are found as;

```
Epoch 1 Train Error: -9687.3706913 Validation Error: 734.3134834
Epoch 2 Train Error: -5595.6833837 Validation Error: 689.5082871
Epoch 3 Train Error: -4348.7754829 Validation Error: 274.0654953
Epoch 4 Train Error: -2501.1421992 Validation Error: 199.0474993
Epoch 5 Train Error: -1837.6690621 Validation Error: 429.6336830
Epoch 6 Train Error: -1728.9232030 Validation Error: 392.2259469
Epoch 7 Train Error: -1432.8025928 Validation Error: 364.9647366
Epoch 8 Train Error: -1299.9208660 Validation Error: 58.5490234
Epoch 9 Train Error: -1215.3524681 Validation Error: 358.3921010
Epoch 10 Train Error: -1387.4843165 Validation Error: 115.4951983
Epoch 11 Train Error: -983.9151339 Validation Error: 53.5102664
Epoch 12 Train Error: -1054.2268753 Validation Error: 142.2858419
Epoch 13 Train Error: -1070.6428760 Validation Error: 156.6740153

Due to Validation Error Data: Early Stoping is happened at 13. Epoch.

Test Results are:
 In 2947 number of test data
 Accurarcy = %90.83814048184594
Miss clasification stats are:
Class 1:10
Class 2:  1
Class 3:  7
Class 4:  36
Class 5:  493
Class 6:  0
Miss clasification stats are:
Class 1:23
Class 2:  7
Class 3:  40
Class 4:  17
Class 5:  183
Class 6:  0
```

Figure 20: Dropout Result

In the results it is seen that the network has been able to train itself to over %90 accuracy rate. And each epoch took much less time compared to the previous examples since the half of the second layer neurons were shut down.

Yusuf Taha Sarı
22003405

Conclusion

In this project we have created a feed forward neural network which aims to recognize human activations from the data of gyroscope and accelerometer of a mobile phone. Different learning rates and some optimization tactics such as momentum, mini batch learning, and dropout are used to prevent overfitting and get better results with less training. Results are shown that all of the aspects of the system effect the learning and there is no one great way to train a network rather than trying again and again. If we compare this project with the second mini project since, the purposes of the networks are same. Using the statistical values of the movement data is more suitable way to classify actions rather than creating an RNN. Creating a RNN by using time series an BPTT is more complex. And more CPU power is need to train the whole data. Therefore, it is seen that using statistical data can be optimal in HARs.

Most of the 16 cases surpassed the %90 limit which is very high for a basic neural network. It is seen that the early stopping condition should be harder to achieve since some tries are become unsuccessful. Generally, increasing the hidden layer count made training more difficult but also made the results better. Increasing learning rate, and the momentum had network to get to the overfitting situation quicker with fairly good results.

Best case in this project was Case 10: N2 = 100, Learning Rate = 0.001, Momentum = 0.9 and Mini Batch = 50. It was surprising because the case with lower hidden layer neuron count was better. But it can happen sometimes since the initial weights are distributed randomly.

After learning the best case, dropout case with p = 0.5 is implemented. By doing dropout process, training become easier. CPU trained the network faster and also the case 17 surpassed the %90 accuracy limit.