Yusuf Taha Sarı
22003405

EEE 443 - Neural Networks

Mini Project 2 – Motion Recognition by BPTT

Introduction

This mini project is aimed to create a small neural network dedicated to recognize 6 different motions from a data contains 150-time sequences. Motion data base used in this project contains 3000 train and 600 test data all have 150-time sequences and 3 input for each time sequence. Constructed neural network has 1 hidden layer with 50 or 100 neurons and 6 output neurons for each motion. There are 2 learning coefficients (0.05, 0.1) and 2 mini batch sizes (10, 30). All of these different cases are trained and their differences and errors are interpreted in this report. The code is also added to the report with comments on it.

Setup

Needed libraries such as Numpy and time is imported. Data file is downloaded and it is read in the program by special code. A 3D train and 2D label matrices are read.

```
import h5py
import numpy as np
from scipy.special import expit
import time

# Open the .h5 file in read mode
with h5py.File('data-Mini Project 2.h5', 'r') as file:
    # List all the groups in the file
    trX = file['trX'][:]
    tstX = file['tstX'][:]
    trY = file['trY'][:]
    tstY = file['tstY'][:]
```

Figure 1: Import - Getting Data

```
print(trX.shape)
print(trY.shape)
```

```
(3000, 150, 3)
(3000, 6)
```

Figure 2: Data Matrix Shapes

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

```
class RNN:

    def initilaze(self, hidden_size, learning_rate, batch_size):
        self.IH_weight = np.random.uniform(-0.01, 0.01, size=(hidden_size, 3))
        self.HH_weight = np.random.uniform(-0.01, 0.01, size=(hidden_size, hidden_size))
        self.HO_weight = np.random.uniform(-0.01, 0.01, size=(6, hidden_size))

        self.learning_rate = learning_rate
        self.hidden_size = hidden_size
        self.batch_size = batch_size
        return
```

Figure 2: Initializing Weights

After these parts, the forward propagation is done with the same logic as standard Backpropagation for times sequences amount of mini batch size. The train function gets one data (data[index]) and one label (label[index]) for forward and backward propagation for 1 data. The function does forward propagation for 1 batch and calculates error, backpropagates, updates weights and, return the cumulative error. Activation functions are Tanh for the hidden layer and Sigmoid for output layer.

```
    def Tanh_activation(self,x):
        return np.tanh(x)

    def sigmoid_activation(self,x):
        y = expit(x)
        return y
```

Figure 3: Activation Functions

```
def train_one_data(self,data_row,result_one):  # data_row means 1 data element (150,3)

    self.one_row_data = data_row
    self.result_one = result_one
    self.feedback = np.zeros((self.hidden_size, 1)) # feedback.shape = (50,1)

    self.result_son = self.result_one.reshape(1, 6)

    error=0.0
    self.one_data_error = 0.0

    #new_column = np.full((150, 1), -1)
    #data_temp1 = np.concatenate((self.one_row_data, new_column), axis=1)
    #data_temp2 = data_temp1.reshape(150, 4)

    EPSILON = 1e-10

    for k in range(int(150/self.batch_size)):
        self.h_list = []
        self.y_list = []
        OH_gradient = np.zeros((6, self.hidden_size))

        for i in range(self.batch_size):
            data_temp3 = self.one_row_data[self.batch_size * k + i]

            data = data_temp3.reshape(3, 1)

            v_1 = np.dot(self.IH_weight,data) + np.dot(self.HH_weight, self.feedback)# v_1 = (50,1)
            activated = self.Tanh_activation(v_1) # activated = (50,1)
            self.feedback = activated # self.feedback = (50,1)
            transposed_activated = activated.T # transposed_activated.shape = (1, 50)
            #add = np.array([-1])
            #transposed_activated_51 = np.append(transposed_activated,add) # transposed_activated_51.shape = (1, hidden_size+
            #transposed_activated_51.reshape(1, 51)
            v_2 = np.dot(transposed_activated, self.HO_weight.T) # (1,51) * (51,6) = (1, 6)
            y_result = self.sigmoid_activation(v_2)
```

Figure 4: Forward Propagation Part.

Since the OH_gradient calculation doesn't require any BPTT, OH_gradient is calculated right after the forward propagation.

```
OH_gradient += np.dot((y_result - self.result_one).T, transposed_activated)
```

Figure 4: OH Gradients

Error is calculated by using close entropy formula.

```
error= -self.result_one*np.log10(np.clip(y.T, eps,1 - eps))-(1 - self.result_one)*np.log10(np.clip(1 - y.T,eps,1 - eps))
self.one_data_error += np.sum((error))
```

Figure 5: Error Calculation

After calculation of the error and founding gradients of OH weights, the BPTT is done for 10 or 30-time sequences according to the mini batch sizes. To do that, y (output layer's results) values and hi (hidden layer's results) values are stored in a list to use in BPTT.

```
self.h_list.append(activated.T)
self.y_list.append(y_result)
```

Figure 6: Stored Values

For the backpropagation through time, its algorithm is as;

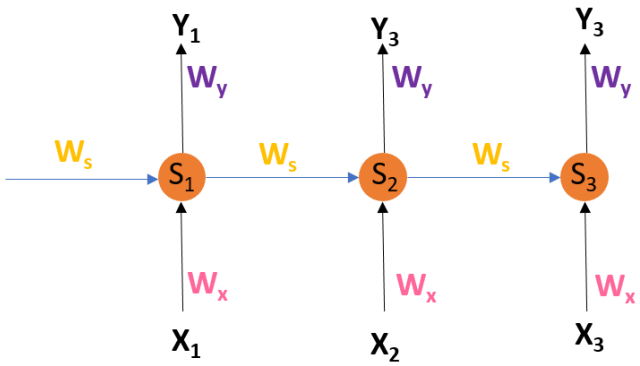

Figure 7: BPTT Algorithm

In backpropagation through time, one gradient is calculated for each mini batch by using the all hi values and errors. The code calculates the common term for the element of each time sequences since next term contains some parts of previous terms.

```
if self.batch_size == 10 or 30: #BACKWARD_10
    self.oh_gradiants = []
    self.hh_gradiants = []
    #HH GRADIENT
    OH_gradient = np.zeros((6, self.hidden_size))
    start_term_temp = np.dot((self.y_list[self.batch_size - 1] - self.result_son),self.HO_weight)
    start_term = start_term_temp * (1-(self.h_list[self.batch_size - 1])**2)
    self.h_list = self.h_list[:-1]

    ind = 0
    HH_gradient = np.zeros((self.hidden_size ,self.hidden_size))
    IH_gradient = np.zeros((self.hidden_size,3))

    for g in reversed(range(1,len(self.h_list))):
        #print(g)
        repeat = start_term
        for j in range(len(self.h_list)-1,len(self.h_list)-ind-1,-1):
            repeat = np.dot(repeat,self.HH_weight) * (1-self.h_list[j]**2)

        HH_gradient += np.dot(repeat.T,self.h_list[g])
        IH_gradient += np.dot(repeat.T,data.T)
        ind += 1
```

Figure 8: BPTT part Code

After calculating the gradients for both HH and IH weights, now gradients can be added to the previous weights for an update. Before updating, gradients are clipped to solve the Vanishing or Exploding Gradient problem.

```
OH_gradient_clipped = np.clip(OH_gradient, -0.5, 0.5)
HH_gradient_clipped = np.clip(HH_gradient, -0.5, 0.5)
IH_gradient_clipped = np.clip(IH_gradient, -0.5, 0.5)
```

Figure 9: Clipping

Clipped gradients are ready to update the old weights before new forward propagation is started. The code updates weights according to the gradients and learning rate.

```
self.HO_weight -= self.learning_rate * OH_gradient_clipped/self.batch_size
self.HH_weight -= self.learning_rate * HH_gradient_clipped/self.batch_size
self.IH_weight -= self.learning_rate * IH_gradient_clipped/self.batch_size
```

Figure 10: Updating

Only the error returns from Train function.

```
return self.one_data_error
```

Figure 11: Returning Error

Since the whole training is ended, code starts to test the network and find accuracies. Accuracy Top-1 function takes the result and desired to find that if the system detected the motion accurately.

```
def acc_calc_top1(self,y, rs):

    max_of_forward = np.argmax(y)
    max_of_label = np.argmax(rs)
    if max_of_forward == max_of_label:
        return 1
    else:
        return 0
```

Figure 12: Accuracy Function

Accuracy Top-2 and Top-3 functions also takes the result and desired to find that if the system's second best and third best guess is correct.

```
def acc_calc_top2(self,y, rs):

    max_of_forward = np.argmax(y)
    y[max_of_forward] = -99999
    second_max_of_forward = np.argmax(y)
    max_of_label = np.argmax(rs)
    if max_of_label ==  second_max_of_forward :
        return 1
    else:
        return 0


def acc_calc_top3(self,y, rs):

    max_of_forward = np.argmax(y)
    y[max_of_forward] = -99999
    second_max_of_forward = np.argmax(y)
    y[second_max_of_forward] = -99999
    third_max_of_forward = np.argmax(y)
    max_of_label = np.argmax(rs)
    if max_of_label == third_max_of_forward:
        return 1
    else:
        return 0
```

Figure 13: Top-2, Top3 Functions

After accuracy functions all the left is the test function. It is the same with the forward propagation function and the only difference is that test data is inputted and backpropagation part is removed.

```
def TEST_one_data(self,data_row,result_one):  # data_row means 1 data element (150,3)

    self.one_row_data = data_row
    self.result_one = result_one
    self.result_son = self.result_one.reshape(1, 6)
    error=0.0
    self.one_data_error = 0.0
    self.feedback = np.zeros((self.hidden_size, 1)) # feedback.shape = (50,1)
    EPSILON = 1e-10

    for k in range(150):

        data_temp3 = self.one_row_data[k]
        data = data_temp3.reshape(3, 1)

        v_1 = np.dot(self.IH_weight,data) + np.dot(self.HH_weight, self.feedback)# v_1 = (50,1)
        activated = self.Tanh_activation(v_1) # activated = (50,1)
        self.feedback = activated # self.feedback = (50,1)
        transposed_activated = activated.T # transposed_activated.shape = (1, 50)
        v_2 = np.dot(transposed_activated, self.HO_weight.T) # (1,51) * (51,6) = (1, 6)
        y_result = self.sigmoid_activation(v_2)

    top_1 = self.acc_calc_top1(y_result.T, self.result_son)
    top_2 = self.acc_calc_top2(y_result.T, self.result_son)
    top_3 = self.acc_calc_top3(y_result.T, self.result_son)
    return top_1, top_2, top_3
```

Figure 14: Test Function

Since the same motion data are put in order to the data file. File gets shuffled for better training. Same process is implemented for labels without changing the labels of data.

```
per = np.random.permutation(trX.shape[0])
shuffled_train_data = trX
shuffled_labels = trY
for old, new in enumerate(per):
    shuffled_train_data[new,:,:] = trX[old,:,:]
    shuffled_labels[new,:] = trY[old,:]
```

Figure 15: Shuffle Code

Training and Testing

Firstly, the main code prints the learning rate batch size and hidden layer size to show which case is trained. The it creates the RNN class and initializes the class accordingly. After initialization system firstly test the random weights to see how well the network performs without training. Then it trains with whole 3000 data for 50 epochs. After training same test procedure is applied to get the accuracy increase after the training.

```
print("Full Data Training\nHidden Size:50\nLearning Rate:0.05\nMini Batch Size:10")
network = RNN()
network.initilaze(50, 0.05, 10)
data_size = 3000

acc_sum_1 = 0
acc_sum_2 = 0
acc_sum_3 = 0
for data_index in range(600):
        acc_top_1, acc_top_2, acc_top_3 = network.TEST_one_data(tstX[data_index], tstY[data_index])
        acc_sum_1 += acc_top_1
        acc_sum_2 += acc_top_2
        acc_sum_3 += acc_top_3

print("\nBEFORE TRAINING:")
print("TOP 1 ACCURACY: %",((acc_sum_1/600)*100))
print("TOP 2 ACCURACY: %",(((acc_sum_1 + acc_sum_2)/600)*100))
print("TOP 3 ACCURACY: %",(((acc_sum_1 + acc_sum_2 + acc_sum_3)/600)*100),"\n")
```

Figure 16: Initialization and Test Before Training

```
startTime = time.time()
for epoch in range(50):
    print("epoch: ", epoch + 1)
    cum_error=0.0
    for data_index in range(data_size):
        onedata_error = network.train_one_data(shuffled_train_data[data_index], shuffled_labels[data_index])
        #print(onedata_error)
        cum_error+=np.sum((onedata_error))

    print("Error after epoch,", epoch + 1, "is:",(cum_error/(data_size*150)))

acc_sum_1 = 0
acc_sum_2 = 0
acc_sum_3 = 0
for data_index in range(600):
        acc_top_1, acc_top_2, acc_top_3 = network.TEST_one_data(tstX[data_index], tstY[data_index])
        acc_sum_1 += acc_top_1
        acc_sum_2 += acc_top_2
        acc_sum_3 += acc_top_3
stopTime = time.time()
print("\n")
print(stopTime - startTime,"Seconds\n")

print("AFTER TRAINING:")
print("TOP 1 ACCURACY: %",((acc_sum_1/600)*100))
print("TOP 2 ACCURACY: %",(((acc_sum_1 + acc_sum_2)/600)*100))
print("TOP 3 ACCURACY: %",(((acc_sum_1 + acc_sum_2 + acc_sum_3)/600)*100))
```

Figure 17: Training and Testing

This given code trains and tests for just one case. The output is:

```
---------------------------------1-------------------------------------
Full Data Training
Hidden Size:50
Learning Rate:0.05
Mini Batch Size:10

BEFORE TRAINING:
TOP 1 ACCURACY: % 17.0
TOP 2 ACCURACY: % 31.166666666666664
TOP 3 ACCURACY: % 48.0
```

Figure 18: Testing Before Training

The epochs and their errors are shown after each epoch.

```
epoch:  1
Error after epoch, 1 is: 10.757270721199973
epoch:  2
Error after epoch, 2 is: 10.73214954997252
epoch:  3
Error after epoch, 3 is: 10.724463915513141
epoch:  4
                    ⋮
epoch:  48
Error after epoch, 48 is: 10.554745681741082
epoch:  49
Error after epoch, 49 is: 10.56105486699108
epoch:  50
Error after epoch, 50 is: 10.554491167178448
```

Figure 19: Epochs

Lastly the training ends and system tests again to see the accuracy.

```
2097.898045063019 Seconds

AFTER TRAINING:
TOP 1 ACCURACY: % 41.333333333333336
TOP 2 ACCURACY: % 46.33333333333333
TOP 3 ACCURACY: % 55.333333333333336
```

Figure 20: Training Time and Test Accurary

In this example Top-1 accuracy increases from %17 to %41 percent.

Yusuf Taha Sarı
22003405

Results

Full data training is done for 8 different cases given in the assignment sheet. Results are;

Cases	Top 1 ACC	Top 2 ACC	Top 3 ACC	CPU TIME
N = 50 LR = 0.05 Batch = 10	%41	%46	%55	2097 sec
N = 50 LR = 0.05 Batch = 30	%35	%50	%67	2809 sec
N = 50 LR = 0.01 Batch = 10	%34	%51	%57	2072 sec
N = 50 LR = 0.01 Batch = 30	%28	%38	%53	2786 sec
N = 100 LR = 0.05 Batch = 10	%38	%43	%53	2590 sec
N = 100 LR = 0.05 Batch = 30	%38	%54	%66	3425 sec
N = 100 LR = 0.01 Batch = 10	%38	%51	%69	2596 sec
N = 100 LR = 0.01 Batch = 30	%26	%41	%58	3430 sec

Full data training results show that Learning Rate is chosen as 0.01 instead of 0.1. Because there was a huge overfitting problem for 0.1 learning rate and this problem even occurs in 0.05 learning rate and it is clearly shown that 0.01 learning rate performs best. The best case is chosen as the Case 1: N = 50 LR= 0.05 Batch = 10. However, for the Top-2 and Top-3 accuracies best case is Case 7. For the c assignment; epoch errors are examined for the best case.

```
Error after epoch, 1 is: 10.835594077485705
Error after epoch, 2 is: 10.83220589200667
Error after epoch, 3 is: 10.828681053563338
Error after epoch, 4 is: 10.824859494871548
Error after epoch, 5 is: 10.820504265325047
Error after epoch, 6 is: 10.815205972755999
Error after epoch, 7 is: 10.808135160331105
Error after epoch, 8 is: 10.797282827165553
Error after epoch, 9 is: 10.777148589930528
Error after epoch, 10 is: 10.734238062119756
Error after epoch, 11 is: 10.699869575052784
Error after epoch, 12 is: 10.687808596959012
Error after epoch, 13 is: 10.687087735870135
Error after epoch, 14 is: 10.685262673163558
Error after epoch, 15 is: 10.684008786060193
Error after epoch, 16 is: 10.681798001415899
Error after epoch, 17 is: 10.68152238410512
Error after epoch, 18 is: 10.680924169525829
Error after epoch, 19 is: 10.684008907141147
Error after epoch, 20 is: 10.684607235461764
Error after epoch, 21 is: 10.67968867925207
Error after epoch, 22 is: 10.67889441164633
Error after epoch, 23 is: 10.679041441203287
Error after epoch, 24 is: 10.677838194468332
Error after epoch, 25 is: 10.678569456213733
Error after epoch, 26 is: 10.677514458697143
Error after epoch, 27 is: 10.677502250872097
Error after epoch, 28 is: 10.677187106493715
Error after epoch, 29 is: 10.67875620497243
Error after epoch, 30 is: 10.677906013274965
Error after epoch, 31 is: 10.678587986728981
Error after epoch, 32 is: 10.679319753005391
Error after epoch, 33 is: 10.680434835020813
Error after epoch, 34 is: 10.679522219179336
Error after epoch, 35 is: 10.678979651079743
Error after epoch, 36 is: 10.679354314807457
Error after epoch, 37 is: 10.679305507298944
Error after epoch, 38 is: 10.679252420775018
Error after epoch, 39 is: 10.679249646837258
Error after epoch, 40 is: 10.679667152918455
Error after epoch, 41 is: 10.680113291309777
Error after epoch, 42 is: 10.681832147270224
Error after epoch, 43 is: 10.68106729597655
Error after epoch, 44 is: 10.682006246045733
Error after epoch, 45 is: 10.68512236349915
Error after epoch, 46 is: 10.685499091201871
Error after epoch, 47 is: 10.683753836041035
Error after epoch, 48 is: 10.679233092559127
Error after epoch, 49 is: 10.677163946409662
Error after epoch, 50 is: 10.678025476997117
```

It is seen that the error shrinks until the 39. Epoch. After 39. Epoch system encounters overfitting and training doesn't make system any better.

Conclusion

In this project we have created a RNN neural network which is specialized to solve the systems that have time sequences. We have learned how to implement BPTT and created RNN by using it. We also learned to shuffle the data and clip it.