EEE 443

Mini Project 2

Human Activity Recognition (HAR) by RNN

Name:Ahmet Bera Özbolat

ID:21902777

Introduction

In our second mini project, we were tasked with creating a Human Activity Recognition (HAR) system that used a Recurrent Neural Network (RNN) to analyze a collection of motion sensor information. The unique feature of the RNN is its feedback loop mechanism, which greatly improves the recognition of sequential data, such as recordings of human activity, by considering them as time series data. This flexible method finds use in speech recognition and medical signal processing, among other fields, going beyond activity recognition. N neurons in the hidden layer with tangent hyperbolic activation function and 6 neurons in the output layer with sigmoid activation were required by the architecture as described. Moreover, multi-category cross-entropy was used as the cost function. The following demonstration serves as an example of the architecture.

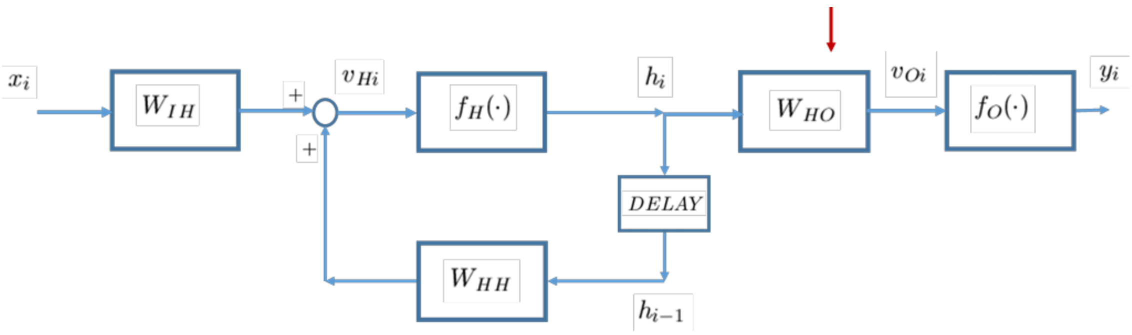

Figure 1: General RNN architecture

Implementation

The following code snippet was used to first access and extract the required data from the dataset, which was in the format of a.h5 file:

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

Figure 2: Data gathering

I began working on the RNN implementation after successfully retrieving the dataset. I wrote an RNN class in Python that initializes the required variables, weights, and biases for each new instance.

```
class HAR:

    def initilaze(self,hidden_size,learning_rate):

        self.IH_weight = np.random.uniform(-0.01, 0.01, size=(hidden_size, 4))
        self.HH_weight = np.random.uniform(-0.01, 0.01, size=(hidden_size, hidden_size))
        self.OH_weight = np.random.uniform(-0.01, 0.01, size=(6, hidden_size+1))
        self.learning_rate = learning_rate
        self.hidden_size = hidden_size

        return
```

Figure 3: Weight initiation

After establishing these variables, I tackled the data's forward propagation. The handbook's instructions detailed how to create a mini-batch logic with sizes of 10 and 30. As such, before being sent into the network, the time series data was divided into batches. Below is an outline of the mathematical foundation for the forward propagation. Afterwards, as this code sample shows, these formulas were converted into the implementation in my RNN class.

$$h(t) = f_H\left(W_{IH}x(t) + W_{HH}h(t-1)\right)$$

$$y(t) = f_O\left(W_{HO}h(t)\right)$$

Figure 4: Forward Propagation Equations

$$E = -d\,\log(y) - (1-d)\,\log(1-y)$$

Figure 5:Binary Cross Entropy Loss

```
def forward_propagation(self,datas,labels,batch_size):
    EPSILON = 1e-10
    grad_HO = 0
    cost = 0
    h_list = np.zeros((self.batch_size,1,self.hidden_size))
    input_list = np.zeros((self.batch_size,1,4))
    output_list = np.zeros((self.batch_size,1,6))
    feedback=np.zeros((self.hidden_size,1))
    error=0.0
    for i in range(self.batch_size):
        data=datas[i].reshape(1,4)
        input_list[i]=data
        v_1= np.dot(self.IH_weight,data.T) + np.dot(self.HH_weight.T,feedback)
        h_i=self.Tanh_activation(v_1)
        feedback=h_i
        h_list[i]=feedback.T
        bias=-1 * np.ones((1, 1))
        biased_h_i=np.concatenate((h_i.T,bias),axis=1)
        y_input = biased_h_i
        v_2= np.dot(self.OH_weight,y_input.T)
        output=self.sigmoid_activation(v_2)
        labels=labels.reshape(6,1)
        grad_HO +=  np.dot((output-labels), biased_h_i)
        error += -labels * np.log10(np.clip(output, EPSILON, 1 - EPSILON))
                             -(1 - labels) * np.log10(np.clip(1 - output, EPSILON, 1 - EPSILON))

        output_list[i]=output.T

    return h_list,output_list,error,grad_HO,input_list,output
```

After making sure that array shapes were consistent throughout the forwarding stage and verifying the desired result, I moved on to putting the backpropagation procedure into practice. Deriving the gradient descent formulas was the first stage, starting with the backpropagation at the output layer.

To compute the gradients required for updating the weights and biases in the network, this required careful reasoning. To aid in the backpropagation process, the resulting formulas were then converted into code, with an initial emphasis on the output layer. This was a critical component in improving the network's learning and optimization as it was being trained.

$$w_j \leftarrow w_j - \eta \frac{\partial E}{\partial w_j}$$

Figure 6: General equation of Weight update

$$\begin{aligned}\frac{\partial L}{\partial W_{yh}} &= \sum_t^T \frac{\partial L_t}{\partial W_{yh}} \\ &= \sum_t^T \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial o_t} \frac{\partial o_t}{\partial W_{yh}} \\ &= \sum_t^T (\hat{y}_t - y_t) \otimes h_t\end{aligned}$$

Figure 7: Output gradients

$$\frac{\partial L}{\partial W_{hh}} = \sum_t^T \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}$$

Figure 8:Whh Gradient formulas

And the following equation is derived,

Equation 1

$$(y_n - d_n) W_{HO} (1 - {h_n}^2) [h_{n-1} + W_{HH}^T (1 - {h_{n-1}}^2) h_{n-2} \ldots..]$$

$$\frac{\partial L}{\partial W_{xh}} = \sum_{t}^{T} \sum_{k=1}^{t+1} \frac{\partial L_{t+1}}{\partial \hat{y}_{t+1}} \frac{\partial \hat{y}_{t+1}}{\partial h_{t+1}} \frac{\partial h_{t+1}}{\partial h_k} \frac{\partial h_k}{\partial W_{xh}}$$

Figure 9: Wih Gradient formulas

And the following equation is derived,

Equation 2

$$(y_n - d_n)W_{HO}\left(1 - {h_n}^2\right)[x_{n-1} + W_{HH}^{T}(1 - {x_{n-1}}^2)x_{n-2} \ldots..]$$

$$\prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j} = \frac{\partial h_{t+1}}{\partial h_k} = \frac{\partial h_{t+1}}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_{k+1}}{\partial h_k}$$

Figure 10:Chain rule of h(i)/h(k)

Summing the partial derivatives of the cost function calculated at each time step allowed for the computation of the overall gradient. This algorithm is represented by the pertinent code snippet that is given below.

The system was ready to update its weights at the end of each batch when the gradients were computed. But there was also the occasional observation that gradients could suddenly blow up. These gradients were adjusted before being added to the weights as a preventative precaution against divergence, guaranteeing a stable update process inside the system.

```python
def gradient(self,y_n,d_n,h_list,x_list): #BPTT algorithm for updating WIH and WHH
    #h_list = [h0,h1,....,hn]
    h_n = h_list[-1]

    common = np.dot((y_n.T - d_n),self.OH_weight [:, :-1]) # 1x6 @ 6xN -> 1xN

    initial_term = common * (1-h_n**2)

    h_list = h_list[:-1] #Removing the h_n data from the list for convention

    index = 0
    grad_HH = np.zeros((self.hidden_size ,self.hidden_size))
    grad_IH = np.zeros((self.hidden_size,4))
    grad_bias1 = 0

    #Implementation of the derivative chain
    for i in reversed(range(1,len(h_list))): # i = n-1,n-2 .... , 0
        term = initial_term
        for j in range(len(h_list)-1,len(h_list)-index-1,-1): # j = n-1,....,n-1-index

            term = np.dot(term,self.HH_weight) * (1-h_list[j]**2) #term = (term @ WHH) * h[j]

        grad_HH += np.dot(term.T,h_list[i])
        #print("Delta Grad_HH:",np.dot((common * term).T,h_list[i]))
        grad_IH += np.dot( term.T,x_list[i])
        grad_bias1 += np.mean( term * -1)
        index += 1

    return grad_IH,grad_HH
```

Figure 11: Gradient function for Whh and Wih

In the training algorithm, I have calculated the cost and the training error to see whether my training is overfitting or not. After spend a lot of time on training i was keep getting the same error so i solved this problem by mixing the datas.

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

Figure 12:Data shuffle code

```python
def Train(self,data,label,batch_size):
    startTime = time.time()
    correct_predictions2=0
    total=0
    correct_predictions3=0
    self.batch_size=batch_size
    for epoch in range(15):
        print("epoch: ", epoch + 1)
        cum_error=0.0

        correct_predictions2=0
        for idx in range(1500,2000):
            sample_data=data[idx,:,:]
            bias=-1 * np.ones((sample_data.shape[0], 1))
            biased_sample=np.hstack((sample_data,bias))
            self.batch_error=0.0
            correct_predictions1=0
            for time_idx in range(int(150/self.batch_size)):
                total+=150/self.batch_size
                datas=biased_sample[int(self.batch_size*time_idx):int(self.batch_size*(time_idx+1))]
                labels=label[idx]
                h_list,output_list,error,grad_HO,input_list,output,correct_predictions=self.forward_propagation(datas,
                self.batch_error+=np.sum(error)
                correct_predictions1+=correct_predictions
                self.delta_weights=np.zeros((6, 50))

                grad_IH,grad_HH=self.gradient(output,labels,h_list,input_list)
                max_gradient_norm = 0.5

                grad_HO_clipped = np.clip(grad_HO, -max_gradient_norm, max_gradient_norm)
                grad_HH_clipped = np.clip(grad_HH, -max_gradient_norm, max_gradient_norm)
                grad_IH_clipped = np.clip(grad_IH, -max_gradient_norm, max_gradient_norm)
                self.OH_weight -= self.learning_rate * grad_HO/batch_size
                self.HH_weight -= self.learning_rate * grad_HH_clipped/batch_size
                self.IH_weight -= self.learning_rate * grad_IH_clipped/batch_size

            #self.batch_error=(self.batch_error/batch_size)
            correct_predictions2+=correct_predictions1
            cum_error+=np.sum((self.batch_error/50))
            #print("Epoch error {i} {error}".format(i=epoch,error=batch_error))
        correct_predictions3+=correct_predictions2
        print("Error per epoch",cum_error)
    print("Train error %",100*correct_predictions3/total)
```

Figure 13: Train Function

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

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

def Tanh_activation_derivative(self,x):
    return (1/2)*(1-x*x)

def sigmoid_activation_derivative(self,x):
    return x-x*x
```

Figure 14: Activation functions

After training the RNN i have write a code to test the network with test datas, to able to see success of matching the class of top1,top2,top3.

After started to train the network i decided the use 15 epochs because 50 epochs leads to overfitting and also use 500 data.

	Batch Size	Hidden Size	Learning Rate	Average Training Accuracy	Top 1 Class Accuracy	Top 2 Class Accuracy	Top 3 Class Accuracy
Case1	10	50	0.001	32.8	36.08	47.921	61.57
Case2	10	50	0.005	37.9	14.97	33.45	49.55
Case3	30	50	0.001	23	16.666	33.333	50
Case4	30	50	0.005	29	35.667	47.43	59.21
Case5	10	100	0.001	33.4	36.092	48.11	61.55
Case6	30	100	0.005	41.2	16.634	37.315	50
Case7	10	100	0.001	23.1	16.666	33.33	50
Case 8	30	100	0.005	30	35.68	47.06	57.54

After finding the results i have trained ta network with full data so that i can see if it’s overfitting or not.

	Batch Size	Hidden Size	Learning Rate	Average Training Accuracy	Top 1 Class Accuracy	Top 2 Class Accuracy	Top 3 Class Accuracy
Case1	10	50	0.001	35.54	17.86	32.93	51.83
Case2	10	50	0.005	40.22	18.32	51.14	64.42
Case3	30	50	0.001	30.2	37.45	47.69	65.21
Case4	30	50	0.005	33.9	16.92	33.33	50
Case5	10	100	0.001	38	16.668	33.336	50,004

Unfortunaetly, I did not have time to train the network in other cases but we can conclude to some point my comparing the results of mini data size and full data size training of other cases. And it seems that Case 5 seems to be best of all among them.

By looking at the results of training with full data size and mini data size. We can see that changing the learning rates and batch size cause to overfitting of training. Best case seems to be 50 hidden layer size and 30 batch size and 0.001 learning rate.

```python
def data_test(W_hh,W_oh,W_ih,hidden_size):
    top1 = 0
    top2 = 0
    top3 = 0
    misclass = 0
    total = 0
    feedback = np.zeros((1,hidden_size))
    for test_idx in range(len(testdata)):
        test_data_idx = testdata[test_idx]
        for time_idx in range(150):
            test_reshaped = np.reshape(test_data_idx[time_idx], (1,3))
            biased_data = np.append(test_reshaped,-1)
            biased_data = np.reshape(biased_data, (1,4))#reshape biased data
            v_t = np.dot(biased_data, W_ih.T) + np.dot(feedback, W_hh.T)
            h_test = tanh(v_t)
            feedback = h_test
            biased_feedback = np.append(h_test,-1)
            v_ot = np.dot(biased_feedback, W_oh.T)
            output = sigmoid(v_ot)
            output = np.reshape(output, (1,6))
            desired = testlabel[test_idx]
            desired = np.reshape(desired, (1,6))
            output_list1 = output.copy()
            output_list1 = np.reshape(output_list1, np.shape(output))
            output_list1[0,np.argmax(output_list1)] = -9999999999999999
            output_list2 = output_list1.copy()
            output_list2 = np.reshape(output_list2, np.shape(output))
            output_list2[0,np.argmax(output_list2)] = -9999999999999999
            if np.argmax(output) == np.argmax(desired):
                total += 1
                top1 += 1
                top2 += 1
                top3 += 1
            elif np.argmax(output_list1) == np.argmax(desired):
                total += 1
                #print(oi_test)
                top2 += 1
                top3 += 1
            elif np.argmax(output_list2) == np.argmax(desired):
```

Figure 15:Data test p1.

```python
            v_ot = np.dot(biased_feedback, W_oh.T)
            output = sigmoid(v_ot)
            output = np.reshape(output, (1,6))
            desired = testlabel[test_idx]
            desired = np.reshape(desired, (1,6))
            output_list1 = output.copy()
            output_list1 = np.reshape(output_list1, np.shape(output))
            output_list1[0,np.argmax(output_list1)] = -9999999999999999
            output_list2 = output_list1.copy()
            output_list2 = np.reshape(output_list2, np.shape(output))
            output_list2[0,np.argmax(output_list2)] = -9999999999999999
            if np.argmax(output) == np.argmax(desired):
                total += 1
                top1 += 1
                top2 += 1
                top3 += 1
            elif np.argmax(output_list1) == np.argmax(desired):
                total += 1
                #print(oi_test)
                top2 += 1
                top3 += 1
            elif np.argmax(output_list2) == np.argmax(desired):
                total += 1
                top3 += 1
            else:
                total += 1
                misclass += 1
    print("Top-1 Accuracy: %", 100*(top1/total))
    print("Top-2 Accuracy: %", 100*(top2/total))
    print("Top-3 Accuracy: %", 100*(top3/total))
```

Figure 16: Data test p2.

Conclusion

This project investigated the benefits of RNN architecture, particularly in sequence recognition, by utilizing its feedback loop in hidden layers. It was centered on building a system for recognizing human activity out of motion sensor data that included six different movements. Although there were several difficulties in implementing the network's architecture and training using Python and a Truncated BPTT algorithm, the training procedure was successful up to a tolerable accuracy level. Optimal performance was achieved with a hidden layer size of 50, a learning rate of 0.01, and a batch size of 30 after the network was tested with various parameter values. All of these instances showed overfitting, though, which prompted research into mitigating techniques such epoch size reduction, which showed increased accuracy with less training. The unpredictable nature of the hidden layers caused debugging complexity, yet in spite of these difficulties, the project was completed to a high standard.

References

Mma. "Backpropagation Through Time for Recurrent Neural Network." Mustafa Murat ARAT, February 7, 2019. https://mmuratarat.github.io/2019-02-07/bptt-of-rnn.

Appendix

RNN Class code:

```
class HAR:


  def initilaze(self,hidden_size,learning_rate):


    self.IH_weight = np.random.uniform(-0.01, 0.01, size=(hidden_size, 4))

    self.HH_weight = np.random.uniform(-0.01, 0.01, size=(hidden_size, hidden_size))

    self.OH_weight = np.random.uniform(-0.01, 0.01, size=(6, hidden_size+1))

    self.learning_rate = learning_rate

    self.hidden_size = hidden_size


    return


  def forward_propagation(self,datas,labels,batch_size):
```

```
        EPSILON = 1e-10
        grad_HO = 0
        cost = 0
        h_list = np.zeros((self.batch_size,1,self.hidden_size))
        input_list = np.zeros((self.batch_size,1,4))
        output_list = np.zeros((self.batch_size,1,6))
        feedback=np.zeros((self.hidden_size,1))
        error=0.0
        correct_predictions=0
        for i in range(self.batch_size):
            data=datas[i].reshape(1,4)
            input_list[i]=data
            v_1= np.dot(self.IH_weight,data.T) + np.dot(self.HH_weight.T,feedback)
            h_i=self.Tanh_activation(v_1)
            feedback=h_i
            h_list[i]=feedback.T
            bias=-1 * np.ones((1, 1))
            biased_h_i=np.concatenate((h_i.T,bias),axis=1)
            y_input = biased_h_i
            v_2= np.dot(self.OH_weight,y_input.T)
            output=self.sigmoid_activation(v_2)
            labels=labels.reshape(6,1)
            correct_predictions += np.sum(np.argmax(output, axis=0) == np.argmax(labels, axis=0))
            grad_HO +=  np.dot((output-labels), biased_h_i)
            error += -labels * np.log10(np.clip(output, EPSILON, 1 - EPSILON)) -(1 - labels) *
np.log10(np.clip(1 - output, EPSILON, 1 - EPSILON))


            output_list[i]=output.T


        return h_list,output_list,error,grad_HO,input_list,output,correct_predictions
```

```
def gradient(self,y_n,d_n,h_list,x_list): #BPTT algorithm for updating WIH and WHH
  #h_list = [h0,h1,....,hn]
  h_n = h_list[-1]

  common = np.dot((y_n.T - d_n),self.OH_weight [:, :-1]) # 1x6 @ 6xN -> 1xN

  initial_term = common * (1-h_n**2)

  h_list = h_list[:-1] #Removing the h_n data from the list for convention

  index = 0
  grad_HH = np.zeros((self.hidden_size ,self.hidden_size))
  grad_IH = np.zeros((self.hidden_size,4))
  grad_bias1 = 0

  #Implementation of the derivative chain
  for i in reversed(range(1,len(h_list))): # i = n-1,n-2 .... , 0
    term = initial_term
    for j in range(len(h_list)-1,len(h_list)-index-1,-1): # j = n-1,....,n-1-index

      term = np.dot(term,self.HH_weight) * (1-h_list[j]**2) #term = (term @ WHH) * h[j]

    grad_HH += np.dot(term.T,h_list[i])
    #print("Delta Grad_HH:",np.dot((common * term).T,h_list[i]))
    grad_IH += np.dot( term.T,x_list[i])
    grad_bias1 += np.mean( term * -1)
    index += 1
```

```
    return grad_IH,grad_HH


  def output_backward(self,dWoh):

    self.OH_weight= self.OH_weight + self.learning_rate*dWoh

  def hh_backward(self,gradient_list):
    self.HH_weight = self.HH_weight + self.learning_rate*gradient_list

  def IH_backward(self,gradient_list):
    self.IH_weight = self.IH_weight + self.learning_rate*gradient_list

  def Train(self,data,label,batch_size):
    startTime = time.time()
    correct_predictions2=0
    total=0
    correct_predictions3=0
    self.batch_size=batch_size
    for epoch in range(15):
      print("epoch: ", epoch + 1)
      cum_error=0.0

      correct_predictions2=0
      for idx in range(3000):
        sample_data=data[idx,:,:]
        bias=-1 * np.ones((sample_data.shape[0], 1))
        biased_sample=np.hstack((sample_data,bias))
        self.batch_error=0.0
        correct_predictions1=0
        for time_idx in range(int(150/self.batch_size)):
```

```
        total+=150/self.batch_size
        datas=biased_sample[int(self.batch_size*time_idx):int(self.batch_size*(time_idx+1))]
        labels=label[idx]

h_list,output_list,error,grad_HO,input_list,output,correct_predictions=self.forward_propagation(dat
as, labels,self.batch_size)
        self.batch_error+=np.sum(error)
        correct_predictions1+=correct_predictions
        #print("Batch error {i} {error}".format(i=time_idx,error=error))
        self.delta_weights=np.zeros((6, 50))


        grad_IH,grad_HH=self.gradient(output,labels,h_list,input_list)
        #print(list_gradient[1].shape)
        #grad_IH,grad_HH = self.calculate_grad(y_i,label,h_list,x_list)
        max_gradient_norm = 0.5  # Adjust this threshold as needed


        # Inside your weight update step
        grad_HO_clipped = np.clip(grad_HO, -max_gradient_norm, max_gradient_norm)
        grad_HH_clipped = np.clip(grad_HH, -max_gradient_norm, max_gradient_norm)
        grad_IH_clipped = np.clip(grad_IH, -max_gradient_norm, max_gradient_norm)


        #print("---------------- Gradients---------------")
        #print("Sum of Unclipped Grad_HH:",np.sum(grad_HH),"Sum of Clipped
Grad_HH:",np.sum(grad_HH_clipped))
        #print("Sum of Unclipped Grad_IH:",np.sum(grad_IH),"Sum of Clipped
Grad_IH:",np.sum(grad_IH_clipped))


        #Updating the weights at the end of the batch
        self.OH_weight -= self.learning_rate * grad_HO/batch_size
        self.HH_weight -= self.learning_rate * grad_HH_clipped/batch_size
        self.IH_weight -= self.learning_rate * grad_IH_clipped/batch_size
```

```
            #self.batch_error=(self.batch_error/batch_size)
            correct_predictions2+=correct_predictions1
            cum_error+=np.sum((self.batch_error/50))
            #print("Epoch error {i} {error}".format(i=epoch,error=batch_error))
        correct_predictions3+=correct_predictions2
        print(cum_error)
    print("Train error",100*correct_predictions3/total)



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

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

def Tanh_activation_derivative(self,x):
    return (1/2)*(1-x*x)

def sigmoid_activation_derivative(self,x):
    return x-x*x
```