Visual Recognition Part-2

REPORT Assignment-1

Name : Hussnain Ashraf
Roll No. : MT2021055
Semester : 2nd
Batch : 2021-2023

1. Derive expressions for $\partial L/\partial \mathbf{b_h}$ and $\partial L/\partial \mathbf{by}$ for the RNN discussed in class.Include the derived bias update equations in the RNN code shared(RNN_from_Scratch.ipynb) and train the RNN for sentence/word. Record relevant observations during training.

Introduction:-

A recurrent neural network(RNN) is a class of neural network that is specialized for processing sequential data. RNNs are called recurrent because they perform the exact same task for every element of a sequence, with the output being dependent on the previous computations. It is the first algorithm that remembers its input due to an
internal memory and thus is suited for problems that involve sequential data.

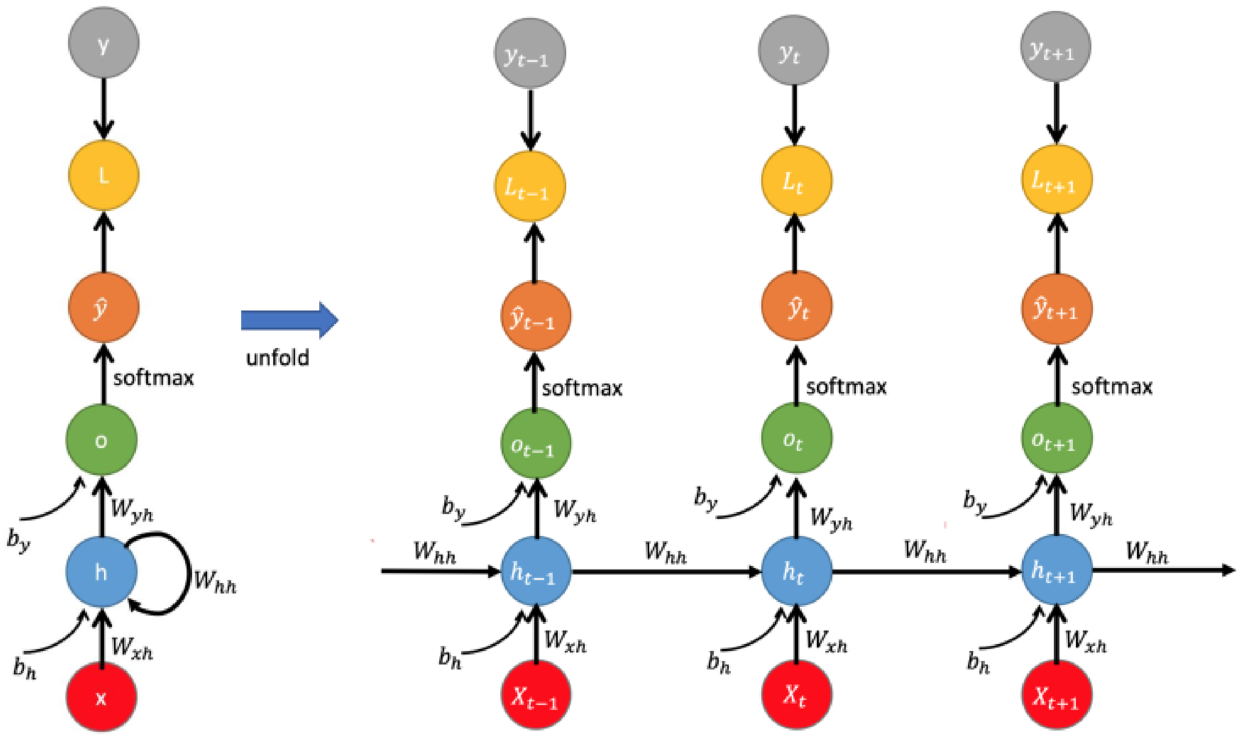
Architecture of RNN:-

- **Input:** x(t) is the input to the network at time step t.

- **Weights**: The input to hidden connection is parameterised by $\mathbf{W_{xh}}$, hidden to hidden layer by $\mathbf{W_{hh}}$, and hidden to output by $\mathbf{W_{yh}}$. Also we have bias terms $\mathbf{b_h}$ and $\mathbf{b_y}$. All these $\mathbf{(W_{xh} , W_{hh} , W_{yh} , b_h , b_y)}$ are shared across time.

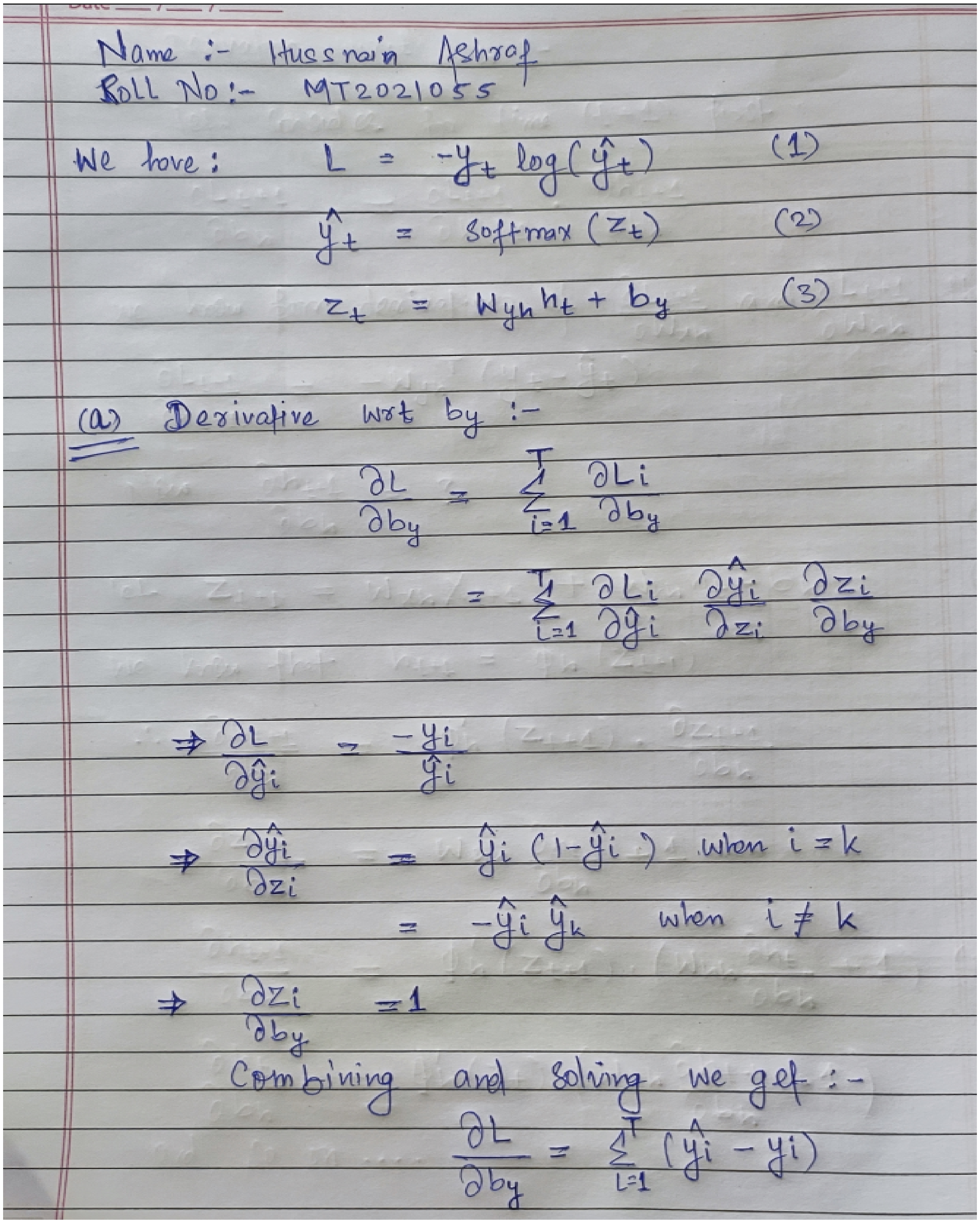
- **Hidden state: h(t)** is the hidden state at **time t** and acts as the network's memory.
Given $\mathbf{h(t-1) , W_{xh} , W_{hh}}$, and some activation function function f, h(t) can be calculated as: $\mathbf{h(t) = f(W_{xh}\, x(t) + W_{hh}\, h(t-1) + b_h)}$. The activation function **f can be tanh, Relu**,etc,

- **Output:** o(t) is the output logit which is computed as: o(t) = $\mathbf{W_{yh}\, h_t}$ +b.
y(t) can be calculated as : y(t) = softmax(o(t)).



RNN-Graph
With Loss Function

Derivation:-



Name :- Hussnain Ashraf

Roll No :- MT2021055

We have:

$$L = -y_t \log(\hat{y}_t) \qquad (1)$$

$$\hat{y}_t = \text{softmax}(z_t) \qquad (2)$$

$$z_t = W_{yh} h_t + b_y \qquad (3)$$

(a) Derivative wrt b_y :-

$$\frac{\partial L}{\partial b_y} = \sum_{i=1}^{T} \frac{\partial L_i}{\partial b_y}$$

$$= \sum_{i=1}^{T} \frac{\partial L_i}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i} \frac{\partial z_i}{\partial b_y}$$

$$\Rightarrow \frac{\partial L}{\partial \hat{y}_i} = \frac{-y_i}{\hat{y}_i}$$

$$\Rightarrow \frac{\partial \hat{y}_i}{\partial z_i} = \hat{y}_i (1 - \hat{y}_i) \quad \text{when } i = k$$

$$= -\hat{y}_i \hat{y}_k \quad \text{when } i \neq k$$

$$\Rightarrow \frac{\partial z_i}{\partial b_y} = 1$$

Combining and solving we get :-

$$\frac{\partial L}{\partial b_y} = \sum_{i=1}^{T} (\hat{y}_i - y_i)$$

b) Derivation w.r.t b_h :-

let's consider for time $(t+1)$ first :

$$\frac{\partial L_{t+1}}{\partial b_h} = \frac{\partial L_{t+1}}{\partial h_{t+1}} \cdot \frac{\partial h_{t+1}}{\partial b_h}$$

we know from derivations of $\frac{\partial L_{t+1}}{\partial W_{xh}}$ and $\frac{\partial L_{t+1}}{\partial W_{hh}}$ that

$$\frac{\partial L_{t+1}}{\partial h_{t+1}} = -W_{yh}^T (y_t - \hat{y}_t)$$

Now $\frac{\partial h_{t+1}}{\partial b_h} = ?$

let $Z_{t+1} = W_{xh} X_{t+1} + W_{hh} h_t + b_h$

we know that $h_{t+1} = \phi_h (Z_{t+1})$

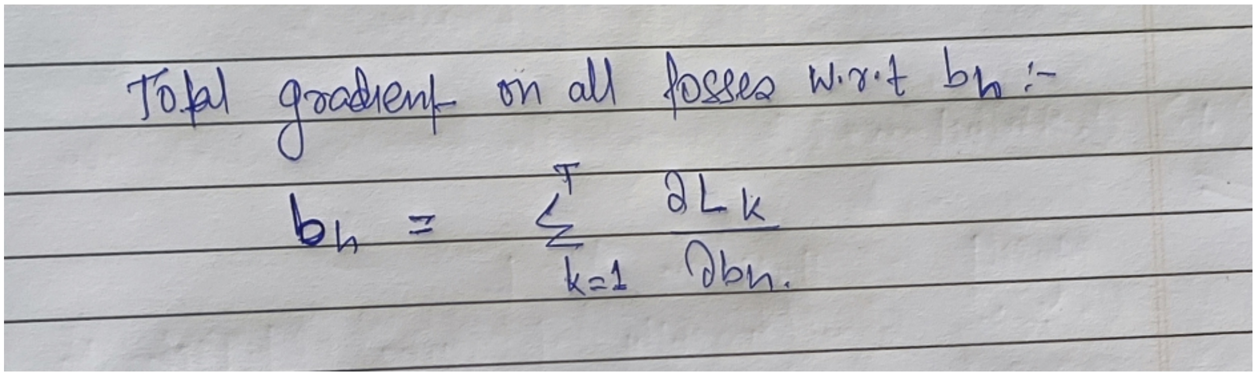
$$\therefore \frac{\partial h_{t+1}}{\partial b_h} = \phi_h' (Z_{t+1}) \cdot \frac{\partial Z_{t+1}}{\partial b_h}$$

$$\frac{\partial Z_{t+1}}{\partial b_h} = W_{hh} \frac{\partial h_t}{\partial b_h} + 1$$

$$\frac{\partial h_{t+1}}{\partial b_h} = \phi_h' (Z_{t+1}) \cdot \left(W_{hh} \frac{\partial h_t}{\partial b_h} + 1\right) \text{ (Recursion)}$$

$$\frac{\partial h_t}{\partial b_h} = \phi_h' (Z_t) \left(W_{hh} \frac{\partial h_{t-1}}{\partial b_h} + 1\right)$$

and so on ....

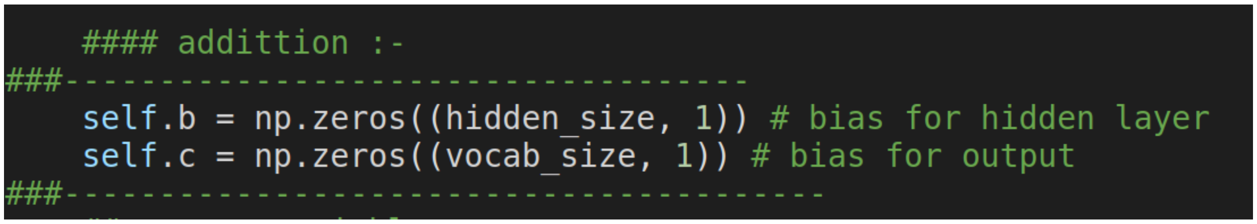


Total gradient on all losses w.r.t b_h :-

$$b_h = \sum_{k=1}^{T} \frac{\partial L_k}{\partial b_h}.$$

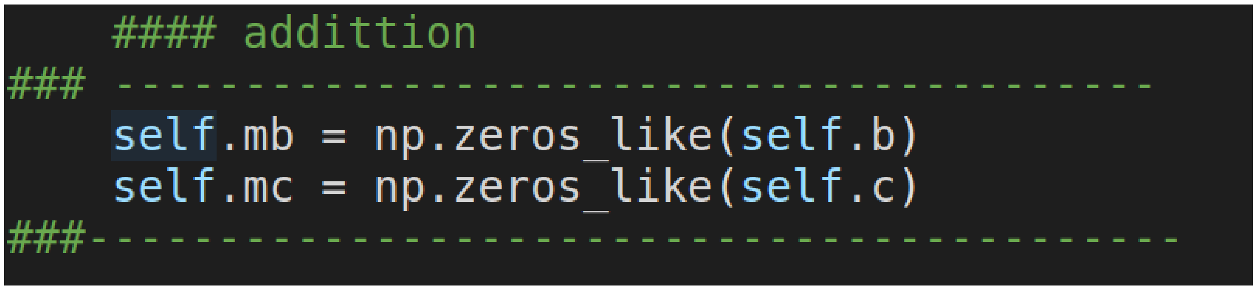
The above equation for gradient of loss wrt b_h is recursive in nature just like the gradient of loss wrt W_{hh} .
And so similar to gradient computation implementation of W_{hh} here also we can recursively calculate whatever we can at time t.

Screenshots of Derived bias update equations in the RNN code shared (RNN_from_Scratch.ipynb) :-



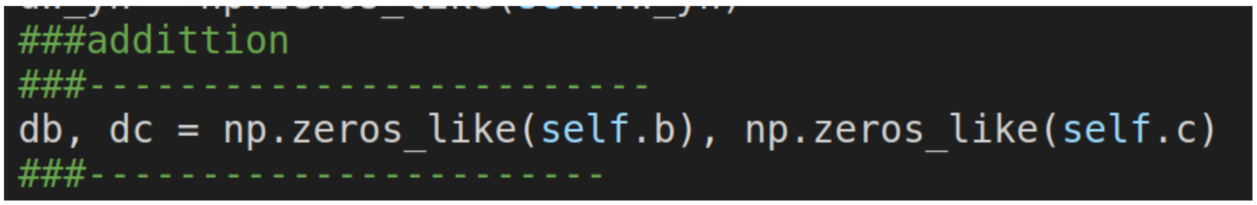
```
    #### addittion :-
###-------------------------------------
    self.b = np.zeros((hidden_size, 1)) # bias for hidden layer
    self.c = np.zeros((vocab_size, 1)) # bias for output
###-----------------------------------------
```

In the **memory variable** part of the **"__init__" method of class RNN** code :-



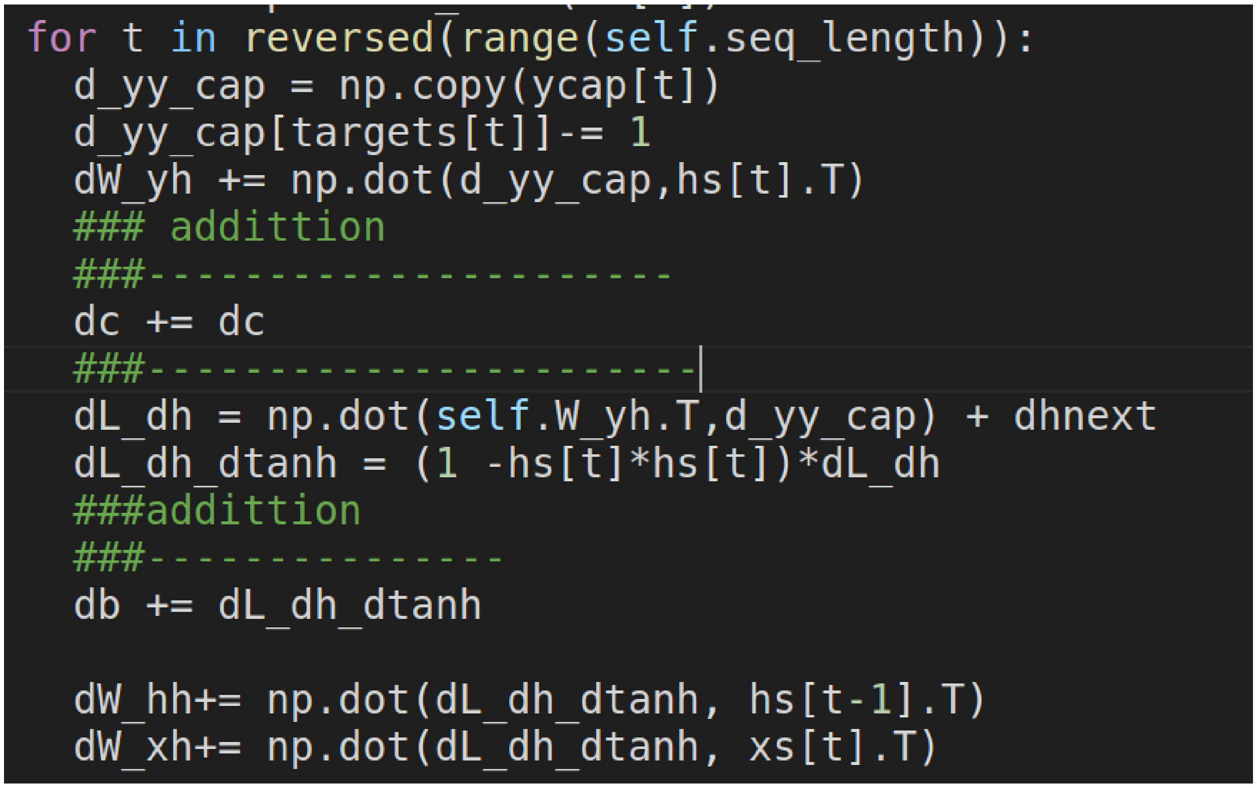
```
    #### addittion
### ----------------------------------------
    self.mb = np.zeros_like(self.b)
    self.mc = np.zeros_like(self.c)
###-----------------------------------------
```

In the **“backward”** method of **class RNN :-**



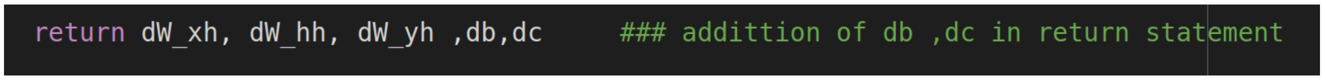
```
###addittion
###------------------------
db, dc = np.zeros_like(self.b), np.zeros_like(self.c)
###----------------------
```

Under the for loop of backward method:-



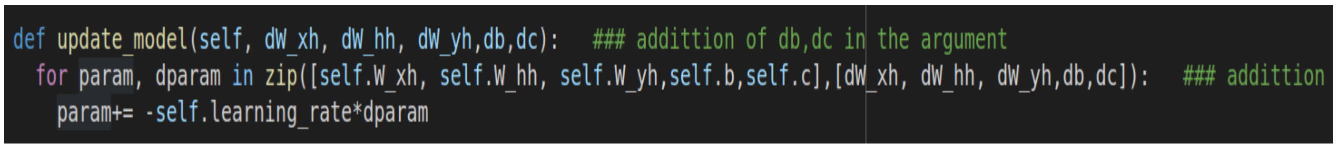
```
for t in reversed(range(self.seq_length)):
  d_yy_cap = np.copy(ycap[t])
  d_yy_cap[targets[t]]-= 1
  dW_yh += np.dot(d_yy_cap,hs[t].T)
  ### addittion
  ###----------------------
  dc += dc
  ###----------------------
  dL_dh = np.dot(self.W_yh.T,d_yy_cap) + dhnext
  dL_dh_dtanh = (1 -hs[t]*hs[t])*dL_dh
  ###addittion
  ###---------------
  db += dL_dh_dtanh

  dW_hh+= np.dot(dL_dh_dtanh, hs[t-1].T)
  dW_xh+= np.dot(dL_dh_dtanh, xs[t].T)
```

Updated the return statement of backward method in class RNN:-



```
  return dW_xh, dW_hh, dW_yh ,db,dc     ### addittion of db ,dc in return statement
```

Edited the **Update_model method of class RNN** :-



```
def update_model(self, dW_xh, dW_hh, dW_yh,db,dc):   ### addittion of db,dc in the argument
  for param, dparam in zip([self.W_xh, self.W_hh, self.W_yh,self.b,self.c],[dW_xh, dW_hh, dW_yh,db,dc]):   ### addittion
    param+= -self.learning_rate*dparam
```

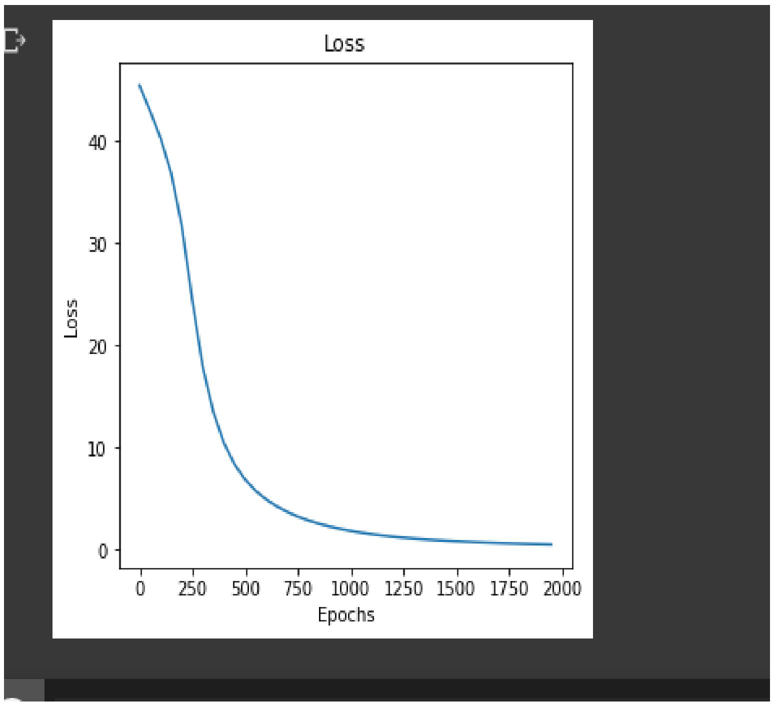
Analysis of training performance for Q.1 With and without bias:-

1. Losses with varying no of epochs :-

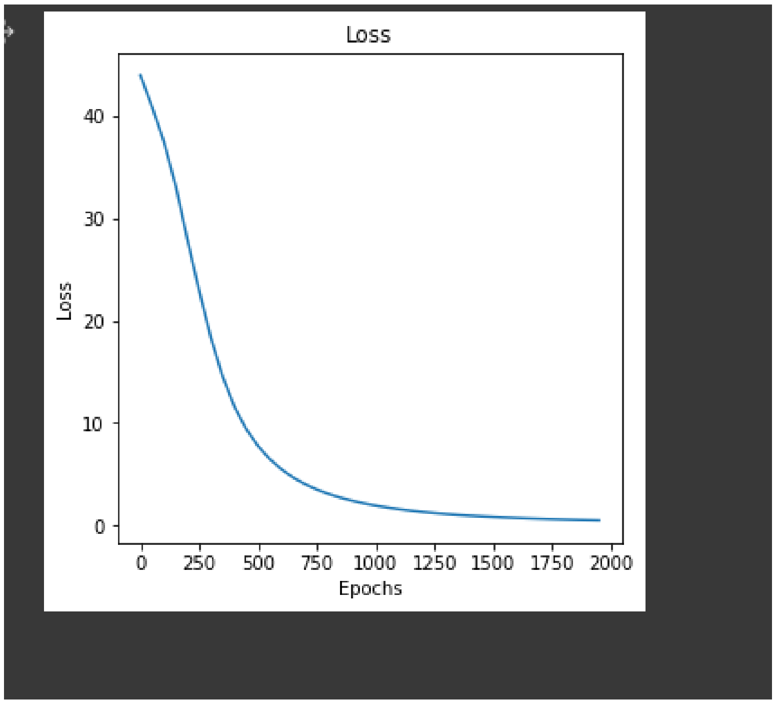
Word used :- “You are a good guy”
Hidden layer size =50
Learning rate :- 0.001

No.of epochs	Loss(with bias term)	Loss(without bias term)
1000	1.805	1.712
2000	0.531	0.540
5000	0.129	0.150
10000	0.015	0.053

Loss vs Epochs graph with bias term :-



Loss vs epochs graph without bias term :-

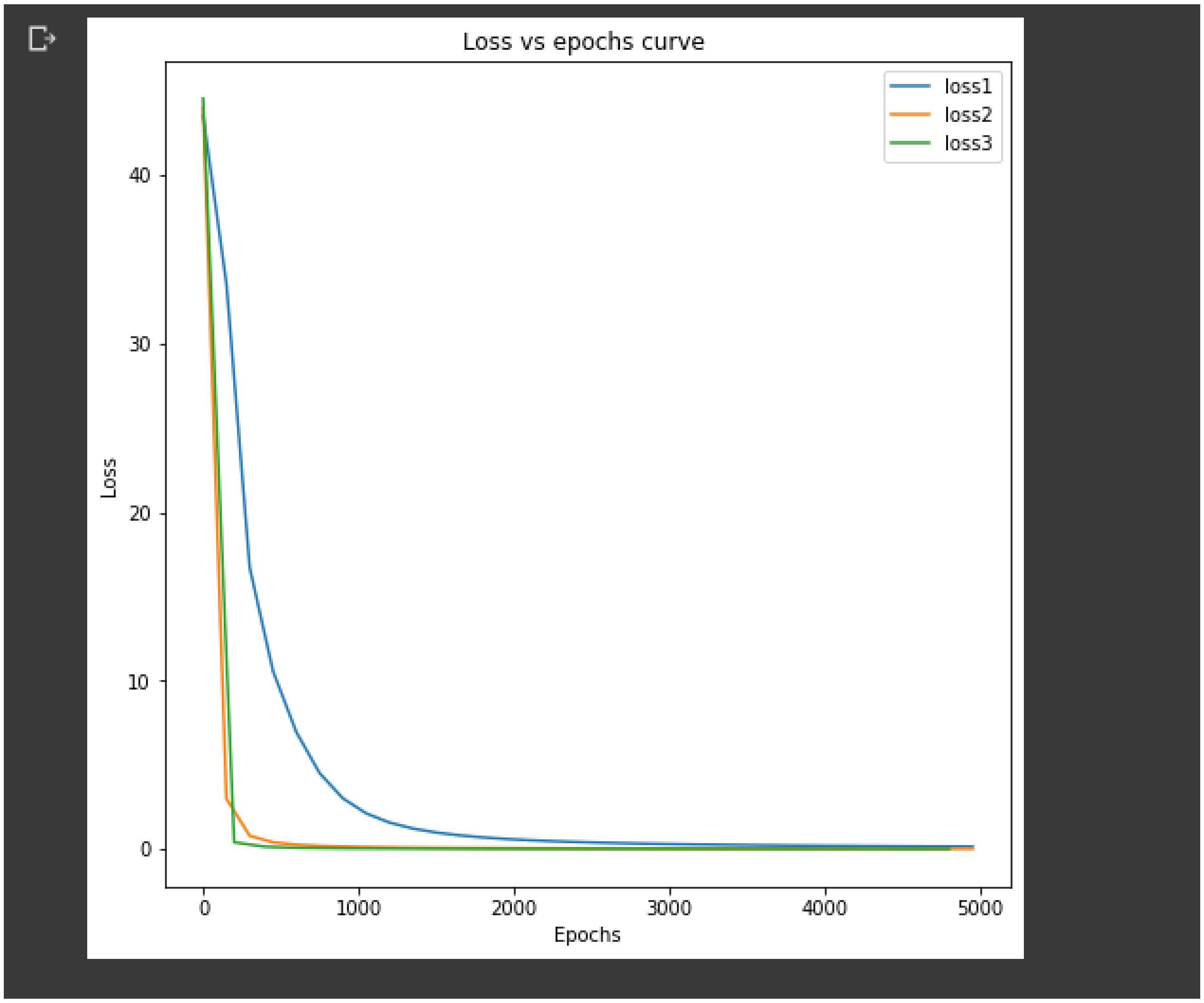


2. Losses with varying learning rate :-

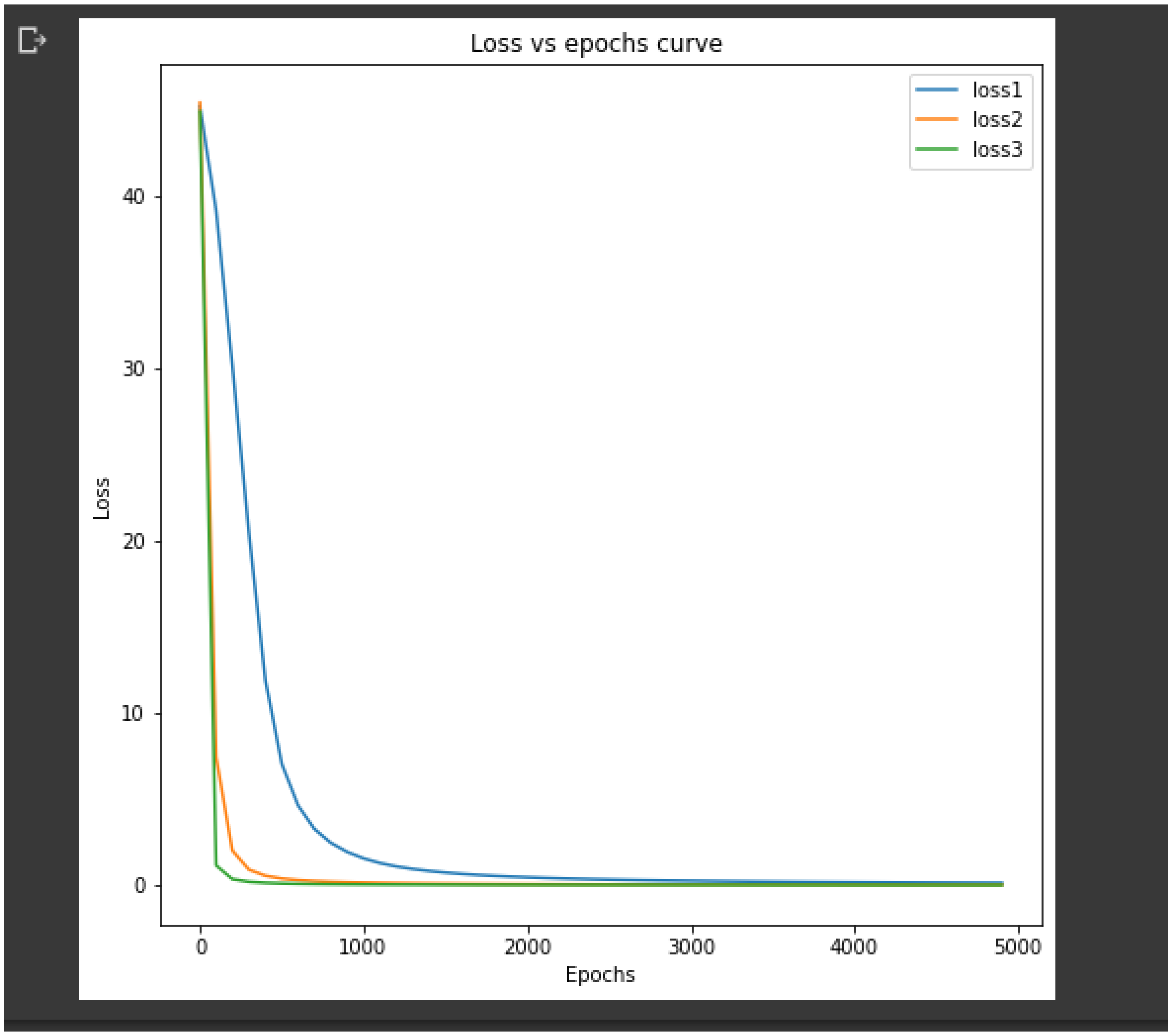
Word used :- “ You are a good guy ”
Hidden layer size =50
No 0f epochs = 5000
Now we are changing Learning rate:

Learning rate	Loss(with bias term)	Loss(without bias term)
0.001	0.1413(loss 1)	0.1454
0.005	0.0212(loss 2)	0.0191
0.012	0.0075(loss 3)	0.0070

Loss vs epoch curve on 3 different learning rates(without bias) as given in above table :-



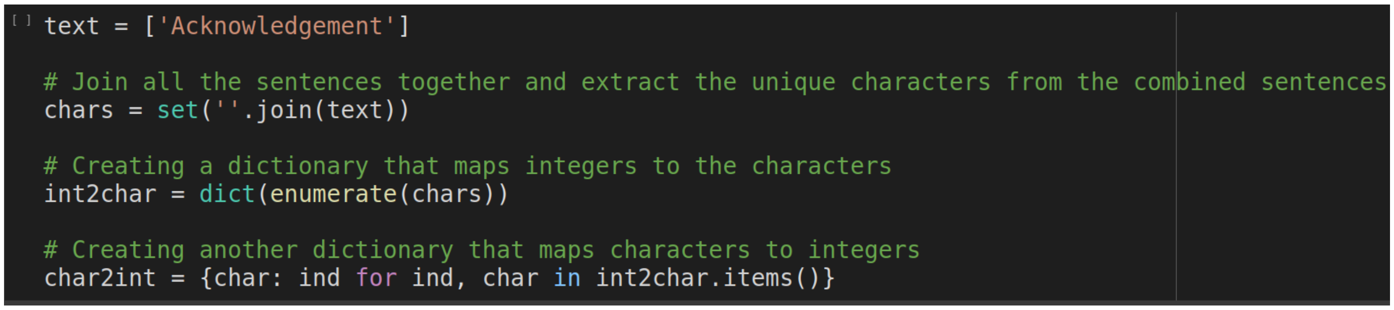
Loss vs epoch curve on 3 different learning rates(with bias) as given in above table :-



2 . i) Train an RNN using pytorch to predict the string **“Acknowledgement” giving the starting character as “A”**

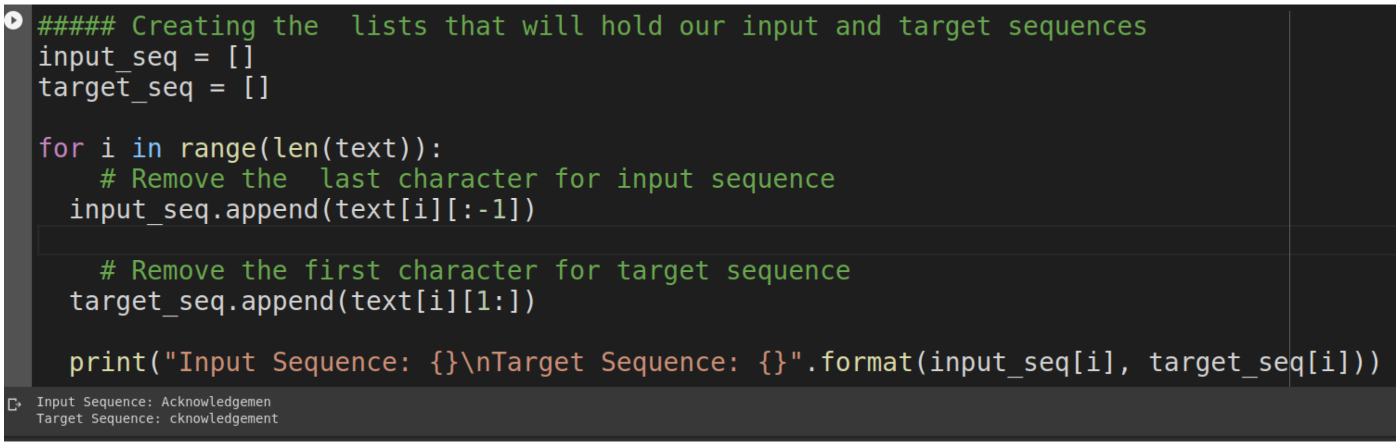
Some screenshots of training and predicting the string **“Acknowledgement” :-**

1. **Creating the dictionary that maps int. To char. and another that maps chars. to int.**



```
text = ['Acknowledgement']

# Join all the sentences together and extract the unique characters from the combined sentences
chars = set(''.join(text))

# Creating a dictionary that maps integers to the characters
int2char = dict(enumerate(chars))

# Creating another dictionary that maps characters to integers
char2int = {char: ind for ind, char in int2char.items()}
```

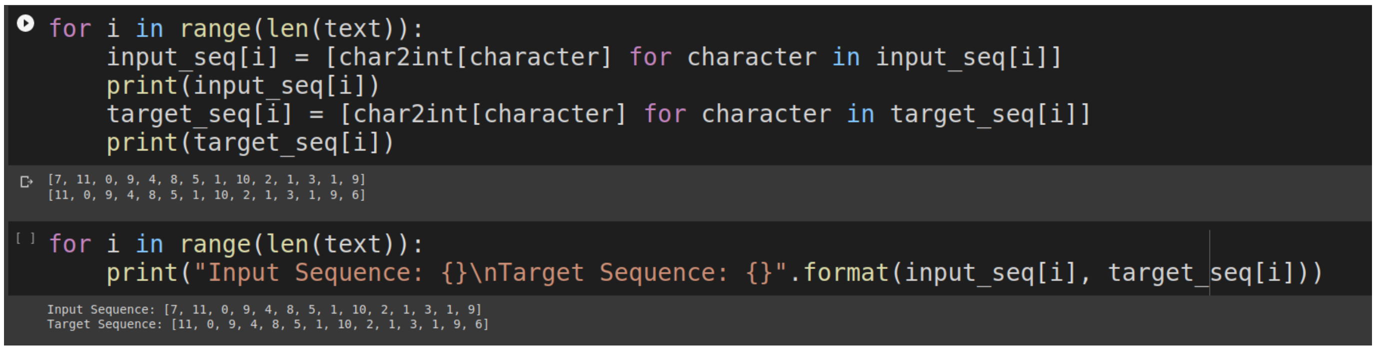
2. **Giving the input sequence and output sequence which we have to feed to our model and then to predict our target sequence .**



```
##### Creating the  lists that will hold our input and target sequences
input_seq = []
target_seq = []

for i in range(len(text)):
    # Remove the  last character for input sequence
  input_seq.append(text[i][:-1])

    # Remove the first character for target sequence
  target_seq.append(text[i][1:])

  print("Input Sequence: {}\nTarget Sequence: {}".format(input_seq[i], target_seq[i]))
```

```
Input Sequence: Acknowledgemen
Target Sequence: cknowledgement
```

3. **Then creating integer mapping of input sequence and output sequence**



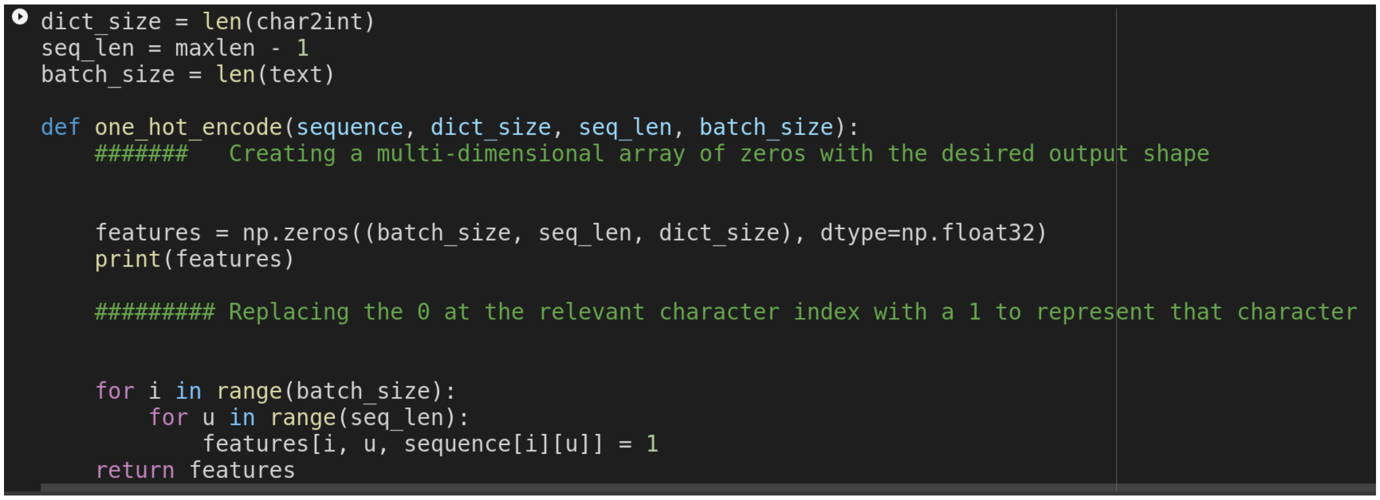
```
for i in range(len(text)):
    input_seq[i] = [char2int[character] for character in input_seq[i]]
    print(input_seq[i])
    target_seq[i] = [char2int[character] for character in target_seq[i]]
    print(target_seq[i])
```

```
[7, 11, 0, 9, 4, 8, 5, 1, 10, 2, 1, 3, 1, 9]
[11, 0, 9, 4, 8, 5, 1, 10, 2, 1, 3, 1, 9, 6]
```

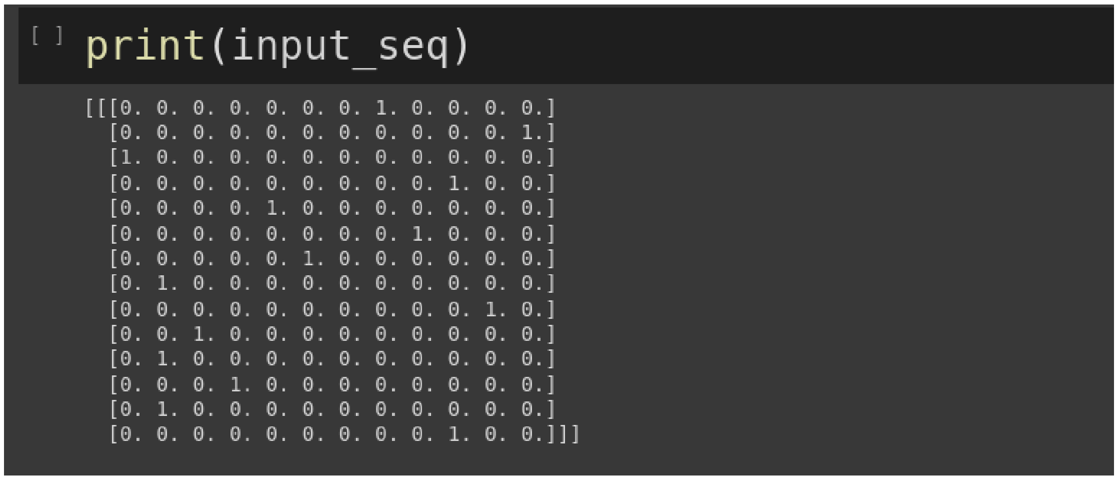
```
for i in range(len(text)):
    print("Input Sequence: {}\nTarget Sequence: {}".format(input_seq[i], target_seq[i]))
```

```
Input Sequence: [7, 11, 0, 9, 4, 8, 5, 1, 10, 2, 1, 3, 1, 9]
Target Sequence: [11, 0, 9, 4, 8, 5, 1, 10, 2, 1, 3, 1, 9, 6]
```

4. To do one hot encoding of input_sequence with their output :-



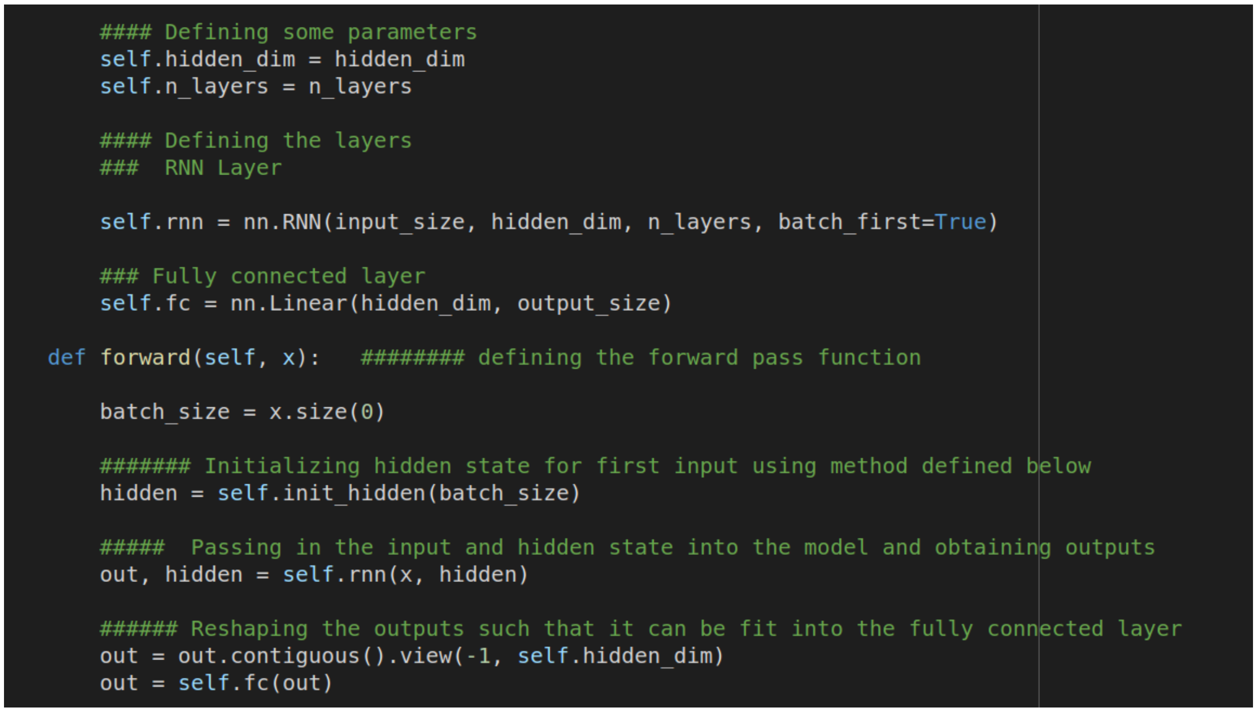
```
dict_size = len(char2int)
seq_len = maxlen - 1
batch_size = len(text)

def one_hot_encode(sequence, dict_size, seq_len, batch_size):
    #######   Creating a multi-dimensional array of zeros with the desired output shape


    features = np.zeros((batch_size, seq_len, dict_size), dtype=np.float32)
    print(features)

    ######### Replacing the 0 at the relevant character index with a 1 to represent that character


    for i in range(batch_size):
        for u in range(seq_len):
            features[i, u, sequence[i][u]] = 1
    return features
```



```
[ ] print(input_seq)
```

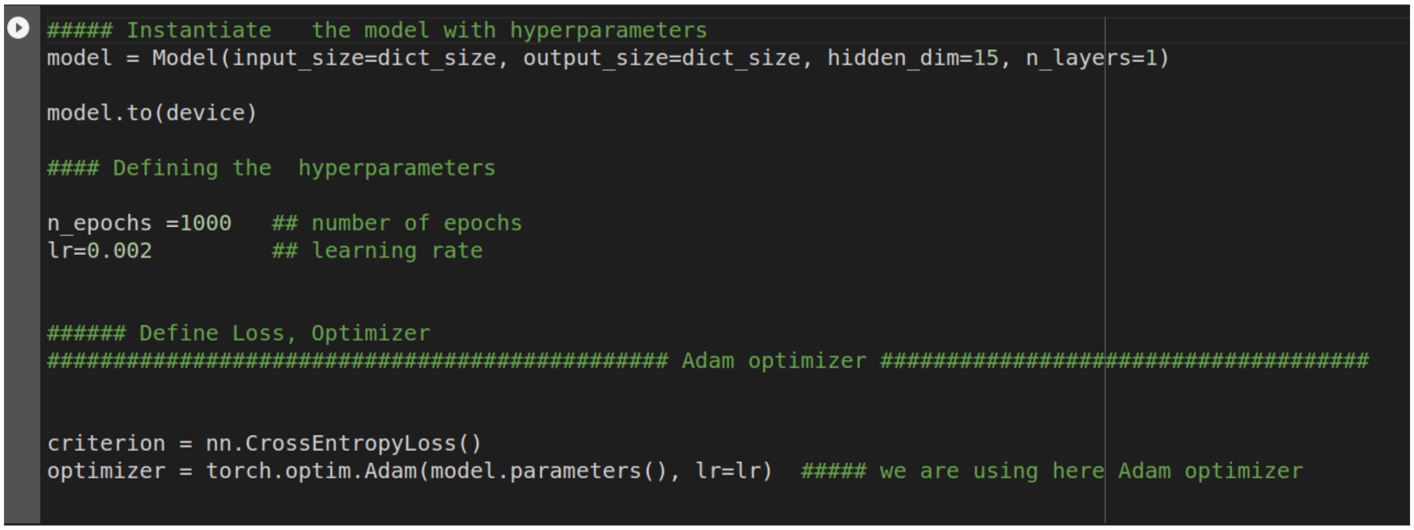
```
[[[0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
  [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
  [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
  [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
  [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
  [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]]
```

5. **Defining the parameters , RNN layer and forward pass function then initialization of hidden state for first input :-**



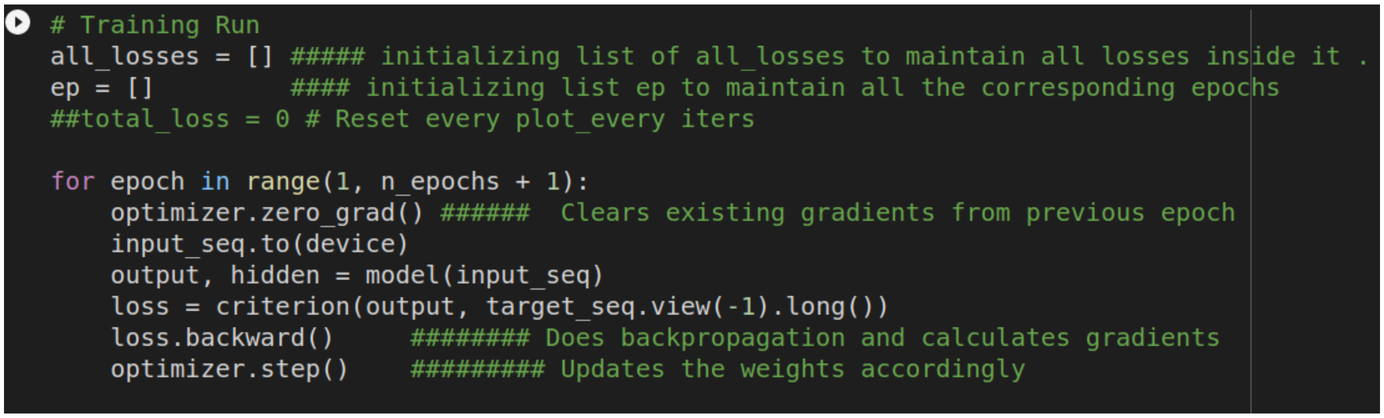
```
        #### Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        #### Defining the layers
        ###  RNN Layer

        self.rnn = nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)

        ### Fully connected layer
        self.fc = nn.Linear(hidden_dim, output_size)

    def forward(self, x):   ######## defining the forward pass function

        batch_size = x.size(0)

        ####### Initializing hidden state for first input using method defined below
        hidden = self.init_hidden(batch_size)

        #####  Passing in the input and hidden state into the model and obtaining outputs
        out, hidden = self.rnn(x, hidden)

        ###### Reshaping the outputs such that it can be fit into the fully connected layer
        out = out.contiguous().view(-1, self.hidden_dim)
        out = self.fc(out)
```

6. **Now instantiating the model with defined hyperparameters (epochs,learning rate) and then using the Adam optimizer first:-**



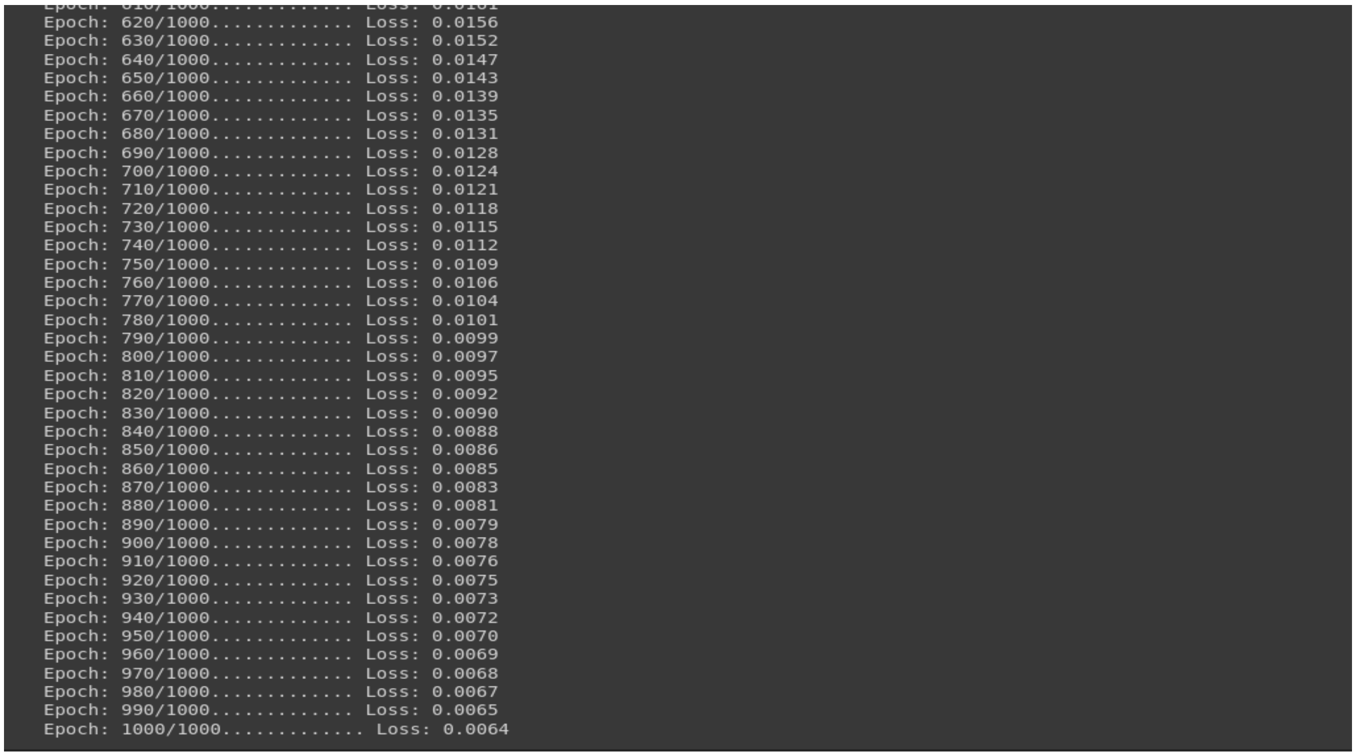
```
##### Instantiate   the model with hyperparameters
model = Model(input_size=dict_size, output_size=dict_size, hidden_dim=15, n_layers=1)

model.to(device)

#### Defining the  hyperparameters

n_epochs =1000   ## number of epochs
lr=0.002         ## learning rate


###### Define Loss, Optimizer
############################################### Adam optimizer #####################################


criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)  ##### we are using here Adam optimizer
```

7. **Now training the model over 1000 epochs first and tracking the losses to plot the graph between losses vs epoch .**



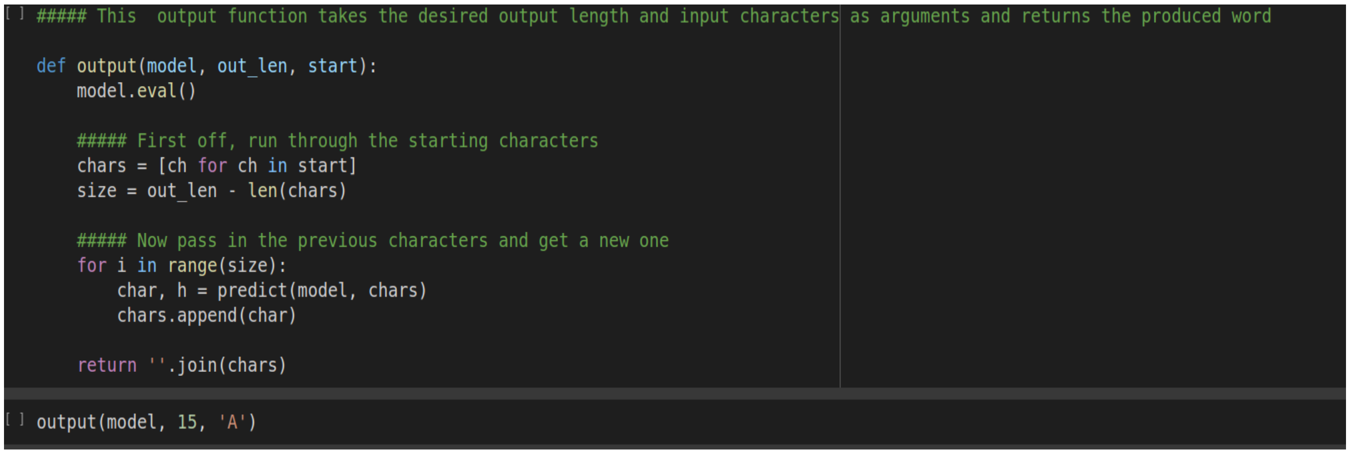
```
# Training Run
all_losses = [] ##### initializing list of all_losses to maintain all losses inside it .
ep = []         #### initializing list ep to maintain all the corresponding epochs
##total_loss = 0 # Reset every plot_every iters

for epoch in range(1, n_epochs + 1):
    optimizer.zero_grad() ######  Clears existing gradients from previous epoch
    input_seq.to(device)
    output, hidden = model(input_seq)
    loss = criterion(output, target_seq.view(-1).long())
    loss.backward()     ######## Does backpropagation and calculates gradients
    optimizer.step()    ######### Updates the weights accordingly
```

Output of training which gives losses per epoch:-



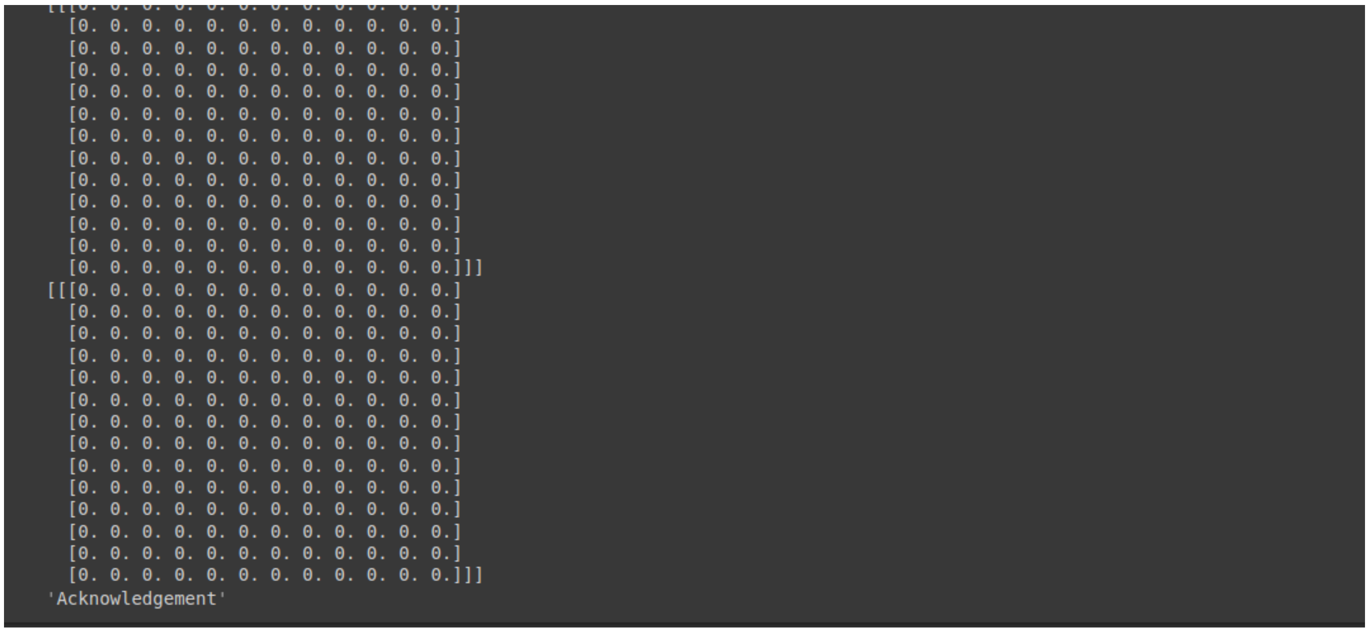
```
Epoch: 620/1000............ Loss: 0.0156
Epoch: 630/1000............ Loss: 0.0152
Epoch: 640/1000............ Loss: 0.0147
Epoch: 650/1000............ Loss: 0.0143
Epoch: 660/1000............ Loss: 0.0139
Epoch: 670/1000............ Loss: 0.0135
Epoch: 680/1000............ Loss: 0.0131
Epoch: 690/1000............ Loss: 0.0128
Epoch: 700/1000............ Loss: 0.0124
Epoch: 710/1000............ Loss: 0.0121
Epoch: 720/1000............ Loss: 0.0118
Epoch: 730/1000............ Loss: 0.0115
Epoch: 740/1000............ Loss: 0.0112
Epoch: 750/1000............ Loss: 0.0109
Epoch: 760/1000............ Loss: 0.0106
Epoch: 770/1000............ Loss: 0.0104
Epoch: 780/1000............ Loss: 0.0101
Epoch: 790/1000............ Loss: 0.0099
Epoch: 800/1000............ Loss: 0.0097
Epoch: 810/1000............ Loss: 0.0095
Epoch: 820/1000............ Loss: 0.0092
Epoch: 830/1000............ Loss: 0.0090
Epoch: 840/1000............ Loss: 0.0088
Epoch: 850/1000............ Loss: 0.0086
Epoch: 860/1000............ Loss: 0.0085
Epoch: 870/1000............ Loss: 0.0083
Epoch: 880/1000............ Loss: 0.0081
Epoch: 890/1000............ Loss: 0.0079
Epoch: 900/1000............ Loss: 0.0078
Epoch: 910/1000............ Loss: 0.0076
Epoch: 920/1000............ Loss: 0.0075
Epoch: 930/1000............ Loss: 0.0073
Epoch: 940/1000............ Loss: 0.0072
Epoch: 950/1000............ Loss: 0.0070
Epoch: 960/1000............ Loss: 0.0069
Epoch: 970/1000............ Loss: 0.0068
Epoch: 980/1000............ Loss: 0.0067
Epoch: 990/1000............ Loss: 0.0065
Epoch: 1000/1000............ Loss: 0.0064
```

8. Defining the output function which returns the produced word:-



```
##### This  output function takes the desired output length and input characters as arguments and returns the produced word

def output(model, out_len, start):
    model.eval()

    ##### First off, run through the starting characters
    chars = [ch for ch in start]
    size = out_len - len(chars)

    ##### Now pass in the previous characters and get a new one
    for i in range(size):
        char, h = predict(model, chars)
        chars.append(char)

    return ''.join(chars)
```

```
output(model, 15, 'A')
```



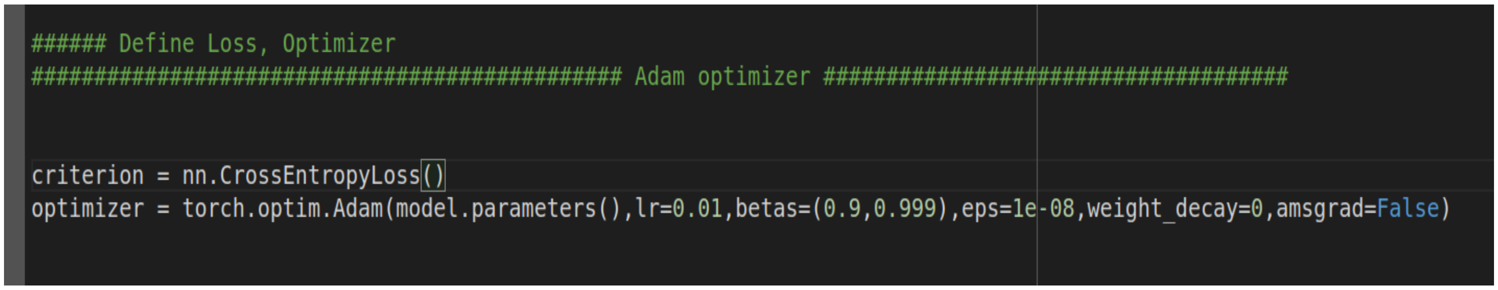
```
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]]
[[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]]
'Acknowledgement'
```

Training level performance analysis of Q.2(i):-

Number of epochs used: 1000
Word used : “Acknowledgement”

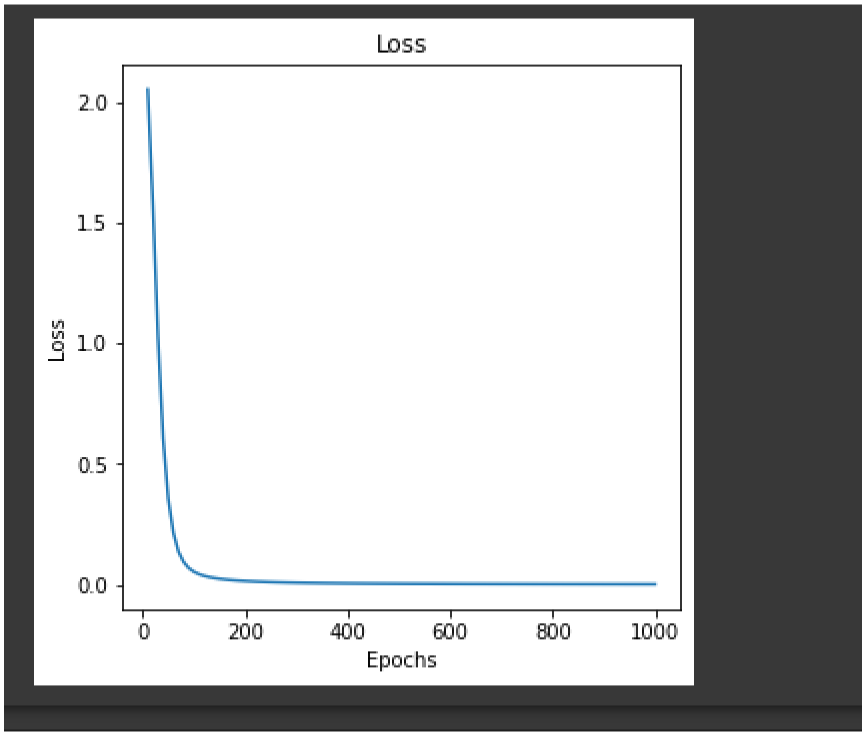
- ❑ **Gradient update Technique: Adam**
  - ➢ **Learning rate = 0.1**
  - ➢ **Eps = 1e-8**
  - ➢ **Beta1 = 0.9**
  - ➢ **Beta2 = 0.99**
- **betas: It is used as a parameter that calculates the averages of the gradient.**
- **eps: It is used for improving numerical stability.**

Hidden layer size	Losses
5	0.0059
20	0.0003
50	0.00001



```
###### Define Loss, Optimizer
############################################## Adam optimizer ####################################


criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(),lr=0.01,betas=(0.9,0.999),eps=1e-08,weight_decay=0,amsgrad=False)
```

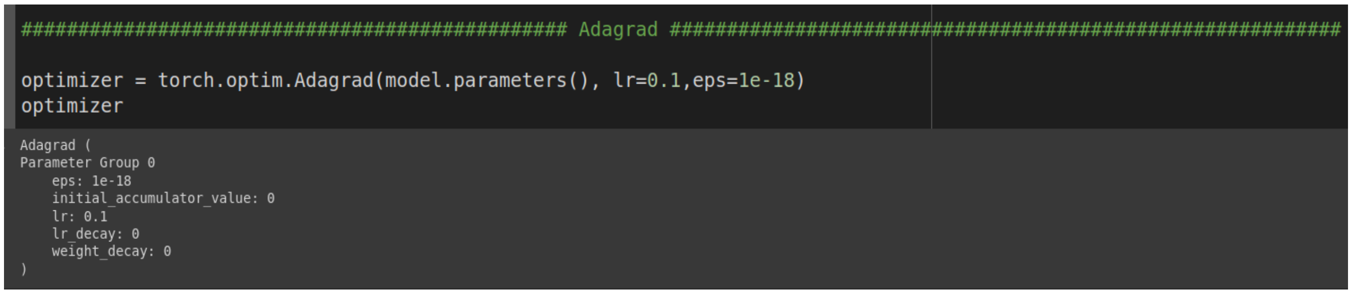
Epoch vs loss graph for hidden layer size of 20 we are plotting per 10 epochs :-



Number of epochs used: 1000
Word used : “Acknowledgement”

- **Gradient update Technique: AdaGrad**
  - **Learning rate, eta = 0.1**
  - **Eps = 1e-18**

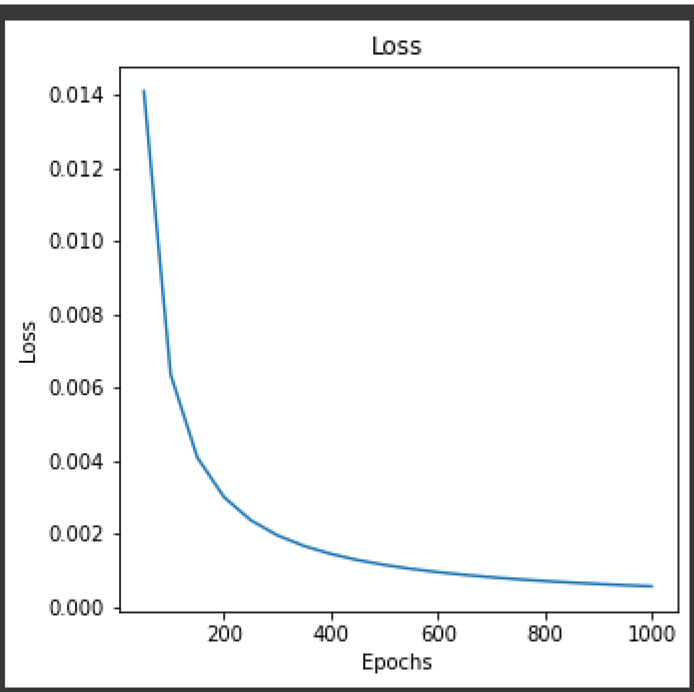
Hidden layer size	**Losses**
10	**0.0016**
20	**0.0007**
50	**0.0005**



```
############################################## Adagrad ################################################################

optimizer = torch.optim.Adagrad(model.parameters(), lr=0.1,eps=1e-18)
optimizer
```

```
Adagrad (
Parameter Group 0
    eps: 1e-18
    initial_accumulator_value: 0
    lr: 0.1
    lr_decay: 0
    weight_decay: 0
)
```

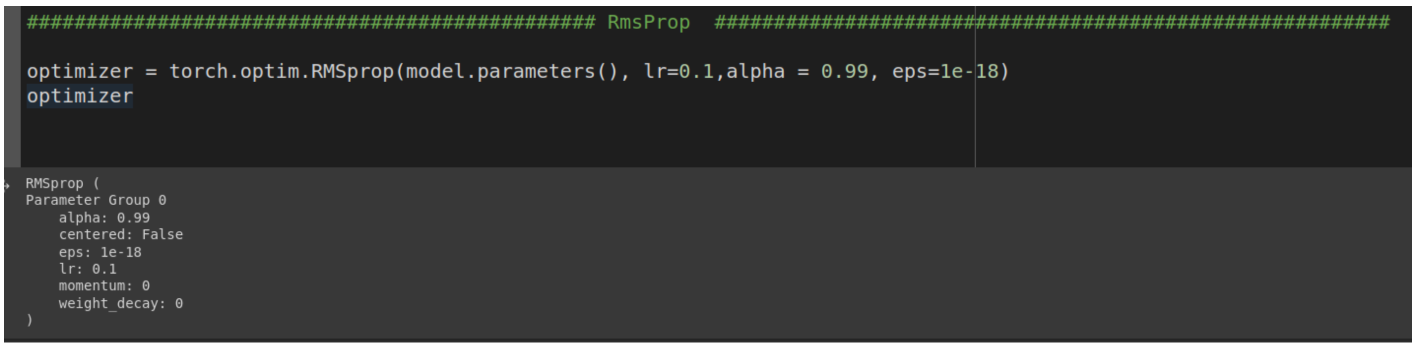
We are plotting graph after per 50 epochs :-



Number of epochs used: 1000
Word used : “Acknowledgement”

- ❑ **Gradient update Technique : RmsProp**
  - ➢ **Learning rate = 0.01**
  - ➢ **Alpha(smoothing constant) = 0.99**
  - ➢ **Eps = 1e-18**

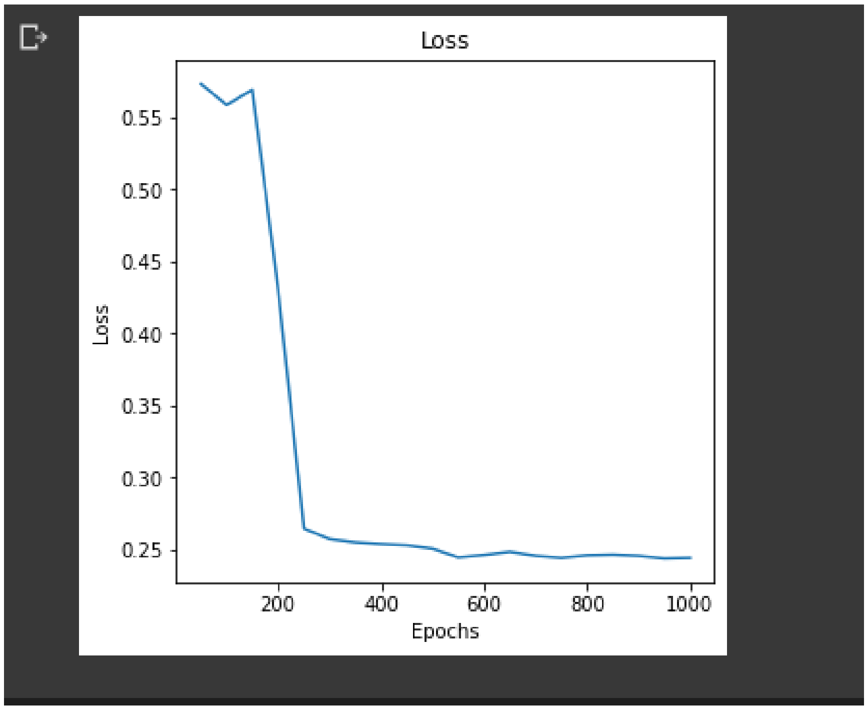
Hidden layer size	**Losses**
10	**0.0001**
20	**0.00001**
50	**0.00000**



```
################################################ RmsProp  ##########################################################

optimizer = torch.optim.RMSprop(model.parameters(), lr=0.1,alpha = 0.99, eps=1e-18)
optimizer
```

```
RMSprop (
Parameter Group 0
    alpha: 0.99
    centered: False
    eps: 1e-18
    lr: 0.1
    momentum: 0
    weight_decay: 0
)
```

Loss vs epoch graph on every 50 epochs we are plotting :-



Conclusions:

- **With increase in hidden layer size, we can see that the loss generally decreases as we are plotting the tables for each optimizer .**

- **It can also be observed that loss with bias terms is generally lesser than the loss without bias terms in Q1 . Few exceptions are there and the reason for it might be overfitting so I am getting inconsistent graphs.**

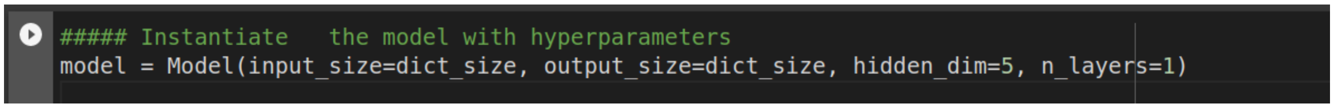
- **It was observed that despite the loss decreasing in each epoch, the target string might become incorrect. However, in the end the results are almost consistent, that is we get the correct target.**

- **RmsProp gradient update technique is the fastest among all the above discussed gradient update techniques.**

Training level performance analysis of Q.2(ii):

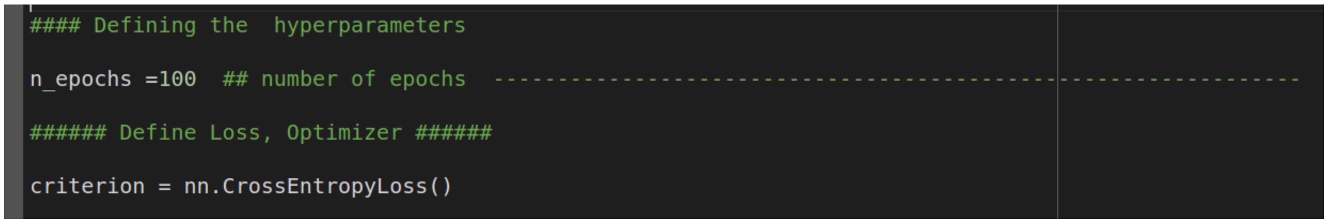
We have done following architectural changes in my code :-

1. **We have given hidden layers size of 5 only as sir said t restrict the hidden layer size of 5 :-**



```
##### Instantiate   the model with hyperparameters
model = Model(input_size=dict_size, output_size=dict_size, hidden_dim=5, n_layers=1)
```

2. **We have given the following hyperparameters :-**



```
#### Defining the  hyperparameters

n_epochs =100  ## number of epochs  ---------------------------------------------------------------

###### Define Loss, Optimizer ######

criterion = nn.CrossEntropyLoss()
```

Since we have to do fast convergence here so, i have taken here only of 100 epochs and figuring out the losses ,

As I have trained my model through various optimizer like Adam , Adagrad , SGD , RMSprop and have adjusted various parameters in optimizer for fast convergence

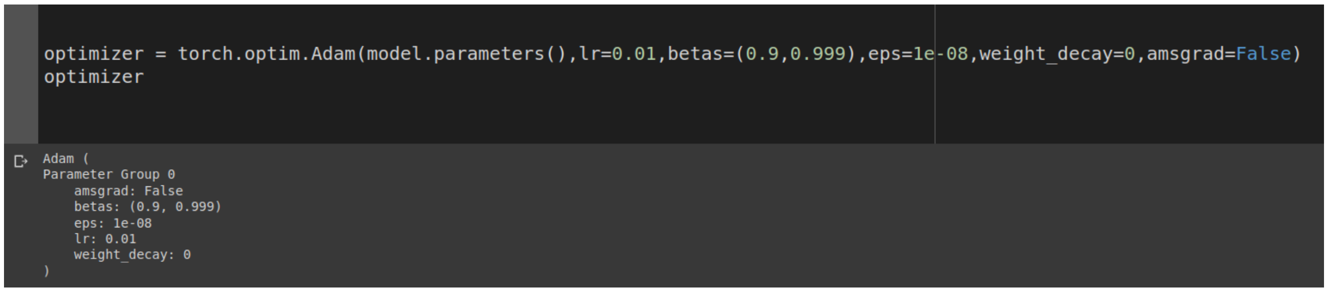
So finally i have found RMSprop gradient optimization technique with the adjusted parameters which is showing the fast convergence that is on lesser than 60 epochs loss is approaching towards 0

So here i am attaching the optimizer implementation in our code :-

So first let's discuss about some of the optimizers that i have implemented by adjusting their different different parameters :-

- ❖ **Adam optimizer :**

  - ➢ **It is one of the most widely used optimizers for training the neural network and is also used for practical purposes.**
  - ➢ **The following syntax is of adam optimizer which is used to reduce the rate of error**



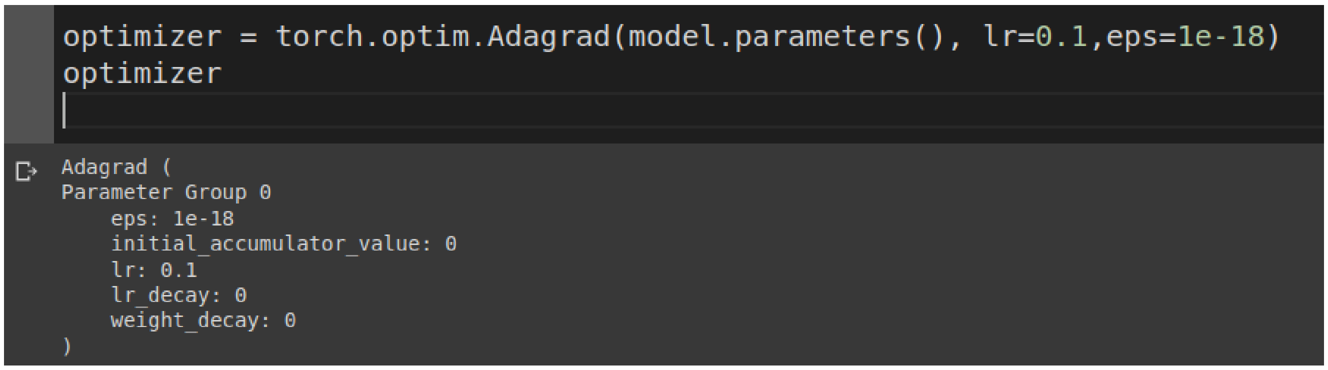
```
optimizer = torch.optim.Adam(model.parameters(),lr=0.01,betas=(0.9,0.999),eps=1e-08,weight_decay=0,amsgrad=False)
optimizer
```

```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    eps: 1e-08
    lr: 0.01
    weight_decay: 0
)
```

- **params:** It is used as a parameter that helps in the optimization.
- **lr :** It is defined as a learning rate helping optimizer.
- **betas:** It is used as a parameter that calculates the averages of the gradient.
- **eps:** It is used for improving numerical stability**.**
- **weight_delay:** It is used for adding the l2 penality to the loss and the default value of weight delay is 0.

❖ Adagrad optimizer :

Adaptive Gradient Algorithm (Adagrad) is an algorithm for gradient-based optimization where each parameter has its own learning rate that improves performance on problems with sparse gradients.



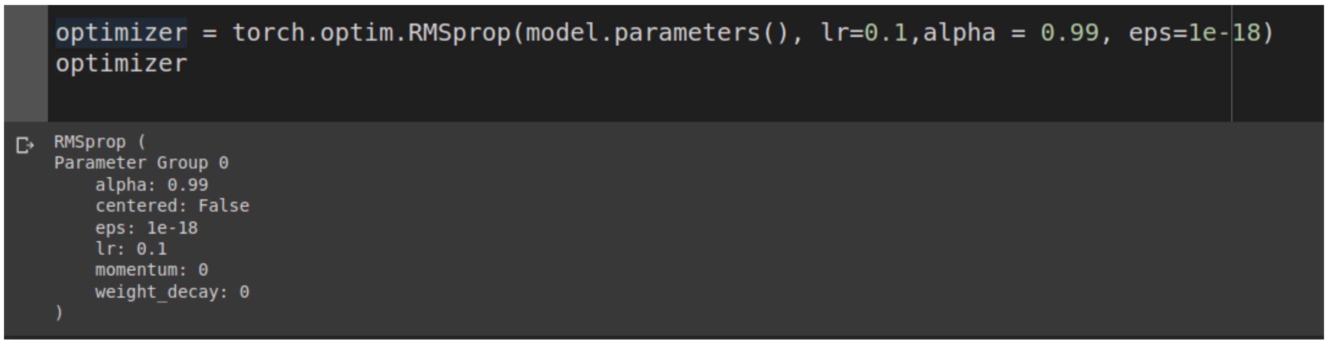
```
optimizer = torch.optim.Adagrad(model.parameters(), lr=0.1,eps=1e-18)
optimizer
```

```
Adagrad (
Parameter Group 0
    eps: 1e-18
    initial_accumulator_value: 0
    lr: 0.1
    lr_decay: 0
    weight_decay: 0
)
```

- **lr (float) :** Learning rate helping optimization (default: 1e-3)
- **lr_decay (float, optional) :** learning rate decay (default: 0)
- **eps (float) :** For improving the numerical stability (default: 1e-8)
- **weight_decay (float) :** For adding the weight decay (L2 penalty) (default: 0)

❖ **RMSprop(root mean square) optimizer :**

RMSProp applies stochastic gradient with mini-batch and uses adaptive learning rates which means it adjusts the learning rates over time.



```
optimizer = torch.optim.RMSprop(model.parameters(), lr=0.1,alpha = 0.99, eps=1e-18)
optimizer
```

```
RMSprop (
Parameter Group 0
    alpha: 0.99
    centered: False
    eps: 1e-18
    lr: 0.1
    momentum: 0
    weight_decay: 0
)
```

- **lr :** Learning rate helping optimization (default: 1e-3)
- **betas (Tuple[float, float], optional) :** It helps in calculation of averages for the gradient squares (default: (0.9, 0.999))
- **eps (float) :** For improving the numerical stability (default: 1e-8)
- **momentum (float, optional) :** momentum factor (default: 0)
- **alpha (float, optional) :** smoothing constant (default: 0.99)

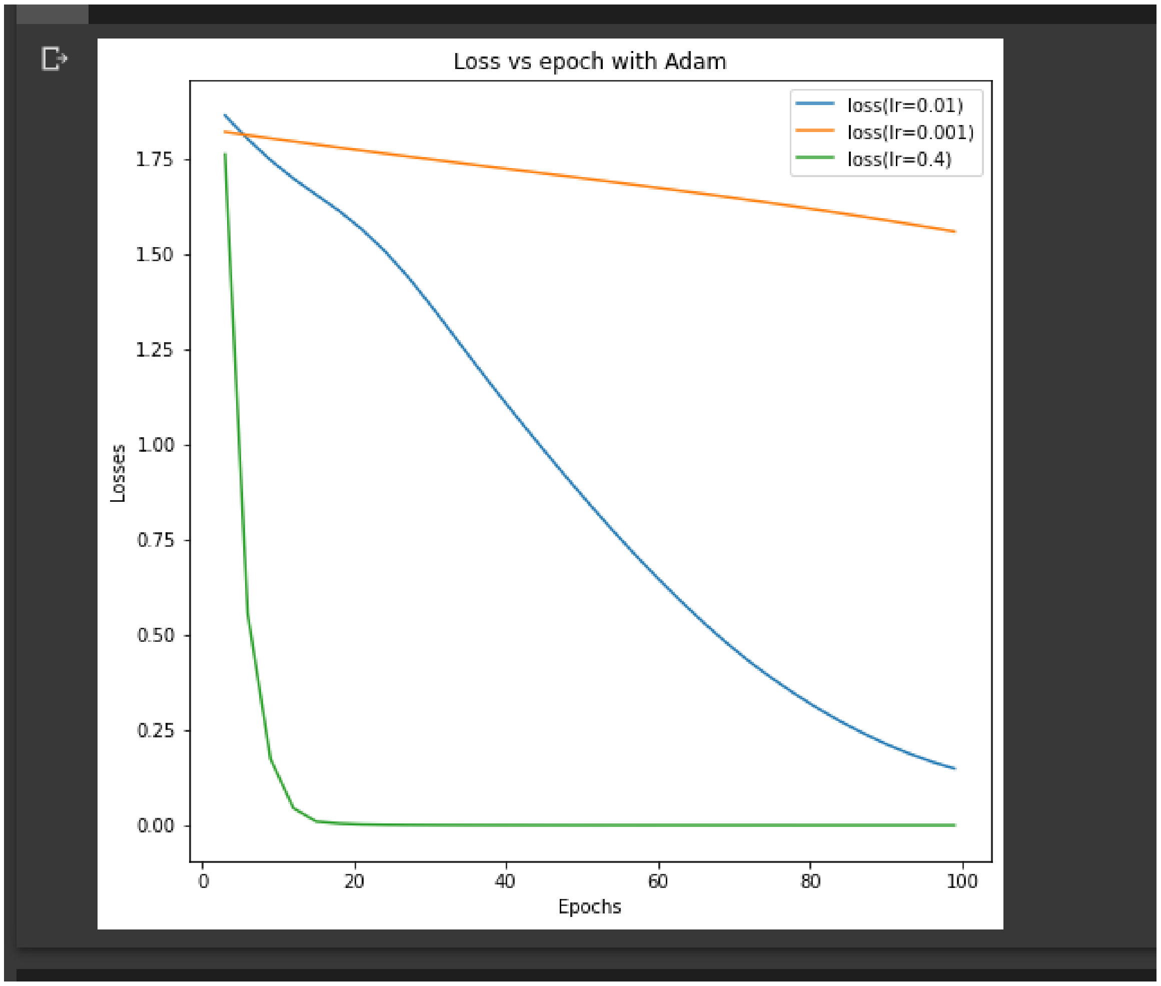
- **centered (bool, optional) :** if True, compute the centered RMSProp, the gradient is normalized by an estimation of its variance
- **weight_decay (float) :** For adding the weight decay (L2 penalty) (default: 0)

Number of epochs used: 100
Word used : "edgement"

- ❑ **Gradient update Technique: Adam**
  - ➢ **Hidden layer size = 5(fixed)**
  - ➢ **Eps = 1e-8**
  - ➢ **Beta1 = 0.9**
  - ➢ **Beta2 = 0.99**
- **betas: It is used as a parameter that calculates the averages of the gradient.**
- **eps: It is used for improving numerical stability.**

Learning rate	**Losses**
0.01	**0.1589**
0.001	**1.5578**
0.4	**0.0002**

Losses vs epochs plotting on Adam optimizer by varying learning rates :

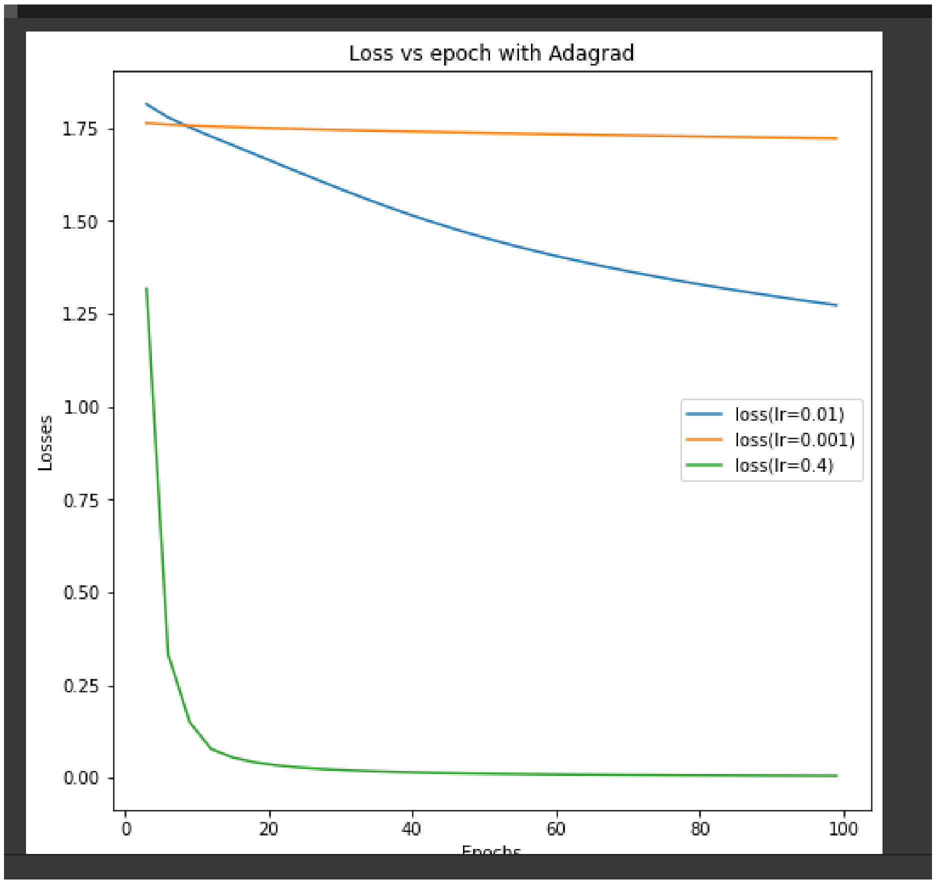


Number of epochs used: 100
Word used : “edgement”

- ❑ **Gradient update Technique: AdaGrad**
  - ➢ **hidden layer size = 5 (fixed)**
  - ➢ **Eps = 1e-18**

Learning rate	**Losses**
0.01	**1.2733**
0.001	**1.7226**
0.4	**0.0052**

Loss vs epoch plotting on Adagrad optimizer with varying Learning rates:

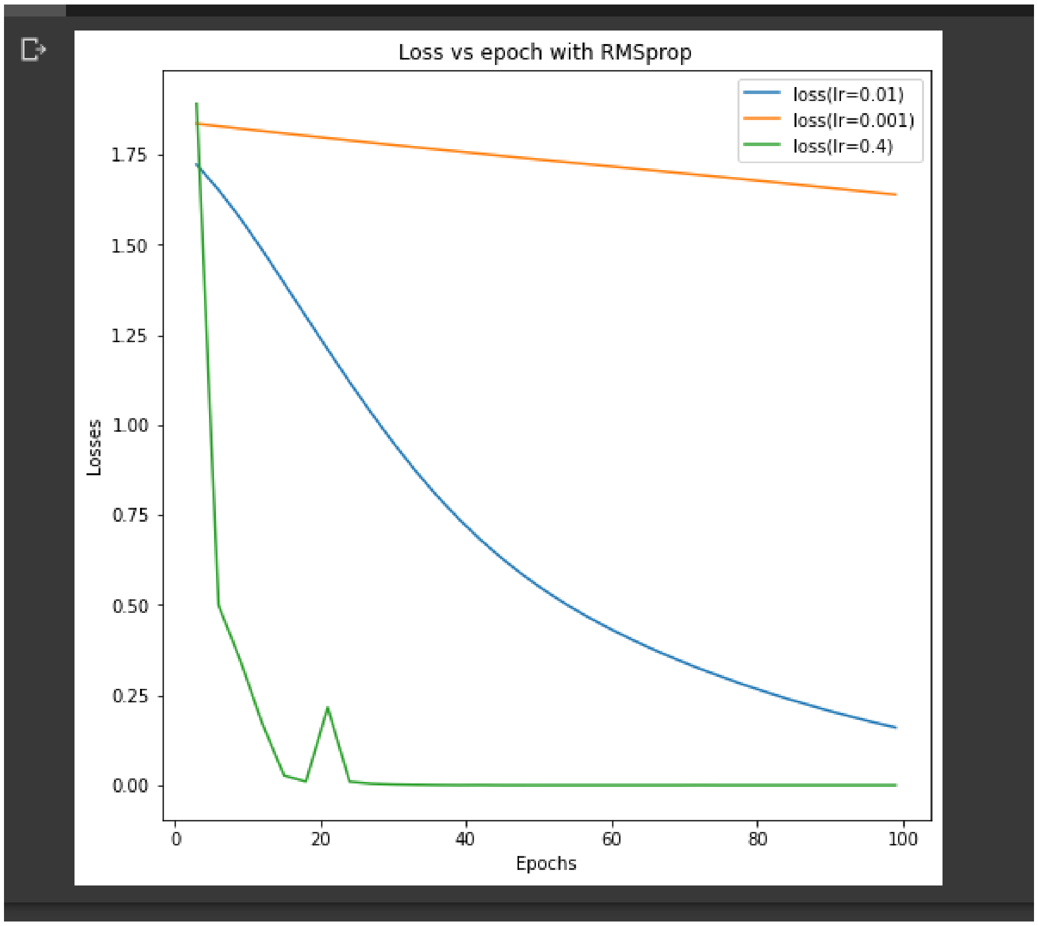


Number of epochs used: 100
Word used : “edgement”

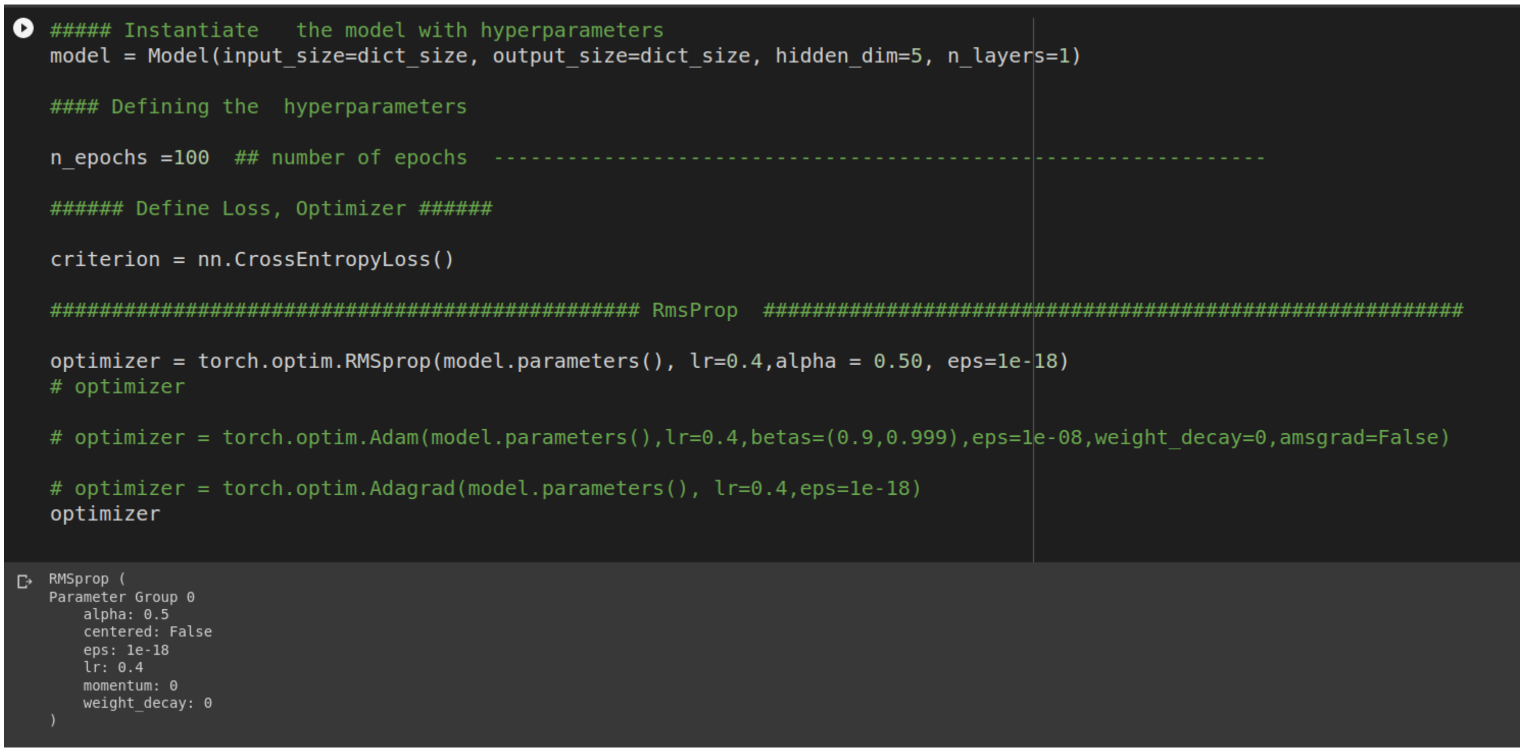
- **Gradient update Technique : RmsProp**
  - **Learning rate = 0.01**
  - **Alpha(smoothing constant) = 0.4**
  - **Eps = 1e-18**

Learning rate	**Losses**
0.01	**0.1604**
0.001	**1.634**
0.4	**0.000**

Loss vs epoch plotting on RMSprop by varying learning rates



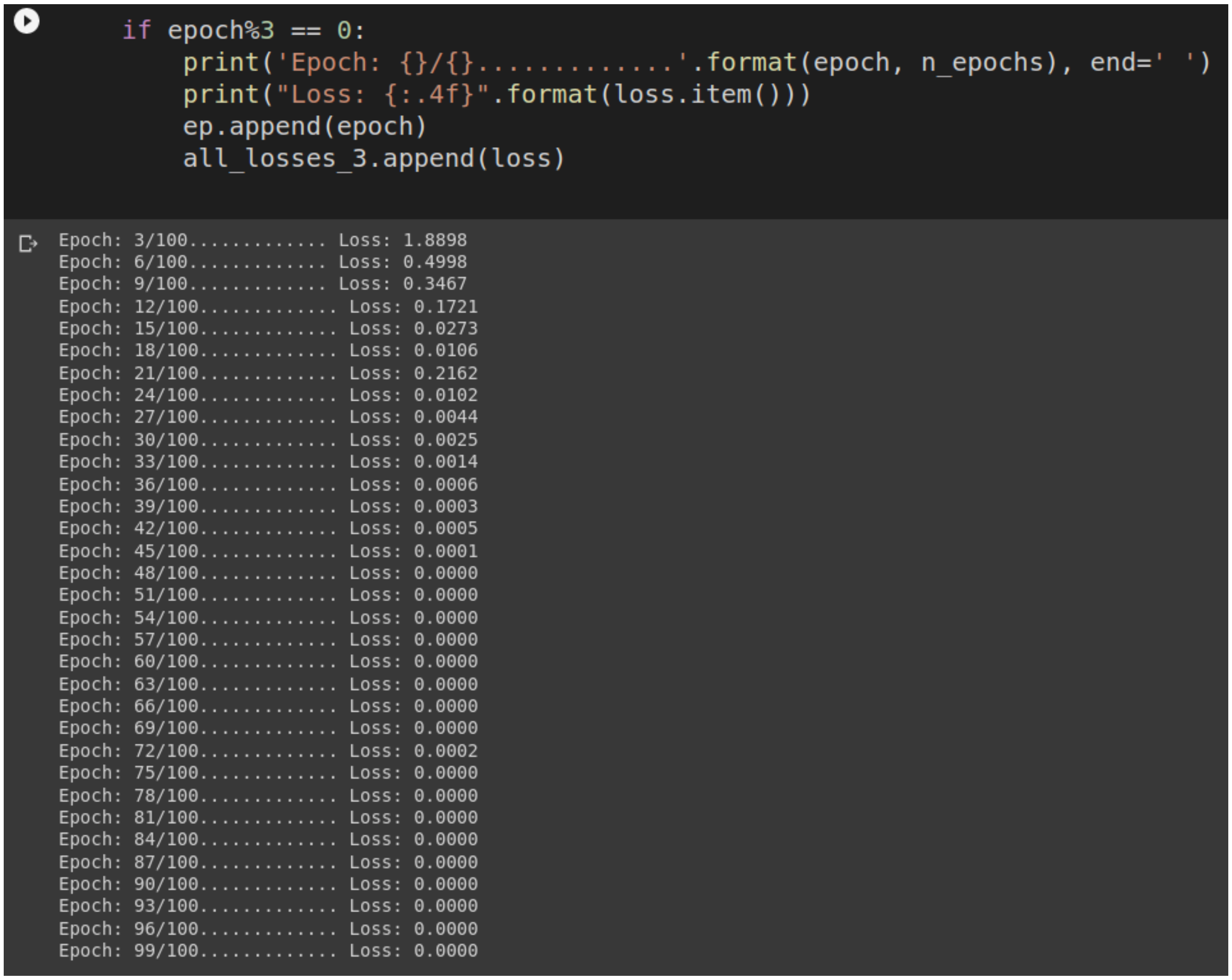
So here as i observed clearly that RMSprop is converging very fast means it just doing the loss convergence to 0.000 at just 50 epochs by adjusting the parameters (alpha at 0.50) .



```
##### Instantiate   the model with hyperparameters
model = Model(input_size=dict_size, output_size=dict_size, hidden_dim=5, n_layers=1)

#### Defining the  hyperparameters

n_epochs =100  ## number of epochs  ---------------------------------------------------------------

###### Define Loss, Optimizer ######

criterion = nn.CrossEntropyLoss()

################################################ RmsProp  #########################################################

optimizer = torch.optim.RMSprop(model.parameters(), lr=0.4,alpha = 0.50, eps=1e-18)
# optimizer

# optimizer = torch.optim.Adam(model.parameters(),lr=0.4,betas=(0.9,0.999),eps=1e-08,weight_decay=0,amsgrad=False)

# optimizer = torch.optim.Adagrad(model.parameters(), lr=0.4,eps=1e-18)
optimizer
```

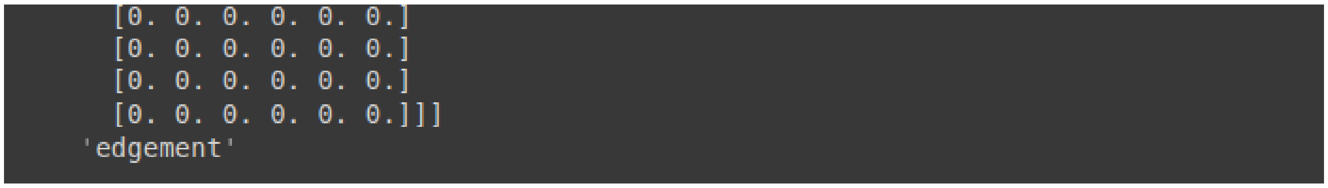
```
RMSprop (
Parameter Group 0
    alpha: 0.5
    centered: False
    eps: 1e-18
    lr: 0.4
    momentum: 0
    weight_decay: 0
)
```

So here i am attaching the epochs output screenshot which is showing the fastness of convergence and it also seen in the above graph plot of epoch vs losses .



```
    if epoch%3 == 0:
        print('Epoch: {}/{}.............'.format(epoch, n_epochs), end=' ')
        print("Loss: {:.4f}".format(loss.item()))
        ep.append(epoch)
        all_losses_3.append(loss)
```

```
Epoch: 3/100............. Loss: 1.8898
Epoch: 6/100............. Loss: 0.4998
Epoch: 9/100............. Loss: 0.3467
Epoch: 12/100............. Loss: 0.1721
Epoch: 15/100............. Loss: 0.0273
Epoch: 18/100............. Loss: 0.0106
Epoch: 21/100............. Loss: 0.2162
Epoch: 24/100............. Loss: 0.0102
Epoch: 27/100............. Loss: 0.0044
Epoch: 30/100............. Loss: 0.0025
Epoch: 33/100............. Loss: 0.0014
Epoch: 36/100............. Loss: 0.0006
Epoch: 39/100............. Loss: 0.0003
Epoch: 42/100............. Loss: 0.0005
Epoch: 45/100............. Loss: 0.0001
Epoch: 48/100............. Loss: 0.0000
Epoch: 51/100............. Loss: 0.0000
Epoch: 54/100............. Loss: 0.0000
Epoch: 57/100............. Loss: 0.0000
Epoch: 60/100............. Loss: 0.0000
Epoch: 63/100............. Loss: 0.0000
Epoch: 66/100............. Loss: 0.0000
Epoch: 69/100............. Loss: 0.0000
Epoch: 72/100............. Loss: 0.0002
Epoch: 75/100............. Loss: 0.0000
Epoch: 78/100............. Loss: 0.0000
Epoch: 81/100............. Loss: 0.0000
Epoch: 84/100............. Loss: 0.0000
Epoch: 87/100............. Loss: 0.0000
Epoch: 90/100............. Loss: 0.0000
Epoch: 93/100............. Loss: 0.0000
Epoch: 96/100............. Loss: 0.0000
Epoch: 99/100............. Loss: 0.0000
```



```
   [0. 0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0. 0.]]]
 'edgement'
```

So here as showing above that losses are converging at 48th epoch and also we are getting the correct output “edgement”.

==============================****=======================**