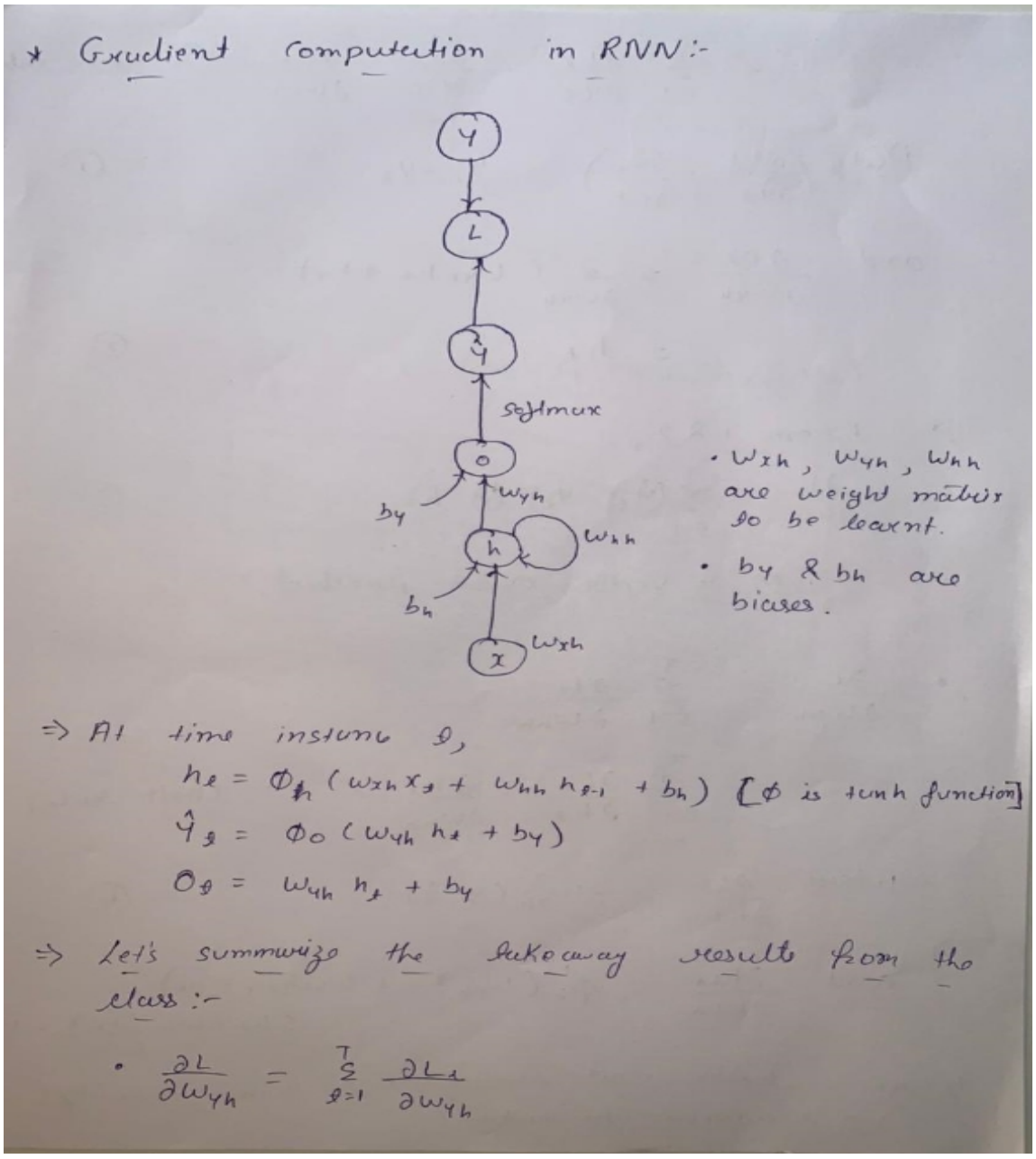
Problem 1: Derive expressions for $\partial L/\partial bh$ and $\partial L/\partial by$ for the RNN discussed in Lectures 2-3. Include the derived bias update equations in the RNN code shared and train the RNN for a sentence/word. Record relevant observations during training after adding bias terms.

Solution: To derive gradients with respect to the biases, I had used the result derived in the lectures. Below I have mentioned the complete calculation for the same.



$$= \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial o_t} \cdot \frac{\partial o_t}{\partial w_{yh}} \quad (\because \text{Chain rule})$$

But, $\left(\frac{\partial L_t}{\partial \hat{y}_t} \cdot \frac{\partial \hat{y}_t}{\partial o_t}\right) = \hat{y}_t - y_t$ -- (1)

and $\frac{\partial o_t}{\partial w_{yh}} = \frac{\partial}{\partial w_{yh}} (w_{yh} h_t + b_y)$

$= h_t$ -- (2)

$\therefore$ From 1 & 2,

$$\frac{\partial L}{\partial w_{yh}} = \sum_{t=1}^{T} (\hat{y}_t - y_t) \otimes h_t \quad \text{-- (3)}$$

- $\otimes$ is vector outer product.

- $\frac{\partial L}{\partial w_{hh}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial w_{hh}}$

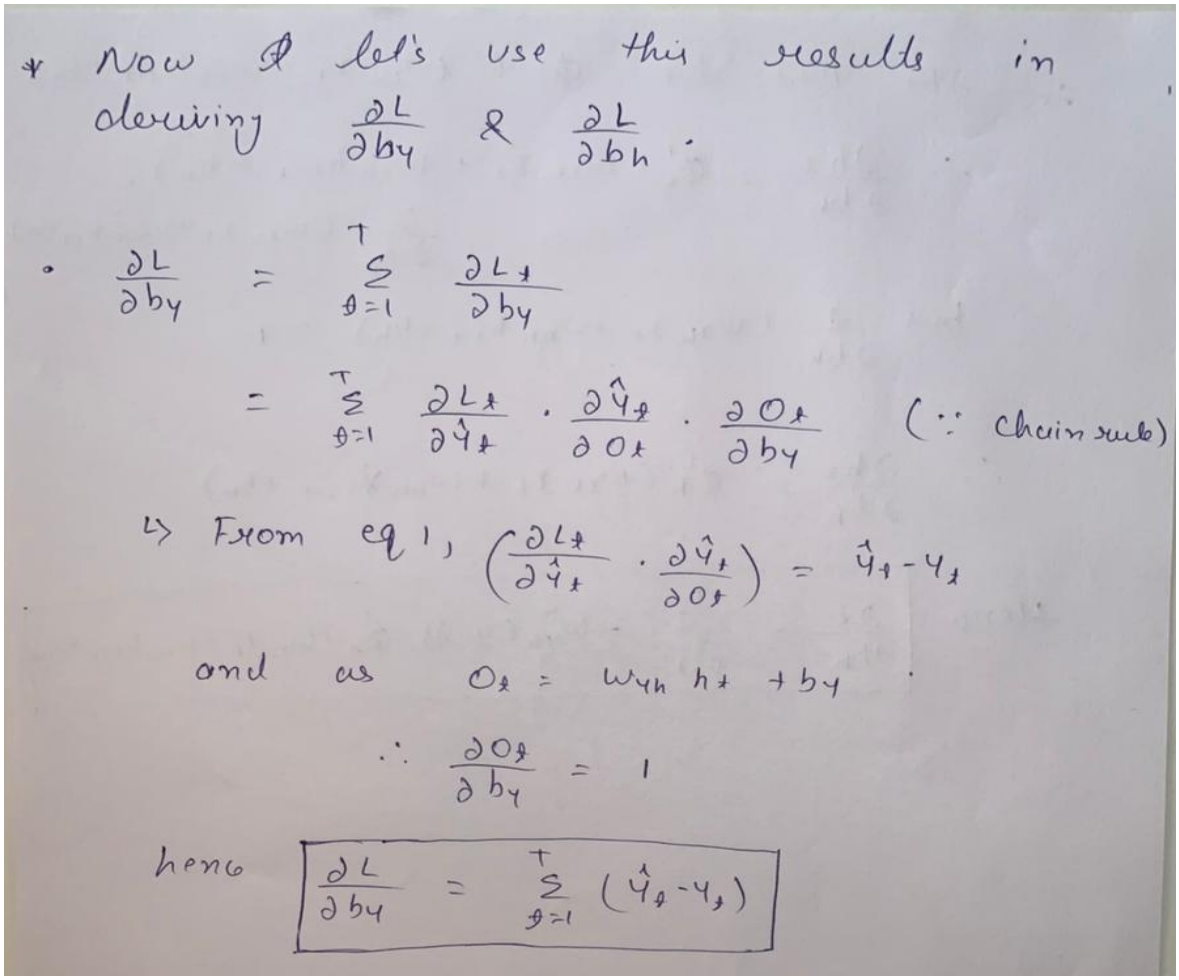
$= \sum_{t=1}^{T} \frac{\partial L_t}{\partial h_t} \cdot \frac{\partial h_t}{\partial w_{hh}} \quad (\because \text{Chain rule})$

where $\frac{\partial L_t}{\partial h_t} = -w_{yh}^T (y - \hat{y})$ -- (4)

and $\frac{\partial h_{t+1}}{\partial w_{hh}} = \phi_h' (w_{xh} x_t + w_{hh} h_{t-1} + b_h) \cdot \left(h_t + w_{hh} \frac{\partial h_t}{\partial w_{hh}}\right)$ -- (5)

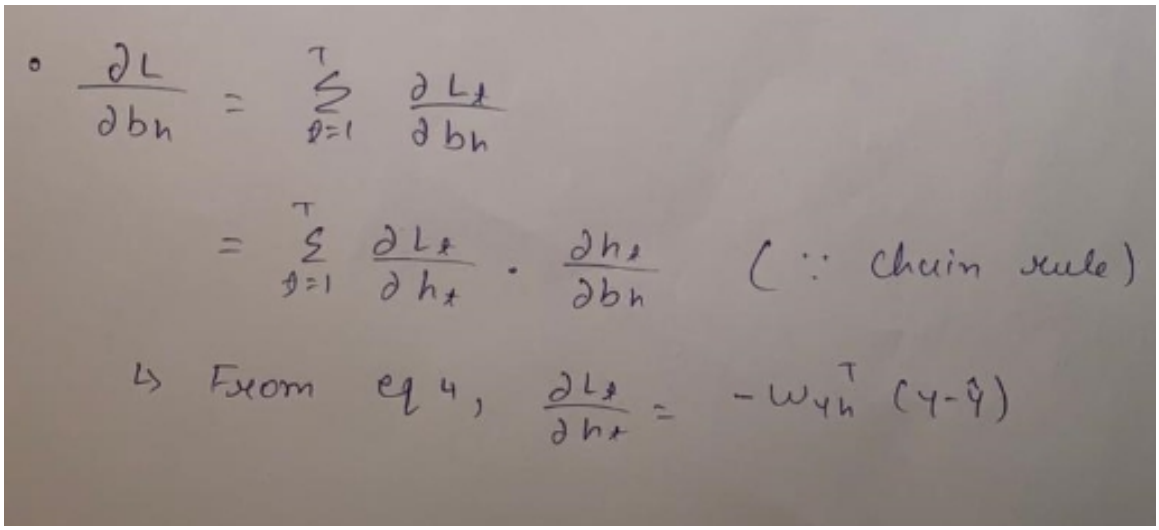
↳ h_t is hidden unit dependend on w_{hh} hence eqs is recursive in nature.

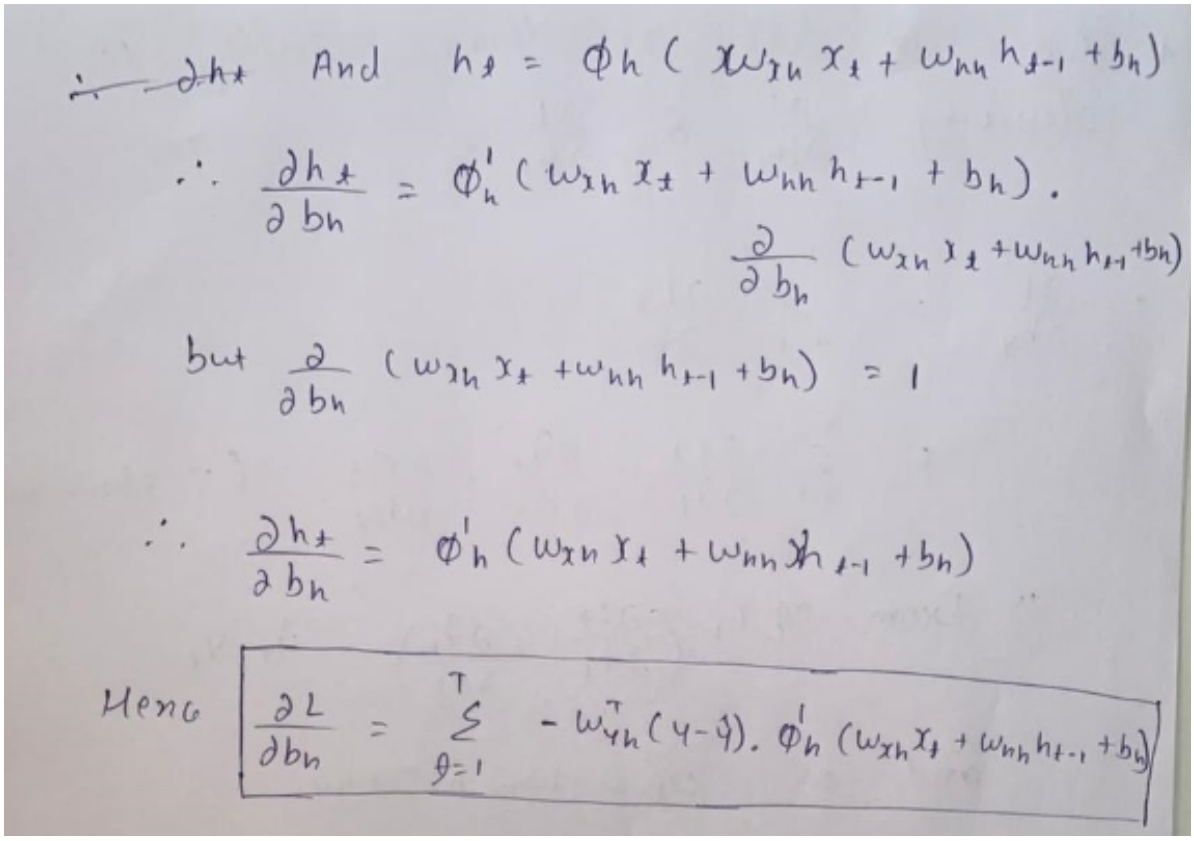
(Derivations from class)



(Derivation of $\partial L/\partial b_y$)

Now let's derive $\partial L/\partial b_h$:





(Derivation of $\partial L/\partial b$h)

Now we need to add this portion in to our code. So below I have mentioned all the changes we need to do to add biases.

Steps:

- First initialize bias variables with respective size in init() method of RNN class.

```
#Added biases
self.bh = np.zeros((hidden_size, 1)) # hidden bias
self.by = np.zeros((vocab_size, 1)) # output bias
```

- Add bias terms in calculation in forward pass.

```
hs[t] = np.tanh(np.dot(self.W_hh,hs[t-1]) + np.dot(self.W_xh,xs[t]) + self.bh)
os[t] = np.dot(self.W_yh,hs[t]) + self.by
```

- To calculate gradients, first declare gradient variables of appropriate size in backward() method of RNN class.

```
#Gradient for biases
dL_dbh = np.zeros_like(self.bh)
dL_dby = np.zeros_like(self.by)
```

- Using the expressions derived above, we will do gradient calculations. Add the below code in the loop to calculate $\partial L/\partial by$.

```
#Compute yhat - y.
d_yy_cap = np.copy(ycap[t])
d_yy_cap[targets[t]]-= 1

# Add this line to calculate gradient with repect to dby.
dL_dby+= d_yy_cap
```

- To calculate $\partial L/\partial b$h we will need derivation of tanh (activation function).

```
# Compute W^Tyh(y-ycap)
dL_dh = np.dot(self.W_yh.T,d_yy_cap) + dhnext
# Diffrentiaion of tanh(x) is (1-tanh^2(x))
dL_dh_dtanh = (1 -hs[t]*hs[t])*dL_dh

#dL_dbh is W^Tyh(y-ycap) * derivation of tanh(hs) so add below line to compute dL_dbh.
dL_dbh+= dL_dh_dtanh
```

Comparison of result with same configuration but without and with biases:

Input: “Amazon is river.”

Output without biases: It did converged at the last 10,000 iterations.

```
iter num 9000 [0.41933032]
Amazon in river.
None
```

Output with biases: It converged at 8000 iterations and loss decreased to 0.3035 .

```
iter num 9000 [0.30353884]
Amazon is river.
```

Problem 2: Replace basic sgd technique used in the function update_model() with any other sophisticated gradient update technique popular in literature. Record relevant observations during training after modifying gradient update method.

Solution:

I have tried 2 gradient update technique namely AdaGrad and RMSprop.

AdaGrad:

AdaGrad (Adaptive Gradient) does not update directly based on learning rate but it actually maintains the running sum of previous gradients and at time of update it divides the learning rate by this running sum. And to avoid division by zero a small offset is used.

Gradient update equation:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

Where vt is,

$$v_t = v_{t-1} + \left[\frac{\partial L}{\partial w_t}\right]^2$$

Code for AdaGrad:

First declare variables to store running sum of gradients.

```
# memory variables
self.mW_xh = np.zeros_like(self.W_xh)
self.mW_hh = np.zeros_like(self.W_hh)
self.mW_yh = np.zeros_like(self.W_yh)
self.mbh = np.zeros_like(self.bh)
self.mby = np.zeros_like(self.by)
```

Call adaGrade() function from update_method.

```
def adaGrade(self, dW_xh, dW_hh, dW_yh,dL_dby, dL_dbh):
  for param, dparam,mem in zip([self.W_xh, self.W_hh, self.W_yh,self.by,self.bh],
                               [dW_xh, dW_hh, dW_yh,dL_dby, dL_dbh],
                               [self.mW_xh, self.mW_hh, self.mW_yh,self.mby, self.mbh]):
    mem += dparam * dparam
    param += -self.learning_rate * dparam / np.sqrt(mem + 1e-3) #Small offset to avoid divison by zero
```

RMSprop:

In RMSprop(Root mean square prop) is similar to adaGrad() but the difference is in RMSprop weights are assigned to current gradient and running sum of gradients.

Gradient update equations:

$$w_{t+1} = w_t - \frac{\alpha}{\sqrt{v_t + \epsilon}} \cdot \frac{\partial L}{\partial w_t}$$

Where,

$$v_t = \beta v_{t-1} + (1 - \beta)\left[\frac{\partial L}{\partial w_t}\right]^2$$

Code for RMSprop:

Call RMSprop() method from update_method.

```
def RMSprop(self, dW_xh, dW_hh, dW_yh,dL_dby, dL_dbh):
  for param, dparam,mem in zip([self.W_xh, self.W_hh, self.W_yh,self.by,self.bh],
                               [dW_xh, dW_hh, dW_yh,dL_dby, dL_dbh],
                               [self.mW_xh, self.mW_hh, self.mW_yh,self.mby, self.mbh]):
    mem = 0.9*mem + (1-0.9)*(dparam * dparam)
    param += -self.learning_rate * dparam / np.sqrt(mem + 1e-3) #Small offset to avoid divison by zer
```

Comparison of SGD, AdaGrad and RMSprop:

Input: "All the faith he had had had had no effect on the outcome of his life."

Output with SGD: Even after 10000 iterations, the loss remained very high and it didn't converged.

```
iter num 9000 [128.40807327]
Alemfeud hlf e d ihd nefeelh ta d hetiad nttc ihefiad h i ilhfohe ust
```

Output with AdaGrad: For AdaGrad too, the loss remained very high after 10000 iterations.

```
iter num 9000 [142.44594829]
Adi nooltoe s ttfahh ofsh dhhfhutahl uheAioec efnadhna otifls dcsie
```

Output with RMSprop: It converged very early i.e at 3000 iteration it started converging and till last iteration loss decreased to 0.07 .

```
iter num 2000 [0.85262771]
All the faith he had had had had no effect on the outcome of his life
iter num 3000 [0.28077867]
All the faith he had had had had no effect on the outcome of his life
```

Input: "Amazon is river."

Output with SGD: It converged at 10000 iterations and loss decreased to 0.3035.

```
iter num 9000 [0.30353884]
Amazon is river.
```

Output with AdaGrad: For AdaGrad , the loss remained very high after 10000 iterations and it did not converged.

```
iter num 9000 [25.3674136]
AvoozAvos. iv vov
```

Output with RMSprop: It converged very early i.e at 2000 iteration with very less loss compared to SGD.

```
iter num 1000 [0.09690172]
Amazon is river.
```

Observation:

From the above results it is clear that performance of RMSprop is best compared to SGD and AdaGrad. Whereas performance of SGD is better compared to AdaGrad.

Problem 3: Experiment with various hidden vector sizes and record your observation.

Solution:

Input: "Amazon is river."

Parameter configuration:

- Learning rate: 0.001
- Iterations: 10000
- Gradient update method: SGD
- Hidden vector size: 10

Output:

```
iter num 9000 [0.30353884]
Amazon is river.
```

With same input and configuration but only hidden vector size increased to 20, the result obtained is as follows:

Output:

```
iter num 5000 [0.30995026]
Amazon is river.
iter num 6000 [0.24641345]
Amazon is river.
```

Now increasing hidden vector size to 30,

```
iter num 2000 [0.72503865]
Amazon is river.
```

It started predicting the output correctly very early but with high loss.

Trying with other input:

Input: "All the faith he had had had had no effect on the outcome of his life"

Parameters: Same parameters just hidden vector size is increased to 30.

Output:

```
iter num 2000 [1.64148478]
All the faith he had had had had no effect on the outcome of his life
iter num 3000 [0.883951]
All the faith he had had had had no effect on the outcome of his life
```

As we saw above, earlier with hidden vector size as 10 and gradient update method as SGD, it was not converging even after 10000 iterations but with hidden vector size increased to 30, it easily converged at very early iteration and till 10000 iteration the loss decreased to 0.21 .

Observation:

From above results we can see that, with increase in size of hidden vectors the result obtained are getting better and this is because as we increase hidden vector size, the RNN is able to store more contextual data and that helps in predicting the next character.