Natural Language Processing

Subhasish Basak (MDS201803)

Assignment 3

April 19, 2020

The link to my **Colab Notebook** is as follows: Click to view Colab Notebook

In this Assignment we implement **Sentiment Analysis** using a simple *Recurrent Neural Network* with one hidden layer.

1 Data Preprocessing

We are working with the **IMDB** movies review dataset. Both the training and testing part contains 25000 reviews half of which are positive and half negative. We preprocess the data using the following steps:

- **Removing punctuation & symbols** : This step includes removing unecessary punctuation and symbols present in the reviews.Since we are intereted in the english words only we preprocess the data to remove them. Regular Expressions are used for the purpose.
- **Removing the stopwords** : Words having too high or too low frequencies are of no use in our study. Most commonly occuring words like "the", "a" etc are present in all the reviews and contribute nothing in our analysis. Similarly there are some rare words also which have no discriminating power. We remove all such words.
- **Lemmatizing & Stemming the words** : Lemmatizing & Stemming are essential part of data preprocessing. This helps in reducing the vocabulary size.
- **Tokenizing the words** : After the preprocessing till this step we tokenize each unique word in our vocabulary. All the reviews are then encoded with these tokens.
- **Resizing reviews** : In order to input the review vectors in a RNN we need all of them of the same size, i.e. this sequence length is same as number of time steps for the RNN. To deal with both short and long reviews, we will pad the small ones and truncate all the large reviews to a specific length. We denote this length by **review_length**.
  For reviews shorter than *review_length*, we will pad with 0s. For reviews longer than *review_length* we will truncate them to the first *review_length* words.
- **Filtering words** : Even after data preprocesing the vocabulary size is 184069. Hence we work with a subset of it by removing the low frequency words. We observed that the top $5k$ most frequent words account for 86% of the total frequency. Hence in order to reduce computation time and increase efficiency we restrict our vocabulary to $5k$ words only.
- **One hot encoding** : Now to input a review into the Neural Network we need to encode each word using One Hot Encoding. Since we have 5000 unique words in our vocabulary , each word in the review will be represented as a vector of size 5000×1.

2 The RNN Architecture

Here we start with a **Recurrent Neural Network (RNN)** framework with only 1 hidden layers. Our Neural network has the following configuration :

- Input Layer
  - **Number of nodes** : 300
  - Corresponding to each time-step ($t = 1(1)300$) there is a english word from the review.
  - The english word is represented as a One-Hot vector of size 5000×1.
- Hidden Layer
  - **Number of nodes** : 10
  - Only one hidden layer.
- Output Layer
  - **Number of nodes** : 2
  - Outputs the probability of the input review to fall in two sentiment classes viz. **1 :** ***positive*** and **0 :** ***negative***.

Our aim is to classify each review to two classes **positive** or **negative**, i.e. given a 300×1 input vector (each element of which is a One-Hot vector of size 5000×1) representing a 80 word review our Neural Network should output the probability vector (a 2×1 vector representing the probability of the review falling in the two sentiment classes.)

3 Building the Recurrent Neural Network

Since this is a classification problem, we are using a **many to one** RNN.

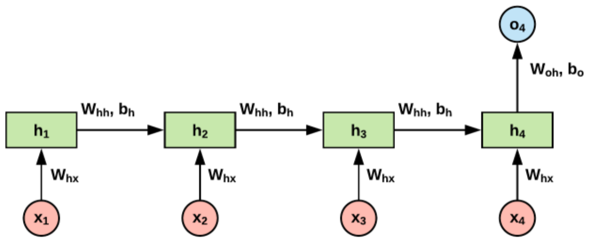

Figure 1: Many to one RNN

Each **x_i** is a One hot vector (of shape 10000×1) representing a word from the review. The output **o** is a vector containing two numbers, one representing positive and the other negative. We have applied ***Softmax*** to turn them into probabilities to decide positive / negative sentiment.

- **Initialization** : We initialize the weights and biases randomly from $U(\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$, where n is the input size. We have the following weights and biases connecting the nodes of our network,
  - W_x : input(x_t) $\longrightarrow$ hidden unit(h_t)
  - W_h : hidden unit($h_{(t-1)}$) $\longrightarrow$ hidden unit(h_t)
  - W_y : hidden unit(h_t)) $\longrightarrow$ output(y_t)
  - b_h : Added while computing h_t
  - b_y : Added while computing y_t

- **Forward Pass** : The main equations which constructs the architecture of the network is given as follows,

$$h_t = \tanh(W_x x_t + W_h h_{t-1} + b_h) \tag{1}$$

$$y_t = W_y h_t + b_y \tag{2}$$

  Here we have used the tanh as the activation function. After y_t is computed we apply the **Softmax** function on it to generate the class probabilities.

- **Back Propagation Through Time (BPTT)** : Now to train the RNN and update the weights we implement **BPTT**. For updating the weights we need the gradients i.e. the dreivatives of the loss function w.r.t the weights and biases. Her we have used the **Categorical Cross Entropy** loss function given by $L = -\log(softmax(y))$. The gradients are derived as below,

$$\frac{\delta L}{\delta y_i} = \begin{cases} p_i & , i \neq c \\ p_i - 1 & , i = c \end{cases} \tag{3}$$

$$\frac{\delta L}{\delta W_y} = \frac{\delta L}{\delta y}\frac{\delta y}{\delta W_y} = \frac{\delta L}{\delta y} h_n \tag{4}$$

  where n is the last time stamp (or, the input size). Similarly we get the following,

$$\frac{\delta L}{\delta b_y} = \frac{\delta L}{\delta y} \tag{5}$$

  For the other derivatives w.r.t W_x, W_h and b_h, we compute the following,

$$\frac{\delta h_t}{\delta W_x} = (1 - h_t^2)x_t, \qquad \frac{\delta h_t}{\delta W_h} = (1 - h_t^2)h_{t-1}, \qquad \frac{\delta h_t}{\delta b_h} = (1 - h_t^2) \tag{6}$$

  We also make use of a recursive equation,

$$\frac{\delta y}{\delta h_t} = \frac{\delta y}{\delta h_{t+1}}(1 - h_t^2)W_h \quad \text{and} \quad \frac{\delta y}{\delta h_n} = W_y \tag{7}$$

  Now using (3), (6) & (7) we can compute the derivatives are as follows,

$$\frac{\delta L}{\delta W_x} = \frac{\delta L}{\delta y}\sum_t \frac{\delta y}{\delta h_t}\frac{\delta h_t}{\delta W_x} \tag{8}$$

$$\frac{\delta L}{\delta W_h} = \frac{\delta L}{\delta y}\sum_t \frac{\delta y}{\delta h_t}\frac{\delta h_t}{\delta W_h} \tag{9}$$

4 Implementation in Python

In this section we describe the code for implementing the RNN. We have build a class called **RNN** which consists of 3 functions as follows,

- **forward** : This function takes a single review as input and outputs a 2×1 vector without applying the softmax activation. The elements of the input are again is consist of *One Hot vectors* of size 5000×1.

- **Softmax** : This is simple implantation of the *Softmax* activation function. This is to be applied on the output of the **foreward** function.

- **BPTT** : This function implements the *Back Propagation Through Time* algorithm. It takes input the derivative $\frac{\delta L}{\delta y}$ (which is calculated using equation (3))and the learning rate.

In [1]:
```python
class RNN:

    def forward(self, inputs):

        # Here the function takes a singe review
        # input is a list of length 300 & vocab_size is 5000
        # each element of the input list is one_hot_vector of size (vocab_size, 1)

        # initializing the Zero-th hidden layer (one corresponding to 0-th time stamp)
        h = np.zeros((self.Wh.shape[0], 1))

        # List for storing the hidden layer units
        self.hidden_layer = list()

        # storing the inputs and the initial hidden layer for future use
        self.inputs = inputs
        self.hidden_layer.append(h)

        # Performing the forward pass for each time step
        for i, j in enumerate(inputs):

            h = np.tanh(np.matmul(self.Wx, j) + np.matmul(self.Wh, h) + self.bh)
            self.hidden_layer.append(h)

        # Final computation of the output using the last hidden layer
        output = np.matmul(self.Wy, h) + self.by

        # Softmax Function for the output array.
        # Outputs 2-class probabilities which sums up to 1
        return np.exp(output) / sum(np.exp(output))
```

4.1 forward

This function does the following :

- **Initialization**
  - a **zero array** as the hidden layer for the 0-th time stamp.
  - a dictionary for storing the all the hidden layer units. The keys are the time stamps ($t = 0(1)n$).
- **Iteration**
  - The main loop in this function implements equation (1) for each of the elements in the given input.
- **Output**
  - The last line of code outside the loop, to compute the output implements equation (2).

4.2 BPTT

The function does the following:

- **Initialization**
  - **Zero arrays** are initiated to store the derivatives of $\frac{\delta L}{\delta W_x}$, $\frac{\delta L}{\delta W_h}$ and $\frac{\delta L}{\delta b_h}$.
- **Direct computation**
  - The quantities $\frac{\delta L}{\delta W_y}$ and $\frac{\delta L}{\delta b_y}$ are computed directly from equation (4) and (5).
  - $\frac{\delta L}{\delta h}$ is also computed for the last hidden layer.
- **Iteration**
  - The main **BPTT** loop implements the equations (8) and (9) to compute $\frac{\delta L}{\delta W_x}$, $\frac{\delta L}{\delta W_h}$ and $\frac{\delta L}{\delta b_h}$. It considers the additional derivative computations from equation (6) and the recursive relation from equation (7).

- **Output**
  - The output is computed using the update equation at the end which implements *Gradient descent* algorithm with the learning rate taken from the input.

In [2]:
```
class RNN:
...

    def BPTT(self, d_y, rate=0.2, bounds = [-1.5, 1.5]):

        # This function performs backpropagation through time (BPTT)
        # Initialization : Derivative arrays initialized to zero
        d_Wh = np.zeros(self.Wh.shape)
        d_Wx = np.zeros(self.Wx.shape)
        d_bh = np.zeros(self.bh.shape)

        # Direct computations
        d_Wy = np.matmul(d_y, self.hidden_layer[len(self.inputs)].T)
        d_by = d_y
        d_h = np.matmul(self.Wy.T, d_y)

        # Updating the gradients
        self.Wy -= rate * np.clip(d_Wy, a_min=bounds[0], a_max=bounds[1])
        self.by -= rate * np.clip(d_by, a_min=bounds[0], a_max=bounds[1])

        # recursive computations
        for i in range(len(self.inputs))[::-1]:

            d_Wh += np.matmul(((1 - np.square(self.hidden_layer[i + 1]))*d_h), self.hidden_la
            d_Wx += np.matmul(((1 - np.square(self.hidden_layer[i + 1]))*d_h), self.inputs[i]
            d_bh += ((1 - np.square(self.hidden_layer[i + 1]))*d_h)
            d_h = np.matmul(self.Wh, ((1 - np.square(self.hidden_layer[i + 1]))*d_h))

        # Updating the gradients

        self.Wh -= rate * np.clip(d_Wh, a_min=bounds[0], a_max=bounds[1])
        self.Wx -= rate * np.clip(d_Wx, a_min=bounds[0], a_max=bounds[1])
        self.bh -= rate * np.clip(d_bh, a_min=bounds[0], a_max=bounds[1])
```

5 Observations

Our RNN model was able to learn the sentiments fro m the sequence of words given as inputs. With the following configuration of the hyperparameters we were able to get the results:

- Hyperparameters
  - **vocabulary size** : $5k$
  - **review length** : 300
  - **learning rate** : 0.2
  - **hidden layes nodes** : 15
- Results
  - **Training accuracy** : 54%
  - **Test accuracy** : 52%
  - **Number of epochs** : 5
  - **Running time/epoch** : 52 mins