CSE616 Neural Networks and Their Applications
Assignment 3

Shahira Hany Hussein Mohamed Amin (2101341)

May 2022

1 Problem 1

1.1 Part 1

$$h_1 = W_h h_0 + W_x x_1 = 1 \times 1 + 0.1 \times 10 = 2$$
$$h_2 = W_h h_1 + W_x x_2 = 1 \times 2 + 0.1 \times 10 = 3$$
$$\hat{y}_2 = W_y h_2 = 2 \times 3 = 6$$

1.2 Part 2

$$\hat{y}_1 = W_y h_1 = 2 \times 2 = 4$$

$$L_t = \sum_{i=1}^{2} (\hat{y}_i - y_i)^2 = (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 = (4-5)^2 + (6-5)^2 = 2$$

1.3 Part 3

$$\begin{aligned}
\frac{\partial L_t}{\partial h_1} &= \frac{\partial L_t}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial h_1} + \frac{\partial L_t}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial h_2}\frac{\partial h_2}{\partial h_1} \\
&= 2(\hat{y}_1 - y_1)W_y + 2(\hat{y}_2 - y_2)W_y W_h \\
&= 2 \times (4-5) \times 2 + 2 \times (6-5) \times 2 \times 1 \\
&= 0
\end{aligned}$$

1.4 Part 4

$$\begin{aligned}
\frac{\partial L_t}{\partial W_h} &= \frac{\partial L_t}{\partial \hat{y}_1}\frac{\partial \hat{y}_1}{\partial h_1}\frac{\partial h_1}{\partial W_h} + \frac{\partial L_t}{\partial \hat{y}_2}\frac{\partial \hat{y}_2}{\partial h_2}\Big(\frac{\partial h_2}{\partial W_h} + \frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_h}\Big) \\
&= 2(\hat{y}_1 - y_1)W_y h_0 + 2(\hat{y}_2 - y_2)W_y(h_1 + W_h h_0) \\
&= 2 \times (4-5) \times 2 \times 1 + 2 \times (6-5) \times 2 \times (2 + 1 \times 1) \\
&= 8
\end{aligned}$$

2 Problem 2

At time step t, the derivative of the cost function J_t is calculated as:

$$\frac{\partial J_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial J_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W}, \text{ where } \frac{\partial h_t}{\partial h_k} = \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \dots \frac{\partial h_{k+1}}{\partial h_k} \tag{1}$$

As the gap between the time steps k and t gets larger, the product in Eq. (1) gets longer and longer. Given that:

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t), \text{ where } \frac{\partial h_t}{\partial h_{t-1}} = W_{hh}^T diag(f^{'}(W_{hh}h_{t-1} + W_{xh}x_t)) \tag{2}$$

where $f^{'}$ is the derivative of the $tanh$. W_{hh}^T is sampled from a standard normal distribution, so it is mostly < 1. Also, since $f^{'}$ is the derivative of the $tanh$, then $f^{'} < 1$. Therefore, in Eq. (1), we are multiplying lots of small numbers together which leads to the vanishing gradients problem (i.e. the network fails to learn long-term dependencies).

3 Problem 3

The use of GRUs would be more efficient than vailla RNNs when the network needs to process long sequences. The reason for this is that a vanilla RNN focuses only on the latest information and cannot store the information in long sequences due to the vanishing gradients problem. To solve the vanishing gradients problem and control the information flow, GRU incorporates two gates: the update gate, which determines the amount of previous information to be passed along the next state, and the reset gate, which decides whether the previous cell state is important or not.

4 Problem 4

In truncated back propagation through time (BPTT), the input sequence is processed one time step at a time. Every k_1 time steps, a back propagation through time (BPTT) update is performed for a truncated/limited fixed number of time steps k_2. The *advantages* of truncated BPTT are:

1. It relieves the need for a complete backtrack through the whole data sequence (which is expensive). Thus, BBTT results in faster parameter updates.

2. It avoids the vanishing gradient problem, which occurs when the back-propagated errors get smaller and smaller, layer by layer.

However, the *disadvantage* of truncated BPTT is that it favors short-term dependencies, i.e., truncated BPTT is not able to capture dependencies that are longer than the truncated length.

5 Problem 5

5.1 Part a

RNNs will perform much better than CNNs because RNNs are designed to recognize sequential or temporal data whereas CNNs are preferred in interpreting visual data that does not come in sequence. RNNs make better predictions due to their ability to relate both past data and input data and memorize things.

5.2 Part b

The model used in this problem is a character-level RNN since the cipher works on the character level. A character-level RNN takes as input an integer referring to a specific character and outputs another integer referring to the character that the input was mapped to.

5.3 Part c

The model is a many-to-many architecture because a sequence of multiple steps as input is mapped to a sequence of multiple steps as output. That is, each time step, a character in the input sequence is mapped to another character in the cipher.

5.4 Part d

Data is pre-processed as follows:

1. Isolate each character in the input sequence as an array element
2. Tokenize the characters, i.e. turn them from letters to integers

For example, if the input sample is 'andtwo', then the first step is to isolate each character in the input sequence as an array element as follows:

$$\begin{bmatrix} 'a' & 'n' & 'd' & 't' & 'w' & 'o' \end{bmatrix} \tag{3}$$

The second step is to tokenize the characters to turn each character into an integer as follows:

$$\begin{bmatrix} 2 & 14 & 5 & 12 & 22 & 15 \end{bmatrix} \tag{4}$$

The output sample corresponding to the input sample 'andtwo' is 'dqgwzr', and will be tokenized as:

$$\begin{bmatrix} 5 & 4 & 6 & 22 & 17 & 9 \end{bmatrix} \tag{5}$$

5.5 Part e

A good way to handle variable length data is to find the longest input sequence and pad shorter sequences with zeros. Therefore, all input sequences will be of the same length. For example, let "one" and "andtwo" are two input sequences. After isolating each character in the input sequence and tokenizing the characters, "one" will be an array of $\begin{bmatrix}15 & 14 & 1\end{bmatrix}$ and "andtwo" will be an array of $\begin{bmatrix}2 & 14 & 5 & 12 & 22 & 15\end{bmatrix}$. Since "andtwo" is the longer sequence, then "one" will be padded to be of the same length as "andtwo" as follows $\begin{bmatrix}15 & 14 & 1 & 0 & 0 & 0\end{bmatrix}$.

5.6 Part f

To be able to train the model, we will need to pre-process the data as follows:

1. Isolate each character as an array element
2. Tokenize the characters to turn them from characters to integers
3. Pad the strings so that all the input and output sequences can fit in matrix form because most machine learning platforms expect the input to be a matrix (of dimension number of data samples $\times$ length of the longest sequence). For the two samples in the example in Part e ("one" and "andtwo"), the input matrix will be constructed as follows:

$$\begin{bmatrix} 15 & 14 & 1 & 0 & 0 & 0 \\ 2 & 14 & 5 & 12 & 22 & 15 \end{bmatrix} \tag{6}$$

5.7 Part g

As shown in Fig. 1, assuming that the input sequence is of length 10 characters, then the input layer is of shape 10×1, where 10 is the temporal dimension (i.e. at each time step, the RNN takes one character as input). The input layer is followed by a Gated Recurrent Unit (GRU) layer with 64 units. The output of the GRU layer is of shape 10×64, where 10 is the temporal dimension (i.e. at each time step, the GRU layer outputs 64 values). The GRU layer is followed by a Dense (fully-connected) layer with 26 units and soft-max activation function. The output of the Dense layer is of shape 10×26, where 10 is the temporal dimension (i.e. at each time step, the dense layer outputs the probability that the input character at this time step is mapped to each of the 26 alphabet characters).

```
def simple_model(input_shape, output_sequence_length, code_vocab_size, plaintext_vocab_size):
    """
    Build and train a basic RNN on x and y
    :param input_shape: Tuple of input shape
    :param output_sequence_length: Length of output sequence
    :param code_vocab_size: Number of unique code characters in the dataset
    :param plaintext_vocab_size: Number of unique plaintext characters in the dataset
    :return: Keras model built, but not trained
    """
    # TODO: Build the model
    learning_rate = 1e-3
    input_seq = Input(input_shape[1:])
    rnn = GRU(64, return_sequences = True)(input_seq)
    logits = TimeDistributed(Dense(plaintext_vocab_size))(rnn)
    model = Model(input_seq, Activation('softmax')(logits))
    model.compile(loss = sparse_categorical_crossentropy,
                  optimizer = Adam(learning_rate),
                  metrics = ['accuracy'])

    return model
```

```
In [32]: simple_rnn_model = simple_model((1000,10,1),10,26,26)
```

```
In [33]: simple_rnn_model.summary()

Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 10, 1)]           0

 gru_3 (GRU)                 (None, 10, 64)            12864

 time_distributed_2 (TimeDis  (None, 10, 26)           1690
 tributed)

 activation_2 (Activation)   (None, 10, 26)            0

=================================================================
Total params: 14,554
Trainable params: 14,554
Non-trainable params: 0
_________________________________________________________________
```

Figure 1: RNN Model Summary

5.8 Part h

For each time step, the integer that gave the maximum Dense layer output is selected to be the cipher of the input character at this time step. The output integer is converted to a letter using the tokenizer. The outputs (from all time steps) are concatenated to obtain the cipher sequence.