ASSIGNMENT 3

CISC 867: Deep Learning

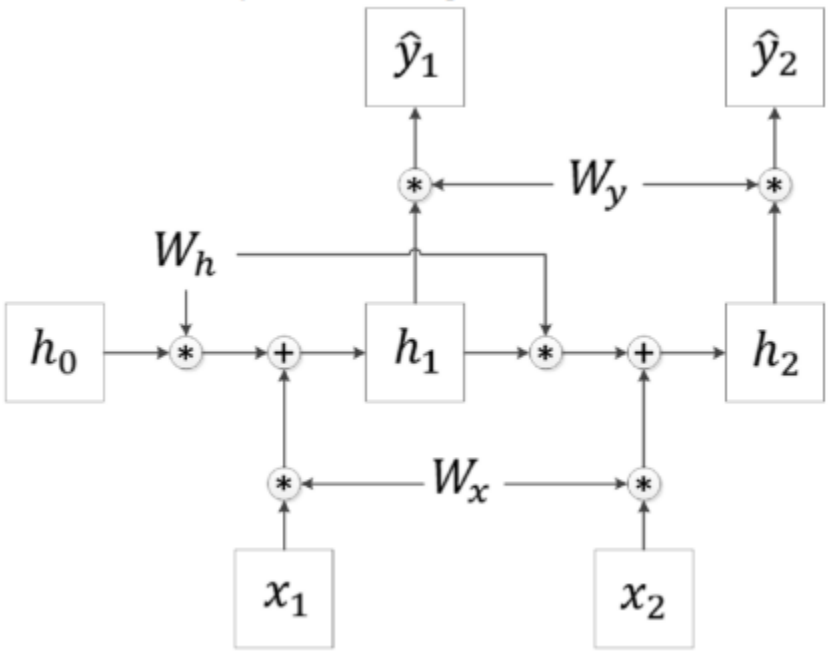
Name: Hanan Fared Mohamed Omara.

ID: 20398559

DR/ Hazem Abbas

Eng/ Asif Mahfuz

1. **A RNN network is illustrated below (as a computational graph). Assume that the identity function is used to generate the output of all neurons.**



Assume that the input at a given time, $h_0 = 1$, $x_1 = 10$, $x_2 = 10$, $y_1 = 5$, and $y_2 = 5$. The initial values of the weights are: $W_h = 1$, $W_x = 0.1$, $W_y = 2$. Answer the following questions:

Assignment 3

CISC 867 : Deep learning

Name: Hanan Fared Mohamed oMara

ID : 20398559

1 Question 1 :-

1=(a) $h_1 = w_x X_1 + w_h h_0 = (0.1 * 10 + 1 * 1) = 2$

$\hat{y}_2 = w_y (w_h h_1 + w_x X_2) = 2(1*2 + 0.1*10) = 6$

2=b) $\hat{y}_1 = w_y (w_h h_0 + w_x X_1) = 2(1*1 + 0.1*10) = 4$

$L_1 = (\hat{y}_1 - y_1)^2 = (4-5)^2 = 1$

$L_2 = (\hat{y}_2 - y_2)^2 = (6-5)^2 = 1$

$L = L_1 + L_2 = 2$

---

3 =(c) $\frac{\partial L_1}{\partial h_1} = 2 (\hat{y}_1 - y_1) \frac{\partial \hat{y}_1}{\partial h_1} = 2 (\hat{y}_1 - y_1) w_y =$

$2 * (4-5) * 2 = -4 = -4$

$h_2 = w_h h_1 + w_x X_2 = 1*2 + 0.1*10 = 3$

$\hat{y}_2 = w_y (w_h h_1 + w_x X_2) = 2 * (1*2 + 0.1*10) = 6$

$\frac{\partial \hat{y}_2}{\partial h_1} = w_y w_h = 2*1 = 2$

$\rightarrow \frac{\partial L_2}{\partial h_1} = 2 (\hat{y}_2 - y_2) \frac{\partial \hat{y}_2}{\partial h_1} = 2 (\hat{y}_2 - y_2) 2 = 2(6-5)*2 = 4$

$\rightarrow \frac{\partial L}{\partial h_1} = \frac{\partial L_1}{\partial h_1} + \frac{\partial L_2}{\partial h_1} = (-4) + (4) = 0$

4) $\frac{\partial L}{\partial w_h} = \frac{\partial L_1}{\partial w_h} + \frac{\partial L_2}{\partial w_h}$

$\frac{\partial y_1}{\partial w_h} = w_y h_0 = 2 * 1 = 2$

$\frac{\partial L_1}{\partial w_h} = 2(\hat{y}_1 - y_1)\frac{\partial y_1}{\partial w_h} = 2(4-5) * 2 = -4$

$\frac{\partial y_2}{\partial w_h} = \frac{\partial (w_y (w_h h_1 + w_x x_2))}{\partial w_h} = \frac{\partial (w_y w_h h_1)}{\partial w_h}$

$\frac{\partial y_2}{\partial w_h} = \frac{\partial (w_y w_h h_1)}{\partial w_h} = w_y h_1 + w_y w_h \frac{\partial h_1}{\partial w_h}$

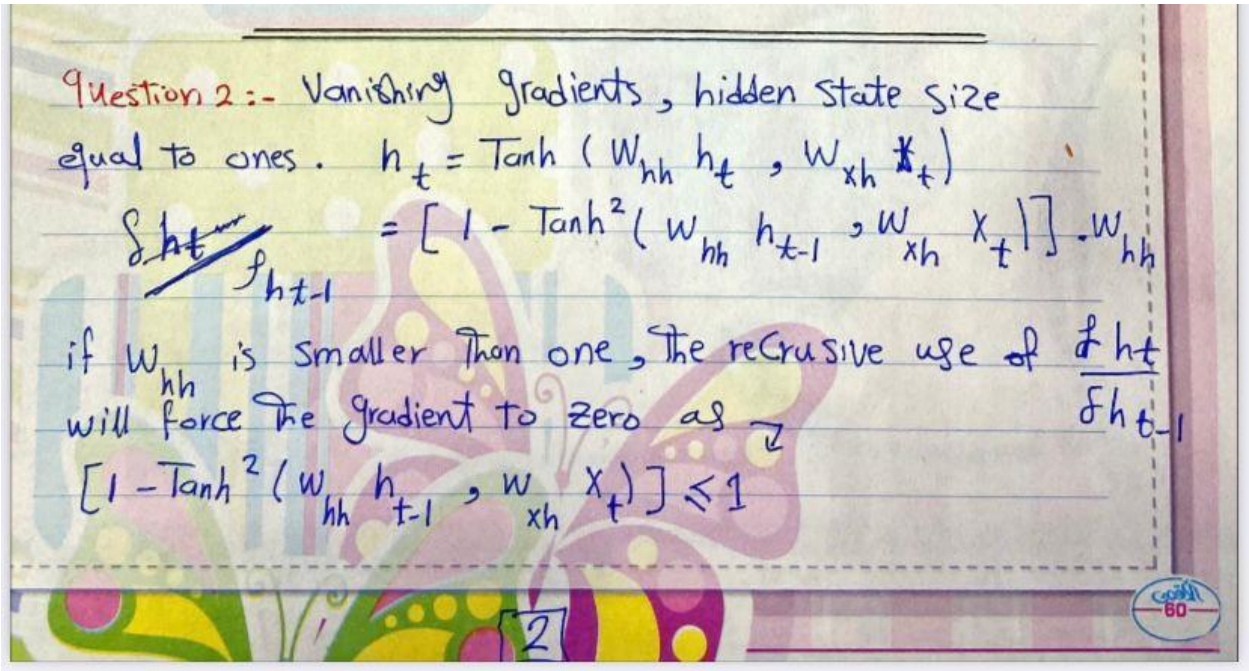
$= w_y h_1 + w_y w_h h_0$

$\therefore \frac{\partial y_2}{\partial w_h} = 2 * 2 + 2 * 1 * 1 = 6$

$\frac{\partial L_2}{\partial w_h} = 2(\hat{y}_2 - y_2)\frac{\partial y_2}{\partial w_h} = 2(6-5) * 6 = 12$

$\frac{\partial L}{\partial w_h} = \frac{\partial L_1}{\partial w_h} + \frac{\partial L_2}{\partial w_h} = (-4) + 12 = 8.$

2. Why are long term dependencies difficult to learn in a RNN? You may use this equation to explain your answer. $H_t = \tanh(W_{hh} h_{t-1}, W_{xh} X_t)$



Long-term dependencies are difficult to learn in a traditional RNN because of the vanishing gradient problem. The vanishing gradient problem occurs because the gradient of the loss function with respect to the network's parameters, which is propagated backwards through time during backpropagation, can become very small as it is multiplied by the derivative of the activation function at each time step. This means that the influence of earlier inputs on the current output can become negligible.

The equation $H_t = \tanh(W_{hh} h_{t-1}, W_{xh} X_t)$ shows how the hidden state h_t at time t depends on the previous hidden state ht-1 and the current input x_t. If the value of W_{hh} is much smaller than 1, then $W_{hh}^{(t-k)}$ will become exponentially smaller as the time lag k increases, and the gradient will vanish. This means that the RNN will have difficulty in retaining information over long time lags, and will not be able to learn long-term dependencies effectively.

To address this issue, several variants of RNNs have been developed, such as LSTMs and GRUs, which introduce gating mechanisms to selectively retain and forget information in the hidden state, allowing them to learn long-term dependencies more effectively**.**

3- When would the use of Gated Recurrent Units (GRU) be more efficient than vanilla RNNs?

- The use of Gated Recurrent Units (GRU) would be more efficient than vanilla RNNs in situations where long-term dependencies need to be modeled and retained over many time steps.
- GRUs are a type of recurrent neural network that are similar to LSTMs, but with fewer parameters, which makes them faster to train and computationally more efficient. GRUs have gating mechanisms that allow them to selectively update or forget information in the hidden state, which helps to mitigate the vanishing and exploding gradient problem that can occur in vanilla RNNs.
- GRUs are particularly effective in tasks where long-term dependencies need to be modeled and retained, such as speech recognition, language modeling, and machine translation. They have been shown to be especially effective in tasks where the sequence length is variable and can be quite long, as they are able to selectively retain important information over long periods of time.
- In addition, GRUs are less prone to overfitting compared to vanilla RNNs, which can lead to better generalization performance on unseen data. Therefore, in situations where long-term dependencies need to be modeled, and computational efficiency and generalization performance are important considerations, GRUs would be a more efficient choice than vanilla RNNs.

4- What are the advantage and disadvantage of Truncated Backpropagation Through Time (TBTT)?

- Truncated Backpropagation Through Time (TBPTT) is a variant of the Backpropagation Through Time (BPTT) algorithm for training Recurrent Neural Networks (RNNs). TBPTT breaks the long sequence of inputs into smaller subsequences or "chunks", and computes the gradients for each chunk separately before updating the model parameters.

- Advantages of TBPTT:

1. Reduced memory requirements: TBPTT reduces the memory requirements during training, as it only needs to store the activations and gradients for the current chunk, rather than the entire sequence.
2. Faster training: TBPTT can be faster to train than BPTT, especially for long sequences, as it reduces the amount of time required to compute the gradients and update the model parameters.
3. Improved generalization: TBPTT can help to improve the generalization performance of the model, as it prevents the gradients from being propagated too far back in time, which can lead to overfitting.

- Disadvantages of TBPTT:

1. Loss of information: TBPTT may lead to a loss of information, as it only updates the model parameters based on the current chunk, rather than the entire sequence. This can lead to suboptimal performance, especially for long sequences where the chunks may not capture all the relevant information.
2. Suboptimal gradients: TBPTT computes the gradients based on a truncated sequence, which can result in suboptimal gradients that do not take into account the entire sequence. This can lead to slower convergence and suboptimal performance.
3. Chunk size selection: Choosing the appropriate chunk size can be challenging, as a smaller chunk size can reduce memory requirements and improve training speed, but may lead to a loss of information, while a larger chunk size may capture more information but may be computationally inefficient.

Overall, TBPTT can be a useful technique for training RNNs, especially for long sequences or when memory requirements are a concern. However, the choice of chunk size and the potential loss of information should be carefully considered when using this technique.

5. You are required to define a simple RNN to decrypt a Caesar Cipher. A Caesar Cipher is a cipher that encodes sentences by replacing the letters by other letters shifted by a fixed size. For example, a Caesar Cipher with a left shift value of 3 will result in the following:

Input: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher: DEFGHIJKLMNOPQRSTUVWXYZABC

Notice that there is a 1-to-1 mapping for every character, where every input letter maps to the letter below it. Because of this property you can use a character-level RNN for this cipher, although word-level RNNs may be more common in practice. Answer the following questions:

a. **A Caesar Cipher can be solved as a multiclass classification problem using a fully-connected feedforward neural network since each letter X maps to its cipher value Y. However, an RNN will perform much better. Why?**

   - A fully-connected feedforward neural network can indeed solve a Caesar Cipher as a multiclass classification problem, where each letter X maps to its cipher value Y. However, an RNN will perform much better because it can take into account the sequential nature of the cipher. In a Caesar Cipher, each letter is replaced by

another letter shifted by a fixed size, so the value of each letter depends on the value of the previous letter.

- An RNN can model this sequential dependency by maintaining a hidden state that captures the context of the previous letters. The hidden state can be updated at each time step based on the current input letter and the previous hidden state, and can be used to predict the output letter. This allows the RNN to take into account the entire sequence of letters, and to make more accurate predictions based on the context of the previous letters.

- In addition, an RNN can handle variable-length sequences, which is important for decoding messages of different lengths. A feedforward neural network, on the other hand, would require a fixed input size, which may not be practical for decoding messages of different lengths.

Overall, an RNN can perform much better than a fully-connected feedforward neural network for solving a Caesar Cipher because it can model the sequential nature of the cipher and handle variable-length sequences.

b. Describe the nature of the input and output data of the proposed model.

The input data for the proposed RNN model to decrypt a Caesar Cipher would be a sequence of encrypted characters, where each character is represented as a one-hot vector encoding. For example, if we consider the Caesar Cipher with a left shift value of 3, the input sequence "HELLO" would be represented as follows:

```
[[0 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],  # H
 [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],  # E
 [0 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],  # L
 [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],  # L
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]  # O
```

The output data would be a sequence of decrypted characters, where each character is represented as a one-hot vector encoding. For the same example of a Caesar Cipher with a left shift value of 3, the output sequence for the input "HELLO" would be:

```
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0],  # E
```

[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], # B

[0 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], # O

[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0], # L

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]] # H

Note that the input and output sequences have the same length, as each input character maps to a unique output character. The goal of the RNN is to learn the mapping between the encrypted characters and the decrypted characters, so that it can accurately decrypt new sequences of characters.

c. Will the model by a character-level one-to-one, one-to-many, or many-to-many (matching) architecture**? Justify your answer.**

- The proposed RNN model to decrypt a Caesar Cipher will have a many-to-many (matching) architecture. This is because the input sequence and output sequence have the same length, and each input character corresponds to a unique output character. In other words, the input sequence and output sequence are aligned on a one-to-one basis.

- In a many-to-many (matching) architecture, the RNN takes a sequence of inputs and generates a sequence of outputs, where each input corresponds to a unique output. This is in contrast to a one-to-one architecture, where the RNN takes a single input and generates a single output, or a one-to-many architecture, where the RNN takes a single input and generates a sequence of outputs.

- Since the proposed RNN model for decrypting a Caesar Cipher takes a sequence of encrypted characters as input and generates a sequence of decrypted characters as output, it will have a many-to-many (matching) architecture. The RNN will process each encrypted character in the input sequence and generate the corresponding decrypted character in the output sequence, based on the context of the previous characters. This allows the RNN to capture the sequential dependency of the cipher and accurately decrypt the input sequence.

d. How should the training data look like? Give an example of a sample input and the corresponding output.

- The training data for the RNN to decrypt a Caesar Cipher should consist of pairs of input and output sequences, where each input sequence is a sequence of encrypted characters and each output sequence is the corresponding sequence of decrypted characters.

- For example, if we consider a Caesar Cipher with a left shift value of 3, the training data might look like:

Input sequence: "HELLO"
Output sequence: "EBBOH"

Here, the input sequence "HELLO" represents the encrypted message, and the output sequence "EBBOH" represents the decrypted message. Each character in the input sequence is encoded as a one-hot vector, and the same is done for each character in the output sequence.

The training data would consist of many such pairs of input/output sequences, where each pair represents a different message encrypted with the same Caesar Cipher. The RNN would learn to map each input sequence to the corresponding output sequence by adjusting its weights during training, so that it can accurately decrypt new messages.

It's worth noting that the training data would need to be generated using a known shift value, and the model would only be able to decrypt messages that were encrypted using the same shift value.

e. What is a good way to handle variable length texts?

- A good way to handle variable length texts in the proposed RNN model for decrypting a Caesar Cipher is to use padding and masking. Padding involves adding zeros to the end of shorter sequences so that they have the same length as longer sequences. Masking involves setting the values of the padded zeros to zero, so that they do not contribute to the computation of the RNN.

- In the case of a Caesar Cipher, where each input and output sequence has the same length, padding and masking would not be necessary. However, if we were working with variable-length texts, such as messages of different lengths, we would need to use padding and masking to ensure that all sequences have the same length.

- For example, we could set a maximum sequence length, and then pad all sequences to that length by adding zeros to the end. We would also need to create a mask that indicates which elements of the sequence are padding and which are not. During training and inference, the mask would be used to ignore the padded elements and only compute the loss and gradients for the non-padded elements.

Overall, using padding and masking is a good way to handle variable length texts in RNNs, as it allows us to work with sequences of different lengths and to process them efficiently using batch processing.

f. In order for the model to function properly, the input text has to go through several steps. For example, the first step is to tokenize is the text, i.e., to convert it into a series of characters. What should be the other required steps in order to train the model?

In addition to tokenizing the input text into a series of characters, there are several other required steps to train the proposed RNN model for decrypting a Caesar Cipher:

1. Encoding the characters: Each character in the input and output sequences needs to be encoded as a one-hot vector. This involves creating a vocabulary of all possible characters (including the padding character), and then representing each character as a vector of zeros with a 1 in the position corresponding to that character in the vocabulary.

2. Padding the sequences: As mentioned earlier, we may need to pad the input and output sequences to ensure that they have the same length. This involves adding zeros to the end of shorter sequences so that they have the same length as the longest sequence in the dataset.

3. Creating batches: To speed up training, we typically process multiple input/output pairs at once, or in batches. This involves grouping multiple input/output pairs together into a batch, and then feeding the batch through the RNN at once.

4. Splitting the dataset: The dataset of input/output pairs needs to be split into training, validation, and testing sets. The training set is used to update the model parameters, the validation set is used to tune the model hyperparameters, and the testing set is used to evaluate the model performance on new data.

5. Preprocessing the data: Depending on the specific characteristics of the data, additional preprocessing steps may be necessary, such as normalization or data augmentation.

Overall, these steps are necessary to properly preprocess the input and output data and prepare it for training the RNN model. Once the data is preprocessed, the model can be trained using backpropagation through time (BPTT) to learn the mapping between the encrypted and decrypted characters in the Caesar Cipher.

g. What should be the architecture of the simple RNN that can be used? You might use Keras API to describe the architecture?

The architecture of the simple RNN for decrypting a Caesar Cipher would typically involve an input layer, one or more LSTM or GRU layers, and an output layer. Here is an example architecture using the Keras API:

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Define the model architecture
model = Sequential()
model.add(LSTM(64, input_shape=(max_sequence_length, num_characters)))
model.add(Dense(num_characters, activation='softmax'))

# Compile the model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

In this example, the model consists of a single LSTM layer with 64 hidden units, followed by a dense output layer with softmax activation. The input shape is set to `(max_sequence_length, num_characters)`, where `max_sequence_length` is the maximum length of the input and output sequences (after padding) and `num_characters` is the number of possible characters in the vocabulary (including the padding character).

The model is compiled with the categorical cross-entropy loss function, the Adam optimizer, and accuracy as a metric for evaluation.

Note that this is just an example architecture, and the specific architecture used will depend on the specific requirements of the problem and the characteristics of the data. It may be necessary to experiment with different numbers of layers, hidden units, and other hyperparameters to achieve the best performance on the task of decrypting the Caesar Cipher.

h. Are there any processing operations that should be applied to the output in order to generate the deciphered text?

Yes, there are processing operations that should be applied to the output of the RNN in order to generate the deciphered text. Specifically, we need to convert the one-hot vector representation of each decrypted character back into its corresponding character in the vocabulary.

One common approach is to use the `argmax` function to find the index of the highest value in the output vector, and then use this index to look up the corresponding character in the vocabulary. For example:

```python
# Convert the output from one-hot encoding to characters
decrypted_chars = []
for output_vector in output_sequence:
    # Find the index of the highest value in the output vector
    index = np.argmax(output_vector)
    # Look up the corresponding character in the vocabulary
    char = vocabulary[index]
    # Add the character to the decrypted message
    decrypted_chars.append(char)

# Convert the list of decrypted characters back into a string
decrypted_text = ''.join(decrypted_chars)
```

In this example, `output_sequence` is the sequence of decrypted characters generated by the RNN, and `vocabulary` is the list of possible characters in the same order as their corresponding one-hot encoding vectors.

Alternatively, we can use a decoding function that takes into account the probability distribution over the possible characters in the output vector, rather than simply selecting the most likely character. One common approach is beam search decoding, which involves keeping track of the top-k most likely character sequences at each time step and selecting the one with the highest overall probability.

Overall, the processing operations applied to the output of the RNN will depend on the specific requirements of the problem and the characteristics of the data, and may involve additional steps beyond simply converting the one-hot vectors back into characters.