# CSE616: Assignment (3)

1. 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: $W_h = 1$, $W_x = 0.1$, $W_y = 2$.

$a)\ h_t = W\begin{bmatrix} x_t \\ h_{t-1} \end{bmatrix} = W_x * x_t + W_h * h_{t-1}$ $\qquad \hat{y_t} = W_y * h_t$

$h_1 = W\begin{bmatrix} x_1 \\ h_0 \end{bmatrix} = W_x * x_1 + W_h * h_0 = 0.1 * 10 + 1 * 1 = 2$ $\qquad \hat{y_1} = W_y * h_1 = 2 * 2 = 4$

$h_2 = W\begin{bmatrix} x_2 \\ h_1 \end{bmatrix} = W_x * x_2 + W_h * h_1 = 0.1 * 10 + 1 * 2 = 3$ $\qquad \hat{y_2} = W_y * h_2 = 2 * 3 = 6$

$b)\ L_t = \sum_i (\hat{y_i} - y_i)^2 = (\hat{y_1} - y_1)^2 + (\hat{y_2} - y_2)^2 = (4 - 5)^2 + (6 - 5)^2 = 2$

$c)\ L_t = (\hat{y_1} - y_1)^2 + (\hat{y_2} - y_2)^2$ $\qquad \hat{y_1} = W_y * h_1$

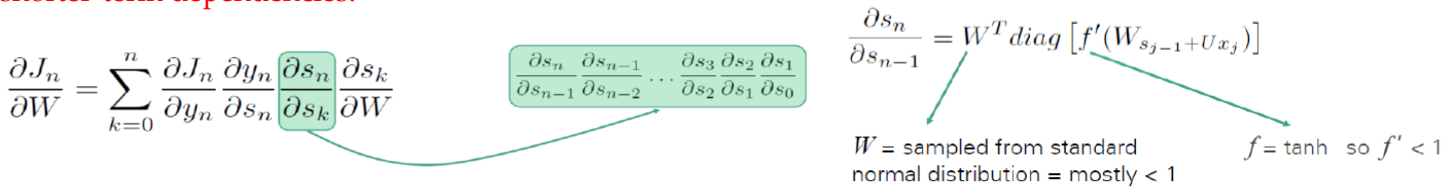
$\frac{\partial L_t}{\partial h_1} = \frac{\partial L_t}{\partial \hat{y_1}} \frac{\partial \hat{y_1}}{\partial h_1} = 2(\hat{y_1} - y_1) * W_y = 2(4 - 5) * 2 = -4$

$d)\ \frac{\partial L_t}{\partial w_h} = \frac{\partial L_t}{\partial \hat{y_1}} \frac{\partial \hat{y_1}}{\partial h_1} \frac{\partial h_1}{\partial W_h} = 2(\hat{y_1} - y_1) * W_y * h_0$ $\qquad h_1 = W_x * x_1 + W_h * h_0$

$\frac{\partial L_t}{\partial W_h} = 2(4 - 5) * 2 * 1 = -4$

2. Why are long term dependencies difficult to learn in a RNN? You may use this equation to explain your answer.

Because of Vanishing Gradient Problem, we're multiplying a lot of small numbers together. So that errors due to further back time steps have increasingly smaller gradients, and parameters become biased to capture shorter-term dependencies.




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

Careful initialization and optimization of vanilla RNNs can enable them to learn long(ish) dependencies, but gated additive cells, like the LSTM and GRU, often justwork.

4. What are the advantage and disadvantage of Truncated Backpropagation Through Time.

In truncated backprop through time (TBPTT), the input is treated as fixed length subsequences. For forward pass, the hidden state of previous subsequence is passed on as input to the next subsequence. However, in gradient computation, the computed gradient values are dropped at the end of every subsequence as we walk back. That is in standard backprop the gradient values at time ***t'*** used in every time step ***t*** if ***t < t'***. In truncated backprop, the gradients do not flow from ***t'*** to ***t*** if ***t'-t*** exceeds the subsequence length. While this approach cuts down computation and memory requirements, it introduces a bias.

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.

a) A Caesar Cipher can be solved as a multiclass classification problem using a fully-connected feed forward neural network since each letter X maps to its cipher value Y. However, an RNN will perform much better. Why?
RNNs are best suited for tasks that require prediction of the next data; they are used widely on temporal data like a sequence of words. When the input is a sequence of words, the RNN will predict the next word of the sequence. It does not require a dataset of labeled data to achieve its function of prediction.

b) Describe the nature of the input and output data of the proposed model.
A character-level RNN model will take as input an integer referring to a specific character and output another integer which we can cover back to a character. By providing a training set of encrypted sentences and their corresponding, the model will learn how to decode and translate back to plain text.

c) Will the model by a character-level one-to-one, one-to-many, or many-to-many (matching) architecture? Justify your answer.
The model will be a character-level many-to-many, because Machine Translation can be thought of as a sequence-to-sequence learning problem.

d. How should the training data look like? Give an example of a sample input and the corresponding output.
The dataset we have consists of 10,000 encrypted phrases and the plaintext version of each encrypted phrase.

```
Plaintext [:5] = ['THE LIME IS HER LEAST LIKED FRUIT , BUT THE BANANA IS MY LEAST LIKED .',
 'HE SAW A OLD YELLOW TRUCK .',
 'INDIA IS RAINY DURING JUNE , AND IT IS SOMETIMES WARM IN NOVEMBER .',
 'THAT CAT WAS MY MOST LOVED ANIMAL .',
 'HE DISLIKES GRAPEFRUIT , LIMES , AND LEMONS .']
Cipher [:5] = ['YMJ QNRJ NX MJW QJFXY QNPJI KWZNY , GZY YMJ GFSFSF NX RD QJFXY QNPJI .',
 'MJ XFB F TQI DJQQTB YWZHP .',
 'NSINF NX WFNSD IZWNSL OZSJ , FSI NY NX XTRJYNRJX BFWR NS STAJRGJW .',
 'YMFY HFY BFX RD RTXY QTAJI FSNRFQ .',
 'MJ INXQNPJX LWFUJKWZNY , QNRJX , FSI QJRTSX .']
```

e. What is a good way to handle variable length texts?
A typical approach is to use a recurrent layer to encode the meaning of the sentence by processing the words in a sequence, and then either use a dense or fully-connected layer to produce the output, or use another decoding layer.

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?

For a neural network to predict on text data, it first has to be turned into data it can understand. Since a neural network involves a series of multiplication and addition operations, the input data needs to be numbers. Preprocessing of the input text is therefore necessary to get the model to work. Key preprocessing steps are:

- Isolating each character (instead of an entire phrase, or word being the element)
- Tokenizing the characters so we can turn them from letters to integers and vice-versa
- Padding the strings so that all the inputs and outputs have similar sequence length. This allows to fit the data into matrix form and benefit from the matrix computation capabilities of the machine

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

```
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
    """
    x = Input(shape=input_shape[1:])   # shape(none,54,1) ie
    # output must be batchsize x timesteps x units
    seq = GRU(units= 64, return_sequences = True, activation="tanh", name='Layer1')(x)
    output = TimeDistributed(Dense(units = plaintext_vocab_size, activation='softmax', name='Layer2'))(seq)
    model = Model(inputs = x, outputs = output)
    model.compile(optimizer='adam', loss= sparse_categorical_crossentropy, metrics=['accuracy'])
    model.summary()
    return model
```

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

Convert the output from a neural network back into text using the tokenizer

```
def logits_to_text(logits, tokenizer):
    """
    Turn logits from a neural network into text using the tokenizer
    :param logits: Logits from a neural network
    :param tokenizer: Keras Tokenizer fit on the labels
    :return: String that represents the text of the logits
    """
    index_to_words = {id: word for word, id in tokenizer.word_index.items()}
    index_to_words[0] = '<PAD>'
    return ' '.join([index_to_words[prediction] for prediction in np.argmax(logits, 1)])
```