Robeir Samir George 1901823

Assignment 3 (Part3)

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?

<u>Solution:</u>

I think using fully connected feedforward network will be much better as we are dealing with a 1-to-1 mapping problem, since the input is one-to-one the fully connected will perform good. I think the main problem will be the weights, but due to the limited number of characters in the language so the neurons at the input and the output layers will have limited number of neurons.

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

<u>Solution:</u>

Each character will be turned into numbers (integers). Each character is going to have a unique id called character ID.
The input and output are going to be numbers (IDs) which indicate and refer to certain characters.

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

Solution:

A character-level **one-to-one** architecture will be used since the Caesar cipher seems to work on the character level. Each encryption process doesn't depend on the previous encryption process. Each character is encrypted/decrypted in isolation from the other characters.

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

Solution:

```
Input:  The quick brown fox jumps over the lazy dog.
Output After Tokenization: [16, 6, 2, 1, 17, 9, 4, 10, 18, 1, 28, 7, 3, 19,
11, 1, 20, 3, 21, 1, 29, 9, 22, 14, 5, 1, 3, 23, 2, 7, 1, 12, 6, 2, 1, 24,
13, 25, 8, 1, 26, 3, 27, 1, 15]
Output After Padding: [16  6  2  1 17  9  4 10 18  1 28  7  3 19 11  1 20
3 21  1 29  9 22 14  5
  1  3 23  2  7  1 12  6  2  1 24 13 25  8  1 26  3 27  1 15  0  0  0  0  0
  0  0  0  0]
```

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

Solution:

Since sentences are dynamic in length, we can add **padding** to the end of the sequences to make them the same length.

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?

Solution:

After tokenization we are going to use "**Padding**"

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

Solution:

```
from keras.layers import SimpleRNN

def simple_model(input_shape, output_sequence_length, code_vocab_size,
plaintext_vocab_size):
    x = Input(shape=input_shape[1:])
    seq = SimpleRNN(units= 64, return_sequences = True, activation="tanh",
name='Layer1')(x)   # output must be batchsize x timesteps x units
    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?

Solution:

We can turn each character into a number or each word into a number. These are called character and word ids, respectively. Character ids are used for character level models that generate text predictions for each character. A word level model uses word ids that generate text predictions for each word. Word level models tend to learn better.