Hangman Strategy Writeup

Name: Sourodeep Datta

Roll Number: 21CS10064 (IIT Kharagpur)

Email: sourodeepdatta@gmail.com

1. Data Preparation

Using the given training dataset, I generated three files, "input_train.txt", "target_train.txt", and "wrong_guesses.txt". input_train.txt consisted of words from the given dataset, with random characters being replaced with "_" s. target_train.txt consisted of the corresponding complete words. wrong_guesses.txt consists of a random number of characters, not in the target word.

To generate these files, for each word of the provided dataset, I iterated through the number of possible characters that could be replaced with "_" s, which is in the range of [0, number of unique characters in the word]. For each of these, I generated up to 7 different combinations of characters and underscores, replacing different sets of characters each time. For each of these combinations, I iterated through the range [0, 11], for each index i, sampling i different characters not in the target word and using them as wrong guesses. Finally, I shuffled the created data and wrote them into their corresponding files.

This process created a training dataset of nearly 105 million examples, with the files looking like:

input_train.txt	target_train.txt	wrong_guesses.txt
t__m_ng	terming	fqb
su_op_i_i_a_io_	suboptimization	fkdhgcrwq
runnin_bir__	runningbirch	dvolmztk

2. Vocabulary

For the model to take these words as inputs, each character needs to be mapped to an integer index. For the input vocabulary, the characters in the range 'a' to 'z' are mapped to integers in the range [1, 26] lexicographically. '_' is mapped to 27. Integer 0 is reserved for the '<PAD>' token, which will be used later.

For the output vocabulary, the characters in the range 'a' to 'z' are mapped to integers in the range [0, 25], lexicographically.

3. Data Generator

Due to the large size of the training data, instead of loading it all at once, I created a data generator that would read the three files line by line parallelly and yield the appropriate training data.

The word from input_train.txt is encoded according to the previously mentioned input vocabulary. After encoding, it is post-padded to increase its length to 29, the max sequence length. This is done to train the model in mini-batches greater than 1. This encoded sequence is returned as input_tokens.

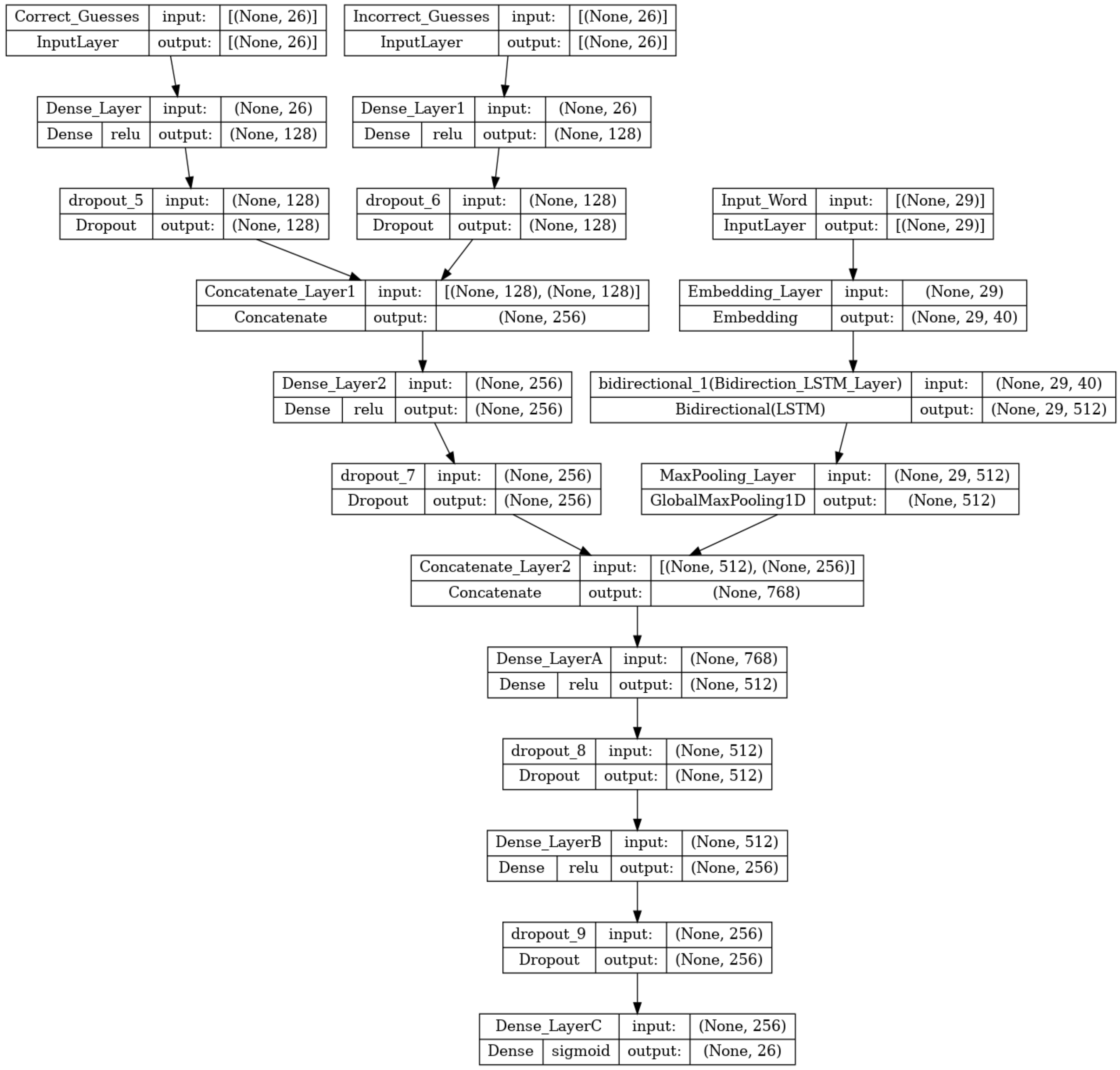
For example, the encoded sequence of "t__m_ng" is:
[20, 27, 27, 13, 27, 14, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

The word from target_train.txt is used to calculate the set of characters present in the target, let that be set A. Let the set of characters from the word in input_train.txt be set B. Then the set C = A – B is the set of all missing characters in the input. This is used to create a multi-hot encoding of size 26, which is returned as target_tokens. For example, the target_tokens of "terming" is:
[0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]

The words form input_train.txt and target_train.txt are used to create a 3rd sequence called correct_tokens of size 26, consisting of characters in the input word other than "_". Even though the input_tokens can also be used to determine this, due to the model architecture mentioned later, having it input separately is useful.
For example, the correct_tokens of "t__m_ng" is:
[0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0]

Finally, the word from wrong_guesses.txt is used to create the sequence incorrect_tokens of size 26, which multi-hot encoding of characters which were guessed incorrectly.
For example, the incorrect_tokens of "t__m_ng" is "fqb", encoded as:
[0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]

4. Model Architecture


Correct_Guesses | input: | [(None, 26)]
InputLayer | output: | [(None, 26)]
Incorrect_Guesses | input: | [(None, 26)]
InputLayer | output: | [(None, 26)]
Dense_Layer | input: | (None, 26)
Dense | relu | output: | (None, 128)
Dense_Layer1 | input: | (None, 26)
Dense | relu | output: | (None, 128)
dropout_5 | input: | (None, 128)
Dropout | output: | (None, 128)
dropout_6 | input: | (None, 128)
Dropout | output: | (None, 128)
Input_Word | input: | [(None, 29)]
InputLayer | output: | [(None, 29)]
Concatenate_Layer1 | input: | [(None, 128), (None, 128)]
Concatenate | output: | (None, 256)
Embedding_Layer | input: | (None, 29)
Embedding | output: | (None, 29, 40)
Dense_Layer2 | input: | (None, 256)
Dense | relu | output: | (None, 256)
bidirectional_1(Bidirection_LSTM_Layer) | input: | (None, 29, 40)
Bidirectional(LSTM) | output: | (None, 29, 512)
dropout_7 | input: | (None, 256)
Dropout | output: | (None, 256)
MaxPooling_Layer | input: | (None, 29, 512)
GlobalMaxPooling1D | output: | (None, 512)
Concatenate_Layer2 | input: | [(None, 512), (None, 256)]
Concatenate | output: | (None, 768)
Dense_LayerA | input: | (None, 768)
Dense | relu | output: | (None, 512)
dropout_8 | input: | (None, 512)
Dropout | output: | (None, 512)
Dense_LayerB | input: | (None, 512)
Dense | relu | output: | (None, 256)
dropout_9 | input: | (None, 256)
Dropout | output: | (None, 256)
Dense_LayerC | input: | (None, 256)
Dense | sigmoid | output: | (None, 26)

The model was made with Tensorflow using the Keras Functional API. It was trained using a compute cluster from Microsft Azure.

The model consists of 3 inputs, Input_Word, Correct_Guesses and Incorrect_Guesses.

Input_Word is passed into an Embedding layer, creating embeddings of size 40 for each token. The embedding size is not larger than 40 to prevent the embeddings from becoming too sparse, due to the small size of the vocabulary. Following this, it is passed into a bidirectional LSTM layer, which is set to return the hidden state for each timestep. A GlobalMaxPoolingLayer follows this, meant to decrease the dimension size by 1, to be passed into following dense layers, as well as keep the important features learnt from the previous layers.

The Correct_Guesses and Incorrect_Guesses are respectively passed into 2 different dense layers, after which they are concatenated and passed into one dense layer. They are then concatenated with the output from the Input_Word section of the network.

This concatenated value is then passed through 3 dense layers, with the final dense layer of size 26, used to predict the missing characters.

All the dense layers in the network except the last one have the ReLU activation function, except the last one, which has a sigmoid activation function used for multi-class classification.

Adams optimizer is used for the model, with a learning rate of 0.001. The loss function is Binary Focal Cross Entropy. This is used instead of the regular Binary Cross Entropy function for 2 reasons. Firstly, due to the sparse nature of the target tokens, the model may give every class a low probability to decrease the loss while not making the model accurate. This is prevented by increasing the weights of the sparse classes in the model. Secondly, it gives higher weights to more difficult examples in the dataset, thus ensuring the model focuses more on reducing their loss.

Technically, using only Input_Word and Incorrect_Guesses would have been sufficient for inputs, as the Input_Word would have implicitly included the Correct_Guesses. However, I found that including them separately increases the model's accuracy by roughly 5%, which I believe is because when passing through the LSTM layer, some of the data on the present characters may be lost. Along with this, if only Incorrect_Guesses had been passed through a dense layer, while the model may have learnt the set of characters that are not present in the word and should not be focused on, it would not be able to learn what characters are present efficiently, and thus not focus on characters typically found alongside these present characters.

The Mini batch size to train this model was 2048, trained for 5 epochs.

5. Creating guess(self, word)

In the guess function, the inputs for the model were pre-processed similarly to how it was done in the data generator section, except that it was not padded (as prediction would be done one sequence at a time). The set of probabilities for each class was then ordered from higher to lowest, with the character with the highest probability that has not been predicted yet being returned. This is essentially a greedy strategy.

6. Conclusion

The model achieves a success rate of roughly 60%, with the final submission having a 63.1% success rate. It is possible that the model can be further improved by increasing the training dataset, including more unique words, increasing the latent dimension size of the LSTM layer, or by possibly adding more LSTM layers.