Strategy:

- If the length of the word is greater than 30, guess the letter which occurs in the maximum number of words in the training dictionary and that has not been guessed before.
- Otherwise,
    - First guess is ‘e’.
    - Second guess is ‘i’.
    - Third guess is ‘a’.
    - After making the above three guesses, encode the masked_word and process it using an Bidirectional LSTM Classifier. Take softmax of the output and guess the letter with maximum probability of occurrence among the letters that had not been guessed previously.

Explanation:

- In the training dictionary, the maximum length of a word is 29. Only in an extremely rare case, we could get a word with length greater than 30 during testing. So, it is sufficient to implement a simple algorithm for such cases. Hence, we guess the letter which occurs in the maximum number of words in the training dictionary and that has not been guessed before.

- Almost all the words have less than or equal to 30 letters. We approach them as follows:

- Since the task is to implement a non n-gram ML solution for the Hangman solver, I considered all the classical ML and DL models. I eliminated all the classical ML models and DL models which don’t capture the sequential information. So, I had to choose between RNN, LSTM and Attention models. Considering the performance and computational resources available I decided to implement a Bidirectional LSTM Classification network.

- I came up with a strategy to guess a certain set of fixed letters in the beginning and then use the LSTM network to play the rest of the game. This is because this could reduce the size of the dataset for training the LSTM model and improve its performance.

- I found the fixed set of letters by choosing the letters which occur in the most number of words in the training dictionary. Note that I didn't choose the letters which occur the most in the dictionary. This is because it doesn't matter how many times letters repeat in a word.

- I found the size of the set by intuition and trial and error. Thus I came up with a fixed set of {'e','i','a'}.

- Generating the training dataset for the LSTM model: For each word in the training dictionary, I generated their masked words by randomly replacing all letters but the letters in the fixed set with underscore. Training dataset has 6.2 M samples.

- Input Encoding: I used numerical representation to encode the underscore and alphabets. I mapped the alphabets 'a-z' to 1-26 and '_' to 27. I used this mapping to convert the masked word into a numerical sequence. Then I padded zeros before the encoded sequence to get a sequence of fixed length 30. For example, 'a_e' is encoded as [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,27,5].

- Output Encoding: To encode the target word, I used a 26 dimensional vector/tensor where the i^{th} element corresponds to the i^{th} alphabet and it is equal to 1 if the alphabet is present in the target word, otherwise it is zero. For example, 'ate' is encoded as [1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0].

- Model: Embedding layer followed by 2 LSTM layers and then feedforward neural network.

- Performance: My final score is 56.9. I could have got a much higher score if I had used a more complex LSTM network, larger training dataset and increased the number of epochs. Due to time and computational constraints, I couldn't.