Date: 17th July 2024 **Hangman Solver**
Name: Leon Davis M S

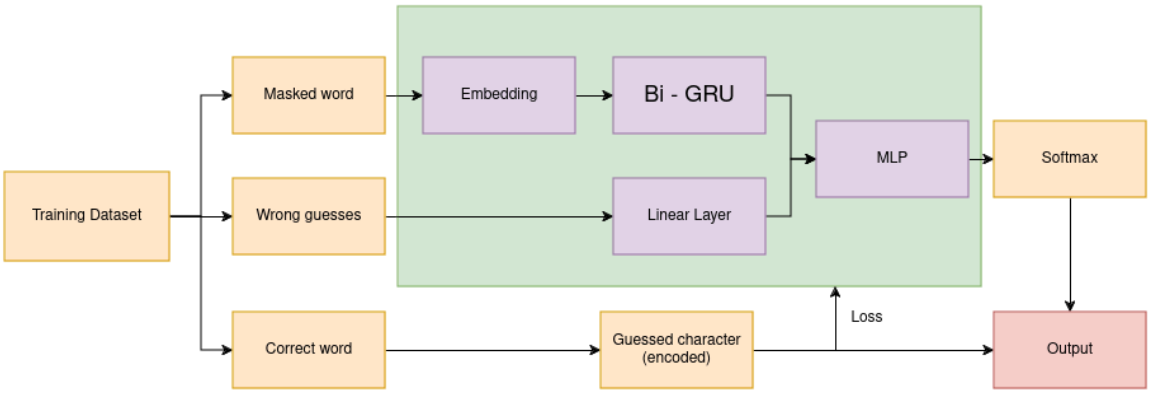
Accuracy achieved: 61.9%

There are two parts to the algorithm:
(i) **Bi-directional GRU**
(ii) **N - grams probability**

Both parts of the algorithm output a probability distribution for what the next letter is. These probabilities are added and the character with highest probability is picked next.

Part - I (Neural Network):

Architecture:



1) The GRU takes in an encoded, masked word to extract contextual information from the word and returns the hidden state at the last time step of the last layer.
2) Wrong guesses are encoded in a similar fashion and passed through a linear layer.
3) The GRU output and the high-dimensional representation of wrong guesses are then passed through an MLP (two Linear+ReLU layers), and finally output the probability distribution over the 26 letters

A previous implementation of this GRU model can be found here: https://github.com/methi1999/hangman?tab=readme-ov-file.

Modifications made:

1. For each training sample, we multiply the loss by a corresponding weight, assigning **higher weights** to the losses of **shorter words**. This is because shorter words are more difficult to guess, and we want to incentivize the model to learn from them more effectively. This change drastically improved performance.

2. As the number of training rounds increases, I gradually increase the probability of **dropping words** every epoch, to make training more challenging. Here, "**drop**" refers to the practice of selectively ignoring certain training samples during a particular epoch. This means that some samples are not used to update the model parameters in that training round.

   The idea behind **dropping samples** is to introduce **variability** and difficulty into the training process, which can help **reduce overfitting**. This is especially important as the model will be tested on a completely disjoint set of words.

Training method:

The given dataset was split into 90:10 train/test datasets. The model was trained for up to 100 epochs and the best model(tested on the 10% split of the dataset) was saved every 5 epochs. Different batch sizes and hidden layer dimensions were tested and optimised.

Best performing parameters:

1) **lr: 0.0005** #learning rate
2) **min_len: 3** #words with length less than min_len are not added to the dataset
3) **embedding_dim: 64** #dimension of embedding
4) **hidden_dim: 512** #hidden dimension of RNN
5) **output_mid_features: 128** #number of neurons in hidden layer after RNN
6) **miss_linear_dim: 128** #miss chars are projected to this dimension using a simple linear layer
7) **num_layers: 4** #number of layers in RNN
8) **dropout: 0.3** #dropout
9) **batch_size: 256** #batch size for training and testing
10) **epochs: 75** #total no. of epochs to train

The GRU model has a strong ability to capture the context of the word, but it may not be able to capture the statistical information of the word. Therefore, I tried to combine the GRU model with the n-gram model to improve the performance of the model.

Part - II (N-grams):

The n-gram is the simplest type of language model. It is typically used to predict the next word in a sentence. It works on the concept of conditional probabilities. An n-gram model looks (n-1) words into the past. One way to estimate probabilities is called maximum likelihood estimation or MLE. We get maximum likelihood estimation (the MLE estimate) for the parameters of an n-gram model by getting counts from a corpus, and normalising the counts so that they lie between 0 and 1. For example, to compute a particular bigram probability of a character w_n given a previous character

w_{n-1}, we'll compute the count of the bigram $C_{(wn-1,wn)}$ and normalise by the sum of all the bigrams that share the same first word w_{n-1}:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)}$$

We can add a **2-gram** if a word is in the form ["_x" or "x_"], **3-gram** if a word is of the form ["_xx", "x_x", or "xx_"] and so on.

We also consider the cases when word is in the form:

1) ["_ _ xx", "_ x _ x", "_ xx _", "x _ _ x", "x _ x_", "xx.."]
   (2 characters missing in 4 - letter word).
2) ["..xxx", ".x.xx", ".xx.x", ".xxx.", "x..xx", "x.x.x", "x.xx.", "xx..x", "xx.x.", "xxx.."]
   (2 characters missing in 5 - letter word)

Next, we can compute the probability of the letter appearing in the corresponding position based on the n-gram model. We use the number of occurrences of the n-gram as the probability estimate (after normalisation)**.**

I also created a character frequency table (characters are sorted in descending order based on frequency of being found in a word of a certain length) for every possible length of a word based on the given dataset. These characters are picked in order when the word is completely masked**.**

Finally, The probability distribution across the whole alphabet obtained from each model is added and the character with highest combined probability is picked next.

Conclusion:

The n-gram model offers a solid baseline by utilising letter frequency and co-occurrence statistics, while the GRU captures more complex patterns and long-range dependencies. By integrating these two approaches, we achieve optimal performance.