### **Hangman Solver Algorithm Documentation**

**Author**:  
**Date**: 24th October 2024  
**Developed by**: Aaditya Porwal  
**Accuracy Achieved**: 60.7%

I built this implementation using a combination of a **Bi-LSTM model** and an **n-gram probability approach** to efficiently solve the Hangman game.

### **Algorithm Overview**

This solution combines two main techniques:

1. **Bi-LSTM (Bidirectional Long Short-Term Memory)**:  
   I employed this recurrent neural network to capture long-term dependencies and predict the next character in the masked word.
2. **n-gram Model**:  
   I used this statistical method to compute letter probabilities based on conditional occurrences of character sequences (n = 1 to 5). This method helps capture common patterns in the word corpus and improves predictions.

Both models generate probability distributions for the next character, and these are combined to select the optimal guess.

### **Part I: Neural Network (Bi-LSTM)**

**Architecture**:

* **Embedding Layer**: Converts input words into a high-dimensional space for better word representation.
* **Bi-LSTM Layers**: Used 5 bidirectional LSTM layers with hidden dimensions of 64, 128, 256, 128, and 64. These layers ensure context is captured from both ends of the word sequence.
* **Regularization**: Applied L2 kernel regularization to prevent overfitting.
* **Dense Layer**: A softmax-activated fully connected layer outputs the probability distribution over 26 letters.

**Modifications**:

* Implemented **categorical cross-entropy loss** and used the **Adam optimizer** for better convergence.
* Removed dropout layers and used **L2 regularization** to improve generalization.
* I also introduced **dynamic learning rate adjustment** using the **ReduceLROnPlateau** scheduler and **Early Stopping**, optimizing training efficiency.

**Training**:

* Words are encoded to a maximum length of 29 characters.
* I split the data into 90% training and 10% testing, running for 11 epochs (out of 20) with early stopping to avoid overfitting.

**Key Parameters**:

* **Learning Rate**: 0.001
* **Batch Size**: 128
* **Epochs**: 20 (stopped at 11)
* **Embedding Dimension**: 128
* **LSTM Hidden Layers**: 5 (64, 128, 256, 128, 64)

### **Part II: n-gram Model**

**Overview**:  
I computed conditional probabilities of letters using the n-gram model, which considers sequences of up to 5 characters. This model is based on **maximum likelihood estimation** and uses a word frequency dictionary from the training corpus.

**n-gram Construction**:  
I built n-grams by counting occurrences of 1-gram to 5-gram sequences in the corpus. These are used to estimate the probability of the next character in a partially guessed word.

**Probability Calculation**:  
For each word, I calculated conditional probabilities based on character sequences. These probabilities are combined with the Bi-LSTM model's output using a weighted sum, where **75%** weight is assigned to the LSTM predictions and **25%** to the n-gram predictions.

### **Final Prediction Mechanism**

1. **Combining Probabilities**:  
   I combined the probability distributions generated by the Bi-LSTM and n-gram models with a **0.75 (LSTM) / 0.25 (n-gram)** ratio. This ensures the model captures both long-term dependencies and statistical patterns.
2. **Next Letter Guess**:  
   The model selects the character with the highest combined probability that hasn't been guessed yet.

### **Code Inspirations and Modifications**

**Inspiration**:  
My work was inspired by the [**DC769/DC\_Hangman\_Solution**](https://github.com/DC769/DC_Hangman_Solution), which provided the foundation for the Bi-LSTM network. The original repository employed:

* **Bi-LSTM**: To process and predict missing characters.
* **Neural Network Architecture**: Stacked Bi-LSTM layers for sequence modeling.
* **Softmax Output**: A softmax layer to produce letter probabilities.

**Modifications**:

* **n-gram Model Integration**: I introduced the n-gram model, combining long-term Bi-LSTM learning with statistical predictions to balance performance.
* **Weighted Combination**: Adjusted the weighted sum of probabilities for optimal results.
* **Increased Epochs**: The model was trained for **20 epochs** (stopped at 11) to ensure better performance.

### **Accuracy**

* **Initial Validation Accuracy**: 33% (from the original implementation).
* **Modified Validation Accuracy**: After incorporating the n-gram model and tuning hyperparameters (learning rate, batch size, epochs), the model's accuracy improved to **77%**.
* **Final Accuracy on Test Dataset**: 60.7%.

### **Conclusion**

In this Hangman Solver, I successfully combined deep learning (Bi-LSTM) and traditional statistical methods (n-gram) to create a robust solution for predicting missing characters. The Bi-LSTM model captures complex patterns, while the n-gram model provides statistical context. Together, these approaches enabled me to achieve more accurate and efficient results compared to previous implementations.