**Overview**

I designed and implemented a **Deep Learning** approach to solve the classic Hangman game. My objective was to significantly surpass a benchmark success rate of 18% and, more importantly, to exceed a 50% success rate threshold over 1,000 recorded games. With **PyTorch**, **Bi-LSTM** modeling, and an **attention** mechanism, my final solver achieves a consistent success rate of **60%+** on unseen words.

**Key Highlights**

1. **Deep Learning Model**: I used a **Bi-LSTM** architecture combined with an **attention** layer to accurately predict the next letter based on partially known word patterns.
2. **High Success Rate**: The solver records a **60%+** success rate (surpassing the 50% target) in official runs of 1,000 Hangman games.
3. **Generalization**: My model was trained on a dictionary of 250,000 words, and it generalizes well to an unseen 250,000-word test set.
4. **Production-Ready**: The solution integrates with the Hangman API, showcasing how deep learning can be applied in real, production-like scenarios.

**How It Works**

**1. Data Preparation**

* **Dictionary and Masking**  
  I was provided a 250,000-word dictionary for training. To simulate different stages of Hangman, I generated multiple “masked” versions of each word, where certain letters are replaced by \_. This approach let the model learn how to guess letters in various partial states of knowledge.
* **Additional Features**
  + **Vowel Ratio**: The fraction of known letters that are vowels (a, e, i, o, u) helps the model decide if it should guess vowels or consonants next.
  + **Word Length**: A normalized measure of the word length is fed into the model to differentiate between short and long words.

**2. Model Architecture**

1. **Embedding Layer**
   * Learns vector representations (size = 64) for each character (a–z plus \_ for masked).
2. **Bi-LSTM Layers**
   * A 4-layer **bidirectional** LSTM with hidden size = 128 captures both forward and backward context around partial word sequences.
3. **Attention Mechanism**
   * Computes attention weights over the Bi-LSTM outputs, highlighting the parts of the sequence that matter most for predicting missing letters.
4. **Fully Connected Output**
   * I concatenate the attention-based context vector with **vowel ratio** and **word length** before passing it through a linear layer.
   * The final output is a 26-dimensional probability vector (for each letter a–z).
5. **Sigmoid Activation**
   * Each of the 26 letters has a probability “score.” During gameplay, I select the letter with the highest probability that hasn’t been guessed yet.

**3. Training Process**

* **Dataset Construction**  
  I created a HangmanDataset that takes each word, randomly masks subsets of its letters, and records which letters appear in the original word.
* **Loss Function**  
  I used **BCELoss**, as the model’s output is effectively a multi-label classification across 26 letters.
* **Optimizer**  
  I used **AdamW** with a scheduler to decrease the learning rate every 10 epochs.
* **Epochs**  
  I trained the model for 30 epochs with a batch size of 256, at which point the loss converged around 0.22, reflecting strong performance.

**4. Guessing Logic During Gameplay**

1. **Mask the Input**  
   I convert the partial game state (underscores plus revealed letters) into the same indices used in training.
2. **Extract Features**:
   * **Vowel Ratio** of known letters
   * **Normalized Word Length**
3. **Run the Model**  
   My solver forwards these inputs through the Bi-LSTM + attention model, producing a probability for each letter.
4. **Make a Guess**
   * Exclude letters already guessed
   * If the vowel ratio is high, the solver may prioritize consonants (a small heuristic on top of the model)
   * Select the highest-probability letter from the model’s output distribution

**5. Final Results**

* After running **1,000 recorded games** on the official Hangman API (with no further modifications allowed), my solver achieved:
  + **> 60% success rate** (well above the required 50%).

**Main Python Files**

* **HangmanModel**
  + Defines the Bi-LSTM layers, attention mechanism, and output layer.
* **HangmanDataset**
  + Dynamically masks words and prepares training samples.
* **train\_model()**
  + Trains the model and saves the final checkpoint.
* **HangmanAPI**
  + Contains my guess function and handles server communication for Hangman.

**Skills Demonstrated**

1. **Deep Learning Design**: I created a custom **Bi-LSTM** + **attention** architecture in PyTorch.
2. **Sequence Modeling**: Handled partial word contexts (underscores) and relevant side features (vowel ratio, length).
3. **Data Engineering**: Generated realistic Hangman states by masking words in multiple ways.
4. **Python Proficiency**: Demonstrated usage of PyTorch, data loading, concurrency, and more.
5. **Constraint-Based Problem Solving**: Adhered to using only the provided dictionary and improved upon a baseline approach.

**Next Steps and Improvements**

* **Enhanced Feature Engineering**: Possibly add letter bigrams or sub-word embeddings to handle unusual letter combinations.
* **Adaptive Guessing**: Incorporate reinforcement learning to optimize guesses directly for improving win rates.