High-Performance Hangman AI

This repository contains my solution for a Hangman AI challenge. The goal was to develop a machine learning-based algorithm to play the classic word-guessing game, where a player tries to figure out a hidden word by guessing letters one at a time within a limited number of incorrect attempts. My final agent significantly outperforms a provided 18% win-rate benchmark, achieving a **67.4% win rate** over 1,000 test games.

My Approach: An End-to-End Deep Learning Solution

I approached this problem as a sequence-to-sequence task. The model takes a partially revealed word (the Hangman pattern) and predicts the complete word. I broke down the project into three core phases: creating a high-quality dataset, building a powerful neural network architecture, and refining the model with specialized training.

Phase 1: Data Generation - The Foundation

The project's success hinges on the quality of the training data. The challenge states that the test words are *not* in the training dictionary, so my model needed to learn the general rules of English spelling, not just memorize a vocabulary list.

To achieve this, I developed a script (`generate_data.py`) that uses a **Masked Language Modeling** approach. Here's how it works:

1. **Iterate Systematically:** The script iterates through every single one of the ~250,000 words in the provided dictionary.
2. **Generate Game-Accurate Masks:** For each word, it generates 80 unique, random patterns that mimic real gameplay. This is a crucial detail: it randomly selects a subset of the word's unique letters to reveal, and then reveals *all* instances of those letters. This prevents the model from being trained on "impossible" patterns (like `a_ple` for `apple`).
3. **Create a Balanced, Massive Dataset:** This process ensures that every word in the dictionary, regardless of its length or complexity, contributes an equal number of training samples. The final result is a massive, high-quality, and well-balanced dataset of approximately 20 million `(masked_pattern, original_word)` pairs.

Phase 2: Model Architecture - Building the Brain

To learn from this massive dataset, I designed a deep, sequential model architecture in PyTorch (`train_model.py`).

The architecture consists of:

1. **Embedding Layer:** Converts the input characters (`a`, `b`, `_`, etc.) into dense numerical vectors.
2. **1D Convolutional (CNN) Layer:** This layer acts as a "pattern spotter." It slides over the sequence of character vectors and learns to recognize small, local patterns and common letter combinations (n-grams like `ing`, `th`, `_es`).
3. **Stacked BiLSTM Layers:** This is the core of the model. It takes the features identified by the CNN and analyzes them as a sequence. By reading the sequence both forwards and backwards, the two stacked

BiLSTM layers learn the long-range dependencies and grammatical structure of words.
4. **Deep Dense Network:** The final fully-connected layers take the rich output from the LSTMs and make a prediction for the most likely character at every single position in the word. Dropout layers are used here to prevent overfitting.

Phase 3: Training & Fine-Tuning

With the data and architecture in place, the final step was to train the model effectively.

1. **Initial Training:** I first trained the model on the entire ~20 million sample dataset. This gave the model a strong, general understanding of the language and achieved a solid baseline performance.
2. **Specialized Fine-Tuning:** Through testing, I identified that the model's biggest weaknesses were in the "opening" (1-2 letters revealed) and the "endgame" (1-2 blanks remaining). To address this, I created a specialized fine-tuning script (`finetune_model.py`) that:
    - Filters the main dataset to create a new, smaller dataset composed of these difficult early-game and endgame scenarios, balanced with a large sample of mid-game patterns.
    - Loads the already-trained model.
    - Continues training for a few more epochs on this specialized data with a lower learning rate. This is like making an athlete run drills that specifically target their weaknesses, and it resulted in a significant performance boost.

Phase 4: The Guessing Strategy

The final piece was the `guess` function in the main notebook. Instead of a simple "greedy" approach, I implemented a more robust strategy. At each turn, the model predicts the probability of every letter for every blank spot. My function then **aggregates these probabilities**, summing up the confidence scores for each potential letter across all available blanks. The letter with the highest total score becomes the guess. This "consensus" approach proved to be much more reliable and was the key to pushing the win rate from ~55% to ~69% during practice, leading the the final win-rate of **67.4%**.

This project was completed as a part of the Trexquant AI Challenge. Link to the repo