Trexquant Hangman Challenge Project Report

Xiaofei Teng

Overview

The problem is to build an algorithm that solves the Hangman problem within 6 wrong guesses. At each step, the guess() function takes in a partial state, where masked letters are denoted as “_” and return a letter from a-z as the next guess.

My approach to this question is to build an ensembled system of a guesser model, which is trained through traditional machine learning approach, and very simple heuristics that are manually forced in the guess() function. The guesser is a transformer, and the forced heuristics are: 1. When the word is fully masked, guess from the conditional probability on letter frequency given the word length (this information is attainable at the first guess), and 2. Never guess repeating letters.

Model Selection

There were 3 candidate models to be selected from: LSTM, transformer, BERT. After careful research, transformer was selected for the following reasons:

First, LSTM is a special version of RNN, which naturally incorporate some kind of “ordering” of the state, but this “ordering” does not meet with the requirement of this game. For example, if there are multiple valid guesses, there’s no clear way to define an ordering within the according states, or at least not necessary to do so. On the other hand, transformer has self-attention which allows it to learn from all valid guesses.

Second, although BERT seems to be perfectly suitable for this task because it is trained from MLM tasks, which are in nature very close to the Hangman game, itself, the scale of BERT makes it unnecessary for this project. BERT has 12 layers of transformer encoders and is trained to predict a word from a total vocabulary of 30,000 words within a sentence, which might have dozens of words in it. However, the Hangman problem only requires predicting from a vocabulary of 26 letters with a word than is most likely to consist of less than 10 unique letters.

While there’s no doubt that with careful, task-specific fine tuning, a pre-trained BERT can easily pass the 55% win rate threshold, it’s too complex and computational heavy, and most importantly, excessive for the task. Thus, I decided to train a smaller transformer with only encoders (similar structure to BERT) from scratch. This model should be able to take in any “state” of a game as input, and predict the most likely next guess from the 26 available options.

Data Preparation

The data provided is a list of 250K English words, which I cannot simply feed to the model. Thus, I generate my own training data from that list by randomly simulate an incomplete state of the word, with "_" as mask, and all unrevealed letters as labels. For example, the word "apple" could be in the state where the masked_word is a _ _ _ e and the label is {p, l}. I generate k different samples from each word where k is the number of unique letters the word has. This gave me a training set with input and label to perform a supervised learning. The original dataset has about 1.6M samples.

For easy batch process, I pad all the samples to fix length, 20, and encode all the letters, mask, padding into integers ready for training.

Training and Fine-Tuning

First, I treated this problem as a multi-classification problem and used CrossEntropyLoss for evaluation to train a 2-encoder-layer transformer. The result is 31% win rate on 100 trials. It is a leap from the 18% base line, but the model is still performing poorly.

The problem with my first approach is that CrossEntropyLoss punishes the model with correct but "none-optimal" guess (argmax of logit), while in real game scenario there are usually multiple valid guesses at a given game state. Instead of multi-classification with CrossEntropyLoss, a multi-label BCEWithLogitsLoss (which basically compare the logits with a multi-hot vector of valid guesses) clearly suit the problem better. Also, I decided the model trains and converts too fast and introduced an extra 2 layers of encoder to the architecture.

The 4-layer model trained with BCEWithLogitsLoss achieved 42% win rate, but shows significantly better performances with longer words (which usually contains more unique characters). Thus, I re-build the training data, which now simulates more states per word with more even distribution across different k's. This larger data ends up to have 2.7M sample. I trained the same model on this new data and achieved 58% win rate with it.

At this stage, the baseline requirement of the project is already met, but I notice another problem: the model sometimes struggle with end-game states where there's only 1 letter left to be revealed. Because I still have time, I planned to fine-tune my model and slightly biased it towards end-game states to achieve better overall performance.

Before I did the fine-tuning, I trained a model with 6 layers to ensure better model capacity and build a dataset of 1.6M samples which are all 1-letter-short to completeness.

Result

The 6-layer model achieved 58.7% win rate across 1000 trials. The fine-tuned version with lr=3e-5 and trained 10 epochs achieved 56.7% win rate across 1000 trials. (This is probably because the model is biased to end-game states so much that influences its early game capacity). The fine-tuned version with lr=1e-5 and trained 5 epochs achieved 61.8% win rate across 1000 trials and is used in the final official test. The final official test result is 61.0% win rate across 1000 trials.

Reflection

This project is both fun and challenging. While 61% overall win rate is solid, there's certainly ways to push that further. With more time and computational resource, I might choose to fine-tune BERT or incorporate a RL network to teach the model about strategies instead of manually forcing some simple heuristics. Either way, I believe the final outcome could easily exceed 65% overall win rate.