Playing Hangman with Large Language Models

Manurag Khullar, Vaibhav Sahu
TA Mentor: Yunshuang Li

May 2024

1 Abstract

Hangman is a classic word-guessing game where one player thinks of a word and the other tries to guess it by suggesting letters within a certain number of guesses. In this project, we have trained Google's CANINE transformer to play Hangman, aiming to teach the model strategic letter guessing based on partial word clues. Our approach involved modeling the game's mechanics and states, allowing the AI to navigate various scenarios effectively. The results were promising, with the model achieving 63% success in guessing words correctly within the standard number of tries and further improvements were noted when additional tries were allowed. This demonstrates the potential of using sophisticated language models in simple, challenging cognitive tasks.

2 Introduction

Hangman's objective is to correctly guess a hidden word by suggesting letters within a limited number of attempts. Despite its apparent simplicity, Hangman presents a significant cognitive challenge, requiring strategic guessing based on partial information. This project aims to leverage modern natural language processing (NLP) techniques, specifically Google's CANINE transformer model, to develop an AI capable of playing Hangman. The importance of this problem lies in its broader implications for NLP and machine learning. Developing an AI that can effectively play Hangman showcases the potential of tokenization-free models in handling character-level text, which is essential for languages and applications where traditional tokenization methods fall short. Additionally, this project highlights the versatility of advanced language models in adapting to various tasks, including games that involve strategic decision-making and pattern recognition.

In our proposed model, the primary inputs and outputs are defined as follows:

- **Inputs**
  - **Word to be guessed** (W): A hidden word represented as a sequence of characters $W = \{w_1, w_2, ..., w_n\}$.
  - **Guessed letters** (G): A set of letters that have been guessed so far.
  - **Game state** (S): A representation of the current state of the word, showing correct guesses and placeholders for remaining letters.

- **Outputs**
  - **Next letter guess** (L_g): The predicted letter the model selects as the next guess.
  - **Updated game state** (S'): The game state after incorporating the new guess, reflecting any correct guesses and remaining placeholders.

3 Background

Several AI models for Hangman have been developed, each with its approach and limitations.

1. We used intuition inspired by Microsoft Azure's training for Hangman players, which can be found here[1]. It helped us with modeling the ML task. However, their project also gave us some good insights into improving our model performance. We plan to enhance our performance by further fine-tuning our model.
   Limitation: Their approach is nice; however, they try to train the model from self-play from the start. This is not optimal as the model takes more time to learn, initially being a weak player.

2. B. Clark, Jonathan H., et al., "Canine: Pre-training an Efficient Tokenization- Free Encoder for Language Representation." [2] This work on the CANINE-S transformer model provides a foundation for understanding how transformers can be adapted for character-level text processing and will serve as a technical reference for our model development.
   Limitation: This is more about generalized Language modeling. We need to tweak it before using it in our Hangman game player.

4 Summary of Our Contributions

We have thought of a pre-training task that would work great with a character-level LLM. We can effectively divide our project into the following tasks and contributions:

1. Analyzing the complexity of the problem and conducting an exploratory analysis to identify effective strategies. Ultimately, we aim to maintain an end-to-end, strictly ML-based approach. However, identifying strategies will aid in debugging our model's performance and assessing its ability to learn and replicate them.

2. Pre-training the model: Our plan involves simulating hangman games in a Monte Carlo-based fashion and encoding them into text to feed into our language model. This serves as the pre-training task we have devised.

3. Self-play fine-tuning: Our model benefits greatly from learning to play itself when it has figured out the initial distribution. We choose a smaller dataset to train the model for this. It differs from Microsoft's approach of training through self-play from the start.

4. Lastly, we aim to enhance our model's performance and analyze its errors and shortcomings. This analysis will enable us to derive insights from the project, outline its contributions, and determine future avenues for improvement, if necessary.

5 Detailed Description of Contributions

1. **Set up the problem**: Hangman can be considered a Markovian Process. Given this conditional time-series-like dependency, we thought sequence modeling could be done using RNNs and Transformers. Effectively, we model the problem as follows:

   Our actions are decisions that our model takes, given the state at a given time step. Hangman states can be completely defined by keeping track of all the guessed letters and the state of our word. For example, consider the example of "birthday." Say we guessed the letters "e," "i," "a," and "s." The state of our word becomes "_ i _ _ _ _ a _." Optimally, our model has to learn to condition its guess based on the updated state at each time step. Notice that the action is a letter guess. We only have 26 letters, so our action can be modeled as a classification task that predicts the correct letter or the "class" out of 26 letters ("classes").

2. **Proposed state**: We can text-encode all the states. Essentially, we could feature design this and pass encoded vectors for the letters that have already been guessed and traverse the game state as a sequence of one-hot vectors denoting letters. However, we know transformers and LLMs are great at learning from text-encoded information. To ensure our transformer learns this mapping, we will discuss some ways to ensure this. For now, the state information is stored in the following way:

   ```
   [CLS] <GUESSED_LETTERS> [SEP] <HANGMAN_STATE> [SEP]
   ```

   The `HANGMAN_STATE` is itself encoded neatly. For example, the hidden letters missing in our word are modeled using `[MASK]` tokens. In this way, our problem is ready to be used as a pre-training task for our Language Model.

   ```
   [CLS] eias [SEP] [MASK] i [MASK] [MASK] [MASK] [MASK] a [MASK] [SEP]
   ```

3. **Self-play finetuning** As the name suggests, we make the model learn from its mistakes. It plays the game 100k times on different words in 10 iterations. For each iteration, we sample 10k words, and the model iteratively corrects itself through gradient-descent

5.1 Method

Machine Learning Task: The objective of our agent is to minimize the number of tries it uses. However, how do we model the loss? There are 2 ways to approach this problem.

A. Multilabel classification task: We can model all the disguised letters that have not been guessed as the correct labels. The label would then be just the encoded vector with 1's on all the correct letters that have not been guessed. This gives the model more slack to work with. It can work well if we train the model for longer. However, given our limited resources, I realized that the more restrictive approach had a smaller search space and gave results faster.

B. Multiclass classification task: This is more restrictive. We predict the most frequent missing letter. The probability of letters can be modeled based on their frequency in our word. We keep track of the frequencies and then normalize them to get the probabilities. This 26-dimensional vector is then used as the ground truth for our model. This way, it learns to predict the most likely letter.

We used the Multiclass classification task as this is more restrictive, has a smaller search space, and needs less training. The loss function is simply the cross-entropy loss between this normalized probability vector and the softmax output from our model,

$$f(s_t, i) = \frac{\sum_j \delta_j \cdot I_{x_j = x_i}}{\sum_i \sum_j \delta_j \cdot I_{x_j = x_i}}$$

where x_i is the i-th letter of the alphabet, and x_j is the letter at the j-th position of the word if it has not been guessed already. The loss function is given by:

$$L(f(s_t), o_{LM}) = \sum_i -f(s_t, i) \log(o_{LM}(i))$$

where o_{LM} is the 26-dimensional softmax output of our classification model. Our model effectively learns to model the conditional distribution of letters.

Model Selection: It's time to select the best model for our ML task. We need a large enough model to learn this end-to-end mapping we have designed. However, it is interesting to note that the token-level structure in our inputs is characters. Therefore, we chose a character-level model. As we stated earlier, we also found a great model called CANINE. Google's CANINE model can learn language-level concepts even though it is a character-level model. It is motivated to make it tokenization agnostic. However, we benefit from its character-level featurization, and the scale is not too huge to pre-train it from scratch (121 Million parameters).

5.2 Experiments and Results

5.2.1 Exploratory Data Analysis

On the entire dataset, we analyze the distribution of words by their length and also the natural calling order of the dataset. It resembles a Poisson distribution with a mode length of about 9 letters. Here is the distribution of word length:

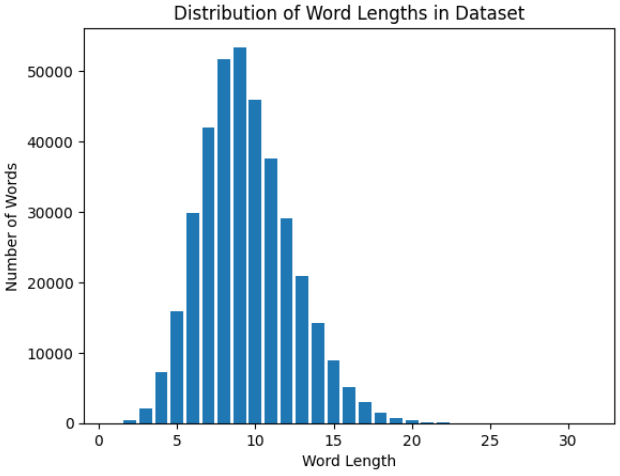


Figure 1: Word length distribution

```
Length 4: ['a', 'e', 'o', 'i', 'u', 'y', 's', 'r', 'd', 't']
Length 5: ['a', 'e', 'o', 'i', 'u', 'y']
Length 6: ['e', 'a', 'o', 'i', 'u', 'y']
Length 7: ['e', 'a', 'i', 'o', 'u']
Length 8: ['e', 'a', 'i', 'o', 'u']
Length 9: ['e', 'i', 'a', 'o']
Length 10: ['e', 'i', 'o', 'a']
Length 11: ['e', 'i', 'o', 'a']
Length 12: ['e', 'i', 'o', 'a']
Length 13: ['e', 'i', 'o', 'a']
Length 14: ['i', 'e', 'o']
Length 15: ['i', 'e', 'o']
```

Figure 2: Calling Order of our Dataset

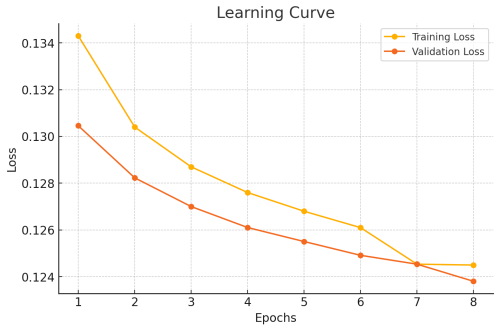
The calling order is the order of letters to call without any correct guesses in Hangman. We can obtain the natural calling order by conditionally calculating the most probable letter and removing words containing that letter. Iteratively doing this, we can obtain the final calling order.

Finally, we split the dataset for our training. As this

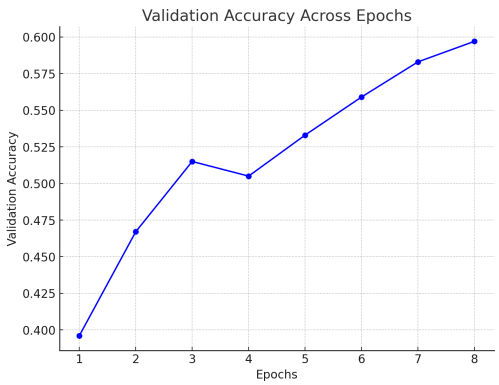
Our dataset has 400k words. For each testing and validation, we take 10k words and use the rest to train our model.

5.3 Pre-training

The pre-training task uses 380k words to train the model. We were able to achieve 59% validation accuracy. Here is the learning curve and the accuracy plot of training our model on the task:



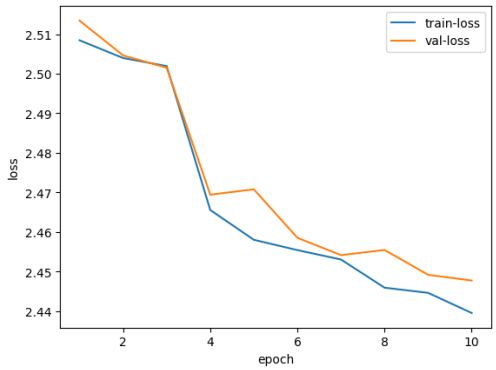
(a) Learning Curve



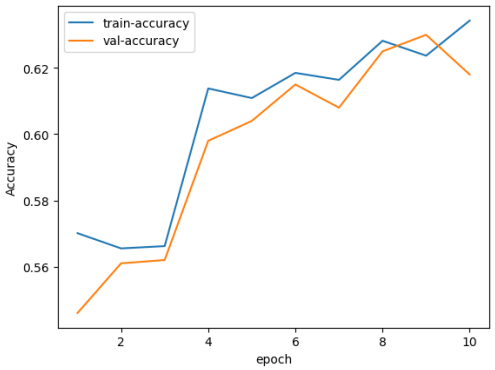
(b) Validation accuracy plot for pre-training

5.4 Self-play Finetuning

For self-play fine-tuning, we got a validation accuracy of 63.5%! The model could be tuned further with better hyperparameters. However, even with the current model settings, we also get an accuracy of 63% on our test set.



(a) Learning Curve



(b) Accuracy plot for self-play

To gain further insight into a model performance, we also plot the accuracy of our model with word length:

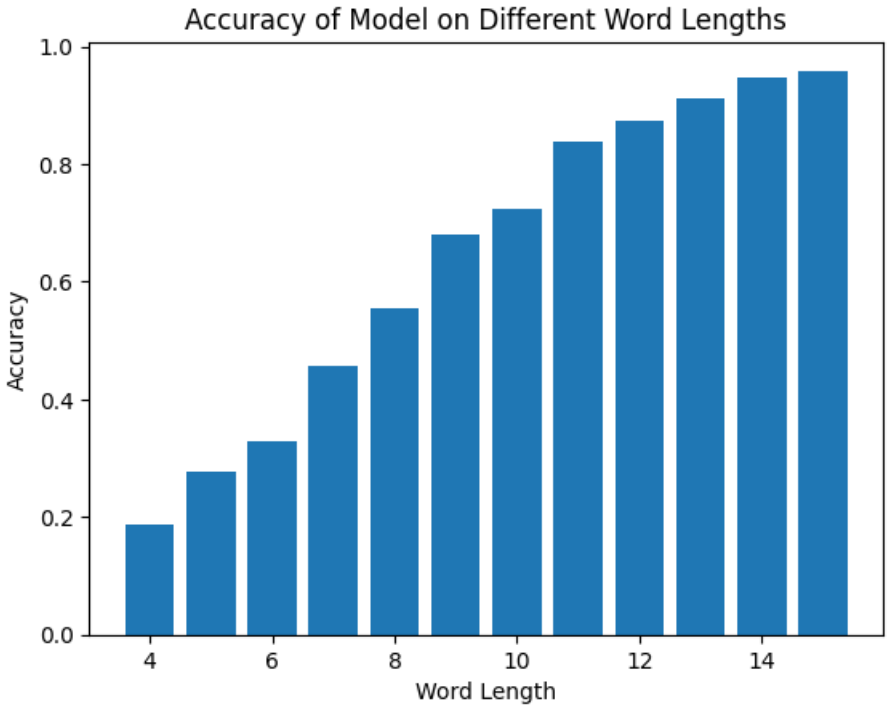


Figure 5: Accuracy vs word length

6 Compute and other resources used

We used high-performance computing resources with GPUs to train our model using deep learning algorithms. Given the size of our model and dataset, pre-training was expensive and took 12 hours on an A100 GPU.

7 Conclusion

8 Appendix

8.1 GANNT Chart for Timeline

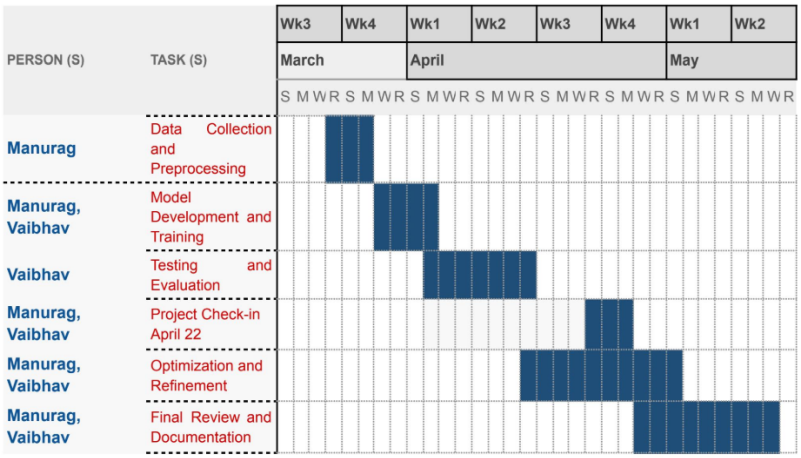


Figure 6: Timeline

8.2 Project Check-in

References

[1] Microsoft Azure. Hangman. `https://github.com/Azure/Hangman/tree/master`, 2024. Accessed: 2024-05-15.

[2] Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. Canine: Pre-training an efficient tokenization-free encoder for language representation. *arXiv preprint arXiv:2103.06874*, 2021. TACL Final Version.