Transformer-Based AI Agent for Playing Hangman

Abstract

This work presents a novel formulation of the classic Hangman game as a sequential decision-making problem. Transformer-based AI agent is designed to play the game of Hangman. The agent is trained to predict the conditional probability distribution of the next character guess given a partially masked word (the current state of the game) and the history of previously guessed letters while minimizing duplicate guesses and optimizing guess efficiency. This document details the dataset formulation, the model's training procedure, and the mathematical foundation underpinning the loss design and conditional prediction structure.

1 Introduction

The Hangman game serves as an interesting sequential reasoning problem: given partial information about a word, a player must guess missing characters based on a history of previous guesses. We adapt this to a machine learning setup, aiming to train a model that learns to predict a sequence of character guesses to reconstruct the hidden word.

2 Dataset Design

2.1 Vocabulary and Input Encoding

The character vocabulary consists of 26 lowercase English alphabets augmented with three special tokens: `<SOS>` (start of sequence), `<MASK>` (hidden character), and `<PAD>` (padding). The vocabulary is:

$$\mathcal{V} = \{a, b, \ldots, z, \texttt{<SOS>}, \texttt{<MASK>}, \texttt{<PAD>}\}$$

Each word in the training set is padded to a maximum word length of 35 characters. For each word, a one-hot vector is generated representing the presence of each character:

$$y_i = \nVdash_{c \in w_i}, \quad y_i \in \{0, 1\}^{26}$$

The encoder input starts as all `<MASK>` tokens, and the decoder receives the previous guesses, beginning with `<SOS>`.

2.2 Game State Representation

The model is trained using a custom PyTorch Dataset, `HangmanDataset`. For each word, a state object is constructed containing:

- Masked encoder input: the current reconstruction of the word.
- Guess history: characters guessed so far.
- Binary masks for valid inputs.
- Target one-hot vector for training supervision.
- Character frequency vector.

3 Model Overview

The model (`HangmanTransformer`) follows a standard Transformer architecture, integrates the encoder and decoder:

- Input `src`: Masked representation of the word to guess (e.g., _ A _ _ M A _)
- Input `tgt`: Sequence of previous guesses
- Output: Probability distribution over the 26 alphabets for the next guess

Let the input word be $w \in \mathcal{V}^{\leq 35}$, and let h_t denote the history of guesses up to time t. The model estimates:

$$P(c_t \mid h_{<t}, \hat{w}_{<t})$$

where c_t is the next guessed character, $h_{<t}$ is the guess history, and $\hat{w}_{<t}$ is the partially revealed word. Formally, at each time step t, the model predicts:

$$\hat{p}_t = \text{softmax}(f_\theta(\hat{w}_{<t}, h_{<t}))$$

where f_θ is the model function parameterized by θ.

3.1 Transformer Encoder

The encoder transforms the masked word input into contextual embeddings using self-attention and positional encoding. A learned embedding layer converts input tokens (masked characters) into dense vectors of fixed dimension d_{model}.

$$\text{Embedding} : \mathbb{Z}^L \to \mathbb{R}^{L \times d_{\text{model}}}$$

To preserve the order of input characters, sinusoidal positional encoding is added to the token embeddings:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

3.2 Transformer Decoder

The decoder generates the next guess character based on previously guessed characters and the encoder output. The decoder input (sequence of past guesses) is similarly embedded and enhanced with positional encoding. To prevent the model from attending to future guesses, a triangular mask is applied:

$$\text{Mask}_{i,j} = \begin{cases} 0, & \text{if } j \leq i \\ -\infty, & \text{otherwise} \end{cases}$$

Each decoder layer attends to both the guess history and the encoder output (masked word), generating a contextual representation. The final decoder output is projected to a vocabulary of 26 English alphabets using a linear layer.

4 Loss Function

The loss consists of two components:

1. **Binary Cross Entropy (BCE) loss** between the predicted logits and the true character presence vector:

$$\mathcal{L}_{\text{BCE}}^{(t)} = -\sum_{i=1}^{26} \left[y_i \log \sigma(z_i^{(t)}) + (1 - y_i) \log(1 - \sigma(z_i^{(t)})) \right]$$

   where $z_i^{(t)}$ is the predicted logit for character i at guess t.

2. **Duplicate penalty loss** to discourage repeated guesses:

$$\mathcal{L}_{\text{dup}}^{(t)} = \sum_{i=1}^{26} \hat{p}_{t,i} \cdot \nVdash_{i \in h_{<t}}$$

The **combined loss per** step is with an exponentially decaying weight:

$$\mathcal{L}^{(t)} = \nVdash_{\text{word not guessed}} \cdot \left(\mathcal{L}_{\text{BCE}}^{(t)} + \lambda \cdot \mathcal{L}_{\text{dup}}^{(t)} \right)$$

$$w_t = \alpha^t w_0, \quad \alpha \in (0, 1)$$

The **total loss** across all guesses for a batch is:

$$\mathcal{L}_{\text{total}} = \sum_{t=1}^{T} w_t \cdot \mathcal{L}^{(t)}$$

5 Training Procedure

5.1 Word Sampling and Splitting

Words are read from a cleaned file and grouped by length to ensure balanced training. Each length-specific group is split 70-30 into training and testing sets. Batches are created using a `DataLoader` with shuffling and multiprocessing support. The batch size is 32.

5.2 Training Loop

Training is performed over 50 epochs. At each time step t in the maximum allowed guesses (26), the model predicts a character distribution. The training steps include:

- Forward pass to obtain logits.
- Softmax to obtain predicted character probabilities.
- Compute the BCE loss and duplicate loss.
- Weight the loss based on step number.
- Backpropagation and optimizer update.
- Update the guessed word state and terminate early if the word is completely guessed.

5.3 Training Setup

- Epochs: 50
- Max guesses: 26
- Batch size: 32
- Weight decay factor: $\alpha = 0.8$
- Initial step weight: $w_0 = 8.0$
- Duplicate penalty coefficient: $\lambda = 10$

6 Fine-Tuning The (GOOGLE) CANINE Model

To provide a strong learned baseline for comparison, we fine-tuned Google's CANINE model, a character-level Transformer pretrained using masked language modeling objectives. Unlike our model, which is trained from scratch with explicit access to guess history via a decoder, CANINE operates solely on masked input without any notion of sequential guesses. We adapted the CANINE architecture for the Hangman task by formulating it as a character prediction problem over randomly masked words, without access to previous guesses. Although CANINE benefits from large-scale pretraining and a deeper encoder (12 layers), it lacks explicit reasoning over guess history, which is critical for efficient gameplay. To compensate for this, we performed hard masking of the previously guessed characters in the output layer to ensure that the model does not repeat guesses during training and inference.

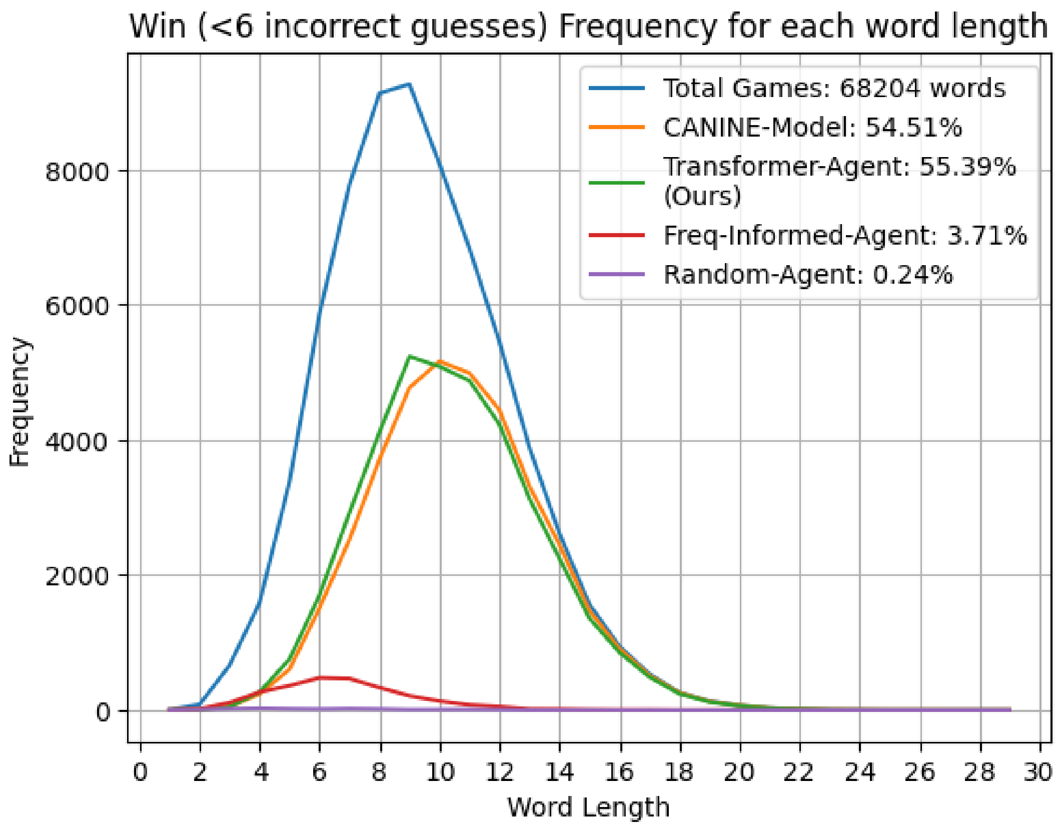
The training corpus was created by generating all possible masking permutations of words, resulting in a large dataset. Due to computational constraints, only a subset (approximately 10%) of the full corpus was used for fine-tuning the CANINE agent.

7 Results

Table below compares various text-based agents on hangman task in terms of their win rates, inference speed, model complexity, use of guess history, and pretraining status. The Random-Agent and Frequency-Informed-Agent do not leverage any guess history and serve as non-learned baselines, while the CanineMode and our Transformer-Agent are learned models. The Transformer-Agent notably achieves a slightly higher win rate than CanineMode despite being significantly smaller in terms of parameters, due to its ability to incorporate guess history via a decoder.

Agent	**Test-Win%**	**Inference**	**Params**	**Updates**	**Guess-History**	**Pretrained**
Random	0.2	**Fastest**	NA	NA	No (masked)	NA
Freq-Informed	3.7	Slowest	NA	NA	No (masked)	NA
CANINE (Google)	54.5	Medium	~121M (12L, enc)	~5.3M	No (masked)	Yes
Transformer (Ours)	**55.4**	Fast	**~1.2M** (2L enc + 2L dec)	**~2.4M**	**Yes** (decoder)	No

Table 1: Performance and characteristics of different Hangman agents. 'masked' in Guess-History means that the model is unaware of past and output probabilites of the past guesses are hard masked to avoid duplicate predictions. 'Updates' refer to the total number of weight updates (batches seen each of size 32) during training.



The chart compares Hangman agents based on how often they win (fewer than 6 incorrect guesses) across word lengths. Transformer-based our method and CANINE models outperform heuristic agents (Random, Frequency-Informed), especially for medium-length words (6–12 characters), where context is rich and guess attempts are sufficient. Our model shows the highest win rate, with CANINE closely following. Heuristic agents struggle with both short (high ambiguity) and long words (high variability). The blue line represents the total test words per length, serving as an upper bound.

8 Conclusion

This work demonstrates the application of Transformer-based sequence modeling to the classic word-guessing game of Hangman. By formulating the problem as a conditional character prediction task, we leverage the power of self-attention to effectively model the masked word structure and the history of past guesses. The agent is trained to infer the next optimal character by minimizing uncertainty and avoiding repetition, thereby simulating intelligent gameplay behavior. The modular design of the encoder-decoder framework enables flexible experimentation, and the formulation allows for further extensions such as reinforcement learning, curriculum learning, or incorporation of linguistic priors. Overall, this study not only validates the effectiveness of Transformers in structured decision-making tasks but also opens avenues for applying such models in other sequential reasoning or language-based games.