Trexquant Hangman Challenge : A Combined Transformer-Based & Heuristic Strategy Approach

Srijan Anand, IIT Kanpur

December 20, 2024

Abstract

This document outlines the development of an intelligent Hangman solver that uses Transformer-based Masked Language Models (MLMs) combined with heuristic strategies. The approach aims to improve the accuracy of letter prediction by integrating deep learning models with statistical heuristics. The initial training achieved a win rate of approximately 60%, which slightly decreased to 56.8% during the final execution due to some silly weight adjustments. Despite computational constraints, the strategy shows promising potential for further improvement with more robust models.

Contents

1 Thought Process

Developing an effective Hangman solver required a blend of advanced machine learning techniques and traditional heuristic methods. Initially, I experimented with a single Transformer-based MLM but observed that it struggled to handle the diverse range of word lengths and complexities encountered during practice games. Recognizing this limitation, I adopted an approach by training multiple models tailored to different word characteristics:

- **High, Medium, and Low Models:** These models were designed to handle words with varying levels of complexity and letter distributions.
- **Short Model:** Specifically trained to excel at predicting letters in shorter words, addressing the shortcomings observed with the initial models.

The heuristic approach, inspired by insights from a GitHub repository (by Shashank Kartikey), provided a statistical foundation by analyzing letter frequencies and word patterns. However, to achieve higher accuracy and adaptability, integrating these heuristics with the strengths of MLMs became essential. This synergy aimed to utilize the contextual understanding of MLMs while refining predictions through empirical statistical insights.

2 Models and Approaches Used

2.1 Transformer-Based Masked Language Models (MLMs)

Transformer-based MLMs are pivotal in natural language processing, excelling at understanding context and predicting missing or masked tokens within a sequence. For the Hangman solver, MLMs predict probable letters in unknown word positions based on the current state of the game.

2.1.1 Model Architecture and Parameters

Each MLM is architected with specific parameters to cater to different word characteristics:

- **Vocabulary Size:** 28 (including a special mask token `[MASK]` and the underscore _ for unknown characters).
- **Embedding Dimension (d_{model}):** 128
- **Number of Heads (n_{head}):** 4
- **Number of Layers (num_{layers}):** 6
- **Feedforward Dimension ($dim_{feedforward}$):** 512
- **Maximum Sequence Length (max_{len}):** 20 for high, medium, and low models; 10 for the short model.

```
class TransformerMLM(nn.Module):
    def __init__(self, vocab_size=28, d_model=128, nhead=4, num_layers=6,
                 dim_feedforward=512, max_len=20):
        super(TransformerMLM,self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_enc = SinusoidalPositionalEncoding(d_model, max_len)
        encoder_layer = nn.TransformerEncoderLayer(d_model, nhead,
                                                   dim_feedforward,
    batch_first=True)
        self.transformer = nn.TransformerEncoder(encoder_layer,
    num_layers)
        self.fc = nn.Linear(d_model, vocab_size)

    def forward(self, input_ids):
        x = self.embedding(input_ids)
        x = self.pos_enc(x)
        x = self.transformer(x)
        logits = self.fc(x)
        return logits
```

Listing 1: Transformer-Based Masked Language Model Implementation

2.1.2 Heuristic Approach

The heuristic strategy complements the MLMs by analyzing letter frequencies and word patterns to refine predictions. This approach involves:

- **Letter Frequency Analysis:** Counting the occurrences of each letter in potential word candidates to prioritize more common letters.
- **Pattern Matching:** Filtering the dictionary of words based on known letters and their positions, narrowing down possible candidates.

These heuristics were empirically defined by closely observing verbose outputs during practice games. This observation highlighted specific scenarios where the solution was struggling, guiding the development and training of specialized models.

2.1.3 Empirical Weight Assignment

In order to properly use all the models effectively, at each instant, I gave weights to each of the techniques, which after a weighted sum gave the finalized probability distribution of the letters which could be guessed, from this distribution we took the letter with the maximum probability which was not already chosen. The following table summarizes the empirically defined weights based on the proportion of unknown letters (`unknown_ratio`) and word length.

These weights were meticulously adjusted by monitoring the solver's performance in various game scenarios, ensuring that each model contributed optimally based on the current state of the game. The adjustments were guided by empirical observations from verbose outputs during practice games, where I identified areas where specific models could better handle certain word complexities.

Unknown Ratio	Short Word	High	Medium	Low	Heuristic	Short
> 0.7	Yes	0.2	0.15	0.15	0.4	0.1
> 0.4	Yes	0.0	0.2	0.2	0.4	0.2
> 0.15	Yes	0.0	0.1	0.3	0.4	0.2
≤ 0.15	Yes	0.0	0.05	0.2	0.05	0.7
> 0.7	No	0.2	0.3	0.2	0.3	0.0
> 0.4	No	0.15	0.4	0.15	0.3	0.0
> 0.15	No	0.1	0.2	0.3	0.4	0.0
≤ 0.15	No	0.05	0.15	0.3	0.5	0.0

Table 1: Empirically Defined Weights for Models and Heuristics

3 Implementation

The implementation is structured into several key components: dictionary management, model inference, guessing logic, and game interaction. Below is an overview of each part (except game interaction as that was already provided), accompanied by relevant code snippets.

3.1 Dictionary Management

Efficient handling of the dictionary is crucial for filtering potential word candidates based on current guesses.

```
def build_dictionary(self, dictionary_file_location):
    with open(dictionary_file_location, "r") as f:
        full_dictionary = f.read().strip().split()
    full_dictionary = [w.lower() for w in full_dictionary if w.isalpha()]
    return full_dictionary

def build_substring_dictionary(self, df):
    max_length = max(len(w) for w in df)
    n_word_dictionary = {i: [] for i in range(3, min(max_length, 30) + 1)}
    for count in range(3, min(max_length, 30) + 1):
        for w in df:
            if len(w) >= count:
                for i in range(len(w) - count + 1):
                    n_word_dictionary[count].append(w[i:i + count])
    return n_word_dictionary
```

Listing 2: Dictionary Management

3.2 Model Inference

Each model predicts the probabilities of letters in masked positions. The inference process involves preparing inputs, obtaining predictions, and aggregating probabilities.

```
def get_model_probs(self, model, clean_word):
    mlm_input_chars = []
    for c in clean_word:
        if c == '.':
            mlm_input_chars.append('[MASK]')
        elif c in self.vocab:
            mlm_input_chars.append(c)
        else:
            mlm_input_chars.append('_')

    input_ids = [self.char_to_idx.get(ch, self.char_to_idx['_']) for ch
    in mlm_input_chars]

    model_max_len = model.pos_enc.pe.size(1)
    if len(input_ids) > model_max_len:
        input_ids = input_ids[:model_max_len]

    input_ids_tensor = torch.tensor([input_ids], dtype=torch.long,
   device=self.device)
    with torch.no_grad():
        logits = model(input_ids_tensor)
        mask_positions = [i for i, ch in enumerate(mlm_input_chars) if
   ch == '[MASK]']
        letter_indices = range(2, 28)  # Indices for lowercase letters
        letter_probs = collections.Counter()
        if mask_positions:
            for pos in mask_positions:
                pos_logits = logits[0, pos]
                pos_probs = torch.softmax(pos_logits, dim=0)
                for li in letter_indices:
                    ltr = self.idx_to_char[li]
                    letter_probs[ltr] += pos_probs[li].item()
            for ltr in letter_probs:
                letter_probs[ltr] = letter_probs[ltr] / len(
   mask_positions)
        else:
            for ltr in string.ascii_lowercase:
                letter_probs[ltr] = 1.0 / 26.0
    return dict(letter_probs)
```

Listing 3: Model Inference and Probability Aggregation

3.3 Guessing Logic

The guessing logic combines model predictions with heuristic scores to determine the next best letter to guess.

1. Basic Setup and Heuristic Scores

```
def guess(self, word):
    vowels = set('aeiou')
    clean_word = word[::2].replace("_",".")
    len_word = len(clean_word)
    num_unknown = clean_word.count('.')
    unknown_ratio = num_unknown / len_word if len_word>0 else 0.0

    # Filter dictionary based on known pattern
    regex_pattern = "^" + clean_word + "$"
    new_dictionary = [w for w in self.current_dictionary if len(w)==
   len_word and re.fullmatch(regex_pattern,w)]
    self.current_dictionary = new_dictionary

    # Compute heuristic scores
    h_counts = self._get_heuristic_counts(clean_word)
    heuristic_scores = self._to_scores(h_counts)
```

Listing 4: Initial Setup in Guessing

Here, we set up vowels, compute `unknown_ratio`, and get heuristic scores from the filtered dictionary.

2. Model Probabilities and Weighting

```
# Based on word length, fetch probabilities from relevant models
if len_word <= 7:
    model_short_prob = self.get_model_probs(self.mlm_short, clean_word)
    model_high_prob = self.get_model_probs(self.mlm_high, clean_word)
    model_medium_prob = self.get_model_probs(self.mlm_medium,
   clean_word)
    model_low_prob = self.get_model_probs(self.mlm_low, clean_word)
    # Set weights for short words scenario (from empirical observations
   )
    # ...
else:
    model_high_prob = self.get_model_probs(self.mlm_high, clean_word)
    model_medium_prob = self.get_model_probs(self.mlm_medium,
   clean_word)
    model_low_prob = self.get_model_probs(self.mlm_low, clean_word)
    # Set weights for longer words scenario
    # ...
```

Listing 5: Model Probability Retrieval

We grab probabilities from the chosen models. If it's a short word, the short model and heuristics might get more weight; for longer words, we rely more on high/medium/low models plus heuristics.

3. Combining and Selecting Letters

```
# Combine all scores: model predictions * weights + heuristic
combined_scores = {}
# ... code to sum up probabilities using the chosen weights ...

# Sort combined scores and pick the best letter not guessed yet
sorted_letters = sorted(combined_scores.items(), key=lambda x:x[1],
    reverse=True)

chosen_letter = self._pick_best_letter(sorted_letters, v_ratio,
    unknown_ratio, vowels)
return chosen_letter
```

Listing 6: Combining Probabilities and Choosing a Letter

We merge model predictions and heuristic scores, then pick the top-scoring letter that's not guessed yet. If everything fails, we use a fallback strategy, ensuring we never loop infinitely.

4 Observation and Results

The Hangman solver exhibited promising performance during training and initial testing phases. Below are the key observations and results:

4.1 Win Rate Statistics

Phase	Win Rate
Training	60%
Final Execution	56.8%

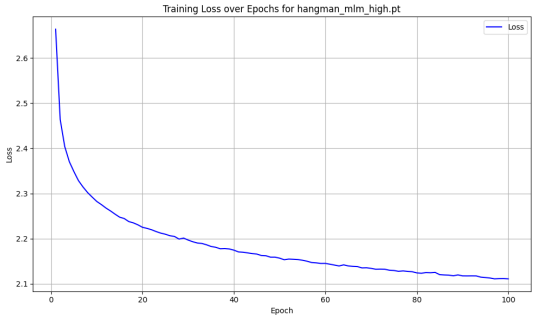
Table 2: Win Rate Comparison

4.2 Training Loss Graphs

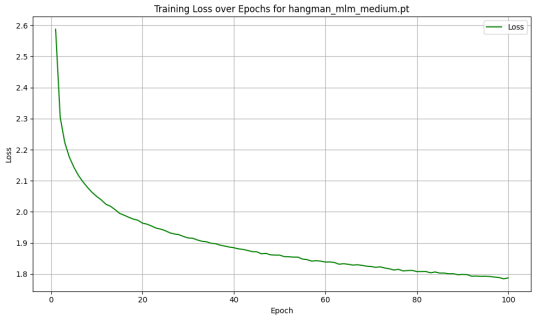
The training loss for each of the four models is depicted below. These graphs illustrate the convergence behavior and training stability across different model configurations.

4.3 Results (& unfortunate tweaking)

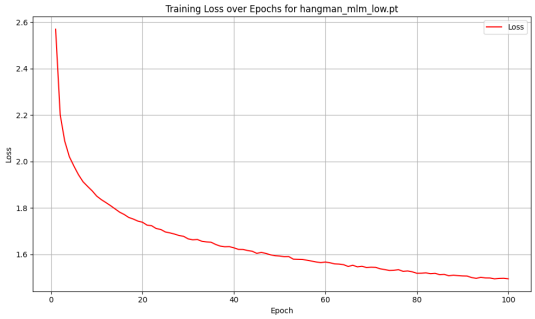
During the training phase, the AI demonstrated a win rate of approximately 60%, demonstrating the effectiveness of integrating Transformer MLMs with heuristic strategies. The initial 60% win rate was encouraging. However, after tweaking the weights (hoping for better results without fully validating on practice games), the final win rate dipped to 56.8%. This taught me the value of patience and thorough testing before final adjustments.



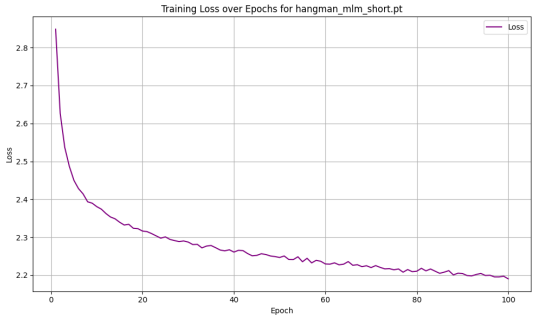
(a) High Model Loss



(b) Medium Model Loss



(c) Low Model Loss



(d) Short Model Loss

Figure 1: Training Loss Graphs for All Models

5 Conclusion & Improvements

- Due to limited computational power (a MacBook M2 Air and free Kaggle GPU sessions), I could not train deeper, more complex models. With stronger hardware and more time, I could train heavier models for even more epochs, which would likely boost accuracy even further.

It was a pleasure and a rewarding experience to work on this problem. Thank You.