

# Neural Hangman Solver

Ashutosh Rabia, Indian Institute of Technology Kanpur

1st June 2025

## Introduction

Hangman is a classic word-guessing game where a hidden word is represented by blanks, and the player proposes letters one at a time. Correct guesses reveal letters; incorrect guesses accumulate until six mistakes result in a loss. Effective solvers leverage linguistic patterns, letter frequencies, and positional cues to maximize success.

This report presents a *hybrid neural-statistical* Hangman solver that:

- Encodes each partially revealed word (with blanks and boundary markers) into a fixed-length tensor, along with embeddings of previously missed letters.
- Trains a bi-directional LSTM (hidden size 512, dropout) to predict, for each blank position, the most likely letter.
- Computes a positional-bias tensor from a 250 000-word dictionary to favor letters historically common at specific positions.
- Combines a fast “frequency filter” (counting how often each letter appears among dictionary words matching the current pattern) with the LSTM’s predicted probabilities. These scores are linearly weighted (=0.6 for neural, 0.4 for frequency) and masked to exclude already-guessed letters.

## Intuition

### 1. Vowel-First Heuristic

Early in each game, if no vowel has appeared and no vowel has yet been guessed, the solver forces a vowel guess in the order {e, a, o, i, u}. English words nearly always contain at least one vowel; forcing a vowel early yields high information gain.

### 2. Regex-Based Candidate Filtering

Once a vowel is placed (or forced), the solver builds a regex pattern  $\text{^p}_1 \text{p}_2 \dots \text{p}_n \text{\$}$ , where each  $\text{p}_i$  is a revealed letter or “.” for a blank. For example, the pattern  $\text{p\_t}$  becomes  $\text{^p.t\$}$ . All dictionary words of length  $n$  matching this pattern form the candidate set. For each unguessed letter  $c$ , compute

$$\text{freq\_score}(c) = \frac{|\{w \in \text{candidates} : c \in w\}|}{\max_{c'} |\{w \in \text{candidates} : c' \in w\}|} \in [0, 1].$$

If there are no candidates or no unguessed letters, all frequency scores are set to 0.

### 3. Neural Prediction (Bi-LSTM with Positional Bias)

The bi-directional LSTM views the framed pattern  $\{\dots\}$  plus an embedding of missed letters. Trained on millions of random “masked + partial-reveal” examples, it learns letter

co-occurrence and positional patterns (e.g., “q” likely followed by “u,” or “th” common at word starts). At inference, the LSTM outputs logits  $\ell_{t,c}$  for each time step  $t$  and each letter  $c$ . For blank positions  $B$ , define

$$\text{neural\_score}(c) = \sum_{t \in B} \text{Softmax}(\ell_{t,c}),$$

with already-guessed letters zeroed out. A precomputed positional-bias tensor  $\text{pos\_bias}[c, p]$  ( $26 \times \text{embedding\_len}$ ) guides training by adding  $\lambda_{\text{pos}} \log(\text{pos\_bias}[c, p])$  to the loss when predicting letter  $c$  at position  $p$ .

#### 4. Score Combination

For each candidate letter  $c$ , compute

$$\text{combined\_score}(c) = \alpha \text{neural\_score}(c) + (1 - \alpha) \text{freq\_score}(c), \quad \alpha = 0.6.$$

The solver picks  $\arg \max_c \text{combined\_score}(c)$ . This hybrid approach leverages both neural positional knowledge and real-time frequency statistics.

## Methodology

### Preprocessing

1. **Load & Split Dictionary:** Read 250 000 words from `words_250000_train.txt`, shuffle randomly, split 95 % (237 500 words) for training, 5 % (12 500 words) for validation.
2. **Frame Words:** For each word  $w$ , define `padded` =  $\{w \mid$ . The start marker `{` and end marker `|` explicitly delimit the word.
3. **Compute Positional Bias:**

- $\text{embedding\_len} = (\max_w |w|) + 2$ .  $\max |w| = 29$ , so  $\text{embedding\_len} = 31$ .
- Initialize `counts`  $\in R^{26 \times 31}$  to zeros. For each framed training word  $\{w \mid\}$ , let  $\text{start} = 31 - |\{w \mid\}|$ . For index  $i = 0, \dots, |\{w \mid\}| - 1$ , let position  $p = \text{start} + i$ . If the character at that index is a letter  $c \in \{a, \dots, z\}$ , increment `counts`[ $c - a, p$ ].
- Normalize columns:

$$\text{pos\_bias}[c, p] = \frac{\text{counts}[c, p]}{\sum_{c'=0}^{25} \text{counts}[c', p] + 1e-9}, \quad c \in \{0, \dots, 25\}, p \in \{0, \dots, 30\}.$$

Store as a `torch.FloatTensor(26 × 31)` on GPU.

4. **Generate Synthetic Training Samples:** For each training word  $w$ :

- (a) Let `padded` =  $\{w \mid$ . Choose a reveal-percentage  $r \in \{0, 0.2, 0.5, 0.8\}$ . Compute  $\text{num\_reveals} = \lfloor r \cdot |w| \rfloor$ . Randomly sample that many interior positions to reveal; add those letters to `guessed_letters`.
- (b) Randomly mask  $m \in [1, \max(1, |w| - \text{num\_reveals})]$  positions (not already revealed) by replacing them with “\_.”
- (c) Call

$(\vec{missed\_vec}, \text{revealed\_mask}, \text{mask\_pos}) = \text{encode\_input}(\text{masked\_string}, w, \text{guessed\_letters}, 31)$ .

$\vec{missed\_vec} \in \{0, \dots, 29\}^{31}$ ,  $\text{missed\_vec} \in \{0, 1\}^{26}$ ,  $\text{revealed\_mask} \in \{0, 1\}^{31}$ ,  $\text{mask\_pos} \subset \{0, \dots, 30\}$ .

- (d) If `mask_pos = ∅`, set `target_pos = 0`, `y_letter = 0`. Otherwise pick `target_pos` randomly from `mask_pos`, find the true letter at that index in `{w |, define y_letter = ord(target_letter) - 97.`
- (e) Return the tuple of tensors:

`X = LongTensor(→), missed_vec = LongTensor(missed_vec), revealed_mask = LongTensor(revealed_mask)`

These examples populate a custom `HangmanDataset` used by `DataLoader(batch_size=256)`.

## Model Architecture & Training

### 1. Character & Missed-Letters Embedding

- `char_embed = nn.Embedding(30, 64)` maps indices `{0, ..., 29}` (padding, `{, |, “-”, letters)` to 64-dim vectors.
- `missed_embed = nn.Linear(26, 16)` embeds the 26-bit “missed letters” mask into 16 dimensions.

### 2. Bi-Directional LSTM

- Input size per time step: `64 + 16 = 80`.
- Hidden size per direction: 512. Layers: 2. `self.lstm = nn.LSTM(80, 512, num_layers = 2, batch_first = True, bidirectional = True)`.
- Output per time `t`: (batch, 1024) (512 forward + 512 backward).

3. **Dropout Layer** `self.dropout = nn.Dropout(p = 0.5)`. Applied to LSTM outputs during training.
4. **Fully-Connected Layer** `self.fc = nn.Linear(512 × 2, 26)`. Maps each (1024) vector at time `t` to 26 logits.
5. **Loss with Positional Bias** For a batch of size `B`:

$$\text{logits} = \text{model}(X, \text{missed}) \quad (\in R^{B \times 31 \times 26}).$$

Let `posi` be the target position for example `i`. Extract

$$L = \text{logits}[:, 0, \dots, B-1, \text{pos}] \quad (\in R^{B \times 26}).$$

Add bias:

$$\text{bias\_term} = \lambda_{\text{pos}} \log(\text{pos\_bias}[:, \text{pos}]^\top + 1e^{-9}) \quad (\in R^{B \times 26}),$$

where `λpos = 0.2`. Then

$$L_{\text{biased}} = L + \text{bias\_term}, \quad \text{loss} = \text{CrossEntropyLoss}(\text{label\_smoothing} = 0.1)(L_{\text{biased}}, y_{\text{letter}}).$$

Backpropagation uses `autocast()` and `GradScaler()` for mixed precision.

### 6. Training Details

- Epochs: 18 (with early stopping if validation win-rate does not improve for 5 consecutive epochs).
- Optimizer: `AdamW(lr = 1 × 10-3, weight_decay = 1 × 10-2)`.
- Scheduler: `CosineAnnealingLR(Tmax = 50)`.
- Mixed precision via `torch.cuda.amp`.
- After each epoch, run full validation over 12 500 held-out words by simulating Hangman games (max 6 misses), compute win-rate, save best model.

## Data Structures

- **CHAR\_MAP:**  $\{_: 27, \{ : 28, | : 29, a : 1, \dots, z : 26\}$ . Index 0 is padding.
- `embedding_len` = 31. Maximum word length (29) + 2 boundary markers.
- `pos_bias`  $\in R^{26 \times 31}$ . Each column sums to 1 across 26 letters, representing  $P(\text{letter} = c \mid \text{position} = p)$ .
- $\vec{c} \in \{0, \dots, 29\}^{31}$  (input indices for `Embedding`).
- `missed_vec`  $\in \{0, 1\}^{26}$  (binary mask of incorrect guesses).
- `revealed_mask`  $\in \{0, 1\}^{31}$  (1 if that position is a revealed letter).
- `mask_pos`  $\subset \{0, \dots, 30\}$  (indices of blanks).
- `target_pos`  $\in \{0, \dots, 30\}$ , `y_letter`  $\in \{0, \dots, 25\}$ .
- Batch-level tensors (B=256):

$$X \in Z^{256 \times 31}, \quad \text{missed} \in Z^{256 \times 26}, \quad \text{pos} \in Z^{256}, \quad y\_letter \in Z^{256}.$$

- LSTM internal: input at each time step is 80-dim; output is (256, 31, 1024).
- Dropout applied to (256, 31, 1024) during training.
- Final FC: (256, 31, 1024)  $\rightarrow$  (256, 31, 26).

## Code Snippet

```
def guess(self, word):
    clean = word.replace(' ', '')
    VOWELORDER = ['e', 'a', 'o', 'i', 'u']
    alpha = 0.6

    # Vowel-First Heuristic
    if (not any(v in clean for v in VOWELORDER)) and \
        (not any(g in self.guessed_letters for g in VOWELORDER)):
        for v in VOWELORDER:
            if v not in self.guessed_letters:
                self.guessed_letters.append(v)
        return v

    # Candidate Filtering via Regex
    pattern = ''.join(c if c != '_' else '.' for c in clean)
    possible_words = [
        w for w in self.full_dictionary
        if len(w) == len(clean) and re.match(f"^{pattern}$", w)
    ]
    candidate_letters = [c for c in string.ascii_lowercase
                        if c not in self.guessed_letters]

    # Frequency Scores
    if possible_words and candidate_letters:
```

```

letter_counts = {
    c: sum(1 for w in possible_words if c in w)
    for c in candidate_letters
}
max_count = max(letter_counts.values(), default=0)
freq_scores = {
    c: (letter_counts[c] / max_count) if max_count > 0 else 0.0
    for c in candidate_letters
}
else:
    freq_scores = {c: 0.0 for c in candidate_letters}

# Neural Model Prediction
padded = '{ ' + clean + ' | '
vec, missed_vec, revealed_mask, unrevealed_emb_positions = encode_input(
    padded, clean, self.guessed_letters, self.embedding_len
)
X = torch.tensor([vec], dtype=torch.long).to(self.device) # (1,50)
missed = torch.tensor([missed_vec], dtype=torch.long).to(self.device)
# (1,26)
with torch.no_grad():
    logits = self.model(X, missed)[0] # (50,26)
    probs = torch.softmax(logits, dim=-1)

    if unrevealed_emb_positions:
        probs_unrevealed = probs[unrevealed_emb_positions] # (num_blanks,26)
        total_probs = probs_unrevealed.sum(dim=0) # (26,)
        masked_total_probs = total_probs.clone()
        for g in self.guessed_letters:
            idx = ord(g) - 97
            if 0 <= idx < 26:
                masked_total_probs[idx] = 0
        model_scores = {
            chr(i + 97): masked_total_probs[i].item()
            for i in range(26)
        }
    else:
        model_scores = {c: 0.0 for c in candidate_letters}

# Combine Scores & Choose
combined_scores = {
    c: alpha * model_scores.get(c, 0.0)
    + (1 - alpha) * freq_scores.get(c, 0.0)
    for c in candidate_letters
}
if combined_scores:
    chosen_letter = max(combined_scores, key=combined_scores.get)
else:
    chosen_letter = 'a' # fallback

self.guessed_letters.append(chosen_letter)

```

```
return chosen_letter
```

## Function Descriptions

Function	Description
<code>encode_input</code>	Given <code>masked_word</code> , <code>original_word</code> , <code>guessed_letters</code> , <code>embedding_len</code> , constructs: 1) $\tilde{\in}\{0, \dots, 29\}^{\text{embedding\_len}}$ (mapping <code>{ }</code> , letters, <code>"_"</code> , padding), 2) <code>missed_vec</code> $\in \{0, 1\}^{26}$ (incorrect guesses), 3) <code>revealed_mask</code> $\in \{0, 1\}^{\text{embedding\_len}}$ (1 if that position is a revealed letter), 4) <code>mask_pos</code> $\subset \{0, \dots, \text{embedding\_len} - 1\}$ (indices of <code>"_"</code> ).
<code>compute_pos_bias</code>	From list of words, frames each as <code>{w  }</code> , tallies letter frequencies per position into <code>counts[26 × 31]</code> , normalizes columns to produce <code>pos_bias</code> $\in [0, 1]^{26 \times 31}$ .
<code>HangmanDataset</code>	On each <code>__getitem__</code> , pads a word <code>{w  }</code> , randomly reveals a fraction $\in \{0, 0.2, 0.5, 0.8\}$ , masks some letters with <code>"_"</code> , calls <code>encode_input</code> , selects one masked position as target. Returns <code>(X, missed_vec, revealed_mask, y_letter, pos)</code> .
<code>HangmanLSTM</code>	Neural model: – <code>char_embed</code> : $(30 \rightarrow 64)$ , – <code>missed_embed</code> : $(26 \rightarrow 16)$ , – <code>lstm</code> : LSTM( $80 \rightarrow 512 \times 2$ , layers = 2), – <code>dropout</code> : $p = 0.5$ , – <code>fc</code> : $(1024 \rightarrow 26)$ . <code>forward(x, missed)</code> returns $(B, 31, 26)$ logits.
<code>train_loop</code>	For each epoch: 1) Iterate over <code>dl_train</code> : compute logits = <code>model(X, missed)</code> $\in (R^{B \times 31 \times 26})$ , extract $L = \text{logits}[:, \text{pos}] \in R^{B \times 26}$ , add $\lambda_{\text{pos}} \log(\text{pos\_bias}[:, \text{pos}])$ , compute <code>CrossEntropyLoss(label_smoothing = 0.1)</code> , backprop with mixed precision. 2) Step scheduler, run validation on 12 500 words by simulating Hangman games (six misses), compute win-rate, save best model, early stop if no improvement for 5 epochs.
<code>guess</code>	Implements the hybrid inference pipeline: (a) Vowel-first for high information early; (b) Regex filter $\rightarrow$ frequency scores; (c) LSTM $\rightarrow$ neural scores; (d) Combined weighted score $\alpha = 0.6$ . Returns the chosen letter.
<code>start_game</code>	Resets <code>guessed_letters</code> = $\emptyset$ , calls API <code>"/new_game"</code> , loops until solved or 6 misses, calling <code>guess()</code> and sending <code>"/guess_letter"</code> each iteration, returns True if word is solved.

Table 1: Summary of key functions

## Conclusion

This neural-statistical Hangman solver integrates:

- A **vowel-first heuristic** for rapid information gain.

- A **regex-based frequency filter** over dictionary candidates.
- A **bi-directional LSTM** (hidden=512, 2 layers, dropout=0.5) trained on synthetic masked examples.
- A **positional bias tensor** guiding the LSTM via biased loss.
- A **linear combination** of neural (=0.6) and frequency (0.4) scores at inference.

This solver achieves a **57.6 %** win-rate, outperforming other naive ML(CNN = 47 %, LSTM = 52 %) approaches. Future directions include dynamic scheduling, transformer-based encoders, and curriculum training from shorter to longer words.