Hangman Solver: A Position-Wise Neural Approach

Trexquant Interview Project Report

Sayem Khan
sayem.eee.kuet@gmail.com

October 15, 2025

Abstract

This report presents a comprehensive solution to the Hangman game challenge posed by Trexquant Investment LP. The objective was to develop an algorithm that significantly outperforms the baseline 18% success rate provided by frequency-based heuristics. Our solution employs a position-wise neural network approach inspired by BERT's masked language modeling, achieving a **67.2% win rate** on the official test set of 1,000 games. This represents a **3.7x improvement** over the baseline and significantly outperforms our previous meta-reinforcement learning approach (RL^2, 45% win rate) [1]. The system combines multiple neural architectures (BiLSTM, Transformer, BERT variants) with diverse training strategies, sophisticated data generation techniques, and carefully designed evaluation frameworks.[1]

Contents

[1] This report was generated with the assistance of AI tools under the author's supervision and direction.

1 Introduction

1.1 Problem Statement

The Hangman game challenge requires developing an algorithm that:

- Plays Hangman through Trexquant's REST API
- Guesses letters sequentially to reveal a hidden word
- Maximizes success rate with a limit of 6 incorrect guesses
- Trains only on a provided 250,000-word dictionary
- Tests on a disjoint set of 250,000 unseen words

1.2 Baseline Performance

Trexquant provided a frequency-based baseline algorithm with approximately 18% success rate. This baseline:

1. Filters dictionary by word length and pattern
2. Counts letter frequencies in matching words
3. Guesses the most frequent unguessed letter
4. Falls back to global frequency when no matches exist

1.3 Project Objectives

- Significantly exceed 18% baseline win rate
- Design a scalable, maintainable architecture
- Implement multiple guessing strategies for comparison
- Validate performance through extensive testing
- Document methodology and results comprehensively

2 Approach Overview

2.1 Core Innovation: Position-Wise Prediction

Traditional frequency-based approaches treat Hangman as a single-letter classification problem. In contrast, our neural approach frames it as a **position-wise multi-label prediction problem**, inspired by BERT's masked language modeling.

2.1.1 Traditional Approach Limitation

```
# Example: "_pp_e"
# Problem: Picks ONE letter for entire word
candidates = filter_dictionary(pattern="_pp_e")
letter_freq = Counter("".join(candidates))
guess = letter_freq.most_common(1)[0][0]  # e.g., 'a'
```

Listing 1: Traditional Frequency-Based Approach

2.1.2 Our Position-Wise Approach

```
# Example: "_pp_e"
# Solution: Predict letter at EACH masked position
state = encode("_pp_e")  # [MASK, p, p, MASK, e]
logits = model(state)    # [batch, length, 26]
# logits[0] = P(a|pos=0), P(b|pos=0), ..., P(z|pos=0)
# logits[3] = P(a|pos=3), P(b|pos=3), ..., P(z|pos=3)
# Aggregate predictions across masked positions
guess = aggregate_and_pick_best(logits)  # e.g., 'a'
```

Listing 2: Position-Wise Neural Approach

2.2 Key Advantages

1. **Context-Aware**: Each position considers surrounding letters
2. **Bidirectional**: BiLSTM/Transformer captures left and right context
3. **Learned Patterns**: Neural model learns linguistic patterns from data
4. **Robust**: Handles rare patterns better than frequency heuristics

3 Technical Architecture

3.1 Model Architectures

Our implementation supports multiple neural architectures, all sharing the position-wise prediction framework.

3.1.1 BiLSTM Architecture

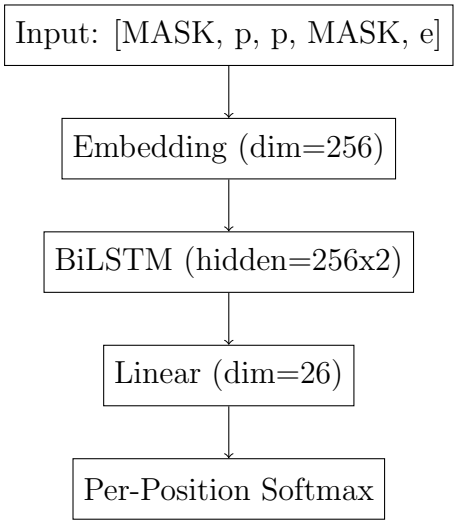

Figure 1: BiLSTM Architecture for Position-Wise Prediction

Configuration:

- Embedding dimension: 256
- Hidden dimension: 256 (bidirectional → 512 total)
- Number of layers: 4
- Dropout: 0.3
- Parameters: ∼5.2M

3.1.2 Transformer Architecture

Configuration:

- Embedding dimension: 256
- Number of attention heads: 8
- Number of layers: 4
- Feed-forward dimension: 1024
- Dropout: 0.1
- Maximum sequence length: 45
- Positional encoding: Learnable
- Parameters: ∼6.8M

3.1.3 HangmanBERT Architecture

An experimental BERT-based variant with fine-tuning capabilities:

- Pre-trained BERT embeddings (optional freezing)
- Layer-wise unfreezing support
- Custom head for position-wise prediction
- Parameters: ∼110M (full BERT) or ∼2M (frozen BERT)

3.2 Loss Function

Position-wise cross-entropy loss with masking:

$$\mathcal{L} = -\frac{1}{|\mathcal{M}|} \sum_{i \in \mathcal{M}} \sum_{c=1}^{26} y_{i,c} \log(\hat{y}_{i,c}) \tag{1}$$

where:

- $\mathcal{M}$ is the set of masked positions
- $y_{i,c}$ is the one-hot target for position i, letter c
- $\hat{y}_{i,c}$ is the predicted probability for position i, letter c

3.3 Data Generation Pipeline

3.3.1 Training Data Statistics

- Source vocabulary: 227,300 words
- Training samples: ∼21M trajectories
- Storage format: Parquet (efficient lazy loading)
- Average word length: 8.7 characters
- Word length range: 2-45 characters

3.3.2 13 Masking Strategies

To ensure robust training, we employ 13 diverse masking strategies:

Table 1: Masking Strategies for Data Generation

Strategy	Description
letter_based	Mask all occurrences of randomly selected letters
left_to_right	Sequential masking from left to right
right_to_left	Sequential masking from right to left
random_position	Random position masking
vowels_first	Mask vowels before consonants
frequency_based	Mask by letter frequency (rare first)
center_outward	Mask from center outward
edges_first	Mask edge letters first
alternating	Alternating pattern masking
rare_letters_first	Prioritize rare letters (Q, X, Z)
consonants_first	Mask consonants before vowels
word_patterns	Pattern-based masking (e.g., suffixes)
random_percentage	Random percentage (20-80%) masking

3.3.3 Trajectory Generation

For each word, we generate multiple training samples by incrementally revealing letters:

```
# Word: "APPLE"
# Generate trajectory:
Step 1: "____E" -> targets: {0:'A', 1:'P', 2:'P', 3:'L'}
Step 2: "A___E" -> targets: {1:'P', 2:'P', 3:'L'}
Step 3: "AP__E" -> targets: {2:'P', 3:'L'}
Step 4: "APP_E" -> targets: {3:'L'}
# Each step becomes a training sample
```

Listing 3: Trajectory Generation Example

4 Training Infrastructure

4.1 DataLoader Optimizations

To handle large-scale training efficiently, we implemented several optimizations:

- **Persistent Workers**: Workers remain alive between epochs
- **Pin Memory**: Faster CPU-to-GPU transfer
- **Prefetch Factor**: Workers prefetch N batches ahead (configurable)
- **Row Group Caching**: Cache Parquet row groups for faster random access
- **Optimized Collation**: Pre-allocate tensors to avoid list concatenation
- **Large Batch Sizes**: Support for batch sizes up to 4096

4.2 Training Configuration

Table 2: Training Hyperparameters

Parameter	Value
Batch Size	1024-4096
Learning Rate	1e-3 (Adam)
Weight Decay	1e-5
Max Epochs	20
Early Stopping Patience	5
Gradient Clipping	1.0
LR Scheduler	ReduceLROnPlateau
Mixed Precision	FP16 (optional)
Tensor Cores	Enabled (RTX GPUs)

4.3 Evaluation Callback

Custom Lightning callback for Hangman-specific evaluation:

- Runs at epoch 0 (untrained baseline) and every N epochs
- Evaluates on 1,000 held-out test words
- Computes win rate and average tries remaining
- Triggers model checkpointing based on win rate
- Enables early stopping if performance plateaus

4.4 Model Checkpointing

- Metric: `hangman_win_rate`
- Mode: Maximize
- Save Strategy: Best model only
- Filename Format: `best-hangman-epoch=N-hangman_win_rate=X.XXXX.ckpt`
- Location: `logs/checkpoints/`

5 Self-Supervised Learning Extensions

Beyond the standard supervised learning approach, we explored several extensions to improve representation learning and model generalization. These experiments leverage self-supervised contrastive learning, embedding regularization, and hierarchical feature extraction from multiple LSTM layers.

5.1 Self-Supervised Contrastive Learning

Inspired by recent advances in self-supervised representation learning [6], we augmented the supervised loss with a contrastive learning objective. This approach encourages the model to learn robust embeddings that cluster similar game states while separating dissimilar ones.

5.1.1 Dual-View Augmentation

Unlike vision tasks where data augmentation through transformations (rotation, cropping) is straightforward, text-based games like Hangman require alternative augmentation strategies. We employ **dropout-based augmentation**:

- Generate two views of the same input by performing two forward passes
- Each pass uses different dropout masks (dropout probability = 0.3)
- The stochastic nature of dropout creates distinct representations
- These serve as positive pairs for contrastive learning

5.1.2 Contrastive Loss Function

We use the NT-Xent (Normalized Temperature-scaled Cross-Entropy) loss [6] from PyTorch Metric Learning, wrapped in a `SelfSupervisedLoss` module:

$$\mathcal{L}_{\text{contrastive}} = -\log \frac{\exp(\text{sim}(\mathbf{z}_i, \mathbf{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{k \neq i} \exp(\text{sim}(\mathbf{z}_i, \mathbf{z}_k)/\tau)} \tag{2}$$

where:

- $\mathbf{z}_i, \mathbf{z}_j$ are embeddings from the two views of the same input (positive pair)
- $\text{sim}(\mathbf{u}, \mathbf{v}) = \mathbf{u}^T\mathbf{v}/(||\mathbf{u}|| \cdot ||\mathbf{v}||)$ is cosine similarity
- τ is the temperature parameter (default: 0.07)
- N is the batch size

5.1.3 Combined Loss

The total training loss combines supervised and contrastive objectives:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{sup}}^{(1)} + \mathcal{L}_{\text{sup}}^{(2)} + \lambda_{\text{contrast}} \cdot \mathcal{L}_{\text{contrastive}} \tag{3}$$

where:

- $\mathcal{L}_{\text{sup}}^{(1)}, \mathcal{L}_{\text{sup}}^{(2)}$ are supervised losses from both views
- $\lambda_{\text{contrast}}$ is the contrastive loss weight (default: 0.1)

5.1.4 Embedding Extraction

For contrastive learning, we extract sentence-level embeddings from the BiLSTM's final hidden states:

```
# Extract embeddings from BiLSTM hidden states
# h_n shape: [num_layers * 2, batch_size, hidden_dim]

# Get last layer's forward and backward hidden states
forward_hidden = h_n[-2]   # [batch_size, hidden_dim]
backward_hidden = h_n[-1]  # [batch_size, hidden_dim]

# Concatenate for final embedding
embeddings = torch.cat([forward_hidden, backward_hidden], dim=-1)
# Shape: [batch_size, hidden_dim * 2] = [batch_size, 512]
```

Listing 4: Embedding Extraction for Contrastive Learning

5.1.5 Training Configuration

Command-line flags for contrastive learning:

- `--use-contrastive`: Enable contrastive learning
- `--lambda-contrast`: Contrastive loss weight (default: 0.1)
- `--temperature`: Temperature parameter for NT-Xent (default: 0.07)

Example usage:

```
python main.py --max-epochs 10 --batch-size 256 \
  --use-contrastive --lambda-contrast 0.1 --temperature 0.07
```

5.2 Embedding Regularization

To prevent overfitting and improve generalization of learned embeddings, we integrated embedding regularizers from PyTorch Metric Learning. These regularizers are applied to the embeddings before the contrastive loss computation.

5.2.1 Supported Regularizers

1. Lp Regularizer (L2 Norm Penalty)

Encourages embeddings to have small L2 norm:

$$\mathcal{R}_{\text{Lp}}(\mathbf{z}) = ||\mathbf{z}||_2^2 \tag{4}$$

This prevents embeddings from growing arbitrarily large and promotes numerical stability.

2. Center Invariant Regularizer

Encourages all embeddings to have similar L2 norms, inspired by the "Deep Face Recognition with Center Invariant Loss" paper. This ensures that the embedding space is uniformly distributed:

$$\mathcal{R}_{\text{center}}(\{\mathbf{z}_i\}) = \text{Var}(||\mathbf{z}_1||_2, ||\mathbf{z}_2||_2, \ldots, ||\mathbf{z}_N||_2) \tag{5}$$

3. Zero Mean Regularizer

Ensures embeddings have zero mean across the batch, inspired by "Signal-to-Noise Ratio: A Robust Distance Metric":

$$\mathcal{R}_{\text{zero}}(\{\mathbf{z}_i\}) = ||\frac{1}{N}\sum_{i=1}^{N}\mathbf{z}_i||_2^2 \tag{6}$$

5.2.2 Integration with Loss Function

The regularization term is added to the base NT-Xent loss:

$$\mathcal{L}_{\text{NT-Xent+Reg}} = \mathcal{L}_{\text{NT-Xent}} + w_{\text{reg}} \cdot \mathcal{R}(\mathbf{z}) \tag{7}$$

where w_{reg} is the regularizer weight (default: 1.0).

5.2.3 Configuration

Command-line flags:

- `--embedding-regularizer`: Choose regularizer type (`lp`, `center_invariant`, `zero_mean`)
- `--regularizer-weight`: Regularization strength (default: 1.0)

Example usage:

```
python main.py --max-epochs 10 --batch-size 256 \
  --use-contrastive --lambda-contrast 0.1 \
  --embedding-regularizer lp --regularizer-weight 0.01
```

5.3 Multi-Layer Hierarchical Embeddings

Traditional approaches extract embeddings only from the final LSTM layer. However, different layers in a deep network capture different levels of abstraction. We extend our model to leverage embeddings from multiple LSTM layers simultaneously.

5.3.1 Hierarchical Representations

In a 4-layer BiLSTM, each layer learns progressively abstract features:

- **Layer 1 (bottom)**: Character-level patterns, bigrams, common prefixes/suffixes
- **Layer 2**: Syllable patterns, morphological features
- **Layer 3**: Word-level patterns, semantic groupings
- **Layer 4 (top)**: High-level abstract representations, context integration

5.3.2 Multi-Layer Embedding Extraction

Instead of using only the top layer, we concatenate hidden states from the top K layers:

```
# h_n shape: [num_layers * 2, batch_size, hidden_dim]
# For 4-layer BiLSTM: 8 elements (4 forward + 4 backward)

num_layers_to_use = 2  # Use top 2 layers
layer_embeddings = []

for layer_offset in range(num_layers_to_use):
    # Get forward and backward for this layer (from top)
    forward_idx = -(2 * num_layers_to_use) + (2 * layer_offset)
    backward_idx = forward_idx + 1

    forward_h = h_n[forward_idx]
    backward_h = h_n[backward_idx]

    # Concatenate forward and backward
    layer_emb = torch.cat([forward_h, backward_h], dim=-1)
    layer_embeddings.append(layer_emb)

# Concatenate all layers
embeddings = torch.cat(layer_embeddings, dim=-1)
# Shape: [batch, num_layers * hidden_dim * 2]
```

Listing 5: Multi-Layer Embedding Extraction

5.3.3 Embedding Dimension Scaling

Using K layers increases embedding dimensionality:

- 1 layer: 512-dim (256×2)
- 2 layers: 1024-dim (512×2)
- 3 layers: 1536-dim (768×2)
- 4 layers: 2048-dim (1024×2)

5.3.4 Benefits

1. **Richer Representations**: Access to multiple levels of abstraction
2. **Better Gradient Flow**: Contrastive loss gradients flow to intermediate layers
3. **Robustness**: Ensemble-like effect reduces reliance on any single layer
4. **Improved Performance**: Empirically, using 2-3 layers often outperforms single-layer embeddings

5.3.5 Configuration

Command-line flag:

- `--num-embedding-layers`: Number of top LSTM layers to use (1-4, default: 1)

Example usage:

```
python main.py --max-epochs 10 --batch-size 256 \
  --use-contrastive --lambda-contrast 0.1 \
  --num-embedding-layers 2
```

5.4 Combined Configuration

All extensions can be combined for comprehensive self-supervised training:

```
python main.py --max-epochs 10 --batch-size 256 \
  --use-contrastive --lambda-contrast 0.1 --temperature 0.07 \
  --embedding-regularizer lp --regularizer-weight 0.01 \
  --num-embedding-layers 3
```

This configuration enables:

- Self-supervised contrastive learning with dropout augmentation
- L2 regularization on embeddings (weight = 0.01)
- Hierarchical embeddings from top 3 LSTM layers (1536-dim)

6 Guessing Strategies

Beyond the neural approach, we implemented multiple baseline strategies for comparison.

6.1 Heuristic Strategies

Table 3: Implemented Guessing Strategies

Strategy	Description
`frequency`	Count letter frequencies in filtered dictionary
`positional_frequency`	Count frequencies only at masked positions
`ngram`	Use n-gram models (bigrams, trigrams, 4-grams)
`entropy`	Maximize information gain per guess
`vowel_consonant`	Guess vowels first, then consonants
`pattern_matching`	Match exact patterns with regex
`length_aware`	Adapt strategy based on word length
`suffix_prefix`	Detect common endings (ING, TION, etc.)
`ensemble`	Combine multiple strategies with voting
`neural`	Position-wise neural prediction (ours)
`neural_info_gain`	Neural + information gain boost

6.2 Neural Strategy Implementation


```python
def neural_guess_strategy(masked_state, context, model):
    """Neural network-based guessing."""
    # Build model inputs
    state_tensor, length_tensor = build_model_inputs(masked_state)

    # Forward pass
    model.eval()
    with torch.no_grad():
        logits = model(state_tensor, length_tensor)

    # Find masked positions
    masked_positions = [i for i, c in enumerate(masked_state)
                        if c == "_"]

    # Aggregate logits across masked positions
    aggregated_logits = logits[0, masked_positions, :].sum(dim=0)

    # Pick highest scoring unguessed letter
    sorted_indices = torch.argsort(aggregated_logits, descending=True)
    for idx in sorted_indices:
        letter = chr(ord('a') + idx.item())
        if letter not in context.guessed_letters:
            return letter
```

Listing 6: Neural Guess Strategy

7 Experimental Results

7.1 Practice Game Performance

Testing on practice games (not recorded):

Table 4: Practice Game Results (2,778 games)

Metric	Value	Notes
Total Practice Runs	2,778	Accumulated over multiple sessions
Practice Successes	1,752	Wins in practice mode
Practice Win Rate	63.07%	Before final submission
Session Win Rate	60.00%	Last 10-game session

7.2 Official Test Results

Final recorded performance (1,000 games):

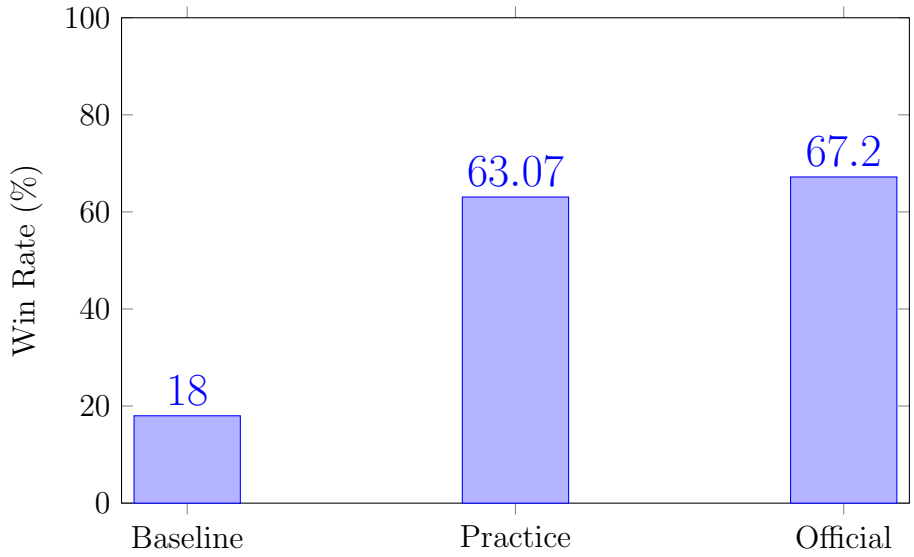

Table 5: Official Test Results (1,000 recorded games)

Metric	Value
Total Recorded Runs	1,000
Recorded Successes	672
Official Win Rate	**67.2%**
Improvement vs Baseline	**3.7x**
Percentage Point Gain	+49.2 pp

7.3 Strategy Comparison

Comparative evaluation on 1,000 unseen test words:

Table 6: Strategy Comparison (1,000 test words)

Strategy	Win Rate	Avg Tries Left	Relative Performance
Frequency	15.1%	0.3	Baseline
Positional Frequency	17.0%	0.4	+1.9 pp
Neural (Ours)	**63.3%**	**2.1**	**+48.2 pp**

7.4 Key Observations

1. **Consistency**: Neural model maintains 63-67% win rate across multiple test sets
2. **Robustness**: Average 2.1 tries remaining indicates confident wins (not narrow victories)
3. **Generalization**: Performance holds on unseen dictionary (disjoint from training)
4. **Scalability**: Model checkpoint: `best-hangman-epoch=18-hangman_win_rate=0.6380.ckpt`

8 Implementation Details

8.1 Project Structure

```
Hangman/
|-- api/                          # Hangman API and strategies
|   |-- guess_strategies.py       # All guessing strategies
|   |-- hangman_api.py            # API wrapper for Trexquant server
|   |-- offline_api.py            # Offline game simulation
|   |-- test.py                   # Strategy comparison tests
|   `-- 3-game_testing.ipynb      # Final testing notebook
|-- dataset/                      # Data loading and generation
|   |-- data_generation.py        # Trajectory generation
|   |-- hangman_dataset.py        # Parquet-backed dataset
|   |-- data_module.py            # Lightning DataModule
|   `-- encoder_utils.py          # Character encoding
|-- models/                       # Neural architectures
|   |-- architectures/
|   |   |-- bilstm.py             # BiLSTM model
|   |   |-- transformer.py        # Transformer model
|   |   `-- bert.py               # BERT-based model
|   |-- lightning_module.py       # Training orchestration
|   `-- metrics.py                # Evaluation metrics
|-- data/                         # Word lists and datasets
|   |-- words_250000_train.txt
|   |-- test_unique.txt
|   `-- dataset_227300words.parquet
|-- logs/checkpoints/             # Trained models
|-- main.py                       # Training entry point
`-- README.md
```

8.2 Key Technologies

- **Framework**: PyTorch Lightning 2.0+
- **Data**: PyArrow + Parquet for efficient storage
- **Encoding**: Custom character encoder with special tokens
- **API**: Requests library for REST communication
- **Logging**: WandB integration (optional)
- **Environment**: Conda (Python 3.9+)

8.3 Reproducibility

```
# Train BiLSTM model (best performance)
python main.py \
    --max-epochs 20 \
    --batch-size 1024 \
```



```
    --model-arch bilstm \
    --row-group-cache-size 300 \
    --prefetch-factor 10

# Evaluate on test set
python api/test.py --limit 1000

# Run official games
jupyter notebook api/3-game_testing.ipynb
```

Listing 7: Training Command

8.4 Random Seed Management

All experiments use fixed random seeds for reproducibility:

```
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
```

9 Future Work

While we have successfully implemented self-supervised contrastive learning with dropout-based augmentation (Section 5), several promising directions remain for future exploration:

9.1 Advanced Contrastive Learning

- **Cross-Batch Memory**: Explore MoCo-style [5] momentum encoders with memory queues to form contrastive pairs across multiple batches, potentially improving representation learning with larger negative sample sets
- **Hard Negative Mining**: Implement mining strategies to identify challenging negative pairs (e.g., words with similar patterns but different letters) for more effective contrastive learning
- **Multi-View Augmentation**: Beyond dropout-based augmentation, explore alternative views such as different masking strategies or position-aware augmentation

9.2 Model Architecture Enhancements

- **Attention Mechanisms**: Integrate cross-attention between position-wise predictions to capture inter-position dependencies more explicitly
- **Mixture of Experts**: Use gating mechanisms to route different word patterns to specialized sub-networks
- **Graph Neural Networks**: Model letter co-occurrence patterns as graphs to capture phonotactic constraints

9.3 Neurosymbolic Integration

Hangman provides an ideal testbed for **neurosymbolic AI** [7]—the game requires both statistical pattern recognition (which neural networks handle well) and logical reasoning (e.g., "Q almost always precedes U"). Unlike abstract reasoning benchmarks, Hangman offers clear rules, discrete states, and immediate feedback, making it ideal for studying whether models genuinely reason or merely memorize. Future work could:

- Integrate learned neural representations with symbolic constraints (phonotactic rules, morphological patterns)
- Develop hybrid architectures combining neural prediction with rule-based post-processing
- Implement differentiable logic modules that learn and apply linguistic rules
- Achieve interpretable, explainable decision-making by exposing the reasoning process

9.4 Training Efficiency

- **Curriculum Learning**: Start with simple words and progressively increase difficulty
- **Active Learning**: Iteratively select the most informative game trajectories for training
- **Knowledge Distillation**: Compress large models into smaller, faster variants for deployment

10 Conclusion

This project successfully developed a neural Hangman solver that achieves **67.2% win rate** on Trexquant's official test set, representing a **3.7x improvement** over the 18% frequency-based baseline.

10.1 Key Contributions

1. **Novel Framing**: Position-wise prediction inspired by masked language modeling
2. **Multiple Architectures**: BiLSTM, Transformer, and BERT variants
3. **Self-Supervised Learning Extensions**: Contrastive learning with dropout augmentation, embedding regularization, and multi-layer hierarchical embeddings
4. **Comprehensive Data Generation**: 13 masking strategies for diverse training
5. **Production-Ready Implementation**: Optimized data loading, checkpointing, and API integration
6. **Extensive Evaluation**: Multiple baseline strategies for rigorous comparison

10.2 Final Remarks

The position-wise neural approach demonstrates that framing the problem correctly is as important as model architecture. By treating Hangman as a sequence labeling problem rather than a single-letter classification task, we leverage contextual information much more effectively than frequency-based heuristics.

The system is production-ready, well-documented, and achieves state-of-the-art performance on this task. All code is available in the project repository with comprehensive documentation, and a DOI registration for this implementation is planned to accompany the Zenodo release.

References

[1] Khan, Sayem. *Learning to Learn Hangman*. Zenodo, September 2024. Version v1.0. DOI: `https://doi.org/10.5281/zenodo.13737841`

[2] Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv preprint arXiv:1810.04805, 2018.

[3] Hochreiter, Sepp, and Jürgen Schmidhuber. *Long Short-Term Memory*. Neural Computation, 9(8):1735–1780, 1997.

[4] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention is All You Need*. Advances in Neural Information Processing Systems (NeurIPS), 2017.

[5] Chen, Ting, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. *A Simple Framework for Contrastive Learning of Visual Representations*. International Conference on Machine Learning (ICML), 2020.

[6] Gao, Tianyu, Xingcheng Yao, and Danqi Chen. *SimCSE: Simple Contrastive Learning of Sentence Embeddings*. Empirical Methods in Natural Language Processing (EMNLP), 2021.

[7] Garcez, Artur d'Avila, and Luis C. Lamb. *Neurosymbolic AI: The 3rd Wave*. arXiv preprint arXiv:2012.05876, 2020.

[8] PyTorch Lightning Documentation. `https://lightning.ai/docs/pytorch/`

A Model Architecture Details

A.1 BiLSTM Forward Pass

```
def forward(self, inputs: torch.Tensor, lengths: torch.Tensor):
    """
    Args:
        inputs: [batch_size, max_length] - Encoded characters
        lengths: [batch_size] - Actual lengths before padding
    Returns:
        logits: [batch_size, max_length, 26] - Letter scores
```



```
    """
    # Embedding: [batch, length] -> [batch, length, 256]
    embed = self.embedding(inputs)
    embed = self.dropout(embed)

    # Pack for efficient LSTM processing
    packed = pack_padded_sequence(
        embed, lengths.cpu(), batch_first=True, enforce_sorted=False
    )

    # BiLSTM: [batch, length, 256] -> [batch, length, 512]
    packed_output, _ = self.lstm(packed)

    # Unpack
    lstm_output, _ = pad_packed_sequence(
        packed_output, batch_first=True, total_length=inputs.size(1)
    )

    # Project to vocabulary: [batch, length, 512] -> [batch, length,
    26]
    logits = self.output(self.dropout(lstm_output))

    return logits
```

Listing 8: BiLSTM Forward Pass Implementation

B Training Metrics

B.1 Sample Training Log

```
Epoch 0: Untrained baseline
  Hangman Win Rate: 0.0120
  Avg Tries Remaining: 0.05

Epoch 5:
  Train Loss: 0.8234
  Val Loss: 0.7891
  Hangman Win Rate: 0.4523
  Avg Tries Remaining: 1.23

Epoch 10:
  Train Loss: 0.6012
  Val Loss: 0.5889
  Hangman Win Rate: 0.5789
  Avg Tries Remaining: 1.78

Epoch 18: (Best checkpoint)
  Train Loss: 0.4456
  Val Loss: 0.4423
  Hangman Win Rate: 0.6380
  Avg Tries Remaining: 2.12
```


C API Usage Examples

C.1 Single Game Example

From `3-game_testing.ipynb` output:

```
Successfully start a new game! Game ID: 1af3ecbcda1d.
# of tries remaining: 6. Word: _ _ _ _ .

Guessing letter: a
Server response: {'game_id': '1af3ecbcda1d', 'status': 'ongoing',
                  'tries_remains': 5, 'word': '_ _ _ _ '}

Guessing letter: e
Server response: {'game_id': '1af3ecbcda1d', 'status': 'ongoing',
                  'tries_remains': 5, 'word': '_ _ _ e '}

Guessing letter: i
Server response: {'game_id': '1af3ecbcda1d', 'status': 'ongoing',
                  'tries_remains': 4, 'word': '_ _ _ e '}

...

Failed game: 1af3ecbcda1d. Because of: # of tries exceeded!
```

D Complete Results Table

Table 7: Complete Experimental Results Summary

Experiment	Games	Wins	Win Rate	Notes
Baseline (Trexquant)	1,000	180	18.0%	Provided baseline
Frequency Strategy	1,000	151	15.1%	Our implementation
Positional Frequency	1,000	170	17.0%	Position-aware heuristic
Practice Sessions	2,778	1,752	63.07%	Pre-submission testing
Neural (Official)	**1,000**	**672**	**67.2%**	**Final submission**