Trexquant Hangman Solver Report

Hangman Solver with BiLSTM Neural Architecture

Name : Hardik Mittal

Email Id : hardik.mittal@research.iiit.ac.in

Phone number : +91 7452804267

Final test success rate : 0.601

Summary

This report presents a model for solving the Hangman word-guessing game using deep learning techniques. Through extensive experimentation with multiple neural architectures and optimization strategies, I finally developed a solution based on a Bidirectional Long Short-Term Memory (BiLSTM) neural network enhanced with positional encoding, context-aware inference, and character-level embeddings. The system achieves really good win rates by making letter predictions in partially revealed words, effectively balancing linguistic knowledge with game-specific constraints and strategies.

Architecture Selection Process

Explored Architectures

I evaluated multiple neural architectures to determine the optimal approach:

1. **Transformer Variants**:
   - Standard encoder-only transformer with self-attention
   - Transformer with reduced head count for efficiency
   - Lightweight transformer with linear attention mechanisms
2. **Recurrent Architectures**:
   - Unidirectional LSTM with varying depths
   - GRU-based models with different hidden dimensions
   - BiLSTM with various configurations
3. **Hybrid Approaches**:
   - CNN + LSTM for character-level feature extraction
   - Transformer encoder with LSTM decoder
   - Attention-augmented LSTM networks

After rigorous comparison, the BiLSTM architecture was showing the best performance for several reasons:

- Better handling of variable-length sequences compared to transformer models
- More efficient capture of bidirectional context critical for character prediction
- Lower computational overhead while maintaining high accuracy
- Superior handling of the specific contextual patterns relevant to Hangman

Final Architecture Details

The optimized BiLSTM architecture includes:

```
ImprovedHangmanPredictor(
    input_size=28,              # Vocabulary size (26 letters + mask token + padding)
    hidden_size=256,            # Increased from initial 128 for better representation
    output_size=28,             # Match input size for character prediction
    max_seq_length=20,          # Maximum word length handled
    embedding_dim=100,          # Character embedding dimension
    pretrained_embeddings=...,  # FastText-derived character embeddings
    num_layers=3,               # Increased from 2 layers for deeper representations
    dropout=0.2                 # Regularization to prevent overfitting
)
```

Technical Implementation Details

1. Embedding System with FastText Integration

The embedding system leverages pretrained FastText subword embeddings, carefully adapted for character-level tasks:

```
# Try to find embeddings for each character in our vocabulary
for char, idx in vocab.items():
    # Skip the mask token
    if char == '_':
        continue

    # Try exact match first
    if char in embeddings_dict:
        embedding_matrix[idx] = embeddings_dict[char]
        found_count += 1
    else:
        # If character not found, try to find similar subwords
        char_embedding = np.zeros(embedding_dim)
        found_similar = False
        similar_count = 0
```

```
        # Check for subwords containing this character
        # Limit to first 1000 words to avoid excessive computation
        for word, vec in list(embeddings_dict.items())[:1000]:
            if char in word:
                char_embedding += vec
                found_similar = True
                similar_count += 1
                # Stop after finding a few examples to save time
                if similar_count >= 10:
                    break

        if found_similar:
            embedding_matrix[idx] = char_embedding / np.linalg.norm(char_embedding)
            found_count += 1
        else:
            # Random initialization for unknown characters
            embedding_matrix[idx] = np.random.normal(0, 0.1, embedding_dim)
```

The model also includes a custom synthetic embedding generator for when pretrained vectors are unavailable:

```
def create_synthetic_char_embeddings(vocab, embedding_dim=100):
    """Create synthetic character embeddings based on character properties"""

    # For each character in our vocabulary
    for char, idx in vocab.items():
        if char == '_':  # Special mask token
            embedding_matrix[idx] = np.random.normal(0, 0.01, embedding_dim)
        else:
            # Create a positionally-encoded embedding
            embedding = np.zeros(embedding_dim)

            # Set specific dimensions based on character properties
            if char.isalpha():
                alphabet_pos = (ord(char.lower()) - ord('a')) / 26.0  # 0 to 1

                # First region: vowel vs consonant pattern
                if char.lower() in 'aeiou':
                    embedding[:region_size] = np.sin(np.arange(region_size) * (alphabet_pos + 0.5))
                else:
                    embedding[:region_size] = np.cos(np.arange(region_size) * (alphabet_pos + 0.5))

                # Additional regions encode other linguistic properties...
```

This embedding approach provides crucial linguistic knowledge to the model before training even begins, encoding:

- Character identity information
- Vowel/consonant distinctions
- Alphabetical position information
- Uppercase/lowercase distinctions
- Common n-gram patterns (through FastText subword information)

2. Positional Encoding Mechanism

The positional encoding system is made following the transformer architecture pattern customized for the Hangman task:

```
def _create_positional_encoding(self, max_seq_length, d_model):
    """Create positional encoding matrix for sequence positions."""
    pe = torch.zeros(max_seq_length, d_model)
    position = torch.arange(0, max_seq_length, dtype=torch.float).unsqueeze(1)
    div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))

    pe[:, 0::2] = torch.sin(position * div_term)
    if d_model % 2 == 0:  # Handle both even and odd dimensions
        pe[:, 1::2] = torch.cos(position * div_term)
    else:
        pe[:, 1::2] = torch.cos(position * div_term[:-(d_model//2)])

    return pe.unsqueeze(0)  # Add batch dimension (1, max_seq_length, d_model)
```

This encoding is particularly valuable for Hangman because:

1. It allows the model to distinguish between the same character in different positions
2. It helps capture position-dependent letter distributions (e.g., 'e' at word endings)
3. It provides absolute positional context for masked tokens
4. The sinusoidal pattern enables smooth interpolation for positions not seen during training

3. Masking Strategies for Training

The training process employs sophisticated masking strategies to simulate realistic Hangman game states:

```
def _create_samples(self):
    samples = []
    for word in self.words:
        for _ in range(self.num_samples_per_word):
            # Create a masked version of the word
            masked_word = list(word)
            mask_positions = []

            # Choose random positions to mask
            indices = list(range(len(word)))
```

```
            random.shuffle(indices)
            num_to_mask = max(1, int(len(word) * self.mask_prob))
            mask_indices = indices[:num_to_mask]

            # Apply masking
            for idx in mask_indices:
                masked_word[idx] = self.mask_token
                mask_positions.append(idx)

            samples.append((word, ''.join(masked_word), mask_positions))
```

I experimented with multiple masking approaches including:

1. **Random Masking**: As implemented above, with 40% masking probability
2. **Frequency-Based Masking**: Higher probability of masking common letters
3. **Position-Based Masking**: Different masking rates for different positions
4. **Pattern-Based Masking**: Simulating real game progression by revealing high-frequency letters first
5. **Curriculum Learning**: Starting with fewer masked characters and increasing difficulty

The random masking with 40% probability was selected for the final model as it provided the best generalization across different word types.

4. Context-Aware Inference Algorithm

```
# Weight the predictions based on context
masked_probs = []
weights = []

for pos in masked_positions:
    # Default weight
    weight = 1.0

    # Consider left context
    if pos > 0:
        if pattern[pos-1] != '_':  # Adjacent to known letter
            weight += 0.5
            # Check for known patterns
            if pattern[pos-1] == 'q':  # 'q' is almost always followed by 'u'
                weight += 1.5

    # Consider right context
    if pos < len(pattern)-1:
        if pattern[pos+1] != '_':  # Adjacent to known letter
            weight += 0.5

    # Extra weight for isolated gaps (highly constrained)
    if (pos > 0 and pattern[pos-1] != '_' and
```

```
            pos < len(pattern)-1 and pattern[pos+1] != '_'):
            weight += 1.0

        # Apply position-specific weighting
        if pos == 0:  # First letter
            weight += 0.25
        elif pos == len(pattern)-1:  # Last letter
            weight += 0.25

        # Apply the weight to this position's probabilities
        masked_probs.append(pos_probs[pos] * weight)
        weights.append(weight)
```

The weighting system incorporates:

1. **Contextual Adjacency**: Higher weights for positions next to revealed letters
2. **Positional Constraints**: Special handling for word beginnings and endings
3. **Gap Isolation Analysis**: Highest priority for letters in isolated gaps between known letters
4. **Pattern Recognition**: Linguistic rules like "u" after "q" receive elevated weights
5. **Position-Specific Letter Statistics**: Leverages knowledge of common letters at specific positions

These weighted probabilities are then aggregated, filtered to remove already guessed letters, and the highest probability letter is selected as the next guess.

5. Comprehensive Linguistic Knowledge Integration

The system incorporates extensive linguistic knowledge through multiple mechanisms:

1. **Letter Frequency Distributions**:

```
ENGLISH_LETTER_FREQ = {
    'e': 12.02, 't': 9.10, 'a': 8.12, 'o': 7.68, 'i': 7.31, 'n': 6.95, 's': 6.28,
    'r': 6.02, 'h': 5.92, 'd': 4.32, 'l': 3.98, 'u': 2.88, 'c': 2.71, 'm': 2.61,
    'f': 2.30, 'y': 2.11, 'w': 2.09, 'g': 2.03, 'p': 1.82, 'b': 1.49, 'v': 1.11,
    'k': 0.69, 'x': 0.17, 'q': 0.11, 'j': 0.10, 'z': 0.07
}
```

2. **Position-Specific Letter Distributions**:

```
POSITION_LETTER_FREQ = {
    'start': {  # First letter frequencies
        't': 16.0, 's': 11.3, 'a': 8.5, 'c': 8.1, 'p': 7.1, 'b': 6.0, 'm': 5.8, ...
    },
    'end': {  # Last letter frequencies
        'e': 16.8, 's': 14.0, 't': 8.7, 'd': 8.5, 'n': 7.6, 'y': 6.7, 'r': 6.6, ...
    }
}
```

3. **Common Letter Patterns**:

```
COMMON_PATTERNS = {
    'q': {'u': 0.95},  # q is almost always followed by u
    'th': {'e': 0.5, 'a': 0.2, 'i': 0.15, 'o': 0.1},
    'ch': {'e': 0.3, 'a': 0.2, 'i': 0.15, 'o': 0.15},
    'sh': {'e': 0.3, 'a': 0.2, 'i': 0.15, 'o': 0.15},
    'ing': {'e': 0.3, 's': 0.2, 'l': 0.1},
    'tion': {'a': 0.3, 's': 0.3, 'e': 0.2},
}
```

This linguistic knowledge serves as both a fallback when the neural model is uncertain and as a bias mechanism to enhance the neural predictions, particularly for word boundaries and special patterns.

Hyperparameter Optimization

After doing an extensive hyperparameter exploration to determine optimal configurations, I got the best performing parameters:

1. **Model Architecture Parameters**:
   - Hidden Layer Size: [64, 128, 256, 512] → Selected 256
   - Number of LSTM Layers: [1, 2, 3, 4] → Selected 3
   - Embedding Dimension: [50, 100, 200, 300] → Selected 100
   - Dropout Rate: [0.1, 0.2, 0.3, 0.5] → Selected 0.2
   - Bidirectional vs. Unidirectional → Selected Bidirectional
2. **Training Parameters**:
   - Batch Size: [32, 64, 128, 256] → Selected 256
   - Learning Rate: [0.0001, 0.0003, 0.001, 0.003] → Selected 0.001
   - Optimizer: [SGD, Adam, RMSprop] → Selected Adam
   - Masking Probability: [0.2, 0.3, 0.4, 0.5] → Selected 0.4
   - Samples Per Word: [1, 3, 5, 10] → Selected 5
3. **Loss Function Analysis**:
   - BCE Loss vs. Cross-Entropy vs. Focal Loss → Selected BCE Loss
   - Weight parameters for loss functions
   - Gradient clipping thresholds

These choices were determined through systematic grid search and manual tuning, with validation performance as the primary metric for selection.

Performance Evaluation and Analysis

Comprehensive Evaluation Metrics

I employed 2 approaches to assess model performance:

1. **Character Prediction Accuracy**:

```
def evaluate_accuracy(model, data_loader, criterion, device):
    # Compute accuracy for masked positions
    for i, positions in enumerate(mask_positions):
        for pos in positions:
            pred_idx = torch.argmax(outputs[i, pos]).item()
            target_idx = torch.argmax(targets[i, pos]).item()
            correct_predictions += (pred_idx == target_idx)
            total_predictions += 1

    accuracy = correct_predictions / max(1, total_predictions)
```

2. **Hangman Game Simulation**:

```
def play_sample_games(model, val_words, char_to_idx, idx_to_char, device, num_samples=
1000):
    # Track metrics from 1000 simulated games
    results = []
    for word in sample_words:
        result = simulate_hangman_game(model, word, char_to_idx, idx_to_char, device)
        results.append(result)

    wins = sum(1 for game in game_results if game['win'])
    win_rate = wins / len(game_results)
    avg_wrong = sum(game['wrong_guesses'] for game in game_results) / len(game_results)
```

Conclusion

The BiLSTM-based Hangman solver demonstrates the power of combining neural sequence modeling with linguistic knowledge and game-specific heuristics. Through careful architecture design, extensive hyperparameter optimization, and innovative inference strategies, I have developed a system that achieves high win rates even on previously unseen words.

The key insight from this research is that while modern architectures like transformers offer powerful sequence modeling capabilities, the specific nature of the Hangman problem makes a well-optimized BiLSTM with appropriate contextual weighting strategies the most effective solution in terms of both performance and efficiency.