Trexquant Task Report

BiLSTM Hangman Solver

> Name: Aanvik Bhatnagar
> Email: aanvik26@gmail.com
> Contact Number: +91 9205376780

1. Summary

This document outlines the architecture and methodology of a ML-based Hangman solver. The objective is to create a program that can accurately and efficiently guess the letters of a hidden word in the game of Hangman.

The core of the solution is a **hybrid strategy** that combines the predictive power of a sophisticated deep learning model with the reliability of a traditional dictionary-based frequency analysis. A **Bidirectional Long Short-Term Memory (BiLSTM) network** serves as the primary guessing engine, trained to understand character patterns and context within words. This is supplemented by a robust fallback mechanism that uses regular expressions and statistical analysis of a word corpus when the model's confidence is low or its prediction is invalid.

The entire project is modular, configurable, and built using modern deep learning best practices to ensure stable training and high performance. The final system demonstrates a powerful synergy between learned contextual patterns and rule-based linguistic knowledge.

The final accuracy that the model achieved on the non-practice dataset is: 0.624

2. Architecture Details

The project is structured into several distinct Python modules, each with a specific responsibility. This modularity makes the codebase clean, maintainable, and easy to extend. All parameters and paths are externalized into a central configuration file.

- `config.json`: A central JSON file that stores all hyperparameters, file paths, and training settings. This allows for easy experimentation without modifying the source code.
- `train.py`: The main executable script. It handles data loading, model initialization, the training loop, and the final performance evaluation.
- `utils.py`: A collection of helper functions for tasks like loading the word dictionary, creating character mappings, downloading embeddings, and saving/loading the model.
- `data_loader.py`: Contains the PyTorch `Dataset` class responsible for preparing the data for training using a masked language modeling approach.
- `model.py`: Defines the neural network architecture, including all its layers and components.

Deep Learning Model Breakdown (`model.py`)

The heart of the solver is the `HangmanGuesserNetwork`, a deep learning model designed to capture sequential patterns in text. Data flows through the network in the following sequence:

1. **Embedding Layer (`token_embedding`)**: This is the first layer. It takes the integer representation of each character and converts it into a dense vector of a fixed size (`vector_dimension`). This vector is a learned representation of the character, capturing its semantic properties.
2. **Positional Encoder (`SequencePositionEncoder`)**: The meaning of a character depends on its position. For example, the 's' in "show" is different from the 's' in "wash". This layer adds information about the position of each character to its vector representation, allowing the model to distinguish between them using sine and cosine functions of different frequencies.
3. **Bidirectional LSTM (`recurrent_layer`)**: The core of the model consists of multiple layers of a Long Short-Term Memory (LSTM) network.
    - **LSTM**: A type of recurrent neural network (RNN) that is excellent at learning from sequential data. It can remember information over long sequences using internal "gates," which is crucial for understanding the context of a whole word.

- **Bidirectional**: Our LSTM reads the word from both left-to-right and right-to-left. This gives it a complete contextual understanding of every character. For a word like `_ppl_`, to guess the first letter, it's helpful to know that the word ends with `l`.

4. **Feed-Forward Network and Skip Connections**: Following the LSTM layers, we use components inspired by modern Transformer architectures (`PointwiseFeedForward`, `SkipConnection`). The `PointwiseFeedForward` network applies two linear transformations with a ReLU activation in between, allowing the model to process the information from the LSTM in a more complex way. The `SkipConnection` (or residual connection) adds the input of a layer to its output, which helps prevent the vanishing gradient problem and allows for deeper, more stable networks.
5. **Output Projection (`OutputProjection`)**: The final layer takes the processed information from the LSTM and projects it back into the space of our vocabulary. For each position in the word, it outputs a raw score (a logit) for every possible character in the alphabet. The higher the score, the more confident the model is that this character belongs in that position. The `Softmax` function is intentionally omitted here, as the `CrossEntropyLoss` function used during training is optimized to work directly with raw logits.

3. Data Preparation and Loading

The model is trained on a large list of English words. The raw data is processed into a format suitable for training a neural network.

`utils.py` - Core Utilities

- `fetch_word_list()`: Reads the `words_250000_train.txt` file, converts all words to lowercase, and filters them to ensure they are of a reasonable length (3-15 characters) and contain only alphabetic characters.
- `generate_token_maps()`: Creates a vocabulary of all unique characters in the dataset and builds two dictionaries: one mapping each character to a unique integer (`token_to_int`) and a reverse map (`int_to_token`). This is essential for feeding character data into the network.

`data_loader.py` - Training Sample Generation

- `HangmanDataset`: This class implements the core training strategy. Instead of just showing the model complete words, we use a **masked language modeling** approach. For each word in our training set, we create multiple training samples by randomly masking some of its letters with an underscore (`_`).
  - **Input**: The word with masked characters (e.g., `h_ngm_n`).
  - **Target**: The original characters that were masked (e.g., `a`, `a`).
  - This technique forces the model to learn the contextual relationships between characters to predict the missing ones, which is exactly the task in Hangman.
- `batching_processor()`: Since words have different lengths, this function pads the sequences in each batch to the same length. This is a requirement for efficient processing on a GPU.

4. Training Methodology (`train.py`)

The training process is designed to be robust and efficient.

- **Data Split**: The word list is split into a **90% training set** and a **10% testing set**. The model learns from the training set, and its final performance is measured on the unseen testing set.
- `execute_training_cycle()`: This function contains the main training loop.
  - **Loss Function (`CrossEntropyLoss`)**: This function measures the difference between the model's predictions (logits) and the actual target characters. The goal of training is to minimize this loss.
  - **Optimizer (`Adam`)**: An advanced optimization algorithm that efficiently updates the model's internal parameters (weights) to reduce the loss.
  - **Gradient Clipping**: A technique used to prevent the gradients (the signals used to update the weights) from becoming too large, which can destabilize training. A `max_norm` of 1.0 is used.
  - **Learning Rate Scheduler (`StepLR`)**: This gradually reduces the learning rate during training. It starts with larger steps to learn quickly and then reduces the step size to make finer adjustments as the model converges on a

solution. The learning rate is prevented from decaying below a `min_learning_rate` of `1e-4`.

- `persist_checkpoint()`: After training is complete, the final state of the model is saved to the file specified in `config.json`.

5. Algorithm Details: The Hybrid Guessing Strategy

A key innovation of this solver is its hybrid guessing strategy, which ensures both high accuracy and robustness by combining two distinct methods in a sequential, two-stage process.

Stage 1: Primary Guesser (`model_guess`)

The first attempt to find a letter is always made by the trained neural network.

1. The current state of the word (e.g., `_ p p _ _`) is pre-processed and fed into the trained `HangmanGuesserNetwork`.
2. The model outputs scores (logits) for all possible characters for each of the blank positions.
3. The scores for all blank positions are aggregated (summed) to get a total score for each character in the alphabet.
4. A small, rule-based **semantic boost** is applied (see section 6).
5. The function identifies the character with the highest final score that has not already been guessed. This character is the model's primary prediction.

Stage 2: Fallback Guesser (`dictionary_guess`)

This method is invoked **only if** the primary model fails to produce a valid guess (e.g., it returns a letter that has already been tried, or the model itself failed to load).

1. The current word pattern is used to create a **regular expression** (e.g., `_pp__` becomes `.pp..`).
2. The entire dictionary is filtered to find all words that perfectly match this pattern in terms of both length and known characters.

3. The frequency of each letter within this new, much smaller, filtered list of words is calculated.
4. The function returns the most common letter from this filtered list that has not yet been guessed.
5. If the filtered list is empty (a rare case), it falls back one more time to the letter frequency of the entire dictionary.

This hybrid approach leverages the model's learned contextual understanding while using the dictionary as a deterministic safety net, making the solver very difficult to beat.

6. English Semantics Integration

While the deep learning model learns complex patterns, we can further improve its accuracy by injecting simple, high-confidence linguistic rules.

The current implementation includes a crucial semantic rule for English: the letter **'q' is almost invariably followed by 'u'**.

- **Implementation**: In the `model_guess` function, before determining the final guess, the code checks if a 'q' is visible in the word and if the following position is a blank. If so, it applies a large **semantic boost** to the score for the letter 'u'. This makes it extremely likely that 'u' will be the next guess, aligning the model's probabilistic output with a near-certain linguistic rule.

7. Performance Evaluation and Analysis

The model's effectiveness is not judged on its training loss, but on its practical ability to win the game of Hangman.

- **Methodology (`assess_network_performance`)**: After training, the script runs a comprehensive evaluation. It takes the unseen test set of words and simulates a full Hangman game for each one using the hybrid guessing strategy.
- **Key Metrics**: The performance is quantified by two primary metrics:
    1. **Success Ratio**: The percentage of games in the test set that the solver won (i.e., guessed the word before running out of tries). This is the most important measure of the solver's overall effectiveness.

2. **Average Errors per Game**: The average number of incorrect guesses made per game. A lower number indicates a more efficient solver.

These metrics provide a clear and interpretable measure of the model's real-world performance on the Hangman task.

8. Configuration

All important parameters are managed in `config.json` for easy modification.

- **`paths`**: Specifies locations for the word list and the saved model checkpoint.
- **`model_params`**: Contains the architectural hyperparameters for the neural network, such as `vector_dimension`, `recurrent_units`, `recurrent_layers`, and `dropout_rate`.
- **`training_params`**: Controls the training process, including `chunk_size` (batch size), `training_cycles` (epochs), `initial_learning_rate`, and `min_learning_rate`.
- **`evaluation_params`**: Defines settings for the performance simulation, such as the `game_count` for validation checks.

9. Execution Instructions

The project is executed from the command line. The main script `train.py` accepts several arguments to control its behavior.

Primary Command:

To run the full process: training the model without pre-trained embeddings and then evaluating its performance on the test set, use the following command:

```
python3 train.py --run-test
```

Command-Line Flags:

- `-config-path`: Specifies the path to the `config.json` file (defaults to `config.json`).
- `-run-test`: (Flag) After training is complete, this will run a final performance assessment on the held-out test set.