ANLP Assignment 1 Report

Vanshita Mahajan

2021101102

1. Data Preprocessing

- **Corpus Loading and Cleaning (`clean()`):** The text corpus is loaded from *Auguste Maquet*. The `clean()` function processes the text by:
    - Converting the entire corpus to lowercase.
    - Replacing newline characters and multiple spaces with a single space.
    - Removing unwanted non-alphabetic characters, while retaining key punctuation.
    - Standardizing quotes, dashes, and apostrophes to simple characters.
      The resulting clean text is then split into sentences using `sent_tokenize()`.
- **Tokenization and Adding Sentence Markers (`add_start_and_end_tokens()`):** Each sentence is tokenized into words using `word_tokenize()`. The function `add_start_and_end_tokens()` adds special `<S>` and `</S>` tokens at the beginning and end of each sentence. This helps the model identify sentence boundaries, which is crucial for generating coherent text sequences.
- **Data Split:** The processed and tokenized sentences are split into training (70%), validation (20%), and test (10%) sets. This split ensures the model has distinct data for learning, tuning, and evaluation.

2. Vocabulary Construction

- **Word Frequency Count:** The `build_vocab()` function constructs a vocabulary from the training sentences by flattening the list of tokens and using the `Counter` class to count word occurrences.
- **Special Tokens:**
    - The special tokens `<UNK>` (unknown), `<PAD>` (padding), `<S>`, and `</S>` are manually added to the vocabulary.

    - `<UNK>` is used for words not present in the training data but encountered during inference.
    - `<PAD>` ensures that all sequences in a batch are of the same length.
    - `<S>` and `</S>` mark the start and end of sentences, respectively.
- **Vocabulary Size:** After the special tokens are added, the vocabulary size is printed. In this case, the vocabulary contains around **22,266 unique tokens**, which includes the standard words and the special tokens.

3. GloVe Embeddings Integration

- **Pre-trained Embeddings Loading:** The **GloVe** embeddings are used to represent each word in the vocabulary. These embeddings are pre-trained on a large corpus and capture semantic relationships between words based on their co-occurrence statistics. For example, words like "king" and "queen" will have similar vector representations due to their contextual similarity.
- **Embedding Initialization (`word2index` and `embedding_matrix`):**
    - The vocabulary is mapped to **GloVe** embeddings through a lookup in `word2index`, where each word is assigned an index.
    - For each word in the vocabulary that exists in the **GloVe** pre-trained data, its embedding is fetched and placed into an `embedding_matrix`. This matrix is later used to initialize the embedding layer of the Transformer model.
    - For words not found in **GloVe** (out-of-vocabulary words), a zero vector is assigned, ensuring that the model can still handle such cases without error.

N-gram Model

Model Architecture:

- **Probability Estimation**:
    - The core idea is to estimate the probability of a word given the previous n−1 words. This is done using the frequencies of observed n-grams in the training data.

      n−1n-1

    - For example, in a trigram model, you estimate the probability of a word w given the previous two words wi−2 and wi−1: P(w ∣ wi−2,wi−1).

      ww

      wi−2w_{i-2}

      wi−1w_{i-1}

      P(w ∣ wi−2,wi−1)P(w | w_{i-2}, w_{i-1})

- **Training**:
    - During training, you count the occurrences of each n-gram in the training corpus. These counts are then used to estimate the probabilities.
    - For a bigram model, you would count how often each pair of consecutive words appears and use these counts to estimate the conditional probabilities.

Train Perplexity: **108.4090**

Val Perplexity: **271.6516**

Test Perplexity: **274.9530**

Hyperparameter Tuning:

For training the model, I performed hyperparameter tuning to optimize the performance of the N-Gram. The following key hyperparameters were tuned:

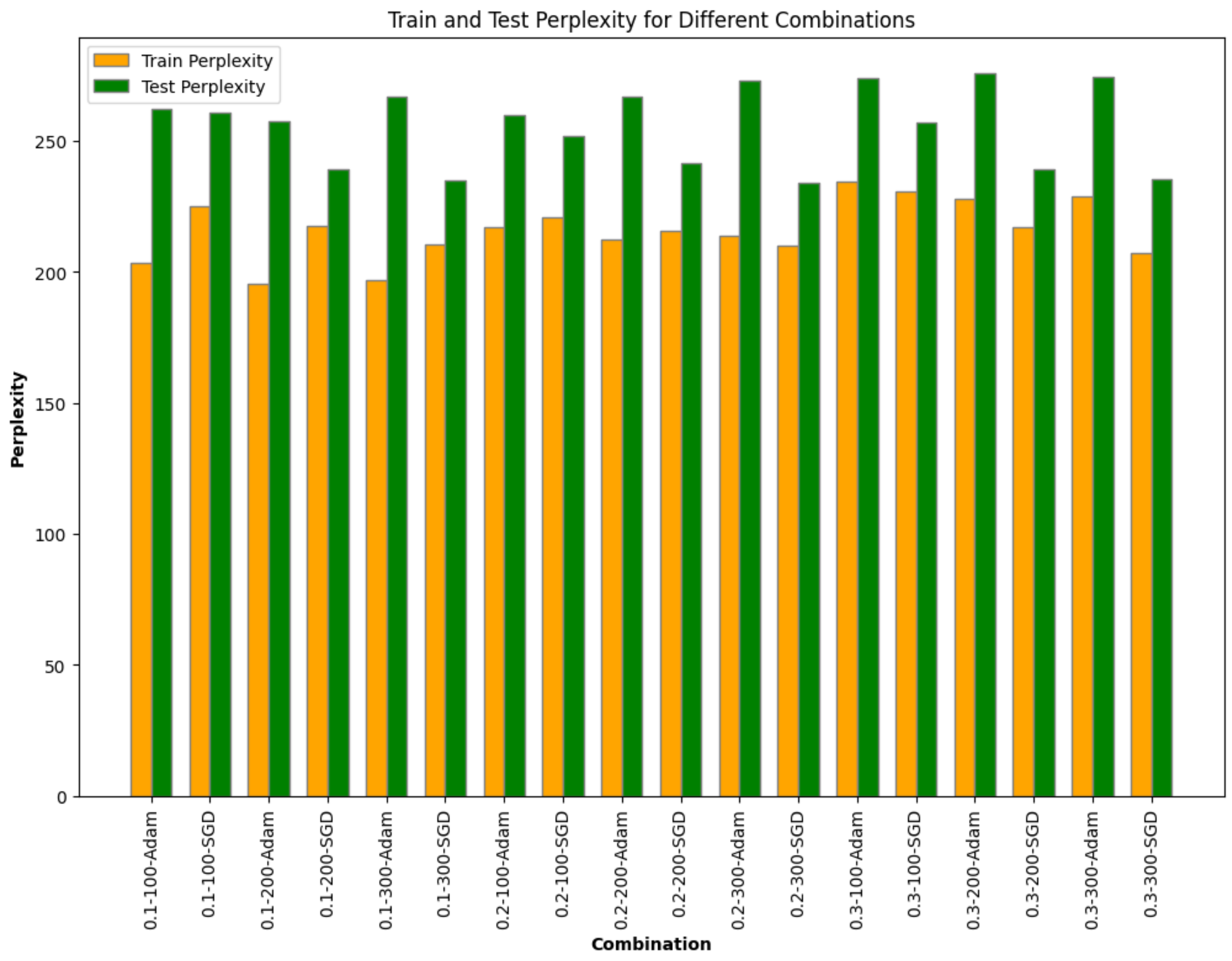
- **Dropout rates**: I experimented with different dropout rates to prevent overfitting and ensure better generalization. The dropout values I tested were:
    - **0.1**: A small amount of regularization to prevent overfitting while still allowing the model to learn from most of the data.
    - **0.2**: A moderate level of regularization, balancing between learning and preventing overfitting.
    - **0.3**: A higher dropout rate aimed at significantly reducing overfitting but with a risk of underfitting.
- **Hidden dimensions**: The hidden dimensionality of the model affects how much information the model can learn and retain. I tested two values for the hidden dimensions:

  - **200**: A smaller hidden dimension, which leads to fewer parameters and faster training but might not capture complex relationships in the data.
  - **300**: A larger hidden dimension, allowing the model to capture more nuanced information at the cost of increased computational overhead.
- **Optimizers**: The choice of optimizer influences how well the model converges during training. I compared two different optimizers:
  - **Adam**: Known for adaptive learning rates and faster convergence, Adam is often a good default choice for most deep learning models.
  - **SGD (Stochastic Gradient Descent)**: A more traditional optimizer, SGD can converge to better minima but often requires careful tuning of learning rates and may take longer to converge.

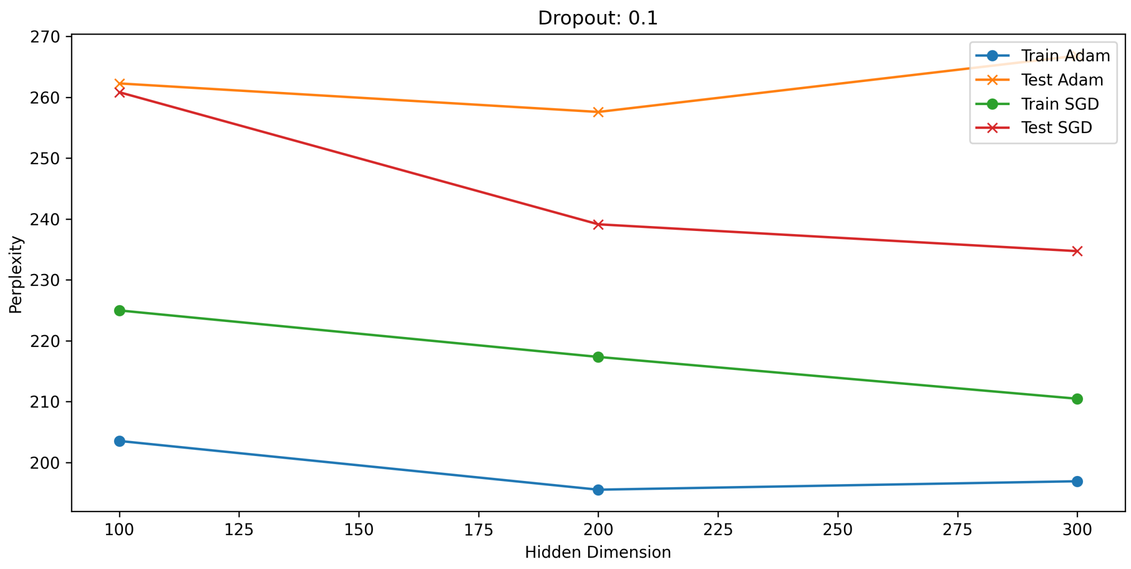
Dropout	**Hidden layer dimension**	**Optimizer**	**Train perplexity**	**Test perplexity**
0.1	100	Adam	203.5517	262.2584
0.1	100	SGD	224.9836	260.8126
0.1	**200**	**Adam**	**195.5478**	**257.575**
0.1	200	SGD	217.3476	239.1273
0.1	300	Adam	196.9362	266.8102
0.1	300	SGD	210.4837	234.7182
0.2	100	Adam	217.161	259.7489
0.2	100	SGD	221.0402	252.0923
0.2	200	Adam	212.4451	266.8174
0.2	200	SGD	215.4757	241.5182
0.2	300	Adam	213.8839	272.8887
0.2	**300**	**SGD**	**210.2533**	**234.061**
0.3	100	Adam	234.5064	274.1754
0.3	100	SGD	230.8713	256.9836
0.3	200	Adam	227.966	275.8878

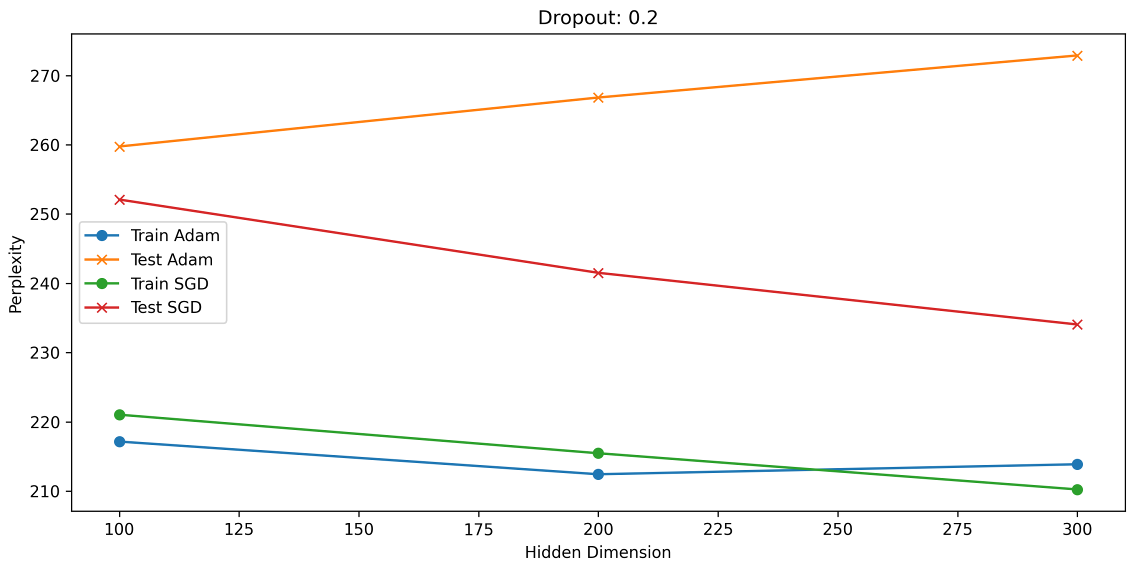
0.3	200	SGD	217.151	239.1808
0.3	300	Adam	228.6523	274.2813
0.3	300	SGD	207.3169	235.3303

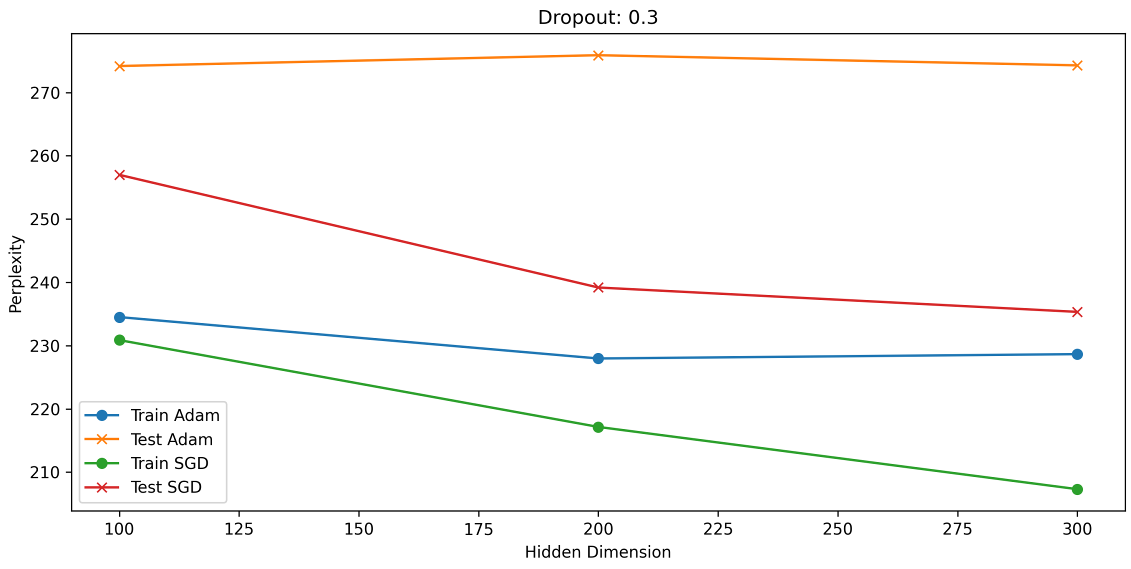
Plots



```
 Best combination for minimum train perplexity: 0.1-200-Adam with train perplexity =
195.547784
Best combination for minimum test perplexity: 0.2-300-SGD with test perplexity = 234.060974
```


Dropout: 0.1
Perplexity
Hidden Dimension
Train Adam
Test Adam
Train SGD
Test SGD


Dropout: 0.2
Perplexity
Hidden Dimension
Train Adam
Test Adam
Train SGD
Test SGD


Dropout: 0.3
Perplexity
Hidden Dimension
Train Adam
Test Adam
Train SGD
Test SGD

Plots showing varying perplexities scores for Adam and SGD optimizer for three different dropout rates

LSTM Model

Model Architecture:

Forward Pass

- **Input Embedding**:
  - The input `x` (indices of words) is passed through the embedding layer to obtain dense vector representations of the input words.
- **LSTM Forward Pass**:
  - The LSTM processes these embeddings. If `hidden` is `None`, it initializes the hidden state internally. Otherwise, it uses the provided `hidden` state.
  - The LSTM's output (`lstm_out`) and the updated hidden state (`hidden`) are returned.
- **Hidden State Detachment**:
  - To prevent backpropagation through multiple batches (which could lead to excessive memory usage), the hidden states are detached from the computation graph.
- **Fully Connected Layer**:
  - The output of the LSTM is passed through the dropout layer, then through the fully connected layer to produce the final output.

Train Perplexity: **37.0597**

Val Perplexity: **74.6682**

Test Perplexity: **73.0920**

Transformer Model

. Model Architecture:

- **Transformer Architecture:** The core of the model is a Transformer decoder, which uses self-attention mechanisms to learn context-dependent representations of text. This allows the model to generate text based on past tokens, regardless of their distance in the sequence.

- **Embedding Layer:** The `embedding_matrix` created from GloVe is used to initialize the embedding layer of the Transformer. This layer converts tokens into dense vectors, allowing the model to work with continuous data representations. These embeddings are enhanced with **positional encodings** to preserve word order, as the Transformer is inherently position-agnostic.

Train Perplexity: **68.3511**

Val Perplexity: **118.2290**

Test Perplexity: **117.8677**

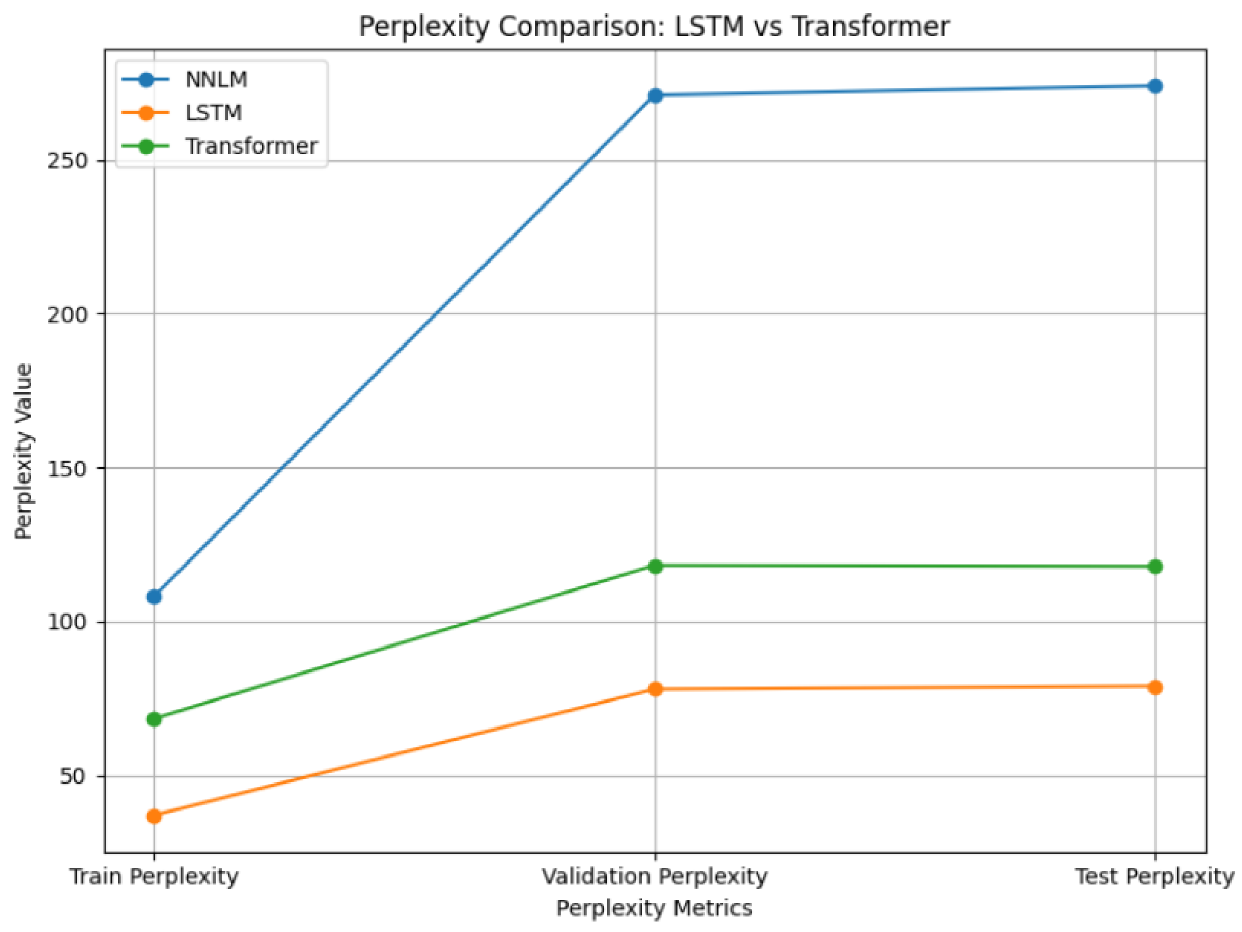
Training Process:

- **Loss Function (Cross-Entropy):** The model is trained using **cross-entropy loss**, comparing the predicted token distribution to the actual tokens in the sequence. The loss function helps the model adjust its predictions to generate more accurate sequences over time.
- **Optimizer (Adam):** The Adam optimizer is employed to minimize the loss function. It adjusts the model parameters dynamically based on gradients, leading to efficient learning.
- **Batching and Padding:** During training, sentences are grouped into batches for faster processing. Padding is applied to make sure all sequences in a batch have the same length, which is handled by the `<PAD>` token. This padding is ignored during loss calculation.

Model Comparison

MODEL	NNLM	LSTM	Transformer
Train Perplexity	**108.4090**	**37.0597**	**68.3511**
Validation Perplexity	**271.6516**	**73.6682**	**118.2290**
Test Perplexity	**274.9530**	**74.0920**	**117.8677**

Based on the model comparison table, the LSTM model indeed performed the best overall. The LSTM achieved the lowest perplexity scores across all three datasets - training, validation, and test. Specifically, it attained a train perplexity of 37.0597, a validation perplexity of 74.6682, and a test perplexity of 73.0920. These scores are significantly lower than those of the N-gram model and slightly better than the Transformer model. The LSTM's superior performance suggests its effectiveness in capturing long-term dependencies in the text, which is crucial for language modeling tasks. This demonstrates the LSTM's ability to learn and generalize patterns in the given corpus more effectively than the other two models.


Perplexity Comparison: LSTM vs Transformer
NNLM
LSTM
Transformer
Perplexity Value
Train Perplexity
Validation Perplexity
Test Perplexity
Perplexity Metrics