Advanced NLP

ASSIGNMENT 1

September 10, 2024

Ashna Dua
2021101072
ashna.dua@students.iiit.ac.in

Contents

1 Neural Network Language Model **3**
- 1.1 About . . . 3
- 1.2 Model Architecture . . . 3
- 1.3 Implementation . . . 3
  - 1.3.1 Pre-processing . . . 3
  - 1.3.2 Tokenization . . . 3
  - 1.3.3 Data Preparation . . . 3
  - 1.3.4 Glove Embeddings and Vocab . . . 4
  - 1.3.5 Custom Dataset for NNLM . . . 4
  - 1.3.6 NNLM Model . . . 4
  - 1.3.7 Evaluation . . . 5
  - 1.3.8 Hyperparameter Tuning . . . 5

2 LSTM-based Neural Language Model **6**
- 2.1 About . . . 6
- 2.2 Model Architecture . . . 7
- 2.3 Implementation . . . 7
  - 2.3.1 Pre-processing . . . 7
  - 2.3.2 Tokenization and Data Preparation . . . 7
  - 2.3.3 Custom Dataset for LSTM . . . 7
  - 2.3.4 LSTM Model . . . 7
  - 2.3.5 Results . . . 8

3 Transformer Decoder based Language Model **8**
- 3.1 About . . . 8
- 3.2 Model Architecture . . . 8
- 3.3 Implementation . . . 9
  - 3.3.1 Pre-processing . . . 9
  - 3.3.2 Tokenization and Data Preparation . . . 9

4 Dataset Preparation **9**
- 4.1 Positional Encoding . . . 9

5 Training Process **9**

6 Results **9**

7 Analysis **10**

1 Neural Network Language Model

1.1 About

Language modeling (LM) is the use of various statistical and probabilistic techniques to determine the probability of a given sequence of words occurring in a sentence. Thus, it may be defined as a probability distribution over the vocabulary words. In traditional N-gram Language Models, the probabilities of words occurring after a given context are defined as: $P(w_n|w_{n-1}, w_{n-2}, ..., w_{n-N})$

This assignment involves creating a Neural Language Model using standard Deep Learning Frameworks with a 5-gram context.

1.2 Model Architecture

The architecture of the Language Model (LM1) is described below:

- The input to the Neural Network would be the pre-trained **Glove** embeddings of the previous 5-words concatenated together(5-gram embedding).
- These embedding would pass into a Hidden Layer which would output a 300-dimension vector.
- The output of the first Hidden Layer would go into another Hidden Layer, which would output a vector of size vocabulary.
- This vector would be passed to a Softmax Layer, which would output the probabilities of entire vocabulary occurring after the given 5-words.
- The vocabulary should consist of all the words that occur any number of times in the training data. Unknown words can be handled with an $< UNK >$ token.

1.3 Implementation

1.3.1 Pre-processing

The pre-processing step involves cleaning the text data by removing special characters, normalizing apostrophes and dashes, and separating punctuation marks for better tokenization. The **preprocess** function, implemented using regular expressions, achieves this.

1.3.2 Tokenization

Tokenization splits the text into meaningful units (words and punctuation). The tokenize function utilizes NLTK's sent_tokenize and word_tokenize to split the text into sentences and further into individual words. It adds start-of-sentence ($< s >$) and end-of-sentence ($< /s >$) markers.

1.3.3 Data Preparation

The **train_val_test_split** function divides the tokenized sentences into training, validation, and test sets according to specified ratios and shuffling.

1.3.4 Glove Embeddings and Vocab

The **create_glove_embeddings** function loads pre-trained GloVe embeddings from a text file. It creates a dictionary mapping words to their corresponding embedding vectors and adds special tokens $< UNK >$ for unknown words, $< s >$ for start-of-sentence, and $< /s >$ for end-of-sentence, with randomly initialized embeddings.

The create_embeddings_and_encode function builds the vocabulary from the training set, creates a mapping (word_to_idx) from words to their indices, and generates embedding matrices. It then encodes the sentences in all sets using the word_to_idx mapping, replacing unknown words with $< UNK >$.

1.3.5 Custom Dataset for NNLM

The NGramDataset class defines a PyTorch Dataset for the NNLM. It creates ngrams (5-word contexts) from the encoded sentences and their corresponding target words (the next word in the sequence), along with their GloVe embeddings.

1.3.6 NNLM Model

The NNLM class defines the neural network model with two hidden layers, a ReLU activation function, and dropout. The embeddings are loaded as a pre-trained embedding layer, frozen to prevent updating their weights during training.

```
class NNLM(nn.Module):
    def __init__(self, embeddings, hidden_dims, n_gram=5, dropout=0.5):
        super(NNLM, self).__init__()

        self.vocab_size = embeddings.shape[0]
        self.embeddings_dim = embeddings.shape[1]

        self.embeddings = nn.Embedding.from_pretrained(embeddings, freeze=True)
        self.fc1 = nn.Linear((self.embeddings_dim) * n_gram, hidden_dims[0])
        self.fc2 = nn.Linear(hidden_dims[0], hidden_dims[1])
        self.fc3 = nn.Linear(hidden_dims[1], self.vocab_size)

        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(p = dropout)

    def forward(self, x):
        x = x.view(x.shape[0], -1)
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.dropout(self.relu(self.fc2(x)))
        return self.fc3(x)
```

The **train_model** function trains the NNLM using the specified optimizer, criterion, and number of epochs. Early stopping is implemented to prevent overfitting. The **test_model** function evaluates the model on a given data loader (train, validation, or test), calculating the average loss and perplexity. The save_perplexities function saves the sentence-level perplexity scores and the average perplexity for each dataset into separate files. The save_model function saves the model's state dictionary for later use.

1.3.7 Evaluation

The model was evaluated using the Perplexity Score. Perplexity measures how well a probability distribution or model predicts a sample, and is calculated as:

$$\text{Perplexity} = 2^{-\frac{1}{N}\sum_{i=1}^{N}\log_2 P(w_i|w_1,\ldots,w_{i-1})}$$

where N is the number of words in the sequence and $P(w_i)$ is the predicted probability of the word.

The model achieved the following perplexity scores:

Table 1: Perplexity Scores

Metric	**Perplexity**
Train Perplexity	205.0736
Validation Perplexity	213.0850
Test Perplexity	209.2092

1.3.8 Hyperparameter Tuning

Hyperparameter Tuning involves looping through various combinations of dropout rates, hidden dimension sizes, learning rates, and optimizers (Adam and SGD). For each combination, the model is trained and evaluated, and the results are logged using Weights & Biases (WandB) to track and visualize the performance.

The best hyperparameters for the model based on the validation and test set are:

Table 2: Hyperparameter Configurations with Perplexities

Split	**Dropout**	**Hidden Dimensions**	**Learning Rate**	**Optimizer**
Val	0.1	[300, 300]	0.01	SGD
Test	0.1	[300, 300]	0.01	SGD

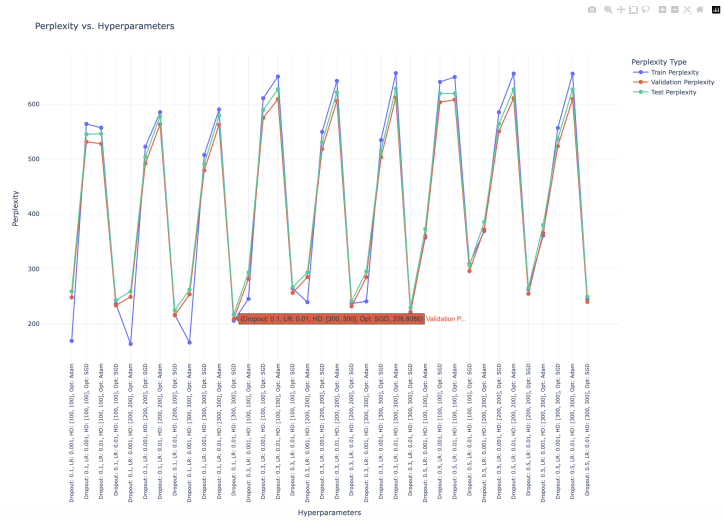

Figure 1: Line Plot for different Hyperparameters

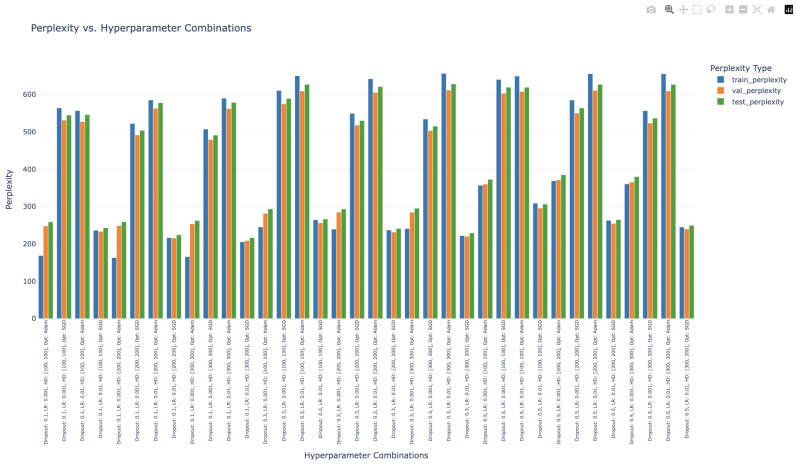

Figure 2: Bar Plot for different Hyperparameters

2 LSTM-based Neural Language Model

2.1 About

Recurrent Neural Networks (RNNs) are used to address the limitations of fixed context size in n-gram models. They improve performance by utilizing information from longer sequences, making them more suitable for sequential data. Long Short-Term Memory (LSTM) networks, a variant of RNNs, help mitigate the vanishing gradient problem by introducing cell states and gates to control

the flow of information. In this assignment, we implement a Neural Language Model using LSTM with pre-trained GloVe embeddings and evaluate its performance.

2.2 Model Architecture

The architecture of the LSTM-based Language Model (LM2) is described below:

- The input to the model is a sequence of pre-trained **Glove** embeddings of the previous words.
- These embeddings are fed into an LSTM layer that processes the entire sequence.
- The output from the LSTM layer is passed into a fully connected layer to predict the next word.
- A Softmax layer provides a probability distribution over the vocabulary.
- The vocabulary includes all words from the training data, with unknown words handled by the $< UNK >$ token.

2.3 Implementation

2.3.1 Pre-processing

Similar to the Neural Network Language Model (NNLM), the text is cleaned using the preprocess function that removes special characters and normalizes the punctuation.

2.3.2 Tokenization and Data Preparation

The tokenization and data preparation processes remain the same. Sentences are tokenized, and the dataset is split into training, validation, and test sets. The Glove embeddings are loaded, and sentences are encoded using the word-to-index mapping.

2.3.3 Custom Dataset for LSTM

The **LSTMDataset** class prepares input-output pairs of sentences, where the input is a sequence of previous words' embeddings, and the output is the next word in the sequence. Padding is applied to sequences to ensure uniform length.

2.3.4 LSTM Model

The **LSTMModel** class defines the LSTM-based language model. It consists of an LSTM layer, followed by a fully connected linear layer to map LSTM outputs to vocabulary probabilities.

```python
class LSTMModel(nn.Module):
    def __init__(self, embeddings, hidden_dim, lstm_layers=1, dropout=0.5):
        super(LSTMModel, self).__init__()

        self.embeddings = nn.Embedding.from_pretrained(embeddings, freeze=True)
        self.lstm = nn.LSTM(embeddings.shape[1], hidden_dim, num_layers=lstm_layers,
            batch_first=True, dropout=dropout)
        self.fc = nn.Linear(hidden_dim, embeddings.shape[0])
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.embeddings(x)
        lstm_out, _ = self.lstm(x)
        lstm_out = lstm_out[:, -1, :]  # Get the output from the last time step
        out = self.fc(lstm_out)
        return self.softmax(out)
```

The training and testing functions for the LSTM model are similar to those used in the NNLM. The train_model and test_model functions handle model training and evaluation, respectively, and early stopping is applied to avoid overfitting. Perplexity is computed for the validation and test sets.

2.3.5 Results

The following perplexity scores were obtained for the LSTM model:

Table 3: LSTM Model Perplexity Scores

Metric	Perplexity
Train Perplexity	103.2390
Validation Perplexity	132.0088
Test Perplexity	131.8619

3 Transformer Decoder based Language Model

3.1 About

The Transformer architecture represents a significant advancement in natural language processing by addressing the limitations of traditional sequence models like RNNs and LSTMs. Unlike these models, Transformers use self-attention mechanisms to process sequences in parallel, allowing for the capture of long-range dependencies without the need for recurrent structures.

This approach overcomes the limitations of fixed context size and the vanishing gradient problem, offering enhanced performance and scalability for text generation tasks. The Transformer Decoder's ability to process sequences in parallel and capture intricate dependencies is evaluated through perplexity scores, providing insights into its effectiveness and efficiency compared to traditional models.

3.2 Model Architecture

- Embedding Layer (initialized with GloVe embeddings).
- Positional Encoding for capturing the order of words in a sequence.

- Transformer Decoder blocks with multi-head self-attention and feed-forward layers.
- Linear layer to project the output to the vocabulary size.
- Softmax layer for the final output to produce the probability distribution over the vocabulary.

3.3 Implementation

3.3.1 Pre-processing

Similar to the Neural Network Language Model (NNLM), the text is cleaned using the preprocess function that removes special characters and normalizes the punctuation.

3.3.2 Tokenization and Data Preparation

The tokenization and data preparation processes remain the same. Sentences are tokenized, and the dataset is split into training, validation, and test sets. The Glove embeddings are loaded, and sentences are encoded using the word-to-index mapping.

4 Dataset Preparation

The dataset for this task was the same as used in the previous LSTM implementation.

4.1 Positional Encoding

The positional encoding added to the input embeddings was based on sine and cosine functions of different frequencies, ensuring that the model is aware of the word's position in the sequence. The positional encoding function used is as follows:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

where pos is the position, i is the dimension, and d is the dimensionality of the model.

5 Training Process

We trained the model using optimizer: Adam, learning rate: 0.001, epochs: 10.

The training was monitored using early stopping to prevent overfitting. We also tracked validation perplexity at the end of each epoch.

6 Results

The perplexity scores for the training and validation sets are reported in Table 4.

Dataset	Perplexity	Loss
Training	90.3379	4.5035
Validation	167.1626	5.1189
Test	168.8852	5.1292

Table 4: Perplexity Scores for the Transformer Decoder Model

7 Analysis

We evaluated three different language models: a Neural Network Language Model (NNLM), a Long Short-Term Memory (LSTM) based model, and a Transformer decoder model. Here's a table summarizing their perplexity scores:

Table 5: Perplexity Scores for Different Language Models

Model	Train Perplexity	Validation Perplexity	Test Perplexity
NNLM (LM1)	205.0736	213.0850	209.2092
LSTM (LM2)	103.2390	132.0088	131.8619
Transformer (LM3)	90.3379	167.1626	168.8852

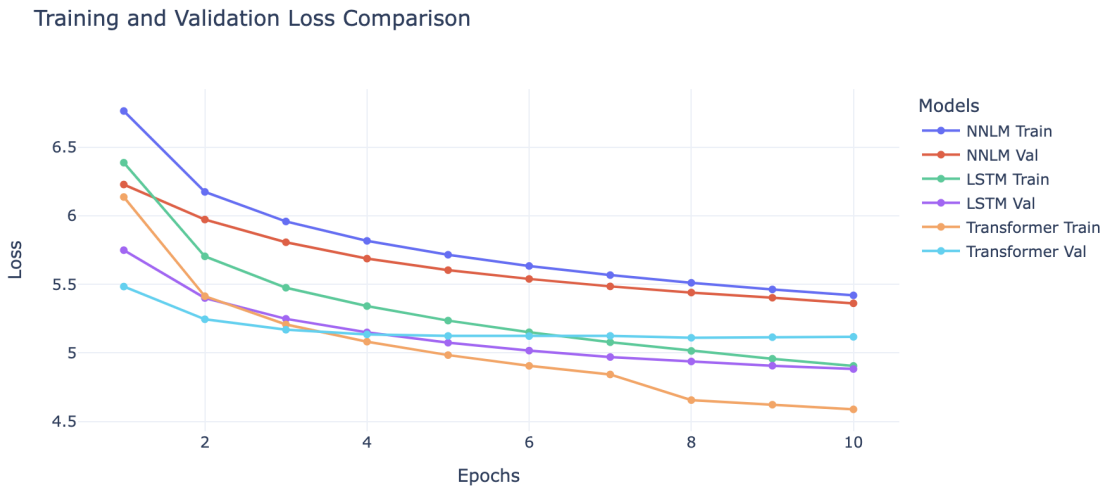

Figure 3: Loss over Epochs

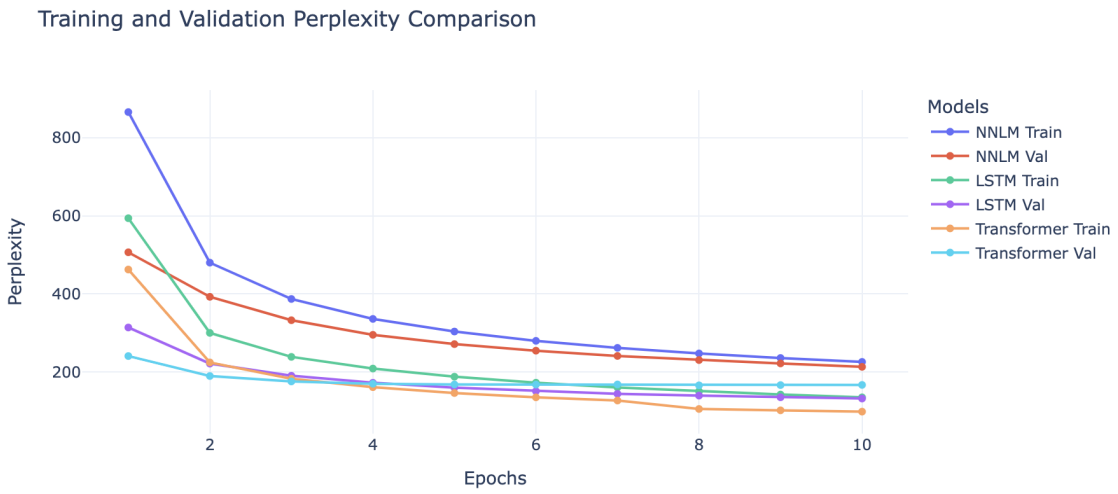

Figure 4: Perplexity over Epochs

Some of the observations from this analysis are:

- **NNLM (LM1)** The line plot showed a generally increasing trend in perplexity across hyperparameter combinations, indicating that the model's performance was sensitive to hyperparameter choices.
- **LSTM (LM2)**: The LSTM model achieved the lowest perplexity scores on both the training and test sets. This indicates that LSTM generalized well to unseen data.
- **Transformer Decoder underperforms on validation and test sets:** While the Transformer decoder is known for excelling with large datasets, the provided dataset size (10,000 validation, 20,000 test) might be too small to train a robust model. Transformers need to learn complex patterns, and with less data and a smaller number of layers due to low GPU compute, leading to poor generalization.

Overall, the performance of LSTM and Transformer is on par with similar scores and next word predictions. However, the LSTM model demonstrates better generalization on the given dataset, whereas the Transformer model may require more data and computational resources to fully exploit its potential.