Assignment-1 Advanced NLP

Prisha 2021101075

N GRAMS LANGUAGE MODEL

ABOUT

Language modeling (LM) predicts the likelihood of word sequences, aiding applications such as text generation, translation, and speech recognition. Traditional models like N-grams calculate word probabilities based on fixed-length contexts, which limits their effectiveness with larger contexts. In contrast, neural language models use embeddings and deep learning architectures to learn more complex patterns in text data.

This section focuses on implementing a neural language model using deep learning frameworks, specifically with a 5-gram context.

ARCHITECTURE

The Neural Language Model (NLM) in this implementation uses a multi-layer architecture with pre-trained GloVe embeddings as input. The main components of the architecture are:

1. **Input Layer**: Pre-trained GloVe embeddings for the previous 5 words (5-gram context) are concatenated to form the input.
2. **First Hidden Layer**: The concatenated embeddings are passed through a linear layer of size 300, followed by GELU activation, layer normalisation, and dropout.
3. **Second Hidden Layer**: The output from the first hidden layer is processed through another linear layer of size 300 with similar activation and normalisation.
4. **Output Layer**: The final output is a linear layer projecting to a vector of vocabulary size, followed by a softmax layer, which generates the probability distribution for the next word in the sequence.

IMPLEMENTATION

1. **Preprocessing**:
    - Text data is preprocessed by tokenizing sentences and cleaning the text using regex.
    - GloVe embeddings are loaded to create a vocabulary, mapping words to their corresponding embeddings.
    - N-grams (5-grams) are generated as input, and labels are created using the next word in the sequence.
2. **Data Splitting**:
    - The dataset is split into training (70%), validation (20%), and test (10%) sets.
    - Sentences are shuffled to ensure a random distribution.
3. **Training**:
    - The model is trained using cross-entropy loss and either Adam or SGD as the optimizer.
    - Dropout (with different rates) is applied for regularization, and early stopping is used based on validation loss.
    - Training involves logging the loss and perplexity using Weights and Biases (wandb).
4. **Hyperparameter Tuning**:
    - The hyperparameters tuned include the optimizer (Adam/SGD), hidden layer size (200, 300, 400), and dropout rate (0.3, 0.5, 0.7).
    - Each combination is evaluated based on validation and test perplexity to find the optimal configuration.

RESULTS

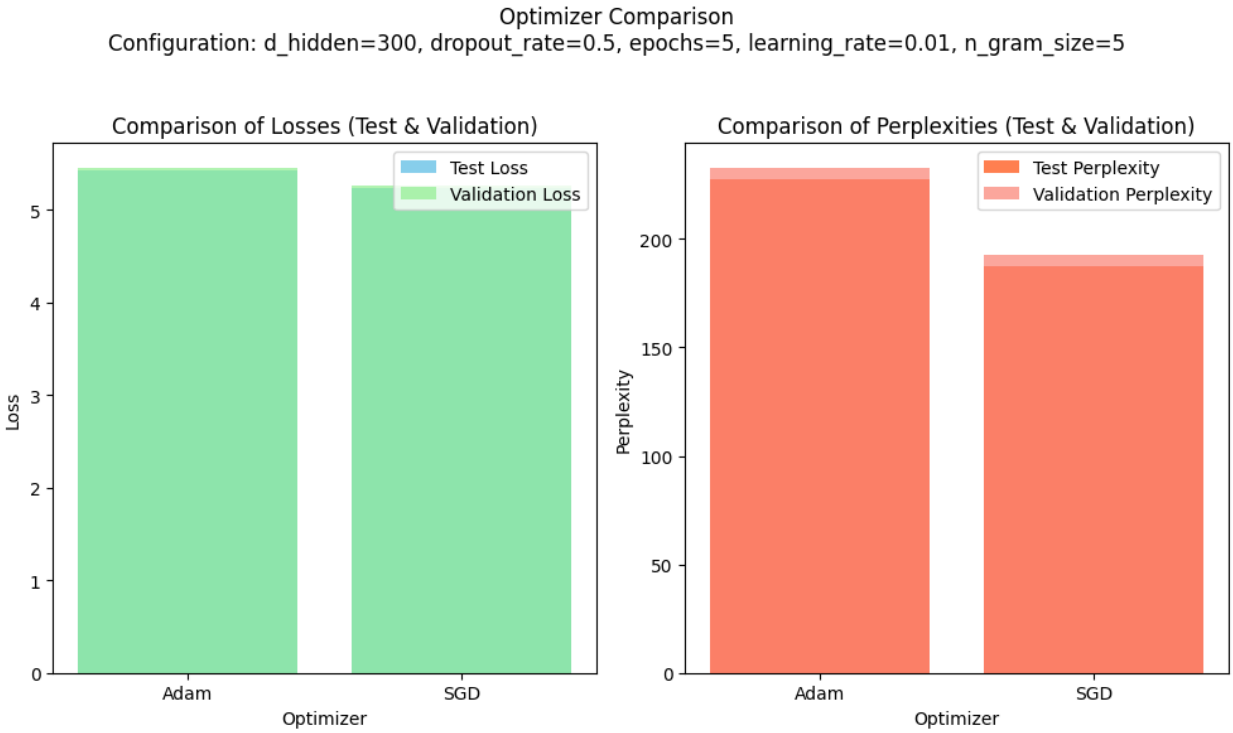
DATASET	LOSS	PERPLEXITY
TRAIN	5.167	175.23
TEST	5.427	227.60

VALIDATION	5.450	232.90

HYPER-PARAMETER TUNING

- The hyperparameters tuned include the optimizer (Adam/SGD), hidden layer size (200, 300, 400), and dropout rate (0.3, 0.5, 0.7).

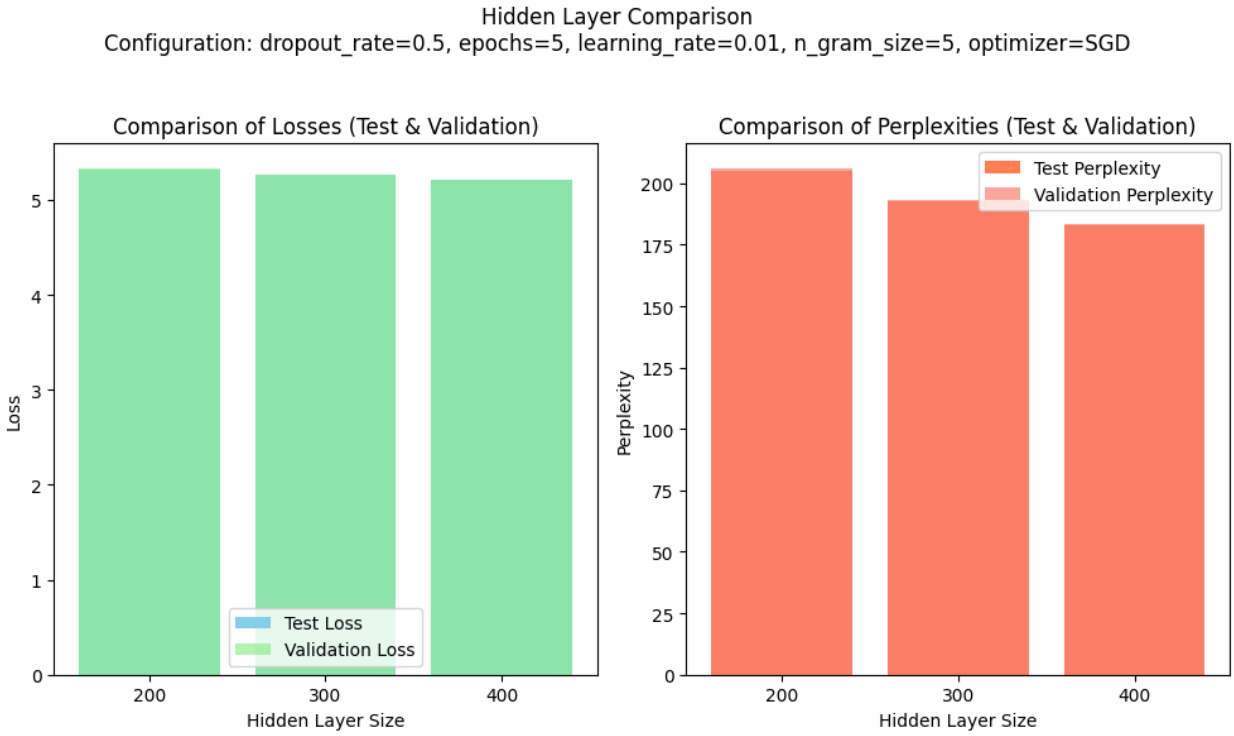
OPTIMIZER COMPARISON :



From the above plots, we can see that when keeping other parameters constant and varying only the optimizers to compare their performance, SGD clearly performed better. Here are the corresponding values : https://wandb.ai/home

- ADAM : https://wandb.ai/oops-moment-iiit-hyderabad/language_model/runs/g51oby8b/overview
- SGD : https://wandb.ai/oops-moment-iiit-hyderabad/language_model/runs/xagiw51h/overview

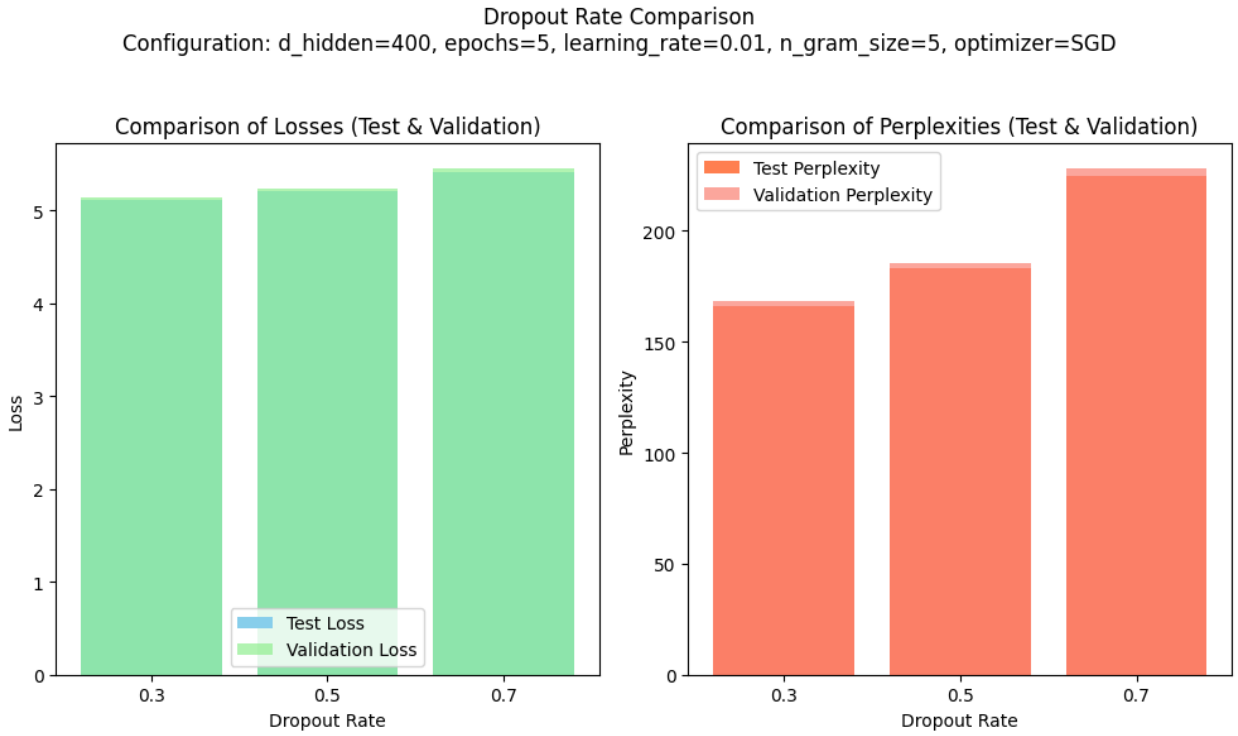
EMBEDDING DIMENSIONALITY COMPARISON :



From the above graph, we can see that higher dimensionality allows the model to capture more complex patterns. In this case, higher dimensionality led to reduced loss and consequently lower perplexity.Here are the corresponding values : https://wandb.ai/home

- Dimension 200 : https://wandb.ai/oops-moment-iiit-hyderabad/language_model/runs/j9fhwx5u?nw=nwuseroopsmoment
- Dimension 300 : https://wandb.ai/oops-moment-iiit-hyderabad/language_model/runs/29gaxfur?nw=nwuseroopsmoment
- Dimension 400 : https://wandb.ai/oops-moment-iiit-hyderabad/language_model/runs/jg0xsci5?nw=nwuseroopsmoment

DROPOUT COMPARISON :



IA lower dropout rate results in reduced loss because higher dropout tends to cause the model to lose important information and fail to capture the full relationships in the data. Here are the corresponding values :

- Dropout : 0.3 https://wandb.ai/oops-moment-iiit-hyderabad/language_model/runs/841gm3jc?nw=nwuseroopsmoment
- Dropout : 0.5 https://wandb.ai/oops-moment-iiit-hyderabad/language_model/runs/j61cwf3q
- Dropout : 0.7 https://wandb.ai/oops-moment-iiit-hyderabad/language_model/runs/t9pfrrfc

Finally, from the hyperparameter tuning, we found that the SGD optimizer works best because of its ability to converge effectively with the given learning rate and data. A higher embedding dimension performed better because it allows the model to capture more complex patterns, leading to better generalization. A lower dropout rate was favorable as it helps retain more information during training, reducing the risk of losing important relationships in the data. It's worth noting that

we tested each parameter individually, rather than in combination, due to computational constraints.

LSTM LANGUAGE MODEL

ABOUT

The task is to create a Neural Language Model using an LSTM (Long Short-Term Memory) network to predict the next word in a sentence. Traditional n-gram models like 5-grams have limitations because they rely on fixed-length contexts, which fail to capture long-range dependencies. Recurrent Neural Networks (RNNs), particularly LSTM networks, address this issue by incorporating memory cells that retain information over longer sequences. This helps in capturing dependencies and context across varying lengths of input sequences.

ARCHITECTURE

The architecture of the LSTM-based Neural Language Model includes the following key components:

1. **LSTM Layers**: The model consists of multiple LSTM layers that process the input sequentially. Each layer's input includes the hidden state and cell state from the previous timestep, along with the embedding of the current word.
2. **Embedding Layer**: Words from the input sequence are converted into dense, continuous vector representations using a pre-trained embedding (e.g., GloVe).
3. **Linear Layer**: The final LSTM layer outputs a 300-dimensional vector, which is passed through a fully connected (linear) layer to transform it into a vector of vocabulary size.
4. **SoftMax Layer**: The output of the linear layer is then passed through a SoftMax function to generate a probability distribution over the entire vocabulary, which is used for next-word prediction.

IMPLEMENTATION DETAILS

1. **Preprocessing**:

Preprocessing is the initial step, where the text is tokenized into sentences, each sentence is cleaned by removing non-alphabetic characters and converting it to lowercase, and then further tokenized into words. This results in a cleaned, word-level representation of the text.

2. **Vocabulary Building**:

   Following this, **Vocabulary Building** is performed by mapping each unique word to an index, while incorporating special tokens such as <UNK> for unknown words and <PAD> for padding. This vocabulary is crucial for converting words into numerical indices that the model can process.

3. **Dataset Preparation**:

   In the **Dataset Preparation** step, sentences are transformed into sequences of word indices. These sequences are then used to create input-output pairs where the input sequence is used to predict the next word in the output sequence. This structured dataset is essential for training the model.

4. **Model Definition**:

   The **Model Definition** involves several components: an **Embedding Layer** that uses pre-trained embeddings like GloVe to convert word indices into dense vectors; multiple **LSTM Layers** to process the sequences while maintaining hidden and cell states; a **Linear Layer** to map LSTM outputs to the vocabulary size; and a **SoftMax Activation** to generate a probability distribution for predicting the next word.

5. **Training & Evaluation**:

   During **Training**, the model's performance is optimized using the Adam optimizer and the CrossEntropyLoss function, which computes the loss between predictions and true labels. Early stopping based on validation loss is also implemented to avoid overfitting.

Evaluation involves calculating perplexity on the test set to assess model performance. Perplexity scores for each batch, as well as the average perplexity for the entire dataset, are saved and analyzed.

DATASET	LOSS	PERPLEXITY
TRAIN	4.73	113.02
TEST	5.12	167.23
VALIDATION	5.10	162.70

TRANSFORMER DECODER LANGUAGE MODEL

ABOUT

The programming exercise focuses on implementing a Transformer Decoder-based language model using PyTorch. Transformer models, such as OpenAI's GPT series, leverage the Transformer Decoder architecture to generate coherent and contextually relevant text. The task involves constructing a language model capable of predicting the next word in a sequence based on the context provided by previous words. The model is trained on a text corpus, learning to mimic the structure and patterns of the training data to generate text effectively.

ARCHITECTURE

1. **Positional Encoding**: Since the Transformer model does not inherently capture the order of words, a positional encoding layer is used to inject positional information into the embeddings. This allows the model to distinguish the positions of words in a sequence.
2. **Transformer Decoder Blocks**: The core of the architecture consists of multiple Transformer Decoder blocks. Each block includes a multi-head self-attention mechanism, a feed-forward neural network, and layer normalization. The self-attention mechanism enables the model to weigh the importance of different words in the sequence when making predictions.
3. **Embedding Layer**: The model uses pre-trained GloVe embeddings to convert word indices into dense vectors. These embeddings are loaded from a file and

incorporated into the model, providing a rich representation of words.

4. **Linear and SoftMax Layers**: After processing through the decoder blocks, the output is passed through a linear layer to map it to the vocabulary size. A SoftMax activation function is then used to generate a probability distribution over the vocabulary, predicting the next word in the sequence.

IMPLEMENTATION DETAILS

Implementation Details

1. **Preprocessing**: Text data is tokenized into sentences and words, cleaned of non-alphabetic characters, and converted to lowercase. This results in a cleaned, word-level representation suitable for model input.
2. **Vocabulary Building**: A vocabulary is created from the preprocessed text and pre-trained GloVe embeddings. Each word is mapped to a unique index, and special tokens like <UNK> (unknown words) and <PAD> (padding) are included.
3. **Dataset Preparation**: Sentences are converted into sequences of word indices. Input-output pairs are created where the input sequence predicts the next word in the output sequence. These pairs are used for training the model.
4. **Model Definition**: The TransformerLanguageModel class defines the architecture, including embedding layers, positional encoding, multiple Transformer Decoder blocks, and a final linear layer. The model processes sequences through these layers to predict the next word.
5. **Training and Evaluation**: The model is trained using CrossEntropyLoss and the Adam optimizer. Training involves monitoring and optimizing the loss, with early stopping implemented to prevent overfitting. The model's performance is evaluated based on perplexity, calculated from the loss on the test set.
6. **Saving and Loading Models**: The best-performing model based on validation loss is saved to a file for future evaluation and inference. This includes saving perplexity scores for train, validation, and test sets.

RESULT

DATASET	LOSS	PERPLEXITY
TRAIN	3.93	51.12
TEST	4.03	56.28
VALIDATION	4.02	56.19

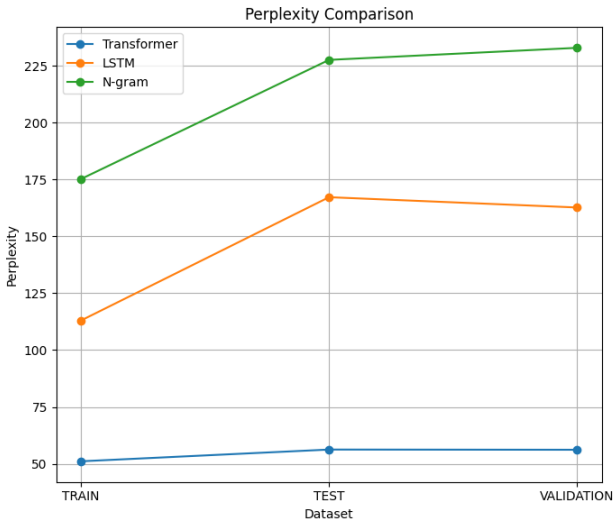
ANALYSIS

The Transformer model outperforms both LSTM and n-gram models primarily due to its unique architecture and capabilities:

1. **Attention Mechanism**: The core innovation in Transformers is the self-attention mechanism. Unlike LSTMs, which process sequences in a linear fashion, the attention mechanism allows the model to weigh the importance of different words in a sequence, regardless of their position.
2. **Handling Long Contexts**: Transformers are particularly adept at managing long contexts. In contrast to LSTMs, which might struggle with very long sequences due to issues like vanishing gradients, Transformers can directly attend to all parts of the input sequence.
3. **Parallel Processing**: Transformers process entire sequences simultaneously rather than sequentially. This parallelism accelerates training and inference compared to LSTMs, which process tokens one at a time.

However, it is important to note that the Transformer model requires significantly more computational resources compared to the other two models. This increased resource demand is due to its complex architecture and the extensive calculations involved.

The graphs below demonstrate the performance of the three models:



Dataset	Trans-former Loss	Trans-former Perplexity	LSTM Loss	LSTM Perplexity	N-gram Loss	N-gram Perplexity
TRAIN	3.93	51.12	4.73	113.02	5.167	175.23
TEST	4.03	56.28	5.12	167.23	5.427	227.60
VALIDATION	4.02	56.19	5.10	162.70	5.450	232.90