Neural Language Model Training Using PyTorch

Dataset

The dataset consists of a single raw text file containing continuous language text, with a length of about 711K characters. Cleaning was done by converting all the text to lowercase and removing whitespace. Character-level tokenization was implemented from scratch. After cleaning and tokenization, the number of tokens was approximately 700,000. A custom vocabulary was implemented by collecting all unique tokens in the dataset and assigning each one a unique ID. The text was numericalized by converting every token into its corresponding ID. This ensures that the model can process the data as integer sequences. The dataset was split into training data and validation data with a validation ratio of 0.3.

Model Architecture

An LSTM was used to train the given dataset. It was implemented completely from scratch in PyTorch. It consists of three components:

1. LSTMCell - It is the lowest-level implementation using the gate equations. For each time step, given the input vector x, previous hidden state h, and cell state c, each gate was computed using the mathematical expressions, and the hidden state and cell states are updated.
2. MultiLayerLSTM - This component stacks multiple LSTM layers. It consists of a forward loop that feeds the output of one layer to the next as input, manages the separate cell and hidden states in every layer, and adds dropout between layers. For each layer, the entire sequence is passed through the LSTMCell and forwarded as explained before. The hidden states and cell states are collected for all the layers.
3. LSTMModel - This combines everything into a single language model. It includes an embedding layer where the token indices are converted to dense vectors, a multi-layer LSTM that processes the embeddings and produces hidden states, and a linear output layer that maps the LSTM outputs into vocabulary logits.

Experiments

The dataset was split into 70% training data and 30% validation data. Preprocessing, as discussed in the dataset section, was implemented before training.
The model was trained using the following:

1. Optimizer - Adam
2. Batch Size - 32
3. Sequence Length - 64
4. Learning Rate - 0.001
5. Number of epochs - 10

All three experiments were done using the Colab T4-GPU.

Model-1

Embedding Size - 32
Hidden Size - 64
Layers - 2
Dropout - 0.2

Model-2

Embedding Size - 128
Hidden Size - 256
Layers - 2
Dropout - 0.2

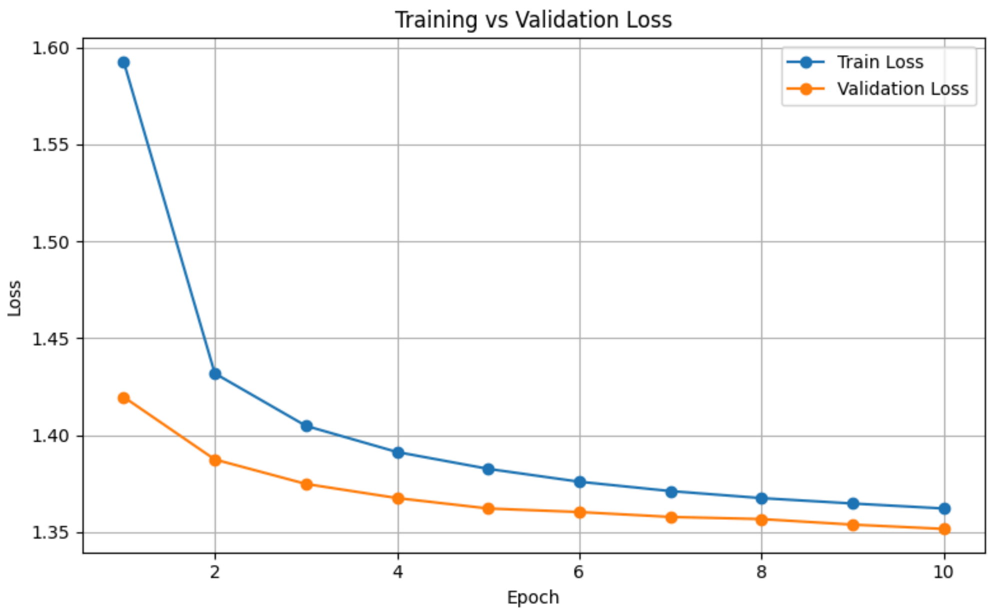
Model-3

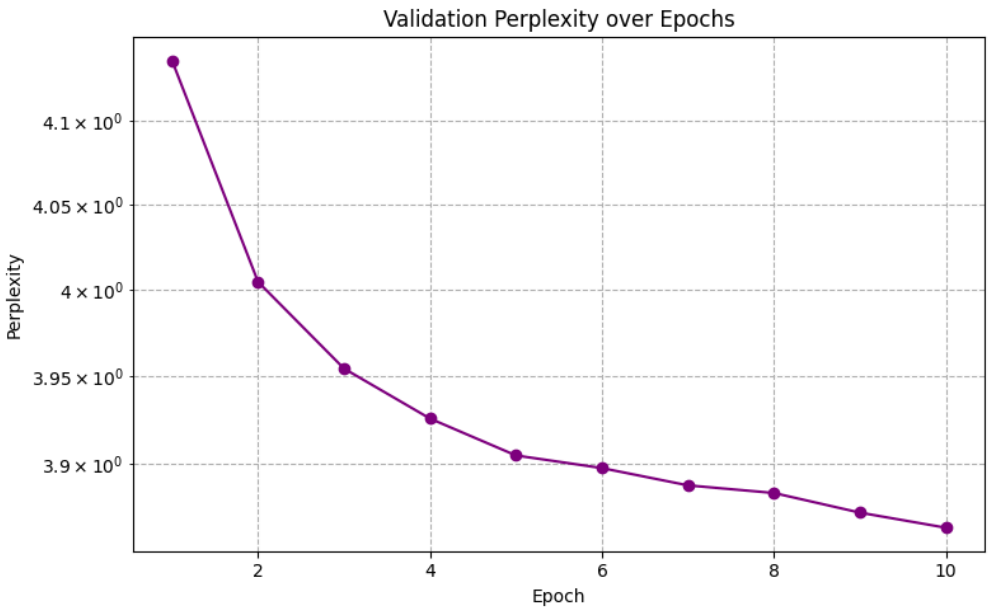
Embedding Size - 256
Hidden Size - 512
Layers - 2
Dropout - 0.2

All the models were evaluated using perplexity. It is a metric that measures how well the model predicts the next token. A lower perplexity value indicates better prediction.

Results

Model-1

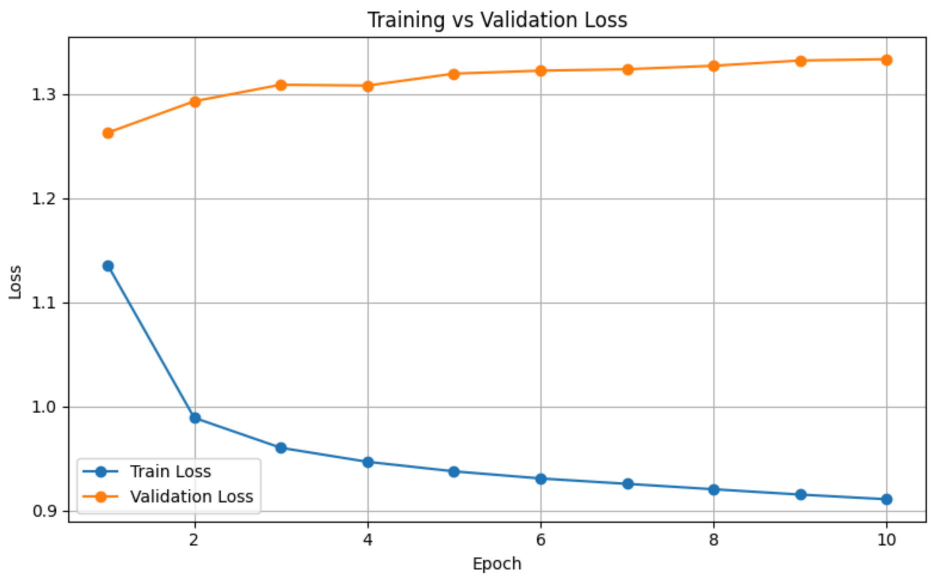


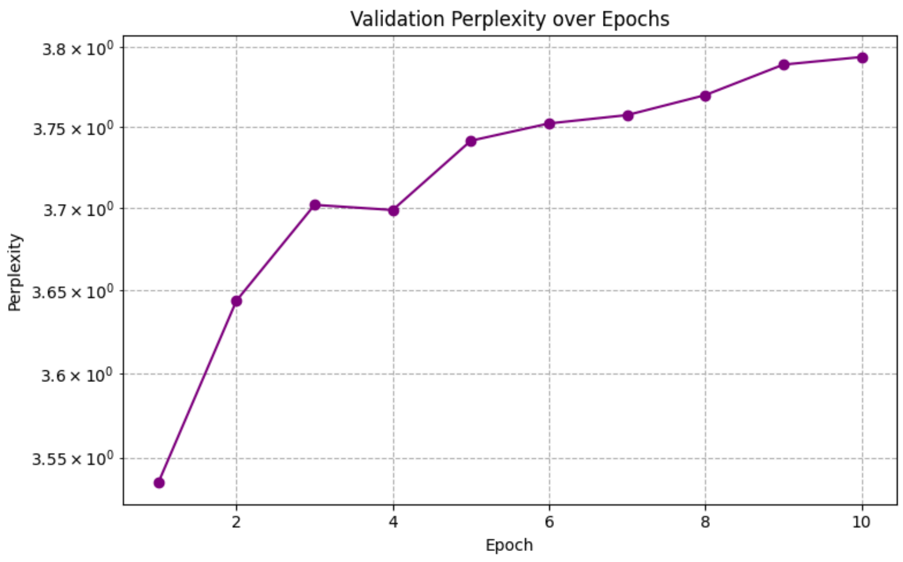


From the loss curve, we can infer that both training and validation loss are decreasing over the epochs, but the curve flattens around the last epochs. The loss is still higher for a language model.

From the perplexity curve, we can infer that there is a steady decrease in the perplexity across epochs.

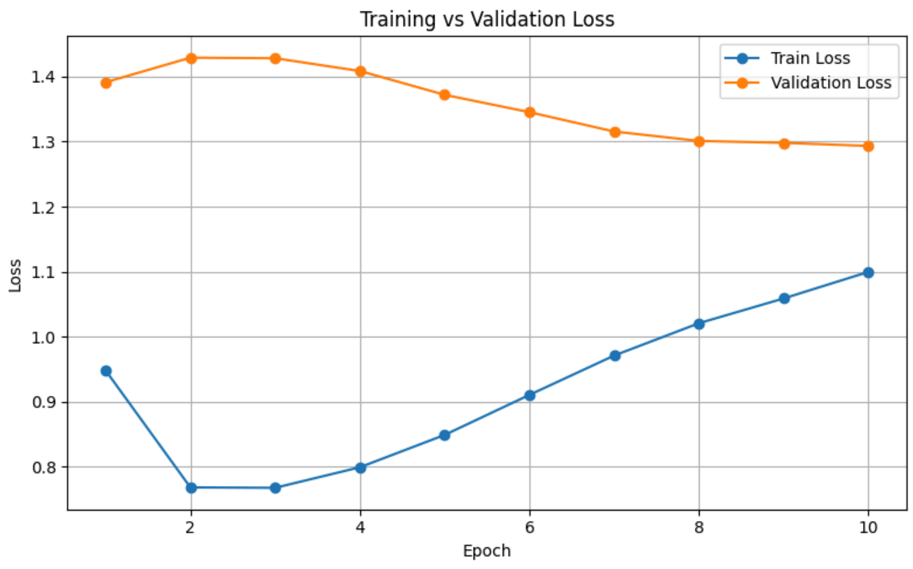
<u>Model-2</u>

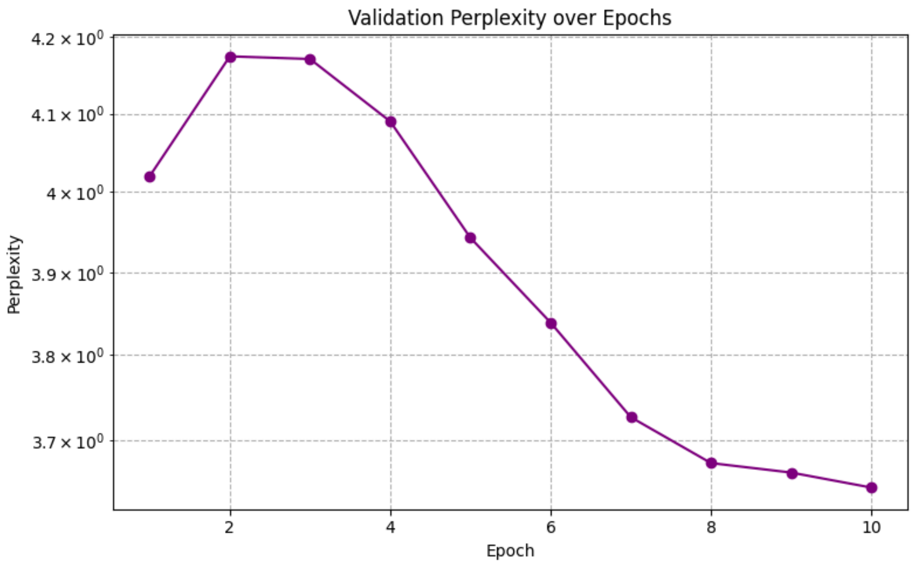




From the loss curve, we can infer that the model overfits because the training loss decreases across epochs but the validation loss increases across epochs. The model doesn’t perform well on new data.
From the perplexity curve, we can see that perplexity increases across the epochs.

Model-3





From the loss curve, we can infer that even though the training loss is increasing, the model is learning well and performing well on the validation set.
From the perplexity curve, we can infer that it decreases steadily across the epochs.