NLP-243: Language Modeling on Penn Treebank using PyTorch

Kushagra Seth
University of California, Santa Cruz
kuseth@ucsc.edu

Abstract

The objective of this homework is to develop a Language Model based on Penn Treebank data set (text-only). The main goal is Text Generation, i.e., to predict the next token in a sequence given the previous N tokens.

I have trained a deep neural network model using PyTorch that outputs the next token given a sequence.

1 Introduction

We don't have hand-written labels(ground truth) for the text in the Penn Treebank corpus but we still need to have target values so our model can calculate loss and learn.

This is an Unsupervised Learning problem, i.e., instead of having a label for each text in the corpus, the target values are simply the next tokens in the input.

For example, consider the following sequence:

```
<start> nlp 243 is the best <t>
0       1   2   3  4   5    6
```

The input-target sequence pairs for the above-mentioned sequence would be as follows:

```
input: <start>
target: nlp

input: <start> nlp
target: 243

input: <start> nlp 243
target: is

...
```

This particular data set with which we have been provided with has already been split into train, val and test sets by datasets library in HuggingFace.

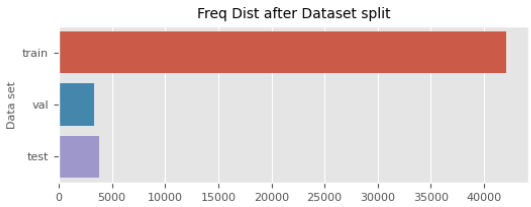

Figure 1: Horizontal Bar Plot that represents the data set

2 Proposed Framework

2.1 Creating Sequence Pairs from the Token List

For example,
consider the following list of tokens:

⟨**start**⟩ **A B C D E F G H I**

len(Token List) = 10
Seq Len = 3
No of Seqs = len(Token List) - Sequence Len = 10 - 3 = 7

Idx	Input Sequence	Target Sequence
0	<start> A B	A B C
1	A B C	B C D
2	B C D	C D E
3	C D E	D E F
4	D E F	E F G
5	E F G	F G H
6	F G H	G H I

Figure 2: Approach used to create Input and Target Sequence Pairs

2.2 Encoding Sequences using word2idx Vocabulary

As is well known, machine learning and neural networks requires data in the form of numbers. To basically turn text into a numeric vector, I have used encoding techniques.
I have converted the sequences of input-target pairs into sequences of dictionary id's with reference to word2idx dictionary that contains an id for each of the unique tokens in the corpus.

2.3 Replacing word2idx id's in Encoded Sequences using Word Embeddings Representations

Word Embeddings is an approach to provide a dense vector representation of words to capture something about their meaning. Each word is represented by a point in the vector space and points are learned and moved based on the words that surround them. This allows word embeddings to learn something more about the meaning of the words. Similar meanings are locally clustered within the vector space. Each vector is represented in tens or hundreds of dimensions.

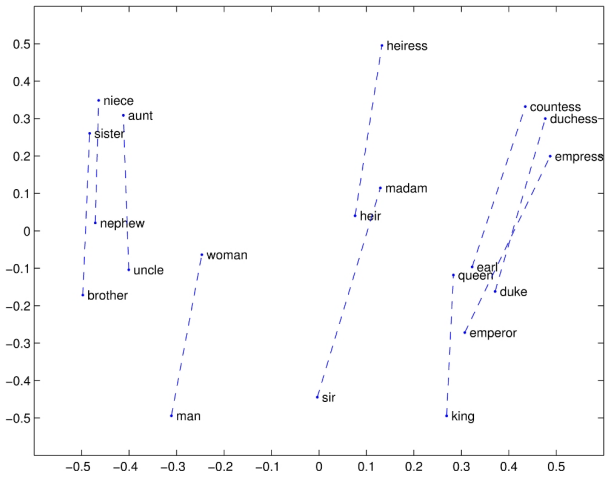

Figure 3: GloVe [2]

I have used **pre-trained GloVe Embeddings** for this problem. I have created a vocabulary in the form of a dictionary which contains unique words present in the corpus. There are about 2 million words in GloVe with each word having its representation in 300 dimensions. I have used the GloVe representation for all of the vocab words .
I have added embedding representation for 2 tokens namely $\langle start \rangle and \langle unk \rangle$.

All words which are not present in the vocab are represented using token $\langle unk \rangle$.

```
['<start>' '<unk>' ',' '.' 'the']
[[ 0.          0.          0.          ...  0.          0.
   0.        ]
 [ 0.22418656 -0.28881998  0.13854443 ...  0.19310581 -0.07767605
  -0.1448164 ]
 [-0.082752    0.67204    -0.14987    ... -0.1918     -0.37846
  -0.06589   ]
 [ 0.012001    0.20751    -0.12578    ...  0.13871    -0.36049
  -0.035     ]
 [ 0.27204    -0.06203    -0.1884     ...  0.13015    -0.18317
   0.1323    ]]
```

Figure 4: Word Embedding Representation of first 5 words in the vocab

2.4 Recurrent Neural Networks (RNN)

An Artificial Neural Network that is specifically designed to operate on data including sequences is known as a Recurrent Neural Network (RNN). Ordinary feedforward neural networks are only meant for data points that are independent of each other.

To include the dependencies between these data points, we must change the neural network if the data are organized in a sequence where each data point depends on the one before it.

RNNs have the concept of "memory," which enables them to save the states or details of previous inputs in order to produce the subsequent output in the sequence.

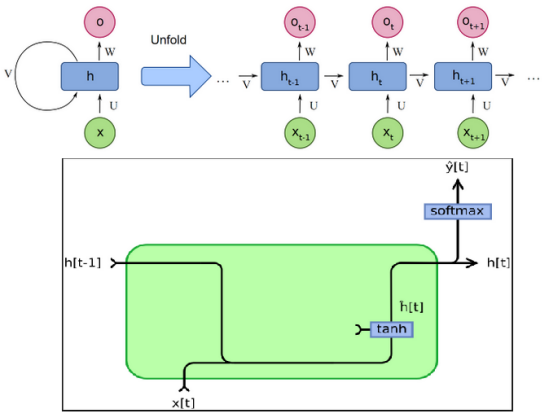

Figure 5: RNN Model [3]

2.5 Long-Short Term Memory Networks (LSTM)

Long Short Term Memory Networks, most commonly referred to as "LSTMs," are a unique class of RNN that can recognize long-term dependencies. LSTMs are explicitly designed to avoid the long-term dependency problem. They don't struggle to learn; rather, remembering information for extended periods of time is basically their default

behavior.

All recurrent neural networks have the shape of a series of neural network modules that repeat. This recurring module in typical RNNs will be made up of just one tanh layer.

Although the repeating module of LSTMs also has a chain-like topology, it is structured differently. There are four neural network layers instead of just one, and they interact in a very unique way.

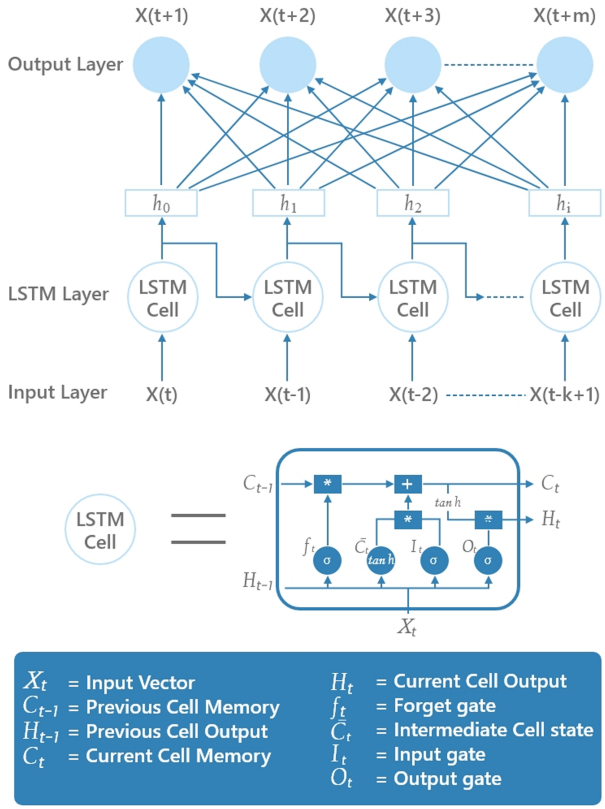

Figure 6: LSTM Model [3]

2.6 Gated Recurrent Units (GRU)

GRUs are an enhanced form of the traditional recurrent neural network. The so-called update gate and reset gate are used by GRU to address the vanishing gradient issue that plagues a normal RNN.

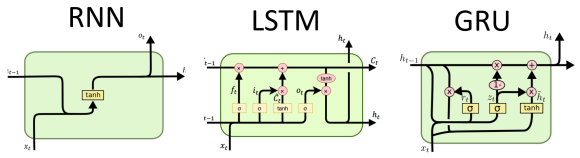

Figure 7: GRU Model [4]

In essence, these two vectors determine what data should be sent to the output. They have the unique ability to be trained to retain information from the past without having it fade away over time or to discard information which is irrelevant to the prediction.
They also have better performance than LSTM when dealing with smaller datasets.

3 Experiments

I came up with a Baseline model with certain Hyper Parameters. Then, I have tuned the model by varying one of these parameters to decrease the perplexity of this model.

I tried using sequence-to-sequence networks like RNN, LSTM and GRU to get the lowest perplexity for this Data Generation Modelling Task.

I got the lowest perplexity with a variation of optimizer Adam called *AdamW* on the test set while the perplexities with other optimizers such as SGD, RMSProp and AdaGrad was way worse and way above 200.

3.1 Experiment-1: Varying the Learning Rate

The *learning rate* or *step size* refers to how frequently the weights are updated during training.

In particular, the learning rate is a hyper-parameter that can be customized and is used to train neural networks. Its value is typically small and positive, falling between 0.0 and 1.0.

How quickly the model adapts to the challenge is determined by the learning rate. Given the smaller changes to the weights made with each update, smaller learning rates necessitate more training epochs, whereas bigger learning rates produce quick changes and necessitate fewer training epochs.

A model may converge to a less-than-ideal solution too soon if the learning rate is too high, whereas the opposite is true if the learning rate is too low.

Table 1: Fixed Hyper Parameters

Hyper Parameters	Values
Batch Size	64
Hidden Dimensions	256
Number of Layers	2
Epochs	10

Table 2: Varying the Learning Rate Parameter

Learning Rate	**0.01**	0.04
Train Perplexity	***90.23***	121.85
Val Perplexity	***127.58***	187.56

3.2 Experiment-2: Varying the Batch Size

Different optimization techniques are used to train neural networks.
In this case, a prediction is made based on the model's current state, and the difference between the predicted and the actual values is used to estimate the error gradient. The model weights are then updated using this error gradient, and the procedure is repeated. The *batch size*, often known as the *batch*, is a hyper-parameter for the learning algorithm that determines how many training samples are utilized to estimate the error gradient.

Table 3: Fixed Hyper Parameters

Hyper Parameters	**Values**
Learning Rate	0.01
Hidden Dimensions	256
Number of Layers	2
Epochs	10

Table 4: Varying the Batch Size Parameter

Batch Size	**64**	128	256
Train PPL	***90.23***	92.88	105.87
Val PPL	***127.58***	135.98	152.56

3.3 Experiment-3: Varying the Hidden Dimensions

Hidden dimension determines the feature vector size of the h_n (hidden state). At each timestep, a sequence network layer will take a h_n and input.
I have used multiple variations of hidden dimensions to get the maximum accuracy on the validation set.

Table 5: Fixed Hyper Parameters

Hyper Parameters	**Values**
Learning Rate	0.01
Batch Size	64
Number of Layers	2
Epochs	10

Table 6: Varying the Hidden Dimensions Parameter

Hidden Dims	128	256	512	1024
Train Perplexity	**82.75**	90.23	131.24	142.53
Val Perplexity	***116.51***	127.58	215.58	242.88

3.4 Experiment-4: Varying the Epochs Parameter

In machine learning, an epoch refers to one full iteration of the algorithm over the training data set. An essential hyper-parameter for the algorithm is this epoch's number.

The amount of complete runs of the whole training data set during the algorithm's training or learning process is specified in terms of epochs. The internal model parameters of the data set are modified at each epoch.

Table 7: Fixed Hyper Parameters

Hyper Parameters	**Values**
Learning Rate	0.01
Batch Size	64
Number of Layers	2
Hidden Dims	128

Table 8: Varying the Epochs

Epochs	3	5	**10**
Train Perplexity	95.23	89.92	***82.75***
Val Perplexity	127.75	121.85	***116.51***

3.5 Experiment-5: Varying Fully Connected Layers

I have used a Feed-Forward Network on the output of the Sequence Network layer.
I have tried two configurations of this feed-forward network with 1 and 2 fully connected layers.

Table 9: Varying Fully Connected Layers

Hyper Parameters	**Train PPL**	**Val PPL**
1 Fully Connected Layer	98.31	128.98
2 Fully Connected Layers	***82.75***	***116.51***

3.6 Experiment-6: Using different Sequence-to-Sequence Networks

As this is a Text Generation model which is based on Sequences, I have used multiple types of Seq-to-Seq Networks.

Table 10: Varying Seq-to-Seq Network Types

Hyper Params	Train PPL	Val PPL
RNN	193.41	204.54
GRU	160.27	188.81
LSTM	***82.75***	***116.51***

3.7 Experiment-7: Varying Dropout

I tried different values of Dropout in range 0.2 - 0.5 but got the best results by fixing value of dropout to 0.2.

I have tried applying Dropout before LSTM Layer and before the output layer.

Table 11: Varying Dropout

Dropout	Train PPL	Val PPL
Before Seq Layer	89.55	125.72
Before O/P Layer	95.21	131.59
Before Both Layers	***82.75***	***116.51***

3.8 Experiment-8: Varying the Optimizers

I tried using SGD and Adagrad for this model but the performance was alot worse than that compared to Adam. The Perplexity with the these two optimizers was way above 200.

I have used different versions of Adam provided in the PyTorch library namely Adamax and AdamW.

Table 12: Fixed Hyper Parameters

Hyper Params	Train PPL	Val PPL
Adam	93.87	127.97
Adamax	87.94	120.78
AdamW	***82.75***	***116.51***

4 Results and Conclusion

In this project, I have addressed the task of building a Text Generation Model commonly referred to as Language Modelling in domain of NLP.

This model was trained on the Penn Treebank data set which contains around 1 million tokens.

I created an LSTM model using PyTorch in python for this data set and evaluated its performance by tuning different hyper-parameters.

I have selected the configuration of Hyper-parameters that produced the lowest Perplexity values on the data set splits.

4.1 Perplexity

Perplexity is calculated using the below-mentioned formula:

$$\text{Perplexity} = \exp\left(\begin{array}{c}\text{Avg Loss}\\ \text{per Epoch}\end{array}\right)$$

Figure 8: Perplexity Formula

Train Perplexity = *82.75*
Val Perplexity = *116.51*
Test Perplexity = *108.72*

Table 13: Hyper Parameters Configuration that produced lowest Perplexity

Hyper Parameters	Values
Type of Seq-to-Seq Network	LSTM
Batch Size	64
Epochs	5
Hidden Dimensions	128
Learning Rate	0.01
Fully Connected Layers	2
Dropout before LSTM & O/P Layer	0.2
Optimizer	AdamW

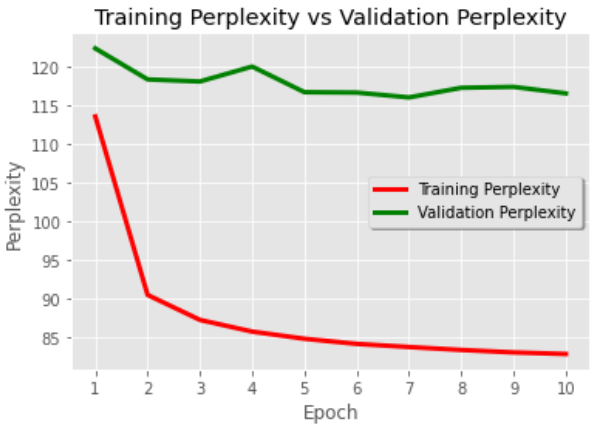

Figure 9: Perplexity vs Epoch Graph

4.2 Average Loss

The training loss indicates how well the model is fitting the training data, while the validation loss indicates how well the model fits new data.

Following is a line plot that compares the Training Loss vs the Validation Loss on the respective data set splits.

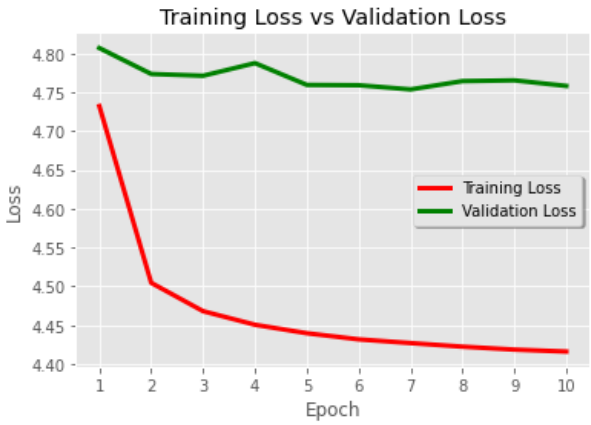

Figure 10: Visualization of Training and Validation Loss

5 Future Work

I got best results with LSTM Model with 2 Fully Connected Layers and Dropout value = 0.2. I can perhaps try to reduce the Perplexity by combining this model with Attention Mechanism.

Acknowledgments

I would like to thank Nilay Patel (Teaching Assistant for the this course, i.e., NLP-243) who provided me with constant support and help during this project.

References

[1] https://www.researchgate.net/publication/344554659_A_Deep_Learning_Approach_for_Human_Activities_Recognition_From_Multimodal_Sensing_Devices

[2] https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0262708

[3] https://towardsai.net/p/l/neural-networks-the-rise-of-recurrent-neural-networks

[4] http://dprogrammer.org/rnn-lstm-gru