ECE 542 Fall 19 Homework 5: Recurrent Neural Network (RNN)

Poorvi Rai
Department of Computer Science
North Carolina State University
prai@ncsu.edu — Student ID - 200263486

Abstract— **This report documents the design and implementation of Recurrent Neural Network using LSTM units to demonstrate the power of LSTMs in applications such as sequence counting and language modeling. For sequence counting, a single LSTM unit was manually configured with different parameters to count the digit 0 in a sequence under different conditions.**

In case of language modeling, both word-level and character-level models were trained on the same neural network architecture and the model perplexity was plotted against the number of epochs. The network architecture comprises of an embedding layer, two LSTM layers followed by a dense layer with softmax activation. The hyperparameters chosen and few other observations are discussed in detail and associated results have been visualized through various plots. The perplexity on the validation set after 50 epochs of training came out to be 233.11 for the word-level model and 2.98 for the character-level model.

I. Introduction

Sequence Counting and Language Modeling have numerous applications in today's world and are widely researched topics. Both problems pose the challenge to "remember" previously seen inputs to draw meaningful conclusions. While counting sequences, we can put restrictions such as to identify the number of 0's that occurred, or the number of even numbers or many different complex formulations. Only if the neural network is successful in remembering the previous inputs can we develop a counter to keep track of sequences based on different imposed criteria.

In case of Language Modeling, the notion is inherently probabilistic. A language model is a function that puts a probability measure over strings drawn from some vocabulary. Besides assigning a probability to each sequence of words, the language models also assigns a probability for the likelihood of a given word (or a sequence of words) to follow a sequence of words. For a neural network to efficiently do this, it must "remember" the context. Humans don't start their thinking from scratch every second and we want the network to behave the same way. As opposed to traditional neural networks that were incapable to model such problems, Recurrent Neural Networks(RNN) addressed this issue and hence became popular in the field of Natural Language Processing. That being said, one major drawback of RNN was that it could only remember recent information. This might not be very helpful in cases where terms, tense or feature occurred further away from the current text with the gap between the relevant information and the point where it is needed to be quite large.

Unfortunately, as that gap grows, RNNs become unable to learn to connect the information. In order to address this issue, researchers added a memory cell to these RNN units which led to the introduction of LSTMs (Long ShortTerm Memory). These networks are powerful and the most successful in the field of language modeling because they can not only remember contextual information and other relevant information located far away from the point where it is needed but they can also remember to forget. The idea of what to remember, how long to remember and when an information can be safely forgotten to make way for new information is modeled in this network with the help of memory cell, input, forget and output gates and thus makes it our choice of network for the given problem.

II. Dataset

For Part 2 of this assignment, we will be working with the Penn Tree Bank dataset. This dataset comprises of a vocabulary size of 10000 words. All words that fall outside this identified vocabulary are marked as <unk>. For the character dataset that was made available, the original data was just rewritten as sequences of characters, with spaces rewritten as '_' in order to facilitate training character-level models. For this assignment, we will be working with both word and character-level datasets.

III. Structure of Network

Long Short-Term Memory networks are kind of like a RNN that is capable of learning long-term dependencies. The key feature of LSTM is a cell state that imparts memory to the unit. Adding or removing information from the cell state is carefully regulated by structures called gates. They optimally let information through and are composed out of a sigmoid neural net layer and a pointwise multiplication operation. The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means let nothing through, while a value of one means let everything through. These gates are named as input gate, forget gate and output gate. Input gate decides which values we will update into the cell state. The tanh layer creates a vector of new values that could be added to the cell state. By combining the outputs of the tanh layer and input gate, we create an update to the cell state. The forget gate, on the other hand, decides which information from the cell state needs to be forgotten. The output gate decides what the output will be.

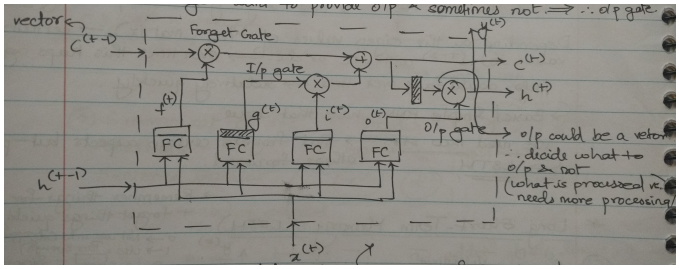

Fig. 1. LSTM Unit

The sigmoid layer first decides which parts of the cell state needs to be output and then the cell state is put through a tanh function and then multiplied with the output of the output gate. This way, we only output the parts we decided to. I have tried to fiddle around with the parameters in order to force the input gate to remain open or update the cell state or force the cell to forget certain information in order to formulate a parameter set to compute the counts of 0s in provided sequences.

IV. LSTM Cells for Counting Digits in a Sequence

A. Count number of all the digit 0 in the sequence

There are several suitable parameter sets to achieve this task. Since we just need to keep track of one input value, which is the count of the digit 0, a one dimensional internal state will suffice. One possible parameter set to solve the problem is as follows:

The input, output and forget gate are always open. This is achieved by providing a high value to their bias parameters bi, bf and bo. The input node gets the value 1 when it sees a 0, and value 0 when receives other digits. Thus, the internal state will be incremented by 1 every time the input node sees a 0.

Parameters:

Dimensions are: 10 x 1 (we have 10 digit values and 1 dimensional internal state hence 1 dimensional output)

Input Node:

wgx =[[100], [0], [0], [0], [0], [0], [0], [0], [0], [0]]

wgh = [0]

bg=[0]

For the parameters associated with the input gate, output gate and forget gate, all weights are zero and biases = [100]. This high bias ensures that the gates are always open.

B. Count number of 0 after receiving the first 2 in the sequence

Here, we need to keep track of two values, the number of digit 0 and digit 2. Only after we have seen the digit 2 once can we proceed with counting the number of digit 0s encountered in the sequence. Hence, the internal state is two-dimensional and so is the output dimension of every gate and input node.

The input node's output for the first dimension is 1 i.e., ON when it sees a 0 and the second dimension is 1 i.e., ON when it sees a 2. This is because both these information are useful and we need them and so we allow it. Now, the input gate will be ON only when it sees a 2 and shall remain ON thereafter. To achieve this, I've set both dimensions to high values of 100 when the input gate sees a 2 in order to move the gate to open or ON state. For any other value it sees, the output is set to -100. After it has seen a value of 2, it shall remain on thereafter. The input node value combined with the hidden state of input gate help in dictating which information needs to be updated. Now, the hidden state associated with the input gate i.e., "wih" ensures that after the input gate is ON, all the later 0s encountered in the sequence will result in the internal cell state counter (first dimension) to be updated to record the number of occurrences of 0. Both the states of Forget gate and Output gate are always ON.The parameters are as follows:

Parameters:

Dimensions are: 10 x 2 (we have 10 digit values and 2 dimensional internal state hence 2 dimensional output)

Input node:

wgx = [[100, 0],[0,0],[0,100],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0]

wgh = [[0,0],[0,0]]

bg = [0,0]

Input gate:

wix = [[-100, -100],[-100,-100],[100,100],[-100,-100],[-100,-100],[-100, -100],[-100, -100],[-100,-100],[-100,-100],[-100, -100]

wih = [[0,0], [200,200]]

bi = [0,0]

For the output and forget gates, all the weights are zero and biases = [100,100]. This high bias ensures that the gates are always open

C. Count number of 0 after receiving the 2 in the sequence, but erase the count after receiving 3, then continue to count from 0 after receiving another 2

Here, we need to keep track of three things, namely, the number of 0s, whether a 2 was encountered and whether or not a 3 was encountered. To solve this problem, a two-dimensional internal state is required. The first part of the problem is similar to the previous one i.e. Part B with a small addition to it. We need to erase the counts after receiving a 3, which means we need to use the forget gate to clear the cell state of any previous count information (number of 0s) that it holds. So we need to push the forget gate to a LOW state i.e, turn it OFF. This will ensure that the previous cell state will be erased. To achieve this, whenever a digit 3 is seen in the input sequence, we set weights to a low value of -100 for both dimensions which force the sigmoid output to 0. Thus, we achieve the "forget" part. The output gate is always ON in this case. The parameters are as follows:

Parameters:

Dimensions are: 10 x 2 (we have 10 digit values and 2 dimensional internal state hence 2 dimensional output)

Input node:

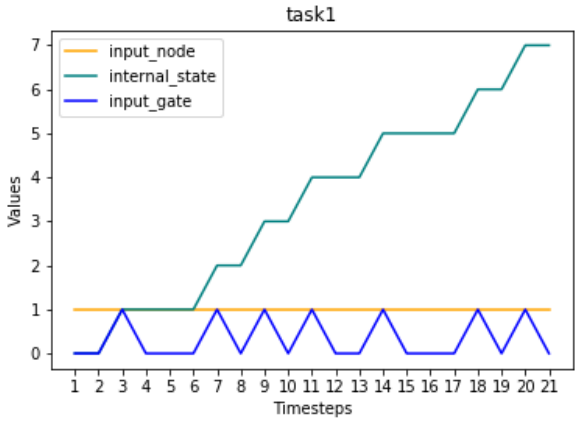

Fig. 2. Task 1 (Input node, Internal state and Input gate)

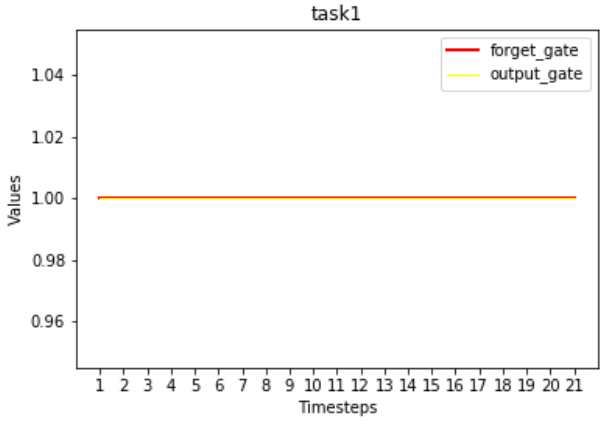

Fig. 3. Task 1 (Forget Gate and Output Gate)

wgx = [[100, 0],[0,0],[0,100],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0],[0,0]
wgh = [[0,0],[0,0]]
bg = [0,0]

Input gate:

wix = [[-100,-100],[-100,-100],[100,100],[-100,-100],[-100,-100],[-100,-100],[-100,-100],[-100,-100],[-100,-100],[-100,-100]
wih = [[0,0], [200,200]]
bi = [0,0]

Forget Gate:

wfx = [[100,100],[100,100],[100,100],[-100,-100],[100,100],[100, 100],[100,100],[100,100],[100,100],[100,1(
wfh = [[0,0],[0,0]]
bf = [0,0]

For the output gate, weights are all set to zero and bias = [100,100]. This high bias ensures that the gates are always open.

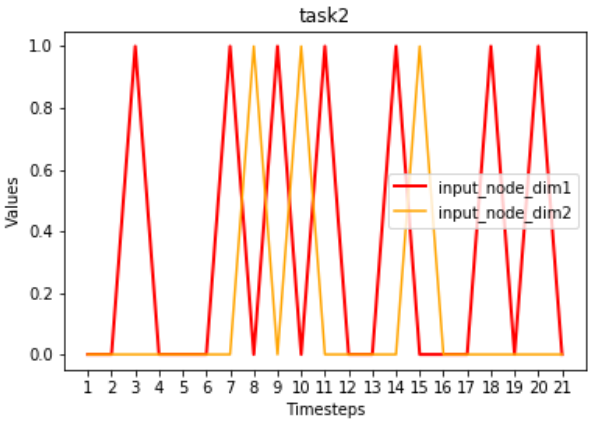

Fig. 4. Task 2 (Input Node)

D. Plots

The plots shown in Figure 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 detail the the internal cell states and states of the input, output and forget gates as observed while processing the given sequence [1, 1, 0, 4, 3, 4, 0, 2, 0, 2, 0, 4, 3, 0, 2, 4, 5, 0, 9, 0, 4] for Task 1, Task 2 and Task 3.

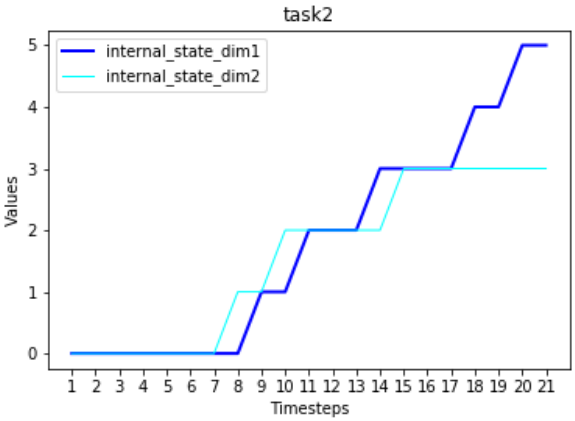

Fig. 5. Task 2 (Internal State)

V. Language Modeling with RNN and LSTM units

A. Data Preprocessing

The PTB dataset is a relatively clean dataset and does not require much of preprocessing to be done. The text is already tokenized. There were two models trained i.e. the word model and the char model.

1) Word-Level Model:: As an initial preprocessing step, all the newline characters were replaced with <eos> tags. As already mentioned, the dataset comprises of a vocabulary of size 10000 and all words that fall outside this vocabulary are marked unknown with the tag <unk>. The model was trained with the unknown tags intact.

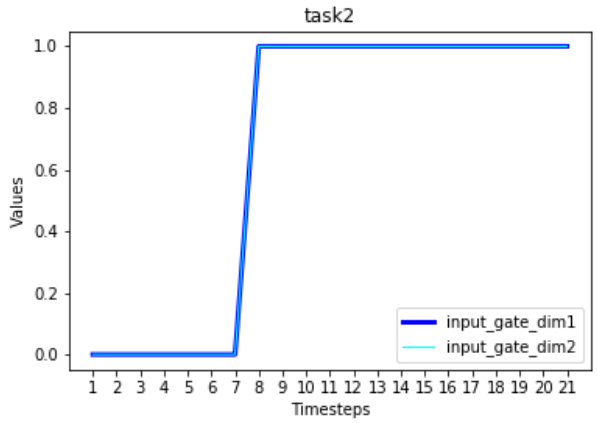

Fig. 6. Task 2 (Input Gate)

Fig. 7. Task 2 (Forget Gate)

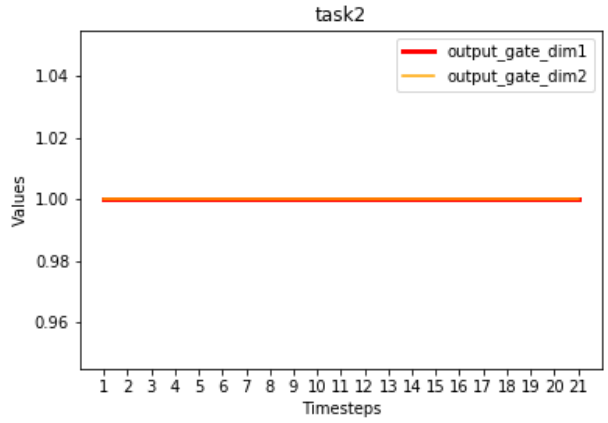

Fig. 8. Task 2 (Output Gate)

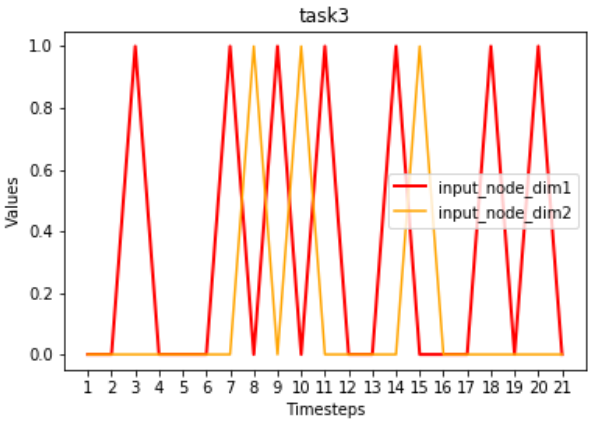

Fig. 9. Task 3 (Input Node)

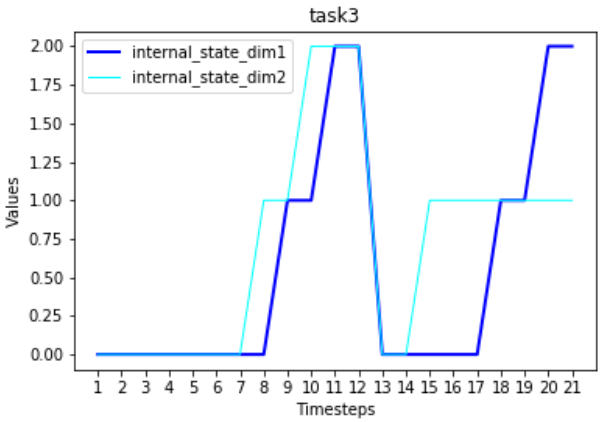

Fig. 10. Task 3 (Internal State)

Fig. 11. Task 3 (Input Gate)

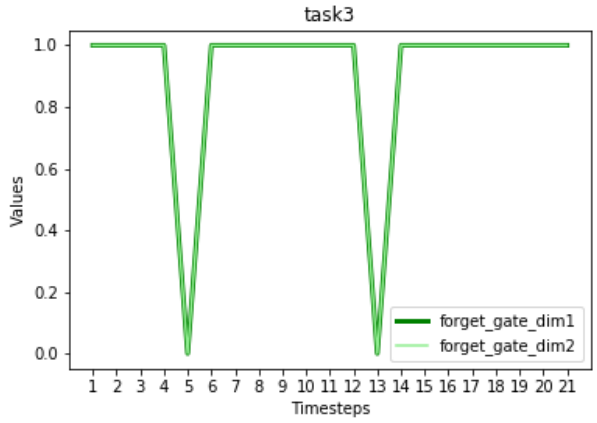

Fig. 12. Task 3 (Forget Gate)

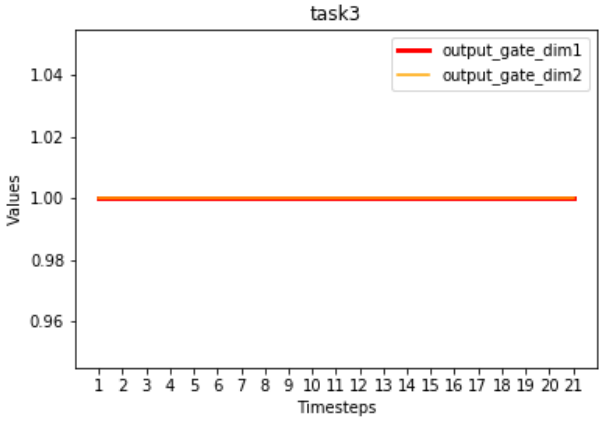

Fig. 13. Task 3 (Output Gate)

2) Character-Level Model:: As an initial preprocessing step, all the newline characters were replaced with a dot ".". The rationale behind this idea was that since the model is being trained on characters rather than words, a newline character could be learned as something that belongs to a word rather than an end-of-line marker. This might interfere with the text generated later on. So in order to aid the network to distinguish between an end-of-line marker and a possible character that could be a part of a potential word, I replaced all newline characters with dots.

Additionally, all the spaces between characters were removed as they weren't essentially spaces but just a character delimiter. The underscores represent the space between words and they were retained as replacing them with spaces will have the same effect because what we finally feed to the model is just a vector representation of character indexes. All the other special characters were also retained because, in the context of natural language, they all make sense. Since, it is a sequence generation problem, all those special characters add meaning to the text as opposed to text classification problem where we would usually want to filter out such characters.

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 32, 300)	14700
lstm_1 (LSTM)	(None, 32, 300)	721200
dropout_1 (Dropout)	(None, 32, 300)	0
lstm_2 (LSTM)	(None, 32, 300)	721200
dropout_2 (Dropout)	(None, 32, 300)	0
time_distributed_1 (TimeDist	(None, 32, 49)	14749
activation_1 (Activation)	(None, 32, 49)	0

Total params: 1,471,849
Trainable params: 1,471,849
Non-trainable params: 0

Fig. 14. RNN Architecture for Character-Level Model

Layer (type)	Output Shape	Param #
embedding_9 (Embedding)	(None, 32, 300)	3000000
lstm_17 (LSTM)	(None, 32, 300)	721200
dropout_17 (Dropout)	(None, 32, 300)	0
lstm_18 (LSTM)	(None, 32, 300)	721200
dropout_18 (Dropout)	(None, 32, 300)	0
time_distributed_8 (TimeDist	(None, 32, 10000)	3010000
activation_8 (Activation)	(None, 32, 10000)	0

Total params: 7,452,400
Trainable params: 7,452,400
Non-trainable params: 0

Fig. 15. RNN Architecture for Word-Level Model

B. Recurrent Neural Network (LSTM) Architecture

The proposed neural network architecture comprises of an embedding layer with an embedded vector-space dimension of 300, two LSTM layers (tanh activation) with 300 units each, followed by a time-distributed dense layer (with units = size of the vocabulary) with softmax activation. A dropout layer with a dropout rate of 0.5 was added after each LSTM layer in order to ensure that the model generalizes well instead of overfitting. We compute the categorical cross entropy loss and use this to evaluate the perplexity of the model. The same model architecture was used for Word-Level and Character-level models. Detailed model summaries are shown in Figure 14 and 15.

C. Hyperparameters

The following Hyperparameters were used in the training of the two models:

- **LSTM Layers:** 2 LSTM layers with 300 hidden units each.

- **Dropout Layer:** A Dropout layer follows each LSTM layer. There are 2 dropout layers in total with dropout rate of 0.5.
- **Activation Function:** For the LSTM layers, I have used default tanh activations and for the final Dense layer, I have used softmax activation.
- **Optimizer:** Adam optimizer with learning rate of 0.001 and decay of 0.0.
- **Epochs:** 50
- **Batch Size:** For both word-level model and character-level model I have used a batch size of 32.
- **Input Length:** For word-level model, the input sequence length is 32 words and for character-level models it is 100 characters.
- **Embedding Layer:** The Embedding layer is defined with a vocabulary of 10000 for word-level model and 49 for character-level model, a vector space of 300 dimensions in which words/characters will be embedded, and input sequences that have 32 words/100 characters each.

D. Evaluation

I have used perplexity as a metric to evaluate the generated results. Perplexity is a popular evaluation metric used for Language modeling. It is the average branching factor in predicting the next word. In other words, perplexity is the exponentiation of the cross entropy loss. Less entropy (or less uncertainty) is favorable over more entropy as predictable results are preferred over randomness. Hence, low perplexity is considered better than high perplexity.

E. Plots and Results

The Figures 16 and 17 are the plots of the obtained perplexity vs the epochs. In case of word-level model, we note that beyond some 15 epochs, the evaluated perplexity on the validation set becomes consistent and increases slightly starting from 30 epochs. Though it doesn't increase by a very large amount. For the character-level model, we achieved a decent perplexity value of 2.98 and the perplexity gradually decreases with increase in epoch which is something we would want and hence this model is performing as expected. The perplexity on the validation set after 50 epochs of training came out to be 233.11 for the word-level model and 2.98 for the character-level model.

Figures 18 and 19 show the text generated by both the models. A starting seed value was randomly picked from the provided test data. From Figure 19, we see that the generated text contains a lot of <unk> tags which could be improved upon with more training. With 50 epochs, the word-model has done a decent job but better results can be obtained with more parameter tuning and training over some 200-300 epochs. The generated text from character-model as seen in Figure 18, show that the model is performing fairly well. Though it is not able to generate text that is grammatically correct, many of the words have correct spelling which is a decent performance given the model was trained for only 50 epochs.

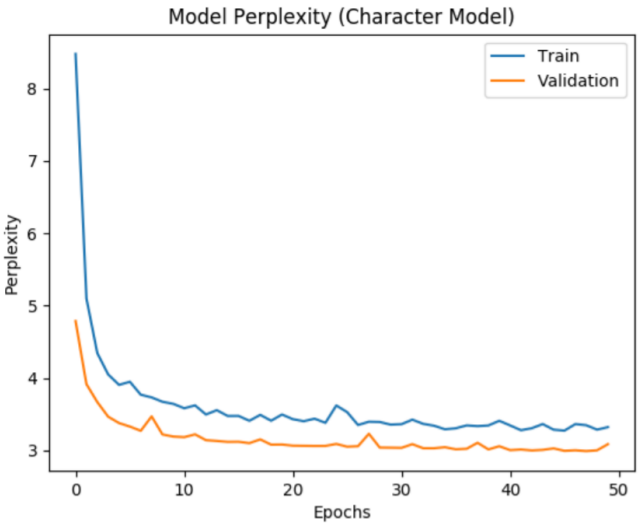

Fig. 16. Character-level Model (Perplexity vs Epochs)

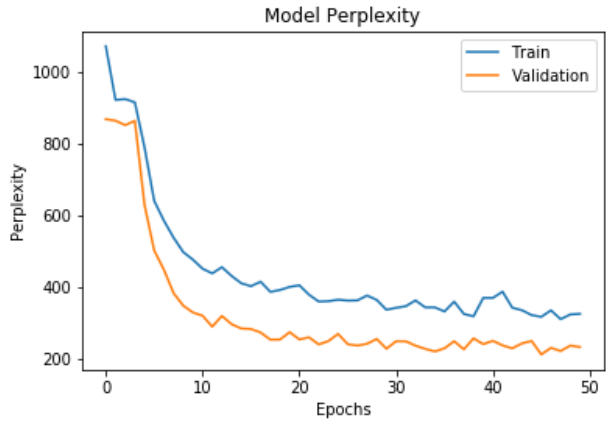

Fig. 17. Word-level Model (Perplexity vs Epochs)

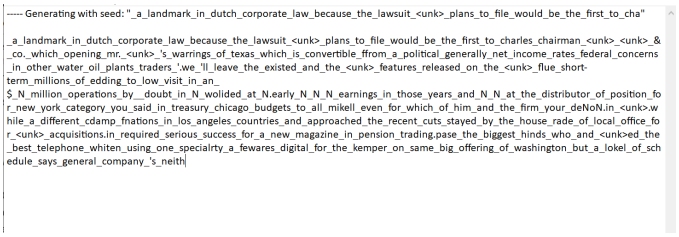

Fig. 18. Character-level Model: Generated Text

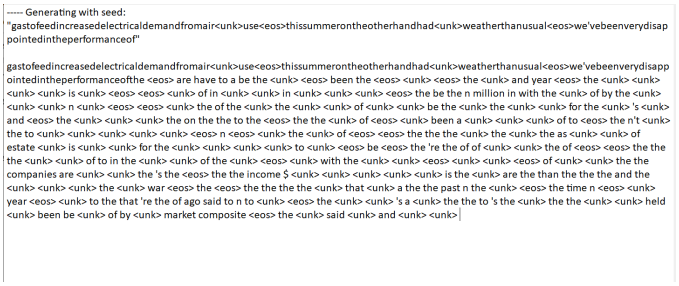

Fig. 19. Word-level Model: Generated Text

VI. Conclusion

I have successfully implemented the Recurrent Neural Network and LSTM cells for sequence counting and language modeling. I have tried tuning the set of hyperparameters to produce the best accuracy and perplexity for the dataset and observed the results generated to make inferences. The completion of this assignment helped me understand the working of RNN and LSTM, its efficiency in producing the desired results, the effect of the hyperparameter values on the network and the difficulties in implementing the algorithm. My network is behaving as expected.

Acknowledgement

I've taken the help of the NCSU EOL lectures of ECE 542 Fall 19 Neural Networks for the completion of this assignment.