# Recurrent Neural Networks:
# Implementing LSTMs for Text Classification, Time Series Predictions, & Nursery Rhyme Generation

## ABSTRACT

LSTMs (Long Short-Term Memory Networks) are a particular type of recurrent neural network (RNN) that supports the capability of learning and understanding long-term relationships in our feature space in a much stronger manner than typical feed-forward networks. These types of neural networks are trained on very large amounts of sequential data (time series, NLP, etc). Because these models are innately trained on these vast, complex datasets, the model needs to have the ability to remember past information and use it to inform future predictions. In order for LSTMs to be capable of utilizing past and current information to make future predictions, they depend on a complex backend architecture that's comprised of different 'gates' and 'cell states.' This report aims to discuss the intricacies of the LSTM architecture and analyze its efficacy in three different use cases, while also exploring the different ways in which these predictions are made in the backend.

## INTRODUCTION

As is the case with all machine learning algorithms, the objective of our LSTM algorithm is to 'learn' information in our dataset, however, in this case, the model is attempting to learn and understand sequential relationships. As aforementioned, LSTMs are trained on vast and complex sequential datasets and are trained like any other machine learning algorithm. However, instead of learning a boundary or set of parameters that minimizes loss, LSTMs learn to recognize and model patterns present in our dataset by adjusting their internal parameters. Similarly to most algorithms, there are a multitude of ways (hyperparameters) that alters how the model learns, which directly impacts the model's performance. The background section will go in-depth about all of LSTMs' backend intricacies, why they work, and what use cases are ideal for them. The discussion section outlines the outcomes across all of our experiments, analyzes why particular architectural decisions were made, and discusses the efficacy of our outcomes. Finally, the conclusion section will reiterate our findings and reiterate the strength of our implementation.

## BACKGROUND

As mentioned previously, the backend architecture of LSTMs is much more complex and intricate than its feed-forward neural network counterparts. This is particularly because all RNNs have a feedback mechanism. This essentially means that instead of a typical FFNN that the data is sequentially passed through layers, RNNs take the output from previous timesteps as input to the current timestep, which directly allows the model to maintain a memory of past inputs and process them in a way that the model understands the context. In the abstract, I mentioned that LSTMs are comprised of different gates and cell states that allow these long and short-term dependencies to be understood, which are directly responsible

for allowing our model to understand and process previous outputs while also handling the issues of exploding/vanishing gradients.

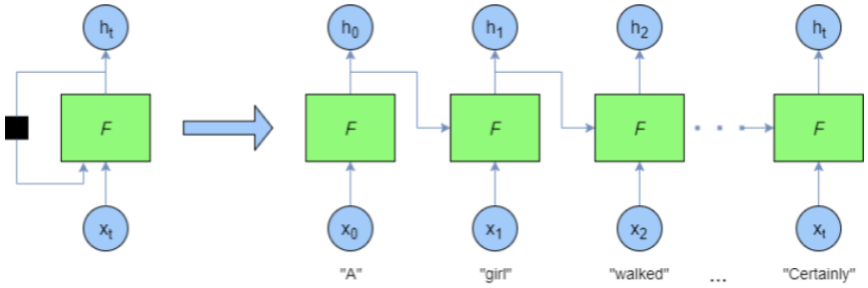


**FIGURE 1: Visualization of RNN**

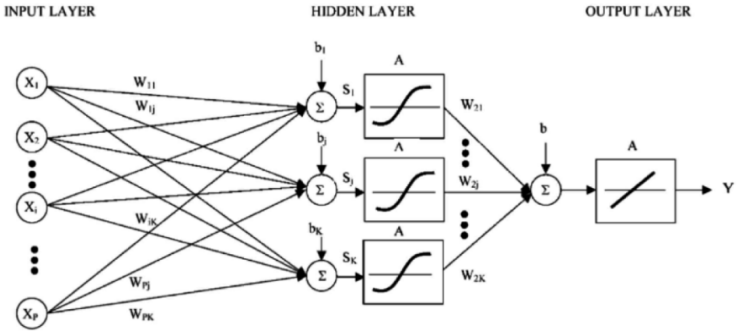


**FIGURE 2: Visualization of FFNN**

Looking at figure 1, each output of our model's iteration is taken with respect to some sort of temporal/sequential piece of information (time series, NLP, speech recognition, etc), while figure 2 depicts a more simple and typical feed forward neural net. Now, let's discuss more of the backend architecture. LSTMs are comprised of three gates, the input gate, forget gate, and the output gate. The input gate is primarily concerned with what encoded information from the current input should be added/'remembered' by the memory cell state. The cell state is the vector pertaining to the long-term memory, essentially acting as the features in a machine-understandable format that understands long-term relationships. The forget gate does the inverse, it uses a sigmoid activation on the input and hidden layer and determines what the cell state should remember or forget for future iterations. Finally, the output gate also computes a sigmoid activation of the input and current hidden state to determine which particular parts of the memory cell state that should be used in order to generate an appropriate output (which elements of cell state to output). Through the implementation of these multiple gates and cell states that rely on differing inputs and activations, LSTMs are better able to understand these temporal and sequential relationships.

Now that there is a general understanding of what the architecture looks like behind the scenes, I will be discussing how these models are trained. In order for the algorithm to more accurately understand the input, we feed the model-encoded vectors that account for the context with respect to the particular time step at hand. In order for the model to understand these vectors during training, we utilize backpropagation through time, a type of backpropagation that adjusts the weights of the LSTM's internal layers at each time step. It works by recursively applying the chain rule and uses attention (or multiheaded attention) in order to learn to attend to different parts of the input at different time steps.

## DISCUSSION

Let's now apply what was discussed above to our particular case at hand. Our first example was using LSTMs for the purpose of multi-class text classification. This dataset is comprised of a class (a single word describing the sentence) and an example sentence. The class is essentially a single word that describes a sentence and the model is attempting to learn the subject of a given sentence given the dataset we are training it on. The model used here is comprised of an embedding layer (to encode the information), a dropout layer, a bi-directional LSTM layer, and finally a dense layer with softmax activation to classify our input. After standardizing our input by making all characters lowercase, removing stopwords (words that are commonly occurring that don't add much value in terms of classification), and encoding these words, we are able to train a pretty accurate language classification model. After training the model on the dataset, I fed it two example sentences for the model to classify. One is primarily concerned with politics and the other pertains to manufacturing jobs and general business health. From our model's output (located in article1Output.txt and seen below) our model correctly classified both sentences' sentiments correctly.

The next example of LSTM I built was for the purpose of time series analysis. In this particular case, we are attempting to use the LSTM as a solution to a regression problem. Essentially we are trying to predict the number of passengers (in thousands) next month given the current and previous months. Here we built a more simple model with only two layers, an LSTM layer, and a singular dense layer. Because the problem statement is less computationally expensive, we don't need to build too complex a network. Looking at our model's output, we see that our model did a pretty good job of fitting both the training and testing sets. However, this model is not perfect, as our training RMSE was notably lower at 22.68, compared to our testing RMSE of 49.35. While not perfect and slightly overfit, our model was still fairly proficient at predicting future passenger amounts.

The next model built was predicting the same output as the model above but was trained using the window method. This slightly decreases the training time as we are no longer stochastically training our model but increases our computational requirements. However, without tuning the window size and network architecture, we get some slightly less optimal results. Our model got a train RMSE of 24.86 and a test RMSE of 70.48. With further testing and hyper-parameter tuning, we could potentially see better results but currently do not given our current implementation.

The next time series example I made using the same dataset for an LSTM for regression with time steps. In this implementation, instead of placing the past observations as distinct input features, they are used as

time steps o the one input feature. This breakdown of the feature space allows for more accurate framing of the problem at hand. As a matter of fact, you can see that in our model's output. We were able to increase the accuracy of these predictions to where the train RMSE was 24.84 and our test RMSE is 60.98.

Following this implementation, I built an LSTM model that uses memory between each batch. Essentially what this does is gives the engineer more control over when the state within the model is reset. Instead of automatically resetting after each training batch, we are able to build the state of the entirety of the training sequence. The thought behind this concept is that some sequential relationships require more time and training iterations in order to truly understand the context of the input with respect to the temporal nature of the dataset. When we implemented this model, we saw again, increased results from the model prior. Here, we saw a training RMSE of 24.48 and a testing RMSE of 49.55.

The final implementation I built out for this particular use case and dataset was stacked LSTMs with memory between batches. The purpose of the memory between batches was mentioned above, however, the stacking of LSTMs offers interesting potential. Essentially this deepens the architecture and allows the model to understand a larger amount of relationships and is able to be trained on deeper and more complex datasets/dictionaries. Keeping the hyperparameters constant across all of these similar, but slightly nuanced architectures, we see a decrease in performance. We receive a training RMSE of 20.58 and a testing RMSE of 55.99. These figures could be improved with additional training epochs, however, for the nature of this assignment I kept these hyperparameters constant across all the different architectures.

Finally our nursery rhyme LSTM. The loss values are associated with how frequently any given word is associated/present around similar words (this is done through a covariance matrix). Here we are training the model on numerous amounts of nursery rhymes with the hope that our algorithm will be able to pick up on the temporal dependencies and relationships present in these nursery rhymes and reproduce one on its own. In this use case, we are using LSTMs for natural language processing and sentiment analysis in order to reproduce these nursery rhymes. Here, I opted to build an LSTM with three layers, and embedding layer, bidirectional LSTM layer, and obviously a dense layer for our softmax activation. The sample nursery rhyme our model computed is at the very end of RhondaNursery.txt. While it is definitely not the most soothing, grammatically correct, nor enticing nursery rhyme, with how simple the architecture is, I'm pleasantly surprised with its results.

## CONCLUSION

Through the various experiments and proven use cases throughout this report, my hope is, that it has become evidently clear the power and ingenuity of the LSTM architecture. Not only is it capable of handling and excelling in more tasks than simpler networks, but also performs better at a wider variety of implementations. Although I was unable to figure out the necessary steps to output my pyplots outside of a notebook environment(tried to use matplotlib.savefig() but wasn't able to figure that out fully), the metrics provided explain and prove the efficacy of our implementations across our use cases are sufficient and optimal at handling minor temporal and sequential datasets and problem statements.

## REFERENCES


Brownlee, J. (2017, July 19). A Gentle Introduction to Long Short-Term Memory Networks by the Experts. Machine Learning Mastery. https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

Brownlee, J. (2019, April 26). Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras. Machine Learning Mastery. https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/

Djaja, F. (2020, June 9). Multi Class Text Classification with Keras and LSTM. Medium. https://djajafer.medium.com/multi-class-text-classification-with-keras-and-lstm-4c5525bef592

Olah, C. (2015, August 27). Understanding LSTM Networks -- colah's blog. Github.io. https://colah.github.io/posts/2015-08-Understanding-LSTMs/

matplotlib.pyplot.savefig — Matplotlib 3.7.1 documentation. (n.d.). Matplotlib.org. Retrieved April 21, 2023, from https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.savefig.html

tf.keras.Model | TensorFlow Core v2.5.0. (n.d.). TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/Model