Using RNN to Predict Stock Market Prices

Ford Smith

March 26, 2019

1 Introduction

Machine learning is a major subset of Artificial Intelligence where the machine is able to adapt to information and study it. However, different problems require different methods of machine learning, which has several different subsections of tools that can be used to teach a machine from previous experiences by adjusting to inputs to "perform human-like tasks". In this particular experiment, Neural Networks will be used. Neural Networks are very powerful tools because they grant flexibility to many different scenarios depending on how they are constructed.

History At its core, machine learning is all about taking in inputs and creating a generalized function that can then be extended to predict future values. In fact, humans have been doing basic variations of this for hundreds of years: linear regressions. While some functions are too difficult to create, a line of best fit can be found on data with an input (x) and output (y) to create a function that can approximate or predict a different set of data. See Figure 1. [1]

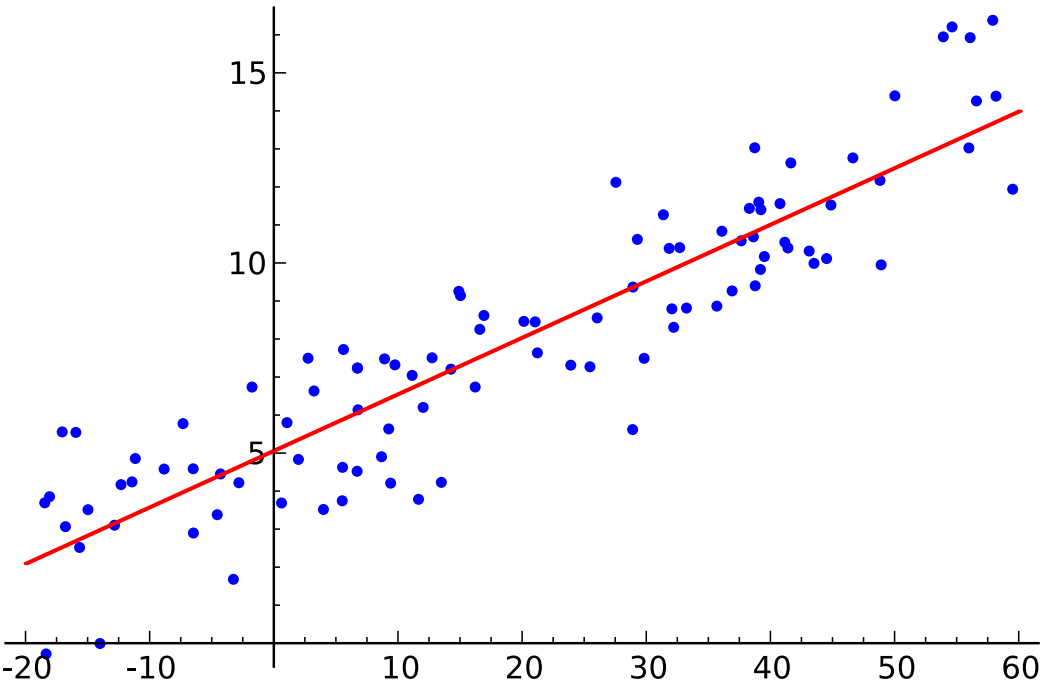


Figure 1: An example of linear regression used on input data to create a function to approximate more data

This type of learning is called supervised machine learning because through the process of approximating the function, input and output data can be used to make the approximation much more accurate. Unsupervised learning, although not important in this experiment, is learning where there are not predetermined output values so different learning methods are used to learn more about the inputs and how it is structured. [2]

Skip forward about 150 years to the 1950s, there is the development of the Perceptron. The Perceptron was created by Frank Rosenblatt to be a mathematical model of the neurons in our brains. It takes a set of inputs from nearby Perceptrons, multiplies each of the inputs by a value weighted, and will output a 1 if the weighted inputs reach a threshold, otherwise a 0. This was massive back then because Perceptrons could create basic OR/AND/NOT gates. This was the gateway to formal logic for computers. The most exciting part, however, was that this model could learn by slightly adjusting the weights whenever the output is too low/high. It followed this

general order: ”

1. Start off with a Perceptron having random weights and a training set
2. For the inputs of an example in the training set, compute the Perceptron's output
3. If the output of the Perceptron does not match the output that is known to be correct for the example: If the output should have been 0 but was 1, decrease the weights that had an input of 1. If the output should have been 1 but was 0, increase the weights that had an input of 1.
4. Go to the next example in the training set and repeat steps 2-4 until the Perceptron makes no more mistakes”

[1]

This procedure is simple, not terribly computationally heavy, and works best when there is only a limited set of outputs due to thresholds. This is exactly what classification requires - out of a set, accurately determine which output the input data is. Rosenblatt implemented the Perceptrons and inputted 20x20 inputs to accurately classify shapes. Although simple, when many Perceptrons are put together in series called **layers**, the computation power becomes much higher and can work on much more complex data. The same happens with multiple layers, where data is put into the input layer, which feeds its output to “hidden” layers which pass their information on to subsequent layers until the information is passed to an output layer. The importance of these hidden layers stems from their ability to find features within the data. Features are essentially specific patterns the model layer is trained to find in the data. For instance, if the network was trained to identify whether there was a cat in the image, the layer could identify whether the image has a cat mouth or not. This, in turn, allows it to become more accurate. However, there becomes several serious issues when done. First, Rosenblatt's method of training does not work with multiple layers. The Perceptrons also can't work on complex features because of the extreme linearity of the weight function inside the Perceptrons.

Modern Networks **Backpropagation** was developed across multiple researchers in the 60s and is still commonly used. With backpropagation, there

is a cost function C with respect to the weights w in the network. We can then use multivariable calculus to find how sensitive the ouput is to nudges in the various weights. The reason backpropagation is so important was because it not only became a much faster learning algorithm, but also describes detailed information about the change in the system in respect to the weights.

With a newer, faster, and all around better learning algorithm, there is still bottle-necking by the actual Perceptron itself. When there is a tiny change in its weights, it could cause a drastic change in the output, which actually makes training more difficult. If there is more fine control of the weights to have more fine control over the output, it is possible to more accurately train layers to get the desired results. This is where the sigmoid function comes in handy. If the threshold is replaced with the sigmoid function, as pictured below, there will be more fine change in the output with the change in the weights, giving a higher degree of accuracy. Since we can finely determine the change in the sigmoid, we can use it to calculate a slightly more accurate weight.

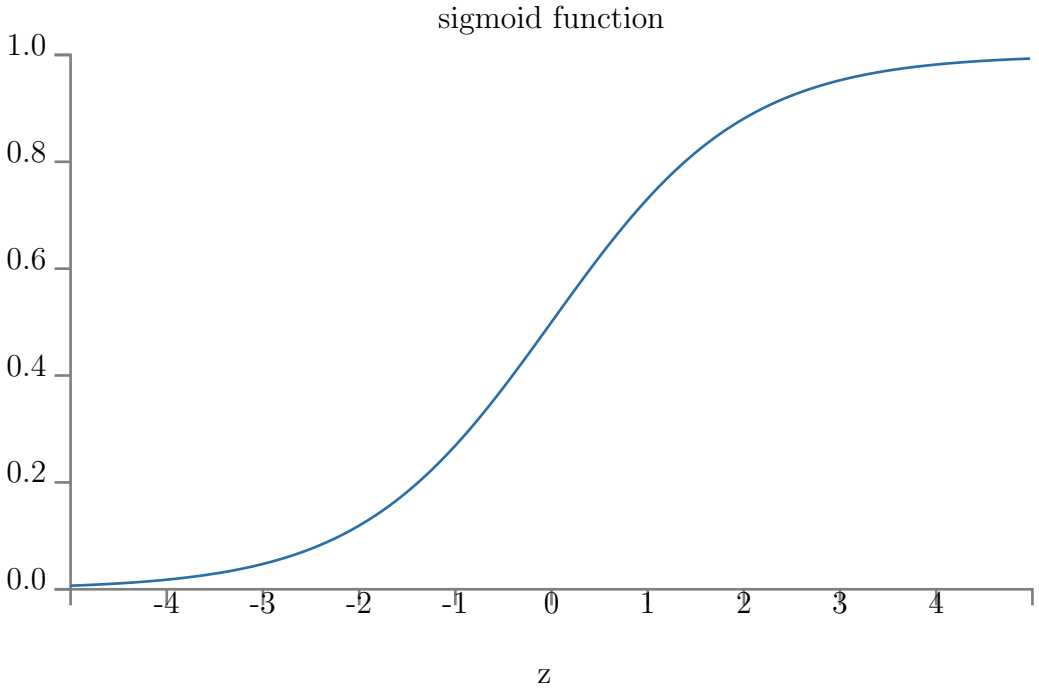


Figure 2: An example of a sigmoid function

You may be wondering, why use the sigmoid function instead of just having the threshold? Well, the threshold is very linear and doesn't give much opportunity to learn around the data. When the **activation function** is non-linear (like the sigmoid function) it allows the network as a whole to better adapt to the data. Note, the activation function is what is used to determine the output of the nodes. Here is an image of what a complete Neural Network will use as Figure 3. [3]

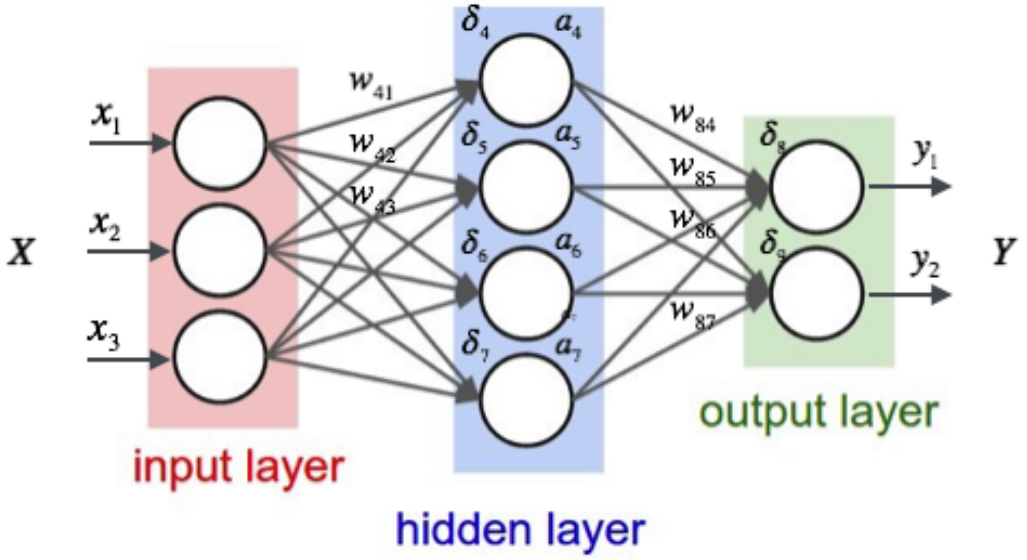


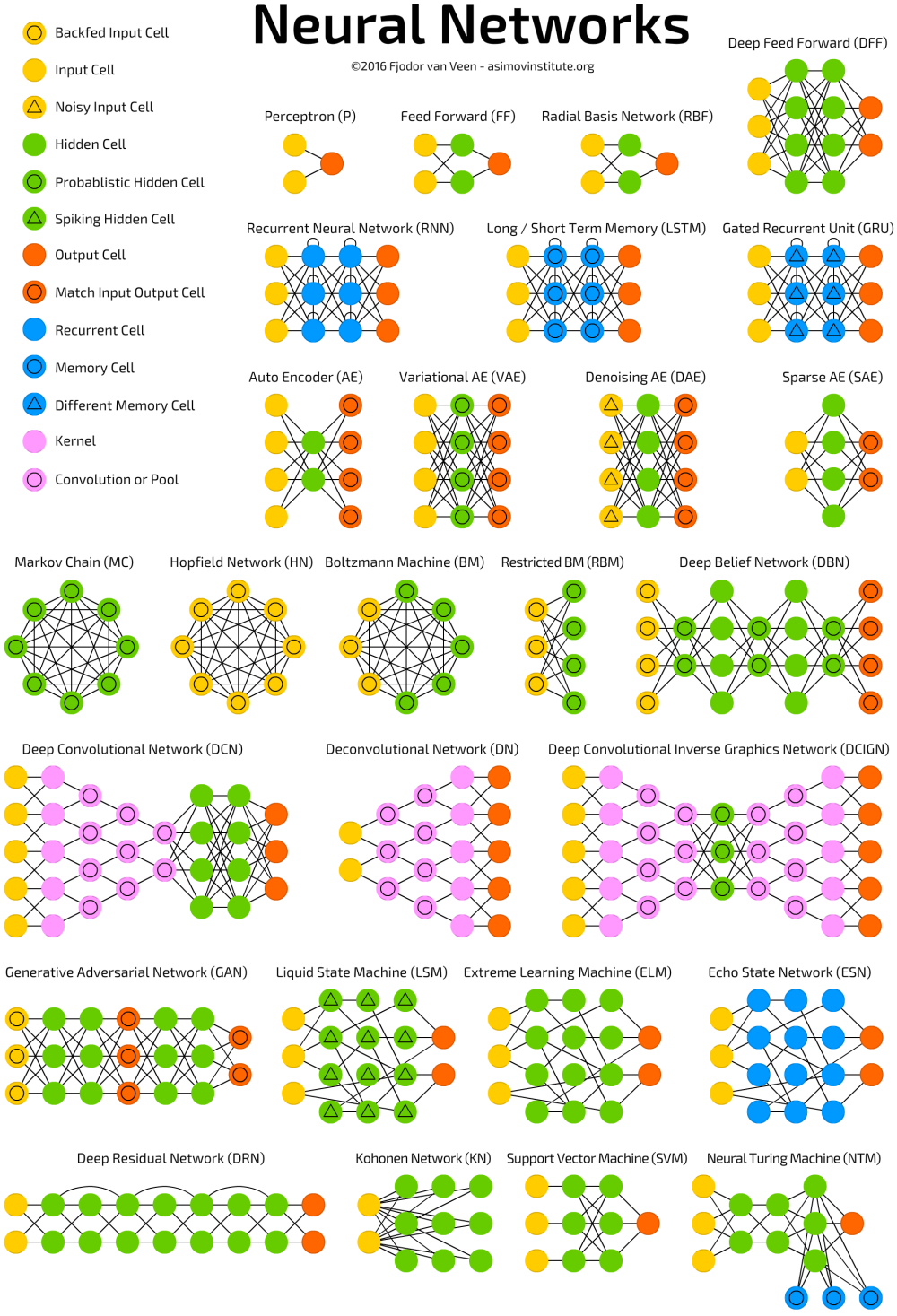
Figure 3: A neural network [4]

2 Different Types

Having discussed what a Neural Network (NN) is and how it was developed, it is important to discuss the different types of Neural Networks to best decide which will be used for this experiment. After reading what follows, it will be clear which type of NN will be chosen and why it is the only NN that can work. The main types are demonstrated in a graphic by Fjodor van Veen from Asimov Institute. [5]

A mostly complete chart of

Neural Networks



Choosing There are many different options to choose from, however, the majority of them can't be used. Why? In this particular experiment, NN are being used to predict stock prices. The stock market doesn't arbitrarily choose the price every second, instead, the price is a derived from the previous prices to create a new price (it is also important to note stocks are very dependent on external factors). This *requires* the NN to remember previous information to make informed decisions. As a result, only Recurrent Neural Networks (RNN) can be used. Something to note, GRU and LSTM are a subsection of RNN not demonstrated in the figure above. [6] [6]

RNN What is so special about RNN? To put it strictly, RNN are a special type of NN where the previous information directly affects the result of the next step in a sequential order. While in theory they are able to have arbitrarily long pieces of sequential information, a generic RNN is unable to do so. This is very bad in the context of running it on stock data as stock data can be over decades. This is why LSTM - long short term memory - are used. [7]

LSTM While generic RNN struggle to remember long term information, LSTM are designed to do so making them the perfect candidate for stock market data! Below, is the break down of exactly how the LSTM is able to do so.

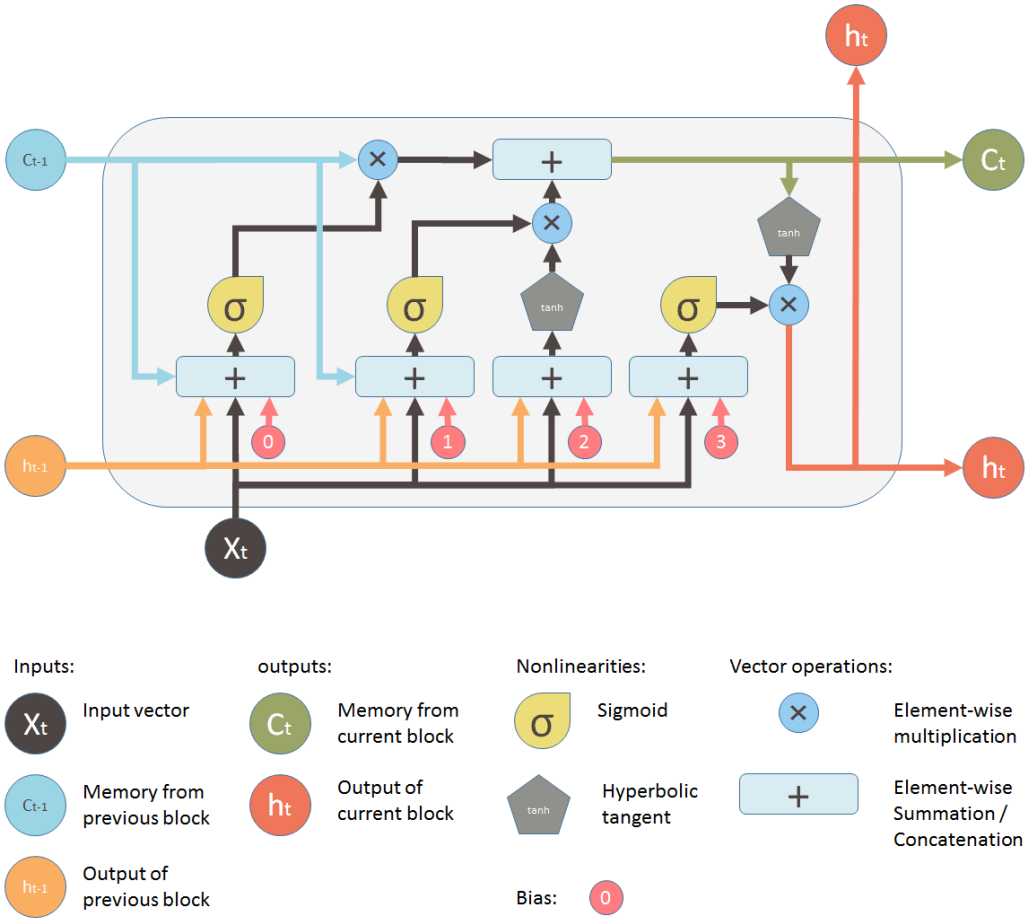


Figure 4: Internals of LSTM cell

Although it looks very intimidating, the operations taking place are fairly straight forward. First, notice what is going in and out of this basic LSTM cell. X_t is the input at the current step, h_{t-1} is the output from the previous LSTM cell and C_{t-1} is the "memory" of the previous cell. Lastly, we have h_t for output of the current network and C_t is the current memory of this cell.

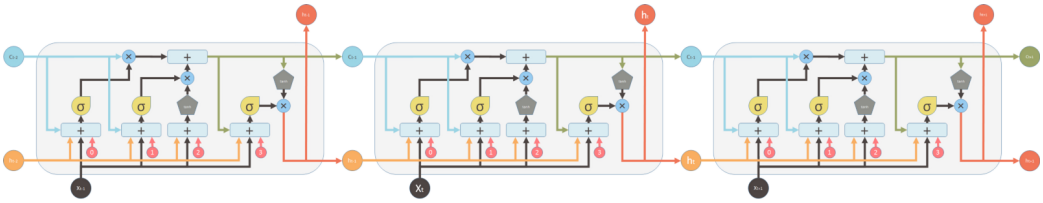


Figure 5: Multiple LSTM strung together in a chain

Here are several cells put together to visualize a segment of what is going on. The top bar (where C_{t-1} is passed) is the transfer of the memory through the network. In Figure 6 (below) we are looking at what is the "forget" gate. The purpose is to use a single layer NN with a sigmoid called activation function which is than applied to the old memory. As a result, it can affect the memory that is to be passed on (which is how the network doesn't get bogged down by trying to remember every feature).

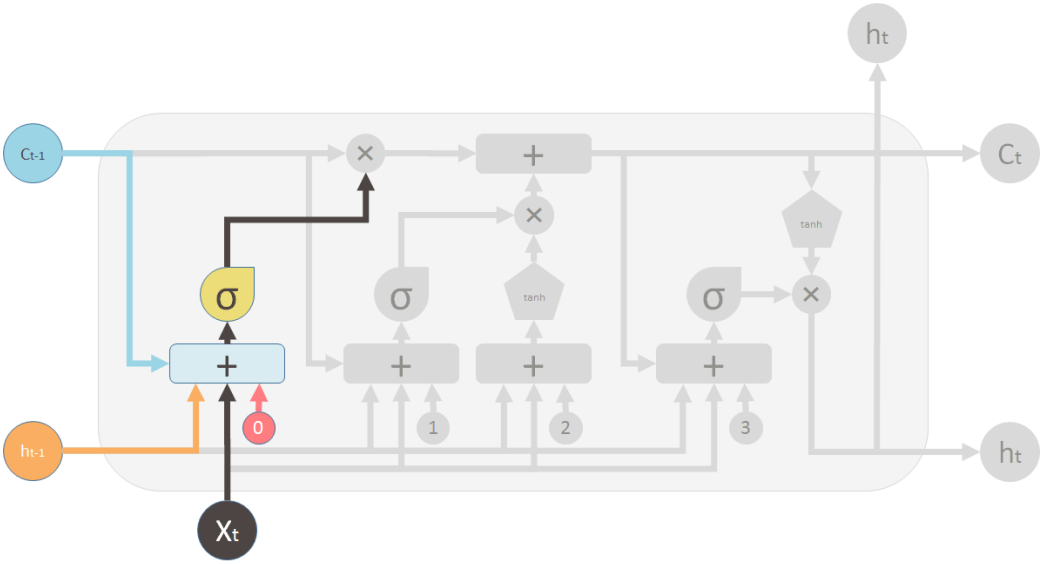


Figure 6: Multiple LSTM strung together in a chain

Afterwards, we go to the "new memory" single layer NN. This NN determines how much the new memory should influence the old memory. However, the new memory is generated by a different single layer NN, this time with tanh as the activation function. The outputs of both of these is combined and applied to the transferring memory.

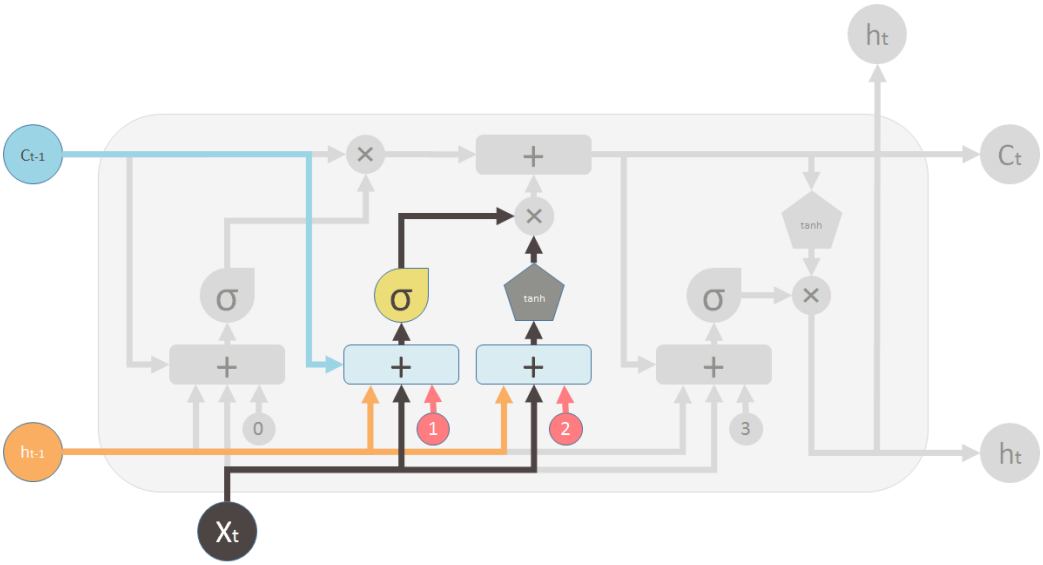


Figure 7: First section of LSTM

Lastly, we need to actually generate the output for this cell. This step once again includes a single layer NN that determines how much the new memory should affect the output of the cell.

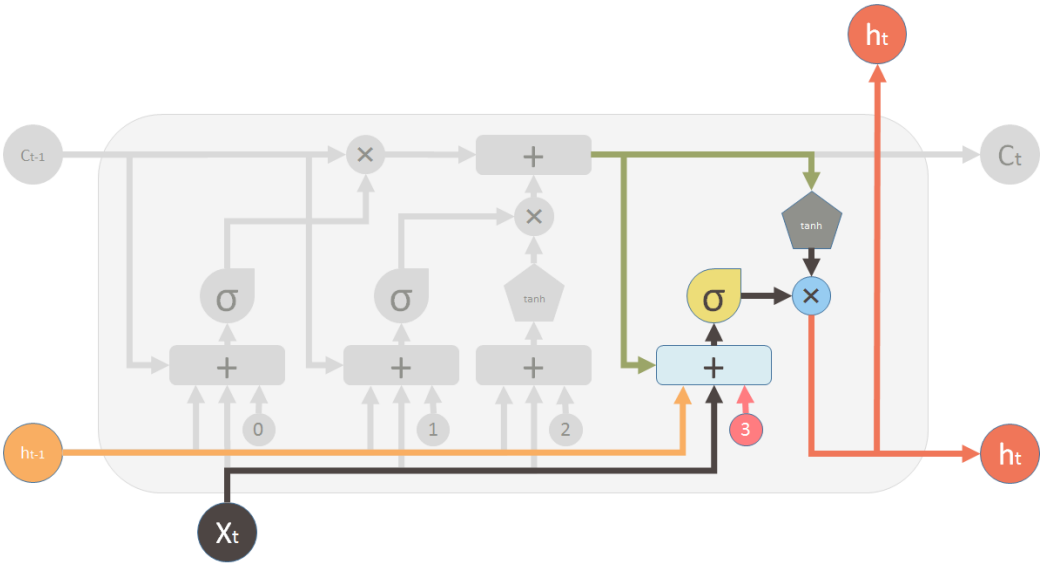


Figure 8: Last section of LSTM

All from [8]

Final Notes In total, because of the ability to be trained to forget unimportant information but still remember the past, it becomes the strongest RNN to use for this project. However, there is one major issue to be aware. Suppose we have a simple NN that we want to use to identify if we have a cat in an image. If we pass the same image to the NN over and over again, with no variation, the NN will become **overfitted**. Overfitting is simply when the NN trains to conform to the test data only. In this case, it could become perfectly accurate at knowing if that specific picture has a cat, but will be unable to accurately determine if any other photo has a cat. One way to counter this is to use dropouts. Dropouts will drop a percentage of the data in order to force more variability in the data being passed in. Although using dropouts seems counter intuitive, especially in the case of stock market prices where order matters (which will be tested), they can bring significant accuracy gains. [9]

3 Setup

Now that we have selected the RNN to use, we can go through and configure the model. All code is on github at: https://github.com/Programatic/StockMarketAI Just a note, all code is in the form of a Jupyter notebook in vscode which allows for more modular development with data science tools.

4 Results

For the first test, the model being used is going to be LSTM $\rightarrow$ Dropout $\rightarrow$ LSTM $\rightarrow$ Dropout $\rightarrow$ LSTM $\rightarrow$ Dropout $\rightarrow$ LSTM $\rightarrow$ Dropout $\rightarrow$ Dense. The dense layer just takes the information from the previous layers and condenses it to a dimension we want, in this case 1 (as in 1 output value for the price of the stock).

Figure 9 is a graph of the full TATA Motors stock. TATA is a motor company from India and it is simply much easier to get stock data for free from India.

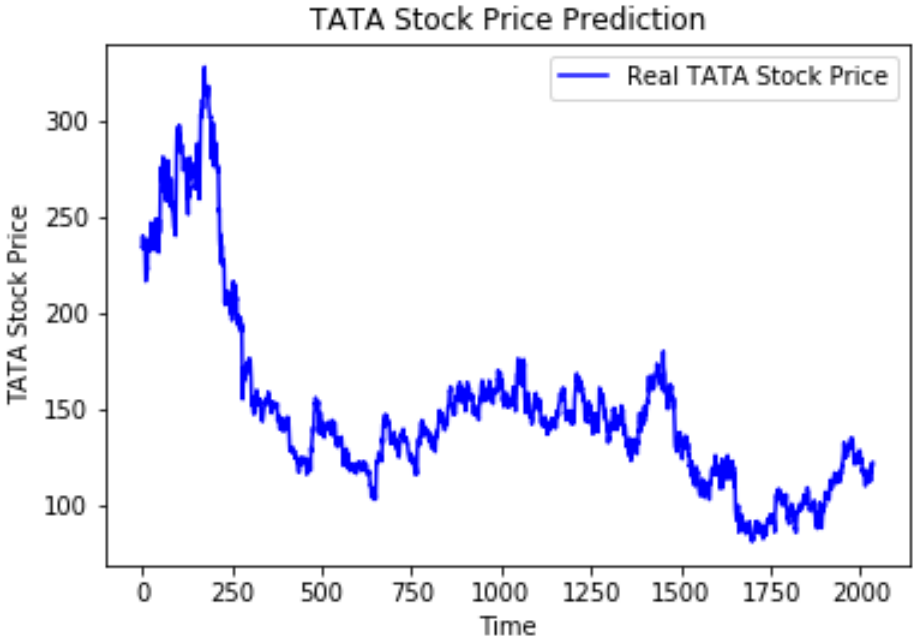


Figure 9: TATA motor company stock

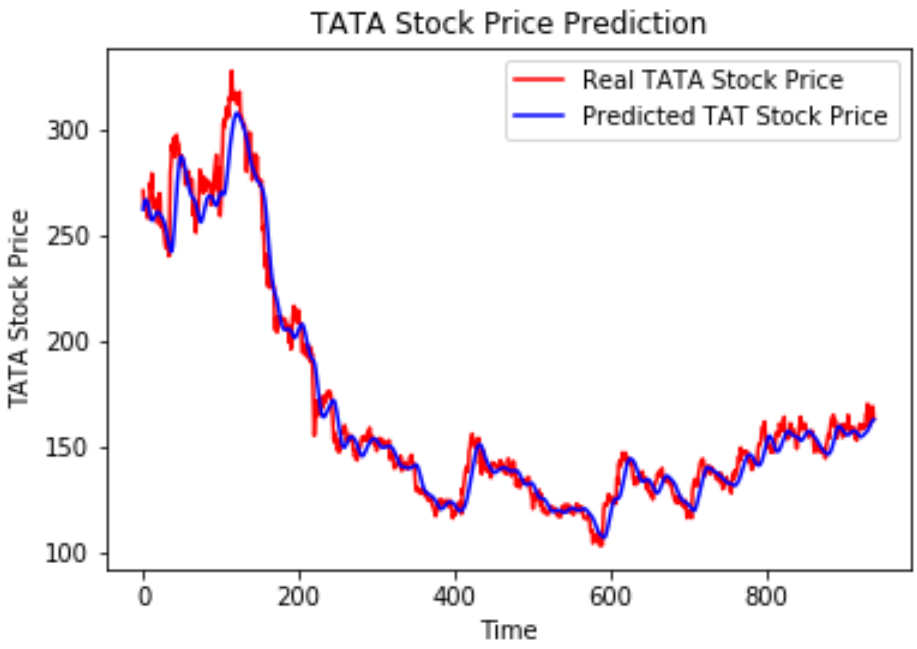


Figure 10: Predicted TATA stock overlapped with predicted TATA stock. The predicted is in blue while the real is in red. This is done with 20% dropout

Furthermore, for this first test, the training data is going to be over the entire data to see how well the NN can match the patterns of the stock. The purpose is to ultimately determine whether mapping NN to crypto-currencies in a real-time manner will be productive and viable. As we can see in figure 10, the NN is able to accurately mimic the overall pattern of the actual stock. Note, this was done using dropout at 20% in between each LSTM layer.

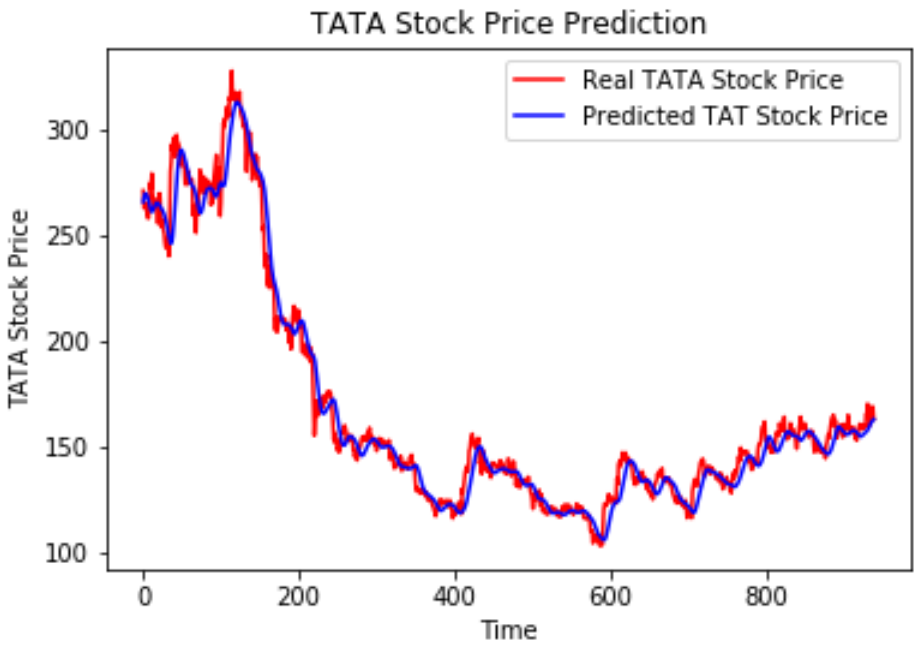


Figure 11: Predicted TATA stock overlapped with predicted TATA stock. The predicted is in blue while the real is in red. This is done with 10% dropout

Next test on this data will be using dropout at 10%. It is immediately clear that there is no real gain. In fact, in some areas, such as at the beginning, there are certain features that are becoming much higher resolution. However, this could also be from overfitting.

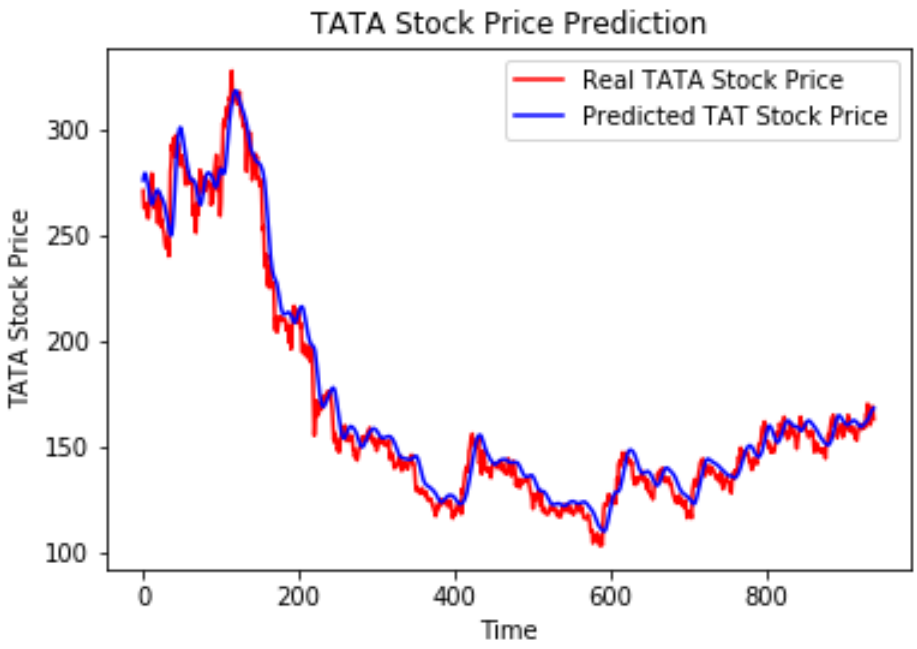


Figure 12: Predicted TATA stock overlapped with predicted TATA stock. The predicted is in blue while the real is in red. This is done with no dropout

Lastly, if we look at the NN without any dropout, it looks nearly identically to that of 10% dropout. However, the NN without dropout consistently over estimated while the 10% dropout was the most accurate. This means that a dropout of 10% will give the most optimal results as 20% underestimates and 0% overestimates. Now that the optimal dropout rate is determined, it is important to determine which optimizer is the best. Our main options are listed below:

1. Adam
2. RMSProp
3. SGDNesterov
4. AdaDelta
5. AdaGrad

Adam is a variation of stochastic gradient descent that is designed to be computationally efficient, memory light, and does well on problems with sparse gradients and non-stationary problems. RMSProp is an adaptive learning method that minimizes the error with root mean square. SGDNesterov is gradient descent but adds momentum to make training much quicker. AdaDelta is adagrad but attempts to optimize the learning rate mechanism to make training more efficient. Lastly, adagrad is a gradient descent algorithm that adapts the learning rate to the parameters of the problem. Below are several graphs demonstrating the loss over 10 epochs of each optimization algorithm. The ideal algorithm is one that is able to significantly reduce initial loss while continuing to learn in minimal time. [10]

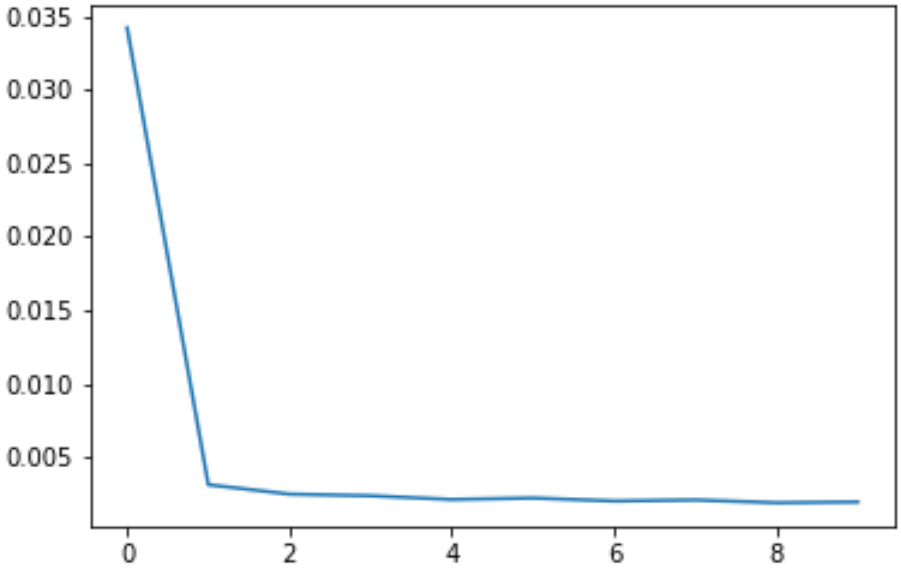


Figure 13: AdaGrad Loss over 10 epochs

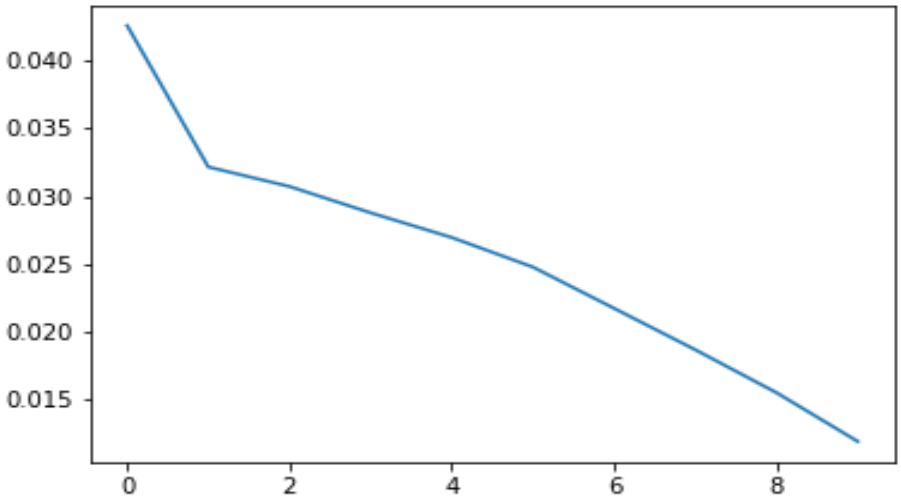


Figure 14: SGDNesterov Loss over 10 epochs

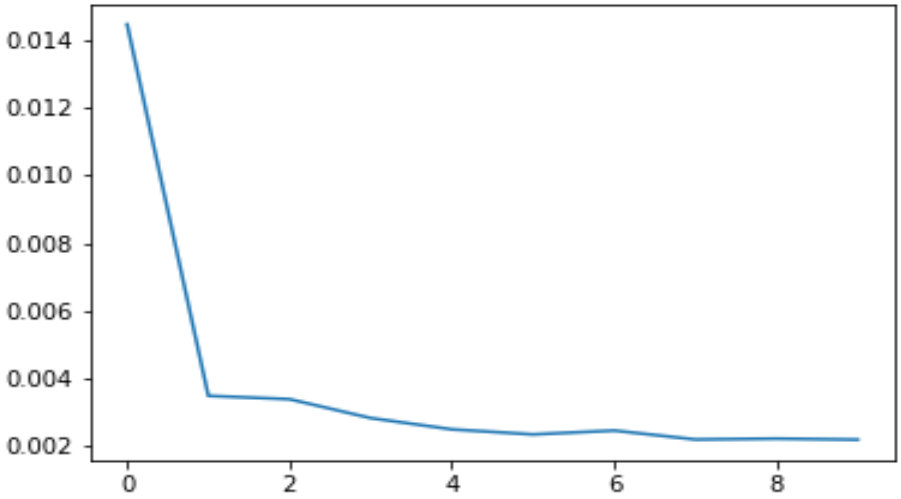


Figure 15: AdaDelta Loss over 10 epochs

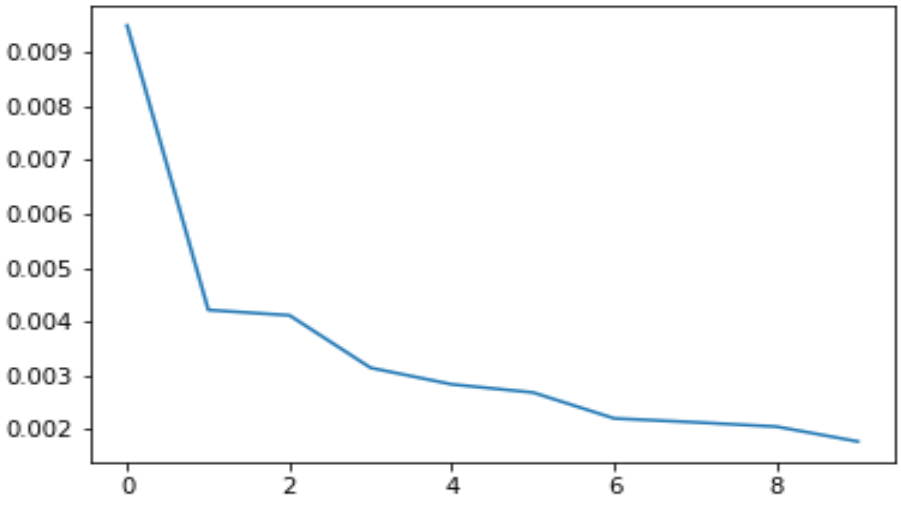


Figure 16: RMSProp Loss over 10 epochs

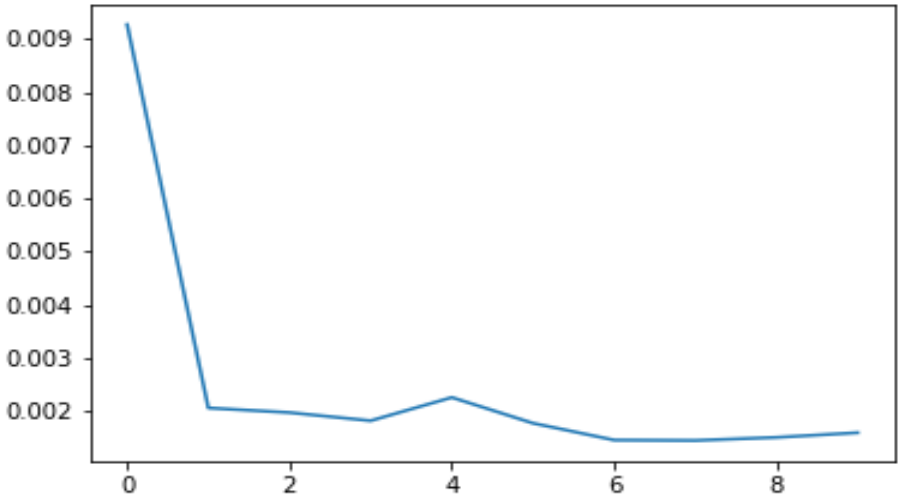


Figure 17: Adam Loss over 10 epochs

5 Conclusion

Neural Networks show tremendous promise in predicting stock markets. Although, as thoroughly explained before, LSTMs are the only NN that are capable of predicting time sequences over large amounts of time, there are many different types of operations that can be done.

When working with stock markets, it is important to get 2 main aspects correct: the model and the optimization. For the model, my data shows that the optimal model is LSTM with dropout layers in between set to dropout 10%. Next, the best optimization model is Adam. Although all the models have fairly low numbers - with some exceptions - adam does so with the least amount of memory and in the quickest time. As a result, it can more quickly output more accurate data which is vital when working with stocks; every second counts.

On github, there is code to a web dashboard that uses the specified findings to attempt to predict stock prices and crypto-currencies in real time. Since stocks depend more on the news and less on patterns, they are more difficult to predict. However, crypto currencies are perfect as they are usually only traded by looking at patterns and make calculated best. In combination with an algorithm to make optimal trading decisions, the NN could potentially be extremely useful in creating a very small income.

References

[1] A. Kurenkov, "A 'brief' history of neural nets and deep learning."

[2] J. Brownlee, "Supervised and unsupervised machine learning algorithms," Sep 2016.

[3] M. Nielsen and M. A., "Neural networks and deep learning," Jan 1970.

[4] V. Valkov, "Creating a neural network from scratch-tensorflow for hackers (part iv)," May 2017.

[5] K. Maladkar, "6 types of artificial neural networks currently being used in ml," Nov 2018.

[6] A. Tch, "The mostly complete chart of neural networks, explained," Aug 2017.

[7] D. Britz, "Recurrent neural networks tutorial, part 1 – introduction to rnns," Jul 2016.

[8] S. Yan, "Understanding lstm and its diagrams – ml review – medium," Mar 2016.

[9] V. Yadav, "Why dropouts prevent overfitting in deep neural networks," 2019.

[10] S. Ruder, "An overview of gradient descent optimization algorithms," 2019.

[11] "Artificial intelligence – what it is and why it matters."