Background:

Based on the concussion listed in Corsi 2009, we attempt to train a neural network model to better forecast market volatility, specifically the VIX index. This is done with the aim to implement this forecast to create a futures trading strategy. With this in mind, we look to forecast the VIX index anywhere from 10-57 days into the future based on future contract expiration dates using a series of independently trained neural networks.

Data:

All the features inputted in this model were regularized to a mean of 0 and a standard deviation of 1. Allowing these values to stay unregularized will result in uneven biases in the weights of the network. For example, if I was to input the S&P500 index, currently sitting at around 5100, the weights for it to predict a VIX value (somewhere in the 20-40 range) would have to be a small decimal. Therefore, if the S&P500 moving 10 points would cause an insignificant change to the vix prediction while the two might be much closer related.

The differing data set consists of multiple permutations of the following features:

Feature Name	Explanation
Bond_10	10-year bond rates at date X
Bond_3	3-month bond rates at date X
Sp500	% change in the S&P500 index
vix	VIX index value at date X (
Mean_inv	Mean inversion or (10-year bond – 3-month bond)
Rus_vol	Volatility index of the Russel2000 index at date X
info	Past h days of vix index values at time X, plus several moving averages (usually a 2-day, 5-day, 1 month, and 3 month)
macro	Initial job claims filed by individuals after separation from employer at date X
arma	ARMA prediction data
Usd_euro	USD to Euro conversion rate at date X
unemployment	Unemployment rate at date X
NFCI	The Chicago Fed's National Financial Conditions Index at date X
VXV	Volatility of Vix index at day X
futures	Fixed horizon futures price data
Market_outlook	Equity market volatility based on Business Investment and Sentiment at date X

Bofa_index	Bofa US high yield index option – adjusted spread at date X
Rus_2000	Russel2000 index at date X
Gen_arma	Fixed horizon ARMA predictions generated from 10 days out to 57 days out
Gen_GARCH	Fixed horizon GARCH predictions generated from 10 days to 57 days, based on Sp500 returns.
RSI	Relative strength index at date X using previous 14 values of VIX

Methodology:

I have experimented extensively with different types of neural networks, I have broken this section down into categories, explaining why I chose each of the features that I did. Finally, I end it off with a section about LTSM's which I am now exploring.

Design:

Generally, when speaking about nodes and hidden layers, the more hidden layers, the more interactions between nodes and the better the model can memorize the data. This of course, comes at a cost of computational efficiency. I started this project with one layer of 128 nodes and quickly increased it to three only to find that the model was taking a significantly longer time to train for less than a 1 MAPE difference in out of sample prediction accuracy (this was while I was still only predicting one day ahead out of sample.) When considering a longer horizon prediction, I decreased the number of layers to two and instead increased the number of nodes to 256 to not memorize the data as much since it caused the out of sample predictions to be extremely inaccurate or just slightly better than the three layer model. The increase in nodes helped the feature set interact with itself more and while giving it the ability to learn complex patterns at a faction of the computing time that an extra layer added. In general, the benefit of an extra layer lies in the activation function which allows model to be nonlinear, but I found the marginal benefit is outweighed by the cost of the extra layer.

Parameters:

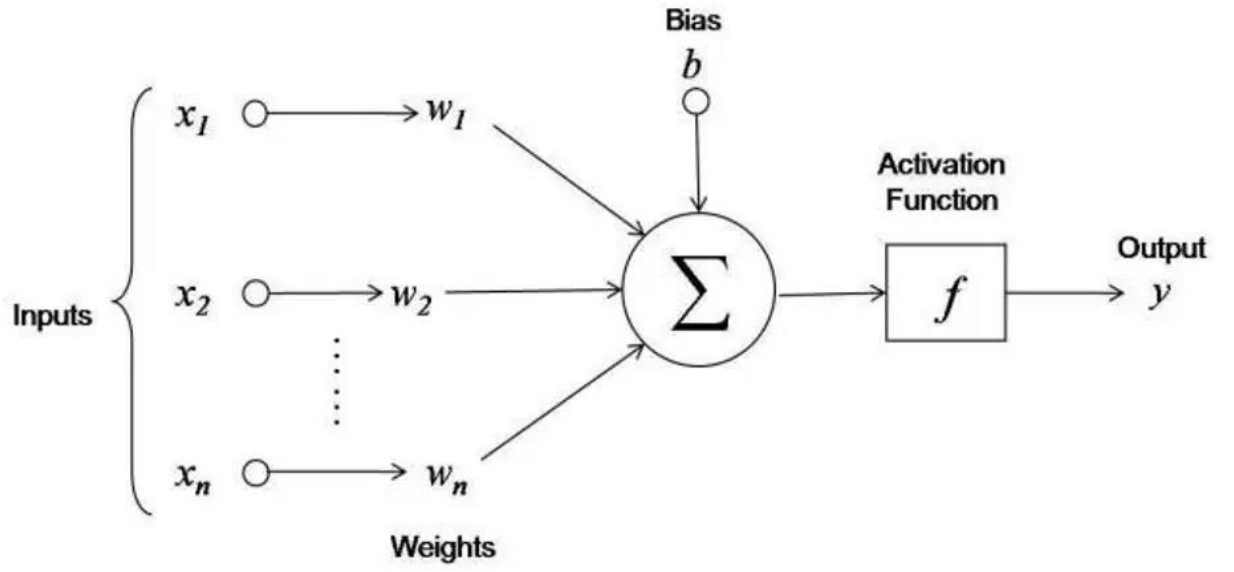
The different adjustable parameters are the following: Learning rate, learning rate decay (decay for short), L1 and L2 regularization loss, momentum, and alpha (spoken about in the activation function section). The learning rate is the factor applied to the gradient during back propagation when adjusting the weights of the model. This decays over time to find the lowest loss for the given data. In my model I went with a learning rate of 0.03 and a decay of 0.005. The decay function is $new\ learning\ rate = old\ learning\ rate \cdot \frac{1}{1 + number\ of\ epochs * decay}$. This allows for the model to make large adjustments to weights at first then as the learning rate gets smaller toward epoch number 1000, it can hone in weights that fit the data the best.

L1 and L2 regularization loss are ways to add loss to the loss functions based on the values of the weights in the model. If the model weights get too large or too small, the model might be memorizing one of its features which is counterproductive. The L1 regularization factor is multiplied to the sum of the weights of all hidden layers in the network then added to the loss function. The L2 regularization factor is similar but multiplied by the square of the weights to account for extremes more so than the L1 loss. This is applied to both the weights and biases of the model. In my model I have found that a L2 regularization loss of 0.0001 and L1 regularization loss of 0.0003 have combined for the best results by not increase the loss by a great enough factor for the model to change its early step approach but rather guides it in the right direction in the later stages of training when the loss function is decreasing at a slowing rate.

Lastly, I used the Adaptive Momentum (Adam) optimizer because it is gaining popularity for its ability to handle deep learning as well as its ability to get over inflation points with its momentum to get to the global minimum loss. If you'd like me to describe this optimizer further, I'd refer you to the NNFS book as they explained the calculus behind it as well as it can be summarized.

Activation Functions:

For this project I settled at using Exponential Linear Unit Activation function (ELU) , and Leaky Rectified Linear Unit (LRelu). Since a neural network is a series of linear combinations, to make the model fit non-linear data we use these non-linear activation functions as seen below:



In this project, I started with Relu, which is a piecewise function defined as:

$$f(x) = \begin{cases} x \; if \; x \geq 0 \\ 0 \; if \; x < 0 \end{cases}$$

The benefit of this function is that its derivative is computationally easy to calculate during back propagation which leads to faster runtimes. However, it has many drawbacks, specifically that due to the nature of the function we get a lot of "dead" neurons. This refers to when a large enough gradient gets passed through the system and the neuron outputs 0. We refer to this as a "dead" neuron because it will most likely not recover to output any other value.

To fix this problem, there are two solutions, the first being a Leaky Relu (LRelu) activation function, defined as the following:

$$f(x) = \begin{cases} x \; if \; x \geq 0 \\ cx \; if \; x < 0 \end{cases}$$

Where c is a constant between 0 and 1. This again, has the benefit of an easily computable gradient while still allowing for non-linearity and usage of negative neuron values. This means that we see a lot less "dead" neurons and therefore use more of the neurons in the model. However, the drawback to this activation function is that you must use a low number for the c value such as 0.01 because the closer c gets to one, the less non-linear the function gets and the harder it is for the network to correctly model non-linear data. With such a low c value, we still discount our negative values by a large amount which means they disproportionally impact our predictions.

This directly leads us to the second solution to this problem, an Exponential Linear Unit activation function or ELU, defined as the following:

$$f(x) = \begin{cases} x \; if \; x \geq 0 \\ \alpha(e^x - 1) \; if \; x < 0 \end{cases}$$

This function allows us to have the non-linearity we are looking for while not discounting negative values as much as the LRelu activation function. The drawback of this activation function is the computational intensity of calculating the gradient since getting an accurate estimate for exp(x) takes much longer than the linear options mentioned before. Because of the large amount of computing power I have access to, I use ELU activation functions as I've found they provide the best representation of all the data which then translates to out of sample prediction accuracy.

Loss Function:

When picking a loss function for training, the two options were between an MAE loss function and MSE loss function. I've found that both work well for our purposes but due to the volatile nature of the data we are examining, using MSE as our loss function would create for the most accurate representation of the data. Now while this may seem like the obvious answer, the question of which loss function to use is a big more complicated than that. Because we determine the accuracy of our model during back testing using the absolute value of the difference between the actual result on the day of expiration and our prediction, training the model with an MAE loss function could provide a better out of sample prediction at the cost of training accuracy. Furthermore, due to the

high volatility, the model tends to predict "jumps" out of sample when they may not happen, resulting in either a large error, or an extremely accurate prediction. This unfortunately can't be avoided but the size of these "jumps" can be decreased making the predictions more consistent if we train the model on MAE. In summary, both MAE and MSE have their benefits for both training and out of sample prediction which is why I alternate between both based on the out of sample prediction.

After doing more research I came across a new loss function called the Huber Loss function:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \le \delta, \\ \delta \cdot \left(|a| - \frac{1}{2}\delta\right), & \text{otherwise.} \end{cases}$$

This function allows for a combination of MSE and MAE. The standard Huber Loss function is when lambda is equal to 1, which allows for the loss function to be MAE for values less than 1 and MSE for values more than 1. This allows for the model to learn the high volatility of the models well while also maintaining a better out of sample prediction. Using this loss function with the LSTMs I was able to get the lowest out of sample validation average error of 3.52.

LSTM:

The Long Short-Term Model looks at both long- and short-term memory. In general, recurrent neural networks have a large gradient decent problem where if we pass in 30 points of data and initialize the weights at 0.1, the first input (say for example 10) would be multiplied by 0.1^{30} essentially making it ineffective. The reversal of this is also true, if the weights are initialized above 1 where the gradient becomes infinite relatively quickly. To resolve this issue, we use a Long Short-Term Model which has a separate recursive part of each node for long term predictions and short-term predictions. The LSTM doesn't allow for weights to directly affect the long-term predictions, effectively protecting them from the vanishing gradient while allowing short term predictions to use the gradient to influence their weights.

Neural Network Results:

The way I have been measuring the results is by comparing the predicted model value of the VIX at contract expiry to the AMRA predicted value. By assigning a value of 1 to dates where the model predicted better than the ARMA and a 0 to dates where the model lost out to the ARMA, we can get a sense for how well the model is doing in comparison to the ARMA predictions. Furthermore, we can also average these differences between the actual value and the model predicted value to get an average error of both models. The average error of the ARMA model is 3.5961 when comparing it over the past 1000 dates (from Feb 26th, 2020, until Jan 30th, 2024.) The average error of the ARMA model when comparing it over the past 500 days is 3.2594 (Feb 17th, 2022, until Jan 30th, 2024.)

Running the neural network over described above with the following feature set: 2 day moving average, 5 day moving average, 1 day ago VIX, 2 days ago VIX, and Russel Volatility Index, we get an average error of 4.067, with the model being better than the ARMA a little under 42 percent of the time.

LSTM Results:

After training the LSTM, the results were far superior to the Neural Network. The LSTM was able to achieve an average error of 3.526 over the past 500 days' worth of data compared to the ARMA's 3.259. Comparing the square average error, we see the model had an average squared error of 17.6135649 while the ARMA had an error of 16.4825057 over the past 500 days (Graphs attached to the bottom of this document). Shifting our focus on the past 100 days, we see that our model beat the ARMA prediction in both average error and average squared error [1.98391567 as compared to 2.22999692 (MAE), and 5.9340639 as compared to 8.58847535 (MSE)]. Furthermore, once as you get closer to today's environment, the prediction is more consistently accurate. I predict this is due to the larger training set in combination with less volatility aftershock from COVID resulting in inflation and sudden changes in interest rates.

Areas for Further Exploration:

I believe we could improve the out of sample predictions of this model by increasing the layers and nodes within the LSTM. Since this is getting into the territory of deep learning, I did not have enough time to finish optimizing the parameters, but I was able to get an average error as low as 3.47 with such models. The reason I did not include them in the results section is due to the complexity of the models, some of them unfortunately succumb to the vanishing and exploding gradient problem and did not finish training. From the approximate 500 models I ran about 12 "died" due to this problem while the other 488 resulted in the said 3.47 average error.

Leaning into the LSTM model, with more time I think we can rethink the way we are training the model. Currently, we are submitting a sequence of 30, 60, 90, or 100 days for the LSTM to make a prediction with. Given more time, I believe I could reprogram the model to focus on a certain contract and predict that contract's value at expiry by being handed in a different sized sequence based on how many days to expiry.

Furthermore, given more time I could code in a grid search algorithm that would constantly run and search for the optimized parameters for each model before running the model, since I am currently constricted to running each model with the same parameters. Building on this, I could also write in a helper function that would re-run any model that doesn't finish training until all models have been trained.

Lastly, I didn't get a chance to include the SVR model into my LSTM, I think with more time I would be able to include and perhaps increase accuracy.

Conclusion:

The most important thing I've learned since embarking on this project is that everything you need to forecast the VIX is a very simple feature set with a complicated model that can find insights that as humans we might not be able to harness quite yet.