IMPERIAL COLLEGE LONDON

Applying machine learning to the forecasting of stock price volatility

Michael Paddon

Supervisor: Prof. David Targett

A Project submitted in partial fulfilment of the
requirements for the MBA degree

May 2016
[incorporating errata as of April 2017]

[16,191 words]

Synopsis

Volatility is an intrinsic property of risk bearing assets and accurate forecasting of this variable is fundamental to effectively managing risk. There exists a rich literature in this domain and many forecasting models are extant. Machine learning techniques have been applied successfully to the problem. In particular, neural nets have been constructed and shown to outperform historical, implied and autoregressive conditionally heteroscedastic volatility models. These neural nets solutions, however, have proven complex in practice to configure optimally. The process has been characterised as “more of an art than science” (Hamid and Iqbal, 2004).

The research hypothesis for this project is that recent advances in machine learning and deep neural nets allow the construction of a high performance volatility forecasting model that is easier to configure correctly and more stable against changes in model hyperparameters or inputs. An LSTM based recurrent neural network architecture is proposed as suitable for time series forecasting. Data from the CRSP US Stock Database and Google Trends is used to construct training patterns and testing patterns and the model is subjected to fifteen different training scenarios. These scenarios vary the model hyperparameters and inputs, and compare the consequent performance against three common benchmark forecasting models.

The model is found to perform well in most scenarios, be easy to configure and demonstrates good resilience to hyperparameter and input changes. In the highest performing configurations, the model demonstrated an RMS error rate less than 50% of the next best performing benchmark.

Acknowledgements

I gratefully acknowledge the supervision and guidance provided by Professor David Targett during the course of this research project.

I sincerely extend my thanks to Kento Tarui for his generosity in agreeing to be interviewed as a machine learning subject matter expert.

I am also deeply grateful to David Barrett for sharing his extensive knowledge and guidance on machine learning techniques.

I would also like to thank Franklin Antonio for his wide ranging discussions on the structure of markets and the nature of volatility.

Last, but not least, I would like to thank Yuriko Paddon for asking the simple questions that turned out to have complicated answers.

Table of Contents

Figures

Tables

"Take calculated risks. That is quite different from being rash."
-- George S. Patton, 1944

1 Introduction

Volatility is an intrinsic property of risk bearing assets. Onwukwe et al (2011) observe that it “permeates finance and it is a key variable used in many financial applications such as investment, portfolio construction, option pricing and hedging as well as market risk management”. It follows that good volatility forecasts are fundamental to effectively managing risk.

Informally, volatility may be thought of as a measure of the unpredictability of asset prices. It is often defined as a statistical measure of the dispersion of returns, which in the case of a stock encompasses both changes in the price of shares and distributions such as dividends. Consider a security for which future returns are entirely stable and certain. The dispersion or “surprise” in these returns is zero and a typical investor would probably agree that this is a low risk asset. Conversely, imagine a security with completely unpredictable and wildly swinging returns. This higher level of dispersion would lead many investors to regard the latter as more risky than the former example. The more risky asset may yield a greater return, but it also may yield less or even nothing.

It is possible to imagine a security with completely predictable and certain future returns, but where the returns are not the same from period to period. For instance a “risk free” bond could be constructed with time varying coupons. The returns on this asset exhibit a clear statistical dispersion, but there is no unpredictability or surprise present and no risk to manage. Volatility can therefore be a bit of a slippery concept.

Markowitz (1952) first introduced the idea of using the mathematical variance of returns as a proxy for risk. He explicitly rejected the hypothesis that investors acted simply to “maximize discounted return” because this did not explain the obvious need for diversification to reduce risk. His “expected returns - variance of returns” model incorporated diversification by showing that variance (or, equivalently, the standard deviation) of returns was related to portfolio risk. Since the publication of Markowitz’s paper, the use of the

standard deviation of returns as a proxy for risk has become a standard practice throughout the financial world.

Volatility may therefore be mathematically defined as the standard deviation of actual returns versus mean returns. In practice, the sample standard deviation is used when looking at a subset of returns:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(r_i - \bar{r})^2}$$

where r_i is the actual return and $\bar{r}$ is the mean return of n samples. Using this formula, it is straightforward to calculate the historical volatility for a given stock or portfolio of stocks.

Volatility is of great importance to market participants as quantification of risk. For instance, the Sharpe Ratio is a measure of risk adjusted return that uses volatility as its denominator (Cuthbertson and Nitcsche, 2008) and is widely used by investors to measure asset or portfolio performance. VaR is a more sophisticated way of measuring risk, often actively used by banks and other institutions to manage their exposure. A common way of calculating VaR is by the variance-covariance method to which volatility is a key input (ibid.).

Volatility appears not only in the measurement of risk, but in the act hedging it away. The Black-Scholes-Merton model, which is widely used to price European options, uses an estimate of future volatility as a key input parameter (Cuthbertson and Nitzsche, 2001). For investors who wish to hedge a position, the price of options fundamentally drives the cost of their strategy and therefore their ultimate returns. Obviously the ability to price options accurately is highly desirable. Indeed, the better a market participant can estimate the correct price of an option the more information asymmetry exists in their favour. This can create arbitrage opportunities as well as optimise hedging.

In the case of the Black-Scholes-Merton model, future volatility is unique in being the only parameter that is not directly observable. All the other inputs to the model are historical in nature and hence may be directly calculated. Using the model to price options effectively therefore requires an accurate

forecast of future volatility. There exists a rich literature in the field of volatility forecasting. Poon and Granger (2003) observe that volatility forecasting “has held the attention of academics and practitioners over the last two decades”. They cite 93 papers in the field of volatility forecasting models covering a wide range of approaches to the problem. With such a surfeit of choice, which model should an investor use? Poon and Granger (ibid.) identified three general classes of models that were best across 66 studies: historical volatility, implied volatility and GARCH. These models will be further defined later in the report.

This research project addresses the question of whether there might exist a more accurate method of forecasting volatility that those in common use today. More specifically the project restricts its scope to stock prices since historical data, including pricing, for these assets are transparent, readily available and voluminous.

Before moving on to the research hypothesis, however, a final general observation about volatility is, perhaps, in order. Volatility is a useful proxy for risk and it is commonly treated as such in the financial literature and mathematical models. However there is a countervailing viewpoint that volatility is not exactly the same thing as risk. A stock that rises exhibits high volatility, but if that rise is based on business fundamentals then the underlying risk may not have moved. Buffett (2014) said “Volatility is far from synonymous with risk… If the investor… erroneously [views] it as a measure of risk, he may, ironically, end up doing some very risky things”. Buffett is widely recognised to be an extraordinarily successful investor so his view is compelling to the empiricist. Kirchner (2010) also argues that “volatility is not risk, but uncertainty” from the view of a probabilistic framework for pricing derivatives. So is volatility a good proxy for risk? For the purposes of this report it will be treated as such but, as always, the practitioner should take case to not confuse the map with the territory.

2 Hypothesis

The research hypothesis for this project is that recent advances in machine learning, specifically deep learning architectures, may be successfully applied to forecasting stock price volatility with less supervision and manual tuning than required by existing methods.

There exists a rich literature on the application of machine learning to stock price forecasting. Atsalakis and Valavanis (2009) surveyed over 100 papers and concluded that “neural nets… are suitable for stockmarket forecasting” and “outperform conventional models in most cases.” However despite this success they note that “difficulties arise when defining the structure of the model (the hidden layers the neurons etc.). For the time being, the structure of the model is a matter of trial and error procedures.”

A classical neural net is a directed graph of computational elements, known as neurons. There are many ways to structure such a graph. Often they are organised into layers of neurons, with an input layer, an output layer and several hidden layers sandwiched in between them. The number layers, the number of neurons in each layer and how they all interconnect is a fundamental design decision. The intrinsic properties of the neurons themselves, including the important “activation function”, is a design choice. The correct selection of input values, or features, is important and there are many learning different algorithms each with their own parameter choices (Hecht-Nielsen, 1990). This large design space explains the need for extensive trial and error. It also raises the concern that a highly “tweaked” neural network might only perform well for a narrow set of conditions. In other words, the design process might “overfit” it. This is a particular issue in financial forecasting because the behaviour of stocks and the entire market is continuously evolving and change is sometimes both rapid and extreme.

In regards forecasting volatility specifically, Hamid and Iqbal (2004) similarly found that neural nets could “outperform implied volatility forecasts and are not found to be significantly different from realized volatility”. However they also note “the potentials of neural networks in forecasting the market involves more of an art than science” due to the many combinations of possible

models and inputs. This is encouraging because prior art suggests that neural nets are indeed suitable for predicting volatility, however the same issue of it being an art rather than a science is present.

Ideally a neural net could be designed to forecast volatility which (a) would require only relatively coarse parameters to be chosen, (b) would be able to consume input features without need for careful curation and (c) would be demonstrably stable over a broad range of market conditions.

In order to explore further how this might be done, a semi-structured interview was conducted on December 14th 2015 with a domain expert. The interviewee, Dr Kento Tarui, has a background in classical machine learning models and was recently supervising a machine learning research project in the private sector. The interview was structured around the design decisions that one might have to make to build a volatility predictor and a significant number of topics were touched on. The key points made by Dr Tarui during the interview were as follows (direct quotes indicated by quotation marks):

- Start with defining desired output and available inputs.
- A time series predictor should use around 4 layers. The two middle layers will extract features and enhance signals respectively.
- The middle layer should be as large as the number of features you are trying to extract.
- Convolution deep networks used for vision applications are not fully connected, but rather treat the input in topological groupings. In the case of stock prediction it is difficult to say if this approach is correct.
- A recurrent neural network may be suitable for predicting time series data.
- A network which outputs volatility predictions for many stocks can work just as well as a network that just predicts one stock.
- Classical neural networks relied heavily on pre-processing of data. However "people… use deep neural nets because they just want to use raw data". The current thinking is to stay as close to the raw data as possible, including noise.

- A classical approach is to use principle component analysis to determine what the inputs should be. However, the “deep neural network philosophy is different”. Basically, the idea is to let the network sort the data out.

This interview yielded an important insight. *Deep* neural networks are qualitatively different from their classical predecessors and they require less input curation than a classical network.

Karpathy (2015) states that recurrent neural networks, a class of deep network, have “unreasonable effectiveness” when dealing with sequences (and provides a number of examples). This echoes Dr Tarui’s observation that a recurrent network may be applicable to predicting time series data.

Because recurrent neural networks can deal with sequences of data, the size of the resulting computation graph can be much. Consider a sequence of 100 stock prices. A single recurrent neural net element can process this sequence. A classical neural net, on the other hand, would require 100 elements (one for each step in the sequence). This property promises to make large scale volatility forecasting more tractable than in the past.

Another development that makes construction of large scale neural nets more tractable is the emergence of the highly parallel commodity GPU hardware. Originally designed for graphics acceleration, GPUs are rapidly becoming a must-have resource in the machine learning world. The reason is that they support hundreds or thousands of processing elements which may be programmed to operate concurrently. This particularly suits them to neural net applications, as parallel execution can reduce training time by many factors. This is especially attractive when experimenting, as many more tests can be run in the same unit time.

2.1 Research Goal

The research goal can now be stated with precision. The objective is to construct a deep learning volatility forecasting model based on a recurrent neural network architecture. The model will be designed to map onto and be executed by commodity GPU hardware. The model will then be trained with

historical stock data from several sources. The input to the model will be kept as raw as possible. Finally, the performance of the model will be measured versus common benchmark methods of forecasting volatility. The goal is build a volatility forecasting model that outperforms the benchmarks over a broad range of scenarios.

3 Theory

This section reviews key theory that will be drawn on to build the forecasting model

3.1 Stochastic Volatility

Heston (1993) provides the following two equations which model the spot asset price as a random walk and the underlying volatility as a stochastic process (equations quoted directly from Heston):

$$dS(t) = \mu S dt + \sqrt{v(t)} S dz_1(t)$$

$$d\sqrt{v(t)} = -\beta\sqrt{v(t)}dt + \delta dz_2(t)$$

Here $S(t)$ is the asset price, $v(t)$ is the variance $z_1(t)$ is a Wiener process to which $z_2(t)$ is correlated.

This model tells us that volatility is continually evolving. Future values can only be predicted if $z_2(t)$ is known. However $z_2(t)$ is a randomly distributed variable and is, by definition, unknowable. Therefore, precise prediction of volatility is not possible, and forecasting can only be an exercise in estimation.

Another interpretation of the model is that there is a true, but unobservable, underlying volatility obscured by random noise. Because it can never be observed, the volatility can only be estimated within error limits.

3.2 Volatility Forecasting

As discussed in the introduction there are three volatility models of particular interest that are useful for forecasting: historical volatility, implied volatility and GARCH. Since any new forecasting model will require benchmarking, it is useful to review current practice.

Brooks (2008) points out that “the simplest model for volatility is the historical estimate.” This approach involves calculating the sample standard deviation across a given period (using the formula provided in the introduction) and then using the result as the future estimate. Simple unweighted historical volatility assumes the future will be much like the past. Heston’s model suggests that this is not necessarily the case, however over a short time frame it may be a reasonable assumption. Brooks (2008) notes that simple historical volatility “was traditionally used as the volatility input to options pricing models” and “is still useful as a benchmark for comparing the forecasting ability of more complex time models.”

A more sophisticated form of historical volatility model is the exponential weighted moving average (EWMA). Brooks (2008) provides a formal description (equation quoted directly from Brooks):

$$\sigma_t^2 = (1-\lambda)\sum_{j=0}^{\infty}\lambda^j(r_{t-j}-\bar{r})^2$$

where λ is a decay factor. Brooks also notes that a value of 0.94 is often chosen for the decay factor.

An ARCH (Autoregressive Conditionally Heteroscedastic) volatility model attempts to capture the phenomenon that “volatility occurs in bursts” and is therefore demonstrates autocorrelation (Brooks 2008). Brooks provides an equation showing how, in ARCH, changes in variance are driven by the square of the previous error (equation quoted directly from Brooks):

$$\sigma_t^2 = \alpha_0 + \alpha_1 u_{t-1}^2$$

More specifically, this is denoted an ARCH(1) model because only one previous error term is used.

A GARCH model (Generalised ARCH) “allows the conditional variance to be dependent upon previous own lags” (Brooks, 2008). To achieve this, another term involving the previous variance is added to the ARCH formula, giving (equation quoted directly from Brooks):

$$\sigma_t^2 = \alpha_0 + \alpha_1 u_{t-1}^2 + \beta\sigma_{t-1}^2$$

More specifically, this is denoted as a GARCH(1,1) model as it uses one previous error term and one previous variance.

Cuthbertson and Nitzsche (2008) give the Black-Scholes formula for a European call option as:

$$C = SN(d_1) - N(d_2)Ke^{-rt}$$

$$d_1 = \frac{\ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$d_2 = d_1 - \sigma\sqrt{T}$$

Future volatility is an input parameter to this equation. Given the observation of option prices in the market, the model can be run in reverse to determine the implied volatility of the underlying asset. A similar calculation may be performed with put options. Implied volatility is therefore determined by the market as part of the price discovery process.

3.3 Artificial Neural Networks

An Artificial Neural Network (ANN) is a computational system loosely modelled on biological processes. An attractive property of these networks is that is they can learn to approximate functions across a large number of input parameters. This is particularly useful when the function is unknown or too complex to solve by other means.

ANNs are constructed from fundamental computation units called *neurons,* which implement the following function (based on Hecht-Nielsen, 1990):

$$y = A(\sum w_i\, x_i)$$

where w_i are weights, x_i is an input and A is the activation function. Each neuron may have many inputs, but it only has one output. The activation function may be any function defined over the range of $\sum w_i\, x_i$. Early neural nets used a threshold function, but modern practice is generally to use a sigmoid.

Neurons are arranged into a directed to form a network. Often they are organised into layers. Layers may be fully connected, in which all neurons in

one layer feed all neurons in the next, or they may be partially connected. A network with more than 2 layers is known as a “deep network”.

ANNs learn by adjusting the weights w_i. A common method of updating weights is the backpropagation algorithm (Hecht-Nielsen, 1990). This uses an optimization algorithm to estimate error and consequently adjust the weights to reduce that error. The error is estimated by a loss (sometimes known as cost or objective) function.

3.4 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is an ANN in which the neurons have “loops”, whereby state may be retained by a neuron between computational steps (Olah, 2015). This persisting state is a form of memory and it allows the neuron to respond to new input based on what it has seen before.

Another interpretation of an RNN neuron is that it is equivalent to a “chain” of traditional neurons which pass state down the line as they fire in sequence (ibid.).

Olah (2015) observes that even though RNNs have the ability to incorporate information from previous time steps, in practice this has limits as the time gap grows larger.

3.5 Long Short-Term Memory

A Long Short-Term Memory (LSTM) network is an RNN “capable of learning long-term dependencies” (Olah, 2015). The key to this ability is that a LSTM neuron can “forget” information. More formally, a set of gates are added to the neuron to modulate how much information is permitted to be added the looped back state. There exist many possible variations on how many gates to add to each neuron and how that can be wired into the inputs, output and looped back state, however they all perform about the same (ibid.).

Olah (2015) notes that LSTMs generally perform better than plain RNNs in many domains, including “speech recognition, language modeling, translation, [and] image captioning.”

3.6 Tensors

When constructing machine learning systems, it is common to deal with highly dimensional data. For instance, a volatility forecasting model's input might consist of multiple features, for each stock in the portfolio, for each trading day, for each member of a batch. It is convenient to have a notation and conventions to describe such objects.

For machine learning purposes, a tensor is an n-dimensional set of values. The *rank* of a tensor is its number of dimensions. A scalar is a tensor of rank 1. Both a 2x2 and 3x3 matrix may be represented as tensors of rank 2.

The shape of a tensor specifies the size (of fixed) of each dimension. Conventionally, the notation "[a, b, c, …]" is used to specify the shape. In that example the first three dimensions of the tensor are of size a, b and c respectively. The shape of a scalar is [1]. A 3x3 matrix may be represented by a tensor of shape [3, 3].

4 Data Sources

The research project draws upon several sources of historical data from which to make forecasts.

The first, and primary source, is the CRSP US Stock Database (CRSP, 2015). The CRSP database, as downloaded, provides US listed stock data from January 1st 1926 to September 30th 2015.It is update quarterly. This data set contains a rich set of information about corporate events and well as a core time series of price related information. Each issue in the CRSP data set is assigned a unique and permanent identifier which makes tracking the stock easier over a long time period which might include complex corporate actions.

Data that was potentially relevant to the project was extracted from the CRSP data set (which was supplied as a CSV file) and stored in a bespoke SQL database suited for access by a forecasting model. The data extracted included:

- Permanent identifier
- Name, symbol, and other identification changes

- The core price time series, including high ask price, low bid price, total return, closing price and volume
- The auxiliary time series data, including return without dividends, opening price, closing ask price, closing bid price and NASDAQ number of trades.

The Google Trends service (Google, 2015) provided data on the relative frequency of selected stock symbols being used as search terms in the Google search engine. This data covered the period April 1st 2004 through September 30th 2015, inclusive. This data was also incorporated into the project's bespoke SQL database.

The constituent stocks of major indices such as the S&P100, the S&P500 and the NASDAQ100 as of September 30th 2015 were downloaded from Siblis Research (2016). This data was used to configure the forecasting model in test scenarios.

5 Model Design

The model's purpose is to predict the volatility of each stock in a portfolio, using historical observations of that portfolio. This may be formalized as

$$v = f(\theta)$$

where θ is the portfolio history, v is the volatility forecast and f is a computable but unknown function. The problem is therefore to approximate function f.

The approximation of function f is a hard problem. There are several widely used techniques including the unweighted historical volatility, the exponentially weight moving historical average volatility, the ARCH variance model and the use of Black-Scholes to determine implied volatility from option prices.

The goal of this research project is to define a machine learning forecasting model which can be trained to approximate function f at least as well, and hopefully better, than several of the extant techniques.

This section describes the basic architecture of the forecasting model, how inputs and outputs are managed and how training works. The model's hyperparameters are identified and defined. Key design decisions are reviewed and the rationale for salient choices made along the way is discussed.

Finally, an overview of salient implementation details is provided.

5.1 Basic Architecture

An architecture based on an LSTM recurrent neural network is capable of processing, learning from and forecasting time series data. Recall that LSTM units contain memory and therefore can be trained with a sequence of inputs to predict an output. The core building block of the model will therefore be a basic LSTM unit as described by Zaremba et al (2014).

These LSTM units can be layered in series, with the output of one feeding the input of the next to form a deep network. There is a practical limit to how deep a network can become, because there exist constraints on available computational resources. The number of layers in the network is a model hyperparameter.

Because of this layering, it will be convenient for each LSTM unit to possess an equal number of inputs and outputs (otherwise intermediate logic would be required to hook them together). These inputs and outputs will be referred to as "hidden" since they are strictly internal to the model. The number of hidden inputs per LSTM unit (and consequently the number of hidden outputs) is a model hyperparameter.

The model produces a volatility forecast output for each stock in the portfolio. Because the size of the portfolio may change, the number of outputs is a hyperparameter.

The model accepts a number of features contributed by each stock in the portfolio. If there are *n* stocks each with *f* features, then the total number of inputs per step is $n \times f$. Because *n* or *f* may change from time to time, the number of inputs is a model hyperparameter.

The model is a recurrent neural network and therefore requires a time series of inputs in order to generate its forecast. The number of steps in the time series represents how much history is being used by this process. Different scenarios may require different amounts of history, and therefore the number of steps is a model hyperparameter.

The model learns by applying a variant of the stochastic gradient descent process. Abadi et al (2015) suggest that by batching inputs together, stochastic gradient descent learning can be sped up because it allows "replicas of the model to each compute the gradient… then combine the gradients and apply updates to the parameters synchronously". The rationale is that by running many copies of the neural network in parallel (which represents the bulk of the computational load), the gradient descent step at the end can work with more data and hence learn faster. Since the model is intended to be deployed on parallel GPU hardware, this is an important design optimisation. Therefore batch size is a model hyperparameter which may be adjusted optimise execution time versus the available hardware constraints.

Cotter et al (2011) suggest the same optimisation on the basis that if the stochastic gradient descent algorithm is able to work with a larger subset of data points, it is likely to get a more accurate estimate of the true (unknown) gradient and hence converge faster on its objective. Therefore adjusting batch size also will have an effect on the learning rate independent of opportunities for parallel execution.

The adoption of batching adds one more dimension to both the inputs and the outputs of the model. To review, the final input tensor is of rank 4 with shape [*b, s, n, f*] and the final output tensor is of rank 2 with shape [*b, n*] where *b* is the number of batches, *s* is the number steps in the time series, *n* is the number of stocks and *f* is the numbers of features per stock. Note that the number of stocks and the number of outputs are the same because the model is forecasting one volatility value for each stock's time series.

In summary, the model has six identified hyperparameters:

- The number of layers
- The number of hidden nodes
- The number of outputs
- The number of inputs
- The number of input steps
- The batch size

These are essentially the dials that can be turned in order to provide different learning and performance.

5.2 Input/Output Transformation

The astute reader will have noticed that the number of hidden layers is constrained to match neither the number of inputs nor outputs. Therefore there must be a mechanism to couple the model's inputs to the hidden inputs of the first LSTM unit, and similarly to couple the hidden outputs of the last LSTM unit to the model's outputs. This may be accomplished with linear activation layers.

A linear transformation layer from *p* inputs to *q* outputs is simply a matrix operation as follows:

$$\boldsymbol{Q} = \boldsymbol{P} \times \boldsymbol{W} + \boldsymbol{B}$$

where **P** is the vector of inputs, **W** is matrix of weights of shape [*p*, *q*], **B** is a vector of biases of shape [*q*] and **Q** is the resulting vector of outputs.

How are **W** and **B** chosen? This is the beauty of stochastic gradient descent optimization. The model does not need to select these values *a priori*. Instead it suffices to initialise them with random values and subject them to the same learning algorithm as the weights and biases in the LSTM units. In this way, the model converges on good values automatically.

First let us consider the input layer which must distribute *i* model inputs to *h* hidden nodes. In this case **W** has shape [*i, h*] and **B** has shape [*h*]. The linear transform layer can be conceptually thought of as distributing signals from the inputs to the LTSM hidden layers in an optimal (once trained) way. The distribution function is purely linear, but this is sufficient for the desired effect.

More complex non-linear relationships between inputs may emerge from the LTSM units if appropriate.

The output layer is symmetrical in design, distributing the outputs of *h* hidden nodes to *n* model outputs. In this case **W** has shape [*h, n*] and **B** has shape [*n*].

5.3 Training

The research goal is to train the model over an arbitrarily defined time period, typically many years. The reason for such a longitudinal view is that markets experience different overall volatility levels at different times. A good way to visualise this is by graphing a market volatility measure such as the VIX Index (Chicago Board Options Exchange, 2016). If the model is not trained over a long term time frame, if will be unable to predict market-wide volatility drift and phase shifts.

In order for the model to learn, it must be provided with expected outputs alongside the inputs. The model does not require another hyperparameter here because the expected outputs are exactly the same shape as the model outputs. These tuples of input with corresponding expected output will be referred to as *training patterns*.

To generate training patterns, the model places a sliding window across a selected history of the stock portfolio. The window therefore gives us a contiguous view of a slice of history. If the window is *t* days long, the model divides it into two contiguous segments of lengths *p* and *f* respectively such that:

$$t = p + f$$

The segment of length *p* represents the “past”, and the segment of length *f* the “future”. Of course this future is also in the past since it comes from the historical data. However, relative to the past segment it is, indeed, the future and one from which one can make a perfect prediction of volatility.

As the model undergoes training, there needs to be a way of telling how well (or badly) it is currently doing. In practice this means comparing the current forecast with the example output from the training pattern. More formally the

model must define a loss function (sometimes also called a cost or objective function in the literature) that tends to zero as forecasts become more accurate. Any function with this property is suitable, providing it is differentiable across its domain. Differentiability is important because the model will utilize the loss function in a stochastic gradient descent process. A non-differentiable function has no gradient to estimate across some of its domain, rendering such functions unsuitable for gradient descent.

For this model a relatively simple loss function has been chosen:

$$loss = \frac{1}{n}\sum(forecast_i - expected_i)^2 \text{ where } 1 \leq \text{i} \leq \text{n}$$

where *n* is the number of forecasts. The loss estimate is therefore the mean squared error (MSE) of the forecasts versus what was actually observed in the stock history. MSE is a commonly used loss function for many statistical applications.

With the loss function defined, the last piece of the puzzle is how the model learns to optimize that function. Bottou (2010) recommends the use of stochastic gradient descent for the efficient optimisation of large scale machine learning problems. The basic technique essentially estimates the gradient of the loss function from a single data point (or a small subset of points) and then adjusts the scalar parameters of the function so as to move down the gradient. This process is applied iteratively in order to find a minimum. The distance moved down the gradient on each step is determined by a parameter known as the learning rate.

The scalar parameters of the loss function are the weights and biases found within the model's LSTM units and linear transformation layers. By adjusting these scalar values, the optimization algorithm is causing the network the learn how to better predict volatility from its inputs.

As it adjusts the weights and biases, it is quite possible for a stochastic gradient descent to get "stuck" in a local minimum rather than finding the global one. Furthermore, it is highly likely that the loss function, which is highly dimensional, will have many local minima. Is this a cause for concern? One mitigating factor is that stochastic nature of the process introduces noise

into each iteration. This is due to the random selection of the points used to estimate the gradient. In practice, this tends to “jolt” the algorithm out of local minima.

Another countermeasure to local minima is to select a specific descent algorithm that is robust in such cases. Ruder (2016) describes how saddle points in the loss function can be especially troublesome and cites a number of widely used algorithms that address this problem. The most recently invented algorithm on his list is *Adam*.

Adam, which stands for Adaptive Moment Estimation, is described by Kingma and Ba (2015) as an “efficient stochastic optimization that… computes individual adaptive learning rates for different parameters from estimates of the first and second moments of the gradients”. In practice, provides robustness against saddle points as well as good learning outcomes. Kingma and Ba go on to present experimental data which shows Adam outperforming other widely used algorithms. Based on this evidence, Adam was chosen as the optimization algorithm for the model.

5.4 Input Features

For the purposes of providing input to a neural network, data must be organised into *features*. While a feature might be anything salient to the model’s requirements, it must necessarily be mapped to a numerical representation. There exists “a common misperception that the inputs… must be in the interval [0, 1]” (comp.ai.neural-nets, 2014), however this is not the case. The optimisation algorithm will choose weights and biases to deal with any reasonable input range.

One practical limit on the choice of features is hardware constraints. The model is designed for deployment on commodity GPU hardware. This effectively constrains all values to IEEE-754 single precision floating point. (IEEE Standards Committee, 2008) which have an 8 bit exponent and a 23 bit fraction. This provides a range of approximately $+/-10^{38.53}$ which is more than sufficient for the model’s needs. However, because the fractional part is of fixed precision, rounding errors can arise and be magnified through a large network of calculations. Are rounding errors going to be a problem? Gupta et

al (2015) make the claim that high precision is generally unimportant for machine learning applications. The intuitive explanation is that neural nets are stochastic by nature and are therefore automatically noise tolerant. Rounding and quantization is just another (small) source of noise which will be automatically filtered during the training process.

The model has access to two sources of potential input features: the CRSP Database of US Stocks (CRSP, 2015) and Google Trends (Google, 2015).

5.5 CRSP Features

The CRSP database provides a time series of price and related trading data for stocks, with one set of observations per issue per trading day. How does one choose or construct features from these observations? In general features should carry a useful signal related to the desired output. Features that carry little to no signal are essentially a source of noise. If a few poor features are selected, the model can learn, over time, to discount or ignore the extraneous information. However, this extra work means that the model may take longer to converge to produce good forecasts. So while mistakes are not critical, one should strive for relevance in feature selection.

One field in the CRSP database that seems intuitively suited as a feature is *ret,* which represents the return including distributions since the last trading day. Return clearly carries a strong signal about volatility. Indeed, the classical definition of historical volatility only uses this variable.

More formally, the documentation defines *ret* as:

$$r(t) = \frac{p(t)f(t) + d(t)}{p(t-1)} - 1$$

where r(t) is return at time t, and p(t), f(t) and d(t) are the price, price adjustment factor and dividend at time t respectively (CRSP, 2015). In practice r(t) was observed to vary across an approximate range of [19,-1] throughout CRSP history[1], with a mean of 7.84×10^{-4} and a standard deviation of 4.28×10^{-2}. Because these are raw returns, they must be log normalized so as to compound properly:

[1] The extraordinary return of 19 occurred on 18th March 2009 for the symbol VOXW.

$$r'(t) = \log(1 + r(t))$$

Recalling that features merely need to fall within the constraints of single precision floating point values, r'(t) can directly be used as a feature. However, the low magnitude of the standard deviation of return is something that will need revisiting when designing model outputs.

This definition of r'(t) also has the attractive property that it is normalised across all stocks in the portfolio. A 5% increase in return is directly comparable whether the stock is trading at $5 or $500. This normalisation therefore captures the movement of the underlying variables (p, f and t) while factoring out idiosyncratic parameters of the stock's history, such as listing price.

There is information lost to the normalisation process. For instance, it might be observed that issues with high absolute prices tend to have been around for a while. For instance the highest priced US stock, BRK.A, at the time of writing trades at over $200,000 and its current business model dates back to the late 1960s. Normalization denies the model any chance of inferring correlations like this from the absolute price. However, such correlations are not very strong anyway; there are many old businesses with low prices and many new "unicorns" in hot industries with high prices. Furthermore, the countervailing argument is that absolute price has little to no bearing since volatility is defined solely in terms of return.

Another reason that normalisation is attractive is that it allows features to use a zero origin for their range. In other words, features may be defined so that zero means no change from the last trading day. This is important because the historical data has gaps. Not all stocks trade on every trading day. Sometimes they are suspended for one reason or another. Furthermore, when looking across long time ranges for a given portfolio, there may well be periods before some of the stocks were listed or after some were delisted. Since the model always requires the same number of inputs, however, those gaps must be filled with dummy, or default, data. By choosing a zero origin, the default feature may be assumed to be zero, thereby making it easy to fill the gaps.

Other fields in the CRSP database are likely to be of relevance to volatility forecasting and should be examined as possible features. For instance the *bidlo* and *askhi* fields provide the lowest bid price and highest ask price of the day. Intuitively, it seems likely that these numbers capture something about the range of investor sentiment or the “frothiness” of the stock, and that this is potentially linked to volatility. There exists a temptation to calculate the difference, or spread, and use the result. However spread is an intraday measure and it is not clear how it should be normalised around zero. One possible solution to this problem is to use the change in spread between trading days instead. However, even more information may be delivered to the model by using the day to day change in *bidlo* and *askhi* as two distinct features. The model is quite capable of inferring the change in spread from these more fundamental inputs, if required.

This analysis highlights an important rule of thumb. Each time a feature is computed from more fundamental variables, some information may be lost to the model. Conversely, “uncooked” features give the model the opportunity to combine them in ways that might be non-obvious. The rule of thumb is therefore to design features to be as close to the source data as possible subject to the goals of inter-stock normalisation with a zero origin.

In order to convert intraday variable like *bidlo* and *askhi* into a normalized feature the model will therefore compute the relative change between trading days. More formally, the feature at time t is defined as:

$$f(t) = \frac{x(t)}{x(t-1)}, \forall\, x(t-1) \neq 0$$

$$f(t) = 0, \forall\, x(t-1) \neq 0$$

where x(t) is the intraday variable.

The precise set of CRSP intraday variables to choose for this treatment is as much an art as a science, driven by perceived relevance to volatility as discussed earlier. For the purposes of the model, the following variables will be used to synthesize features:

- *askhi*: the highest intraday asking price

- *bidlo*: the lowest intraday bidding price
- *vol*: trading volume
- *ask*: closing asking price
- *bid*: closing bidding price

The model also directly uses the log normalised *ret* variable as a feature, as discussed above, plus the analogous *retx* variable, also log normalised, which measures return without dividends.

Several CRSP variables were considered as potential features but rejected as being unsuitable or of only marginal benefit. For instance, the closing price could be used to construct a day to day feature, but the result is close enough to *retx* to be superfluous. Opening price probably carries more information since it reflects the pre-market price discovery process. However since this is conducted by observing bid/ask spreads the model is already receiving much of this signal from the features constructed over *bidlo*, *askhi*, *ask*, and *bid*. Finally, the *numtrd* variable provides the number of trades for NASDAQ listed issues. This carries distinct information from the aggregate trading volume and is potentially a valuable signal contributing to volatility. However the portfolio will not be restricted to NASDAQ stocks. This raises the possibility that a feature constructed over *numtrd* could be zero for many stocks. The effects of this information asymmetry on model stability and convergence are unclear and therefore the model does not use this information in its current form (although it would be trivial to add).

5.6 Google Trends Features

In addition to the CRSP database, Google Trends data is available for each stock symbol in the portfolio (Google, 2015). This data set maps a time period (generally a week) to a “trend” value between 0 and 100. The trend value is a relative measure of search activity for the symbol as a term on the Google search engine.

The working hypothesis is that at times of unusual volatility there may be heightened search activity for the relevant symbol. Google search is a pervasive tool used by a broad cross section of the population; it potentially captures the investment zeitgeist in ways different to the CRSP data and, in

some cases, earlier. Therefore the model may be able to learn to incorporate this signal to produce better forecasts. There are three interesting classes of volatility shift that may be signalled by Google Trends: idiosyncratic, sector and systemic.

Idiosyncratic volatility shift affects only a single, or a handful of stocks. Usually it occurs in response to company specific news or rumours. The result is often a substantial shift in volatility and trading volume. While it seems unlikely that the model can predict idiosyncratic events, it seems plausible that internet searches might be able to signal changes early and adjust forecasts accordingly.

Sector volatility affects a broad class of stocks. Usually it occurs in response to business conditions that are confined to the sector. An example would be when a particular industry attracts increased speculation as often happens with technology stocks. Another example might be when demand quickly changes for a major commodity. When a significant number of companies across a given sector see heightened search activity then the likelihood of a volatility shift for that entire sector is potentially higher and the model may be able to take advantage of this effect.

Systemic volatility affects the broader market. Usually it occurs when the business cycle changes, or occasionally when there is a general correction under way. Major world events may also trigger a shift in market wide volatility. The signal in internet searches for this type of shift may be more subtle than the previous two classes precisely because the entire market is moving. Even during the Financial Crisis of 2007-2008 there were no obvious correlated changes in search activity across all or even a majority of symbols in the data set. However, just because the signal is not obvious does not mean it is not there. If there is a signal for system volatility the model should be able to isolate it with enough training.

The working hypothesis has several potential flaws. People may not search for the stock by symbol but rather by name. Some stock symbols may collide with commonly used acronyms and therefore activity for that search term does not correlate well with interest in the stock. However, the biggest

weakness is the coarseness of the time resolution. As mentioned above, each trend value for a symbol is generally for a one week interval. This substantially reduces the information that the model has to work with and makes it less timely. Since the rationale for including in it the model is to get an early signal of volatility shift, this might be a significant weakness.

However, it is unknown how much these potential flaws reduce the utility of the data to the model. Therefore it was felt useful to include trend data in order to determine the matter empirically.

The model can easily match the date range and symbol of each trend observation with a corresponding CRSP issue and trading day. This allows it to synthesize a trend feature. Since the trend feature often will not change from day to day (because of the week long blocks) the method used to normalize CRSP intraday variables is not appropriate in this case. Instead, the trend value is simply scaled to a range of [0, 1] and used as is.

5.7 Input Tensor

Based on the preceding sections, it is helpful to review the final form of the model's input tensor.

There are up to eight scalar features organised into a *feature vector*. A fully populated vector is of the form:

<Δaskhi, Δbidlo, ret, Δvol, Δask, Δbid, retx, trend>

where Δ indicates a normalised feature constructed from the change between two trading days.

There is a feature vector provided for each stock on each trading day. This may be represented as a tensor of shape [*n*, *f*], where *n* is the number of stocks in the portfolio and *f* is the number of features selected. For instance, if the portfolio consists of the S&P100 (which had 97 members as of the 30th of September 2015) each trading day generates 97 feature vectors, and if the model uses all possible features the resulting tensor is of shape [97, 8]. This contains a total of 776 scalar features.

Each training pattern provides an input consisting of a contiguous sequence of trading days. This may be represented as a tensor of shape [*s, n, f*], where *s* is the number of steps, or trading days. For instance, if the training patterns provide 30 days of history across the example above the resulting tensor is of shape [30, 97, 8]. This contains a total of 23,280 scalar features.

The model processes training patterns in mini-batches for computational efficiency. This may be represented as a tensor of shape [*b, s, n, f*], where *b* is the batch size. For instance, if the batch size is 20 patterns from the example above the resulting tensor is of shape [20, 30, 97, 8]. This contains a total of 465,600 scalar features. Recalling that these values are represented as 32 bit single precision floating point, the model consumes approximately 1.8 megabytes of input data per iteration under this scenario.

The final input tensor is therefore of rank 4. This high dimensionality means that when one varies a model hyperparameter that specifies one of these dimensions the effective change on the total number of inputs can be large. In turn, the number of inputs drives the size of the LSTM units and the linear transformation layers in the model. Because the model is designed for deployment on commodity GPU hardware the available hardware resources impose a hard limit on the size of the input tensor. The easiest way to determine this limit is empirically.

5.8 Output Tensor

The model's output tensor is considerably simpler than its input. A volatility prediction for each stock in the portfolio is returned for each batch. It is therefore a tensor of rank 2 and shape [*b*, *n*].

However, there is an additional design issue with the output tensor that must be considered. What type of values should its members take on? Recall that when the model is being trained, it is being passed not only an input tensor but a corresponding expected output tensor (when the model is not training the expected output is simply ignored). The model learns from these patterns and, after (hopefully) converging, will output similar values as the forecast. This means that there is considerable freedom in defining the output

representation. However, unlike with input choice, output selection can exert great effect on performance.

In this case, there are two obvious output representations: variance or standard deviation of returns. Consider that the standard deviation of all daily returns in the CRSP data set is 4.28×10^{-2}. Further consider that the variance is the square of this, or 1.83×10^{-3}. Note that since the variance is, in itself, an average, that some subset of the data will necessarily have even smaller sample variance.

The magnitude of the output values takes on significance when one considers the stochastic gradient decent process. As described earlier, the basic algorithm is to estimate the gradient of the loss function for the current inputs and to adjust the scalar parameters so as to move down that line. The distance moved depends on estimated gradient scaled by the chosen learning factor. The sought after effect is to rapidly descend when the gradient is steep and to more slowly descend as the gradient lessens. This allows the algorithm to quick get close to a minimum and then explore the space in that region with more resolution.

Recall that the loss function is defined as the mean squared error between the forecast and the expected outputs. When using variances as outputs, even a large error will still have small magnitude. Consider an expected variance of 1×10^{-3}. If the forecast is 2×10^{-3} that is 100% error, yet the difference is 1×10^{-3} and the calculated loss is 1×10^{-6}. This can result in a small gradient estimate, even though the error is huge, with the result that the model takes more training iterations to converge.

In this case it is clearly better to use standard deviation rather than variance. Even better is to make the output values larger again, which one can do by annualising the volatility according to the common method:

$$\sigma_{annual} = \sigma_{daily}\sqrt{252}$$

Note that this choice of output value does not change the theoretical operation of the model. However, it is a useful real world optimisation. It has

the happy side effect of also making the output easier to interpret since it is conventional to quote volatility as an annualised value.

5.9 Implementation Outline

Abadi et al (2015) introduced the TensorFlow system, which is specifically designed for quickly and efficiently implementing machine learning algorithms, including deep neural networks. The system was developed by the Google Brain project and open sourced in late 2015. The core idea is that a “computation is described by a directed graph, which… represents a dataflow computation” (Abadi et al, 2015).

A dataflow architecture permits an elemental computation to occur as soon as its inputs are available. The theoretical number of concurrent operations is only bounded by the parallelism inherent in the expressed algorithm. In practice, of course, there may be a limited number of actual computation units available to which work must be scheduled. Describing algorithms in this way, while quite different from common programming practice, allows the underlying parallelism in the algorithm to be easily exploited.

As it turns out, neural net architectures are intrinsically highly parallel since each neuron may fire as soon as its inputs are ready. Indeed describing neural net architectures as a data flow is an entirely natural mapping.

The TensorFlow system, quite conveniently, provides high level building blocks for recurrent neural networks, LTSM units and stochastic gradient descent optimizers “out of the box”. This made constructing the model remarkably straightforward, allowing the author to concentrate on the research goals rather than the software plumbing. In fact, the supporting code to marshal the input data and assess the resulting forecasts was substantially larger than the model itself. Furthermore, TensorFlow enables the easy deployment of the resulting models on dedicated GPU hardware with essentially no further effort. This allowed the model to be developed on a laptop and then moved without change to cloud services supporting commodity GPUs for full scale testing.

The detailed operation and API of the TensorFlow system falls outside the scope of this report. It is well documented by Abadi et al (2015) and online at http://tensorflow.org. However, the salient point is that it was an invaluable tool for constructing and evolving a conceptually complex recurrent neural net. The author experimented with other alternatives before ultimately adopting TensorFlow for the project and the difference was chalk and cheese. Rao (2015) put it best when he said “upgrading to TensorFlow should feel like moving from a Honda Civic to a Ferrari”.

The most complete definition of the final model, of course, is the source code. While the code of the entire system is too large to include in this text, the core model itself is small enough to be presented in summarized and annotated form. As the high level TensorFlow libraries supporting neural net building blocks are implemented in the Python language, this was the language chosen to express the model.

In order to learn how to use TensorFlow to build the model, the author drew on tutorials from TensorFlow (2016) and Damien (2016). The Tensorflow tutorials demonstrate common conventions, basic building blocks and how to combine them. Damien demonstrates how to efficiently transform batched input firstly for linear transformation and then for passing into a recurrent neural network building block. The author directly adopted techniques and structures from these sources when implementing the model.

To begin, the model accepts 6 hyperparameters: the batch size, the number of time steps in each input window, the number of inputs, the number of outputs, the number of hidden inputs/outputs in the LSTM stack and the number of layers of LTSM units. Note that in this implementation, the input tensor has been dropped to a rank of 3 by merging all the feature vectors for a given trading day into one giant vector in a preceding step. Therefore the code does not require the number of features per feature vector, or the

number of stocks in the portfolio.

```
    def __init__(self, batchsize, steps, inputs, outputs,
            hidden, layers):
        """Create a forecasting model.

        Parameters:
        batchsize -- size of learning batch
        steps -- number of input time steps
        inputs -- number of inputs
        outputs -- number of outputs
        hidden -- number of hidden nodes
        layers -- number of layers
        """

        self._batchsize = batchsize
```

Next, placeholders are created for the model inputs. A placeholder is simply a node via which input tensors are fed into the dataflow graph. There are actually two inputs to consider: the true input tensor from which the model generates a forecast and the example output tensor which is used to calculate the loss function. These are conventionally known as x and y, respectively. The shape of these tensors must be defined, however TensorFlow allows us to leave the first dimension unspecified as “None”. In this case, the unspecified dimension is batch size.

```
        # create input pattern placeholders
        self._x = tensorflow.placeholder(tensorflow.float32,
            [None, steps, inputs])
        self._y = tensorflow.placeholder(tensorflow.float32,
            [None, outputs])
```

The input tensor is of the shape [batches, steps, inputs]. TensorFlow provides a recurrent neural network building block that requires an input tensor with steps as the outermost dimension. In preparation for this, the input is transposed resulting in a tensor of shape [steps, batches, inputs].

```
        # transpose input for rnn core
        # this technique adopted from Damien (2016)
        x = tensorflow.transpose(self._x, [1, 0, 2])
```

Next, the inputs must be passed through a linear transformation layer in order to distribute them to the hidden nodes in the first LSTM unit. The

transformation requires a matrix of weights and a vector of biases, which are initialised to random values following a gaussian distribution. The input tensor must be flattened before applying the transform to each input vector. The resulting tensor is of shape [steps * batches, hidden].

```
# create weights and biases for inputs
input_weights = tensorflow.Variable(
   tensorflow.random_normal([inputs, hidden]))
input_biases = tensorflow.Variable(
   tensorflow.random_normal([hidden]))

# apply input linear transform
# this technique adopted from Damien (2016)
x = tensorflow.reshape(x, [-1, inputs])
x = tensorflow.matmul(x, input_weights) + input_biases
```

Next, a stack of LTSM cells is created, with each cell feeding the next. The number of inputs and outputs for each cell is the hidden nodes hyperparameter.

```
# create a multilayered stack of basic LSTM cells
cell = tensorflow.models.rnn.rnn_cell.BasicLSTMCell(hidden)
stack = tensorflow.models.rnn.rnn_cell.MultiRNNCell([cell] * layers)
```

Next, the LTSM stacks are used to create a recurrent neural network. This network expects an input tensor of the shape [steps, batches * inputs], so a final split operation is performed on the input to yield that shape. If a previously trained neural net was being re-instantiated, the state would be initialised from the previous training sessions. In this case, an untrained network is simply initialised with zero state.

```
# create RNN network
x = tensorflow.split(0, steps, x)
initial = stack.zero_state(batchsize, tensorflow.float32)
out, states = tensorflow.models.rnn.rnn.rnn(stack, x,
   initial_state = initial)
```

The neural net produces an output tensor of the shape [steps, batches, hidden]. Only the last step is of interest, as that output is the forecast based on all the preceding steps. Therefore that tensor of shape [batches, hidden] is passed through a second linear transformation layer in order to populate the outputs of the model. This requires another set of weights and biases,

again initialised randomly and the resulting output tensor is of shape [batches, outputs].

```
 # create weights and biases for outputs from the LSTM stack
output_weights = tensorflow.Variable(
   tensorflow.random_normal([hidden, outputs]))
output_biases = tensorflow.Variable(
   tensorflow.random_normal([outputs]))

# apply output linear transform
# this technique adopted from Damin (2016)
self._output = tensorflow.matmul(out[-1], output_weights)\
      + output_biases
```

In order to train the model, a loss function is required. In this case loss is defined as the mean squared error of the output (forecast) compared to the expected output.

```
# define loss function
self._loss = tensorflow.reduce_mean(
   tensorflow.square(self._output - self._y))
```

Finally, the model requires a stochastic gradient descent optimiser to adjust the neural net in order to be able to learn from the provided examples.

```
# define optimizer
self._optimizer = tensorflow.train.AdamOptimizer()\
     .minimize(self._loss)
```

5.10 Model Visualisation

The Tensorflow system provides a tool called TensorBoard which, among other things, can provide a visualisation of an instantiated dataflow network. For instance the model with hyperparameters of batchsize = 20, steps = 30,

inputs = 776, outputs = 97, hidden = 776 and layers = 4 is depicted as:

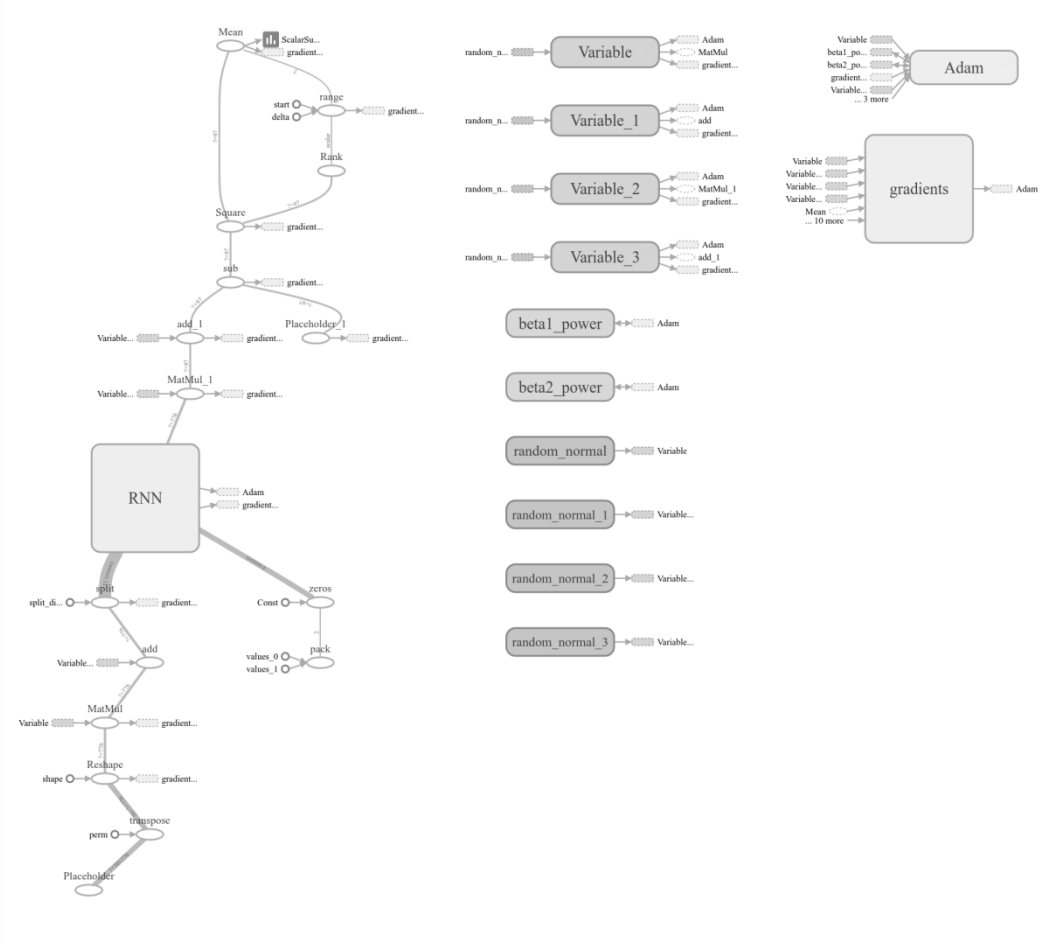


Figure 1: Example Model Visualisation (Source: TensorBoard software)

It can be seen that the relatively simple code outlined above produces quite a complex structure. Furthermore, the RNN core in the diagram expands to show 30 internal units, each 4 layers deep.

6 Experimental Results

A sequence of 15 experimental scenarios, denoted *alpha* through *pi*[2], was devised to determine efficacy of the model. Each experiment follows the same general protocol:

For each scenario, a portfolio of stocks is defined along with a start and an end date. A time series of feature vectors, one per stock per trading day is

[2] *Lambda* was omitted from the naming sequence since it conflicted with a reserved word in the Python programming language.

generated across this history. If a stock did not trade on a day, a default feature vector is substituted.

A sliding window is run across this time series in order to generate fixed sized frames of feature vectors, representing an ordered, contiguous sequence of trading days. Each frame is divided into two parts, a “past” and a “future”. The standard deviation of returns across the future part is calculated. This becomes the expected output. The past segment of the frame and the expected output, put together, form a training pattern.

The size of the past and future parts of sliding window is selectable, as is the “stride” which determines how many days forward the window moves each time it slides.

In general, a large number of training patterns are generated. Each pattern is random placed in one of two sets: a training set and a testing set in the proportion 9:1. Therefore 90% of the patterns are used for training purposes and 10% for testing purposes. Patterns from the two sets are never mixed in the course of an experiment.

A training epoch consists of randomly shuffling the training set and then training the model with each pattern in the set exactly once. At the end of each epoch the model is tested for accuracy by shuffling and using patterns from the test set. Patterns from the test set are never shown to the model when it is in learning mode. This blinding protocol ensures that the model effectively has no *a priori* knowledge of the testing data, so that the results are representative of how the model might perform when making true forecasts.

In testing mode, the model’s performance is measured by an objective function defined as the root mean squared difference of the forecast versus the observed future. More formally:

$$objective = \sqrt{\frac{1}{n}\sum(forecast_i - expected_i)^2} \quad \text{where } 1 \leq \text{i} \leq \text{n}$$

Recall that the model defined the output values to be annualised volatility. Therefore the objective function may be informally interpreted as a measure of error in annualised volatility.

By observing the objective function it can easily be determined if the model is getting better (objective function decreases) or getting worse (objective function increases). While useful to see if training is having any effect, this is an internally relative measure and it tells the observer nothing about how well the model is performing versus other methods of forecasting volatility.

In order to assess real world performance, some benchmarks are required. For this experimental protocol, three have been selected:

- Simple unweighted historical volatility
- An exponential moving average (EWMA) of historical volatility
- A GARCH(1,1) model of historical volatility

Each of these methods is used to forecast volatility using the same input data as the model. The objective function for each method can then be directly compared.

The EWMA uses a decay factor of 0.94 as is common practice (González-Rivera, 2007).

The GARCH(1,1) implementation used was provided by Sheppard (2015). In practice, this implementation may generate negative or large variances when it lacks enough information to converge correctly. Empirically, across a large data set this occurred often enough to regularly perturb the objective function. The filter out this effect, daily variance forecasts from the model are clamped to a range of [0, 0.04] because negative variances make to sense and anything larger than 0.04 suggests a >20% move in price which is large enough to trigger a trading halt.

In addition to the objective function, it was felt that a measure of maximum error might also be of value. A forecasting model that is good on average but makes whopping mistakes from time to time is qualitatively different to one with the same average but lower outliers. Therefore the experimental

protocol also defines a measure of maximum error which is also calculated for the model's forecast and the comparison benchmarks:

$$\text{maxerror} = \max(|forecast_i - expected_i|) \text{ where } 1 \leq \text{i} \leq \text{n}$$

6.1 Scenario Alpha

The first test scenario, Alpha, was designed to establish a baseline from which to explore the model's performance. It attempts to forecast the volatility of the 97 member stocks of the S&P 100 index, as constituted on September 30th 2015.

The historical range chosen for this scenario was from 1st October 2005 to 30th September 2015. This provides the model with exactly one decade of data. Ten years was chosen because it is long enough to have seen different volatility modes in the market, including periods of expansion, contraction and the global crisis of 2007-2008. At the same time it is a period short enough to make multiple experimental runs practical.

A sliding window of 40 trading days with a stride of 1 day was used to generate the training patterns. Each window was defined to be 30 days of "past" data and "10" days of future. This means the model will be forecasting volatility across 10 trading days, using the preceding 30 trading day's history. Because there are usually 5 trading days in a week, this means the model will be looking approximately 6 weeks into the past in order to forecast the next fortnight of real time. An aggregate two months real time was chosen because it is congruent to the expiry period of a common class of standard equity options (Chicago Board Option Exchange, 2016). Volatility forecasts in this time frame are therefore potentially useful to traders, using models like Black-Scholes, to make decisions at various points in the lifetime of a typical option. In addition, a predictive period of 10 trading days might be useful to many organisations managing risk on a weekly or fortnightly basis using tools like VaR.

This scenario covers 1258 trading days, from which 2516 patterns may be generated. Of these, a random selected 2264 (90%) were used for training and the remainder, 252 (10%) for testing.

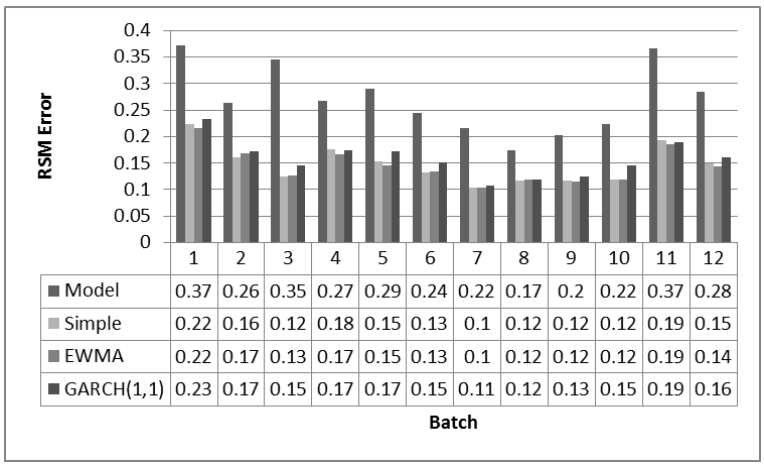
Since the intent of this scenario was to baseline model performance the only stock feature it used was the daily return (including dividends).

The model hyperparameters that must be explicitly specified are batch size, number of hidden nodes and the number of layers (the rest were fixed by the choice of input data).

A batch size of 20 was chosen to balance parallelization versus available GPU resources. Larger batches mean more parallelization and reduced communication costs with the GPU (Li et al, 2014). With a batch size over 20, however, it was empirically determined that some of the later, more complex models would fail to run. Therefore the “goldilocks” value of 20 was chosen as the baseline; not too small and not too large, but just right.

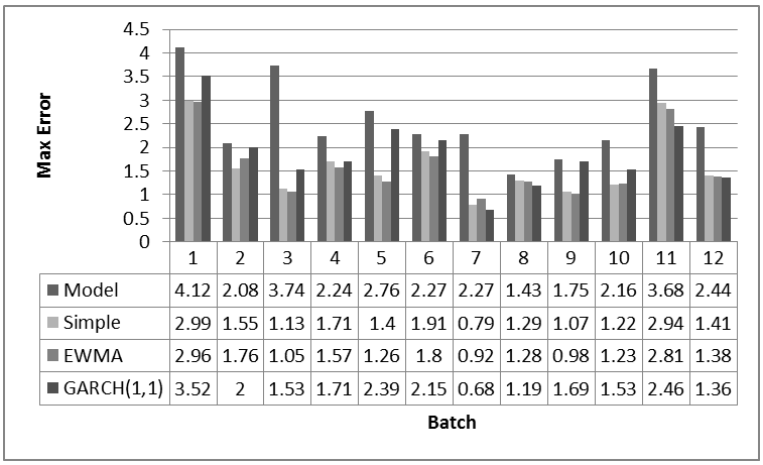
Again keeping in mind the goal of establishing a baseline, the number of hidden layers was set equal to the number of model inputs (i.e. the number of features times the number of stocks) and the network was defined as one layer deep.

After training for one epoch, 12 batches were randomly constructed from the corpus of 252 test patterns and the objective function measured. The following graphs show the value of the objective function and the maximum error for each of the batches:



	1	2	3	4	5	6	7	8	9	10	11	12
Model	0.37	0.26	0.35	0.27	0.29	0.24	0.22	0.17	0.2	0.22	0.37	0.28
Simple	0.22	0.16	0.12	0.18	0.15	0.13	0.1	0.12	0.12	0.12	0.19	0.15
EWMA	0.22	0.17	0.13	0.17	0.15	0.13	0.1	0.12	0.12	0.12	0.19	0.14
GARCH(1,1)	0.23	0.17	0.15	0.17	0.17	0.15	0.11	0.12	0.13	0.15	0.19	0.16

Figure 2: [Scenario Alpha] RSM Error After One Epoch



	1	2	3	4	5	6	7	8	9	10	11	12
Model	4.12	2.08	3.74	2.24	2.76	2.27	2.27	1.43	1.75	2.16	3.68	2.44
Simple	2.99	1.55	1.13	1.71	1.4	1.91	0.79	1.29	1.07	1.22	2.94	1.41
EWMA	2.96	1.76	1.05	1.57	1.26	1.8	0.92	1.28	0.98	1.23	2.81	1.38
GARCH(1,1)	3.52	2	1.53	1.71	2.39	2.15	0.68	1.19	1.69	1.53	2.46	1.36

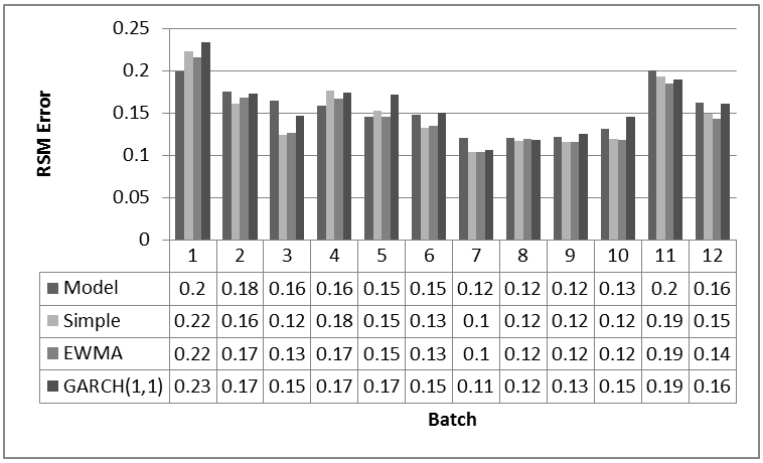
Figure 3: [Scenario Alpha] Max Error After One Epoch

The model clearly performed much worse than all three of the benchmarks, across each and every batch. Looking at batch 11 specifically, the forecast was nearly twice as bad. This is hardly surprising since the model has had no chance to reinforce what it has learned from the first epoch. Just like a biological system, a neural net can benefit from repetition.

The benchmarks, on the other hand, were all quite close in performance as would be expected. Interestingly, GARCH(1,1) appears to the worst performer of the three which is consistent with the findings of Poon and Granger (2003). It is arguable that GARCH is not well suited to this particular forecasting window.

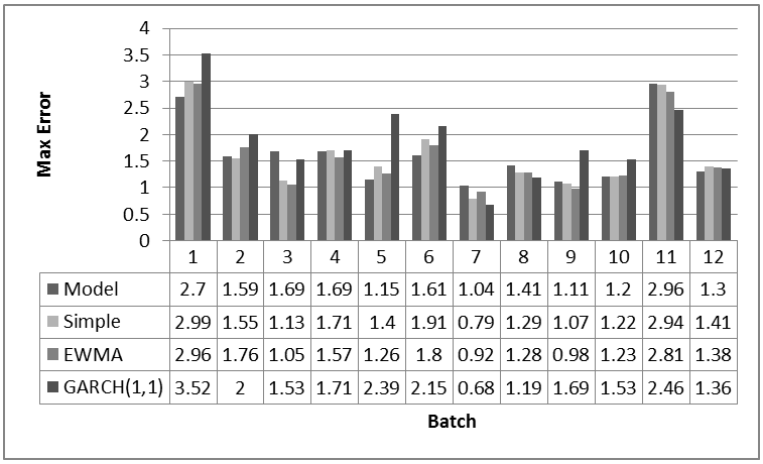
What happens when the model is trained for more epochs? How many epochs should it train for? Early versions of the model were programmed with training termination criteria, triggered when the mean of the root-mean-squared error of the training batches increased between epochs. However, it was empirically determined that this occurs quite frequently and, furthermore, that it often signalled that the model was escaping a local minimum and would continue to improve substantially thereafter. The construction of more complex termination criteria is a possible solution, but the alternate approach of simply training for a fixed number of epochs was adopted. For this scenario, a 15 epoch training regime was selected.

After the full 15 epochs the object function and maximum error may be graphed again to unveil a significant improvement in the model's performance:



	1	2	3	4	5	6	7	8	9	10	11	12
Model	0.2	0.18	0.16	0.16	0.15	0.15	0.12	0.12	0.12	0.13	0.2	0.16
Simple	0.22	0.16	0.12	0.18	0.15	0.13	0.1	0.12	0.12	0.12	0.19	0.15
EWMA	0.22	0.17	0.13	0.17	0.15	0.13	0.1	0.12	0.12	0.12	0.19	0.14
GARCH(1,1)	0.23	0.17	0.15	0.17	0.17	0.15	0.11	0.12	0.13	0.15	0.19	0.16

Figure 4: [Scenario Alpha] RSM Error After 15 Epochs



	1	2	3	4	5	6	7	8	9	10	11	12
Model	2.7	1.59	1.69	1.69	1.15	1.61	1.04	1.41	1.11	1.2	2.96	1.3
Simple	2.99	1.55	1.13	1.71	1.4	1.91	0.79	1.29	1.07	1.22	2.94	1.41
EWMA	2.96	1.76	1.05	1.57	1.26	1.8	0.92	1.28	0.98	1.23	2.81	1.38
GARCH(1,1)	3.52	2	1.53	1.71	2.39	2.15	0.68	1.19	1.69	1.53	2.46	1.36

Figure 5: [Scenario Alpha] Max Error After 15 Epochs

While on average, the model wad still performing worse than the benchmarks, on one batch it outperformed all three and on another it was equal best. The error rate also reduced to be much closer to the benchmarks. Another way to visualise what was happening is to graph the mean of the objective function and the maximum of the maximum errors varying by epoch, as depicted:

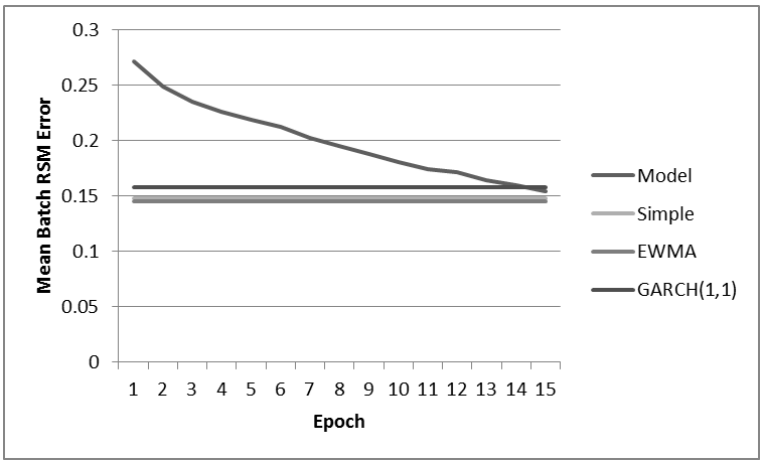


Figure 6: [Scenario Alpha] RSM Error By Epoch

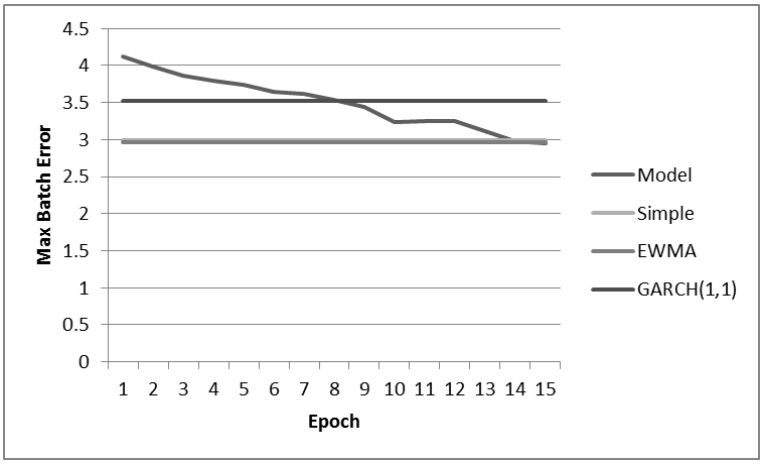


Figure 7: [Scenario Alpha] Max Error By Epoch

In these graphs, the benchmark models appear as straight lines because they were not learning to get better, epoch to epoch. The progress of the forecasting model, however, is clearly visible. It can be seen from the RSM error, that the model approaches but does not exceed the benchmarks after 15 epochs of training. This is unsurprising since the model was only provided with daily returns. In other words, it only had access to the same information that the other models were using. Nevertheless, this is encouraging because it validates that the model can learn over time. But what would happen if it had more information to work with?

6.2 Scenario Beta

The second experimental scenario, Beta, made more features available to the model. In all other respects it was the same as Scenario Alpha. The feature vector selected was:

<Δaskhi, Δbidlo, ret, Δvol >

The objective function and the maximum error were graphed across the standard 15 epochs:

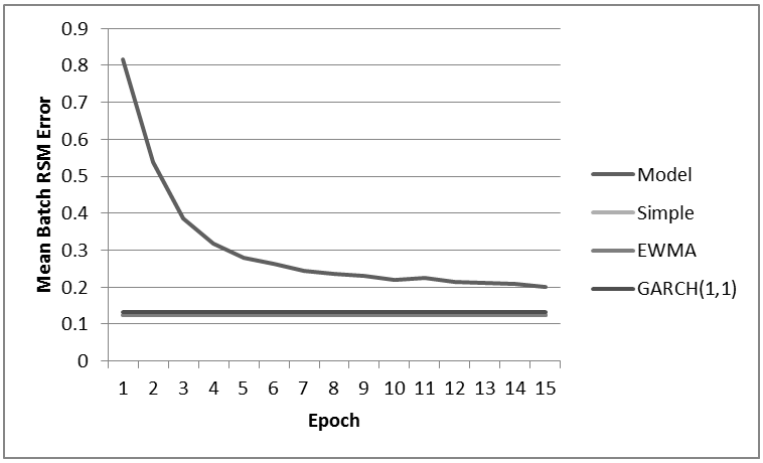


Figure 8: [Scenario Beta] RSM Error By Epoch

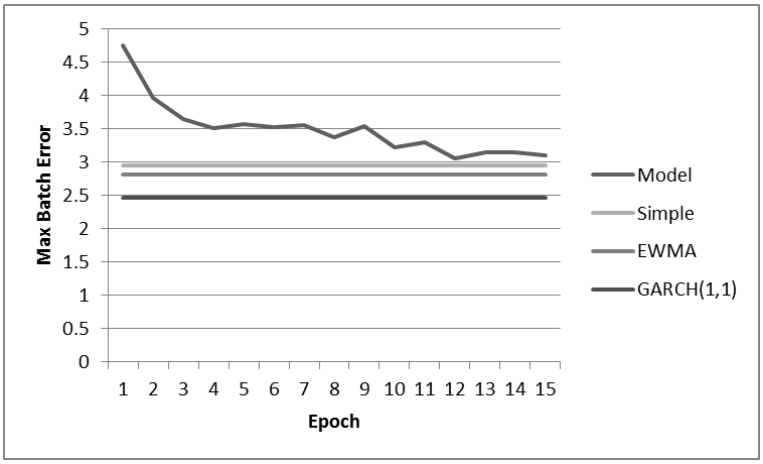


Figure 9: [Scenario Beta] Max Error By Epoch

These results are counterintuitive. The model had access to more data, but it performed substantially worse compared to the benchmarks and itself in scenario alpha. How can adding more information make performance worse? Perhaps the neural network does not have enough structure to take advantage of all the information. In that case the surplus would act as noise and slow the convergence of the learning algorithm. At present the network consists of but a single layer. What would happen if it were converted it into a deep(er) network?

Also note that the benchmark lines have moved. This is as expected because the random shuffling of data makes every experiment slightly different. This serves as a reminder that while the objective function may be directly compared within the bounds of a scenario, care must be taken when contrasting results between scenarios.

6.3 Scenario Gamma

The third experimental scenario, Gamma, doubled the depth of the LSTM units, from one to two. The model consequently became a deep network. In all other respects it was the same as Scenario Beta.

The objective function and the maximum error were graphed across the standard 15 epochs:

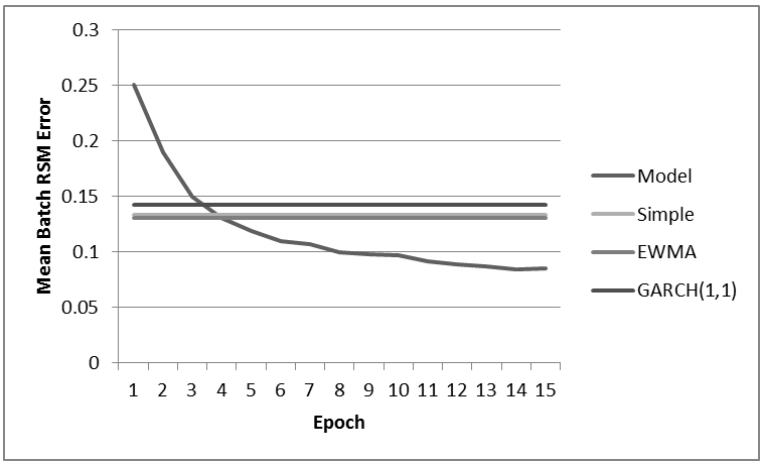


Figure 10: [Scenario Gamma] RSM Error By Epoch

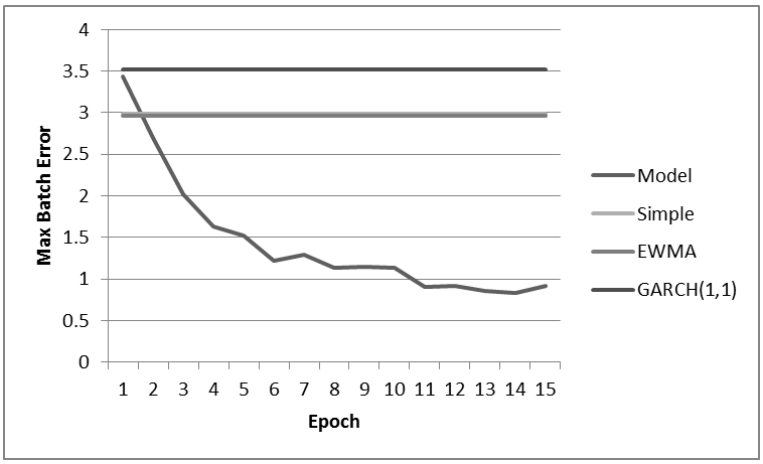
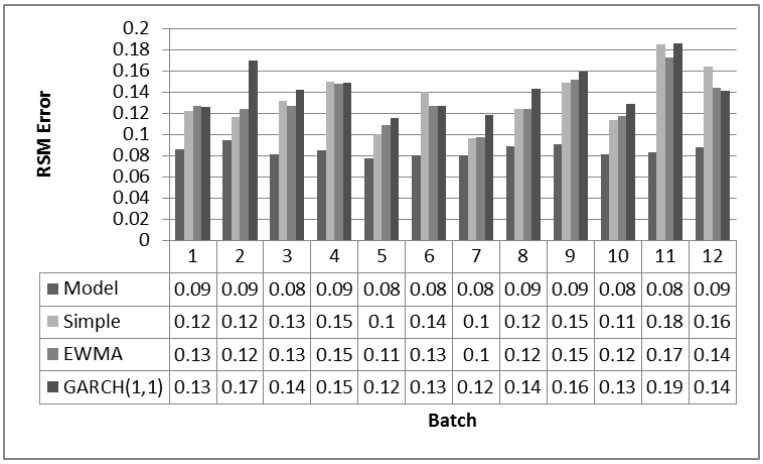


Figure 11: [Scenario Gamma] Max Error By Epoch

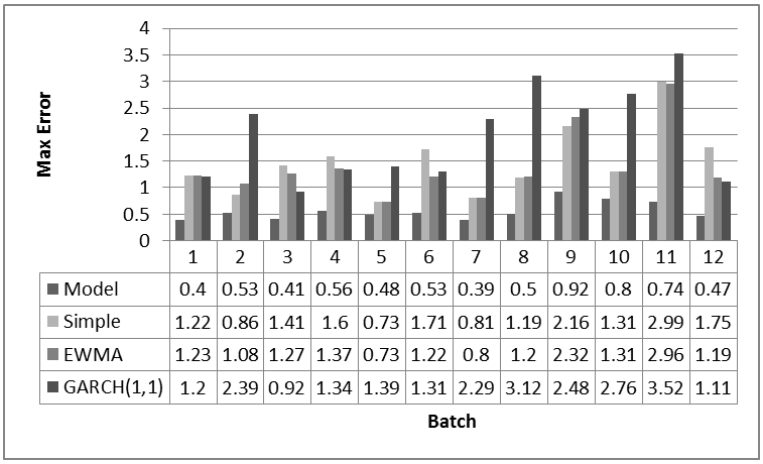
This was an extraordinarily encouraging result. The model now clearly outperformed all three of the benchmarks by a significant margin. It is genuinely surprising that simply adding depth to the network can lead to such a profound change. It is also interesting to note that the rate of improvement seemed to be levelling off around the 15 epoch mark. This suggests that the model was nearly optimised for those particular parameters and input. The maximum error story was even better than the RMS error dropping to less than one third of the benchmarks.

Given the performance improvements, it is probably worth drilling down to the per batch results on epoch 15:



	1	2	3	4	5	6	7	8	9	10	11	12
Model	0.09	0.09	0.08	0.09	0.08	0.08	0.08	0.09	0.09	0.08	0.08	0.09
Simple	0.12	0.12	0.13	0.15	0.1	0.14	0.1	0.12	0.15	0.11	0.18	0.16
EWMA	0.13	0.12	0.13	0.15	0.11	0.13	0.1	0.12	0.15	0.12	0.17	0.14
GARCH(1,1)	0.13	0.17	0.14	0.15	0.12	0.13	0.12	0.14	0.16	0.13	0.19	0.14

Figure 12: [Scenario Gamma] RSM Error After 15 Epochs



	1	2	3	4	5	6	7	8	9	10	11	12
Model	0.4	0.53	0.41	0.56	0.48	0.53	0.39	0.5	0.92	0.8	0.74	0.47
Simple	1.22	0.86	1.41	1.6	0.73	1.71	0.81	1.19	2.16	1.31	2.99	1.75
EWMA	1.23	1.08	1.27	1.37	0.73	1.22	0.8	1.2	2.32	1.31	2.96	1.19
GARCH(1,1)	1.2	2.39	0.92	1.34	1.39	1.31	2.29	3.12	2.48	2.76	3.52	1.11

Figure 13: [Scenario Gamma] Max Error After 15 Epochs

It can be clearly seen that the model was outperforming the benchmarks consistently on every single batch and the same was true for the maximum

error. It is striking how the model's RMS error and max error remain within a narrow band across all the batches while the benchmarks vary more widely. This suggests that not only is Scenario Gamma better at forecasting volatility than the benchmarks, but that it is more stable.

What would happen if the model's LSTM network were even deeper?

6.4 Scenario Delta

The fourth experimental scenario, Delta, doubled the depth of the LSTM stack again, from two to four. The model therefore became a much deeper network. In all other respects it was the same as Scenario Gamma.

The objective function and the maximum error were graphed across the standard 15 epochs:

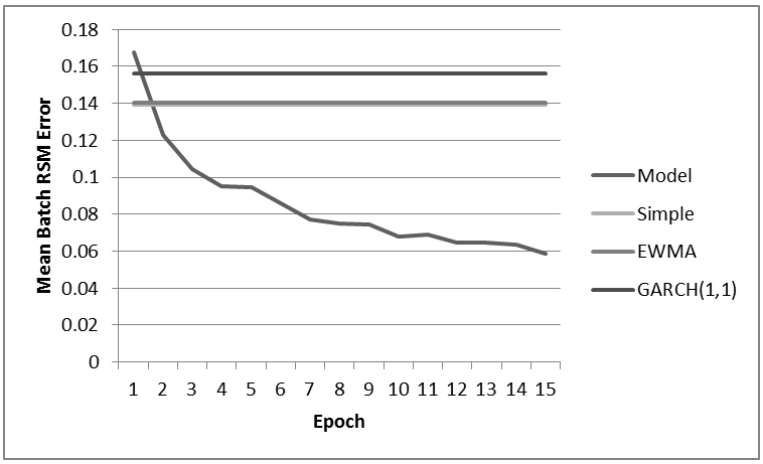


Figure 14: [Scenario Delta] RSM Error By Epoch

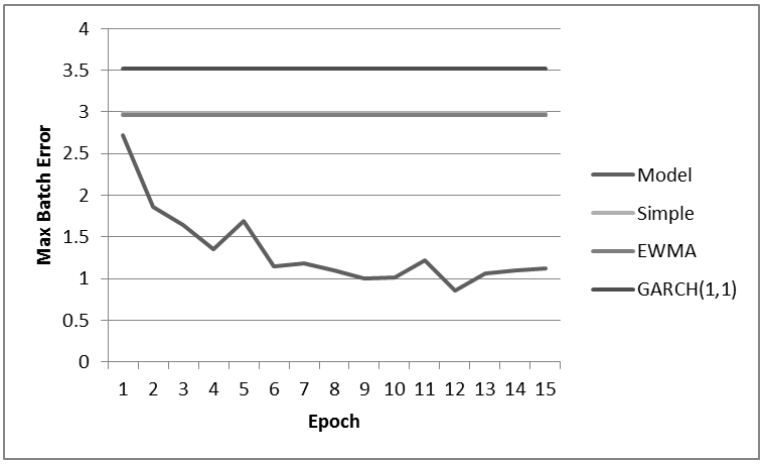


Figure 15: [Scenario Delta] Max Error By Epoch

Here another step in performance may be observed. The objective function reaches 0.06 as compared to around 0.08 as seen in Scenario Delta. While one should take care in comparing the absolute value of the objective function between scenarios, it can be observed that the benchmarks are averaging around the same level in both cases. This gives some confidence to the claim that performance has improved.

However, there are two additional observations to make. Firstly, the *increase* in performance going from two layers to four is much less pronounced that when going from one to two. Secondly, the maximum error began to rise again after iteration 12 or so. Together, these observations suggest that increasing the depth of the network has diminishing returns. Clearly the next scenario should try something different. Now that the network is deeper, what would happen if it were given even more data to work with?

6.5 Scenario Epsilon

The fifth experimental scenario, Epsilon, made even more features available to the model. In all other respects it was the same as Scenario Gamma. The feature vector selected was:

<Δaskhi, Δbidlo, ret, Δvol, Δask, Δbid, retx>

The objective function and the maximum error were graphed across the standard 15 epochs:

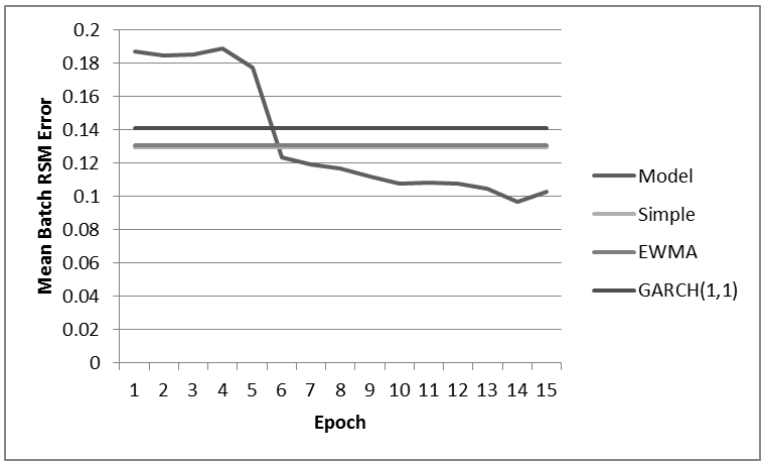


Figure 16: [Scenario Epsilon] RMS Error By Epoch

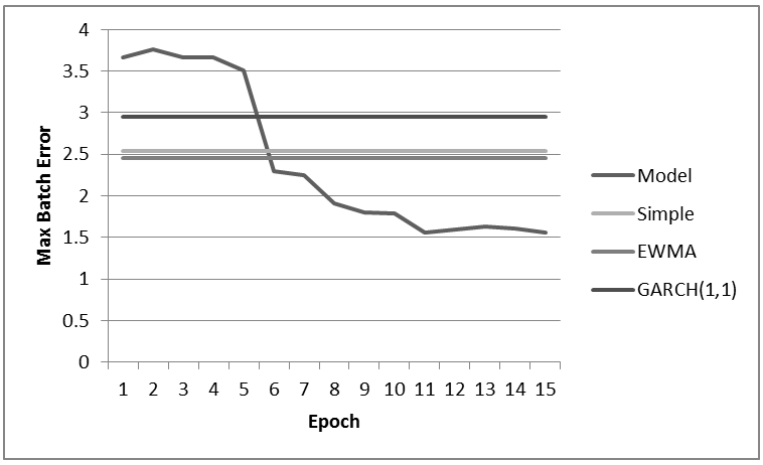


Figure 17: [Scenario Epsilon] Max Error By Epoch

In this scenario, both the objective function and the maximum error generated very different profiles than those of previous scenarios. It appears as if the model was not learning very quickly until around epochs 4 and 5, when it suddenly converged. After that, the learning rate was quite moderate until a minimum at epoch 14 after which it appears to have ticked upwards. It is worth noting that the ultimate performance level fell far short of than reached by Scenario Delta.

What does the shape of the graph mean? When Scenario Beta added extra features and performance dropped off, the solution found in Scenario Gamma was to make the network deeper. Perhaps an even deeper neural network is the answer?

6.6 Scenario Zeta

The sixth experimental scenario, Zeta, originally doubled the depth of the LSTM stack again, from four to eight. However this caused the GPU hardware being used to run out of resources. The depth was therefore cut back to 6 layers. In all other respects this scenario was the same as Scenario Epsilon.

The objective function was graphed across the standard 15 epochs:

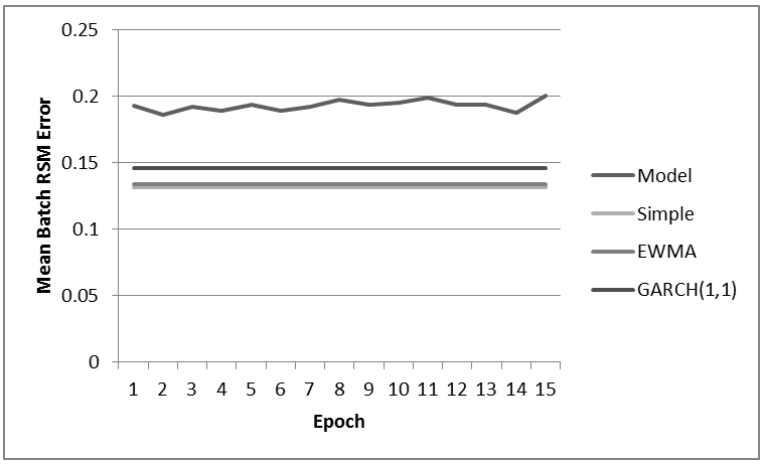


Figure 18: [Scenario Zeta] RMS Error By Epoch

Scenario Zeta was clearly a failure. The model did not converge and consistently produced worse forecasts than the benchmarks. This is clear evidence that simply making a neural network deeper does not improve performance. Indeed, it can have a profound negative effect. It seems that the next experimental direction should backtrack and try varying a different hyperparameter. What about batchsize?

6.7 Scenario Eta

The seventh experimental scenario, Eta, set the number of layers back to 4 and halved the input batch size, from 20 to 10. In all other respects it was the same as Scenario Zeta.

The objective function was graphed across the standard 15 epochs:

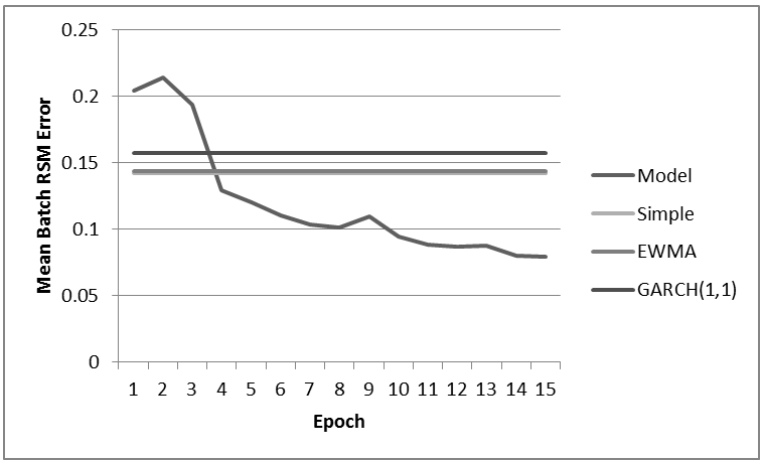


Figure 19: [Scenario Eta] RMS Error By Epoch

The model is not only back in business but it has improved compared to Scenario Epsilon. In the first couple of epochs it seemed to be having trouble converging, but quickly recovered. This is potentially a truncated version of the longer initial delayed converging observed in Scenario Epsilon. Since this phenomenon has been seen in multiple scenarios, it is helpful to give it a name. From here on, a delayed initial convergence shall be referred to as the "early wobbles".

Recall that the model batches inputs in order to speed up the stochastic gradient descent optimiser. However, *reducing* the batch size *ceteris paribus* appears to have improved convergence over the same number of epochs. What happens if the batch size is increased instead?

6.8 Scenario Theta

The eighth experimental scenario, Theta, quadrupled the input batch size, from 10 to 40. In all other respects it was the same as Scenario Eta.

The objective function was graphed across the standard 15 epochs:

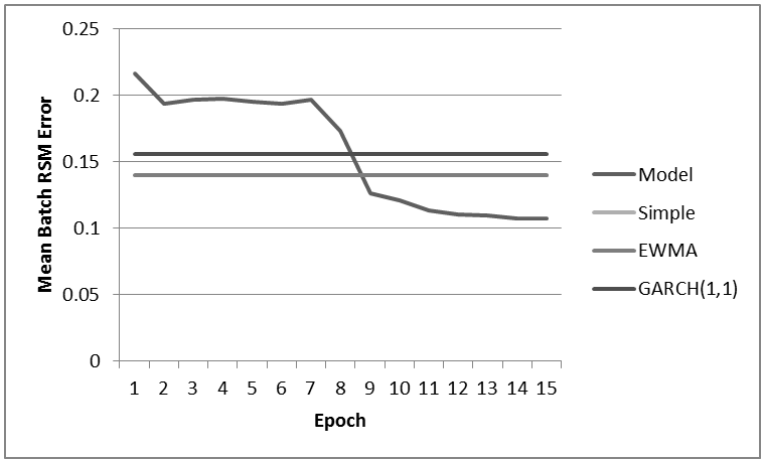


Figure 20: [Scenario Theta] RMS Error By Epoch

This scenario saw a strong recurrence of the early wobbles, and the ultimate performance of the model was clearly worse than in Scenario Eta. This was completely contrary to expectations. A larger batch size is expected to benefit convergence rates, but raising it has the opposite effect. This is a useful empirical result.

What other hyperparameters may be varied to test the original assumptions?

6.9 Scenario Iota

The ninth experimental scenario, Iota, reduced the input batch size back to 20 and increased number of hidden layers by around 15%. In all other respects it was the same as Scenario Theta.

The original version of this scenario specified a doubling of the hidden layers. However, that model exhausted the hardware resources available to the GPU. Therefore, it was scaled back until it fit hardware.

The objective function was graphed across the standard 15 epochs:

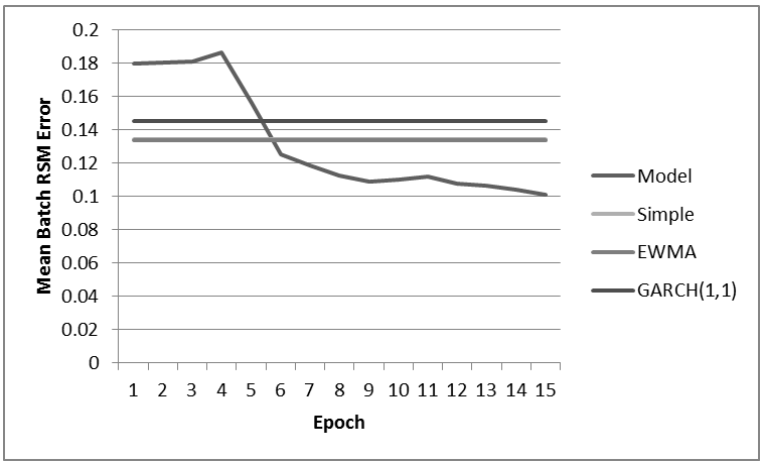


Figure 21: [Scenario Iota] RMS Error By Epoch

The best comparison for these results is perhaps Scenario Epsilon. The output function exhibited analogous early wobbles and ultimately ended up in the same performance ballpark. However, it did not exhibit a similar uptick in the objective function in epoch 15. To the contrary, it seemed to still be converging at a moderate rate when the run terminated.

It appears, therefore that adding additional hidden nodes may be of benefit in this case. More hidden nodes essentially allow the neural network to synthesise more features and derive more relationships in any given layer. However, it turned out in practice that the ability to ramp up the hidden layers was constrained by GPU hardware resources. This is an area where more resources will become available to commodity hardware over time.

Having exhausted the explicitly set model hyperparameters, perhaps it is appropriate to revisit the input data again?

6.10 Scenario Kappa

The tenth experimental scenario, Kappa, returned to the settings of Scenario Epsilon and added the Google Trends feature. Since that was a while back, it is worth recapping all the parameters:

- portfolio: the SP&100
- time frame: 2005-10-01 through 2015-09-30, inclusive
- a 40 day sliding window with a stride of 1 day
- using a past of 30 days to forecast 10 days ahead
- a batch size of 20
- the number of hidden nodes equal to the number of model inputs
- a total of 4 layers
- a feature vector consisting of
  <Δaskhi, Δbidlo, ret, Δvol, Δask, Δbid, retx, trend>

The objective function was graphed across the standard 15 epochs:

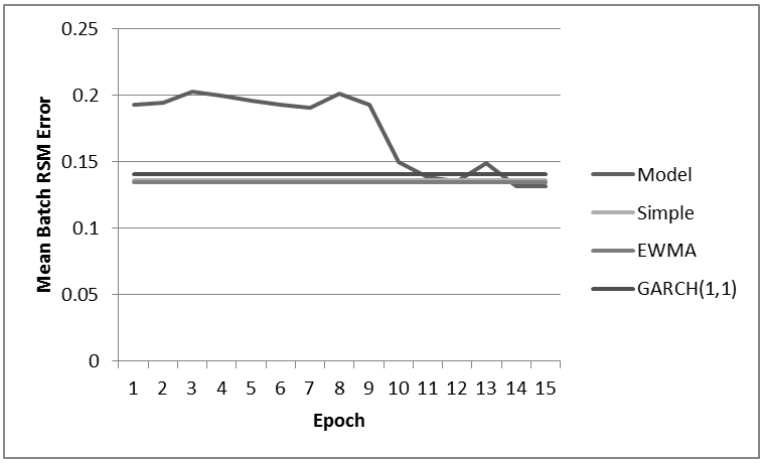


Figure 22: [Scenario Kappa] RMS Error By Epoch

This was a very interesting result. Scenario Epsilon was a robust model using 7 features and yielding good performance (albeit with some early wobbles). Adding just one more feature has essentially reduced the performance back to that of the benchmarks. Furthermore, it looks like convergence had pretty much levelled off by epoch 15.

The two obvious possible causes for this outcome are (1) that the extra feature pushed the model over a performance cliff, or (2) that the Google

Trends data is essentially noise. Starting with the latter hypothesis, the model should be able to learn to deal with random noise without this magnitude effect on performance. Indeed, all the other data it is being fed is noisy since it comes from market observations, and convergence is demonstrably not a problem under those conditions.

But perhaps the Google Trends data is not random noise? If it is strongly correlated to future volatility at some times, and strongly anti-correlated at other times, the model may be having trouble integrating these observations. Because the model views the world in 40 day chunks, independent of any external factors, it cannot necessarily understand global variables that may be driving Google Trends. Form the point of view of the model, Google Trends appears to be a Hidden Markov Model with at least one unknown and hidden variable. If this hypothesis holds, then there exists a class of feature which should be actively excluded from the model.

However, the author's intuition is that the former hypothesis was true. The extra feature has overloaded the model's capability to converge. It may be that the Google Trends data is especially noisy in some way, exacerbating the situation. Nevertheless, the hypothesis is that the problem is with the model and not the trends data *per se*.

What if the model's input data is perturbed in other ways?

6.11 Scenario Mu

The eleventh experimental scenario, Mu, returned to the settings of Scenario Epsilon and switched the S&P500 portfolio for the NASDAQ 100. This portfolio contained 106 stocks as of September 30th 2015.

The objective function was graphed across the standard 15 epochs:

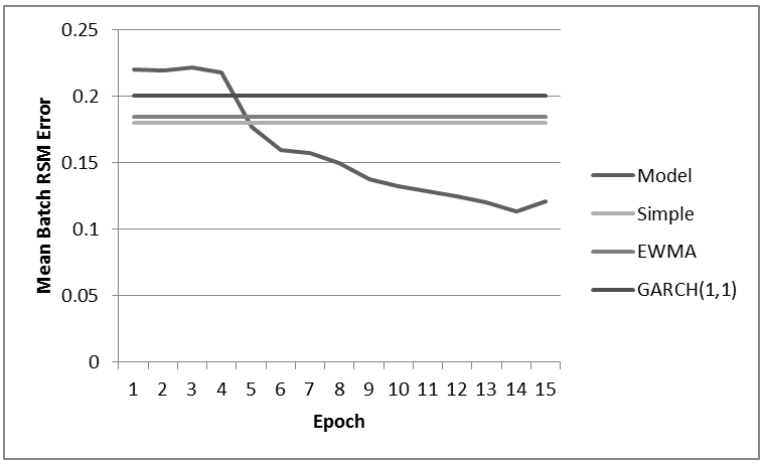


Figure 23: [Scenario Mu] RMS Error By Epoch

This scenario yielded comparable results to Scenario epsilon. Note that the early wobble were still present, which means they are an artefact of the model rather than the input data (although the *size* and *features* of the input data might still have bearing). More importantly, this scenario shows that the model is robust to a change in the input portfolio. The results are not idiosyncratic to a particular set of stocks. This creates confidence that the model is stable for any moderately large set of major issues.

How is stability affected if the model is presented with a much larger portfolio?

6.12 Scenario Nu

The twelfth experimental scenario, Nu, replaced the NASDAQ100 portfolio with the S&P500 portfolio, consisting of 504 stocks as of September 30th 2015. In all other respects this scenario was the same as Scenario Epsilon.

Unfortunately, this scenario made the GPU processor (figuratively) explode. There were nowhere near enough resources available to deploy the model. In retrospect this was not surprising. Scenario Iota had struck resource limits when deepening the network and experimental testing with variants of Scenario Theta had shown that much more than 40 hidden nodes could also trigger resource exhaustion. So quintupling the amount of input was unlikely to be successful.

This scenario is therefore left as an open question. How does the model perform when given even broader market information? Are there signals in

stocks outside the S&P100 that may contribute to individual stock volatility forecasts? There are fundamental issues lurking behind those questions. Is volatility primarily a market phenomenon or does it devolve to individual stocks? If is a market phenomenon then the S&P100 volatility alone might suffice to predict that of other issues. On the other hand, perhaps the trading of stocks outside the S&P100 exerts unexpected influence on the largest companies' returns?

If a large data set is intractable given the available computational resources, how does the model perform on a much smaller set?

6.13 Scenario Xi

The thirteenth experimental scenario, Xi, returned to the S&P100 portfolio, but drew on a much shorter historical period. More specifically data was drawn from October 1st 2014 to September 30th 2015, a period of exactly one year. In all other respects this scenario was the same as Scenario Nu.

The objective function was graphed across the standard 15 epochs:

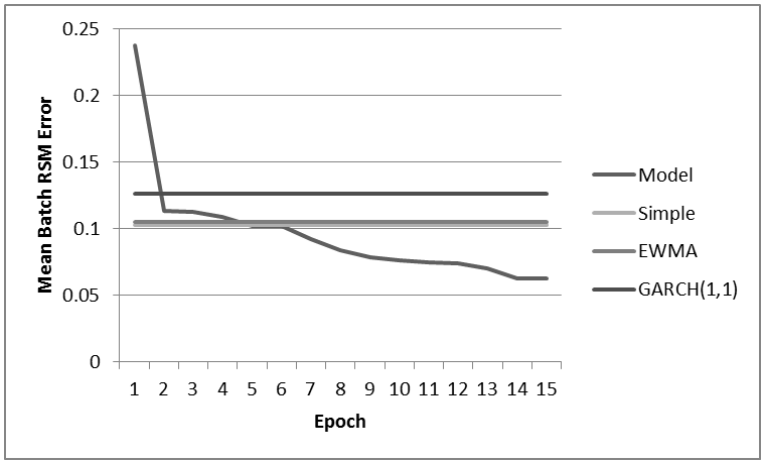


Figure 24: [Scenario Xi] RMS Error By Epoch

At first inspection, this result appears quite strong. The objective function achieved an all-time low. However, note that the benchmark performance is also at an all-time low. It appears that stock volatility over the period of one year is somewhat easier to predict than across a decade. Given that the

longer period incorporates more of the business cycle plus a major crisis, this is hardly surprising.

Nevertheless this scenario shows that the model can outperform the benchmarks on time scale as short as a year. The risk with relying on a model trained with a short horizon is that when a major shift occurs in the market (as inevitably happens) the model will be taken into entirely new territory in which it has no experience. One would expect the performance of the model during such a phase shift to be extremely poor. In essence it would be trying to apply the old rules to a new game.

This scenario suggests that in times of stable markets, a model trained on a shorter historical corpus may be able to maximise forecasting performance. However it would be wise to only do relatively short term forecasting with such a device so that one could react to major market changes.

This begs the question: how does the model perform with longer range predictions?

6.14 Scenario Omnicron

The fourteenth experimental scenario, Omnicron, returned to using the decade of historical data from October 1st 2005 through September 30th 2015, inclusively. However instead of using a 40 day window for extracting training patterns, an 80 day window was used instead with 40 days of “past” data and 20 day forecasts. In all other respects this scenario was the same as Scenario Xi.

The objective function was graphed across the standard 15 epochs:

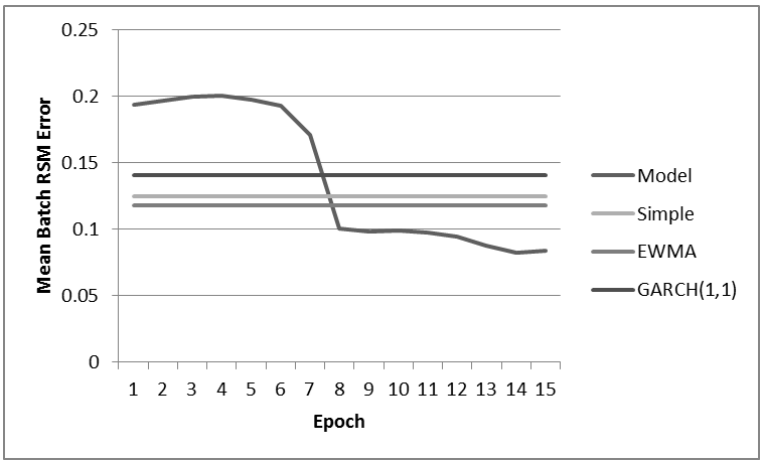


Figure 25: [Scenario Omnicron] RMS Error By Epoch

This scenario produced comparable, or even better, performance to Scenario Epsilon, but again with the characteristic early wobbles. This is an important result because the prediction period was doubled vis-a-vis earlier scenarios. The ability to make longer range forecasts is an attractive property as it increases the applicability of the model.

Of course, using a larger window means that each training pattern is larger, requiring concomitantly more computation resources. More simply, larger windows increase the training time. However, the stride of the window could be adjusted to account for that effect. How does the model deal with sparser input data?

6.15 Scenario Pi

The fifteenth and last experimental scenario, Pi, returned to using 40 data window for generating training patterns. However, instead of a window stride of 1 trading day the window moves 5 trading days at each step. In all other respects this scenario was the same as Scenario Xi.

The objective function was graphed across the standard 15 epochs:

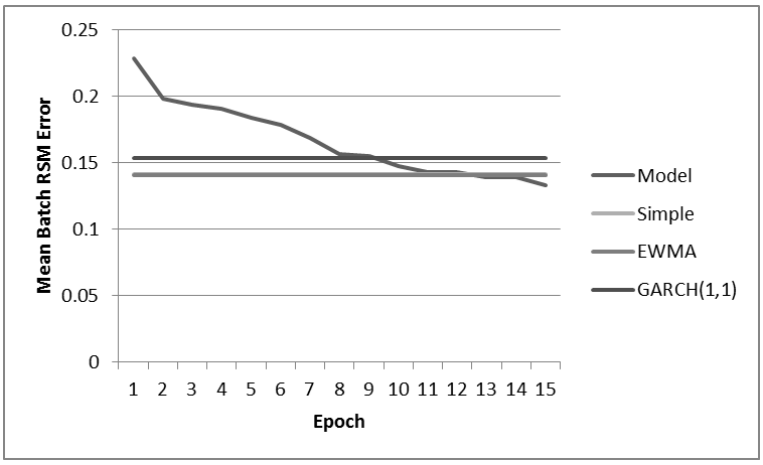


Figure 26: [Scenario Pi] RMS Error By Epoch

This scenario produced substantially worse performance than the comparable Scenario Epsilon. Interestingly, the early wobbles were absent, however the final performance achieved was only slight better than the benchmarks. What is going on here?

By adopting a stride of 5 trading days, the model can train on only 20% of patterns across the portfolio history. The consequential drop in observed performance of the objective function suggests that the devil resides in the fine detail. A sparse view of history does not appear to be sufficient to forecast volatility significantly better than the benchmark methods.

In retrospect, this may be why Scenario Kappa encountered problems with the Google Trend data set. That data, too, is provided at a weekly resolution. It may be that effective volatility forecasting intrinsically requires fine grained observations around the frequency of a trading day.

6.16 Execution Time

The elapsed wall clock time for running selected scenarios was captured, as shown the following table. Note that this measurement captures some overhead, including the marshalling of data, the generation of training patterns and the calculation of benchmarks. Nevertheless, the majority of the execution time was consumed by the forecasting model.

Scenario	Wall Time (seconds)
alpha	389

beta	416
gamma	527
delta	709
epsilon	844
zeta	1138
eta	1286
theta	756
iota	983
kappa	992
mu	1139
xi	199

Table 1: Execution Time of Selected Scenarios

There is a significant variation in the execution time depending on the chosen hyperparameters, both those explicitly set and those determined by the shape of the input data.

The longest two scenarios were Eta which featured reduced a reduced batch size, and Zeta which deepened the neural net to 6 layers. Cleary these hyperparameters are particular expensive in terms of effect on run time. It was also observed that it was easy to exceed the hardware constraints by adjusting these parameters.

The best performing scenario was Delta, which was relatively average in terms of run time. It is worth taking a moment to reflect that in less than 12 minutes, the model was able to train 15 epochs of a decade of data across the S&P100 portfolio. When working with data *en masse*, one can easily lose sight of the actual amount of information being manipulated and the raw computational capability that is now readily accessible. Tools like the model proposed in this report, and the software and hardware that it is built on, enable the analysis and perhaps the understanding of financial information in ways that was previously simply unthinkable.

7 Analysis

Scenario Alpha showed that a LSTM based, deep learning, recurrent neural network could approach the performance of the benchmark methods of forecasting volatility with just one layer and sensible default hyperparameters. While this is a very expensive way of replicating much cheaper calculations, it demonstrated that the model did not require extensive tuning or extreme input normalisation to achieve respectable performance.

Scenario Beta showed that just throwing more data at the model without sizing it correctly was counterproductive. But Scenarios Delta and Gamma demonstrated that deepening the network had a profound influence, with performance rapidly outstripping the benchmark by a healthy margin. Not only was RMS error lower, but the maximum error as well.

Scenario Delta, in retrospect was the best performing of the experimental sequence. It marked the sweet spot of network depth and optimal input features. It is notable that it only took four iterations to get to this place. This suggests that the model design is relatively insensitive to parameter choices so long as they are reasonably sensible. This is a significant departure from Hamid and Iqbal's (2004) observation that configuring a neural net to forecast volatility is "more of an art than science". In fact the model presented in this project is quite the opposite; it proved malleable to configuration with only a small number of iterations required.

Scenario Delta performed so well, in fact, that it is a significant result in its own right. The model in that configuration exhibited an RMS error and maximum *less than 50%* of the lowest benchmark. Keep in mind that this was over a decade's worth of data, across the S&P100, so it was not a cherry picked outcome.

Scenario Epsilon repeated the lesson of Beta and again demonstrated that more data is not always a good thing for an LSTM network. Encouragingly the model still performed significantly better than the benchmarks, showing that it was quite robust to errors in feature selection. This recalls Dr Tarui's advice of sticking to the raw data and letting the deep network sort the signal from the noise.

Scenario Zeta was a huge surprise, in that making the network even deeper caused a complete collapse in performance. When the scenario was specified, the author expected perhaps an increase in performance, or perhaps a decrease, but not such a massive negative effect. Clearly there is a limit beyond which more layers are unhelpful. This is actually encouraging because very deep networks can require massive computational resources. If optimal performance is achievable with moderate layering, optimisation efforts can focus on areas other than raw computing horsepower.

Scenarios Eta and Theta demonstrated a contra-expectation response in performance to varying the batch size. There are subtle factors at interplay here and the author does not yet understand them all. This is clearly an area for further investigation. Nevertheless, it is worth noting that performance in both cases remained better than the benchmarks. The model demonstrated its resiliency yet again.

Scenario Iota proved very little except that the model could easily exhaust GPU resources if configured in certain directions (a lesson reinforced by Scenario Nu). If there is anything to be learned here it is that there is more potential to improve performance and to operate on larger data sets as commodity GPU hardware increases in capability. The rate of improvement of GPU hardware in recent years has been on an exponential “Moore’s Law” curve, unlike mainstream CPUs which are improving much more slowly.

Scenario Kappa demonstrated that added “social” data like Google Trends information was unhelpful. In fact it made the model perform much more poorly, which was somewhat of a surprise because noise should be easy to learn to filter. There are several ways to interpret this result. One is that the data is bad. Perhaps, because the data was of week-long resolution, it was often late to the party and pointing in the wrong direction? Another interpretation is that the markets are significantly more efficient and less noisy than social media. Maybe Google search data is the wrong place to look for a volatility signal and places like Facebook or Twitter are more appropriate? It is true that this study used Google Data because it is freely

available, compared to other social networking platforms. It remains an open question as to whether crowd sourced data can help predict stock volatility.

Scenario Mu validated that the model could work just as well on a different large portfolio. Omnicron showed that the forecast window could be successfully extended. Xi showed that much less historical data could be used to generate good forecasts. In all three cases, performance was consistently better than the benchmarks. Again, the model demonstrated good stability versus input changes.

Lastly Scenario Pi showed that ignoring 80% of the patterns in the input data is a bad idea. Remarkably, this scenario still beat the benchmarks but it established an anti-pattern for configuration.

Overall the model performed better than the benchmark forecasts in 11 of the 14 experimental scenarios that ran to completion.

8 Conclusions and Future Directions

The research goal was to construct a deep neural network that would outperform benchmark volatility forecasting methods. This report presents evidence that this goal has been met. In the highest performing configurations, the model demonstrated an RMS error rate less than 50% of the next best performing benchmark. In addition, the model demonstrated robustness to hyperparameter and input choices, making it more stable than previous generations of neural network solutions.

The driving hypothesis was that recent advances in machine learning have made it possible to solve problems that were previously intractable. The forecasting model presented drew heavily on new technology, including:

- The Adam optimization algorithm
- The TensorFlow system
- Advances in commodity GPU hardware

Without all three of these advances, a model with the accuracy and execution speed demonstrated would have been difficult or impossible.

It appears that the machine learning research community is on the cusp of a renaissance in technique and capability. In the area this project has chosen to focus on, better volatility forecasts over the short term could create information asymmetry for some market participants, creating arbitrage opportunities to profit from. A typical trade would be to find mispriced options using Black-Scholes with the superior forecast. In the long term better volatility forecasts could lower the cost of hedging and deliver better returns to many investors.

Volatility forecasting is but one of many important forecasting problems in finance and econometrics. The positive results of this research suggest that advances in deep learning may have made some of these other problems more tractable as well.

There are several areas which require further investigation:

- Is the model's optimiser getting stuck in local minima? Several of the scenarios exhibited an uptick at the end of the training sequence. Had the global minimum been hit or was there more potential to unlock?
- A key parameter of the optimiser is the learning rate. The presented model left it at its default setting. Could changing it improve convergence?
- Perhaps an alternate optimising algorithm could perform better? There are many to choose from.
- More research is needed in the area of batch sizes. Why does converging improve with smaller batches? Does performance respond to batch size monotonically or is there a more complicated relationship?
- More research is needed into incorporating social media features into the model. Are there any out there that can help predict volatility better than market data?
- Experiments should be run with some of the data that was explicitly ignored in this study. For instance, with a NASDAQ100 portfolio, does a NASDAQ number-of-trades feature improve forecasting performance?

- Implied volatility was not used as a benchmark in this study due to time and data availability constraints. How does the model compare against this benchmark?
- Larger hardware should be sourced to run some of the scenarios that failed or had to be downsized due to hardware constraints. How does the model perform against the full S&P500?

9 References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M. & Ghemawat, S. (2015), *TensorFlow: Large-scale machine learning on heterogeneous systems*. [Online] Available at http://download.tensorflow.org/paper/whitepaper2015.pdf [Accessed 2nd February 2016].

Atsalakis, G. S., & Valavanis, K. P. (2009), *Surveying stock market forecasting techniques–Part II: Soft computing methods.* Expert Systems with Applications, 36(3), 5932-5941.

Bottou, L. (2010), *Large-scale machine learning with stochastic gradient descent.* In Proceedings of COMPSTAT'2010 (pp. 177-186). Physica-Verlag HD.

Brooks, C. (2008), *Introductory econometrics for finance*. Cambridge university press.

Buffett, W. (2014). *Letter to Sharholders of Berkshire Hathaway Inc.* [Online] Available at http://www.berkshirehathaway.com/letters/2014ltr.pdf [Accessed 15th April 2016].

Chicago Board Option Exchange (2015), *Equity Options Product Specifications*. [Online] Available at http://www.cboe.com/products/equityoptionspecs.aspx [Accessed 20th February 2016].

Chicago Board Option Exchange (2016), *Volatility Indexes at CBOE*. [Online] Available at http://www.cboe.com/micro/vix/pdf/cboe30c7-volindex_qrg.pdf [Accessed 22th February 2016].

comp.ai.neural-nets (2014), *AI FAQ/Neural Nets.* [Online] http://www.faqs.org/faqs/ai-faq/neural-nets/ [Accessed 20th February 2016].

Cotter, A., Shamir, O., Srebro, N. and Sridharan, K., (2011), *Better mini-batch algorithms via accelerated gradient methods.* In Advances in neural information processing systems (pp. 1647-1655).

CRSP (2015), *CRSP US Stock Database*. Center for Research in Security Prices (CRSP), The University of Chicago Booth School of Business. Retrieved from Wharton Research Data Service.

Cuthbertson K. & Nitzsche, L. (2001), *Financial Engineering*. Wiley, England.

Cuthbertson K. & Nitzsche, L. (2008), *Investments*. Wiley, England.

Damien, A. (2016), *TensorFlow Examples.* [Online] https://github.com/aymericdamien/TensorFlow-Examples [Accessed 10th February 2016].

González-Rivera, G., Lee, T.H. and Yoldas, E., (2007), *Optimality of the RiskMetrics VaR model.* Finance Research Letters, 4(3), pp.137-145.

Google (2015), Google Trends Tool, [Online] Available at https://www.google.com/trends/ [Accessed 15th December 2015].

Gupta, S., Agrawal, A., Gopalakrishnan, K. and Narayanan, P., (2015), *Deep learning with limited numerical precision.* arXiv preprint arXiv:1502.02551.

Hamid, S. A., & Iqbal, Z. (2004), *Using neural networks for forecasting volatility of S&P 500 Index futures prices.* Journal of Business Research, 57(10), 1116-1125.

Hecht-Nielsen, R. (1990), *Neurocomputing.* Addision Wesley. Reading, Mass.

Heston, S.L. (1993), *A closed-form solution for options with stochastic volatility with applications to bond and currency options*. Review of financial studies, 6(2), pp.327-343.

IEEE Standards Committee. (2008*), 754-2008 IEEE standard for floating-point arithmetic.* IEEE Computer Society Std.

Karpathy, A. (2015). *The unreasonable effectiveness of recurrent neural networks.* [Online] Available at http://karpathy.github.io/2015/05/21/rnn-effectiveness/ [Accessed 28th December 2015]

Kingma, D. and Ba, J. (2014), *Adam: A method for stochastic optimization.* arXiv preprint arXiv:1412.6980.

Kirchner, U. (2010), *A Subjective and Probabilistic Approach to Derivatives*. arXiv preprint arXiv:1001.1616.

Krollner, B., Vanstone, B., & Finnie, G. (2010), *Financial time series forecasting with machine learning techniques: A survey.* [Online] Available at http://epublications.bond.edu.au/cgi/viewcontent.cgi?article=1113&context=infotech_pubs [Accessed 10 October 2015].

Markowitz, H. (1952). *Portfolio selection*. The journal of finance, 7(1), pp.77-91.

Li, M., Zhang, T., Chen, Y. and Smola, A.J., (2014), *Efficient mini-batch training for stochastic optimization.* Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 661-670). ACM.

Olah, C.(2015), *Understanding LSTM Networks.* [Online] Available at http://colah.github.io/posts/2015-08-Understanding-LSTMs/ [Accessed 17th April 2016]

Onwukwe, C.E., Bassey, B.E.E. and Isaac, I.O. (2011). *On modeling the volatility of Nigerian stock returns using GARCH models.* Journal of Mathematics Research, 3(4), p.31.

Poon, S.H. and Granger, C.W. (2003). *Forecasting volatility in financial markets: A review*. Journal of economic literature, 41(2), pp.478-539.

Rao, D. (2015), *Google TensorFlow: Updates & Lessons.* [Online] Available at http://deliprao.com/archives/98 [Accessed 7th March 2016]

Ruder, S. (2016), *An overview of gradient descent optimization algorithms.* [Online] Available at http://sebastianruder.com/optimizing-gradient-descent/ [Accessed 2nd February, 2016].

Sheppard, K. (2015), *ARCH for Python.* [Online] Available at https://pypi.python.org/pypi/arch/3.1 [Accessed 12th February, 2016]

Siblis Research (2016), *Search Equity Valuation Data*. [Online] http://siblisresearch.com/ [Accessed 9th March 2016]

TensorFlow (2016), *TensorFlow Open Source Software Library for Machine Intelligence*. [Online] Available at https://www.tensorflow.org/ [Accessed 4th February 2016]

Malkiel, B.G. and Xu, Y. (1997), *Risk and return revisited*. The Journal of Portfolio Management, 23(3), pp.9-14.

Zaremba, W., Sutskever, I. and Vinyals, O. (2014), *Recurrent neural network regularization.* arXiv preprint arXiv:1409.2329.

10 Bibliography

Andersen, T. G., & Bollerslev, T.. (1998), *Answering the Skeptics: Yes, Standard Volatility Models do Provide Accurate Forecasts.* International Economic Review, 39(4), 885–905.

Goldstein, D. G. & Taleb, N. N. (2007), *We Don't Quite Know What We are Talking About When We Talk About Volatility*. Journal of Portfolio Management, Vol. 33, No. 4, 2007.

Patton, A.J. (2011), *Volatility forecast comparison using imperfect volatility proxies.* Journal of Econometrics, 160(1), pp.246-256.

Pérez-Cruz, F., Afonso-Rodriguez, J. A., & Giner, J. (2003), *Estimating GARCH models using support vector machines**. Quantitative Finance, 3(3), 163-172.

Rosenblatt, F. (1958). *The perceptron: A probabilistic model for information storage and organization in the brain.* Psychological Review, Vol 65(6), Nov 1958, 386-408

Appendix A: Hardware and Software

- The GPU card used for running the final model scenarios was an Nvidia Quadro K2200, providing 640 cores clocked at 1000MHz and 4GB of memory.
- The model was built with TensorFlow version 0.8
- The complete model source code is available online at https://github.com/MichaelPaddon/volatility