Machine Learning for Time Series Prediction, Comparative analysis

Olu Salami

A Thesis in the Field of Computational Finance

for the Degree of Master of Science in Computational Finance

University of Greenwich

January 2019

Abstract

Autonomous and Automated decision making in the area of trading securities

Keywords: Machine learning; Convolutional Neural Networks; Recurrent Neural Networks; LTSM; Reinforcement learning; Machine Learning; Forex; Algorithmic Trading; Portfolio Management; Quantitative Finance; Q-learning; Markov Decision Process;

Dedication (optional)

To delete this or any other unwanted section, select it in its entirety, including the title and the Section Break, and press Backspace or Delete.

Acknowledgments (optional)

To delete this or any other unwanted section, select it in its entirety, including the title and the Section Break, and press Backspace or Delete. To see the Section Break, show formatting symbols by clicking the ¶ button in the Paragraph section

Table of Contents

Machine Learning for Time Series Prediction, Comparative analysis 1

Chapter 1 1

Dedication (optional) 5

Acknowledgments (optional) 6

Table of Contents 7

List of Tables (optional) 11

List of Figures (optional) 12

2. Chapter I. Introduction 13

Introduction 14

Problem Background 14

Problem Statement 14

Purpose of Study 14

Project Objectives 14

Scope of Study 14

The Significant of Study 14

Organization of Report 14

3. Chapter II. Literature Review 15

Introduction 15

Phishing 15

Existing Anti-Phishing Approaches 15

Summary 15

4. Chapter III. Research Methodology 16

Introduction 16

Stock prediction 16

Efficient market theory 17

Predictability of stock 17

Time series forecasting 17

Arima 18

Autoregressive Integrated Moving Average Model 18

Var 24

Vector Auto Regression 24

Machine Learning 28

Neural Network 29

CNN 29

RNN 29

GRU 29

Attention based networks 29

LSTM 29

Inference from review 30

Inference from research 30

Summary 50

5. Chapter IV. Classical Methods 51

Exploratory Data Analysis 51

Auto Regressive Integrated Moving Average 55

6. Chapter IV. LSTM 56

Mathematical Formulation 56

Experiment 56

Dataset Division 56

Building The Model 56

Training Model 56

Validation Technique 56

Tuning The Model 56

Summary 56

7. Chapter V. Comparative Analysis 57

Introduction 57

ARIMA 57

VAR 57

LTSM 57

Experimental Data 57

Training and Validation Dataset 57

Building the model 58

Epoch size 61

Model training Vs Validation Loss 64

Batch size 66

Number of neurons 67

Summary 70

8. Conclusion. 71

Concluding Remarks 71

Key Points 72

Recommendations for Future Research 72

Dropout, Optimization Algorithm, Loss Function. 73

LSTM in deep reinforcement learning for time series prediction 73

9. [References] 74

Bibliography 74

List of Tables (optional)

Table 1. Table Title. 2

Table 2. Table Title 2

List of Figures (optional)

Figure 1. Figure Title. 2

Figure 2. Figure Title 2

2. Chapter I.

Introduction

There are two ways to add a new chapter. First, you can simply type the name of

Research question,

1) How well does LTSM perform with relatively limited dataset (illiquid asset) in a machine learning context. The commodity Gold was selected for its low volatility.

2) In contract too existing forecasting model, how well does LSTM perform. My hypothesis is that LSTM will outperform existing (non ML) models

The question of using limited dataset is significant as this is sometimes considered ineffective machine learning domain. Where experiments are often performed with millions of data records

Introduction

Problem Background

Problem Statement

Purpose of Study

Project Objectives

Scope of Study

Background

The Significant of Study

Organization of Report

.

3. Chapter II.

Literature Review

Introduction

Phishing

Existing Anti-Phishing Approaches

Summary

4. Chapter III.

Research Methodology

Introduction

Stock prediction

- What is it,
- whats been done on it,
- how does it relate to what you are doing
- general news about it
-

Asset price prediction is an oft researched topic in finance. The body of work dedicated to predicting asset price movement is extensive. There has been limited success in application of new predictive methods for stock price movements. The challenges of predicting asset price movement lies in the underlying convolution of simulating market dynamics. In Fundamental analysis,. In Technical analysis, it is believed that market timing is key. Technicians utilize charts and modeling techniques to identify trends in price and volume. These later individuals rely on historical data in order to predict future outcomes. One area of limited success in Stock Market prediction

. Companies may want to get their stock listed on a stock exchange. Other stocks may be traded "over the counter", that is, through a dealer. A large company will usually have its stock listed on many exchanges across the world. Exchanges may also cover other types of security such as fixed interest securities or interest derivatives. Vivek Rajput et al, International Journal of Computer Science and Mobile Computing, Vol.5 Issue.6, June-

Participants in the stock market range from small individual stock investors to larger traders investors, who can be based anywhere in the world, and may include banks, insurance companies or pension funds, and hedge funds. Their buy or sell orders may be executed on their behalf by a stock exchange trader. An example of such an exchange is the New York Stock Exchange. The other type of stock exchange is a virtual kind, composed of a network of computers where trades are made electronically by traders. An example of such an

Efficient market theory

Define it,

How does it relate to forecasting

Can the market be tamed it predicted , what been don on this for ref

Predictability of stock

General preamble about predicting stock prices, human, emotions, fundamental economics principe, historical importance of gold as commodity

Whats in literature

Time series forecasting

General info on time series forcasting

History and applications of it

Univariate and multivariate,

Some generica mathemathica

Reference any scholarly work and discus one point

Data Mining approach

General info, define this, explain use in industry

How has it been applied, current research work on it, its relationship to machine learning

Arima

Describe, define, uses, maths, current work, your view

Autoregressive Integrated Moving Average Model

It explicitly caters to a suite of standard structures in time series data, and as such provides a simple yet powerful method for making skillful time series forecasts.

ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average and adds the notion of integration.

This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- **AR**: *Autoregression*. A model that uses the dependent relationship between an observation and some number of lagged observations.

- **I**: *Integrated*. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.
- **MA**: *Moving Average*. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Each of these components are explicitly specified in the model as a parameter. A standard notation is used of ARIMA(p,d,q) where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used.

The parameters of the ARIMA model are defined as follows:

- **p**: The number of lag observations included in the model, also called the lag order.
- **d**: The number of times that the raw observations are differenced, also called the degree of differencing.
- **q**: The size of the moving average window, also called the order of moving average.

A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model.

A value of 0 can be used for a parameter, which indicates to not use that element of the model. This way, the ARIMA model can be configured to perform the function of an ARMA model, and even a simple AR, I, or MA model.

Adopting an ARIMA model for a time series assumes that the underlying process that generated the observations is an ARIMA process. This may seem obvious, but helps to motivate the need to confirm the assumptions of the model in the raw observations and in the residual errors of forecasts from the model.

Next, let’s take a look at how we can use the ARIMA model in Python. We will start with loading a simple univariate time series.

3.1. Preamble Early attempts to study time series, particularly in the 19th century, were generally characterized by the idea of a deterministic world. It was the major contribution of Yule (1927) which launched the notion of stochasticity in time series by postulating that every time series can be regarded as the realization of a stochastic process. Based on this simple idea, a number of time series methods have been developed since then. Workers such as Slutsky, Walker, Yaglom, and Yule first formulated the concept of autoregressive (AR) and moving average (MA) models. Wold’s decomposition theorem led to the formulation and solution of the linear forecasting problem of Kolmogorov (1941). Since then, a considerable body of literature has appeared in the area of time series, dealing with parameter estimation, identification, model checking, and forecasting; see, e.g., Newbold (1983) for an early survey. The publication Time Series Analysis: Forecasting and Control by Box and Jenkins (1970)3 integrated the existing knowledge. Moreover, these authors developed a coherent, versatile three-stage iterative

cycle for time series identification, estimation, and verification (rightly known as the Box–Jenkins approach). The book has had an enormous impact on the theory and practice of modern time series analysis and forecasting. With the advent of the computer, it popularized the use of autoregressive integrated moving average (ARIMA) models and their extensions in many areas of science. Indeed, forecasting discrete time series processes through univariate ARIMA models, transfer function (dynamic regression) models, and multivariate (vector) ARIMA models has generated quite a few IJF papers. Often these studies were of an empirical nature, using one or more benchmark methods/models as a comparison. Without pretending to be complete, Table 1 gives a list of these studies. Naturally, some of these studies are more successful than others. In all cases, the forecasting experiences reported are valuable. They have also been the key to new developments, which may be summarized as follows. 3.2. Univariate The success of the Box–Jenkins methodology is founded on the fact that the various models can, between them, mimic the behaviour of diverse types of series—and do so adequately without usually requiring very many parameters to be estimated in the final choice of the model. However, in the midsixties, the selection of a model was very much a matter of the researcher's judgment; there was no algorithm to specify a model uniquely. Since then, many techniques and methods have been suggested to add mathematical rigour to the search process of an ARMA model, including Akaike's information criterion (AIC), Akaike's final prediction error (FPE), and the Bayes information criterion (BIC). Often these criteria come down to minimizing (in-sample) onestep-ahead forecast errors, with a penalty term for overfitting. FPE has also been generalized for multistep-ahead forecasting (see, e.g., Bhansali, 1996, 1999), but this generalization has not been utilized by applied workers. This also seems to be the case with criteria based on cross-validation and splitsample validation (see, e.g., West, 1996) principles, making use of genuine out-of-sample forecast errors; see Pen˜a and Sa´nchez (2005) for a related approach worth considering. There are a number of methods (cf. Box et al., 1994) for estimating the parameters of an ARMA model. Although these methods are equivalent asymptotically, in the sense that estimates tend to the same normal distribution, there are large differences in finite sample properties. In a comparative study of software packages, Newbold, Agiakloglou, and Miller (1994) showed that this difference can be quite substantial and, as a consequence, may influence forecasts. They recommended the use of full maximum likelihood. The effect of parameter estimation errors on the probability limits of the forecasts was also noticed by Zellner (1971). He used a Bayesian analysis and derived the predictive distribution of future observations by treating the parameters in the ARMA model as random variables. More recently, Kim (2003) considered parameter estimation and forecasting of AR models in small samples. He found that (bootstrap) bias-corrected parameter estimators produce more accurate forecasts than the least squares estimator. Landsman and Damodaran (1989) presented evidence that the James-Stein ARIMA parameter estimator improves forecast accuracy relative to other methods, under an MSE loss criterion. If a time series is known to follow a univariate ARIMA model, forecasts using disaggregated observations are, in terms of MSE, at least as good as forecasts using aggregated observations. However, in practical applications, there are other factors to be considered, such as missing values in disaggregated series. Both Ledolter (1989) and Hotta (1993) analyzed the effect of an additive outlier on the forecast intervals when the ARIMA model parameters are estimated. When the model is stationary, Hotta and Cardoso Neto (1993) showed that the loss of efficiency using aggregated data is not large, even if the model is not known. Thus, prediction could be done by either disaggregated or aggregated models. The problem of incorporating external (prior) information in the univariate ARIMA forecasts has been considered by Cholette (1982), Guerrero (1991), and de Alba (1993). As an alternative to the univariate ARIMA methodology, Parzen (1982) proposed the ARARMA methodology. The key idea is that a time series is transformed from a long-memory AR filter to a shortmemory filter, thus avoiding the bharsherQ differencing operator. In addition, a different approach to the dconventionalT Box–Jenkins identification step is used. In the M-competition (Makridakis et al., 1982), the ARARMA models achieved the lowest MAPE for longer forecast horizons. Hence, it is surprising to find that, apart from the paper by Meade and Smith (1985), the ARARMA methodology has not really taken off in applied work. Its ultimate value may perhaps be better judged by assessing the study by Meade (2000) who compared the forecasting performance of an automated and non-automated ARARMA method. Automatic univariate ARIMA modelling has been shown to produce one-step-ahead forecasts as accurate as those produced by competent modellers (Hill & Fildes, 1984; Libert, 1984; Poulos, Kvanli, & Pavur, 1987; Texter & Ord, 1989). Several software vendors have implemented automated time series forecasting methods (including multivariate methods); see, e.g., Geriner and Ord (1991), Tashman and Leach (1991), and Tashman (2000). Often these methods act as black boxes. The technology of expert systems (Me´lard & Pasteels, 2000) can be used to avoid this problem. Some guidelines on the choice of an automatic forecasting method are provided by Chatfield (1988). Rather than adopting a single AR model for all forecast horizons, Kang (2003) empirically investigated the case of using a multi-step-ahead forecasting AR model selected separately for each horizon. The forecasting performance of the multi-step-ahead procedure appears to depend on, among other things,

Var

Describe, define, uses, maths, current work, your view

Vector Auto Regression

A VAR model is a generalisation of the univariate autoregressive model for forecasting a vector of time series. It comprises one equation per variable in the system. The right hand side of each equation includes a constant and lags of all of the variables in the system. To keep it simple, we will consider a two variable VAR with one lag. We write a 2-dimensional VAR(1) as Eq1.1

$$\begin{aligned} y_{1,t} &= c_1 + \phi_{11,1} y_{1,t-1} + \phi_{12,1} y_{2,t-1} + e_{1,t} \\ y_{2,t} &= c_2 + \phi_{21,1} y_{1,t-1} + \phi_{22,1} y_{2,t-1} + e_{2,t}, \end{aligned}$$

where $e_{1,t}$ and $e_{2,t}$ are white noise processes that may be contemporaneously correlated. The coefficient $\phi_{ii,\ell}$ captures the influence of the ℓth lag of variable y_i on itself, while the coefficient $\phi_{ij,\ell}$ captures the influence of the ℓth lag of variable y_j on y_i.

If the series are stationary, we forecast them by fitting a VAR to the data directly (known as a "VAR in levels"). If the series are non-stationary, we take differences of the data in order to make them stationary, then fit a VAR model (known as a "VAR in differences"). In both cases, the models are estimated equation by equation using the principle of least squares. For each equation, the parameters are estimated by minimising the sum of squared $e_{i,t}$ values.

The other possibility, which is beyond the scope of this book and therefore we do not explore here, is that the series may be non-stationary but cointegrated, which means that there exists a linear combination of them that is stationary. In this case, a VAR specification that includes an error correction mechanism (usually referred to as a vector error correction model) should be included, and alternative estimation methods to least squares estimation should be used.

Forecasts are generated from a VAR in a recursive manner. The VAR generates forecasts for *each* variable included in the system. To illustrate the process, assume that we have fitted the 2-dimensional VAR(1) described in Equations (1.1) –(1.2) for all observations up to time Then the one-step-ahead forecasts are generated by

$$\hat{y}_{1,T+1|T} = \hat{c}_1 + \hat{\phi}_{11,1} y_{1,T} + \hat{\phi}_{12,1} y_{2,T}$$
$$\hat{y}_{2,T+1|T} = \hat{c}_2 + \hat{\phi}_{21,1} y_{1,T} + \hat{\phi}_{22,1} y_{2,T}.$$

except that the errors have been set to zero and parameters have been replaced with their estimates. For $h=2$, the forecasts are given by

$$\hat{y}_{1,T+2|T} = \hat{c}_1 + \hat{\phi}_{11,1}\hat{y}_{1,T+1} + \hat{\phi}_{12,1}\hat{y}_{2,T+1}$$
$$\hat{y}_{2,T+2|T} = \hat{c}_2 + \hat{\phi}_{21,1}\hat{y}_{1,T+1} + \hat{\phi}_{22,1}\hat{y}_{2,T+1}.$$

Again, this is the same form as (1.1)–(1.2), except that the errors have been set to zero, the parameters have been replaced with their estimates, and the unknown values of y1 and y2 have been replaced with their forecasts. The process can be iterated in this manner for all future time periods.

There are two decisions one has to make when using a VAR to forecast, namely how many variables (denoted by K) and how many lags (denoted by p) should be included in the system. The number of coefficients to be estimated in a VAR is equal to K+pK2 (or 1+pK per equation). For example, for a VAR with K=5 variables and p=3 lags, there are 16 coefficients per equation, giving a total of 80 coefficients to be estimated. The more coefficients that need to be estimated, the larger the estimation error entering the forecast.

In practice, it is usual to keep K small and include only variables that are correlated with each other, and therefore useful in forecasting each other. Information criteria are commonly used to select the number of lags to be included.

Vector autoregressions (VARs) constitute a special case of the more general class of VARMA models. In essence, a VAR model is a fairly unrestricted (flexible) approximation to the reduced form of a wide variety of dynamic econometric models. VAR models can be specified in a number of ways. Funke (1990) presented five different

VAR specifications and compared their forecasting performance using monthly industrial production series. Dhrymes and Thomakos (1998) discussed issues regarding the identification of structural VARs. Hafer and Sheehan (1989) showed the effect on VAR forecasts of changes in the model structure. Explicit expressions for VAR forecasts in levels are provided by Arin˜o and Franses (2000); see also Wieringa and Horva´th (2005). Hansson, Jansson, and Lo¨ f (2005) used a dynamic factor model as a starting point to obtain forecasts from parsimoniously parametrized VARs. In general, VAR models tend to suffer from doverfittingT with too many free insignificant parameters. As a result, these models can provide poor outof-sample forecasts, even though within-sample fitting is good; see, e.g., Liu, Gerlow, and Irwin (1994) and Simkins (1995). Instead of restricting some of the parameters in the usual way, Litterman (1986) and others imposed a prior distribution on the parameters, expressing the belief that many economic variables behave like a random walk. BVAR models have been chiefly used for macroeconomic forecasting (Artis & Zhang, 1990; Ashley, 1988; Holden & Broomhead, 1990; Kunst & Neusser, 1986), for forecasting market shares (Ribeiro Ramos, 2003), for labor market forecasting (LeSage & Magura, 1991), for business forecasting (Spencer, 1993), or for local economic forecasting (LeSage, 1989). Kling and Bessler (1985) compared out-of-sample forecasts of several thenknown multivariate time series methods, including Litterman's BVAR model. The Engle and Granger (1987) concept of cointegration has raised various interesting questions regarding the forecasting ability of error correction models (ECMs) over unrestricted VARs and BVARs. Shoesmith (1992), Shoesmith (1995), Tegene and Kuchler (1994), and Wang and Bessler (2004) provided empirical evidence to suggest that ECMs outperform VARs in levels, particularly over longer J.G. De Gooijer, R.J. Hyndman / International Journal of Forecasting 22 (2006) 443–473 449 forecast horizons. Shoesmith (1995), and later Villani (2001), also showed how Litterman's (1986) Bayesian approach can improve forecasting with cointegrated VARs. Reimers (1997) studied the forecasting performance of seasonally cointegrated vector time series processes using an ECM in fourth differences. Poskitt (2003) discussed the specification of cointegrated VARMA systems. Chevillon and Hendry (2005) analyzed the relationship between direct multi-step estimation of stationary and nonstationary VARs and forecast accuracy.

Artificial Intelligence

Describe, define, uses, maths, current work, your view

Machine Learning

Describe, define, uses, maths, current work, your view

Neural Network

Describe, define, uses, maths, current work, your view

CNN

Describe, define, uses, maths, current work, your view

RNN

Describe, define, uses, maths, current work, your view

Go into detail maths

GRU

Describe, define, uses, maths, current work, your view

Attention based networks

Describe, define, uses, maths, current work, your view

LSTM

Describe, define, uses, maths, current work, your view

Go into detail

Extra maths

Vanishing gradient / exploding gradient maths

Gradient descent maths

Regularisers maths

Weights and biases maths

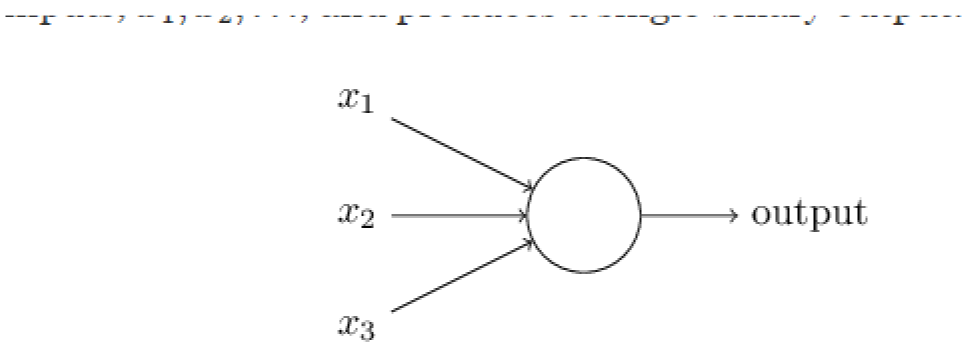
Inference from review

Inference from research

Artificial Neural Networks

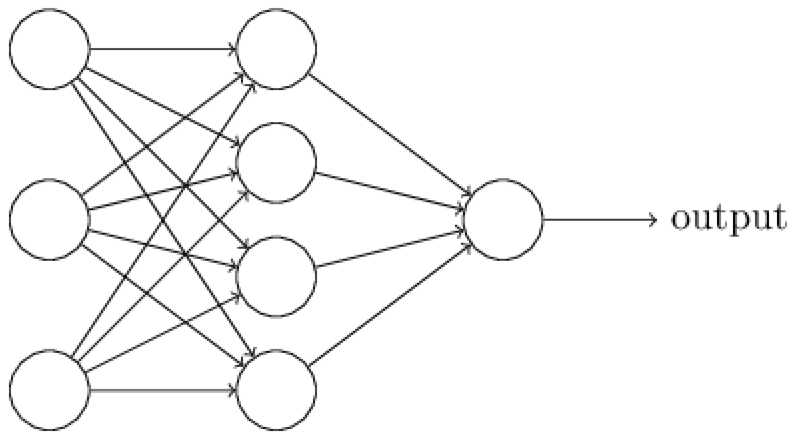
What is a neural network? To get started, I'll explain a type of artificial neuron called a *perceptron*. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neurons - in this book, and in much modern work on neural networks, the main neuron model used is one called the *sigmoid neuron*. We'll get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand perceptrons.

So how do perceptrons work? A perceptron takes several binary inputs, $x_1,x_2,\ldots$x1,x2,…, and produces a single binary output:

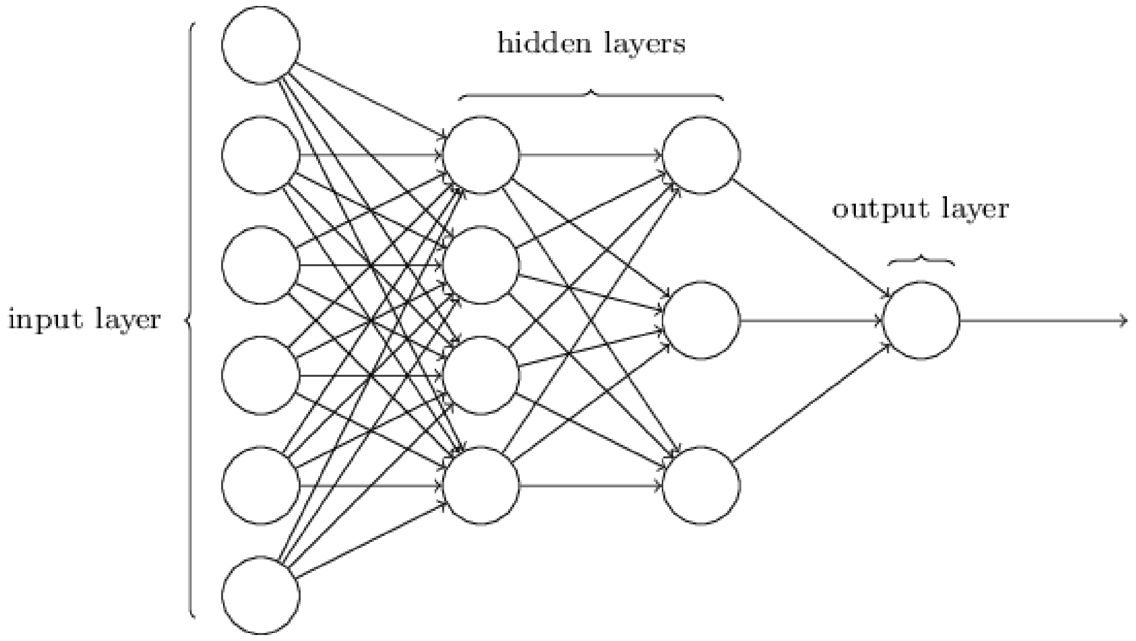


Recurrent Neural Network

In the next section I'll introduce a neural network that can do a pretty good job classifying handwritten digits. In preparation for that, it helps to explain some terminology that lets us name different parts of a network. Suppose we have the network:



As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called *input neurons*. The rightmost or *output* layer contains the *output neurons*, or, as in this case, a single output neuron. The middle layer is called a *hidden layer*, since the neurons in this layer are neither inputs nor outputs. The term "hidden" perhaps sounds a little mysterious - the first time I heard the term I thought it must have some deep philosophical or mathematical significance - but it really means nothing more than "not an input or an output". The network above has just a single hidden layer, but some networks have multiple hidden layers. For example, the following four-layer network has two hidden layers:



Somewhat confusingly, and for historical reasons, such multiple layer networks are sometimes called *multilayer perceptrons* or*MLPs*, despite being made up of sigmoid neurons, not perceptrons. I'm not going to use the MLP terminology in this book, since I think it's confusing, but wanted to warn you of its existence.

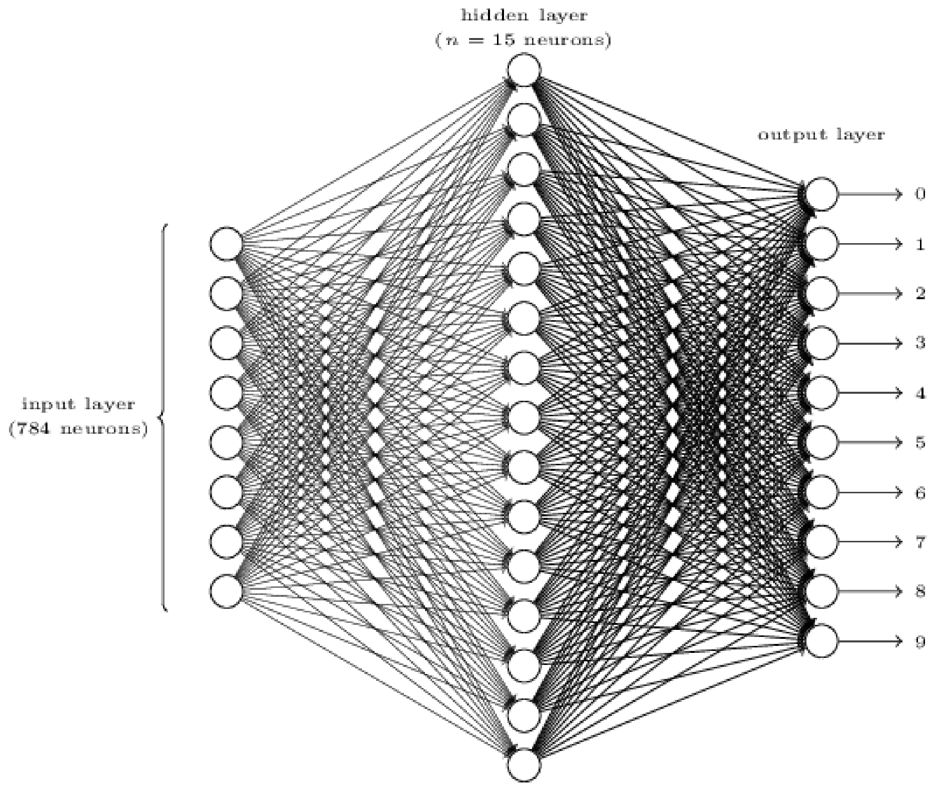
The design of the input and output layers in a network is often straightforward. For example, suppose we're trying to determine whether a handwritten image depicts a "9" or not. A natural way to design the network is to encode the intensities of the image pixels into the input neurons. If the image is
a 6464 by 6464 greyscale image, then we'd
have 4,096=64×644,096=64×64 input neurons, with the intensities scaled appropriately between 00 and 11. The output layer will contain just a single neuron, with output values of less than 0.50.5 indicating "input image is not a 9", and values greater than 0.50.5 indicating "input image is a 9 ".

While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers. In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb. Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets. For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network. We'll meet several such design heuristics later in this book.

Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called *feedforward* neural networks. This means there are no loops in the network - information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σσ function depended on the output. That'd be hard to make sense of, and so we don't allow such loops.

However, there are other models of artificial neural networks in which feedback loops are possible. These models are calledrecurrent neural networks. The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least to date) less powerful. But recurrent networks are still extremely interesting. They're much closer in spirit to how our brains work than feedforward networks. And it's possible that recurrent networks can solve important problems which can only be solved with great difficulty by feedforward networks. However, to limit our scope, in this book we're going to concentrate on the more widely-used feedforward networks.



The input layer of the network contains neurons encoding the values of the input pixels. As discussed in the next section, our training data for the network will consist of

many 2828 by 2828 pixel images of scanned handwritten digits, and so the input layer contains 784=28×28784=28×28 neurons. For simplicity I've omitted most of the 784784 input neurons in the diagram above. The input pixels are greyscale, with a value of 0.00.0 representing white, a value of 1.01.0representing black, and in between values representing gradually darkening shades of grey.

The second layer of the network is a hidden layer. We denote the number of neurons in this hidden layer by nn, and we'll experiment with different values for nn. The example shown illustrates a small hidden layer, containing
just n=15n=15 neurons.

The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output ≈1≈1, then that will indicate that the network thinks the digit is a 00. If the second neuron fires then that will indicate that the network thinks the digit is a 11. And so on. A little more precisely, we number the output neurons from 00 through 99, and figure out which neuron has the highest activation value. If that neuron is, say, neuron number 66, then our network will guess that the input digit was a 66. And so on for the other output neurons.

You might wonder why we use 1010 output neurons. After all, the goal of the network is to tell us which digit (0,1,2,…,90,1,2,…,9) corresponds to the input image. A seemingly natural way of doing that is to use just 44 output neurons, treating each neuron as taking on a binary value, depending on whether the neuron's output is closer to 00 or to 11. Four neurons are enough to encode the answer, since 2^4=1624=16 is more than the 10 possible values for the input digit. Why should our network use 1010 neurons instead? Isn't that inefficient? The ultimate justification is empirical: we can try out both network designs, and it turns out that, for this particular problem, the network with 1010output neurons learns to recognize digits better than the network with 44 output neurons. But that leaves us wondering *why* using 1010output neurons works better. Is there some heuristic that would tell us in advance that we should use the 1010-output encoding instead of the 44-output encoding?

To understand why we do this, it helps to think about what the neural network is doing from first principles. Consider first the case where we use 1010 output neurons. Let's concentrate on the first output neuron, the one that's trying to decide whether or not the digit is a 00. It does this by weighing up evidence from the hidden layer of neurons. What are those hidden neurons doing? Well, just suppose for the sake of argument that the first neuron in the hidden

Backpropagation through Time

The recurrent structure makes traditional backpropagation infeasible because of that with the recurrent structure, there is not an end point where the backpropagation can stop. Intuitively, one solution is to unfold the recurrent structure and expand it as a feedforward neural network with certain time steps and then apply traditional backpropagation onto this unfolded neural network. This solution is known as Backpropagation through Time

(BPTT), independently invented by several researchers including (Robinson and Fallside,1987; Werbos, 1988; Mozer, 1989)

However, as recurrent neural network usually has a more complex cost surface, naïve backpropagation may not work well. Later in this paper, we will see that the recurrent structure introduces some critical problems, for example, the vanishing gradient problem, which makes optimization for RNN a great challenge in the society.

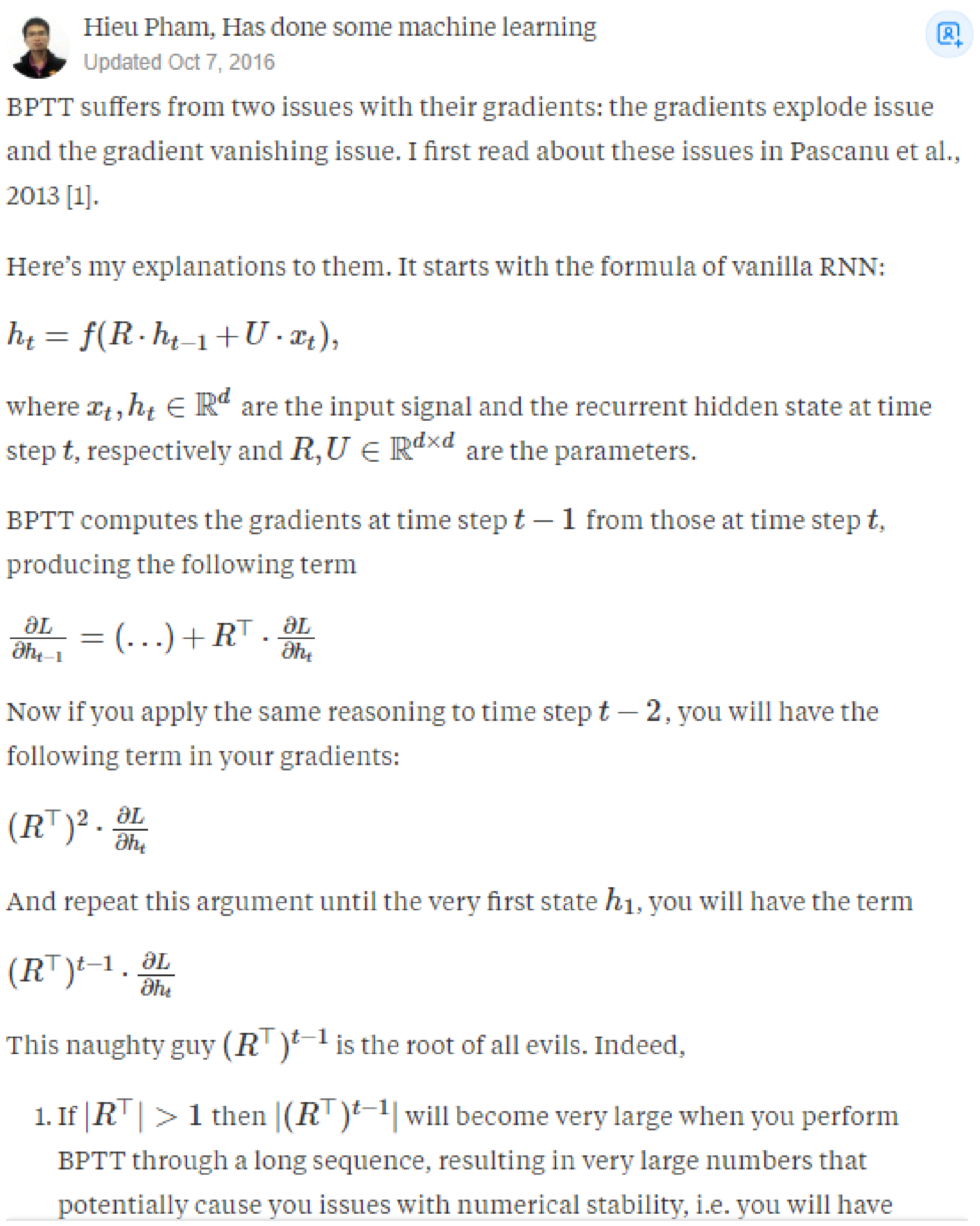
The backpropagation algorithm

The backpropagation equations provide us with a way of computing the gradient of the cost function. Let's explicitly write this out in the form of an algorithm:

1. **Input x:** Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3. **Output error δ^L:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Examining the algorithm you can see why it's called *back*propagation. We compute the error vectors δ^l backward, starting from the final layer. It may seem peculiar that we're going through the network backward. But if you think about the proof of backpropagation, the backward movement is a consequence of the fact that the cost is a function of outputs from the network. To understand how the cost varies with earlier weights and biases we need to repeatedly apply the chain rule, working backward through the layers to obtain usable expressions.

Vanishing Gradient



Hieu Pham, Has done some machine learning
Updated Oct 7, 2016

BPTT suffers from two issues with their gradients: the gradients explode issue and the gradient vanishing issue. I first read about these issues in Pascanu et al., 2013 [1].

Here's my explanations to them. It starts with the formula of vanilla RNN:

$$h_t = f(R \cdot h_{t-1} + U \cdot x_t),$$

where $x_t, h_t \in \mathbb{R}^d$ are the input signal and the recurrent hidden state at time step t, respectively and $R, U \in \mathbb{R}^{d \times d}$ are the parameters.

BPTT computes the gradients at time step $t-1$ from those at time step t, producing the following term

$$\frac{\partial L}{\partial h_{t-1}} = (\ldots) + R^\top \cdot \frac{\partial L}{\partial h_t}$$

Now if you apply the same reasoning to time step $t-2$, you will have the following term in your gradients:

$$(R^\top)^2 \cdot \frac{\partial L}{\partial h_t}$$

And repeat this argument until the very first state h_1, you will have the term

$$(R^\top)^{t-1} \cdot \frac{\partial L}{\partial h_t}$$

This naughty guy $(R^\top)^{t-1}$ is the root of all evils. Indeed,

1. If $|R^\top| > 1$ then $|(R^\top)^{t-1}|$ will become very large when you perform BPTT through a long sequence, resulting in very large numbers that potentially cause you issues with numerical stability, i.e. you will have

Now if you apply the same reasoning to time step $t-2$, you will have the following term in your gradients:

$$(R^\top)^2 \cdot \frac{\partial L}{\partial h_t}$$

And repeat this argument until the very first state h_1, you will have the term

$$(R^\top)^{t-1} \cdot \frac{\partial L}{\partial h_t}$$

This naughty guy $(R^\top)^{t-1}$ is the root of all evils. Indeed,

1. If $|R^\top| > 1$ then $|(R^\top)^{t-1}|$ will become very large when you perform BPTT through a long sequence, resulting in very large numbers that potentially cause you issues with numerical stability, i.e. you will have lots of `NaN` and `inf`. This is called the **gradient exploding** problem.
2. If $|R^\top| < 1$ then $|(R^\top)^{t-1}|$ will become very small, and whatever happens at time step t would have very little impact on the first state h_1. This is undesired, as when we use RNNs, we want them to model the long-term dependencies in our sequences. This is called the **gradient vanishing** problem.

That said, the BPTT algorithm and its variation truncated BPTT are still very relevant to RNNs, as they are the only (sufficiently efficient) way to compute gradients of RNNs. They are nothing more than an application of the chain rule, aka the same rule that backs the normal back-propagation algorithm. In fact, if you use automatic differentiating software such as Torch, TensorFlow or Theano, they all use the chain rule to compute the gradients of their computational graphs. Being applied on RNNs, those will be nothing more than BPTT or truncated BPTT. So yes, you should learn them.

[1] http://www.jmlr.org/proceedings/...

1.9k Views · View Upvoters

Upvote · 11 Share

LTSM

Another breakthrough in RNN family was introduced in the same year as BRNN. Hochreiter and Schmidhuber (1997) introduced a new neuron for RNN family, named Long Short-Term Memory (LSTM). When it was invented, the term “LSTM” is used to refer the algorithm that is designed to overcome vanishing gradient problem, with the help of a special designed

memory cell

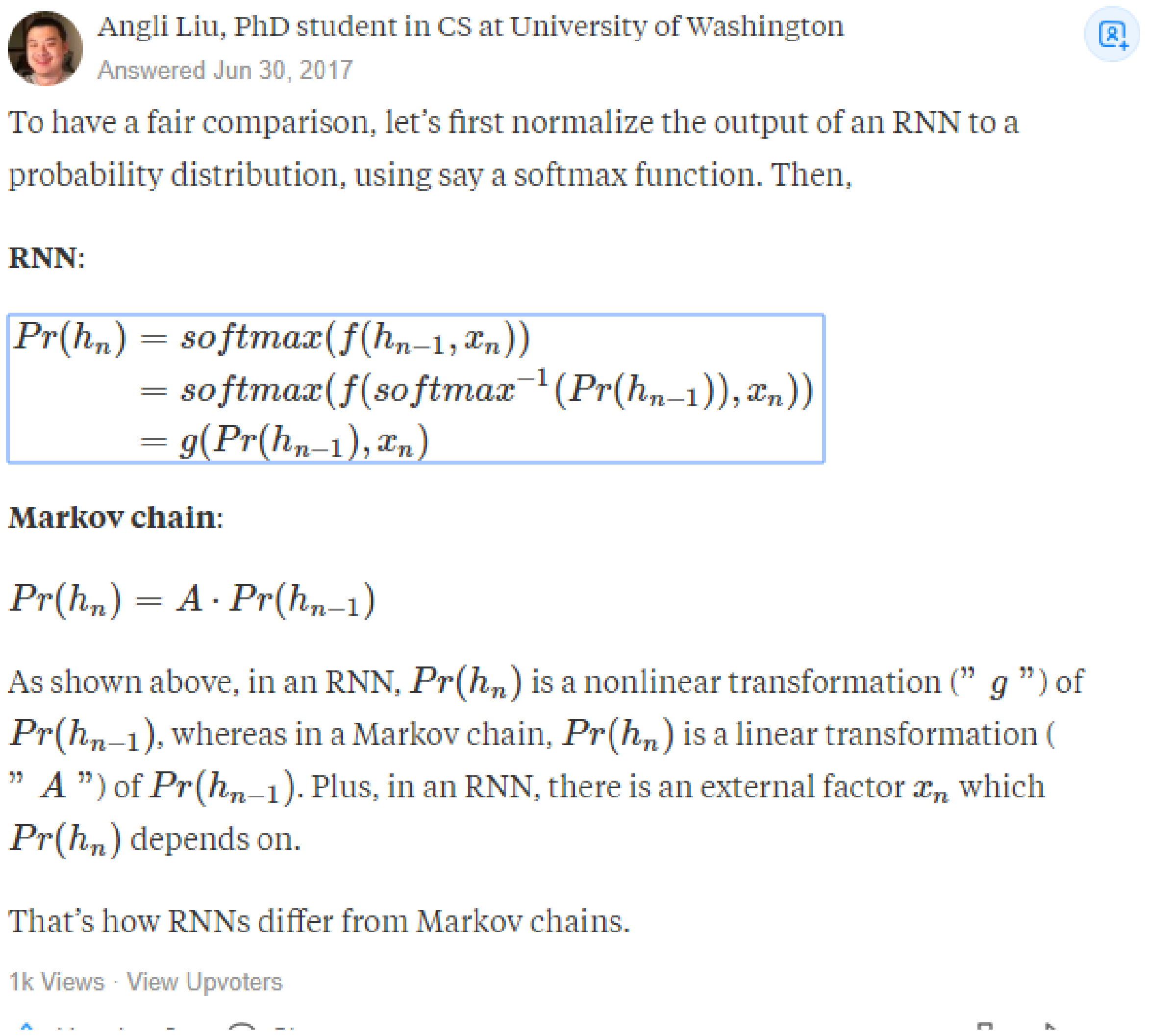
LSTM is widely used to denote any recurrent network that

with that memory cell, which is nowadays referred as an LSTM cell.

LSTM was introduced to overcome the problem that RNNs cannot long term dependencies (Bengio et al., 1994). To overcome this issue, it requires the specially designed memory cell, as illustrated in Figure 24 (a).

LSTM consists of several critical components

Distinction between Markov Chains and RNN



Angli Liu, PhD student in CS at University of Washington

Answered Jun 30, 2017

To have a fair comparison, let's first normalize the output of an RNN to a probability distribution, using say a softmax function. Then,

RNN:

$$\boxed{\begin{aligned} Pr(h_n) &= softmax(f(h_{n-1}, x_n)) \\ &= softmax(f(softmax^{-1}(Pr(h_{n-1})), x_n)) \\ &= g(Pr(h_{n-1}), x_n) \end{aligned}}$$

Markov chain:

$$Pr(h_n) = A \cdot Pr(h_{n-1})$$

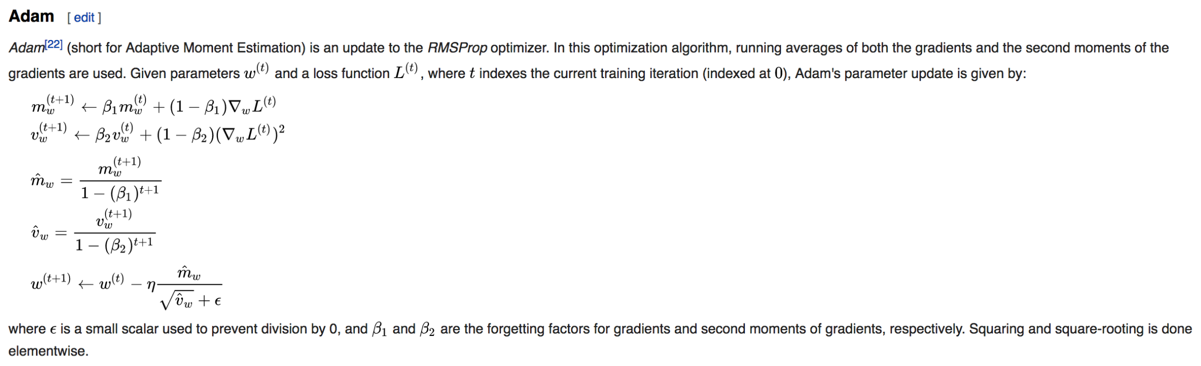
As shown above, in an RNN, $Pr(h_n)$ is a nonlinear transformation (" g ") of $Pr(h_{n-1})$, whereas in a Markov chain, $Pr(h_n)$ is a linear transformation (" A ") of $Pr(h_{n-1})$. Plus, in an RNN, there is an external factor x_n which $Pr(h_n)$ depends on.

That's how RNNs differ from Markov chains.

1k Views · View Upvoters

I suspect this is because `fittedvalues` are predictions of the differenced series. If you want predictions in terms of the original series, you can do:

Adam Optimizer



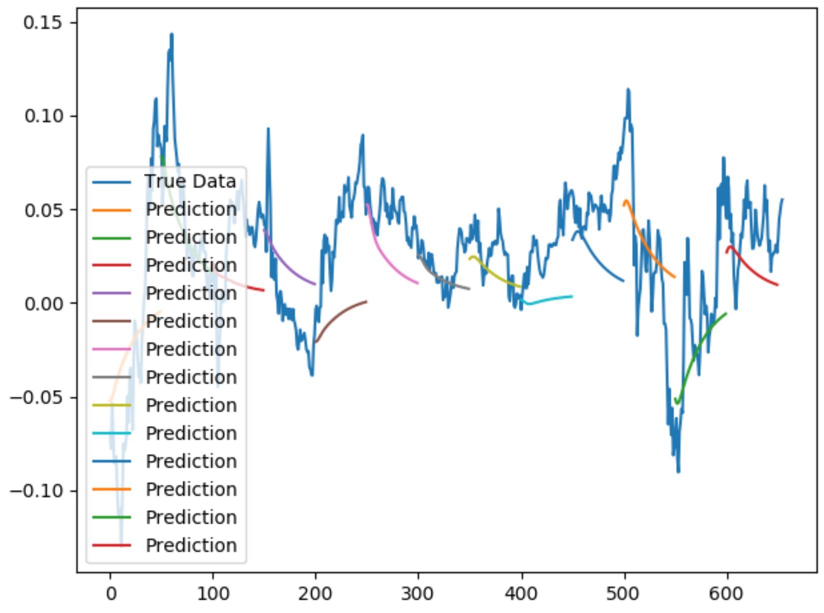
Adam [edit]

Adam[22] (short for Adaptive Moment Estimation) is an update to the *RMSProp* optimizer. In this optimization algorithm, running averages of both the gradients and the second moments of the gradients are used. Given parameters $w^{(t)}$ and a loss function $L^{(t)}$, where t indexes the current training iteration (indexed at 0), Adam's parameter update is given by:

$$m_w^{(t+1)} \leftarrow \beta_1 m_w^{(t)} + (1-\beta_1)\nabla_w L^{(t)}$$
$$v_w^{(t+1)} \leftarrow \beta_2 v_w^{(t)} + (1-\beta_2)(\nabla_w L^{(t)})^2$$
$$\hat{m}_w = \frac{m_w^{(t+1)}}{1-(\beta_1)^{t+1}}$$
$$\hat{v}_w = \frac{v_w^{(t+1)}}{1-(\beta_2)^{t+1}}$$
$$w^{(t+1)} \leftarrow w^{(t)} - \eta\frac{\hat{m}_w}{\sqrt{\hat{v}_w}+\epsilon}$$

where ϵ is a small scalar used to prevent division by 0, and β_1 and β_2 are the forgetting factors for gradients and second moments of gradients, respectively. Squaring and square-rooting is done elementwise.

```
fittedvalues = res.predict(typ='levels')
```



Batch size determination

too small and you risk making your learning too stochastic, faster but will converge to unreliable models, too big and it wont fit into memory and still take ages.

In the neural network terminology:

- one **epoch** = one forward pass and one backward pass of *all* the training examples
- **batch size** = the number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need.
- number of **iterations** = number of passes, each pass using [batch size] number of examples. To be clear, one pass = one forward pass + one backward pass (we do not count the forward pass and backward pass as two different passes).

Example: if you have 1000 training examples, and your batch size is 500, then it will take 2 iterations to complete 1 epoch.

When **solving** with a CPU an **Optimization Problem**, you **Iteratively apply** an **Algorithm over** some Input **Data**. In each of these iterations you usually update a Metric of your problem **doing** some **Calculations on** the **Data**. Now **when** the size of your **data** is **large** it might **need** a **considerable** amount of **time** to complete every iteration, **and** may consume a lot of **resources**. **So** sometimes you choose to **apply** these iterative **calculations on** a **Portion of** the **Data** to save time and computational resources. This portion is the batch_size and the process is called (in the Neural Network Lingo) batch data processing. When you apply your computations on all your data, then you do online data processing. I guess the terminology comes from the 60s, and even before. Does anyone remember the .bat DOS files? But of course the concept incarnated to mean a thread or portion of the data to be used.

Glossary

Batch size is a term used in machine learning and refers to the number of training examples utilised in one iteration. The **batch size** can be one of three options: **batch** mode: where the **batch size** is equal to the total dataset thus making the iteration and **epoch** values equivalent.

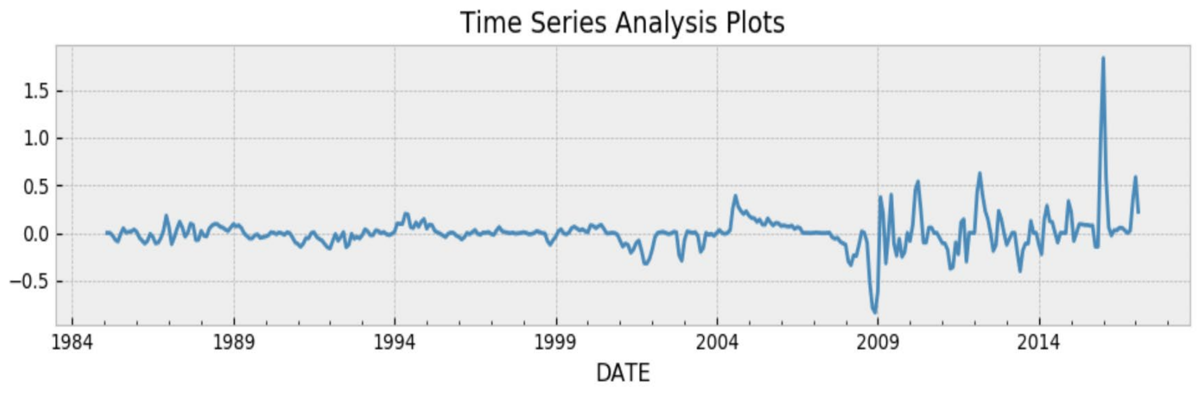
Summary

5. Chapter IV.

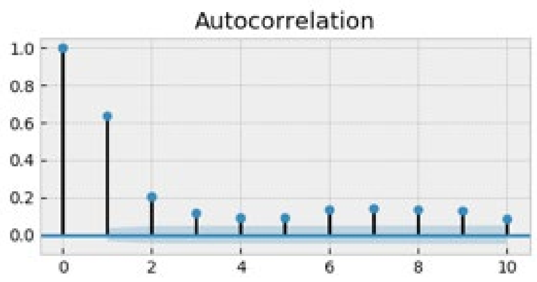
Classical Methods

In this section a limited analysis of two classic linear predictor is presented. The commodity selected for analysis is **Gold**. The commodity was selected for the following reasons, historic and reliable price quotes for are monthly (not) is you wish to go back a long time historically Gold as an asset relative to other financial instrument is illiquid and relatively stable. In comparison to other asset Gold is not as volatile, thus not prone to arbitrary jumps. Forecasting gold prices with machine learning is one of the motivations for this paper. There is a common conception that deep learning algorithm perform poorly with small datasets. In this experiment the commodity Gold, with monthly dataset for over 30 years will be applied to various predictive models, the objective is to compare the relative performance of classic models, including VAR and ARIMA, then contrast the predictive performance with a machine learning algorithm LSTM.

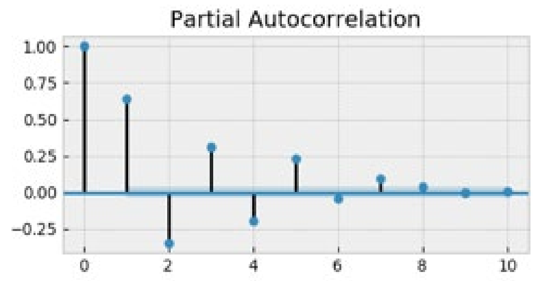
Exploratory Data Analysis



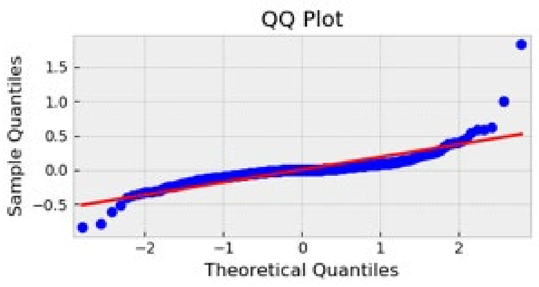
Autocorrelation



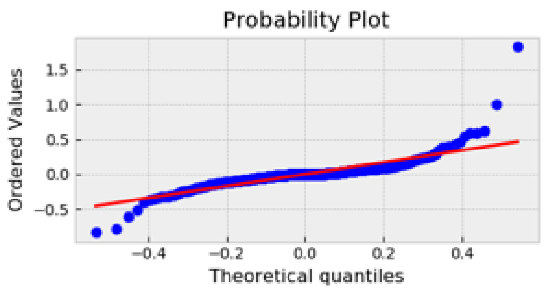
Partial Autocorrelation



QQ Plot



Probability Plot



Descriptive Statistics

Distribution of returns

Test for stationarity

Dicky fuller Test	Box-Ljung test

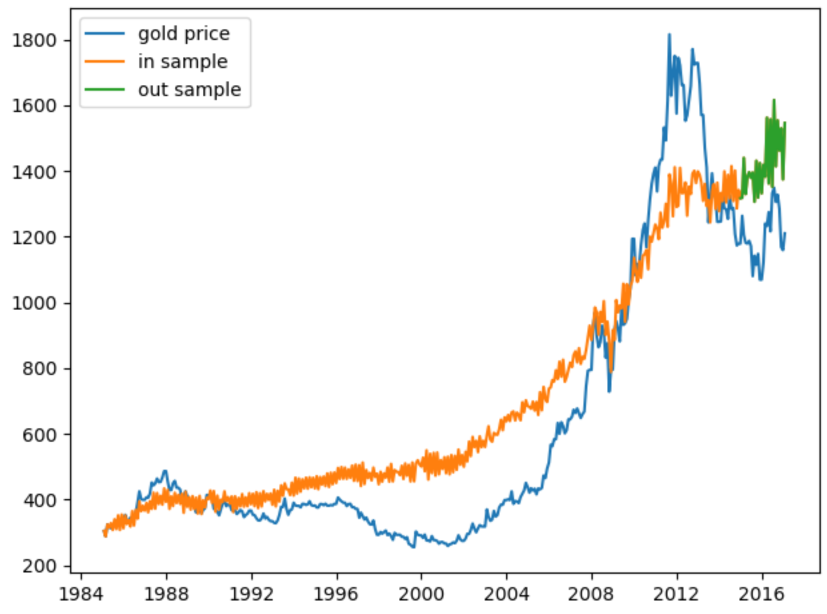
VAR Model

Model Summary

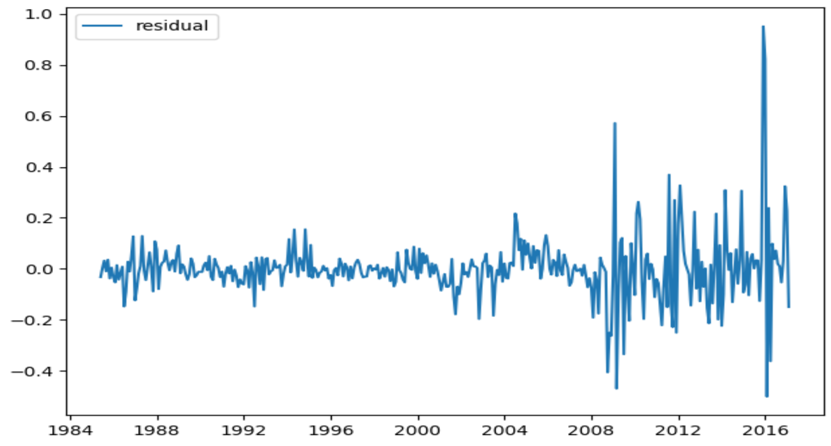
Model:	VAR		
Method:	OLS		
No. of Equations:	4	BIC:	-27.3445
Nobs:	381	HQIC:	-27.7690
Log likelihood:	3248.71	FPE:	6.59078e-13
AIC:	-28.0482	Det(Omega_mle):	5.53483e-13

	coefficient	std. error	t-stat	prob
const	0.008636	0.004346	1.987	0.047
L1.Gold_ret	0.650066	0.051679	12.579	0.000
L2.Gold_ret	-0.499789	0.059456	-8.406	0.000
L3.Gold_ret	0.31606	0.059173	5.341	0.000
L4.Gold_ret	-0.14064	0.051525	-2.73	0.006

VAR Gold



Residual plot



Error Metrics

rmse 238.01521006034054

mse 56651.24022006804

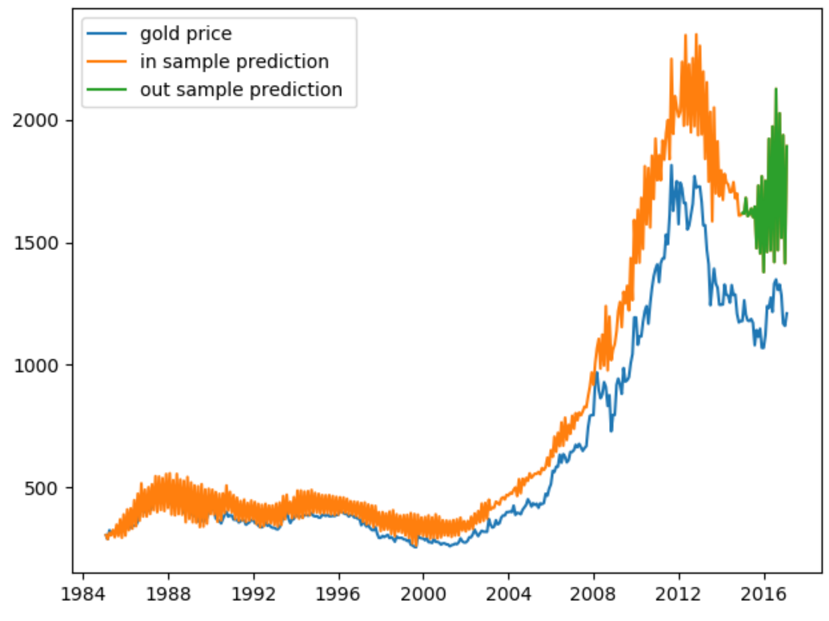
mse_log 0.03317009614943625

rsquared -6.330709837631952

mae 218.71415906552355

Auto Regressive Integrated Moving Average

ARIMA Gold



Residual plot

6. Chapter IV.

LSTM

Mathematical Formulation

Experiment

Dataset Division

Building The Model

Training Model

Validation Technique

Tuning The Model

Summary

7. Chapter V.

Comparative Analysis

Introduction

ARIMA

VAR

LTSM

Experimental Data

The dataset used in the experiment describes the monthly prices for Gold over a 30-year period from 1985-02-01 to 2017-02-01. Gold was chosen for it relative stability with respect to price, Gold was selected instead of monthly interest rate because although gold price is generally stable, it can become volatile when markets become uncertain and as business cycle changes. There are 386 observations in the dataset. The original dataset is credited to yahoo finance.

Training and Validation Dataset

The two key tasks regarding data splitting were

1. with fewer training sets, parameter estimates variance will be high.
2. With fewer testing sets, performance statistic variance will be high.

The dataset was divided such that both concerns were addressed. In spite of the limited dataset, the model can be computationally expensive, using a limited dataset was a deliberate design decision for this experiment. Training Data was 80 percent, Testing Data, 20 percent Validation Data, 20 percent of Training Data.

Data	Count
Training	310
Testing	76
Validation	62

Several runs were executed with different amounts of training data, e.g randomly sample 25 percent of the training data a few times to observe performance on the validation data, again we repeat with 30 and 50 percent, to observe lower variance across the .You should see both greater performance with more data, but also lower variance across the various samples.

Data Transformation

The dataset was transformed to ensure the data is stationary, one of the ways to achieve this is by differencing the data to remove properties such as trends and seasonality. To complete the forecasting process, the data will be inverted restore the data back into its original scale. Python provides handy function to achieve differencing. Input data into the neural network will have be scaled to fit into the activation function of LSTM, the function output is within the range (-1,1). Th scaling coefficient values is calculated on the training set and applied to the test set. The dataset "array" is reshaped into a matrix before scaling the data to range between (-1,1)

Building the model

The model was implemented using industry standard deep learning python packages

Kera, Tensorflow. The solution was developed on mac book pro without GPU. One of the key contributions of LSTM architecture is its capacity to remain stateful and maintain memory over long sequences, without reliance on lagged observation as input. The Kera package exposes several high level api that gives us fine grained access to the model's properties.

Before we build the model, we need to simplify and reframe the problem into a supervised learning problem. We can achieve this by feeding in observation from previous time step as input into the current time step. The data is modified such that we push all values in a series down by a specific number of place e.g 1. The sifted data will become the input variable and the original time series then become the output variable.

We frame the problem as each time step in the original sequence is one separate sample, with one timestep and one feature. Below is high level pseudocode

1. We specify the shape of the input layer, using the *batch_input_shape* parameter

    This parameter specifies the number of

    - Batch size
    - Time step
    - Feature
2. neuron is specified in the output layer with an activation function
3. Loss function: “mean squared error”
4. Specify optimiser: “Adam”
5. Compile the model
6. Fit model to training data

Note there are lots of options provided by the Kera library that were used while experimenting

Running the Forecast

We update the model each time step of the test data as new observations from the test data are made available (the dynamic approach). The *predict* function returns an array of predictions, one for each input row provided. Because we are providing a single input, the output will be a 2 dimensional array with one value. Given a fit model, a batch-size used when fitting the model (e.g. 1), and a row from the test data, the function will separate out the input data from the test row, reshape it, and return the prediction as a single floating point value. We would like the model to build up state as we forecast each time step in the test dataset. We seed the state by making a prediction on all samples in the training dataset. the internal state should be set up ready to forecast the next time step.

Evaluating the model

7. Evaluate forecast, print root mean squared error
8. Print forecast and diagnostic graphs

Rsquared and

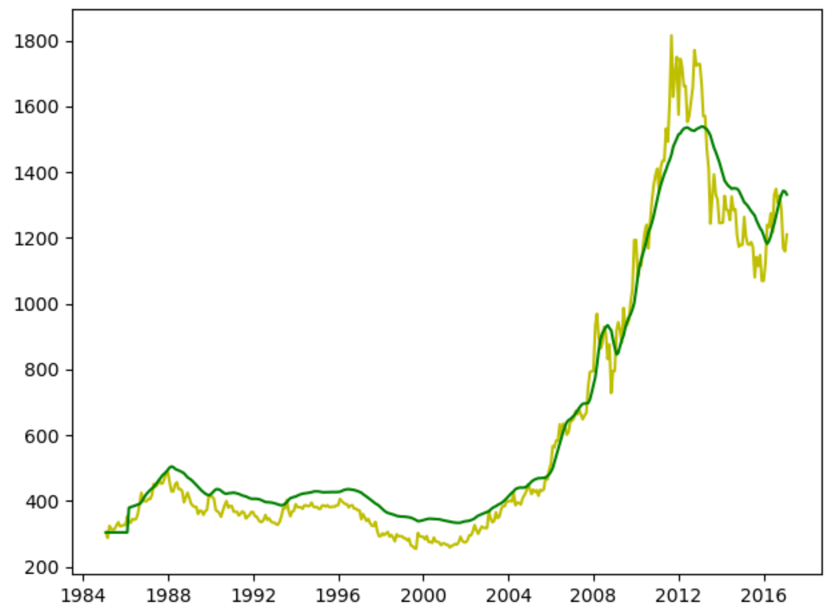
T test statistc

LTSM accuracy score

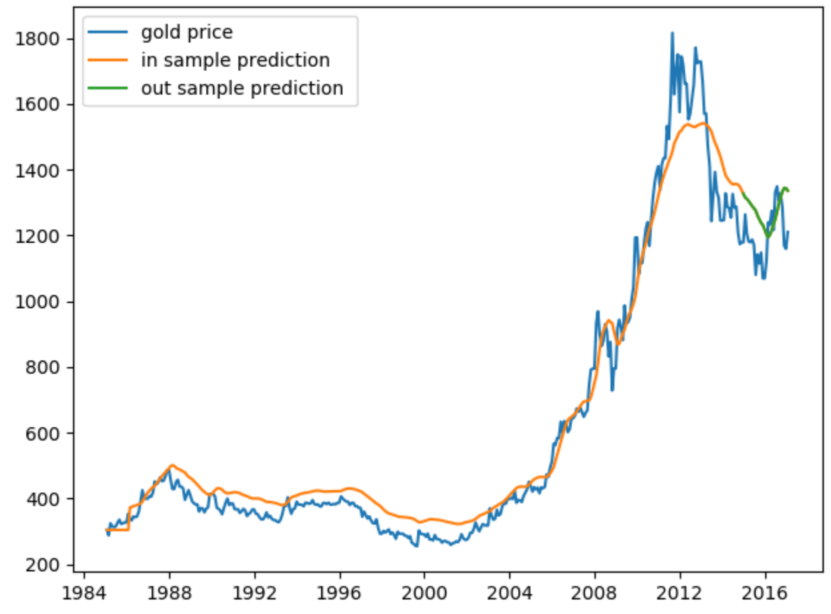
Dropout 0.1 Epoch 400 Batch size = 4

RMSE for all 3 models to compare

From were able to achieve good result as

LTSM Result


LTSM Result

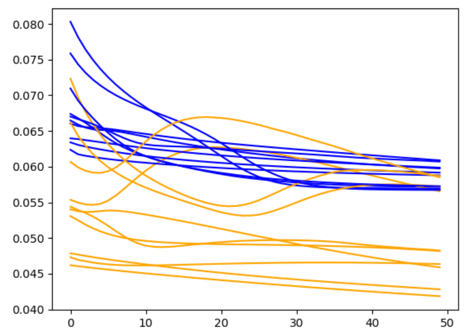


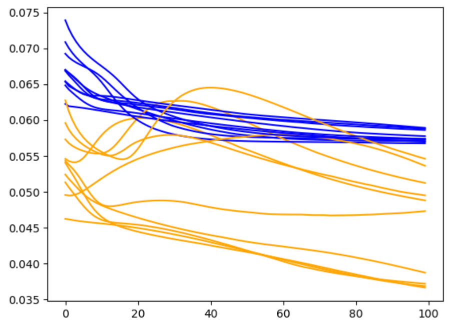
Comparison of RMSE across three models

Epoch size

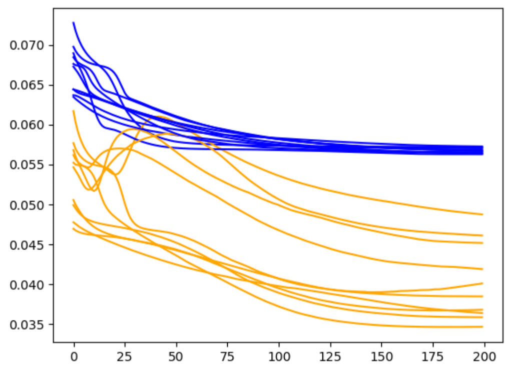
The RMSE for train dataset in blue and test dataset in orange. The performance of the model increases as the average error rate decreases with the gradual increment in epoch sizes up until around 370 epoch. The charts below were generated by training the model at least 30 times to capture the average rate.

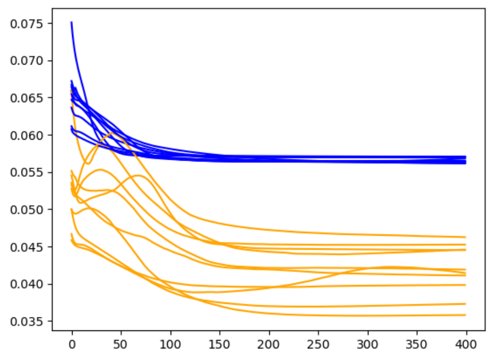
epoch = 50	**epoch = 100**



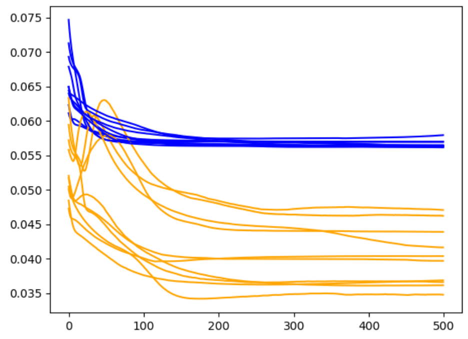


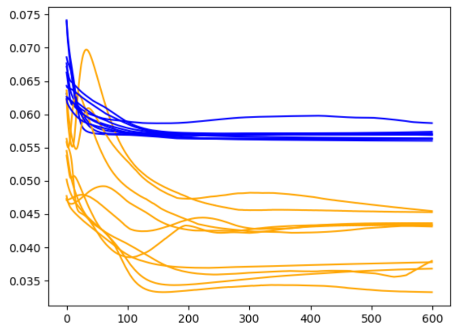
epoch = 200 epoch = 400





epoch = 500 epoch = 600





RMSE Summary statistic for multiple epochs

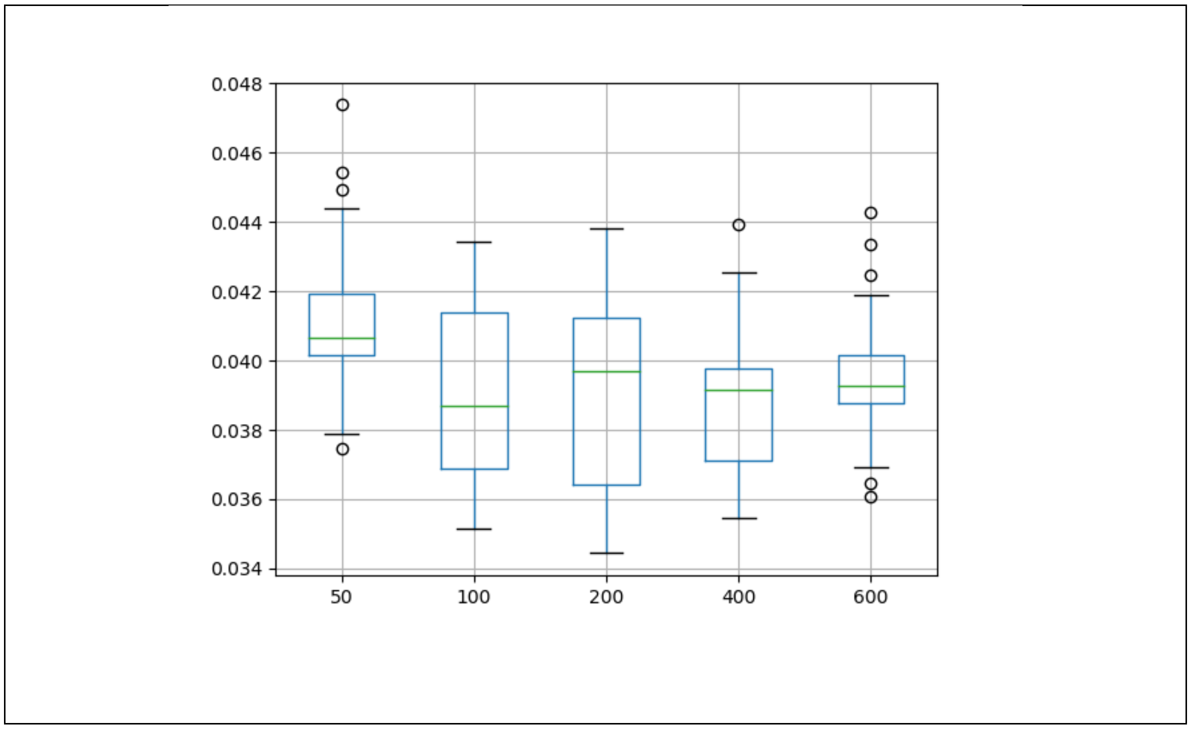
epoch	50	100	200	400	600
count	30	30	30	30	30
mean	0.041167	0.038932	0.039237	0.039051	0.039483
std	0.002293	0.002645	0.002693	0.002132	0.001907
min	0.037459	0.035155	0.034444	0.035461	0.03608

25%	0.040149	0.036894	0.036413	0.037103	0.038752
50%	0.040642	0.038702	0.039712	0.039171	0.039282
75%	0.04192	0.041404	0.041226	0.039775	0.04015
max	0.04738	0.043446	0.0438	0.043938	0.044287

Distributions shown on a box and whisker plot, helps us to visually compare errors somewhat objectively the distributions directly compare. The green line shows the median and the box shows the 25th and 75th percentiles, or the middle 50% of the data. This comparison also shows that setting epochs round about the 400 mark, it has the second lowest average performance but also the fewest lower percentile error. It took nearly 8 hours on a quad core 2017 mac book pro, 16G of ram, but no GPU to get a single run for the box plot below.

<u>Box and whisker plot for multiple epochs</u>

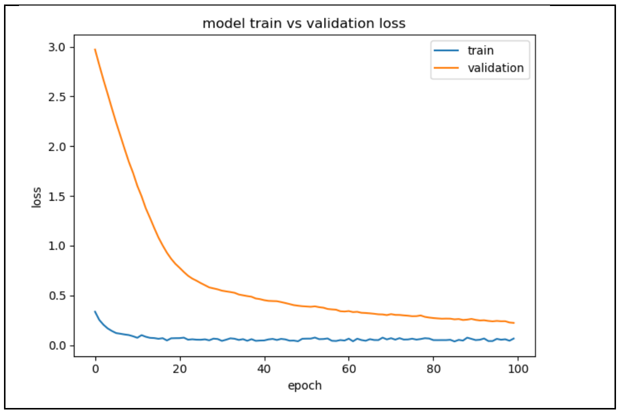
Box Plot Multiple Epochs



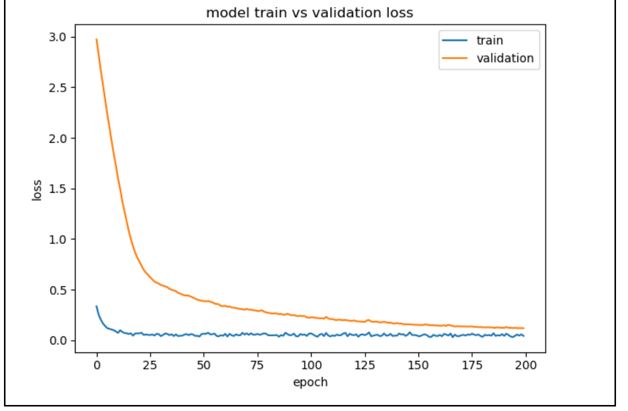
Model training Vs Validation Loss

The charts below illustrates the model converging nicely at around 300 epochs and starts to overfit slightly at around 370 epoch. These validation loss graphs are provide by the Kera's library. The plot is model training loss against the model's validation loss.

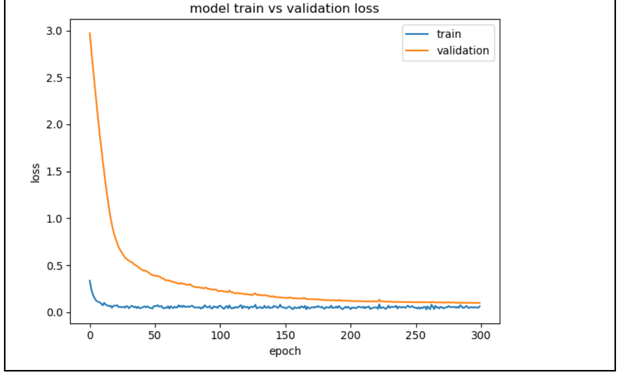
epoch = 200



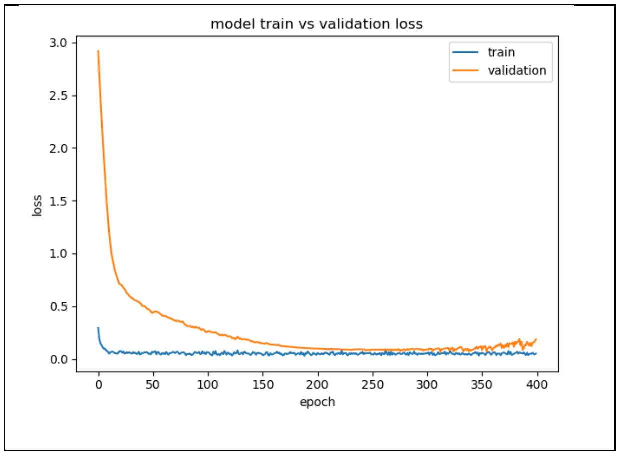
epoch = 200



epoch = 300

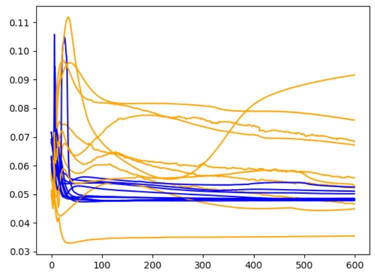
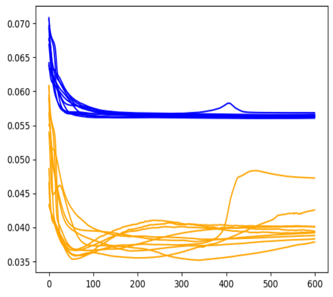
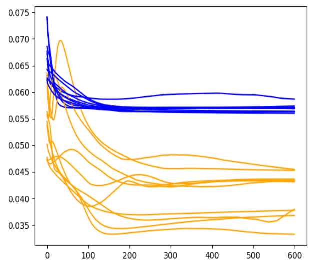


epoch = 400

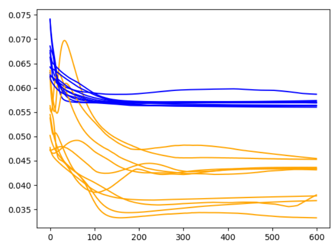
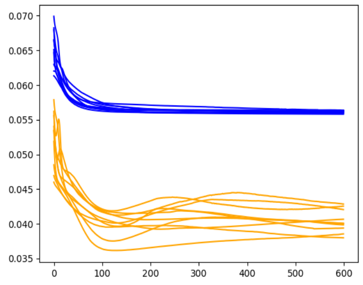
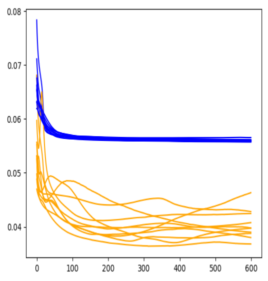


Batch size

The results from tuning the batch size was surprising, conventional knowledge is the bigger the batch size the better the model gets trained. Note the size of the data in the experiment was limited by design. Batch number = 2 seems to offer the best performance, once again notice where batch =2 the inflection point at around 370 epoch the instability is reflected in both the train and test RMSE.

epoch = 600, batch = 1	epoch = 600, batch = 2	epoch = 600, batch = 4
		

Number of neurons

epoch = 600, neuron = 1	epoch = 600, neuron = 2	epoch = 600, neuron = 3
		

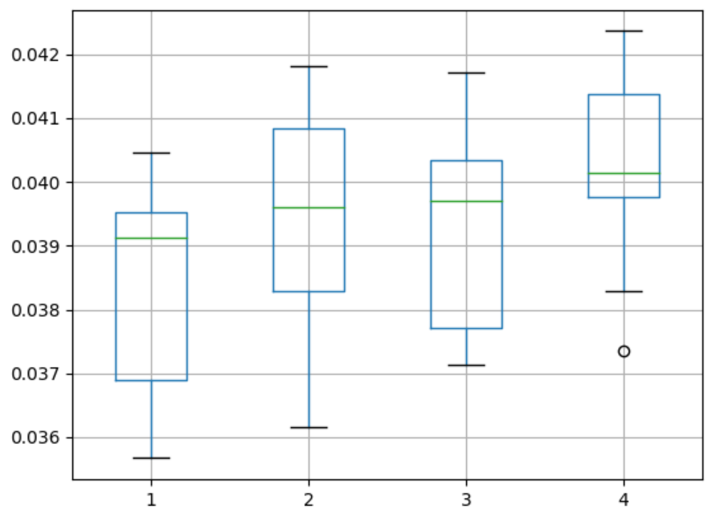
There appears to be clear performance gain from increasing the number of neurons to two the average of the error rate decreases markedly with 2 neurons.

<u>RMSE Summary statistic for multiple neuron</u>

neuron	1	2	3	4
count	10	10	10	10
mean	0.038439	0.039395	0.039196	0.040182
std	0.001838	0.001855	0.001599	0.001527
min	0.035673	0.036152	0.03713	0.037353
25%	0.036902	0.038298	0.037716	0.039769
50%	0.039125	0.039594	0.039698	0.040146
75%	0.039521	0.040842	0.040349	0.041369
max	0.040451	0.04181	0.041721	0.042362

Box and whisker plot for various neuron sizes

Box Plot Multiple Neurons



Pertubing the Model

Mathematical formulation of LTSM,

$$i_t = \sigma\left(W^i x_t + U^i h_{t_{-1}}\right) \tag{1}$$

$$f_t = \sigma\left(W^f x_t + U^f h_{t_{-1}}\right) \tag{2}$$

$$O_t = \sigma\left(W^o x_t + U^o h_{t_{-1}}\right) \tag{3}$$

$$\tilde{c}_t = \tanh\left(W^c x_t + U^c h_{t_{-1}}\right) \tag{4}$$

$$c_t = i_t \odot \tilde{c}_t + f_t \odot \tilde{c}_{t_{-1}} \tag{5}$$

$$h_t = o_t \odot \tanh(c_t) \tag{6}$$

$\left[W^i, W^f, W^o, U^i, U^f, U^o\right]$ are weight matrices,

x_t is the vector input at time step t,

h_t is the hidden state, c_t is the memory cell state

$\odot$ is the element by element multiplication

σ is the sigmoid function

Optimisation

Stochastic gradient descent is a popular training method, the problem can be framed as non convex optimisation problem

$$\min_{w} = \sum_{i=1}^{N} f_i(w), \tag{7}$$

Where f_i is the loss function for the i^{th} data point,

w are the weights of the network, Given a sequence of learning rate γ_κ,

SGD takes the form

$$w_{\kappa_{+1}} = w_\kappa - \gamma_\kappa \widehat{\nabla} f(w_k) \tag{8}$$

Summary

8. Conclusion.

Concluding Remarks

The advancement in machine learning techniques and deep learning algorithms has made deep learning area a keenly researched discipline. The question of accuracy and stability of Machine learning model has drawn so much interest and thus the quest here was to explore a segment of time series forecasting with machine learning. Models may be inspired by domain knowledge outside of engineering or statistics, e.g the human visual cortex is said to have inspired the development of convolutional neural networks.

This paper primarily focuses on LTSM's model performance mainly, but also compares the accuracy of ARIMA and VAR to LSTM, as representative techniques when forecasting monthly time series for Gold. These two techniques were implemented and applied on a set of financial data for Gold. In this paper. Three major family of forecasting models were reviewed, implemented and analysed. Auto Regressive Integrated Moving Average, Vector Auto Regression and Recurrent Neural Network composed of Long Short Term Memory unit. The research question this paper set out to answer was to identify the efficacy and accuracy of predictions LSTM architecture in predicting commodity prices, with limited dataset and to verify if LSTM model outperform ARIMA or VAR.

Key Points

The findings and conclusion of this experiment is consistent with the hypothesis that deep learning neural network architecture, with limited datasets, can produce robust and better fitting models than ARIMA and VAR. It is also clear that deep learning models LSTM when tuned correctly can perform and converge faster than ARIMA and VAR, the speed of convergence was not documented as part of this experiment but observed while running the experiment. By Increasing the number of neurons and epoch the LSTM model experienced clear performance improvement, These two hyper parameters are essential blocks for tuning the model.

By visually inspection of the graph and observing the RMSE, it is apparent that the LSTM-model seemed to have higher prediction accuracy than the VAR and ARIMA model. The VAR model perform comparably well at the beginning of out sample comparison (see VAR graph) Given that VAR and ARIMA are well established and mature models that are widely used in industry, the LSTM architecture results stands out, based on the limited subset of data used in this experiment

focus however was on the application of the Long Short Term Memory variant of recurrent neural network.   Deep learning models: the deep generative model family, convolutional neural network family, and recurrent neural network family as well as some topics for optimization techniques. A proposed model with significantly more parameters than contemporaneous ones must solve problems that no others can solve unambiguously to be noteworthy.

Recommendations for Future Research

More work in area optimizing LSTM hyper parameters. In areas such as

Dropout, Optimization Algorithm, Loss Function.

Experiment and explore by trying out the following, apply other regularization methods e.g dropout on LSTM connections. Apply other optimization algorithms, experiment with different loss functions, to tweak model performance. Test training speed by using lag observations as input features and input time steps of the feature to see if they improve learning and/or forecasting performance of the model.

LSTM in deep reinforcement learning for time series prediction

An area of recent interest in research is reinforcement learning. LSTM is a way in which an agent can build a model of hidden or unobserved state in order to enhance its predictiveness, when information from direct observation do not suffice, but a history of observations may be more reliable. Applying LSTM in a Reinforcement learning environment will be challenging. LSTM requires input of multiple related time steps at once, not randomly sampled individual time steps. Some LSTM architecture might yield better result, if histories of longer trajectories are preserved and long trajectories or even complete episodes are sampled.

9. [References]

Last Name, Full First Name. *Basic MLA Style Only*. City: Publisher, Year. Print.

Last Name, First and Middle Initials (Year). *Basic APA Style Only.* City, State Abbreviation: Publisher.

Last Name, First Name. *Basic CMS Style Only, Requires Footnotes or Endnotes.* Place of publication: Publisher, Year.

Bibliography

Deng, L. & Yu, D., 2014. Deep Learning: Methods and Applications. *Foundations and Trends in Signal Processing,* , 7(3–4), p. .

Heaton, J. B., Polson, N. G. & Witte, J. H., 2016. Deep Learning for Finance: Deep Portfolios. *Applied Stochastic Models in Business and Industry,* , 33(1), pp. 3-12.

Glossary

Samples: These are independent observations from the dataset.

Time steps: These are individual time steps for a variable for a given observation period.

Features: Individual measures identified at the time of observation