A Brief Comparison of Time Series Forecasting Methods:

From Classical to Deep Learning Models

Jadon Costa

University of Colorado Denver

Fall 2021

Table of Contents

1 Abstract

Time series forecasting has historically involved classical techniques that use traditional statistical methods to gain insights. These techniques persist in the modern age because of their robustness, accuracy, and computationally cheap implementation. Moreover, the increasing size and complexity of data has brought about the need to explore information using alternate methods, namely those of machine learning and deep learning. The heart of the debate between these methods is whether or not the machine and deep learning methods can outperform the classical statistical ones. The newer deep learning methods offer the promise of higher accuracy despite their higher computational complexity, which is supported by the literature across various fields. However, some research shows that the machine and deep learning models fail to outperform the traditional statistical techniques. The motivation of this project is to empirically replicate these results to either support or weaken the claim that classical statistical methods outperform machine learning and deep learning methods. In this paper, select time series forecasting models from three major classes of models are compared, including classical statistical techniques (exponential smoothing, ARIMA), basic machine learning methods (support vector regression, classification and regression trees, K-nearest neighbors, multilayer perceptron), and deep learning neural networks (recurrent neural network, long short-term memory, gated recurrent unit). Two datasets of time series are used to test the forecasting performance of each model. The performance is measured in terms of prediction accuracy and computational efficiency, and the models are compared both among and within the sets of method classes. Additionally, as a secondary objective, the obtained performance metrics are compared between the two datasets to observe how certain models perform for different types of data (e.g., traditional seasonal data vs. nontraditional volatile data). The datasets include average hourly air temperature data obtained from the National Oceanic and Atmospheric Administration (NOAA) and hourly cryptocurrency data obtained from bitcoin-to-dollar (BTC/USD) spot prices.

2 Introduction

Time series forecasting has been an important and necessary endeavor for data analysis and statistics. Time series data is data that is ordered over time (e.g., heart rate, stock prices, temperature) and forecasting involves using that historical sequence of data to make future predictions. Indeed, prediction remains ubiquitous in the modern age since almost all domains of research require some form of it. In the current age of computing, the increasing focus is on big data and internet-of-things (IoT), where incredibly large datasets are being collected more and more rapidly. Time series data is only a subset of this new trend of data collection, but its analysis and further development is justified by the increasing demand to model and understand these vastly emerging datasets. For example, a social media company might track a user's behavior over time and could use that information to predict when specifically to place advertisements in that user's feed. As another example, epidemiologists could have historical data on an emerging virus from various geographic regions and might want to predict future infection rates.

A univariate time series is a sequence of data of one variable over a certain interval of time. For example, a person's heartrate over the course of one day. A multivariate time series is a sequence of data of many variables over a certain interval of time. In mutlivariate datasets, one variable is usually chosen as the predictor variable (i.e., the variable that needs to be predicted) and the remaining variables are considered exogenous. For example, a person's heartrate could be the predictor variable while their blood pressure, body temperature, and blood sugar level could be the exogenous variables. The idea is that the exogenous variables might inherently contain some information about the predictor variable and can be used in the forecasting model to make more accurate foreacasts than an univariate model. Datasets for multivariate time series can be quite large and the forecasting techniques can be quite complicated. This paper focuses solely on univariate time series forecasting techniques.

Time series forecasting is particularly difficult because of the ordered nature of the data. With unordered data, typical statistical methods, such as ordinary least squares (OLS) regression, can be implemented by considering each data point as an independent and identically distributed sample from a population. Information about the population can then be deduced by observing trends and patterns in the sample data. However, when the data points are ordered in time, the same assumptions do not hold. Each data point may be dependent on its previous values, which is hardly independent and identically distributed. Additionally, techniques, such as bootstrapping, lose their utility, as the order of the data points cannot be disarranged.

Historically, forecasting time series has involved classical techniques that use traditional statistical methods to gain insights. These techniques persist in the modern age because of their robustness, accuracy, and computationally cheap implementation[15]. Moreover, the increasing size and complexity of data has brought

about the need to explore information using alternate methods, namely those of machine learning and deep learning. The machine and deep learning frameworks have the same goal as their classical statistical counterparts: to optimize an objective (loss) function. Where they differ is in how that optimization is performed; where statistical methods implement linear functions, machine learning methods allow for non-linear techniques[24]. Overall, the heart of the debate between these methods is whether or not the machine and deep learning methods can outperform the classical statistical ones. The newer deep learning methods offer the promise of higher accuracy despite their higher computational complexity, which is supported by the literature across various fields[3][16][29][32]. However, some research shows that the machine and deep learning models fail to outperform the traditional statistical techniques[15][24][25].

The abandonment of the simpler, classical methods in favor of more complex ones by practitioners might be supported by collective biases on the performance of simple methods. For example, one bias may be that the high accuracy and efficiency of classical methods are conditional on model assumptions that are not real. That is, there may exist a more complex class of methods that better fit time series data, and the exploration of these machine and deep learning methods may be the search for that improved class. Green & Armstrong (2015)[14] discuss the popularity of complex methods in general. The authors reviewed a set of studies that compared simple versus complex forecasting methods and found that, of the 25 papers with quantitative comparisons, complex models actually increased forecast error by an average of 27%. They go on to highlight some cognitive biases that make people favor complexity over simplicity, despite simple models outperforming the complex ones. Ultimately, the appeal of complex models is unknown and remains to be understood. Only by objectively comparing empirical results can the forecasting community further support or weaken the prevailing assumptions of complex methods.

In the following sections, a brief history of forecasting competitions will be reviewed (Section 2.1), followed by a literature review of the three broad areas of forecasting methods: classic statistical (Section 2.2), machine learning (Section 2.3), and deep learning (Section 2.4). Section 2.5 follows with a review of how models compare across method classes in the literature. The introduction concludes with the objectives of the paper in Section 2.6. Section 3 introduces the two datasets. Section 4 discusses the experimental design choices and any general assumptions made throughout this paper, including parameter selection, overfitting precautions, and data preprocessing. Section 5 briefly outlines the models implemented and any specific parameter selection tools used. Section 6 displays the results followed by a discussion. Finally, Section 7 concludes the paper by summarizing the results and highlighting areas of future research.

2.1 Forecasting Competitions

Over the past few decades, there has been a growing popularity in time series forecasting, primarily encouraged by various international competitions. These competitions involved a simple premise: each participant is to develop and submit a forecasting model that makes accurate predictions on a given dataset. In most competitions, a preprocessed dataset consisting of a large number of individual time series (e.g., 1000 unique time series), was split into training and test sets, where the participants were given the training data to build their models on. Then their models were implemented on the unseen test data by the hosting organization of the competition to record a measure of accuracy for that model. Prizes were rewarded to the team or participant that developed the most accurate model [25].

The most famous of these competitions are the M competitions. The first M competition was held in 1982 and was organized by Spyros Makridakis. It consisted of 1001 unique time series and involved about 24 different model submissions, mostly consisting of variations on classical techniques. The competitions have changed and developed dramatically over the decades, with the most recent M competition, "M4", in 2018 containing 100,000 time series with 61 different forecasting model submissions, described in detail in an engaging paper written by the creator of the M4 dataset [25]. M5, which involved a hierarchical time series provided by Wal-Mart, already took place in 2020, with preliminary results expected during 2021 and final results during 2022. M6 is scheduled for 2021-2022 and will involve real-time financial data on 100 assets.

The most intriguing insight from the reports of past competition is how different and unique the submitted forecasting models were. There is a clear trend of growing diveristy among methodologies, not necessarily complex ones, which demonstrates precisely where the forecasting community is currently. That is, (1) the search for a high performing forecasting method still persists and (2) the search for such a method has inspired the creation of new methods, forming an entire taxonomy of forecasting models that continues to expand.

2.2 Classical Forecasting Methods

Classical forecasting methods generally refers to traditional statistical techniques. All forecasting models in general perform a minimization of some loss function (e.g., sum of squared errors) to estimate the model's parameters. Statistical models usually perform this optimization using linear algorithms. The linearity of these methods makes them fairly efficient and computationally cheap to implement. These were some of the first and simplest forecasting methods developed, making them well known among practitioners. Some common types include ordinary least sqaures (OLS) regression, the class of exponetial smoothing (ETS) models, and the class of autoregressive moving average (ARMA) models.

Though traditional regression can be used for time series forecasting, it is not commonly implemented. This is primarily due to how time series violate some assumptions of the OLS model. The OLS model usually assumes that the errors (residuals) are independent and identically distributed. However, in time series data, this is hardly the case since there exists a high amount of autocorrelation. Autocorrelation is when a time series is highly correlated with a lagged version of itself (e.g., day t correlated against day $t-1$). Autocorrelation is almost always present in time series data, and not taking this into consideration when performing OLS may result in a falsely high correlation coefficient, a phenomenon known as spurious regression[13]. To avoid these misleading results, time series are commonly forecasted either by using some sort of moving average model that attempts to pick up the average trend or pattern across the series or by using an autoregressive model that uses lagged versions of the series as independent variables.

Moving average (MA) models are quite old, where one of the first methods, simple exponential smoothing (SES), was developed in the 1950s[7]. SES assumes the next forecast in the series is simply the arithmetic mean of all previous observations. This model was quickly improved upon by several people, primarily Charles Holt and Peter Winters, to form the larger family of models known as exponential smoothing (ETS)[17][33]. The general assumption is that the next forecast in the series is a weighted average of all the previous observations, where the weights are determined by the specific ETS model being implemented. For being mathematically simple, these models are excellent at determining the overall trend of the underlying series[19]. Seasonality in a time series can be defined as a recurring pattern of fluctuations that have a fixed and known period, usually corresponding to daily, weekly, or yearly periods. Cyclicality in a time series can be defined as a pattern of fluctuations that does not have a fixed or known period and usually has long patterns of recurrence on the scale of years. One major downside to ETS models is that they have trouble making predictions for series that are seasonal or cyclic in nature because, mathematically, they produce a trend line that averages over the fluctuation[19]. For example, if the underlying series is, say, economic data that is marked by annually recurring peaks and troughs, then the series has annual seasonality and the ETS model would predict a trend line moving through the center of the fluctuations. This would not lead to a useful prediction since a fundamental property of the time series, the fluctuations, were "averaged away". Thus, it is difficult to make ETS models sensitive to seasonality in the dataset. Another disadvantage is that ETS models, by construction, will forecast with a lag, making for an expected trendline that's slightly behind where the actual one should be. More information regarding the math of ETS models can be found in the Exponential Smoothing section below.

Autoregressive (AR) models are regression models in which the independent variables are lagged versions of the given time series. The assumption here is that the autocorrelation in the time series can be used as valuable information for predictions. In other words, it is a regression of a variable against earlier versions

of itself. Future values of the time series are forecasted as linear combinations of previous values of that series, where the coefficients of the previous values are determined via multivariate regression. However, the interpretations of these coefficents differ from the traditional interpretation. In non-sequential data for a continuous predictor variable, the i^{th} coefficient corresponding to the i^{th} explanatory variable represents the amount that the predictor variable would change if the i^{th} explanatory variable were to increase by 1, holding all other explanatory variables constant. In an autoregression model, the presence of lagged versions of the predictor variable in the regression polynomial implies that the predictor variable's value at time t can only be interpreted conditional on all of the previous values of the predictor for times $t-1$ to 0. Because of this, there is the slight disadvantage of losing some interpretability when making predictions in autoregressive models.

Autoregressive moving average (ARMA) models are a clever construction of both moving average models and autoregessive models. Essentially, the moving average component is performed on the residuals of the series and is added to the autoregressive component. ARMA models are exceptionally popular in the forecasting community because of their robustness in making highly accurate forecasts. They can be made to be more or less sensitive to seasonal fluctuations in the series (via the autoregressive portion) while also capturing the underlying trend of the series (via the moving average portion). The ARMA class of models is, in fact, a subset of a broader class of models called the autoregressive integrated moving average (ARIMA) models, where the I component defines a linear transformation perfomed on the time series called differencing (described in detail later on). Furthermore, the ARIMA class of models is yet a subset of another, broader class of models called seasonal ARIMA, or SARIMA, which extend the capabilities of ARIMA to handle seasonality in the series. For example, if there is clear monthly or yearly seasonality, the forecasted value could be explained not just in terms of the observations that immediately precede it but also in terms of the observations from the last seasonal period. As such, SARIMA models are highly robust and can accommate lots of different types of time series. These will be discussed more in detail in the ARIMA methods section. One major downside to ARIMA models is that their relative forecast accuracy tends to drop quickly after more than a few forecast periods (e.g., day, seasonal year), making them only suited for short-term forecasts. Similar to ETS models, ARIMA predictions also possess a lag as compared to the actual pattern due to the construction of the model (only previous values are used for the current prediction).

Overall, ETS and ARIMA models vary in their relative performance. In a paper done by Madden & Tan (2007)[23], the average forecast accuracies of 8 statistical methods were compared on 3003 unique time series, where the accuracy measure was the median absolute percentage error (MAPE). They included 4 ETS models, an ARIMA model, and 3 other statistical methods, all compared against a benchmark model, Naïve 1. Naïve 1 is a simple random walk process in which the next predicted value is equal to the previous

value (i.e., a constant line over the forecast horizon). Predictions were made across 4 different time horizons, resulting in the ARIMA model performing the best for the shortest time horizon and an ETS model (Holt ETS with damped trend) performing best for the other 3 time horizons. The application of an ETS or ARIMA model may depend entirely on the nature of the data (e.g., amount of autocorrelation, presence of seasonality), so model selection should be chosen on a case-by-case basis. Overall, statistical methods provide the benefit of being mathematically simple, highly computationlly efficient, relatively interpretable, and robust[15].

2.3 Machine Learning for Time Series

Machine learning refers to a broad class of models, all varying in structure and application. Similar to statistical models, they attempt to minimize some loss function, but usually through nonlinear techniques. A major benefit of machine learning methods is that they are often designed to be able to handle large, high-dimensional datasets, with the intention to extract valuable information from that data faster and more reliably than statistical techniques. This explains the growing popularity of machine learning in recent years, as it provides a novel way to analyze and extrapolate from the rapidly emerging Big Data datasets.

One of the major differences between statistical methods and machine learning methods is how the models are measured and interpreted. Statistical methods are usually discussed by referring to maximum likelihood estimators, p-values, confidence intervals, and other traditional metrics developed over the decades by Fisher, Neyman, and Pearson, among others. On the other hand, machine learning methods are typically interpreted using a confusion matrix or by referring to the out-of-sample predictive performance and the bias-variance tradeoff[10]. Machine learning methods also sometimes require a much more lengthy training period due to the nonlinearity of the math involved, so computational efficiency is a factor to consider when building and implementing these methods. In this context, "learning" is defined generally to be the process of a model finding optimal values of parameters such that it can fot a training set of data and perform well on new, unseen test data. Naturally, time series forecasting has adopted many machine learning methods because of their ability to learn the underlying data and make reasonable forecasts.

There are two major areas of machine learning: supervised and unsupervised learning. Supervised learning involves a dataset that is labeled. That is, the data consists of a set of features that describe every instance of the data and a set of labels that correspond to each instance. Unsupervised learning involves a dataset that has no labels corresponding to each instance. Time series data is an example of supervised learning since, for every value of the predictor variable, there is a timestamp (label). The goal is to build a mathematical model that takes a data point (instance) with its feature information and map it to an

appropriate label. For prediction, a forecast is determined by providing a model with a new instance (i.e., future timestamp) as an input and computing an expected output (i.e., value of predictor variable). In multivariate time series, the exogenous variables are considered to be additional features. There are myriad machine learning methods that can be modified for time series forecasting, but some of the more common ones include mulitlayer perceptrons (MLPs), K-nearest neighbors (KNN), support vector machine (SVM) regression, Gaussian processes (GP), kernel regression, classification and regression trees (CART), gradient boosting trees (GBTs), radial basis functions (RBFs), Bayesian neural networks (BNNs), random forests (RF), and ensembles. These models will not be defined here, but refer to the Machine Learning Methods section for a mathematical breakdown of the ones used in this paper. For this project, a representative set of machine learning models was chosen such that any observations or results may be generalized to the entire class of machine learning methods.

The literature is diverse and inconsistent in regards to which methods are the best performers. Ahmed et al. (2010)[1] compared eight different machine learning methods by running each model on over 1000 different time series and averaging the accuracy performance over all series. The results concluded that multilayer perceptrons and Gaussian processes performed best overall with radial basis function performing worst. Zhang et al. (2021)[34] compared five machine learning methods using monthly copper price data and found that, again, the multilayer perceptron (MLP) performed best, outperforming random forest (RF), K-nearest neighbors (KNN), support vector machine (SVM), and gradient boosting trees in accuracy. In contrast, Balli (2021)[5] found that SVMs outperformed regression, MLPs, and random forest for COVID-19 pandemic data. Chowdhury et al., (2020)[8] showed that gradient boosting trees and ensemble models outperformed KNN.

Though these inconsistencies could be due to differences in the actual structure of the data, another plausible explanation for these differences could be due to parameter selection. Many of these models require hyperparameters (also called tuning parameters) that define the general structure or behavior of the model before it begins training on the dataset. Hyperparameter selection relies ultimately upon the practicioner, where the selection can be based on experience, domain-specific assumptions on the data, budget constraints (e.g., computational time), etc. These contribute to a potential downside to machine learning methods: the optimal choice of hyperparameters may always be unknown, and are subject to human error. Usually, even if the correct hyperparameters are theoretically known, there may be additional computational downsides, such as inefficiency or numerical overflow. Luckily, hyperparameter selection methods exist that rely less on human decision making, such as grid search and random search, in which ranges of values of hyperparameters are tested. Cross-validation is another useful method in which the data is partitioned into subsets and each individual subset is rotationally used as the test set for prediction. The performance of the model on each of the subsets is then averaged and used as the model's overall performance, from which the hyperparameters

may be selected. Ultimately, machine learning provides a data-focused set of nontraditional methods that are capable of extrapolating information from large datasets.

2.4 Deep Learning for Time Series

Deep learning refers to a specialized subset of machine learning, consisting of highly nonlinear algorithms built to handle a wide variety of tasks. The most common form of deep learning models are neural networks, generally referred to as artificial neural networks (ANNs). The structure of ANNs was inspired by the structure of biological neurons, which mesh together forming a vastly interconnected web of links. Individually, each neuron cell is simple and contains no intelligence, but when all the cells act together, they exhibit properties that cannot be seen on the individual level (e.g., consciousness, language). This phenomenom is called emergence, and is the fundamental motivation to the study of ANNs in mathematics and computer science.

Neural networks are being used for many widespread applications. Convolutional neural networks (CNNs) are being used to classify images, recuurent neural networks (RNNs) are being used for speech recognition, and even more advanced neural network structures are being used for the recommendation algorithms that back everyday apps and websites, such as Amazon, YouTube, and Instagram[11]. Deep learning went through a decades-long "winter" when it was nearly forgotten about. Research was sparse until a seminal paper[16] in 2006 lead by Geoffrey Hinton was published showing that a multilayered neural network could outperform the popular support vector machine (SVM) in classifying the MNIST dataset. Indeed, Hinton became known as the "Godfather of deep learning" for his inventions and improvements in the deep learning field, including backpropagation, deep belief networks, dropout, and rectified linear unit (ReLU)[30].

The general structure of a neural network consists of the input layer and an output layer. The input is what is passed into the model and the output is, of course, what is computed by the model. Layers of neurons can be placed in between the input and output layers, called hidden layers. It is the integration of these hidden layers that makes the neural networks so powerful. In 1989, Hornik et al.[18] showed that multilayered feed-forward networks (FFNNs) are universal approximators, functions that can approximate any other nonlinear function for any arbitrary level of accuracy. However, much of the current hype and focus involve applications of *deep* neural networks (DNNs), which are networks with more than two hidden layers of neurons. DNNs are a major area of interest due to how they learn information. Generally, the upper hidden layers (i.e., the ones closer to the input layer) learn small details about the input information, such as whether a pixel is black or white. Later in the network, the lower hidden layers piece together the small details from the earlier layers to identify more abstract patterns in the data, such as whether a

shape is curved or straight. By composing multiple layers of neurons, the network can identify abstractions and often nonlinear patterns in the underlying dataset that traditional, linear models cannot. The cost, however, of using such versatile models is the vast number of hyperparameters, neural network structures, and function choices required to build and tune such complex networks. DNNs often demand long training times with unstable mathematical structures that require the nuance of an experienced practitioner to debug. Additionally, due to their complexity, neural networks are considered "black boxes", where it is hardly clear how exactly the model reaches its conclusion from its components (recall the concept of emergence). This makes them less accessible and hardly interpretable. Research on deep learning networks is an ongoing endeavor that will likely expand as new forms of data emerge from the Internet-of-Things (IoT) and Big Data movements of the modern age.

For time series forecasting, recurrent neural networks (RNNs) and their variants have shown to be highly successful for prediction. Unlike feed-forward neural networks (FFNNs) where information flows in one direction through the network, RNNs are built with feedback loops that give the network a form of short-term memory. This makes them well-engineered for sequence modeling, including time series modeling, natural language processing (NLP), and even signal processing. Advanced variants of the simple RNN, namely the long short-term memory (LSTM) and gated recurrent unit (GRU) architectures, have been shown to perform very well for time series[4]. Sezer et al. (2020)[29], in a financial time series forecasting literature review spanning from 2005 to 2019, reported that more than half of all published papers on time series forecasting consisted of RNNs, most of which were of the LSTM architecture (60%). This research justifies the inclusion in this paper of the three most popular neural networks used for time series forecasting: the simple RNN, LSTM, and GRU. More details on the structures of these RNNs can be found in the Deep Learning Methods section.

2.5 Comparing Across Classes

A natural question to ask is how do these models compare across method classes? An excellent paper by Makridakis et al. (2018)[24] explored this question. The paper served as a follow up to another paper by Ahmed et al. (2010)[1] in which eight machine learning models were compared. The dataset used was actually a subset of the 3003 time series used in the M3 competition in 2000. Makridakis et al. (2018)[24] followed up by expanding their analysis to include eight statistical methods and two deep learning methods in addition to the eight machine learning models of the original paper. Among the other extensions to the experiment, the most notable of which was comparing *both* the computational complexity and the predictive accuracy of all the models against a benchmark model, Naïve 2. Naïve 2 is a modified version of the random walk Naïve 1 described earlier, in which there is a seasonal component added to the simple, straight-line prediction function. In regard to the accuracy metric, the results concluded that the most accurate models were the statistical models and the worst performing ones were the deep learning models (RNN and LSTM), with the machine learning methods falling somewhere in between. Their results are shown in the bar chart in Figure 1 below, where they additionally included the accuracy performance of the models from Ahmed et al., 2010[1]. See Glossary for a quick reference of the acronyms. The measure of accuracy (vertical y-axis) was the symmetric mean absolute percentage error (sMAPE). The astonishing observation from this figure is that the majority of the machine learning models and all of the deep learning models performed *worse* than the Naïve 2 benchmark.

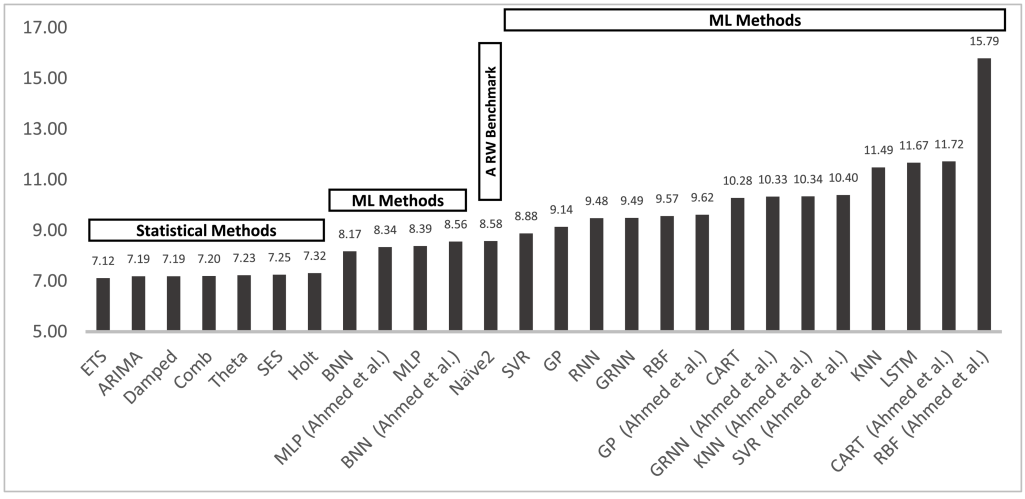

Figure 1: Accuracy Ranking Results; See Glossary
Image Credit: Makridakis et al., (2018)[24]

The article included another chart that plots accuracy (sMAPE) versus computational complexity (CC), where computational complexity was determined as the ratio of computational time of the model to the

computational time of the benchmark model, Naïve 2[24]. The plot is shown below in Figure 2.

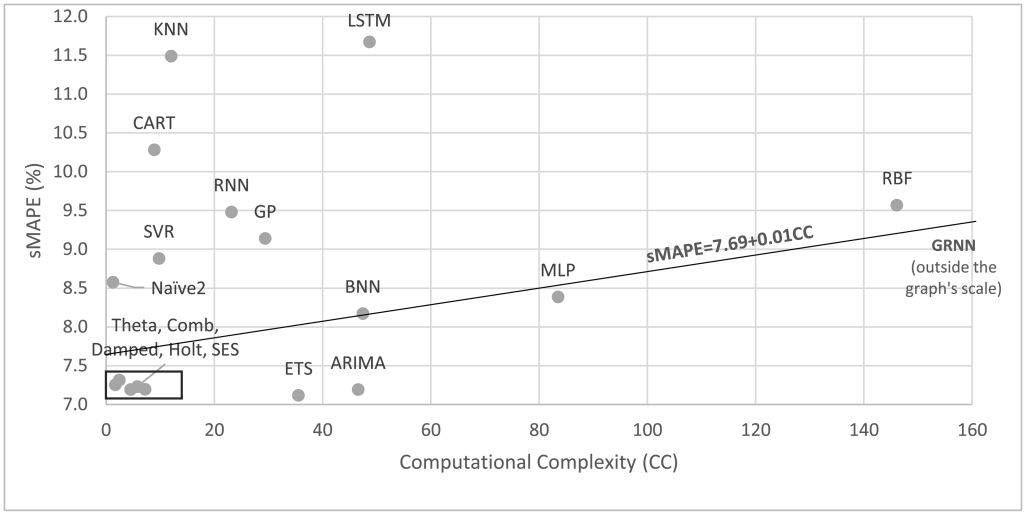

Figure 2: Accuracy (sMAPE) vs. Computational Complexity (CC); See Glossary
Image Credit: Makridakis et al., 2018[24]

Looking at Figure 2, there is a clear negative correlation coefficent between model accuracy (sMAPE) and computational complexity (CC), implying that higher complexity does not necessarily imply higher accuracy. Note that a lower sMAPE value implies a higher accuracy since it is a measure of prediction error. The majority of the statistical models are clustered in a region on the lower left of the plot indicating high accuracy (low sMAPE) and low complexity. This is an important observation as it provides evidence against the claim that deep learning models are "better" than the simpler statistical methods, and supports the Green & Armstrong (2015)[14] paper about biases on complex models, mentioned earlier.

In direct contrast to the results above, there have been various publications demonstrating clear outperformance of deep learning methods over statistical ones. Hinton et al. (2006)[16] demonstrated that DNNs outperformed SVMs in MNIST classification. Li et al. (2019)[21] showed that CNNs can outperfrom statistical methods by using sentiment analysis to predict crude oil forecasting. Tang et al. (1991)[32] compared ANNs to traditional Box-Jenkins (ARMA) methods for time series forecasting and concluded that, in addition to outperforming the traditional methods, neural networks may serve as robust forecasting tools for long-term prediction horizons. Recently, Azarafza et al. (2020)[3] showed that LSTM outperfomed simple RNN, SARIMA, ETS, and moving average (MA) models in forecasting COVID-19 infection rates in Iran. The financial time series literature review by Sezer et al. (2020)[29] claimed that, not only were RNN-based deep learning models the most popular publication topic, but also that deep learning models outperformed machine learning models in the majority of the studies.

It is clear that the literature is divided as to whether machine and deep learning models will live up to

their expectations or merely become a passing fad. Some claim that the deep learning models perform well because they are able to model the non-linear properties of data that statistical models cannot extrapolate as easily[2][15]. Much work is left to be done in the field of machine and deep learning before the claims of any methodology dominance can be justified. Moreover, despite the impressively thorough results of Makridakis et al. (2018)[24] and other contributors, there still remains a large gap in the literature that compares all three model classes for both accuracy *and* computational efficiency. One of the major motivations of this project is an attempt, at least in part, to close some of this gap by empirically testing and comparing each model class on the same data in two experiments (one for each dataset) using measures of accuracy and computational complexity. Speculatively speaking, as computing resources improve and become more available, the issue of computational complexity may minimize, and the competition may reduce to measures of accuracy.

2.6 Objectives

Primary Objective

The large diversity of time series forecasting methods creates a daunting problem for the practicioner: which model should be selected? In this paper, this question is explored by replicating the experimental structure of many of the cited articles. The contribution here is to include multiple variants of the statistical models as well as three deep learning models. Select time series models from each of the three method classes are empirically compared in their forecasting ability. The models were chosen such that they may be respresentative of their corresponding method class, with many of them being popular, well-known methods in the literature. From the classical statistical methods, six models were implemented: four instances of the exponential smoothing (ETS) class and two instances of the ARIMA class (seasonal and nonseasonal). From the machine learning methods, four models were implemented: support vector regression (SVR), K-nearest neighbors (KNN), classification and regression trees (CART), and multilayer perceptron (MLP). These were chosen based on their popularity and relative diversity, not necessarily as the best performing models from the literature. Each model is structured so differently from the others that collectively they are representative of the entire class of machine learning methods. From the deep learning methods, three neural networks were constructed: a recurrent neural network (RNN), a long short-term memory network (LSTM), and a gated recurrent unit network (GRU). All models included in this paper are succinctly displayed in Figure 3 below and are explained in detail in the Methods section.

Secondary Objective

As a secondary objective, the models' performance is compared between two datasets representing different time series patterns. The idea is that some models may be better equipped to handle different types of data and thus may perform better when forecasting that data. A "traditional" series with periodic seasonality was chosen and a "nontraditional" series was chosen with no seasonality and high volatility. Some of the models included in the project (e.g., SARIMA, ETS) are specifically built to model seasonality, while many of the machine and deep learning models are not. So using only one of the two datasets may put the seasonal models at an unfair advantage or disadvantage relative to the other models, hence the inclusion of the secondary objective of this project.

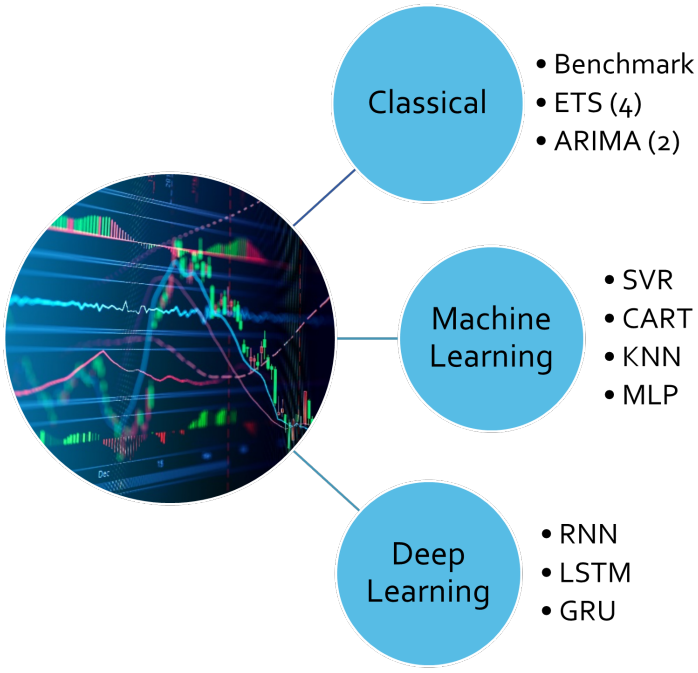

Figure 3: Models by Method Class

3 Data

Two datasets were chosen, each representing different general patterns in time series. The first dataset represents traditional data with seasonal fluctuations and the second dataset represents nontraditional data with no apparent trend or seasonality but with high volatility. Datasets with a large number of continuous observations were chosen since deep learning models generally require longer datasets to train on.

3.1 Traditional Dataset

The first of the datasets is weather data obtained from the National Oceanic and Atmospheric Administration (NOAA)[27]. This organization tracks and collects various weather metrics from across the U.S. One method of data collection is via a nework of weather stations posted all around the country that continuously collect weather metrics, such as sunlight, humidity, wind speed, air temperature, surface temperature, atmospheric pressure, etc. The benefit of using data such as this is that it is relatively continuous and highly granular. The NOAA website offered data from hundreds of locations around the U.S. with data as granular as minute-by-minute observations. This type of data is great for time series modeling as there is an abundance of metrics to choose from, all representing seasonality. For this project, the average air temperature (in degrees Celcius) was obtained from the Boulder, Colorado weather station from 2017 to 2021. A granularity level of hourly observations was selected, resulting in a time series of over 42,000 observations. A plot of the entire time series is shown below in Figure 4.

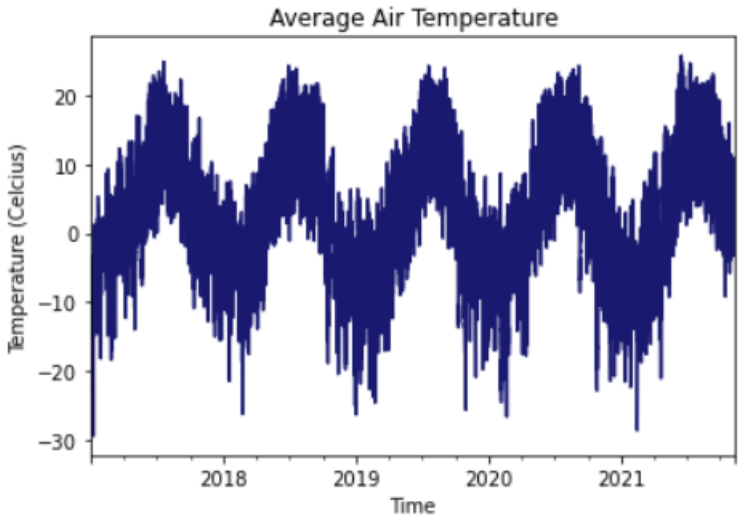

Figure 4: NOAA Seasonal Time Series

As shown, this series is a typical seasonal dataset with clear annual variation and possible daily variation. There doesn't seem to be any significant trend over the years and the fluctuations appear to be constant in their periodicity. The intent of using a dataset like this is that the seasonally-equipped forecasting models might produce more accurate predictions than the models that are not specifically constructed to identify seasonality (i.e., machine learning and deep learning methods).

3.2 Nontraditional Dataset: Crypto

The second dataset obtained was Bitcoin crytocurrency data[6]. A crytopcurrency is a decentralized currency (i.e., no country, nation, or economy owns or controls it) built on the blockchain technology. Since it is decentralized, it is not subject to regulations that other financial assets are, such as trade restrictions and limitations on market hours. For this reason, Bitcoin trades continuously, day and night, and thus has a relatively continuous dataset. This is unlike U.S. financial exchanges which are only able to conduct transactions when the market is open, limited to only several hours a day and not everyday of the year. For this project, Bitcoin-to-USD spot prices (USD/BTC) were obtained with a granularity of hourly observations, ranging from 2017 to 2021. This resulted in a time series with over 37,000 observations. A plot of the entire time series is shown below in Figure 5.

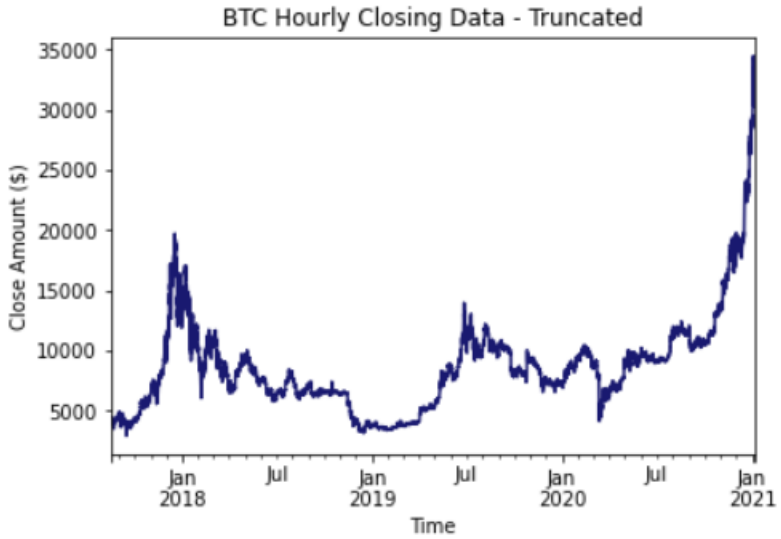

Figure 5: Bitcoin Time Series

As shown, this series does not have any apparent seasonality and, though it has an overall upward trend, it doesn't appear to have a constant trend. Additionally, in the latter months of the graph, there is a large amount of volatility. This dataset was chosen because it represents a "nontraditional" dataset. That is, it has unusual, unpredictable properties that might be difficult for a forecasting model to extrapolate accurate information from.

Unfortunately, due to how the dataset was split into training and test sets, the training set aligned with the less volatile section of the series while the test set aligned almost perfectly with the highly volatile section. Naturally, this resulted in the initial forecasts of many of the models to perform terribly, as a model trained on relatively non-volatile data would not generalize well to highly volatile data. Thus, an experimental decision was made to truncate the Bitcoin series so that the latter, highly volatile section was removed. The

latter 25% was removed giving the updated series shown below. Implementing the models on this truncated Bitcoin series has some justification. In early 2021, an abnormal financial event occurred in which internet groups on social media hyped up several financial securities, including the Bitcoin cryptocurrency, in an attempt to artificially increase the price for profit. This event is what caused the Bitcoin series to have the sharp jump in price in January 2021, shown in the original dataset above in Figure 5. It can be argued that, although crytocurrency is already highly volatile, this rare financial event caused an unnaturally large jump in price and volatility, making that latter section of the series not representative of its usual behavior. It is unlikely that any forecasting model could predict such an event in an univariate series and so only the first 75% of the series was used, as shown in Figure 6 below.

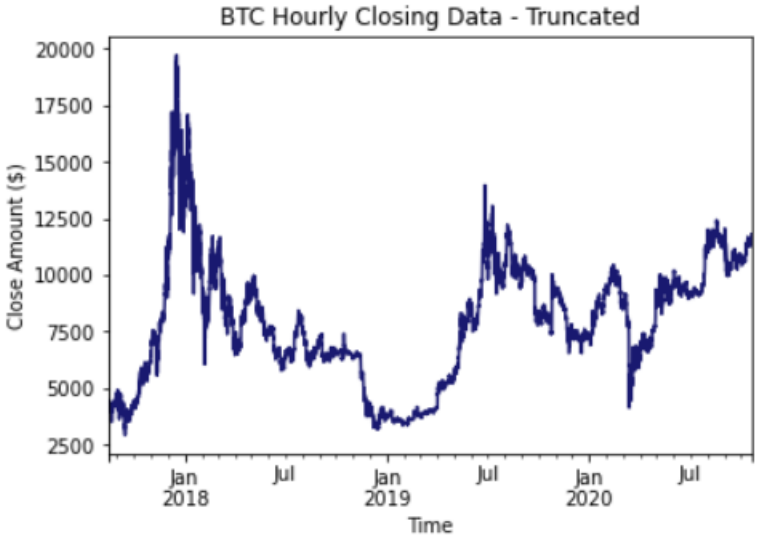

Figure 6: Bitcoin Time Series

4 Experimental Design

4.1 Parameter Selection

All of the models implemented in this project required some form of hyperparameter selection. A hyperparameter is a variable that defines a particular component or instance of a model. For example, the K-nearest neighbors model has its primary hyperparameter in its name, K, which defines how many neighbors to use in the computation. For the sake of making comparisons fair and equal across all models, all of the models were built using pre-built functions from Python packages with the default hyperparameters selected, unless otherwise stated. The default hyperparameters were not used in cases where it was not computationally possible to run (e.g., a learning rate so small that convergence required too much time

or computer memory). Additionally, many of the hyperparameters were selected based on rules of thumb found in textbooks or the literature, or via experimentation. The idea was to implement the most basic, vanilla version of each model. In the Methods section below, each model description is followed by a brief description of which hyperparameters were selected.

4.2 Overfitting

Model fitness was also considered when building the models and selecting their hyperparameters. A model's fitness refers to how much it has learned the underlying data through training and how well it can generalize what it has learned to new, unseen data. If a model learns the training data too well, then it cannot generalize well on forecasts and is said to be overfit. If a model, does not learn the training data well enough, then it also cannot generalize well and is said to be underfit. In general, overfitting presents as the most difficult task to manage in time series forecasting.

There are various techniques used to curb overfitting, ranging differently across the method classes. Some models contain built-in hyperparameters that directly assess overfitting, while others do not. For the classical models, statistical tests along with some rules of thumb were used to address model fitness. For the machine learning models, each was varied so differently in structure that it was difficult to measure and compare model fitness. Graphical analysis of how well the model fit the training set was used as a proxy to an actual metric.

For the deep learning models, the functions used contained metrics that track model fitness during training. One such metric is the loss metric, which measures the training error as the algorithm learns the data. There is another metric that measures accuracy, which is usually only available for classification problems because it is binary (e.g., a prediction is either correctly or incorrectly classified with its true label). However, a custom accuracy metric was created and passed into the MLP and deep learning models to track model accuracy during training. The mathematical details of the loss and accuracy metrics are explained in detail in the Deep Learning Methods section.

Additionally, the MLP and deep learning models contain random components that affected the model's performance. That is, given the exact same data and hyperparameters, different forecasts were observed with different performance metrics for a model. This posed a difficulty for comparison as it was uncertain if or when the model would perform reasonably. It was decided to fix the random state in the code before running these models so that the same output would be produced after each run. This allows for reproducibility of the code and results. The random states were grid searched over a set of values and the best performing models were selected for this project. The varying of the outputs was likely due to the model's optimizer

reaching local minima in the parameter space and resulting in overfit or underfit models. Thus the decision to choose the best performing random states is justified as these represent the models with the best model fitness and performance, given their parameter selection.

Finally, to test the robustness of each model, the datasets were each divided into three training and test splits, where the train/test percentage ratios where chosen to be: 80/20, 85/15, and 90/10. These splits were chosen arbitrarily and were not based on any best practices found in the literature. The 80/20 split was chosen as the primary split upon which the hyperparameters were chosen and model fitness was assessed. If the same model can perform just as well on the 85/15 and 90/10 splits as it did on the primary 80/20 split, then this would provide evidence that the model is robust and can generalize well to new data. If not, then the model may have been overfit to the training set of the primary split. The Results section only discusses the results of the primary 80/20 split. Tables displaying and comparing the results of all three train-test splits are included in Appendix B.

4.3 Data Preprocessing and Transformations

Data imputation is the process of handling missing data in a given dataset. This is important for time series, as missing data points can reduce the performance of, or even halt, the sequencial models being applied to them. Both datasets had a minimal amount of missing data (less than 1%), so advanced data imputation techniques were not considered. A missing value was simply replaced by the arithmetic mean of its boundary values.

A common technique in forecasting is to apply some form of transformation to the dataset before feeding it through a model. Also known as data preprocessing, transformations can have an affect on forecast accuracy or the computational efficiency of those models. However, according to Makridakis et al. (2018)[24], there is a disagreement in the literature as to whether or not preprocessing is necessary for forecasting models. There is still work to be done in determining if data preprocessing affects the predictive performance of certain models.

Some common transformations involve applying a stationarity transformation that removes the trend and/or the seasonality in the data. The idea is that the trend or seasonality inherent in the time series can actually be too complex for a model to understand, so removing these components can simplify the series such that accurate forecasts can be computed. These transformations are also called detrending or mean reversion. Another common transformation that can improve the computational efficiency of the model is scaling the data before running it through a model. For example, the `StandardScaler` transformation from the `sklearn` Python package scales the data so that it looks like standard, normally distributed data. The

`MinMaxScaler` sets the maximum value to be 1 and the minimum value to be 0 and scales all other data to be somewhere in between these. Scaling a time series does not usually remove the trend or seasonality, but just shrinks the y-axis to a smaller range. Whether a time series is transformed or scaled, the predicted values must be converted back to the original data's format by using an inverse transformation or a descaling transformation.

In this project, some transformations and scalers were applied where necessary. The SARIMA class of models actually have a detrending component built into the model itself, called differencing, denoted by the I parameter. This will be explained in more detail in the ARIMA section below. For the ETS models, where there are multiplicative components, the dataset given must not have any values equal to zero to avoid division by zero. So for the NOAA dataset, a simple linear transformation was applied to increase each value such that all values were above zero. No such transformation was required for the BTC dataset.

For the machine learning models, the detrending transformation of differencing was applied only when it improved the model's accuracy performance without jeopardizing the model's fit on the training set. Through experimenting, it was found that the KNN models performed much better when the data was detrended. The CART models had better performance when given the detrended data, but severely underfit the training set. Thus, the decision was made to use the detrended data for the KNN models and the original data for the CART models to preserve model fitness. Additionally, it was found that passing in the original data with the `DateTime` index from the `pandas` package was too complex for some models to understand as an observation label. So a simple range of integer values was passed into such models instead, where the training set would be labeled from 1 to t and the test set would be labeled from $t+1$ to T.

For the MLP and deep learning models, all were given scaled data. This is a common, almost necessary transformation used in practice. It improves the efficiency of the networks, especially for data that has large values (e.g., Bitcoin). The `MinMaxScaler` of the `sklearn` package was selected for these models. The performance of using other scalers was not tested.

4.4 Materials

Programs, Packages, and Code

The code for this project was implemented using Python and Jupyter Notebooks. The primary packages and their versions are shown in the table below. All of the timestamp indices for the sequential dataframes were defined via the exceptionally helpful `datetime` package.

Python package	Version
Python	3.8.3
`numpy`	1.19.5
`matplotlib`	3.2.2
`pandas`	1.4.2
`scipy`	1.5.0
`sklearn`	0.23.1
`keras`	2.6.0
`tensorflow`	2.6.0
`pmdarima`	1.8.0
`statsmodels`	0.12.2

Table 1: Python packages and their versions used for this paper

Link to GitHub

The tentatively complete Python code for this project and copies of the datasets can be found on GitHub via the following link: https://github.com/jadonc121/MS-Statistics-Project

5 Methods

This section is devoted to outlining the mathematical theory of the models used in this project. For the sake of brevity, each model will be defined as succintly as possible. In general, many of the models used have multiple alternative versions, characterized by either different choices of hyperparameters or the inclusion of additional terms. Since this project seeks to compare these models generally, only the most basic version of each model is defined and implemented. Following each model description is the specific parameterization used in this paper. That is, each model is first defined, then the specific hyperparameters and how they were selected are outlined. The models are organized by method class, then this section concludes with the definitions of the benchmark models and the performance metrics.

For the following model descriptions, let Y_t be a time series of length T for $t = 1, ..., T$ and let $\hat{Y}_{T+1}$ be the next predicted value in the series. In general, it's common to split a known time series up into training and test sets, where the training set consists of the first n values in the series, $Y_1, ..., Y_n$, and the test set consists of the remaining $T - n$ values in the series, $Y_{n+1}, ..., Y_T$. The length of the test set usually defines the length of the forecast horizon. The model is trained (i.e., parameterized) on the training set, then the predictions, $\hat{Y}_{n+1}, ..., \hat{Y}_T$, are produced over the forecast horizon. The error of the forecasted values can be measured by comparing them to the true observed values in the test set, $Y_{n+1}, ..., Y_T$, using a loss metric.

5.1 Classical Statistical Methods

This method class consists of the classical statistical models and is defined by linearly parameterized functions trained using regression techniques, such as OLS. The exponential smoothing (ETS) and autoregressive integrated moving average (ARIMA) models are themselves large classes of models, so particular detail is spent describing their general structures.

Exponential Smoothing

The exponential smoothing (ETS) class of models represents models that define the next predicted value as the weighted average of the previous values in the series, with the weights decaying exponentially in value as their corresponding observations get older[19]. This class of models can be broken down into three components: error (E), trend (T), and seasonality (S). Each of these components can take on various values that define how the exponential smoothing equation should look. In general, the components can take three values: None (N), Additive(A), or Multiplicative(M). For example, simple exponential smoothing (SES) is a common, fairly simple ETS that has no trend or seasonality, but does include additive errors, so its ETS model name would be ANN. There are additional values that the trend component can take, such as damped

versions in which the trend is "damped down" over time to converge to a constant, but these were omitted from this project since the model assumptions did not match the time series' behaviors.

Four total ETS models were implemented in this project. Two models compared the effects of an additive trend component versus a multiplicative trend component (i.e., T = A and T = M), while the other two did the same but with an additive seasonal component (i.e., S = A). All four of the ETS models assumed additive errors (i.e., E = A), where the error terms are assumed to be normally distributed with zero mean and variance equal to the variance of the given data, $\epsilon_t \sim N(0, \sigma^2)$. A summary of the ETS models used in this paper and their acronyms is provided below in Table 2. ETS models with multiplicative seasonality were found to not fit the datasets in this paper and were not included.

ETS	Error	Trend	Seasonality
AAN	Additive	Additive	None
AMN	Additive	Multiplicative	None
AAA	Additive	Additive	Additive
AMA	Additive	Multiplicative	Additive

Table 2: ETS Models and Acronyms

ETS - AAN: Holt Linear Additive

The first ETS model is known as the Holt linear trend model with additive trend and was developed as a modification to the SES to handle series with a trend component[17]. There is an additive trend component and no seasonality, so this model's ETS acronym is AAN. This model can be decomposed into what is called a level equation, ℓ_t, and a trend equation, b_t, each with its own smoothing parameter. Using the training data, the model finds values for the smoothing parameters that minimize its loss function, and then uses these values to make forecasts for a given number of time steps in the future.

$$\begin{aligned} Y_t &= \ell_{t-1} + b_{t-1} + \epsilon_t \\ \ell_t &= \ell_{t-1} + b_{t-1} + \alpha\epsilon_t \ \text{ (level equation)} \\ b_t &= b_{t-1} + \beta\epsilon_t \ \text{ (trend equation)} \end{aligned} \tag{1}$$

where $0 < \alpha < 1$ and $0 < \beta < 1$ are the smoothing parameters for the level and trend equations, respectively. Note that by expanding the ℓ_{t-1} term in the first equation, we arrive at:

$$Y_t = Y_{t-1} + b_{t-1} + (\alpha - 1)\epsilon_{t-1} + \epsilon_t \tag{2}$$

which makes the model more intuitive. The current value of the series, Y_t, is a function of its previous value, the trend equation, and a weighted sum of previous errors. (Simliar relationships can be found in the

other ETS models.) This model performs well for series that have linear trend but no seasonality.

ETS - AMN: Holt Linear Multiplicative

The next ETS model is known as the Holt linear trend with multiplicative trend. It is similar to the AAN model above except that it assumes the trend grows multiplicatively instead of linearly. Its ETS acronym is AMN and the general formula is outlined in Equation 3 below. The smoothing parameters have the same purpose here as they do in the AAN model above.

$$\begin{aligned} Y_t &= \ell_{t-1} b_{t-1} + \epsilon_t \\ \ell_t &= \ell_{t-1} b_{t-1} + \alpha \epsilon_t \quad \text{(level equation)} \\ b_t &= b_{t-1} + \beta \frac{\epsilon_t}{\ell_{t-1}} \quad \text{(trend equation)} \end{aligned} \tag{3}$$

where $0 < \alpha < 1$ and $0 < \beta < 1$ are the smoothing parameters for the level and trend equations, respectively. Note that the trend equation is now a function of the level equation. This model is useful for series with exponential growth and no seasonality.

ETS - AAA: Holt-Winters Additive

The first seasonal ETS model is a modified version of the Holt linear model called the Holt-Winters additive model. It assumes additive (linear) trend and encorporates additive seasonal variation in the time series. It has a similar decomposition as the AAN model but with an additional component corresponding to the seasonal property of the series. The seasonal equation, s_t, has the fixed hyperparameter, m, which corresponds to the seasonal period (e.g., 24 hours, 7 days). The model acronym is AAA and the general formula is shown below in Equation 4. It has the same two smoothing parameters as before, α and β, but includes a third, γ, corresponding to the seasonal equation.

$$\begin{aligned} Y_t &= \ell_{t-1} + b_{t-1} + s_{t-m} + \epsilon_t \\ \ell_t &= \ell_{t-1} + b_{t-1} + \alpha \epsilon_t \quad \text{(level equation)} \\ b_t &= b_{t-1} + \beta \epsilon_t \quad \text{(trend equation)} \\ s_t &= s_{t-m} + \gamma \epsilon_t \quad \text{(seasonal equation)} \end{aligned} \tag{4}$$

where $0 < \alpha < 1$, $0 < \beta < 1$, and $0 < \gamma < 1$ are the smoothing parameters for the level, trend, and seasonal equations, respectively. This model is commonly used for series in which the seasonal fluctuations have a constant period of recurrence.

ETS - AMA: Holt-Winters Multiplicative

The last ETS model is the same Holt-Winters model described above (AAA) but with multiplicative trend. It can also be seen as the Holt linear with multiplicative trend (AMN) with additive seasonality. It also assumes that the trend grows exponentially. It acronym is AMA and the general formula is shown below in Equation 5.

$$\begin{aligned} Y_t &= \ell_{t-1} b_{t-1} + s_{t-m} + \epsilon_t \\ \ell_t &= \ell_{t-1} b_{t-1} + \alpha \epsilon_t \quad \text{(level equation)} \\ b_t &= b_{t-1} + \beta \frac{\epsilon_t}{\ell_{t-1}} \quad \text{(trend equation)} \\ s_t &= s_{t-m} + \gamma \epsilon_t \quad \text{(seasonal equation)} \end{aligned} \tag{5}$$

where $0 < \alpha < 1$, $0 < \beta < 1$, and $0 < \gamma < 1$ are the smoothing parameters for the level, trend, and seasonal equations, respectively. This model is used for series with exponential growth but with regular seasonal variation.

For the two seasonal ETS models used in this paper (AAA and AMA), the seasonal period hyperparameter, m, was fixed at 24. This is because the observations are hourly and this might pick up on any daily seasonality. Larger seasonal periods, such as yearly ($m = 24 * 365$) were not considered due to a lack of computational resources to run such models.

Autoregressive Integrated Moving Average

The autoregressive integrated moving average (ARIMA) models are a large class of models that decompose time series into autoregressive (AR) and moving average (MA) components. They are robust and versatile and can even be modified to handle seasonality in a series. They have many terms and components which will be explained in the following sections, building up to a holistic model.

The first component is the autoregressive (AR) component, which can exist as a model on its own. It assumes that the next predicted value in a series is explained by a linear combination of its previous, lagged values, where the weights of those previous values are detetmined by a typical OLS method. It is parameterized by the parameter, p, which defines how many lagged terms are used to predict the next value. So an AR(1) uses its previous value to form a prediction, an AR(2) uses its previous two terms, and so forth. The general equation for an AR(p) model is shown below:

$$T_t = \mu + \phi_1 Y_{t-1} + ... + \phi_p Y_{t-p} + \epsilon_t \tag{6}$$

where μ is a constant, ϕ_i, for $i = 1, ..., p$ are the AR coefficients determined by regression, and $\epsilon_t \sim N(0, \sigma^2)$ is the remaining stochastic error that is unaccounted for. The assumption about the distribution of the error terms is inherited from the traditional normal error regression model. AR models are useful when there exists a high amount of autocorrelation in the series, which occurs when a series is highly correlated with lagged versions of itself. The idea is that, since there is high autocorrelation, this information can be extracted and used to make predictions.

The next component is the moving average (MA) component, which can also exist as its own functional model. MA models assume that the next predicted value is explained by a linear combination of previous forecast errors. However, these previous forecast errors cannot be observed, so it is not a regression in the traditional sense. MA models are not to be confused with the moving average smoothing models in which the next predicted value is simply the weighted average of a moving "window" of previous values. Instead, previous error values are used as the explanatory variables. A MA model is parameterized by the parameter q, which defines how many previous error values to consider. The general equation for an MA(q) model is shown below:

$$Y_t = \mu + \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} \tag{7}$$

where $\epsilon_t \sim N(0, \sigma^2)$, while the other ϵ_i, for $i = t - 1, ..., t - q$ are the recorded errors from previous forecasts. MA models are useful when previous variations in a series affect current values (e.g., financial data).

Since the AR(p) and MA(q) models are both linear in their parameterization, they can actually be added together to form an ARMA(p, q) model that takes into account both the autoregressive and moving average nature of the time series. Thus, a general ARMA(p, q) equation can be expressed as follows:

$$Y_t = \mu + \phi_1 Y_{t-1} + ... + \phi_p Y_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} \tag{8}$$

where the coefficients are the same parameters as described in the AR and MA models above.

In time series forecasting, sometimes a transformation is applied to the time series dataset before a model is fit to it. The justification of this comes from the idea that by simplifying the series with a transformation first, certain models might form better fits and predictions. Generally, a series is given a transformation, then a model is applied and forecasted values are calculated, and then the inverse transformation is applied to the forecasted values to get them in terms of the original series. One such transformation is called differencing, where the current value of the series is substracted by the previous value. This can be visualized by the following equation:

$$y_t = Y_t - Y_{t-1} \tag{9}$$

where $\{Y_t\}$ is the original series and $\{y_t\}$ is the differenced series. The differencing transformation can actually be applied multiple times and can be parameterized by the parameter d, which indicates how many times to apply the transformation. So Equation 9 above represents a differencing of order $d = 1$. Applying a differencing of order $d = 2$ to the orginal series, we have:

$$\begin{aligned} z_t &= y_t - y_{t-1} \\ &= (Y_t - Y_{t-1}) - (Y_{t-1} - Y_{t-2}) \end{aligned} \tag{10}$$

where $\{z_t\}$ is the series of differencing order $d = 2$, $\{y_t\}$ is the series of differencing order $d = 1$, and $\{Y_t\}$ is the original series.

The main reason for differencing a time series for an ARMA model is that differencing is an example of mean reversion, where the mean value of the series is now centered around a constant value. Recall that the ARMA models assume normal errors with zero mean, so differencing actually converts the series into a set of fluctuations centered around zero, detrending it.

The ARMA model is actually a subclass of a larger family of models called the autoregressive integrated moving average (ARIMA) models, where the AR and MA components are exactly the same as an ARMA model and the integrated component, I, refers to the order of differencing, d, applied to the data beforehand. That is, ARIMA models are ARMA models with a built-in differencing transformation. The general equation for an ARIMA(p, d, q) model takes the same form as the ARMA model in Equation 8, but uses the differenced series.

ARIMA models are themselves yet another subset of a larger class of models called Seasonal ARIMA, or SARIMA. These extend the ARIMA model to be able to handle seasonality in a time series. The model gets futher complicated because there are 4 more components for the seasonal portion in addition to the main 3 components. A full SARIMA model can be parameterized by: $(p, d, q)(P, D, Q, m)$. The parameters p, d, q refer to the autoregessive, differencing, and moving average components of the ARIMA model described above. The P, D, Q parameters refer to the seasonally indexed autoregessive, differencing, and moving average components, with parameter m defining the length of the seasonal period. For example, in an AR(p=1) model, the prediction is a linear function of its previous value, but in an SAR(P=m) model, the prediction is a linear function of the value from the last seasonal period. The D component refers to how the series is seasonally differenced beforehand. That is, it's possible to difference a series by an order of d, to remove the trend, and to difference it by an order of D, to remove the seasonal fluctations. SARIMA

model equations can grow to be quite complicated, so their general form will be omitted. However, Duke University[28] and R.J. Hyndman[19] offer excellent breakdowns of these models.

For parameter selection, there exist tests of model fitness and common rules of thumb[19][28] used in practice that simplify the process. Additionally, Hyndman and Khandakar[20] have developed Python packages that use automatic model selection techniques, using the AIC and BIC criteria. In general, simple models with fewer hyperparameters are favored over ones with more parameters. In this paper, the `auto_arima` function of the `pmdarima` Python package was used to find appropriate parameters for both the ARIMA and SARIMA models, and tests of model fitness were conducted. For the sake of comparison, both datasets were given the same parameter selection: (1, 1, 1) for the ARIMA model and (1, 1, 1)(1, 1, 0, 24) for the SARIMA model. Similar to the ETS models, larger seasonal periods, $m > 24$, were not explored due to computational restraints.

5.2 Machine Learning Methods

This method class consists of the machine learning models and is primarily characterized by nonlinear functions. Each is unique in its own right and contains various alternatives according to hyperparameter selection. Recall that time series forecasting is a form of supervised learning in which the data is labeled by the timestamps. Thus, the common structure among these machine learning models is predicting the forecast labels of the time series.

Support Vector Regression

Support vector regression (SVR) is a modified form of the canonical machine learning tool, the support vector machine (SVM). A SVM is a classification algorithm that seeks to find a separating hyperplane or surface that divides a set of points into subsets according to their labels. This is a form of linear optimization called a convex quadratic program[9]. This program can be modified to switch the objective from finding a surface that *separates* the data to finding a surface that *aligns with* the data. This is known as support vector regression, since it essentially finds a surface that fits the data, similar to the OLS method. It is capable of forming predictions on unseen data points using that optimal surface.

SVR allows for the use of kernel methods. Given some tranformation, ϕ, and vectors x and y, a kernel is a function with special properties that can compute the dot product of two vectors, $\phi(x)^T\phi(y)$, without having to compute the the transformation itself[11]. These allow for the application of nonlinear surfaces, analogous to nonlinear regression. Soft-margin SVRs allow for values to be near the optimal surface, as opposed to hard-margin SVRs where a surface containing all the training points is required. Soft-margin

SVRs are parameterized by ϵ, which defines the error tolerance of the margin violations, and by C, which serves as a regularization constant. The general optimization formula for soft-margin kernelized SVR is given below in Equation 11[26].

$$\begin{aligned} \text{minimize: } & \frac{1}{2}\sum_{t=1}^{T}\sum_{s=1}^{T}(a_t - a_t^*)(a_s - a_s^*)K(x_t, x_s) + \epsilon\sum_{t=1}^{T}(a_t + a_t^*) - \sum_{t=1}^{T} y_t(a_t - a_T^*) \\ \text{subject to: } & \sum_{t=1}^{T}(a_t - a_t^*) = 0 \\ & 0 \leq a_t \leq C, \ \forall t \\ & 0 \leq a_t^* \leq C, \ \forall t \end{aligned} \tag{11}$$

where T is the number of training points, x_t are the training labels (timestamps), y_t are the response variables for points x_t, a_t and a_t^* are the nonnegative Lagrangian multipliers for points x_t, ϵ is the error tolerance of margin violations, C is the regularization constant, and $K(*, *)$ is the kernel function.

For this paper, the `SVR` class of the `sklearn` package was used, with the default parameters for $\epsilon = 0.1$ and $C = 1.0$. A polynomial kernel of degree 2 was used, with the formula given in Equation 12 below.

$$K(x_t, x_s) = (\gamma x_t^T x_s + 1)^2 \tag{12}$$

The default value for the scale parameter, γ was used, which was defined internally in the function. The degree of the polynomial kernel was chosen to be low so as to not overfit the training data. Experimentation showed that the Gaussian RBF kernel did not perform well in forecasting the test sets. Model fitness was assessed graphically.

Classification and Regression Trees

Classification and regression trees (CART) represent the machine learning class that involves decision trees. The tree is defined by a series of simple logical if statements, called nodes, that can divide each data point according to its value or properties. When applied to an entire dataset, the data gets partitioned into a set of uniquely defined groups, called leaves. For example, a tree node may ask if a data point's value is positive. If so, it'll either be sorted into a leaf with other positive points or be channeled down to the next node in the tree. Though commonly used for classification, decision trees can be used for regression. For a given time series, a decision tree regressor will decide where the nodes are to split the data by applying a loss criterion to measure the quality of each split. Graphing this process results in an approximation to the plot of the original series with a set of piecewise constant functions. If the number of leaves in the tree is

large, the regressor will be able to make many splits and approximate the data with low error but is at risk of overfitting. If the number of leaves is too low, there may not be enough splits and the model will underfit.

For this paper, the `DecisionTreeRegressor` class from the `sklearn` package was used. It was found through experimentation that the `max_leaf_nodes` constraint was the best regularizer for both datasets and was set to 20. The default loss criterion was used, defined to be the squared error. Model fitness was assessed graphically. It was also found that using the detrended data caused the models to poorly fit the training data, so the original, untransformed data was used.

K-Nearest Neighbors

The K-Nearest neighbors (KNN) model is one of the simplest to understand and implement. Primarily used for classification tasks, the KNN algorithm assigns the predicted value (or class) of a new data point by taking the average (or majority "vote") of the K nearest points to the new data point. Thus, K is the primary hyperparameter. The measure of distance is another hyperparameter, most often defined to be the Euclidean distance. For time series, the predicted value is computed as the average of the K previous points.

For this paper, the `KNeighborsRegressor` of the `sklearn` package was used. The value of K = 2000 was found to give a reasonable level of fit to the training data with the Euclidean distance set as the distance measure. Though this seems like a large value for K, setting K to be anything less than about 1500 severly overfit the training data. A value more than 2000 was averaging over too many values and underfit the training data. Model fitness was assessed graphically. It was found that using the detrended data allowed for the model to extrapolate appropriately to form reasonable forecasts.

Multilayer Perceptron

The multilayer perceptron (MLP) verges on the edge between machine learning and deep learning, where a perceptron is one of the simplest forms of a neural network. A single data value, x, enters the model from the input node. Before it is passed to the hidden layer, a linear transformation is applied by scaling it by weight, w_1, and adding a bias term, b_1. The hidden cell then applies a function transformation, $g(*)$, yielding $h = g(x \cdot w_1 + b_1)$, where h is the hidden layer's output. This is finally passed to the output node, where another linear transformation is applied by scaling it by weight, w_2, and adding a bias term, b_2. The final output defines y, where $y = h \cdot w_2 + b_2$. The output cell does not usually apply a function of its own. The weights range from 0 to 1 and determine how much of an influence that node's transformation has on the data point being passed to it. The function that the hidden cell applies, $g(*)$, is called an activation function, and is usually chosen to be a nonlinear function, such as the sigmoid, hyperbolic tangent, or the popular

recified linear unit (ReLU). As described, the network simply applies a series of function transformations to the original input point. Figure 7 to the left displays a single-neuron perceptron with a single input and a single output.

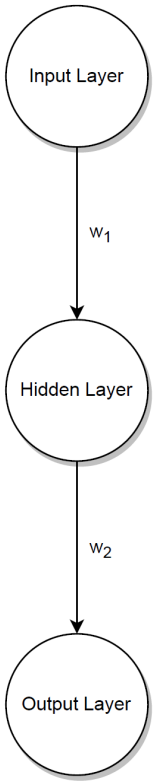

Figure 7:
A single-neuron perceptron

Usually, the hidden cells are expanded to contain an entire layer of cells, called a hidden layer. This is referred to as a fully-connected layer, where all of the cells are connected to all of the cells in the previous layer. The function transformations behave similarly except they are now vectorized. The input value, $\vec{x}$, is linearly transformed by weights, $\vec{w}_1$, and bias terms, $\vec{b}_1$, and passed to the hidden layer. Element-wise, the hidden layer applies its activation function, $g(*)$, resulting in $\vec{h} = g(\vec{x}^T\vec{w}_1 + \vec{b}_1)$. Finally, a linear transformation is applied to the hidden layer's output, resulting in $\vec{y} = \vec{h}^T\vec{w}_2 + \vec{b}_2$. In general, the inputs and outputs can be of any dimension, but for univariate regression MLPs (used for univariate time series forecasting), the input layer and output layer each have one node, where the output is a real-valued scalar representing the next predicted value in the series. Figure 8 below shows a single-layer perceptron, often referred to as a shallow multilayer perceptron (shallow because the "multi" layer is only one). In this case, the hidden layer contains 50 hidden cells, each connecting to the single input cell and single output cell.

To train an MLP, the weight vectors are intialized as random samples from a standard uniform distribution, such that $\vec{w}_1, \vec{w}_2 \in U[0, 1]$. The training data is passed through the network, yielding predictions, $\hat{\vec{y}}$. The error of these predictions against the true target values, $\vec{y}$, is measured using a given loss function (e.g., MSE, RMSE). Using the calculated errors, the model begins to update the weight vectors by adjusting them according to a process called backpropagation. In backpropagation, a given optimizer function moves backward through the network, calculating the gradients of each layer with respect to the error, determining how much that layer (and thus how much each cell) had an influence on the error. The weight vectors are then updated proportionally to how much they affected the error of the output. For example, if a particular cell had a large influence on the error, then its weight will be reduced; while a cell that had little effect on the error will not have its weight adjusted (or even have its weight increased). Using these new weight vectors, the process repeats, continually updating the network weights. Each pass through the network is called an epoch. After every epoch, the network incrementally "learns" the training data. After training, new unseen data provided by the test set is given to the network and a forecast is computed.

If the network did not learn the training data well enough, then it will not perform very well on the unseen test data. Conversely, if the network learns the training data too well, then it will be overfit and will not perform well on the test data. The goal is to construct the network in such a way that it can properly learn the training data so that it may generalize well on unseen test data. The problem for the practitioner is to construct and train the correct neural network, which is dependent on the hyperparameter selection. Neural network hyperparameters define network structure (e.g., number of hidden layers, number of cells per layer), activation function (e.g., sigmoid, ReLU), loss metric (e.g., MSE), optimizer (e.g., stochastic gradient descent, adaptive momentum estimation), and regularization techniques (e.g., dropout, normalization, gradient clipping). For the sake of brevity, the methods by which to select and tune neural network hyperparameters will not be outlined in this paper. Where necessary, the hyperparameter selections used for the MLP and deep learning models in this paper will be briefly described. Refer to the Resources section for more information on neural networks.

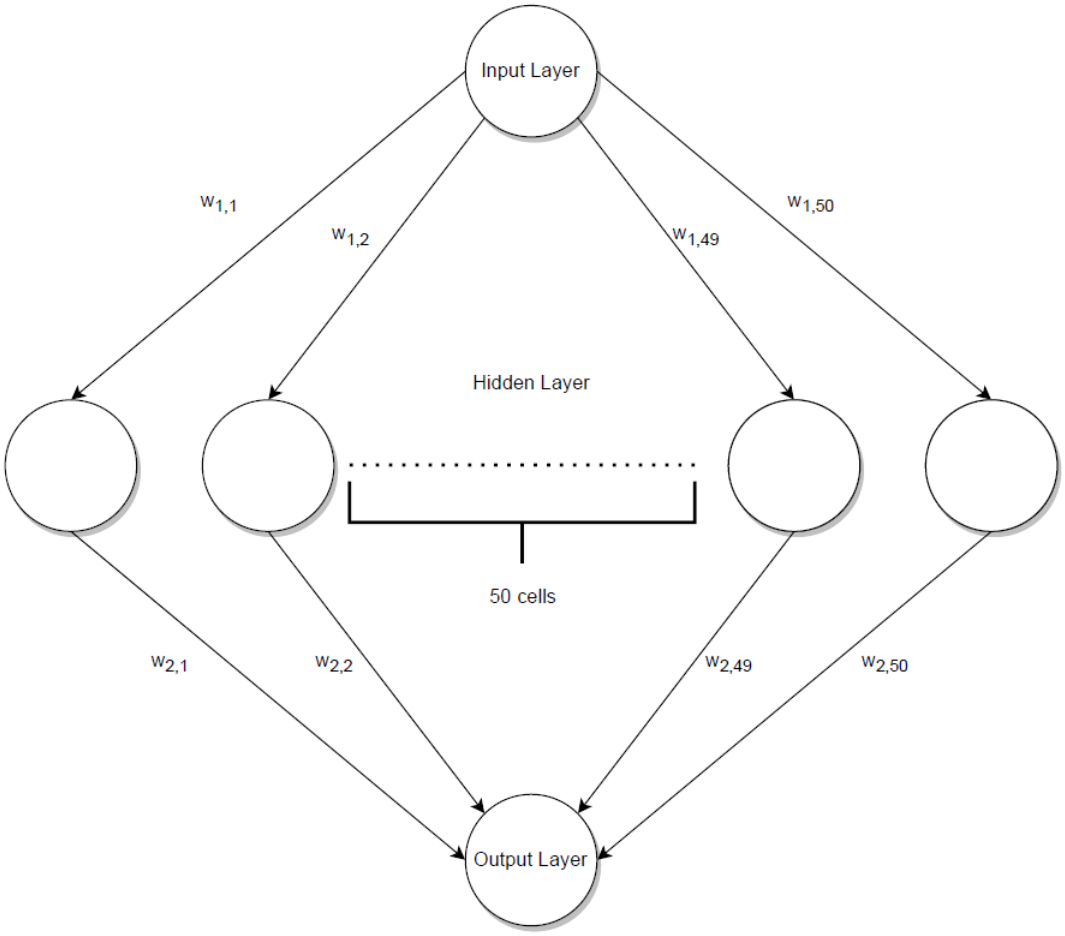

Figure 8: A single-layer perceptron (MLP)

The shallow MLP shown in Figure 8 above was the one implemented in this paper. It was built using the `Sequential` class of the deep learning package `keras`. It was defined to have a single, fully-connected (`Dense`) hidden layer of 50 cells. A neural network is considered "deep" if it has at least three hidden layers. Thus, this MLP model was considered a machine learning model due to its single hidden layer. The loss function chosen was to be the mean squared error (MSE), the activation function was chosen to be the recified linear unit (ReLU), and the optimizer was chosen to be adaptive momentum estimation (ADAM). These hyperparameters were chosen because they have been shown to perform well for any general network[11][12]. Stochastic gradient descent (SGD) is the optimizer that provides the most accurate training at the cost of performing very slowly. However, the ADAM optimizer has been shown to speed up the training without losing much accuracy[11]. The MLP was trained for 20 epochs which was experimentally chosen.

Additionally, due to the randomness involved in the initialization process, it was observed that the same MLP network with the same hyperparameter selection was providing wildly different forecasts when ran multiple times. This is likely due to the parameter space being nonconvex, resulting in the optimizer settling in local minima, as opposed to the optimal minimum value. Thus, a short grid search over various random states (i.e., seeds) was conducted for each dataset and the most accurate model was selected as the final result. This has the added benefit of making the code reproducible.

As a precaution to overfitting, the training loss (MSE) was recorded during the training process over each epoch. A custom accuracy metric was designed to track the accuracy of the predictions against the true values throughout training. If a predicted value, $\hat{\vec{y}}_t$, was within a given percent error tolerance, ϵ, of the true observed value, $\vec{y}_t$, then this was considered an accurate prediction and was labeled as "1"; "0", otherwise. After each epoch, the sum of accurate predictions was divided by the total number of points in the training set to yield an accuracy percentage. Typically, such binary accuracy metrics are reserved for classification models, where the predicted class either is or is not correctly predicted; however, it seemed justified to include an accuracy metric to assess overfitting in the networks. A properly fit model should exhibit a downward trend in training loss and an upward trend in accuracy over the epochs, with each metric appearing to converge. An example of what this behavior should look like for a successfully trained network is provided in Figure 9 below. As shown, over an interval of 20 epochs, the training loss drops and converges to a stable value, while the accuracy percentage rises and converges to stable value. These indicate that the network has found a minimum in the parameter space and has learned the training data. For all of the MLP models, the percent error tolerance, ϵ, chosen for the training accuracy was 5%.

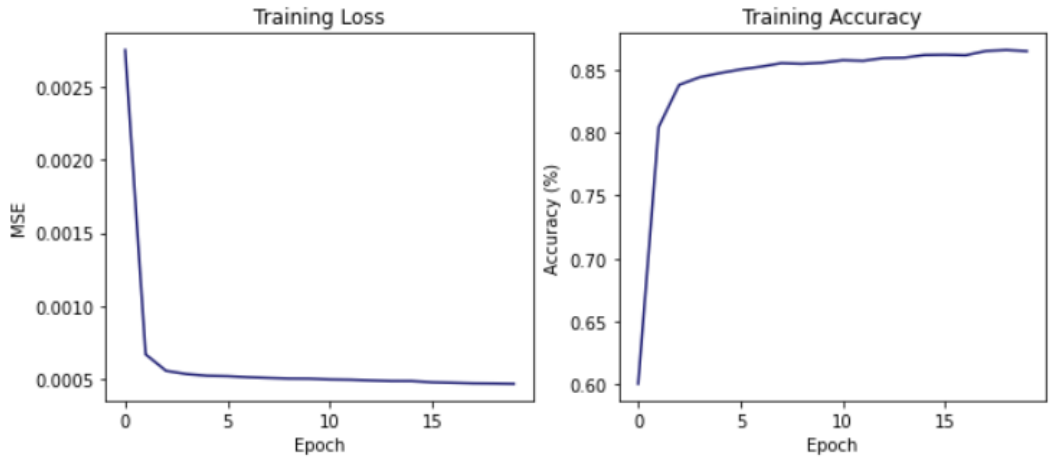

Figure 9: Example of training loss and accuracy plots

5.3 Deep Learning Methods

This method class consists of three sequential deep learning models: the recurrent neural network (RNN), the long short-term memory neural network (LSTM), and the gated recurrent unit neural network (GRU). The LSTM and GRU are special variants of the simple RNN, where their differences are described in their respective sections below. Each is considered a deep network because they were built with three hidden layers. The general structure of each follows directly from the discussion of MLP network above. Figure 10 below shows a deep neural network with three fully-connected hidden layers. This structure is simply an extension from the shallow MLP network displayed in Figure 8.

One of the primary differences between RNNs and MLPs is that MLPs are feed-forward neural networks (FFNNs), where the input data flows in one direction through the network until it reaches the output layer. That is, each layer (and each cell) sees the data point once for each pass through. RNNs, on the other hand, are a special class of neural networks in which feedback loops are built into their structure. Each hidden layer recieves new data points from the previous layer as an initial input. Then, in the next iteration, it receives its own previous output as well as the next iteration's new data point. This idea of recurrence gives the networks a form of short-term memory that makes them well designed for sequential data (i.e., time series). They attempt to pick up on nonlinear patterns in the series by seeing how previous values affect future ones. These feedback loops are depicted as the bold arrows on the side of the hidden layers in Figure 10. They are bolded to signify that the feedback loop applies to *each* cell in the hidden layer, not just the ones on the edges.

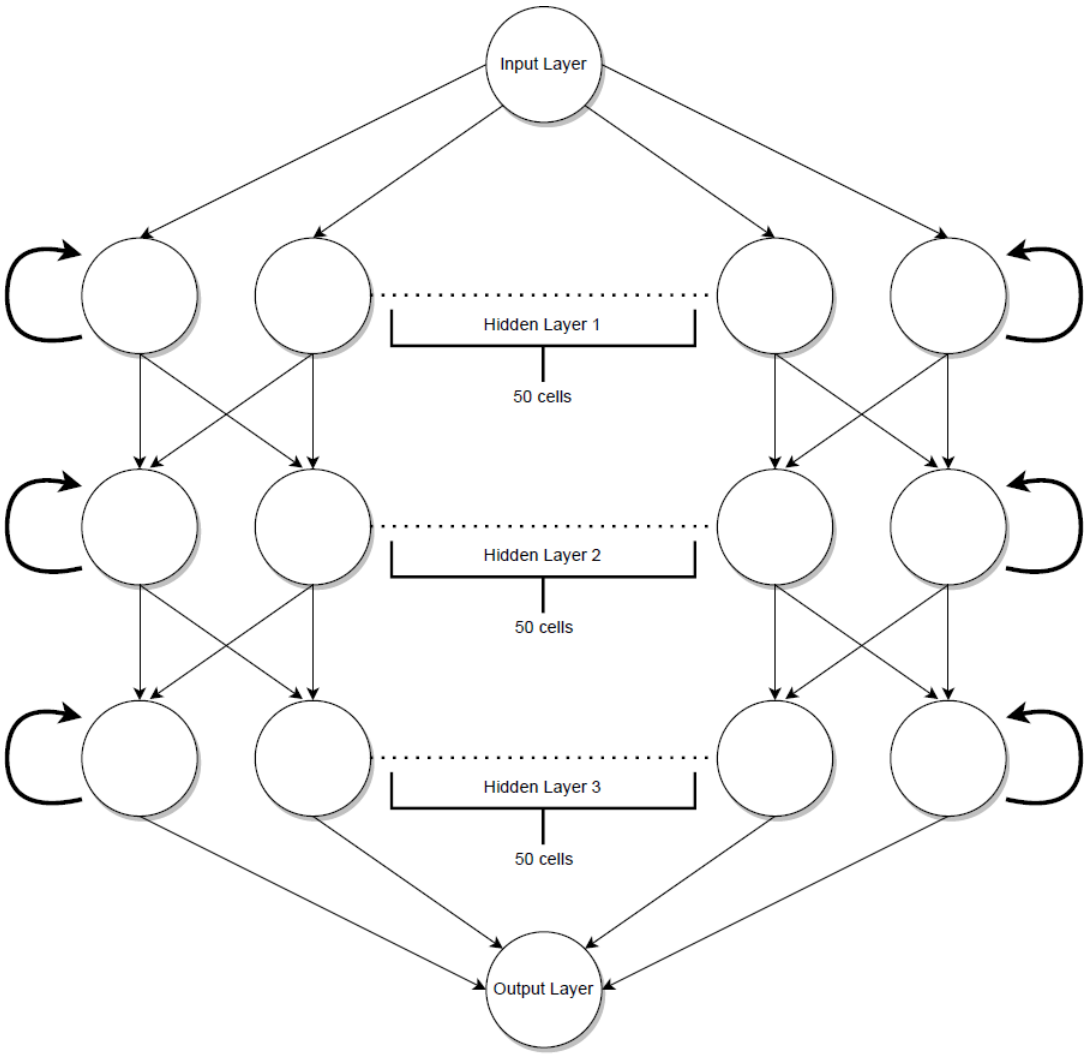

Figure 10: A deep recurrent neural network (RNN)

RNNs are trained much the same way as MLPs, where linear transformations are applied in between each layer by the weight and bias vectors, and each hidden layer applies its own nonlinear transformation. However, the vector equations of this process are less intuitive since they are now processed through time. Naturally, the backpropagation algorithms also perform much more operations as they have to "unroll" the network through time to determine which cells affected the training error and when. The weights for each layer are now represented as matrices instead of vectors, since each layer has a row of vectors for each timestep.

Training deep neural networks comes with issues that may not appear in shallow networks. The two most common ones include the vanishing gradient and the exploding gradient. When the backpropagation algorithm passes backward through the network to compute the gradient of each layer and cell with respect to the training error, it is navigating a nonlinear parameter space. Sometimes this space may have very flat

"plateaus", making the gradient very small, and other times it may have steep "cliffs" or "valleys", making the gradient very large. Since the weight vectors are adjusted by a function of the gradient, if the gradient iteratively shrinks or vanishes after each training epoch, then the weight vectors will not update after each iteration and the network will not learn the training data. Conversely, if the gradient explodes to a large value, the weight vectors will be dramatically over or under adjusted, also causing the network to improperly learn the training data. Thus, the proper selection of hyperparameters is crucial to network construction. The choice of optimizer and its learning rate are the most common remedies for this problem[11][12]. The learning rate is the rate or step size by which how much the gradient should adjust the weight vectors. Traditional optimizers such as stochastic gradient descent usually perform well, but tend to get stuck in local optima and be computationally intensive, making training long, inefficient, and prone to error. There exist newer classes of optimizers called adaptive optimizers that attempt to improve training speed by navigating the parameter space using techniques that adjust the gradient after each iteration; these include adaptive momentum estimation (ADAM), Nesterov-accelerated adaptive momentum (NADAM), and AdaMax (a variant of ADAM). Moreover, scaling the data given to a network can dramatically improve training time and avoid numerical overflow issues.

Regularization techniques are also commonly applied to ensure the deep network is properly fit and not overfit. Scaling the outputs of each hidden layer can constrict their range and keep the network from accidentally forming large prediction errors, which in turn can keep the error gradients from exploding. Regularization techniques involve anything that attempt to keep the network from overfitting the training data. These include (but are certainly not limited to) layer normalization, weight initialization choices, gradient clipping, dropout, and traditional ℓ_1-norm and ℓ_2-norm regularization used in regression settings. Layer normalization scales the outputs of each layer by normalizing them, making the data numerically easier to process, affecting how quickly the model backpropagates. Weight initialization techniques, such as Glorot and He initialization, guarantee that the variance of the input weight vectors is equal to the variance of the output weight vectors for any given layer. This has been shown to stabilize the gradient and avoid the exploding gradients problem[11]. Dropout involves randomly "dropping" a number of cells in a given layer (adjusting their weights to zero) during a training iteration, forcing the network to learning by other cell-to-cell connections, curbing overfitting. Finally, traditional ℓ_1-norm and ℓ_2-norm regularization can be applied by computing these metrics during backpropagation and adding them back to the error measures, creating an additional bias that can curb overfitting.

For the three RNNs outlined in this section, all of these hyperparameter selections were experimentally tested and chosen. All three models were given scaled data using the `MinMaxScaler` from the `sklearn` package. It was found that the structure shown in Figure 10, with three hidden layers of 50 cells each,

generalized best for all three networks. The default activation for RNNs was selected, the hyperbolic tangent, because other, nonsaturating activation functions such as the ReLU have been shown to not perform as well for RNNs[11]. The AdaMax optimizer with a lower-than-default learning rate of 0.0001 was selected in addition to a gradient norm clipping value of 1.0 (the norm of the gradients cannot exceed 1.0). Finally, He initialization was applied to the weight vectors for each hidden layer as a precaution to the exploding gradient problem. It was found through testing that dropout and ℓ-norm regularization techniques did not improve the performance for these models and datasets. Random-state dependence was also observed for the RNNs as was in the MLP models, so a short grid search was conducted over a set of random states (i.e., seeds) and the best performing seeds were chosen to include in this paper. Furthermore, the same loss metric (MSE) and custom accuracy metric ($\epsilon = 5\%$) described in the MLP section above were used to asses overfitting after training for 20 epochs. Model fitness was assessed graphically, similar to the methods described for Figure 9.

All three RNNs were constructed using the `Sequential` class from deep learning package `keras`. The hidden layer cells were selected to be the `SimpleRNN`, `LSTM`, and `GRU` classes for the models, respectively. In the following sections, since the network structure and hyperparameter selection was the same across all three networks and both datasets, the model structures will not be discussed. The networks only differ in the type of layers they include (e.g., simple RNN, LSTM, or GRU), so a brief breakdown of how each network cell operates will be provided. Major differences between the three types of cells will be highlighted.

Recurrent Neural Network

A recurrent neural network (RNN) was designed to handle sequential data by adding feedback loops into their structure, creating a form of short-term memory. This structure attempts to find sequential patterns in the underlying series that feed-forward neural networks (FFNNs) are not built to handle. A simple RNN cell takes as inputs new data, x_t, and its own previous output from the previous timestep, h_{t-1}. It then applies its activation function, typically the hyperbolic tangent, *tanh*, and provides the output for the current timestep, h_t. This process for an RNN cell is visually represented in Figure 11 above.

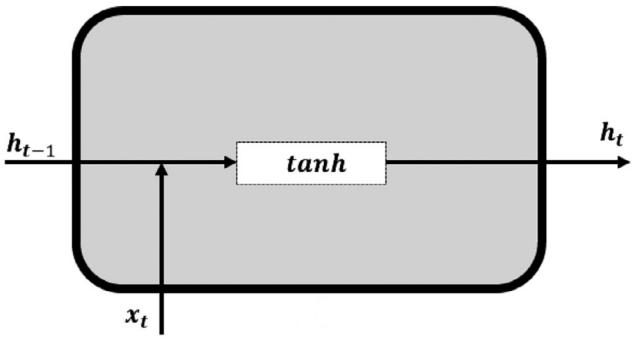

Figure 11: Recurrent Neural Network (RNN) Cell
Image Credit: ArunKamar et al. (2021)[2]

A major disadvantage of a deep RNN is that its short-term memory is *very* short, and earlier information

in the sequence is relatively lost by the time the network sees the latter part of the sequence. When backpropagating, the earlier timesteps of the upper layers hardly have an effect on the training error and the latter timesteps of the lower layers are overweighted.

Long Short-Term Memory Neural Network

The long short-term memory (LSTM) RNN attempts to modify the simple RNN by extending its short memory such that earlier sequence information get propagated through the latter parts of the network. This is done by adding another feature, called cell state, which is passed through the network. It serves as the memory of the network. The LSTM cell is much more complicated then a simple RNN cell by the addition of four logic gates: forget gate, input gate, update gate, and output gate. These control how information is stored, transformed, and transferred between timesteps. The four-step process is summarized below.

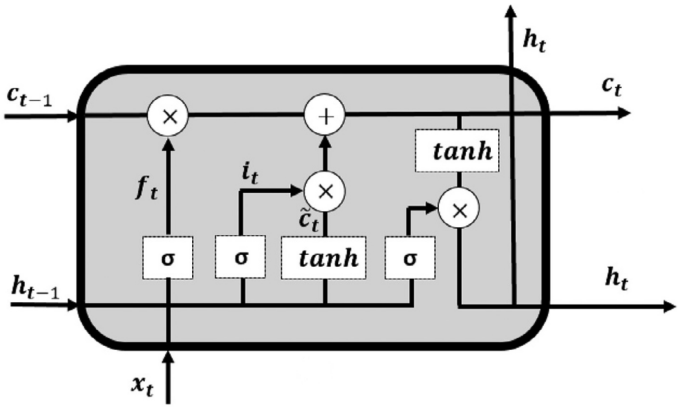

Figure 12: Long Short-Term Memory (LSTM) Cell
Image Credit: ArunKamar et al. (2021)[2]

The forget gate, f_t, receives as inputs new data, x_t, and its output from its previous timestep, h_{t-1}, and decides whether or not to forget or keep this information by passing it through the sigmoid activation function.

The input gate, i_t, takes as inputs new data, x_t, and its output from its previous timestep, h_{t-1}, and decides what information should be passed into the cell, transformed by the sigmoid activation function.

The update gate, c_t, has a two-step process, whereby it first takes as inputs new data, x_t and the previous timestep output, h_{t-1}, and passes them through the *tanh* activation function, computing $\tilde{c}_t$. Second, it updates the cell state, c_t, as a weighted sum of the previous cell state, c_{t-1}, and $\tilde{c}_t$, given by $c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t$. Notice that the weights are the outputs from the input gate and forget gate. The update gate determines what information should continue to be held in memory and carried to future timesteps.

Finally, the output gate, o_t, takes as inputs new data, x_t and the previous timestep output, h_{t-1}, and passes them through the sigmoid activation function, computing o_t. It then scales this output by the hyperbolic tangent of the current cell state and sets this as the new cell output, given by $h_t = o_t \cdot tanh(c_t)$.

This process for a LSTM cell is visually represented in Figure 12 above, where the $+$ and $\times$ represent the addition and multiplication operations used in the update and output gates. The LSTM improves the simple RNN by the addition of cell states that carry information over longer timesteps, but at the cost of

being more computationally intensive and making for longer training times.

Gated Recurrent Unit Neural Network

The gated recurrent unit (GRU) RNN serves as a simplified version of the LSTM, developed to reduce training time while still preserving long-term memory. The primary differences between an LSTM cell and a GRU cell is that the GRU (1) combines the forget gate and input gate into a single reset gate, and (2) combines the cell state and cell output into one state by using one update gate.

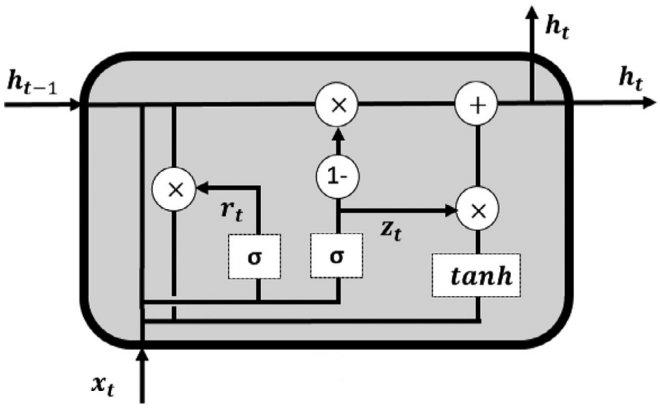

Figure 13: Gated Recurrent Unit (GRU) Cell
Image Credit: ArunKamar et al. (2021)[2]

First, the GRU computes the reset gate, r_t, which takes as inputs new data, x_t and the previous timestep output, h_{t-1}, and applies the sigmoid activation function. It then computes how much information to remember, stored in $\tilde{h}_t$, which takes in x_t, h_{t-1}, and $\tilde{h}_t$ and applies the *tanh* activation function.

Next, the GRU computes the update gate, z_t, which takes as inputs new data, x_t and the previous timestep output, h_{t-1}, and applies the sigmoid activation function. It then updates the output value, given by $h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t$. That is, the new output is a weighted sum of its previous output and the reset value.

The process for a GRU cell is visually represented in Figure 13 above, where the + and × represent the addition and multiplication operations used in the reset and update gates.

5.4 Benchmarks and Performance Metrics

Benchmark Models

In order to properly compare the performance metrics of the above models, a benchmark model was introduced known as Naïve 1. This is the same flat forecast model used in Madden & Tan (2007)[23]. The model simply assumes that the next forecasted value is the previous value, resulting in a constant line over the prediction interval set to be the very last value in the training set. This represents a low-information model that extrapolates almost nothing from the training set, so a "good" model could be defined as one that outperforms this benchmark. Recalling the discussion above in Comparing Across Classes, Makridakis et al., (2018)[24] showed that nearly all of the machine and deep learning models performed worse than Naïve

2, the seasonal version of Naïve 1, making this a reasonable selection for a benchmark. Figure 14 below displays the forecast graphs of the Naïve 1 benchmark for both datasets over each train-test split, where the dark blue lines are the training sets, the grey lines are the test sets, and the red lines are the forecasted values.

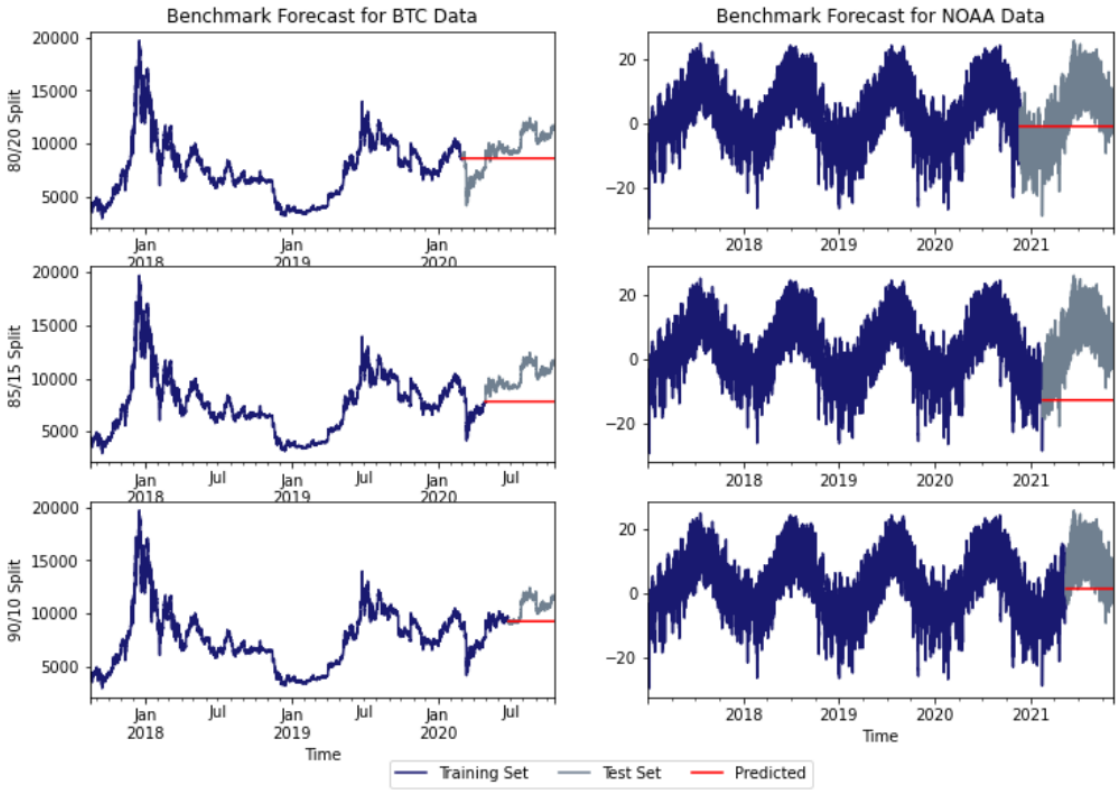

Figure 14: Forecast graphs for the benchmark model, Naïve 1

Accuracy Metrics

For the following metrics, let $\vec{y} = \{y_1, y_2, ..., y_N\}$ be the actual values and $\hat{\vec{y}} = \{\hat{y_1}, \hat{y_2}, ..., \hat{y_N}\}$ be the forecasted values, where N is the length of the series.

The first measure of accuracy is the root mean-squared error (RMSE). This is a common metric used in forecasting and has the benefit of being in the same units as the underlying data. It can also be interpreted as a measure of "fitness", of how well the forecasted graph matches the true graph. Due to its quadratic term, it has the tendency to give higher weight to large errors and smaller weight to tiny errors, making it sensitive to outliers.

$$RMSE(\vec{y}, \hat{\vec{y}}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2} \tag{13}$$

The next metric is the mean absolute error, which is the average absolute difference between the actual values and the predicted values. MAE is also in the same units as the underlying data and does not place higher weight on larger differences, like RMSE. It assumes errors are linearly related, e.g., an error of 10 is twice as bad as an error of 5.

$$MAE(\vec{y}, \hat{\vec{y}}) = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{14}$$

Finally, the symmetric mean absolute percent error (sMAPE) is used, which is another popular metric used in the machine learning literature. It is expressed as a percentage, so it is scale-independent. It removes any asymmetry concerning over- and under-estimates (like its predecessor, MAPE), but it has the disadvantage of being unstable when both the actual and predicted values are close to zero.

$$sMAPE(\vec{y}, \hat{\vec{y}}) = \frac{1}{N} \sum_{i=1}^{N} \left| \frac{y_i - \hat{y}_i}{y_i} \right| \tag{15}$$

Efficiency Metrics

To measure the efficiency of each model, the run time in seconds was chosen as the measure of computational complexity (CC). Run time here is defined as the total time the model takes to fit the training data and form the predictions over the forecast horizon. It should be noted that computational time is being used here as a proxy for computational complexity, where the assumption is that a model with longer computational time is more complex. There may exist more fitting metrics for complexity, likely in the field of information theory, such as Shannon's entropy, or information criteria, such as AIC or BIC. The exploration and comparision of these alternative metrics is left to future research.

Below is a table displaying the accuracy and efficiency metrics for the benchmark models. A "good" model would be one that could outperform these metrics.

Dataset	RMSE	MAE	sMAPE	Run Time (in seconds)
NOAA	10.06	8.21	86.36	0.001
BTC	1892.77	1624.37	9.02	0.001

Table 3: Benchmark model performance metrics

6 Results and Discussion

This section displays the results in a series of tables, where the best method will have its metric bolded for that given column. In the case of the run time measure for computational complexity, the benchmark significantly outperformed all the other models, so the second-best model will be bolded for the sake of comparison among the methods. The methods are first compared within their respective method class and then compared across method classes. Since the RMSE and MAE metrics are in terms of the original units of the underlying dataset (e.g., degrees Celcius or USD/BTC spot prices), these metrics cannot be easily compared between datasets because they have different scales. Recall that the accuracy metrics are measures of error, so a smaller value indicates better model performance.

6.1 Comparing Results Within Methods Classes

Classical Methods Results

As shown in Table 4 below, for the NOAA dataset the ARIMA model performed best for RMSE and MAE accuracy metrics, outperforming the benchmark, with the ETS - AAN and ETS - AAA (linear trend) models following closely behind, also outperforming the benchmark. The nonseasonal ETS models had the shortest run times, with the SARIMA model being slowest.

For the BTC dataset, the ETS - AMN (multiplicative trend, nonseasonal) was the clear winner across all accuracy metrics, performing second to ETS - AAN in run time (linear trend, nonseasonal). The ARIMA and ETS - AMA (multiplicative trend, seasonal) model followed closely behind in the accuracy metrics. For RMSE and MAE metrics, all models outperformed the benchmark except for the SARIMA and ETS - AAN.

Dataset	Model Name	RMSE	MAE	sMAPE (%)	Run Time (in seconds)
NOAA	Benchmark	10.06	8.21	86.36	*0.001*
	ARIMA	**9.81**	**8.02**	91.37	5.16
	SARIMA	1323.37	1136.01	97.73	94.63 (1.6 minutes)
	ETS - AAN	11.28	9.24	80.55	1.57
	ETS - AMN	32480.21	15813.01	98.85	**1.10**
	ETS - AAA	14.60	11.93	**73.04**	23.72
	ETS - AMA	2711.90	1567.11	97.00	26.21
BTC	Benchmark	1892.77	1624.37	9.02	*0.001*
	ARIMA	1496.72	1186.84	6.60	5.13
	SARIMA	4363.39	3675.87	23.56	262.44 (4.4 minutes)
	ETS - AAN	2980.57	2599.96	15.15	**0.88**
	ETS - AMN	**1492.98**	**1182.95**	**6.58**	1.17
	ETS - AAA	1736.22	1462.47	8.10	12.60
	ETS - AMA	1498.84	1191.55	6.63	15.60

Table 4: Results for the Classical Methods; See Glossary

In general, for classical models, ARIMA and linear trend ETS models performed well on seasonal NOAA data, while ARIMA and multiplicative trend ETS models performed well on volatile BTC data.

Machine Learning Methods Results

As shown in Table 5 below, for the NOAA dataset the MLP model performed best for the RMSE and MAE metrics. SVR followed closely behind for these metrics and took the lead for the sMAPE metric. SVR, CART, and MLP all outperformed, or were comparable to, the benchmark for the RMSE and MAE metrics, with all machine learning models outperforming the benchmark for the sMAPE metric. The KNN model performed worst for this dataset. The CART model had the shortest run time for this dataset.

For the BTC dataset, all machine learning models outperformed the benchmark on all accuracy metrics, with the MLP model again performing best. The CART model had the shortest run time despite having the worst accuracy metrics for this dataset.

Dataset	Model Name	RMSE	MAE	sMAPE (%)	Run Time (in seconds)
NOAA	Benchmark	10.06	8.21	86.36	*0.001*
	SVR	10.06	8.39	**62.05**	27.64
	CART	10.10	8.24	84.95	**0.03**
	KNN	20.01	16.26	74.33	2.66
	MLP	**9.68**	**8.10**	66.35	548.50 (9.1 minutes)
BTC	Benchmark	1892.77	1624.37	9.02	*0.001*
	SVR	1406.33	1053.30	5.94	11.04
	CART	1730.79	1345.97	7.45	**0.02**
	KNN	1377.73	1109.80	6.21	1.44
	MLP	**1305.04**	**1002.14**	**5.71**	337.18 (5.6 minutes)

Table 5: Results for the Machine Learning Methods; See Glossary

In general, machine learning models perform better or comparable to the benchmark on seasonal data, and outperform for volatile data. MLP appears to be the best choice for accuracy metrics and CART appears to be the best for computational efficiency.

Deep Learning Methods Results

As shown in Table 6 below, for the NOAA dataset none of the deep learning models outperformed the benchmark on the RMSE and MAE accuracy metrics, but all outperformed on the sMAPE metric, with the RNN showing the lowest sMAPE. The LSTM model outperformed the other two models in terms of RMSE, with the GRU performing worst among the three. All of the models had very long run times with the RNN having the shortest among the three.

For the BTC dataset, the RNN was the only model to outperform the benchmark on the accuracy metrics,

though only marginally. The RNN also again demonstrated the shortest run time among the three deep learning models.

Dataset	Model Name	RMSE	MAE	sMAPE (%)	Run Time (in seconds)
NOAA	Benchmark	**10.06**	**8.21**	86.36	*0.001*
	RNN	30.54	22.52	**76.74**	**10758.15 (3.0 hours)**
	LSTM	26.97	25.21	81.91	20050.75 (5.6 hours)
	GRU	28.63	26.98	82.85	24678.69 (6.9 hours)
BTC	Benchmark	1892.77	1624.37	9.02	*0.001*
	RNN	**1719.28**	**1329.74**	**7.36**	**6951.04 (1.9 hours)**
	LSTM	4481.38	4120.16	27.05	13784.16 (3.8 hours)
	GRU	2769.61	2423.16	13.99	16049.09 (4.5 hours)

Table 6: Results for the Deep Learning Methods; See Glossary

In general, the deep learning models built for this paper did not perform very accurately on either dataset and demonstrated a high demand for computational resources.

6.2 Comparing Results Across Method Classes

NOAA Dataset

Table 7 below displays the model metrics for the NOAA dataset for all method classes with the best performing metrics bolded. As shown, the MLP had the best RMSE with ARIMA as a close second. The ARIMA model had the best MAE with SVR, CART, and MLP following closely behind. SVR had the best performance for the sMAPE metric. The worst performing models were the ETS - AMN and ETS - AMA (multiplicative trend) and SARIMA.

In terms of run time, CART clearly outperformed all other models, with the nonseasonal ETS models following closely behind. Overall, ARIMA and MLP demonstrate themselves as highly accurate models for seasonal data.

Model Class	Model Name	RMSE	MAE	sMAPE (%)	Run Time (in seconds)
Benchmark		10.06	8.21	86.36	*0.001*
Classical	ARIMA	9.81	**8.02**	91.37	5.16
	SARIMA	1323.37	1136.01	97.73	94.63 (1.6 minutes)
	ETS - AAN	11.28	9.24	80.55	1.57
	ETS - AMN	32480.21	15813.01	98.85	1.10
	ETS - AAA	14.60	11.93	73.04	23.72
	ETS - AMA	2711.90	1567.11	97.00	26.21
ML	SVR	10.06	8.39	**62.05**	27.64
	CART	10.10	8.24	84.95	**0.03**
	KNN	20.01	16.26	74.33	2.66
	MLP	**9.68**	8.10	66.35	548.50 (9.1 minutes)
DL	RNN	30.54	22.52	76.74	10758.15 (3.0 hours)
	LSTM	26.97	25.21	81.91	20050.75 (5.6 hours)
	GRU	28.63	26.98	82.85	24678.69 (6.9 hours)

Table 7: Results across all method classes - NOAA dataset; See Glossary

BTC Dataset

Table 8 below displays the model metrics for the BTC dataset for all method classes with the best performing metrics bolded. As shown, the MLP model outperformed all other models in terms of accuracy. SVR, KNN, ARIMA, and the multiplicative ETS models (AMN and AMA) followed closely behind in accuracy. The worst performing models were the deep learning models and SARIMA.

In regard to run time, the CART model again had the shortest run time with the nonseasonal ETS models (AAN and AMN) following behind. Overall, the MLP model performed very accurately for the volatile BTC dataset.

Model Class	Model Name	RMSE	MAE	sMAPE (%)	Run Time (in seconds)
Benchmark		1892.77	1624.37	9.02	*0.001*
Classical	ARIMA	1496.72	1186.84	6.60	5.13
	SARIMA	4363.39	3675.87	23.56	262.44 (4.4 minutes)
	ETS - AAN	2980.57	2599.96	15.15	0.88
	ETS - AMN	1492.98	1182.95	6.58	1.17
	ETS - AAA	1736.22	1462.47	8.10	12.60
	ETS - AMA	1498.84	1191.55	6.63	15.60
ML	SVR	1406.33	1053.30	5.94	11.04
	CART	1730.79	1345.97	7.45	**0.02**
	KNN	1377.73	1109.80	6.21	1.44
	MLP	**1305.04**	**1002.14**	**5.71**	337.18 (5.6 minutes)
DL	RNN	1719.28	1329.74	7.36	6951.04 (1.9 hours)
	LSTM	4481.38	4120.16	27.05	13784.16 (3.8 hours)
	GRU	2769.61	2423.16	13.99	16049.09 (4.5 hours)

Table 8: Results across all method classes - BTC dataset; See Glossary

Comparing Between Datasets

To asses if a model perfomed better or worse on different datasets, Table 7 and Table 8 above can be compared by seeing how each model performed relative to the other models.

For the classical models, the ARIMA model performed relatively well between both datasets, making it a potentially robust model for general time series forecasting. However, its seasonal counterpart, SARIMA, did not perform well on either dataset. This could have been due to the type of seasonal period chosen for the SARIMA model. A daily seasonality was chosen ($m = 24$), though a longer seasonal period (e.g., yearly) could have a different affect on the performance of the SARIMA model.

The ETS models exhibited an interesting behavior between datasets. Whether or not the model had a seasonal component was not the factor that dictated its performance accuracy; rather, it was which type of trend component it had. For the NOAA dataset, the linear trend ETS models (AAN and AAA) performed best, with the multiplicative trend ETS models (AMN and AMA) performed worst. However, this situation is reversed for the BTC dataset. The linear trend ETS models (AAN and AAA) performed worst while the multiplicative trend ETS models (AMN and AMA) performed best. It can be hypothesized that multiplicative trend ETS models perform better for volatile, nonseasonal data and linear trend ETS models perform better for seasonal data. However, it must be noted that, similar to the SARIMA models, the seasonal period in the seasonal ETS models was chosen to be daily. It is unclear how these may perform on the NOAA dataset with alternative seasonal periods, namely yearly seasonality.

For the machine learning models, SVR and MLP had similar ranks of performance accuracy, performing very well across both datasets, making them potentially robust models for general time series data. CART performed relatively better on the NOAA dataset than its BTC counterpart, while KNN exhibited the opposite behavior, performing realatively better on the BTC dataset than its NOAA counterpart.

For the deep learning models, the RNN and GRU showed relatively similar results between datasets. The LSTM performed relatively better on the NOAA dataset than its BTC counterpart. It should be noted that these differences could be due to the particular choice of random state set for that deep learning model. Additionally, recall that the same hyperparameter selection was given to each deep learning model for the sake of comparison, but it could be justified in future research to provide different hyperparameter selections for each model (or even each dataset) such that the deep network may learn the data appropriately.

Finally, all models generally had longer run times when ran on the longer NOAA dataset versus the shorter BTC dataset, with the only major exception being the SARIMA model which ran almost 3 times as long on the BTC dataset.

Select Performance Plots

Below are the forecast plots for the best performing models from each method class.

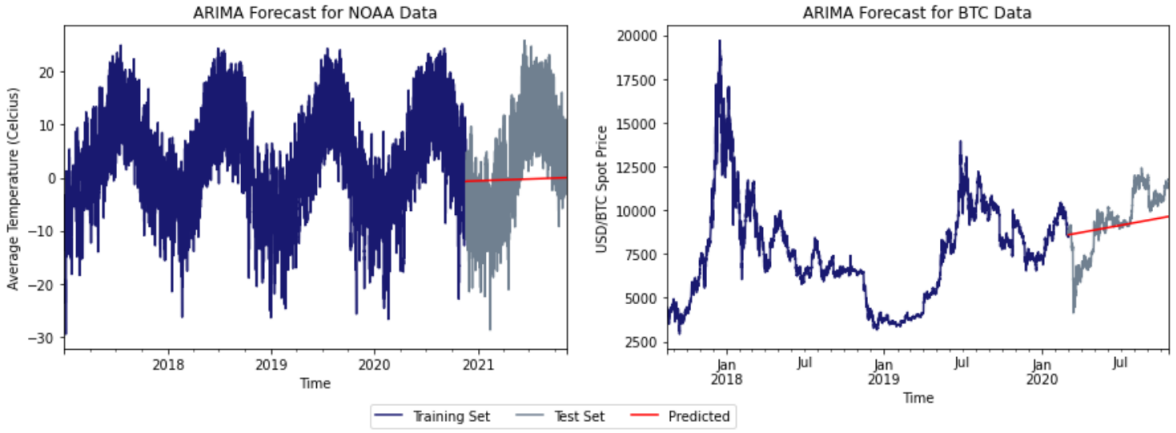

Figure 15: Forecast plots for the ARIMA model

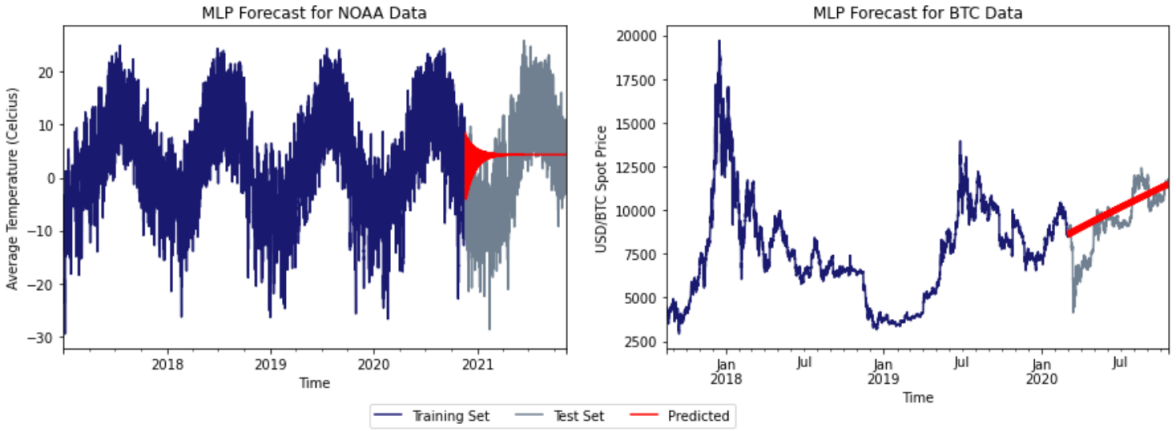

Figure 16: Forecast plots for the MLP model

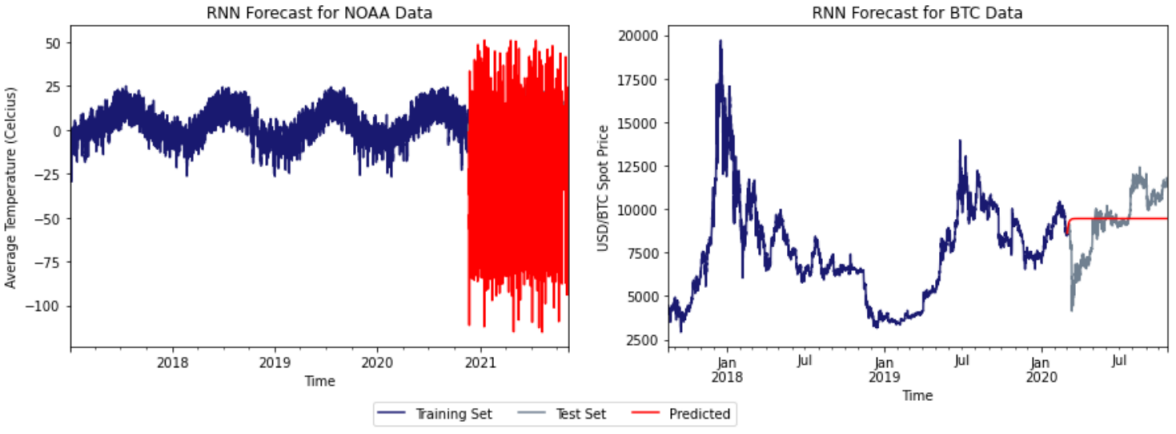

Figure 17: Forecast plots for the RNN model

Looking at the Figure 15 above, the forecast plots are shown for the ARIMA model. In general, this forecast behavior was similar across the other classical models that performed well. The forecast lines exhibit straight forecasts with slight trend.

For Figure 16 above, the MLP forecast plots are shown, each showing a different behavior. The NOAA forecast seems to oscillate and then eventually converge to a linear forecast, while the BTC forecast has a thick upward trend, also demonstrating oscillation. What is common among these graphs is the oscillation. It can be possible that the MLP network picked up on some sort of short-term seasonality (e.g., daily, weekly) and used this information in the predictions. Unlike the classical methods, these MLP forecast plots are not representative of the forecast plot behaviors of the other machine learning models.

For Figure 17 above, the RNN forecast plots are shown, each showing wildy different behavior. The NOAA forecast plot seems to exhibit a seasonality but without the proper scale. It can be possible that if this neural network were to be training for more epochs or its hyperparameter selection was tuned slightly differently, then it may have formed more accurate predictions. For the BTC forecast plots, the prediction values show a sharp jump and then a flat forecast follows. In general, this plateau behavior was observed for all of the other deep learning models, explaining why their accuracy metrics were deficient.

This plateau problem is speculated to be an exploding gradients problem, defined in detail in the Methods section. This occurs during backpropagation of the neural network, when the error-correcting gradient "explodes" to a large value and distorts the parameter values of the weight and bias vectors. Strangely, model fitness plots were generated for all of the deep learning models and they all exhibited proper model fitness as shown in Figure 9, where the training loss and training accuracy each converged to a stable level. So it could also be possible that the deep learning models rendered a degree of overfitting or underfitting of the training data. Lower learning rates, stricter gradient norm clipping, and more training epochs may remedy this problem.

6.3 Limitations and Challenges

The major limitation of this paper was a lack of computational resources to properly experiment the hyperparameter selections for the deep leaning models. The limited computational power of the PC on which these models were trained on resulted in long training times and maxed-out local memory (RAM). If more powerful computing resourses were available, then a proper grid search of the hyperparameter space could have been explored, potentially resulting in more finely tuned neural networks. The same issue applied to the seasonal classical models with a periodicity set higher than daily ($m = 24$). Yearly periodicity ($m = 24 * 365$) was attempted, but Python returned warnings stating that local memory was maximized.

These models may also benefit from improved computational resources.

It was due to these computational challenges that the results for the other two train-test splits are delayed. Once conducted, these results will be displayed in Appendix B.

It should alse be noted that the sMAPE error metric may not be the best choice for future experiments. As shown in the results tables above, the sMAPE occassionally provided misaligned results relative to the RMSE and MAE metrics. For example, the results for the ARIMA model on the NOAA dataset show an sMAPE value of 91.37%, indicating that the forecast values have a nearly imperfect accuracy score, which is an incorrect conclusion given the RMSE and MAE values and the forecast plot shown above in Figure 15. Similar performance misinterpretations could be concluded from the sMAPE metric for the CART and KNN models on the NOAA dataset. Thus, alternative percent error metrics should be considered.

7 Conclusion

7.1 Overall Summary

This paper sought to experimentally explore and compare various time series models across three different method classes: classical statistical, machine learning, and deep learning. Six classical models (from two different methods), four machine learning methods, and three deep learning methods were implemented against a naive benchmark. Three accuracy metrics (RMSE, MAE, sMAPE) and one computational complexity metric (run time) were computed for each forecast prediction. Additionally, two different datasets were used, each representing a different general pattern in time series (seasonal versus nonseasonal and volatile). Results were compared both within and among model classes, as well as between datasets. It was shown that the autoregressive integrated moving average (ARIMA) and the shallow multilayer perceptron (MLP) successfully beat the benchmark and provided accurate forecasts for both datasets. These may serve as potential robust forecasting methods for various time series patterns. It was shown that all of the machine learning models were comparable to or outperformed the benchmark on both datasets. The deep learning models performed worst but their hyperparameter selection is suspect.

In regard to computational complexity, the nonseasonal exponential smoothing (ETS), classification and regression trees (CART), and K-nearest neighbors (KNN) models performed most efficiently with the lowest run times. Run time was used as a proxy for more favorable metrics that may be found in fields such as information theory.

The contribution of this paper was comparing time series forecasting methods across method classes in terms of *both* accuracy and computational complexity metrics, a dual comparision that is scarce in the

literature. This papaer may also serve as a preliminary experiment in how certain forecasting methods perform on particular patterns in time series, namely seasonal variation versus volatility.

7.2 Further Research and Considerations

There are many areas of future research that may improve and expand the objectives of this paper. These primarily involve improving computational resources, especially for the computationally expensive deep learning neural network models. It may be crucial for modeling with deep learning models to thoroughly explore the hyperparameter space and develop an individual neural network for each time series. This requires the proper computational capacity to perform such grid searches. This paper was also limited by only using two time series datasets for comparison. Given a sufficient level of computational resources, the models in this paper could be applied to thousands of time series, which would gain the benefit of statistical implications (e.g., sample statistics, t-tests for time series grouped by pattern). Parallel processing the Python code (e.g., Spark) and/or the implementation of large GPUs may also decrease the computational burden of deep learning.

The experiments in this paper could be expanded to include additional models, such as the classical GARCH models designed to use variance in the time series as a predictor, and other machine learning models such as Gaussian processes (GP), random forests (RF), and gradient boosting trees (GBT), which have been shown to perform very well in literature.

Makridakis et al. (2018)[24] explored the effects of applying various transformations to the time series as a preprocessing method, such as differencing and log transformations. They showed that differencing resulted in poorer forecast accuracy than other transformations. Future research is needed to replicate and confirm how time series transformations affect performance, as this would provide helpful rules of thumb for the forecasting community.

Finally, combination and hybrid models offer an promising area of research in the literature. Referring back the the Comparing Across Classes section, the Makridakis et al., (2018)[24] paper included an method referred to as the "combination", or COMB, model. It was defined simply as the arithmetic mean of three other ETS models. Surprisingly, it performed fairly well and even surpassed two of its components in the accuracy rankings from shown in Figure 1. Combination models are known for their exceptional performance. Referring back to the M competitions, Makridakis et al. (2020)[25] details the results of the M4 competition from 2020. Of the 40 valid model submissions, 27 were combination methods (i.e., a weighted average of either statistical or machine learning methods), 17 were pure statistical methods, 4 were pure machine learning methods, and one was a hybrid method. Hybrid models can be broadly defined as a model that is

constructed with the components of two or more different models (e.g., statistical and machine learning, or two different deep learning methods). The results of the M4 competition concluded that the hybrid model won first place, with the other top performing models all being combination models. Indeed, the combination models out performed all but one of the pure models, with the worst performing models being either the pure statistical or pure machine learning ones. Combination models are thought to be excellent performers because they cancel out the errors of their component models, effectively decomposing and predicting the underlying time series pattern more accurately[25]. Lu et al. (2020)[22] showed that a CNN-LSTM hybrid outperformed other pure deep learning models in forecasting stock prices. Despite their performative success, this project did not include any hybrid or combination models.

8 Resources

Resources for classical statistical methods:

- Forecasting: Principles and Practice This is an excellent textbook written by R.J. Hyndman, a leading practitioner in the forecasting community. It is freely available online and details ETS and ARIMA modeling using R.
- Duke.edu Robert Nau provides a useful description of ARIMA models and the rules of thumb used in practice for hyperparameter selection

Resources for machine and deep learning methods:

- Hand-on Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition by Aurélien Géron outlines how to build machine and learning models in Python without being too math heavy. It was a valuable resource for this paper.
- Deep Learning by Ian Goodfellow et al., provides much of the mathematical theory behind deep learning.

9 References

[1] Ahmed, N. K., Atiya, A. F., Gayar, N. E., & El-Shishiny, H. (2010). An Empirical Comparison of Machine Learning Models for Time Series Forecasting. Econometric Reviews, 29(5-6), 594–621. https://doi.org/10.1080/07474938.2010.481556

[2] ArunKumar, K. E., Kalaga, D. V., Kumar, Ch. M. S., Kawaji, M., & Brenza, T. M. (2021). Forecasting of COVID-19 using deep layer Recurrent Neural Networks (RNNs) with Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) cells. Chaos, Solitons & Fractals, 146, 110861. https://doi.org/10.1016/j.chaos.2021.110861

[3] Azarafza, M., Azarafza, M., & Tanha, J. (2020). COVID-19 Infection Forecasting based on Deep Learning in Iran. MedRxiv. https://doi.org/https://doi.org/10.1101/2020.05.16.20104182

[4] Bai, Y., Xie, J., Liu, C., Tao, Y., Zeng, B., & Li, C. (2021). Regression modeling for enterprise electricity consumption: A comparison of recurrent neural network and its variants. International Journal of Electrical Power & Energy Systems, 126, 106612. https://doi.org/10.1016/j.ijepes.2020.106612

[5] Balli, S. (2020). Data Analysis of Covid-19 Pandemic and Short-Term Cumulative Case Forecasting Using Machine Learning Time Series Models. Chaos, Solitons & Fractals, 142(110512). https://doi.org/10.1016/j.chaos.2020.110512

[6] Binance. (2021). Binance Exchange Data. www.cryptodatadownload.com. https://www.cryptodatadownload.com/data/binance/

[7] Brown, R. G., & Davies, O. L. (1960). Statistical Forecasting for Inventory Control. Journal of the Royal Statistical Society. Series a (General), 123(3), 348. https://doi.org/10.2307/2342487

[8] Chowdhury, R., Rahman, M. A., Rahman, M. S., & Mahdy, M. R. C. (2020). An approach to predict and forecast the price of constituents and index of cryptocurrency using machine learning. Physica A: Statistical Mechanics and Its Applications, 551(124569). https://doi.org/10.1016/j.physa.2020.124569

[9] Deisenroth, M. P., Faisal, A. A., & Ong, C. S. (2020). Mathematics for machine learning. Cambridge; New York, Ny Cambridge University Press.

[10] Dixon, M., Halperin, I., & Bilokon, P. (2020). Machine Learning in Finance. Springer International Publishing.

[11] Géron, A. (2019). Hands-on machine learning with Scikit-Learn and TensorFlow concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc.

[12] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. The Mit Press.

[13] Granger, C. W. J., Hyung, N., & Jeon, Y. (2001). Spurious regressions with stationary series. Applied Economics, 33(7), 899–904. https://doi.org/10.1080/00036840121734

[14] Green, K. C., & Armstrong, J. S. (2015). Simple Versus Complex Forecasting: The Evidence. SSRN Electronic Journal, 68. https://doi.org/10.2139/ssrn.2643534

[15] Hewamalage, H., Bergmeir, C., & Bandara, K. (2021). Recurrent Neural Networks for Time Series Forecasting: Current status and future directions. International Journal of Forecasting, 37(1), 388–427. https://doi.org/10.1016/j.ijforecast.2020.06.008

[16] Hinton, G. E., Osindero, S., & Teh, Y.-W. (2006). A Fast Learning Algorithm for Deep Belief Nets. Neural Computation, 18(7), 1527–1554. https://doi.org/10.1162/neco.2006.18.7.1527

[17] Holt, C. C. (2004). Forecasting seasonals and trends by exponentially weighted moving averages. International Journal of Forecasting, 20(1), 5–10. https://doi.org/10.1016/j.ijforecast.2003.09.015

[18] Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. Neural Networks, 2(5), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[19] Hyndman, R. J., & Athanasopoulos, G. (2021). Forecasting: principles and practice (3rd ed.). Otexts.

[20] Hyndman, R. J., & Khandakar, Y. (2008). Automatic Time Series Forecasting: TheforecastPackage forR. Journal of Statistical Software, 27(3). https://doi.org/10.18637/jss.v027.i03

[21] Li, X., Shang, W., & Wang, S. (2019). Text-based crude oil price forecasting: A deep learning approach. International Journal of Forecasting, 35(4), 1548–1560. https://doi.org/10.1016/j.ijforecast.2018.07.006

[22] Lu, W., Li, J., Li, Y., Sun, A., & Wang, J. (2020). A CNN-LSTM-Based Model to Forecast Stock Prices. Complexity, 2020, 1–10. https://doi.org/10.1155/2020/6622927

[23] Madden, G., & Tan, J. (2007). Forecasting telecommunications data with linear models. Telecommunications Policy, 31(1), 31–44. https://doi.org/10.1016/j.telpol.2006.11.004

[24] Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2018). Statistical and Machine Learning forecasting methods: Concerns and ways forward. PLOS ONE, 13(3), e0194889. https://doi.org/10.1371/journal.pone.0194889

[25] Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2020). The M4 Competition: 100,000 time series and 61 forecasting methods. International Journal of Forecasting, 36(1), 54–74. https://doi.org/10.1016/j.ijforecast.2019.04.014

[26] MathWorks. (2022). Understanding Support Vector Machine Regression - MATLAB & Simulink. www.mathworks.com. https://www.mathworks.com/help/stats/understanding-support-vector-machine-regression.html#References

[27] National Oceanic and Atmospheric Administration. (2021). Index of /pub/data/uscrn/products/hourly02. www.ncei.noaa.gov. https://www.ncei.noaa.gov/pub/data/uscrn/products/hourly02/

[28] Nau, R. (2019). Statistical forecasting: notes on regression and time series analysis. Duke.edu. https://people.duke.edu/~rnau/411home.htm

[29] Sezer, O. B., Gudelek, M. U., & Ozbayoglu, A. M. (2020). Financial time series forecasting with deep learning: A systematic literature review: 2005–2019. Applied Soft Computing, 90, 106181. https://doi.org/10.1016/j.asoc.2020.106181

[30] Stanko, I. (2020). The Architectures of Geoffrey Hinton. In Guide to deep learning basics: logical, historical and philosophical perspectives (pp. 79–92). Springer.

[31] Svetunkov, I. (2022). Time Series Analysis and Forecasting with ADAM. In openforecast.org. bookdown (GitHub). https://openforecast.org/adam/

[32] Tang, Z., de Almeida, C., & Fishwick, P. A. (1991). Time series forecasting using neural networks vs. Box- Jenkins methodology. SIMULATION, 57(5), 303–310. https://doi.org/10.1177/003754979105700508

[33] Winters, P. R. (1960). Forecasting Sales by Exponentially Weighted Moving Averages. Management Science, 6(3), 324–342. https://doi.org/10.1287/mnsc.6.3.324

[34] Zhang, H., Nguyen, H., Vu, D.-A., Bui, X.-N., & Pradhan, B. (2021). Forecasting monthly copper price: A comparative study of various machine learning-based methods. Resources Policy, 73, 102189. https://doi.org/10.1016/j.resourpol.2021.102189

10 Appendix A: Glossary

AAA ETS model with additive error, additive trend, and additive seasonality

AAM ETS model with additive error, additive trend, and multiplicative seasonality

AAN ETS model with additive error, additive trend, and no (None) seasonality

ADAM Adaptive Momentum Estimator. An optimizer for machine and deep learning.

AI Artificial Intelligence

AIC Akaike Information Criterion. A model fitness metric.

AMA ETS model with additive error, multiplicative trend, and additive seasonality

AMM ETS model with additive error, multiplicative trend, and multiplicative seasonality

AMN ETS model with additive error, multiplicative trend, and no (None) seasonality

ANN Artificial Neural Network. A deep learning model.

AR Autoregressive. A classical statistical model.

ARIMA Autoregressive Integrated Moving Average. A classical statistical model.

ARMA Autoregressive Moving Average. A classical statistical model.

BIC Bayesian Information Criterion. A model fitness metric.

BNN Bayesian Neural Network. A machine learning model.

BTC Bitcoin. A cryptocurrency, one of the datasets used in this paper.

CART Classification and Regression Trees. A machine learning model.

CC Computational Complexity. Any efficiency metric.

CNN Convolutional Neural Network. A deep learning model.

COMB Combination model. A model defined as the average of other, simpler models.

Damped A classical statistical model, a variant of the ETS model.

DL Deep Learning

DNN Deep Neural Network. A deep learning model.

ETH Ethereum. A cryptocurrency.

ETS Exponential smoothing. A classical statistical model.

FFNN Feed-Forward Neural Network. A deep learning model.

GARCH Generalized Autoregressive Conditional Heteroskedasticity. A classical statistical model.

GBT Gradient Boosting Tree. A machine learning model.

GP Gaussian Process. A machine learning model.

GPU Graphics Processing Unit.

GRNN Generalized Regression Neural Network. A machine learning model.

GRU Gated Recurrent Unit. A deep learning model, a variant of the RNN.

Holt A classical statistical model, a variant of the ETS model.

IoT Internet of Things

KNN K-Nearest Neighbors. A machine learning model.

LSTM Long Short-Term Memory. A deep learning model, a variant of the RNN.

M1, M2, M3, M4, M5 The "M" Competitions for time series forecasting.

MA Moving Average. A classical statistical model.

MAE Mean Absolute Error. A loss metric.

MAPE Mean Absolute Percent Error. A loss metric.

MF Model Fitness

ML Machine Learning

MLP Multilayer Perceptron. A machine learning model.

MNIST Modified National Institute of Standards and Technology database. A benchmark dataset for classification models.

MSE Mean Squared Error. A loss metric.

NADAM Nesterov-accelerated Adaptive Momentum Estimator. An optimizer for machine and deep learning.

Naïve 1 A simple benchmark model. Also known as a random walk process.

Naïve 2 A simple benchmark model. Also known as a random walk process.

NLP Natural Language Processing. A deep learning application.

NOAA National Oceanic and Atmospheric Administration, one of the datasets used in this paper.

OLS Ordinary Least Squares

ReLU Rectified Linear Unit. An activation function.

RBF Radial Basis Function. A machine learning model.

RF Random Forest. A machine learning model.

RMSE Root Mean Squared Error. A loss metric.

RNN Recurrent Neural Network. A deep learning model.

SAR Seasonal Autoregressive. A classical statistical model.

SARIMA Seasonal Autoregressive Integrated Moving Average. A classical statistical model.

SES Simple Exponential Smoothing. A classical statistical model, a variant of the ETS model.

SGD Stochastic Gradient Descent. An optimizer for machine and deep learning.

SMA Seasonal Moving Average. A classical statistical model.

sMAPE Symmetric Mean Absolute Percent Error. A loss metric.

SVR Support Vector Regression. A machine learning model.

Theta A classical statistical model, a variant of the ETS model.

USD U.S. Dollar

11 Appendix B: Tables of Results

TBD: Additional tables displaying the results of the other train-test splits are delayed due to a limitation of computational resources. See the Limitations and Challenges section for more details.