Master of Science in Statistics

Dissertation: Draft #1

Online non-linear prediction of financial time-series patterns

Joel da Costa

Supervisor: A/Prof. T. Gebbie

Statistics,

Department of Statistical Sciences,

University of Cape Town, Rondebosch

2017

Abstract

We consider pattern prediction of financial time-series data. The algorithm framework and workflow is developed and proved on daily sampled closing time-series data for JSE equity markets. The input patterns are based on input data vectors of data windows pre-processed into a sequence of daily, weekly and monthly or quarterly sampled feature measurement changes (log feature fluctuations). The data processing is split into an online batch processed step where data is compressed using a stacked autoencoder (SAE) via unsupervised learning, and then batch supervised learning

is carried out using the data-compression algorithm with the output being a point prediction of measured time-series feature fluctuations (log differenced data) in the future (ex-post) from the training and validation data. Weight initializations for these networks are implemented with Restricted Boltzmann Machine (RBM) pre-training, and variance based initializations. The historical simulation is then run using an online feedforward neutral network (FNN) initialised with the weights from the online training and validation step. The validity of results is considered under a rigorous assessment of backtest overfitting (Combinatorially Symmetrical Cross Validation). Results are further used to develop a view on the phenomenology of financial markets and the value of historical data.

DSR?

Keywords: online learning, feedforward neural network, restricted boltzmann machine, variance weight initialization, stacked autoencoder, pattern prediction, JSE, non-linear, financial time series, combinatorially symmetrical cross validation, backtest overfitting

Contents

List of Figures

List of Tables

1 Introduction

This dissertation explores several ideas for the purposes of effective and valid stock price prediction, and suggests a novel framework for generating and assessing such predictions. The framework combines several concepts to this end: the training of deep neural networks, feature selection through the use of Stacked Autoencoders, online learning for stock price fluctuation prediction and current assessment techniques such as Combinatorially Symmetric Cross-Validation (CSCV). In doing so, a modular process is presented, where indivual components are decoupled and can be configured as necessary. This delivers a simplest high complexity framework which allows further adaption for the exploration of a high dimensional solution space.

add DSR?

As Bailey et al. have noted, and which is discussed more fully in 2.6, backtesting overfitting for trading strategies have become problematically widespread in financial literature. Neural networks, in their capacity as universal function approximators with few model limitations, offer an effective and appropriate methodology to generate predictions in an environment as complex as financial markets. However, the increased complexity and nature of backtest overfitting leave traditional validation methods such as hold-out or cross validation falling short. The framework presented investigates how more rigorous validation techniques can be applied to deep learning models in order to avoid such overfitting. Further validation takes place in assessing the potential profitability of the model in a live market.

The literature review in chapter 2 has a fuller discussion of work that has been done to precede the various techniques which have been implemented. A brief introduction to technical analysis in the financial sector, with it's perspective and history, is discussed and forms the basis for proposing and using the technical analysis methods throughout this paper (2.1). The literature review then moves onto covering the usage and history of Neural Networks, which has progressed enourmously in recent years. The section covers both the basic foundations, and also discusses the recent work which has resulted in the widespread use of deep learning models (2.2). This is extended to the coverage of Stacked Autoencoders, and their efficacy in data reduction for complex systems, which has lead them to be pivotal tools in deep learning models (2.3). Online learning methods are discussed in 2.4 with a coverage of both the historical basis as well as the more recent devopments which have allowed for further improvement in the algorithms. The chapter finishes off with discussing the impact of backtest overfitting and how some notable works developed by Bailey et al. have provided tools with which this may be avoided 2.6.

Chapter 3 discusses the implementation tools chosen as well as the libraries created in the process of developing the framework. The libraries offer a generic set of tools to train with highly configurable parameters in order to train and assess neural networks, including ane extensive collection of visualizations for said assessment.

Chapter 4 provides the details around the data which has been used in the paper. The chapter starts with describing how the data is processed using log feature fluctuations, and then is expanded to include the fluctuations over rolling window periods. A discussion of Geometric Brownian motion, which is used for the generation of synthetic data, is included with notes around it's characteristics and usage in financial prediction models.

Chapter 5 provides more indepth details on the algorithms and structures used to implement the framework. The structure of feedforward neural networks is discussed, as well as how they are trained using the backpropagation algorithm, and how that can be applied in a stochastic descent framework (5.2). The chapter also provides details for how these network weight initializations can impact performance, and how this can be improved with either RBM pre-training (as per 5.3) or through newer variance based techniques (5.5) - this includes the structures and training techniques used for the Stacked Autoencoders (5.4). How the CSCV techniques suggested by [1] are implemented and used to derive a Probability of Backtest Overfitting figure for the full training and testing processes is detailed in 5.6. Finally, the Money Management System (MMS), which constitutes the implementation of trading decisions and actions based on the network model predictions, is detailed in 5.7.

Where Chapter 5 discusses the functional implementations of each module of the framework, Chapter 6 is concerned with how these modules fit together to form an end to end system, from the data preparation to the output of a Probability of Backtest Overfitting (PBO) figure. This includes discussions of various considerations for individual aspects of the framework, as well as decisions around the combinations of these parts with their justifications and relevant advantages or disadvantages. In light of the configurable and modular system developed, potential alternatives and how they might be used are also discussed here.

Chapter 7 provides a brief overview of the datasets used the experiments ran, including both the synthetic and actual datasets. Visualisations for the individual asset prices are also included here.

Finally, Chapter 8 discusses the full set of results from the experiments, and offers several key takeaways:

1 Long term historical financial data is of limited use in financial model training, and predictions are served best by a recent cross sectional view.

2 RBM pre-training, a seminal development for deep learning, has potentially detrimental effects on when used for finanical time series. Variance based initializations are shown to be more effective.

3 Feature selection is possible, though the features learnt are not static through time, once again emphasising the value of recent financial data.

4 Learning optimizations

5 The framework was able to deliver returns within a successful range of the benchmark, and with only a low likelihood calculated for the PBO.

add potential line here

Further discussion around network structures, activation functions and learning optimisations are included here, with a focus on how these parameters might result in differences between results for synthetic and actual datasets.

2 Literature Review

2.1 Technical Analysis

Technical analysis is a financial analytical practice that makes use of past price data in order to identify market structures, as well as forecast future price movements. The techniques are typically objective methodologies which rely solely on past market data (price and volume). They stand in contrast to fundamental analysis, where experts will consider a companies operations, management and future prospects in order to arrive at an evaluation. The basis of much technical analysis, originally developed through Dow Theory, is the belief that stock market prices will move directionally (upwards, downwards or sideways), and that past movements can be used to determine these trends [2].

One of the primary methods in technical analysis is the use of charts in order to identify price patterns. These charts will be produced using the available market data and a known design, such as the popular candle-bar plot, which can then be compared to historical data to match it to a particular pattern. These patterns are thus indicative that the stock is likely to take on a particular price trend, or is in a particular state [2]. There is a certain amount of controversy around technical analysis, where many argue that it is contradictory to the random walk and weak form efficient market hypotheses, and as such is not valuable or useful [3]. The argument against this, is that technical analysis does not rely on past action to predict the the future, but is rather a measure of current trading, and how the market has reacted after similar patterns have occurred in the past [4]. Further, even if the analysis is unable to effectively forecast future price trends, it can still be useful to exploit trading opportunities in the market [5].

With the advent of processing power becoming cheaply available, there has been an increase in research to adapt computing techniques to technical analysis. The breadth and superhuman speed in which systems are able to perform technical analysis far outstrips what was possible before, and as such they have become the focus of competitive performance for many market participants [6]. To this end, there has been much research to apply machine learning algorithms to perform pattern recognition on stock price movements.

Financial markets have been shown to be complex and adaptive systems, where the effects of interaction between participants can be highly non-linear [7]. Complex and dynamic systems such as these may often exist at the 'order-disorder border' - they will generate certain non-random patterns and internal organisation, which can be assessed and identified, however they will also exhibit a certain amount of randomness in their behaviours, or 'chaos' [8]. Further, it's been shown that there is enough signal to reconstruct the phase space of chaotic systems from singular observations, encouraging that we might be able to perform effective technical analysis in the context of financial markets [9, 10]. As a result, trying to identify these patterns and structures is a simultaneously reasonable and notoriously difficult goal. While it is often clear in hindsight that the patterns exist, the amount of noise and nonlinearity in the system can make prediction challenging. Fittingly then, neural networks have become a popular choice for modelling within the financial markets. Due to their structure, they are able to learn non-linear interactions between their inputs and outputs, with even early research showing their ability to achieve statistically significant results, which

lends weight to the argument against the efficient market hypothesis [11].

2.2 Neural Networks

A Neural Network (NN) is a learning model which was originally inspired by the biological mechanisms of neurons in brains. The structure is essentially that of a network system, with connected nodes and edges, or neurons and weights in context. The neurons are based on the same idea as synapses as seen in the brain - where a buildup of input results in a firing of output. The input here is determined by the models input (real numbers typically), and processed through the weights and activation functions of the neuron, which then results in an output value either at an intermediate level, or as the models final output. The system learns by considering input samples sequentially, and adjusting the weights between edges to result in more accurate outputs, which may either be classification or regression values.

Structured neural networks that learn to some extent have been around since the second half of the 21st century [12], though have been through several cycles of popularity. The first versions tended to be very simple with one one layer of hidden neurons [13]. It was only later, through the application of the backpropagation algorithm, that they started to become more practical and popular [14].

With the rise in popularity, many different network formations were developed and suggested. One of the initial suggestions was the conceptually simple Feedforward Neural Network (FNN) as described above - an acyclic graph where inputs are processed in a single direction until the output is reached. The other notable earlier model was the Recurrent Neural Network (RNN), which has a cyclic graph instead - this results in a more powerful computational system than the standard FNN, which was shown to be effective quite early on [15]. The Long Short-Term Memory (LSTM) network was another that used recurrent dynamics, though at a neuron level, in that the neuron is responsible for remembering values for an arbitrary time period [16]. Convolutional Neural Networks (CNN) have a non-recurrent structure, but implement separate pooling layers of neurons which consider the adjacent input values for each feature (e.g. pixels next to each other). These have been shown to be incredibly effective at tasks such as image recognition, as elaborated on later.

There are three primary learning paradigms used in neural network training - Supervised Learning (SL), where the network is trained on inputs with known outputs; Unsupervised Learning (UL), where the network is trained to identify unknown structures as an output; and Reinforcement Learning (RL), where environmental reactions are used as inputs to train a network for certain outputs [12]. While all of these configurations and paradigms have their benefits and uses, this paper will largely focus on FNN and RNNs, trained through SL and UL.

2.2.1 Training and Backpropagation

Historically, the crux of neural networks popularity has often been based on the development of novel training methodologies, and how they have increased performance. In line with this, the Backpropagation (BP) algorithm (as defined in 5.2.3) has played a pivotal part: while neural network (or perceptron) models were around long be-

fore NN popularity, they were largely deemed ineffective, at least in comparison to other available models of the time [17]. It was only during the 1980s that the BP algorithm was applied to NNs, and the field started to gain in popularity again [18, 19].

Rumelhart et al. showed that the BP algorithm as applied in NNs resulted in useful feature representations occuring in hidden layers and the empirical success that resulted thereof [20]. Shortly after, LeCun et al. applied the BP algorithm to CNNs with adaptive connections. They were able to show impressive performance for the time in classifying handwritten images, with the images as a direct input (rather than a feature vector) [21].

While many improvements were made during this time via gradient descent modifications, the models were typically of a shallow nature due to problems encountered trying to train deeper networks. Early experiments with deep networks resulted in poor performance due to what is now widely known as the problem of either vanishing or exploding gradients [22]. Essentially, as more layers are added to the network, the backpropagation algorithm (with typical activation function neurons), results in error signals that either shrink or grow out of bounds at an exponential rate. One of the first suggested and primary solutions to the problem is to perform pre-training on the network through unsupervised learning [12], which is discussed more fully in 2.3.2. Variance based weight initialization techniques have grown in popularity in recent years, as discussed in 2.2.4.

There were also initial concerns that the BP algorithm as applied to high dimensional neural networks would result in the network weights being trapped in local minima if a simple gradient descent was used (e.g. where no small changes to the configuration would reduce the average error rate) [23]. However, empirically, this tends not to so problematic, and large networks usually reach solutions of equitable performance. More recent research has shown that the solution spaces largely consist of many saddle points, each with varying gradients of the features, but which also tend to have similar values of the objective function [24]. Ge et al. have also shown that it is possible to escape saddle points and offer a guaranteed global convergence in certain non-convex problems [25].

2.2.2 Activation Functions

One of the upfront configuration choices necessary is the activation function, which allows the mapping of input to output at the neuron level. There have been many suggestions and experiments with different functions, though there are some common features amongst functions which might make them appropriate: Non-linearity allows for neural networks to operate as universal approximators, as shown in [26], continuous differentiability allows for the use of gradient descent and whether the function is monotonic has been shown to indicate whether the solution can be guaranteed to have a unique periodic solution [27]. Lastly, the range of the function (infinite or finite) can impact both the stability and efficiency of the training.

Some of the most popular functions that have been used are the Sigmoid, TanH, ReLU and Softsign. There have been various studies showing the effectiveness of the different activations under varying initialization (or pre-training) for weights. Glorot and Bengio noted that the typical Sigmoid and TanH functions performed poorly with standard minimization, and result in slower convergence and worse minima, and

showed that Softsign with a non-standard initialization resulted in quicker convergence [28]. Further research with Bordes and Bengio found that the rectifier (ReLU) functions were more effective in deep sparse networks compared to the TanH function [29].

2.2.3 Deep Learning

As noted above, most of the earlier work using neural networks relied on shallow models with few layers. However, a resurgence in interest occurred in 2006 after several papers demonstrated the efficacy of unsupervised pre-training of networks prior to supervised training. The effect was substantial enough to allow much deeper layered networks to be trained than before [30, 31].

The essential point behind the unsupervised learning was to initialize the weights in the network to sensible values in light of the problem context. The methods used trained each layer to be able to reconstruct the model of the features in the layer below (to a varying degree of accuracy). Sequentially pre-training and combining layers like this, the process generated a deep neural network with appropriate weights. Once done, a final output layer was added and the entire network could then be fine-tuned through backpropagation, without suffering such performance degradation through vanishing or exploding gradients [31, 32, 33]. This is expanded up further in 2.3.2. Le Roux and Bengio were able to show that within the DBNs produced by Hinton [31], adding hidden nodes resulted in strictly improved modelling capabilities, and they suggested that increasing the number of layers is likely to result in increased representational ability (subject to efficacy of previous layers), thus establishing the argument for deep networks in theory as well as practice [34].

FNNs had been shown to be effective in modelling high dimensional data even prior to the breakthroughs in deep networks [35], so it fits that the deep networks were shown to be extremely effective in high dimensional data classification. Early implementations shown increased efficacy in handwriting recognition, as well as pedestrian recognition [36]. When it came to data types such as sound and images, CNNs were implemented on several occasions with record breaking model performances in recognition, notably in ImageNet and WaveNet [37, 38].

As more research into deep networks was conducted, it became apparent that with large enough datasets, the layerwise pre-training of networks was not actually necessary to achieve high performance standards [37, 29, 39]. When training for long enough, it was reported that the pre-training offered little to no benefit, though these models were typically using datasets far larger than were attempted before (as a result of hardware improvements enabling as much). While these results did require that certain attention was paid to the initialization, as well as the use of nonlinear activation units, it did suggest that pre-training largely acted as a prior which may not be necessary if large enough labelled datasets are available [40]. Naturally, pre-training was still implemented to prevent overfitting in smaller datasets.

2.2.4 Weight Initialization Improvements

One of the more critical innovations to allow effective training of deep networks without using pre-training was the development of more sophisticated techniques for weight initialization. One of the first and most popular of these was presented by Glorot and

Bengio for use in Sigmoid based networks, and is commonly referred to as Xavier/Glorot initialization [28]. The technique is layer specific and based on a linear activation hypothesis, such that the initial weights would maintain the same variance for input for information that is passed backwards as in accordance with the nature of the activation function. While the assumption of linearity is not entirely correct, they point out that as the start of the learning process, it is typically the area of the activations where the gradient is close to 1 which is being explore, thus intially approximating a linear effect. The effect is a technique that increases learning efficacy and optima found [28].

This same methodology was extended for the ReLU activation by He et. al, once again based on the linearity hypothesis around the relevant activation function, (Parametric Rectified Linear Units in this case). The combination of ReLU activations, which are computationally inexpensive and do not suffer from learning slowdown, as well as an effective weight initialization technique such that pretraining was not necessary produced a seminal FNN training framework. He et al. were able to achieve state of the art performance and produced the best known error rate at the time on the ImageNet dataset [41]. The efficacy of these techniques has established them as norms in the training of deep neural networks.

2.3 Stacked Autoencoders

2.3.1 High Dimensional Data Reduction

As noted, machine learning techniques have been shown to be extremely effective at modelling non-linear inputs to outputs - neural networks have even been shown to be universal function approximators in this regard [26]. More traditional statistical models will typically process the available feature data to select the most significant features to be used in the model once its defined - evident in a processes such as subset selection [42]. Machine learning techniques are no different in this regard, and feature data will typically be transformed to smaller observations of more significance prior to being used as input to a model, such as the neural networks described above.

Financial data, in line with the complex and dynamic system that it represents, is often of a very high dimensional nature, which offers opportunities through more sophisticated analysis, but also introduces the curses of dimensionality [43]. The increased dimensionality can result in higher processing complexities when needing to do basic tasks such as estimating a covariance matrix (a commonplace necessity in finance), as well as increase the risk of incorrect assumptions based on spurious variable collinearity [44]. Noise accumulation in high dimensional data can create further problems, resulting in problems performing variable selection and ultimately having a large impact on classification and regression models [45].

Time series data can introduce its own set of challenges - there is often not enough data available to understand and predict the process [46], the time variable dependence creates complexity in how much past data to consider at any point, and the data is typically non-stationary [47]. Thus, high dimensional time series data (which many financial problems focus on), require careful consideration on how to handle their inputs and analysis.

Deep learning techniques are a natural choice in this context, and much research has been done to show their (varying) efficacy on time series data. The most successful

of these models have been ones which modify deep learning techniques to incorporate the temporal aspect of the data (e.g. Conditional Restricted Boltzmann Machines or Recurrent Neural Networks), rather than static, and those which have performed feature selection processes rather than operating on the raw data (e.g. Auto-encoders) [47].

Two of the seminal pieces of research that have lead to the resurgence in machine learning and deep learning were the algorithms for training deep belief networks [31], as well as the usage of stacked auto-encoders [32, 30].

2.3.2 Deep Belief Networks

Autoencoders were suggested by Hinton et. al as a method of transforming high dimensional data to lower dimensional input vectors, in order to alleviate some of the problems detailed above, and increase performance of deep belief networks [33].

One of the more prominent classical techniques for dimension reduction is principal components analysis (PCA), which uses linear algebra to find the directions of greatest variance, and represent the observation samples features along each of these directions, thus maximising the variational representation. Hinton et al. show that autoencoders are a nonlinear generalization of PCA. The structure and training algorithms of the autoencoder show it to be a specialised neural network - there is a multilayer encoder network which is able to transform to a lower dimension, and a symmetrical decoder network to recover the data from the code as represented in Figure 1. As with neural networks, the gradient weights can be trained through the feedforward and backpropagation algorithms.

The primary challenge presented here was the initial weighting of the networks - with large initial weights the autoencoder will often find a poor local minima, and with small initial weights the gradients are too small to effectively train deep layered networks. The critical suggestion by Hinton et al. was to used layered Restricted Boltzmann Machines (RBM) in order to initialise the weights. For each layer of the desired autoencoder, a RBM is formed and trained with the previous layer (or RBM) [48]. Once all the layers have been trained in this way, they are mirrored to form the decoder network. This then forms the initial weights to be fine tuned further, as per the Fine-tuning step in 1. They showed the deep autoencoder networks were significantly more effective than PCA or shallow autoencoders on multiple dataset types.

2.3.3 Stacked Denoising Autoencoders

The second important piece of work was the development of a denoising autoencoder (DAE), by Vincent et al. [49]. One of the problems identified in the DBN model (and those similar), is that if the encoder dimensions were too high, it is likely that the encoder would learn a trivial encoding - essentially creating a copy the input model. The one way of tackling this issue is to constrain the representation with bottlenecks and sparse autoencoder layers, which can be seen in figure 1.

Vincent et al. explore a very different approach to the problem, which was to develop an implementation of autoencoder which focused on partially corrupting the input, and so force the network to denoise it. The theory here is based on two ideas - the first, is that a higher dimensional representation should be robust to partial

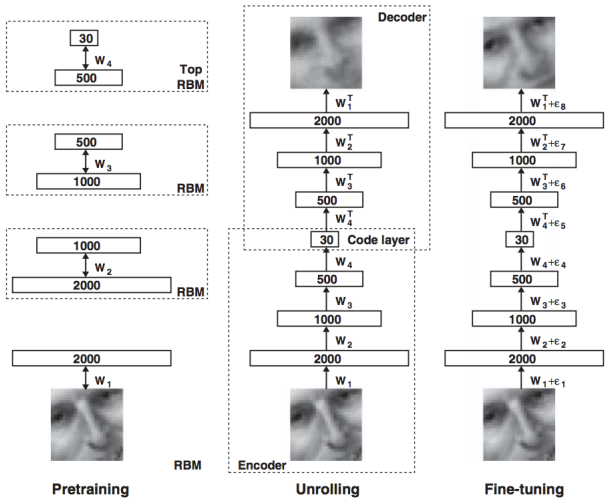


Figure 1: The Autoencoder training steps [33]

corruption of the input data; and the second is that the denoising process will force model focus to shift to extracting useful features from the input.

The algorithms and structures are largely the same as described for DBNs above, with the key difference being that the model is trained to reconstruct the original input, but only using a corrupted version of the input (where noise has been added to it), and so is forced to learn smarter feature mappings and extractions. The DAE suggested then is a stochastic variant of the autoencoder, which has the benefit of being able to implement higher dimensional representations without risking training of a trivial identity mapping. Notably, in the Stacked Denoising Autoencoder (SDAE) formation, only the initial input is corrupted (as opposed to the input from layer to layer). It was shown that the SDAE model outperformed previous AE and DBN networks on numerous benchmark datasets [49] .

2.3.4 Pre-training

The methods described above follow a similar approach: greedy layer-wise unsupervised pre-training in order to determine initial weights, followed by supervised fine tuning to arrive at the final model. It is shown numerous times, that the pretraining process can result in significant performance gains [49]. However it is not immediately apparent, given the nature of backpropagation algorithms and the like, why this is the case. Erhan et al. performed extensive empirical simulations in order to suggest an explanation to the mechanism of pre-training [50].

While their results were not entirely conclusive, they did lend themselves to a reasonable hypothesis: the unsupervised pre-training results in a form of regularization on the model - variance is minimized, and the bias introduced acts as a prior to direct the model configuration towards a sample space that is effective for the unsupervised learning generalization optimisations.

2.3.5 Time Series Applications

The autoencoder papers reviewed so far in this section derive their results primarily from classification problems, and so do not necessarily account for the problems involved with time series as described in 2.3.1. Due to the inherent difficulties with predictions in the financial system, it can sometimes be unclear if the shortcoming in results is due to this system complexity or if the methodologies used are unsuited for the purpose. In light of this it is worth pointing out that Stacked Autoencoder (SAE) implementations have been shown to be effective in many time series systems.

Lv et al. implemented a deep learning SAE model using the methods described in 2.3.3 in order to predict traffic flow at various time intervals (15, 30, 45 and 60 minutes) - a problem not so structurally dissimilar from what will be presented in this paper [51]. They were able to show that the deep SAE was able to offer prediction results which were both objectively good and also persistently outperformed the comparison models used (backpropagation neural network, random walk forecast, support vector machine and a radial basis function neural network).

In a review of unsupervised feature learning and deep learning methods on time series, Langkvist et al. noted that the use of autoencoders, either as a technique in themselves, or as an auxiliary technique to models such as convolutional neural networks, were able to offer performance increases in areas such as video analysis, motion capture data and bacteria identification [47].

2.3.6 Financial Applications

There have of course also been successful applications of stacked autoencoders and deep learning models in finance as well. Takeuchi et al. performed some earlier work showing the use of autoencoders when applied to a momentum trading strategy. They implemented an RBM pre-trained DBN as per 2.3.2, and assessed the networks classification performance for ordinary shares on NYSE, AMEX and Nasdaq. This showed that using a DBN network resulted in significant performance increases compared to the standard momentum strategy [52].

Zhao et al. used SDAEs and combined them with the bootstrap aggregation ensemble method (bagging) in a study of predicting the crude oil price. They compared the proposed model to a variety of benchmarks, including standard SAE, bagged and standard feedforward networks and SVRs. The results indicated that the SAE models were more accurate, with the bagged SAE model performing the best, though at a significant increase in computational costs in comparison to standard SAE [53].

While much of the financial literature has focused on the use of RBM based models, Autoencoders and SAEs have recently been gaining popularity in performing feature reduction. Troiano et al. specifically investigate the use of different feature reduction models for trend prediction in finance [54]. In line with being primarily interested in the effect of feature reduction techniques, rather than the classification performance itself, only an SVM model was used to test results. Using various periods from historical S&P 500 data, they were able to show that AE outperformed the RBM model significantly in numerous accuracy measures, and was able to do so at a fraction of the training time.

Bao et al. note that the research has been lacking with regards to whether SAEs

should be used for financial prediction models or not [55]. They suggest a novel model which combines Wavelet Transformation, SAEs and a Long Short Term Memory (LSTM) network. Using data from several financial exchanges (considering a range of developed and undeveloped markets), they assess the models applicability to OHLC prediction. Comparing the model to configurations without the SAE layers, and a RNN model as benchmark, they showed that the inclusion of SAEs resulted in less volatility and greater accuracy, which in turn offered higher profitabilities in a buy-and-hold trading strategy.

More novel autoencoder applications have also been attempted, with Hsu suggesting the use of a Recurrent Autoencoder for multidimensional time series prediction [56]. There is a clear pattern through the literature that the use of AEs and SAEs both by themselves and when used as an assisting technique result in more accurate prediction results and less computationally expensive training.

2.4 Online Learning Algorithms and Gradient Descent

Most classic machine learning algorithms operate under the assumption that, for all intents and purposes, the full dataset has been collected and that the amount of training data for the model is both finite and immediately available. However, as the growth of information grows in an exponential fashion, there are numerous areas where the expected training data for the model will continue to grow. In these cases it would be disadvantageous to go through the full training and validation process again in order to incorporate the newly available data.

Online algorithms are designed to offset these issues by adjusting the batch training technique to rather repetitively draw on single samples from the data on which the models parameters can be adjusted. The benefit is that they are able to quickly process a large number of observations and readjust the model, though the downfall is that they are not always able to optimize the cost function to the same extent as offline batch algorithms [57].

Bottou and Cun argue that as the size of the dataset grows significantly, online algorithms advantages result in them outperforming offline models, despite any initial drawbacks [58]. Previous research had shown that online algorithms typically perform as fast as batch algorithms during the search phase of parameter optimization, but that final phase convergence tended to fluctuate around the optima due to the noise present in single sample gradients [59, 60]. Bottou and Cun showed in fact, that it is more practical to consider the convergence towards the parameters of the optima, rather than the optima itself (as defined by the cost function) - the difference between the learning speed and optimization speed, respectively [58]. Theoretical and empirical findings were presented to show that a stochastic online gradient descent algorithm was able to outperform the batch model for parameter estimation, and was able to asymptotically outperform in the number of samples processed in a time period. The stochastic aspect of the algorithm is related to random observations from batch sample groups being used as the gradient basis. Theoretically, this slows down the convergence, but speeds up the processing speed of each batch - a technique which has later been shown to be generally successful [61, 62].

Stochastic gradient descent algorithms have resulted in a fair amount of further research due to their applicability to machine learning and the online benefit, especially in relation to their usage in neural networks.

2.5 Gradient Learning Improvements

2.5.1 Gradient Adjustments and Regularization

One of the earlier improvements to convergence rates was the Momentum algorithm, as developed by Tseng [63]. As noted, stochastic descent often introduces significant oscillation around an optima, which slows down convergence. Momentum reduces this by decreasing movement in directions of high curvature, and increasing movement towards directions consistent with previous gradients (this is achieved through combining gradient movements in opposite directions).

There have been several attempts to introduce effective regularization into the SGD process. Bartlett et al. presented Adaptive Online Gradient Descent, which implements an adaptive step size through a λ penalty on the learning rate, which was shown to be nearly optimal in a strong sense [64]. Langford et al. demonstrated a variation named Truncated Gradient, which introduced an enforced weight sparsity parameter. The weight sparsity is able to achieve equitable effects to L_1 regularization (similar to Lasso Regression). They were able to show that implementation performed effective feature reduction, while having little effect on performance [65]. Other approaches, such as AdaGrad, aim to improve the robustness of gradient training by adjusting the updates to parameters according to frequency - e.g. larger updates to infrequent parameters, and smaller updates to frequent parameters [66, 67].

2.5.2 Dropout

One of the instrumental improvements in further achievements, was to modify the backpropagation algorithm using the dropout technique, as suggested by Hinton et al [68]. When training of large networks is attempted on small datasets, it often results in overfitting and poor results on out of sample data. Dropout helps resolve this by randomly excluding a certain percentage (usually 0.5) of feature detectors on each training iteration. The effect is to stop co-adaptations of feature detectors, and by rather training each neuron in a wide variety of internal configurations, it forces them to take on more usefully generalizable characteristics (it was noted that this is not a dissimilar technique to ensemble methods, or bagging). The authors were able to show that the method resulting in significant improvements on benchmark data sets (e.g. MNIST, CIFAR-10), and that a simpler model using dropout was able to achieve near comparable performance for the ImageNet dataset.

Goodfellow et al. used the dropout technique as the basis for their maxout activation function technique, which leverages and improves on dropouts fast optimisation and accuracy through averaging characteristics. The maxout model was shown to achieve state of the art performance on benchmark datasets, as well as have a strong theoretical grounding [69]. Further work was done by Wang et al., which improved on the dropout (and potentially maxout) techniques through fast sampling, resulting in an order of magnitude speedup in training [70].

2.5.3 Learning Rate Schedules

Several more recent approaches look towards learning rate adjustment schedules rather than strategies to alter the weight update, as with momentum and the like. A constant learning rate often suffers from one of two problems - if set too high, it can cause divergent behaviour in the loss function, though if set too low, it can result in slow learning or an inability to escape saddle points effectively. Finding an optimal learning rate requires some degree of testing, though even then a singular value may fail to achieve the same degree of efficacy as a range of values explored throughou the solution space.

Learning rate scheduling has been proposed as a solution to this problem. These methods implement an repetitive cycle for the learning rate, such that it is set through a range of values between a mimima and maxima, being adjusted slightly with each new epoch. Smith suggests using the Cyclical Learning Rate to achieve this, and points out that part of the benefit in learning rate schedules is the ability to jump out of sharp optima points which may not generalise well to unseen data [71]. It's shown that this implementation can have significant performance effects in reaching either the same optima in fewer epochs or a better overall optima, including when used in conjunction with other learning optimizations noted in 2.5. These results were shown against standard datasets such as CIFAR-10, CIFAR-100 and Imagenet [71].

Similarly, Loshchilov and Hutter, expand on this and present Stochastic Gradient Descent with Warm Restarts [72]. The approach is implemented similarly, but rather follows an asymmetric cycle from a maximum to a minimum, and then starting once again at the maximum once the set number of epochs has passed. They too note the increased performance on CIFAR-10 and CIFAR-100 datasets in reaching optima far quicker, as well as noting the efficacy of the learning rate schedule even without implementing the restarts [72].

2.6 Backtesting and Model Validation

Much of financial academic literature is currently facing a problem in terms of validation and verification of results. The primary method of going about these ends in the past has been to perform historical simulations, or backtests , in order to prove profitability of a trading strategy. The recent advances in both technology and the algorithms available to construct these strategies has resulted in researchers being able to run so many iterations of a model or strategy configuration through these backtests, that its become increasingly difficult to control for spurious results, with some papers suggesting that most published research findings are false [73].

The standard way of implementing backtests is to split the data into two portions: an In Sample (IS) portion which is used to train the model, and an Out of Sample (OOS) portion which is used to test the model and validate results. The problem lies in that millions of different model configurations might be tested, and if more sophisticated test measures are not in place (i.e. not just the standard Neyman-Pearson hypothesis testing framework is implemented), then it is only a matter of time before a false positive result occurs which shows high performance both IS and OOS (i.e. overfitting). The nature of financial data, where there is a low signal-to-noise ratio in a dynamic and adaptive system, and where there is only one true data sequence,

makes it difficult to resolve these issues effectively [1, 74].

Overfitting is not a novel issue, and has of course been tackled in various literature areas, including machine learning. However, in that context, the frameworks are often not suited to the buy/sell with random frequency structure of investment strategies. They also do not account for overfitting outside of the output parameters, or take into consideration the number of trials attempted. Other methods, such as hold-out, are arguably still faulty due to researcher knowledge while constructing models [75]. One of the downfalls of the typical IS-OOS set up in the financial context is also that the most recent (and relevant) data will not be able to be used for the model training.

There have been some suggestions to resolve the problem that is occuring in the literature as a result of this - some work suggesting new frameworks, which this section will cover, and others which focus on the review process or how data and replication procedures are made available [76]. While the points made with regard to the review process and so on are certainly important, they don't aid with more effective model training for the researcher up front, and so will not be covered here.

2.6.1 Testing Methodologies

Considering the issues laid out above, there has been much work to develop alternative approaches to backtesting. One of the common approaches to avoid backtest overfitting is the hold-out strategy, where a certain portion of the dataset is reserved for testing true OOS performance. Numerous problems have been pointed out with this approach, including that the data is often used regardless, or that awareness of the movements in the data may, consciously or otherwise, influence strategy and test design by the researchers [75]. For small samples, a hold-out strategy may be too short to be conclusive [77], and even for large samples it results in the most recent data (which is arguably the most pertinent) not being used for model selection [78, 1].

There has been work by several authors to try and lay out techniques to try and avert backtest overfitting. The Model Confidence Set (MCS), as developed by Hansen et al. [79], starts with a collection of models or configurations, and remove models iteratively according to a defined loss function. The confidence set is defined by the remaining models once a non-rejection takes place within the process, and these models are considered to be statistically similar within a certain confidence range. MCS is thus able to facilitate equitable model selection. However, Aparicio et al. [80], showed that while MCS is a potential strategy, in practice is is ineffective due to the inordinate requirement of signal-to-noise necessary to identify true superior models, as well as a lack of penalization over the number of trials attempted.

Bailey et al. [1] have developed a more robust approach to backtesting and how overfitting during strategy selection might be avoided, called Combinatorially Symmetric Cross-validation (CSCV). Their research defines backtest overfitting as having occurred when the strategy selection which maximizes IS performance systematically underperforms median OOS in comparison to the remaining configurations. They use this definition to develop a framework which measures the probability of such an event occuring, where the sample space is the combined pairs of IS and OOS performance of the available configurations. The probability of backtest overfitting (PBO) is then established as the likelihood of a configuration underperforming the median IS while outperforming IS.

The CSCV methodology provides several important benefits over traditional testing frameworks, including the usual K-fold cross validation used in machine learning. By recombining the slices of available data, both the training and testing sets are of equal size, which is particularly advantageous when comparing financial statistics such as the Sharpe Ratio (SR), which are susceptible to sample size. Additionally, the symmetry of the set combinations in CSCV ensure that performance degradation is only as a result of overfitting, and not arbitrary differences in data sets. It is pointed out that while CSCV and PBO should be used to evaluate the quality of a strategy, they should not be the function on which strategy selection relies, which in itself would result in overfitting.

2.6.2 Test Data Length

shorten section?

The CSCV methodology offers an important but highly generalised framework to assess models and backtest overfitting. It doesnt however indicate which metrics should be used to assess the IS and OOS performance, nor any indication on the amount of data needed to do so effectively. One of the noted limitations of the framework is that a high PBO indicates overfitting within the group of N strategies, which is not necessarily indicative that none of the strategies are skillful - it could be that all of them are. Also, as pointed out, it should not be used as an objective function to avoid overfitting, but rather as an evaluation tool. To this end it helps assess overfitting, but not necessarily avoid it.

A typical measure of evaluation used for financial models is the Sharpe Ratio (SR), which is the ratio of between average excess returns and the returns standard deviation - a measure of the return on risk. In the context of comparing models, SR is typically expressed annually to allow models with different frequencies to be compared. Lo et al. [81] show that annulaized SR can be expressed as

$$\texttt{SR} = \frac{\mu}{\sigma}\sqrt{q} \tag{1}$$

Using sample means and deviations, $\hat{\mu}$ and $\hat{\sigma}$, SR can be shown to converge as follows (as y $\to \infty$)

$$\hat{SR} \to \mathcal{N}[SR, \frac{1 + \frac{SR^2}{2q}}{y}] \tag{2}$$

Thus, when using SR estimations, which follow a Normal distribution, it is possible that where the true SR mean is zero we may still (with enough configurations attempted) find an SR measurement which optimises IS performance. This is shown by Bailey et al. [82], who propose the non-null probability of selecting an IS strategy with null expected performance OOS. Notably, typical methods such as hold-out once again fail, as the number of configurations attempted are not recorded. They add a further derivation, thich is the Minimum Backtest Length (MinBTL), ultimately showing that

$$MinBTL \approx (\frac{(1-\gamma)Z^{-1}[1-\frac{1}{N}] + \gamma Z^{-1}[1-\frac{1}{N}e^{-1}]}{E[max_N]})^2 < \frac{2ln[N]}{E[max_N]}^2 \tag{3}$$

The statistic highlights the relationships between: selecting a strategy with a higher IS SR than expected OOS, the number of strategies tested (N), and the number of

years tested (y). The equation shows that as the number of strategies tested increases, the minimum back test length much also increase in order to contain the likelihood of overfitting to IS SR.

As shown extensively throughout ML literature, increased model complexity and number of parameters is one of the primary causes of overfitting. In context of the MinBTL formula, model complexity affects the number of configurations that are available and which may be tested, which in turn will increase likelihood of overfitting. A lack of consideration, or reporting, of the number of trials makes the potential for overfitting impossible to assess.

Bailey et al. expanded on this view with assessing the impact of presenting overfit models as correct. They were able to show that in lieu of any compensation effects (i.e. a series following a Gaussian random walk), there is no reason for overfitting to result in negative performance. However, where compensation effects apply (e.g. economic/investment cycles, bubble bursts, major corrections etc.), then the inclusion of memory in a strategy is likely to be detrimental to OOS performance if overfitting isnt controlled for [82].

2.6.3 Sharpe Ratio

The use of the Sharpe Ratio in financial backtesting is not just an arbitrary or persistent literature choice. The statistic offers two benefits: the effectively strategy-agnostic financial information contained, as well as being relatable to the t-statistic, and so simple to perform hypothesis testing. The SR ratio (estimate from sample as $\hat{SR}$) is defined as

$$\texttt{SR} = \frac{\mu}{\sigma} \tag{4}$$

The t-ratio is defined as

$$\texttt{t-ratio} = \frac{\hat{\mu}}{\hat{\sigma}/\sqrt{T}} \tag{5}$$

Evidently, the link heree ratio (though it can be generalized to another statistic with a probabilistic interpretation). Additionally, PBO is generally more in line with machine learning literature with the cross validation like approach on time series data.

It should be noted, that the literature detailing usage of the Sharpe ratio for strategy comparison is extensive, with numerous variations and methodologies offered [83]. However, the crux of this paper lies in whether an online neural network is able to make effective enough predictions that a strategy might use the predictions to be profitable. The subtlety here is that we will consider the usage of such forecasting *within* a strategy, rather than *as* a strategy itself. In line with this, statistics such as the Sharpe ratio will be used, but not form a critical consideration of the research here as the comparison of strategies will not be the focus.

3 Software Libraries and Development

A significant aspect of this dissertation was the implementation of software libraries that fulfilled the criteria of generalised training of neural networks, an implementation of the full process as described later in Chapter 5.1, the creation and management of database records for analysis and an extensive set of diagnostic tools in order to assess performance. This Chapter details the implementation choices as well as the final libraries which were produced.

3.1 Programming Languages

The libraries developed were primarily written in Julia, a relatively new scientific computing language (initially developd in 2009). The main motivation, both for Julia as a language as well as choosing it for this task, is its efficiency and speed. Julia was created with high performance tasks in mind, and with it's use of a Just-In-Time compiler, can approach similar benchmarks to C. This stands in stark contrast to some of the more popular analytical languages used, such as R or Python, which often suffer from performance issues. In the context of a project which required extensive processing time, it was natural to prioritize this computational efficiency.

Julia's focus on technical computing is emphasised with it's built in support for mathematical and statistical work, composable functional base and dyamic and optional typing schemes. It's ability to fulfil general purpose programming, familiar syntax (which also works effectively for mathematical commands) and it's interactive (REPL) command line also make it a practical choice for the task at hand.

SQLite was chosen as the database management language. Relational schemas were a fitting choice for the nature of the data being captured, and SQLite offers a fast, effective and free implementation of SQL.

3.2 Data Generation & Processing

Data processing libraries were implemented to offer configurable processing for the steps outline in Chapter 4. The Data Generator module implemented Geometric Brownian Motion (GBM) generation for a set number of steps and assets with their mean and variances specified [1].

The Data Processor module [2] offers a more extensive set of operations:

1 Dataset scaling. Available implementations: Normalize, Standardize, Limited-Normalize, Limited-Standardize

2 Log Differencing with rolling windows for past and future time point aggregations

3 Partitioning and creating datasets for training processes

4 Encoding datasets using Stacked Autoencoders

5 Reverse scaling and price reconstruction (i.e. reversing the log differencing from 2) to process predictions into the original format

6 Add noise to datasets for denoising training

[1] `https://github.com/joel11/Masters/blob/master/Code%20Libraries/DataGenerator.jl`
[2] `https://github.com/joel11/Masters/blob/master/Code%20Libraries/DataProcessor.jl`

3.3 Network Training

Neural networks were implemented with highly configurable training, such that the parameters are easily specified within Julia types [3] and then passed onto the training functions. The parameters are grouped accordingly to the different aspects of the training that they pertain to.

3.3.1 Network Parameters

1 Network Structure: Number of layers, and size of each layer

2 Activations: Activation functions for each layer. Available implementations: Linear, Sigmoid, ReLU, LeakyReLU, Softmax, Tanh[4]

3 Network Initialization: The initialization method for the weights. Available implementations: Normal Random, Xavier, He, He-Adj [5]

3.3.2 SGD & CD-1 Training Parameters

1 Maximum learning rate, minimum learning rate and learning rate epoch cycles

2 Minibatch size

3 Maximum training epochs, or stopping function

4 L1 regularization lambda

5 Cost function. Avaialable Implementations: Mean Squared Error, Cross Entropy Error, Log Likelihood Error [6]

6 Denoising variance

7 Training / Testing data split

3.3.3 OGD Training Parameters

1 Maximum learning rate

2 Cost function. Avaialable Implementations: Mean Squared Error, Cross Entropy Error, Log Likelihood Error [7]

3.3.4 Network Trainer Module

The Network Trainer [8] module then offers 3 primary methods, given the relevant parameter configurations and dataset:

1 Train a Feedforward Neural Network, using variance based weight initializations

2 Train an SAE, using variance based weight initializations

3 Train an SAE, using RBM based pre-training for weight initialization

[3] https://github.com/joel11/Masters/blob/master/Code%20Libraries/TrainingStructures.jl
[4] https://github.com/joel11/Masters/blob/master/Code%20Libraries/ActivationFunctions.jl
[5] https://github.com/joel11/Masters/blob/master/Code%20Libraries/InitializationFunctions.jl
[6] https://github.com/joel11/Masters/blob/master/Code%20Libraries/CostFunctions.jl
[7] https://github.com/joel11/Masters/blob/master/Code%20Libraries/CostFunctions.jl
[8] https://github.com/joel11/Masters/blob/master/Code%20Libraries/NetworkTrainer.jl

These functions are supported by the following modules: SGD[9], RBM[10] and Gradient Functions[11]

3.4 Process Implementation

The process implementation, as described in Chapter 5.1, is largely implemented in the Experiment Process module [12].

The RunSAEConfigurationTest method uses the SAEExperimentConfig Type to wrap up all necessary configurations and with a passed dataset runs through the following steps:

1 Either generate or process the raw dataset, according to the specified configuration

2 Train the SAE, either by RBM or variance based weight initializations

3 Passes back the SAE with original and reconstructed testing datasets

4 Store the SAE as a bson object to be used later

The RunFFNConfigurationTest method similarly uses the FFNExperimentConfig Type and a passed dataset to run through the following steps:

1 Either generate or process the raw dataset, according to the specified configuration

2 Encodes the datasets for the different steps of the process using the specified SAE

3 Train the FFN using variance based weight initialization and SGD

4 Use the FFN to generate predictions for IS data, reconstruct them and record them in the database

5 Train the FFN using OGD on OOS data

6 Use the FFN to generate predictions for OOS data, reconstruct them and record them in the database

Two further modules allow the exploration of the configuration space, such that diagnostics can be performed on different aspects of the training. The ConfigTestsFFN[13] and ConfigTestsSAE[14] modules make use of the Config Generator[15] module in order to generate a full grid configuration space for a given range of values for any chosen parameters. Each of these combination configuration sets is then run as per the procsses detailed above, with the various results recorded as usual.

3.5 Database Implementation

As noted, the database platform chosen was SQLite, with Julia modules handling the creation and management. The Database Creator module [16] creates the database with the following schemas:

[9] https://github.com/joel11/Masters/blob/master/Code%20Libraries/SGD.jl
[10] https://github.com/joel11/Masters/blob/master/Code%20Libraries/RBM.jl
[11] https://github.com/joel11/Masters/blob/master/Code%20Libraries/GradientFunctions.jl
[12] https://github.com/joel11/Masters/blob/master/Code%20Libraries/ExperimentProcess.jl
[13] https://github.com/joel11/Masters/blob/master/Code%20Libraries/ConfigTestsFFN.jl
[14] https://github.com/joel11/Masters/blob/master/Code%20Libraries/ConfigTestsSAE.jl
[15] https://github.com/joel11/Masters/blob/master/Code%20Libraries/Generator.jl
[16] https://github.com/joel11/Masters/tree/master/Code%20Libraries/DatabaseCreator.jl

1 configuration_run, as the primary record for any experiment process

2 dataset_config: the dataset configuration that may relate to an experiment

3 network_parameters: the network configuration for an experiment

4 training_parameters: the SGD/RBM/CD training configuration for an experiment

5 epoch_records: details relating to ongoing training performance for the duration of an experiment

6 backtest_results: predictions for IS data

7 prediction_results: predictions for OOS data

8 cscv_returns: MMS level returns by timestep for an experiment

9 cscv_cost_returns: MMS level returns by timestep for an experiment with cost accounted for

10 config_oos_pl: the total OOS profit & loss for an experiment

11 config_oos_pl_cost: the total OOS profit & loss for an experiment, with cost accounted for

12 config_oos_sharpe_ratio: the OOS sharpe ratio for an experiment

13 config_confusion: the percentages of correct trades made for an experiment

The DatabaseOps [17] module handles the creation of records for these schemas, as well as the writing and reading of bson files which are used to store actual networks.

[17] `https://github.com/joel11/Masters/tree/master/Code%20Libraries/DatabaseOps.jl`

4 Data Processing and Generation

4.1 Data Processing

The datasets used go through several transformations throughout the training and prediction process:

1 The raw data is log differenced and aggregated to rolling windows

2 The transformed data is split into Training and Prediction subsets

3 The predicted outputs have the scaling and log differencing reversed in order to reconstruct the actual price points

4.1.1 Log Difference Transformation and Aggregation

All datasets are transformed into log feature fluctuation values, and were then aggregated to include fluctuations over rolling window periods. The log feature fluctuation for timepoint t is calculated using (6):

$$\texttt{log\_diff\_x}_t = ln(x_t) - ln(x_{t-1}) \tag{6}$$

This log feature fluctuation, is processed for each asset's closing price and for each time point (from the previous time point). The log fluctuations have the benefit of taking compound effects into account in a systematic way and are symmetric in terms of gains and losses.

The datasets are then expanded with the rolling window summations both in the past, for input, and in the future, for predicted output. A typical example would be past data aggregation windows of 1, 5 and 20, and a future prediction point of 5. These are calculated as summations of the log differences, such that for d days:

$$\texttt{past\_agg\_x}_{(t,d)} = \sum_{i=t-d}^{t} \texttt{log\_diff\_x}_i \tag{7}$$

$$\texttt{future\_agg\_x}_{(t,d)} = \sum_{i=t+1}^{t+d} \texttt{log\_diff\_x}_i \tag{8}$$

Only data points with a full set of features are used for training and prediction later on in the framework.

4.1.2 Data Scaling

Once the log difference data has been processed, the datasets are either standardized or normalized to allow for better learning. This is done in the typical ways:

$$\texttt{standardized\_x}_i = \frac{(x_i - \bar{x})}{\sigma_x} \tag{9}$$

$$\texttt{normalized\_x}_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{10}$$

Two extensions of these are used later on, dubbed 'Limited Standardization' and 'Limited Normalization', which are used when the data is split up into the Training

and Prediction sets. In this case all the data needs to be scaled, but the variance or range should not be travelling from the Prediction set to the Training set through the use of aggregated measures such as $\bar{x}$. Thus, if the data is split at point t out of n data points, the scaling would be implemented as follows:

$$\texttt{limited\_standardized\_x}_i = \frac{(x_i - \bar{x_{1:t}})}{\sigma_{x_{1:t}}}, \forall i \in (1, n) \tag{11}$$

$$\texttt{limited\_normalized\_x}_i = \frac{x_i - min(x_{1:t})}{max(x_{1:t}) - min(x_{1:t})}, \forall i \in (1, n) \tag{12}$$

This log-differenced, aggregated and scaled data is then used as the input for the neural network models.

4.1.3 Reverse Data Scaling

The predicted data points are transformed back into actual prices for returns analysis. The first step is to reverse the scaling that is done in 4.1.2 in order to retrieve the log difference fluctuations.

$$\texttt{reverse\_standardized\_x}_i = \texttt{standardized\_x}_i * \sigma_x + \bar{x} \tag{13}$$

$$\texttt{reverse\_normalized\_x}_i = \texttt{normalized\_x}_i * (max(x) - min(x)) + min(x) \tag{14}$$

These are applied similarly for the limited variations.

4.1.4 Price Reconstruction

The predicted log fluctuations produced in 4.1.3 are used to reconstruct the predicted prices.

$$\texttt{reconstructed price}_{s,i} = o_{s,i-t} * e^{p_{s,i}} \tag{15}$$

where:

o = original asset prices

s = asset selected

i = timepoint predicted from

d = timepoint predicted to

p = predicted log fluctuation

These reconstructed prices are then used to assess model returns later in the Money Management Strategy (5.7).

4.2 Synthetic Data Generation

Synthetic data generation is implemented using Geometric Brownian Motion (GBM) as described in Algorithm 1, which allows the stocks simulations to be implemented with a drift (σ) and variance (μ) for each stock. The GBM simulation has the benefit of generating prices in a random walk, such that future movements are independent

of past prices and so suitable for emulating an efficient financial market. The implications of using such data are discussed more fully in 8.2. Each dataset was generated with a random seed (using a Mersenne Twister pseudorandom number generator).

Algorithm 1: Geometric Brownian Motion Simulation

Input: σ, μ, S_0, *steps*
t = 1/steps;
prices = $[S_0]$;
foreach *i in 1:t* **do**
  z = random() $\sim N(0,1)$;
  $S_t = S_{t-1} * e^{(\mu - \frac{\sigma^2}{2})t + \sigma\sqrt{t}z)}$;
  prices = [prices; S_t];
end
Result: prices

4.2.1 GBM Data Distributions

An important aspect of GBM generated data is that the price changes follow a Log-Normal distribution. If this data is then processed as per the steps above in section 4.1, then after taking the `log_diff` values using equation (6), the values will follow a Normal distirbution. Once scaled using standardization or normalization, these values will become centered and possibly be following a standard normal distribution. However, due to the effect of the limited scaling techniques used, only the Training dataset will be consistent in following this distribution, with the Prediction dataset having an increasing number of outliers as the timepoint moves further away from the Training set demarcations. These issues have notable impacts through out the results, and are discussed more in Section 8.

4.3 Price Considerations

The Actual10 dataset, as described in Section 7.2.1, is a price relative dataset. It should be noted that this does not account for inflation, which arguably maks the effective training of SAE and predictive networks more difficult. However, not being able to fully account for inflation would be an issue in any forward looking predictive model, and so it seems sensible to keep it this way in our current dataset. It is also not expected to have a material impact on short term predictions, though it does make comparisons across time periods (i.e. IS and OOS) more difficult. Additionally, the dataset is price relative starting at 1.0, and so only price changes from the start onwards are accounted for and impact P&L, rather than the initial starting prices which might have then caused some imbalances. This does not have a large impact on the process, but does mean the P&L figures quotes in the results section are of unit measures, rather than any particular currency.

5 Models and Algorithms

5.1 Process Overview

The implementation focuses on bringing together several ideas: data reduction, deep learning with pre-training/weight initialization, online learning and backtest overfitting validation for the purposes of stock price prediction. The process implementation is discussed fully in 6.3, but can be summarised as the following:

1 The dataset is split into 2 subsets: the Training portion, and the Prediction portion.

2 The Training set is used to train the SAE and predictive FFN using the SGD algorithm. These networks are constructed with pre-training or weight initialization techniques.

3 Once the SGD training is complete for the predictive network, the same Training dataset is used to generate the IS predictions to be used for the CSCV process.

4 The Prediction set of data is used to continue training the network in an online manner using OGD.

5 The predictions made during steps [3] and [4] are descaled and prices are reconstructed.

6 These reconstructed price predictions are used by the MMS to calculate returns and P&L.

7 The returns and P&L calculated by the MMS are used by the CSCV process, to estimate the probability of backtest overfitting.

The rest of this chapter will detail the algorithms used to train the relevant FFN, RBM and SAE networks, as well as the trading strategy and CSCV & PBO testing procedures.

5.2 Feedforward Neural Networks

Constructing Feedforward Neural Networks (FFN) in the form of multilayer perceptrons is a well established network technique, providing effective nonlinear representations for both shallow and deep structures [12]. Specifically, a FFN is made up of several non-cyclical layers: the first and last are the input and output layers, respectively, and any inbetween are referred to as 'hidden' layers. Each layer is made up of nodes which are fully connected to the nodes in the previous and following layers, but does not have connections to nodes within the layer - information only travels forward. Each node has an activation function, which acts on the weighted input from the previous layers' nodes.

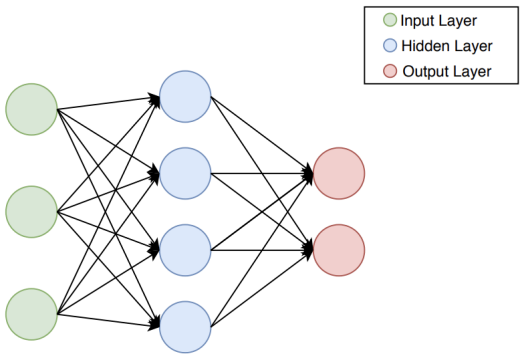


Figure 2: An example diagram of a Feedforward Neural Network with 1 hidden layer

5.2.1 Notation and Network Representation

For the purposes of this chapter, the following notation will be used:

1 A network N will be contructed with L layers, each with e nodes

2 The weights from the j^{th} node in the l^{th} layer from the k^{th} node in the $(l-1)^{th}$ layer are represented by w^l_{jk}

3 The bias for the j^{th} node in layer l is represented by $w^l_0 j$

4 The output for the j^{th} node is defined as $a = o(z_j)$, for an activation function o (with inverse o') and weighted input z

5 The weighted input for the j^{th} neuron in a layer is

$$z^l_j = \sum_{i=1}^{e^{l-1}} a^{l-1}_i w_{ij} + w_{0j} \tag{16}$$

These definitions allow an input into the network to be propagated through it, having the original values processed through the weights and activations functions, and have an output in the form of the network's last layer.

5.2.2 Activation Functions

As noted in 2.2.2, there are 3 primary characteristics of concern for activation functions: non-linearity, continuous differentiability and monotonicity. While many different functions have been suggested and used, several of the more popular were implemented here.

Sigmoid The sigmoid, or logistic, function is one of the most popular and widely used activation functions historically, and is defined as

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{17}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{18}$$

The Sigmoid function is in the range [0,1], making it a suitable choice for problems requiring a probabilistic output. The slope of the function curve is both a boon and

a drawback: it allows for fast learning initially, but results in learning slowdown later (often casuing what is referred to as node 'saturation'). The exponent calculation is also computationally expensive, relatively speaking.

ReLU The Rectified Linuear Unit (ReLU) is a newer activation function which has been shown to be effective in deep learning networks. It is defined as

$$\mathtt{o}(x) = \mathtt{max}(0, x) \tag{19}$$

(20)

$$\mathtt{o}'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

The function has the benefits of quick learning which doesn't saturate, as well as being computationally cheap. The downside is that the non-gradient for the negative range of the function can result in 'dead' nodes, which stop updating with the learning process.

Leaky ReLU The Leaky ReLU resolves the dying ReLU problem by adding a small gradient to the negative range of the function. This results in a slow learning being applied to 'dead' ReLU's, which may in turn result in them being used again if necessary.

(21)

$$\mathtt{o}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01x, & \text{otherwise} \end{cases}$$

(22)

$$\mathtt{o}'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0.01, & \text{otherwise} \end{cases}$$

Linear Activation Linear activations don't transform the input, and have a constant gradient, which can be useful for key layers where no loss in error signal is desired, such as the output or encoding layers.

$$\mathtt{o}(x) = x \tag{23}$$

$$\mathtt{o}'(x) = 1 \tag{24}$$

5.2.3 Backpropagation

The backpropagation algorithm, as discussed in 2.2.1, has allowed for effective training of FNNs for given data. The algorithm relies on incremental improvements of the model, as defined by decreasing the cost function. A common choice for cost is Mean Squared Errors (MSE):

$$C = \frac{1}{2}\|y - a^L\|^2 \tag{25}$$

A technical definition of the backpropagation algrithm is given in Algorithm 2, though the three primary steps of the algorithm are:

1 **Forward Pass**: The samples are propagated through the network, in order to generate the output $\hat{y}_s$. Vectorization is typically used to calculate the results for multiple samples at once, though conceptually each sample is used as input for the first layer, and subsequent layers use the weighted output from the previous layer passed through the activation function as input, using equation (16) for the weighted output and the equations defined in 5.2.2 for the activation functions.

2 **Calculate Cost**: The cost between the training output y_s and the model output $\hat{y}_s$ is calculated. Using this cost function, the rate of change for the error in the output layer by the rate of change in the activation function in the last layer. This is captured in equation (26):

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} o'\left(z_j^L\right) \tag{26}$$

The vectorized version of this is often used, and expressed in (27), where $\nabla_a C$ represents a vector of components which represent the partial derivatieves, such that they represent the rate of change of C relative to the activation functions' outputs.

$$\delta_j^L = \nabla_a C \otimes o'(z^L) \tag{27}$$

If using quadractive cost, as per 25, then the term $\frac{\partial C}{\partial a_j^L}$ can be reduced to $(a^L - y)$, such that the equation for the output layer error is

$$\lambda^L = (a^L - y) \otimes o'(z^L) \tag{28}$$

3 **Backwards Pass**: The backwards pass consists of 2 steps, where the errors for each previous layer are calculated, and the weights for that layers are then adjusted accordingly:

3.1 The activation values are propagated back through the network to calculate the delta values based on the cost value. This uses the same rationale as (26), but rather uses $(w^{l+1})^T$ by the next layers error in order to get an indication of how the error rate and how it moves backwards through the network.

$$\lambda^l = ((w^{l+1})^T \lambda^{l+1}) \otimes o'(z^l) \tag{29}$$

3.2 In order to update the network, each weights output error and input activation are multiplied to find the weights gradient and this gradient is reduced by a factor of the learning rate η, which is then subtracted from the network weights to update them proportionately to the error observed and chosen learning rate.

$$w^l \rightarrow w^l - \eta\lambda^l(a^{l-1})^T \tag{30}$$

Choices regarding the learning rates are discussed in 5.2.6 and the full algorithm is presented below in Algorithm 2.

Algorithm 2: Backpropagation

Input: Neural Network N, with randomly initialized weights w, L layers, and activation functions o and learning rate η

Data: Testing set with inputs x, and outputs y

repeat
  Select a sample x_s from x;
  // Perform the Forward Pass, to calculate the network output, y_s, for input sample x_s
  $a_0 = x_s$;
  foreach *l in 1:L* **do**
    foreach *j in 1:e^l* **do**
      $z_j^l = \sum_{i=1}^{e^{l-1}} a_i^{l-1} w_{ij} + w_{0j}$;
      $a_j^l = o(z_j^l)$;
    end
  end
  // Calculate the error term (aka cost), as
  $\lambda^L = (a^L - y) \otimes o'(z^L)$;

  // Perform the Backward Pass, to propagate the errors back and update the network accordingly
  foreach *l in (L-1):1* **do**
    // Calculate the delta values
    $\lambda^l = ((w^{l+1})^T \lambda^{l+1}) \otimes o'(z^l)$;
    // Update the weights
    $w^l \rightarrow w^l - \eta\lambda^l(a^{l-1})^T$;
  end
until *no new samples can be drawn from x*;
Result: Updated Network N

5.2.4 Gradient Descent Algorithms

The backpropagation algorithm is defined at a single sample level, and the learning from a dataset is usually repeated for a number of 'epochs'. As noted though, implementation is often implemented using a vectorized version of the algorithm which either samples the entire dataset at once (batch), or a subset of samples (minibatch). The latter is the Stochastic Gradient Descent (SGD), as discussed in 2.4, which has been shown to increase the speed and stability at which the backpropagation algorithm can converge to a minima in terms of cost. The algorithm runs backpropagation over the entire dataset for the number of epochs, and updates the network incrementally through the epoch. The stopping condition for the algorithm is usually defined as

either a particular number of epochs being reached, or cost no longer decreasing for some number of epochs (i.e. a minima has been reached).

The changes to the initial backprop algorithm, detailed in 2, are relatively minor:

1 Selection is now performed such that at each epoch m samples x_s are selected from x which have not yet been sampled.

2 The weight update is now performed such that it represents an aggregate update according the m samples that were selected:

$$w^l \to w^l - \frac{\eta}{m} \sum_x \lambda^{x,l} (a^{x,l-1})^T \tag{31}$$

Online Gradient Descent Where SGD is appropriate and effective for scenarios where the entire dataset is available, Online Gradient Descent (OGD) is applicable for when the model is learning in an online fashion. In this case, the backpropagation is run as defined above in Algorithm 2 but with no repetition (i.e. only 1 epoch).

5.2.5 Regularization

Regularization is a commonly used technique in machine learning used to reduce a model's capacity to overfit to the data, and in so doing reduces the variance in the model results. One of the typical methods, $L2$ (or "weight decay"), is implemented by adding an extra term to the cost function which is itself a function of the weights in the model. This extra term forces the learning process to favour smaller weights, and only allows large weights to occur if they are able to offer an appropriate increase in performance. The modified cost function is displayed below in 32.

$$C = \frac{1}{2n} \sum_x \left\| y - a^L \right\|^2 + \frac{\lambda}{2n} \sum_w w^2 \tag{32}$$

The additional term is scaled by the configurable regularization parameter, λ, which if small approximates the original cost function or if large will increase the degree of regularization used.

In order to implement this in backpropagation, the weight update rule is changed to the following (while biases remain the same):

$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \tag{33}$$

For SGD, this can be simplified to the following:

$$w \to \left(1 - \frac{\eta \lambda}{n}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w} \tag{34}$$

5.2.6 Learning Rate Schedule

Learning rates schedules are implemented to allow for a more dynamic exploration of the possible solution space, and allow fine tuning around an optima. With a constant

learning rate implementation, one has to choose between a larger learning rate which allows faster progress but less optimisation, or a smaller learning rate which learns slower but is more effective at optimising (by avoiding the learning algorithm from bouncing around a minima valley). Learning rate schedules aim to get the best of both scenarios by cycling through a minimum and maximum range across epochs.

The learning rate schedule is specified to cycle through from min to max every i epochs, from $\eta_{\min}$ to $\eta_{\max}$. Thus for the current epoch T, the learning rate is calculated using 35 below:

$$\eta_t = \eta^i_{\min} + \frac{1}{2}\left(\eta^i_{\max} - \eta^i_{\min}\right)\left(1 + \cos\left(\frac{T_{\text{current}}}{T_i}\pi\right)\right) \tag{35}$$

This is the Cyclical Learning Rates approach, though using sinusoidal form rather than linear [71]. For a minimum learning rate of 0.1, a maximum of 1.0 and an epoch cycle of 100, it will produce learning rates as displayed in figure 3 below.

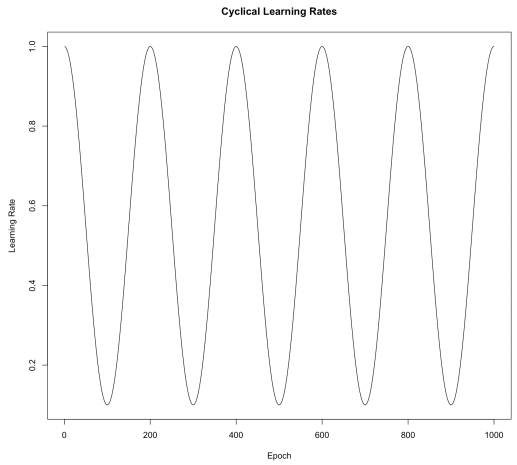


Figure 3: Learning rates calculated over 1000 epochs with $\eta_{\min} = 0.1$ to $\eta_{\max} = 1.0$ and $i = 100$

5.2.7 Dropout

Input layer dropout has been implemented in order to perform regularization on the data features. The methodology used will set a percentage of the features for each sample to 0 during the SGD training process. While the percentage stays the same throughout the testing process, the features selected for each sample are rechosen every time it is used (i.e. for subsequent epochs). Conceptually, the dropout as applied to the input layer may result in the network learning relations between assets and possible patterns that would improve predictive efficacy.

5.3 Restricted Boltzmann Machines

Restricted Boltzmann Machines (RBM) are generative networks which can be trained to learn probability distributions over a dataset. They are structurally different from a FFN in that they have a recurrent weight function - a typical RBM has one visible layer (input/output), and one hidden layer. A sample will be processed from the input layer to the hidden layer, and the activation values from the hidden layer will be used

by the input layer to provide a recontruction. The hidden units thus correspond to feature detection of the visible unit data structures, and the learning process of the network results in effective parameter estimation.

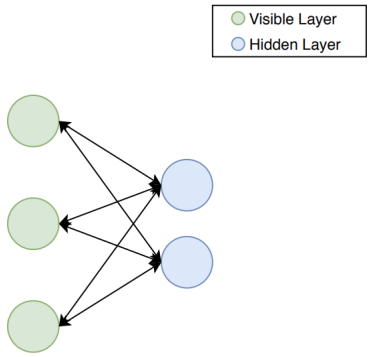


Figure 4: An example diagram of a Restricted Boltzmann Machine network

One of the primary differences from a FFN lies in the stochastic unit determination - the values in a hidden layer will typically be implemented such that the take on a binary value with a probablistic likelihood. Thus, the input and output have the same structure, and the processing from the hidden layer creates the generative process learned by the RBM.

The joint configuration (v, h) of the visible and hidden units has an energy given by:

$$E(v,h) = -\sum_{i \in visible} a_i v_i - \sum_{j \in hidden} b_j h_j - \sum_{i,j} v_i h_j w_i j \tag{36}$$

where v, h correspond to the binary states of the visible and hidden units, with biases a, b, and weights w [84]. It can be shown, that network weights can be adjusted to change the probabilities assigned to a particular training sample [84]. The derivations of 36 show that performing a stochastic descent for the log probability of the data can be implemented through the following weight adjustment:

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}) \tag{37}$$

The angular brackets here indicate expectations under the distribution specified by the following subscript. Due to the probabilistic nature of RBMs, the [0,1] ranged Sigmoid activation function is typically used. Thus, for hidden nodes and a data sample v, it is easy to get unbiased sampling of $\langle v_i h_j \rangle_{data}$

$$p(h_j = 1) = \sigma(w_{0j} + \sum_{i=1} v_i w_{ij}) \tag{38}$$

Similarly, the visible states (or reconstruction), can then be calculated as

$$p(v_i = 1) = \sigma(w_{0i} + \sum_{j=1} h_j w_{ij}) \tag{39}$$

Getting an unbiased sample of $\langle v_i h_j \rangle$ proves more problematic, and so the model reconstructions via Gibbs sampling are used instead (this is explained below), resulting in the weight updates of

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{reconstruction}) \tag{40}$$

While it's important for the hidden units to take on a binary value (and so avoid communicating real values rather than learning structure), the visible units may be chosen to take on the probability values, rather than the stochastic samples, particularly if real valued output is necessary.

5.3.1 Contrastive Divergence

The process of sampling and resampling may be run for many iterations between the two layers before finishing on an output - this potentially long running and stochastic process results in the generative aspect of the network, and constitutes a Gibbs sampling chain. Multiple sampling steps in this chain is known as Contrastive Divergence, or CD-n, where n represents the number of steps, and which allows for effective parameter estimation. Thus, CD-1 is the following:

Algorithm 3: CD-1

Input: RBM Network N, with visible nodes N and hidden nodes H and distribution $\langle vh \rangle$
Data: Training sample dataset V'

For a random training sample V', sample H' from $P(V|H)$;
Sample V from $P(V|H')$;
Sample H from $P(H|V)$;
Return $(V, H) \sim R$, where R is the reconstructed distribution for $\langle vh \rangle$;

`// Using the reconstructed distribution` $R(V, H)$`, the weight update for CD-1 is, as above, then`

$$\Delta CD(W) = \eta(E_{P(H|V)D(V)}[VH^T] - E_{R(V,H)}[VH^T]) \tag{41}$$

Result: Reconstructed distribution for $\langle vh \rangle$

There's no upper bound on the iterations used for CD, and running for many can prove more effective for certain purposes. In this case however, where RBMs are used for the purposes of weight initialization, CD-1 is usually deemed sufficient.

5.3.2 CD-1 and SGD

In the same way that the backpropagation learning algorithm in 5.2.3 can be implemented in an SGD process for FFNs, so can the CD learning algorithm for RBMs. The framework is kept the same, with the implementation of epochs and weight updates based on stochastically chosen minibatches, but the calculation used to update the weights is CD-1 instead.

5.4 Stacked Autoencoders

As noted in 2.3, the use of Stacked Autoencoders (SAE) have resulted in significant improvements in deep learning networks, and allowed effective reduction of high di-

mensional data. A single layer autoencoder is a specialized type of FNN with 3 layers: one input, one hidden, and one output. The network is trained (using backpropagation as per 5.2.3) to reconstruct the input, so the input and output layers have the same structure, and the hidden layer needs to have fewer nodes than the input. This forces the hidden layer to learn effective features of the data, and reduce the dimensional representation.

Stacked autoencoders follow a similar structure, but with multiple hidden layers. The only strict requirement of the hidden layers is that the middle one, which will be used as the encoder layer, still has fewer nodes than the input. This structure can still be trained using backpropagation, but with more layers, it is likely to begin suffering from the vanishing or exploding gradient problem. As noted, the work by Hinton et al. for initialization of weights helps resolve this.

5.4.1 Sigmoid based Greedy Layerwise SAE Training

The steps for implementing the SAE training suggested by Hinton et al [33] is as follows

1 Define a network structure which conforms to the requirements of an SAE, with L layers

2 For the first hidden layer, train as you would for an RBM with 2 layers - with the input layer, and the first layer as the hidden layer, using CD-1 and SGD

3 For each layer l in $(2, L/2)$:

  3.1 Process the data through the previously trained layers using the forward propagation as defined in 5.2.1

  3.2 This processed data then forms the input to the l^{th} layer, which can be trained once again using CD-1 and SGD as if it were two layers

4 Once all the layers up until the encoder layer have been trained in this greedy layerwise fashion, mirror the weights and layers structures after the encoder to create a fully L layered FFN with pre-trained weights

5 This network can then be trained using the backpropagation and SGD algorithms, where cost of reproducing the network input is minimised

6 Once a minima or acceptable level of reconstruction has been reached, the network can be truncated as the encoder layer, and so the first $L/2$ layers are used as the SAE

Notably, this weight initialization will only be effective if the RBM and SAE networks use the same activation function, which due to the RBM implementation, needs to be a function that can output a probabilistic value in [0,1].

5.4.2 ReLU based SAE Training

ReLU activations differ from Sigmoid in that they are not fitting for probability estimations, which makes the algorithm suggest by Hinton et al. unsuitable. The process used here relies on effective weight initialization, and is a simplification of the above.

1 Define a network structure which conforms to the requirements of an SAE, with L layers

2 Use an effective weight initialization, such as Xavier or He (discussed fully in 5.5)

3 This network can then be trained using the backpropagation and SGD algorithms, where cost of reproducing the network input is minimised

4 Once a minima or acceptable level of reconstruction has been reached, the network can be truncated as the encoder layer, and so the first $L/2$ layers are used as the SAE

5.4.3 Denoising Autoencoders

As noted in 2.3.3, denoising can be used as an optimization technique for autoencoders in order to improve general performance and reconstruction. The methodology works by corrupting the input data for training, but using the non-corrupted data as the expected output sample. In doing so, this forces the autoencoder to learn more fundamental representations of the data rather then fitting to sample noise, hence "denoising". Two of the more typical techniques for achieving this have been implemented, as detailed below. In both cases, the noise is reapplied each time training samples are chosen.

Additive Gaussian Noise With Gaussian noise, samples are corrupted such that a degree of variance is added to the input according to a parameterized Gaussian distribution. In this case, the distribution variance is a configurable training parameter.

$$\tilde{x}|x \sim \mathcal{N}\left(x, \sigma^2 I\right) \tag{42}$$

Masking Noise With masking noise, a fraction of the features in the data are set to 0, thus masking them for that sample. The percentage of features chosen is a configurable training parameter.

5.5 Variance Based Weight Initializations

Recent research, as noted in [41], has shown that weights can be initialized to maintain expected variance between the input and output layers. These methodologies have the immediate advantages of simpler implementation, as well as faster computation (as no pre-training is required). The also allow for effective weight initialization of non-probabilistic activation functions, such as ReLU. Whether they result in better reconstructions or predictions is less clear (especially as the linearity assumption would prove faulty), and so the methods are tested here as well.

A common initialization heuristic is to use a uniform, and but not layer agnostic, initialization such as 43 below. While simple enough, it's been shown that this does not always lead to the best training results, and can be outperformed by variance based initializations [28].

$$\text{weights} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right] \text{ , for a layer with } n \text{ nodes} \tag{43}$$

5.5.1 Initialization Rationale

The variance balancing methodology is based on balancing the variance of a linear network. For input X, with n components and linear neurons with weights W, and

output Y,

$$Y = W_1X_1 + W_2X_2 + ... + X_nW_n \tag{44}$$

It can thus be shown, that for $i.i.id$ samples with mean 0, then

$$Var(Y) = nVar(W_i)Var(X_i) \tag{45}$$

For the variance of both input X and output Y to be balanced on the forward and backward propagation, then it is necessary for

$$Var(W_i) = \frac{1}{n_{in}} = \frac{1}{n_{out}} \tag{46}$$

In the instance where there are not an equal number of nodes in the two layers, the average can be taken, such that

$$Var(W_i) = \frac{2}{n_{in} + n_{out}} \tag{47}$$

5.5.2 Initializations

Xavier Considering that $Var(U(-a, a)) = a^2/3$, then ensuring balance variance for weights as per equation (47) provides us with the Xavier Glorot initialization [28] which can be used for Sigmoid activations, and is defined as follows

$$w_{ij} \sim U(-r, r), r = \sqrt{\frac{6}{n_i + n_j}} \tag{48}$$

where n is the number of nodes in the i^{th} or j^{th} layer.

He The initialization for ReLU is different on accounts of the function being equal to zero for half it's potential input range - in this case it makes sense to double the weight variance, and so the He [41] initialization is used

$$w_{ij} \sim U(-r, r), r = \sqrt{\frac{6}{n_i}} \tag{49}$$

He-Adjusted An adjusted initialization is presented here, dubbed 'He-Adjusted', where the ReLU initialization uses a mean of the input and output layers in order to scale the weight variance. For networks with constant layer sizes, the intialization is the same as He, though for SAE networks which have layer size changes by definition, the He-Adjusted initialization will result in more appropriately sized weights.

$$w_{ij} \sim U(-r, r), r = \sqrt{\frac{6}{(n_i + n_j)/2}} \tag{50}$$

5.6 CSCV & PBO

The combinatorially symmetric cross-validation (CSCV) method developed by Baily et al., as discussed in 2.6.1, can be used to assess the likelihood of backtest overfitting

through comparison of IS and OOS return metrics. Formulaically, the definition of backtest overfitting is given by

$$\sum_{n=1}^{N} E[\overline{r_n}|r \in \Omega_n^*]Prob[r \in \Omega_n^*] \leq N/2 \tag{51}$$

Where the search space Ω consists of the N ranked strategies, and their ranked IS performance r and OOS performance $\bar{r}$. This allows the PBO, using the Bayesian formula, to be defined as

$$PBO = \sum_{n=1}^{N} Prob[\overline{r} < N/2|r \in \Omega_n^*]Prob[r \in \Omega_n^*] \tag{52}$$

Notably, the above definitions consider IS as the data made available to the strategy selection, rather than the models calibration (e.g. the full IS dataset, rather than, by was of example, the number of days used in a moving average). This allows the model-free and non-parametric nature of the definition.

They further developed the CSCV framework as a methodology to reliably estimate the probability used in PBO, which allows a concrete application of the concept. The CSCV framework does not require using the typical hold-out strategy (and thus avoids credibility issues), and is ultimately able to provide a bootstrapped distribution of OOS performance. The full methodology is shown in algorithm 4 below [1].

Algorithm 4: CSCV

Input: N configuration trials over T time periods with P&L

1 Generate a TxN performance series matrix, M, representing the profits and losses by the N configuration trials over T time periods

2 Partition the M matrix by rows into S submatrices, each of even size $(T/S * N)$

3 Generate the combinations C_S of M_S, in groups of size $S/2$, for total $\binom{S}{S/2}$ of combinations

4 For each combination in $c \in C_S$:

   a Form a training set J by joinined $S/2$ M_S submatrices, in their original order. J is a matrix of order $(T/S)(S/2) * N$

   b Form the test set $\bar{J}$ as the complement of J in M, once again in the original order

   c Form a vector of R^c of performance statistics of order N, where the Nth component R^c_n of R^c reports the performance associated with the n^{th} column of J

   d Repeat [c] for $\bar{J}$ to derive $\bar{R}^c$ and $\bar{r}^c$

   e Determine the element n^* such that $R^c_n \in \Omega^*_n$ - i.e. n^* is the best performing strategy IS

   f Define the relative rank of $\bar{r}^c_{n^*}$ by $\bar{\omega}_c := \bar{r}^c_{n^*}/(N+1) \in (0,1)$. This is the relative rank of the OOS performance associated with the strategy chosen IS, which should systematically outperform OOS if no backtest overfitting has taken place

   g Define the logit $\lambda_c = ln\frac{\bar{\omega}_c}{(1-\bar{\omega}_c)}$. High values here indicate consistency between IS and OOS performances (and so low ovefitting)

5 The logit values can now be used to compute the distribution ranks of OOS, by collecting all λ_c for $c \in C_S$. The relative frequency for λ occurring across all C_S is

$$f(\lambda) = \sum_{c \in C_S} \frac{\chi_\lambda(\lambda_c)}{\#(C_S)} \tag{53}$$

where χ is the characterization function and $\#(C_S)$ is the number of elements in C_S, and so $\int_{-\infty}^{\infty} f(\lambda)d\lambda = 1$

Result: Reconstructed distribution for $\langle vh \rangle$

The CSCV framework and results thus allows the consideration of several notable statistics. First and foremost, the PBO may now be estimated using the CSCV method and using an integral over the $f(\lambda)$ function as defined above which offers a rate at which the best IS strategies underperform the median of OOS trials. The PBO is estimated using

$$\Phi = \int_{-\infty}^{0} f(\lambda)d\lambda \tag{54}$$

If $\Phi \approx 0$, it is evidence of no significant overfitting (inversely, $\Phi \approx 1$ would be a sign of probable overfitting). Critically then, a PBO measure may be used in a standard hypothesis test to determine if a model should be rejected or not. This can be extended, as shown by Bailey et al., to show the relationship between overfitting and performance degradation of a strategy. It becomes clear that with models overfitting

to backtest data noise, there comes a point where seeking increased IS performance is detrimental to the goal of improving OOS performance.

5.7 Money Management Strategy and Returns

In line with the general approach here, the Money Management Strategy (MMS) has been designed to be relatively simple, such that the results are more indicative of the underlying modelling rather than intricate trading strategies. The MMS follows an arithmetic long strategy of buying any stock for which the predicted $t + D$ price is above the current price, and selling the stock at $t + D$. Trading costs have been included at 10% capital costs per annum for borrowing to purchase, and 0.45% for transaction costs. Results are compared to a "perfect" model which has exact knowledge of the price in D days, which represents an upper bound on performance. This MMS implentation does not make any attempt to account for volume in the trades, and only trades one asset unit at a time.

Input Variables The MMS takes the following values as input for each time point t and prediction point D

1 $\mathtt{y}_t$, the closing price for t

2 $\mathtt{y}_{t+D}$, the closing price in D days

3 $\hat{\mathtt{y}}_{t+D}$, the predicted closing price in D days

Calculated Stock Variables Using these, the MMS calculates and uses the following:

1 A flag indicating whether a trade will take place on t

$$\mathtt{trade\_long}_t = bool(\hat{y}_{t+5} > y_t) \in (0, 1) \tag{55}$$

2 A flag indicating whether a trade will take place on t in the perfect model

$$\mathtt{trade\_long}_{p,t} = bool(y_{t+5} > y_t) \in (0, 1) \tag{56}$$

3 The observed return at $t + D$

$$\mathtt{r}_{o,t+D} = y_{t+D} * trade\_long_t \tag{57}$$

4 The expected return at $t + D$

$$\mathtt{r}_{e,t+D} = \hat{y}_{t+D} * trade\_long_t \tag{58}$$

5 The perfect return at $t + D$

$$\mathtt{r}_{p,t+D} = y_{t+D} * trade\_long_{p,t} \tag{59}$$

6 The cost incurred in buying the stock at time t

$$\mathtt{cost}_t = y_t * trade\_long_t \tag{60}$$

7 The cost incurred in the perfect model at t

$$\texttt{cost}_{p,t} = y_t * trade\_long_{p,t} \tag{61}$$

8 The possible capital and transaction costs

$$\texttt{trading\_cost} = y_t * (0.1/365 * 2) + y_t * (0.45/100) \tag{62}$$

9 The full cost of trading the stock at t in the predictive model

$$\texttt{fullcost}_t = (y_t + trading\_cost_t) * trade\_long_t \tag{63}$$

10 The full cost of trading the stock at t in the perfect model

$$\texttt{fullcost}_{p,t} = (y_t + trading\_cost_t) * trade\_long_{p,t} \tag{64}$$

Calculated Strategy Variables The MMS is then implemented using these calculated values which are aggregated at each time point t for all stocks, s, as follows:

1 The expected return for $t + D$ from all stocks traded at t

$$\texttt{R}_{e,t+D} = \sum_{s \in stocks} r_{s,e,t+D} \tag{65}$$

2 The observed return for $t + D$ from all stocks traded at t

$$\texttt{R}_{o,t+D} = \sum_{s \in stocks} r_{s,o,t+D} \tag{66}$$

3 The perfect return for $t + D$ from all stocks traded at t

$$\texttt{R}_{p,t+D} = \sum_{s \in stocks} r_{s,p,t+D} \tag{67}$$

4 The cost of all trades at t

$$\texttt{C}_t = \sum_{s \in stocks} cost_{s,t} \tag{68}$$

5 The cost of all trades at t in the perfect model

$$\texttt{C}_{p,t} = \sum_{s \in stocks} cost_{s,p,t} \tag{69}$$

6 The cost of all trades at t including transaction costs

$$\texttt{FC}_t = \sum_{s \in stocks} fullcost_{s,t} \tag{70}$$

7 The cost of all trades at t, including transation costs, in the perfect model

$$\texttt{FC}_{p,t} = \sum_{s \in stocks} fullcost_{s,p,t} \tag{71}$$

The final model return is thus established as $\sum_t R_t / \sum_t C_t$, for any variations of observed, expected or perfect returns as well as base and full costs. These model returns are used to establish the performance of the different predictive models which were trained.

An additional consideration of the MMS is the daily rates generated by the strategy, which incorporate the costs of any trades incurred, as well as any returns seen at t, such that

$$\texttt{dailyrate}_t = R_{t-D} - C_t \tag{72}$$

6 Full Process

Having covered the technical implementations in Chapters 4 and 5, a higher level overview of the end to end experimental process can be detailed. The process here rests on two key principles: implementing a generalised version of a system which could offer exploration of more complex techniques, and ensuring an effective modularisation of steps such that the process can be reconfigured accordingly while maintaining its integrity. In doing so, a separable system is created which brings together data reduction, deep learning with pre-training/weight initialization, online learning and back test overfitting validation.

6.1 Parameter Space Exploration

The parameter space is explored using a phased gridsearch approach. For each stage, the relevant parameters are specified as a set of values (or one, if so chosen), and all sets are then used to generate the full combinatorial space, such that each possible combination of the specified parameters is tested.

1 Stage 1: The data configuration (i.e. data windows, prediction point, scaling, data split points) as well as the SAE configuration (network size, learning rates, learning optimization parameters, SGD epochs) are set in Stage 1 and used to train the SAE networks.

2 Stage 2: The preferred SAEs are chosen from Stage 1 manually, and determine the data configuration used for Stage 2. These are then used to encode the datasets, which will be used for FFN training. The FFN SGD and OGD parameters are set in this stage (network size, learning rates, SGD epochs etc.), and will be combined combinatorially with the SAEs that were chosen for testing as well.

6.2 Synthetic Data

The use of synthetic GBM generated data offers a convenient way of exploring the parameter space without running configuration experiments on the actual data which might then raise the probability of backtest overfitting. The synthetic data has enough similarity that it can allow a broad exploration to get ideas of networks sizes, learning rates, data considerations and so on. This is paricularly useful in light of the concerns around increasing the number of particularly poor performing networks when it comes to calculating PBO, as discussed in section 8.7. That said, there are characteristic differences between GBM generated data and actual financial time series. With these in mind, the results section will make comparisons of the two in order to offer a sufficient understanding for when synthetic data can be relied upon to guide future training.

6.3 Data Preparation

The fulldataset goes through several steps of pre-processing - these are covered in more detail in 4.1, though the steps are included here:

1 The data is processed into day to day log fluctuations

2 At each time point, rolling historic fluctuation summations are calculated (e.g. the past 1, 7 and 30 days)

3 At each time point, rolling future fluctuation summations are calculated (e.g. the next 5 days)

4 The dataset is truncated to only include points with all aggregations or predictions available

5 The dataset undergoes scaling - the different functions are detailed in 4.1.2, though `limited normalize` was used for most of the results produced

6.3.1 Data Window Aggregations

The time aggregations were chosen according to domain knowledge - it's expected that prices may move in daily, weekly, fortnightly, monthly and quarterly patterns as trading is actioned by day traders such as speculators to institutional investors such as mutual funds. Indeed, it can be shown that markets can be modelled such that they are hierarchical in space and time, with actors at different heirarchical levels within the market interacting under different causitive structures [85]. The effect is such that price trends and patterns can be expected to exist in time at multiple levels, with price being set as an emergent property of these hierarchies. Taking this into account, variations of 1, 5, 10, 20 and 60 working days were considered throughout the configuration tests, in the following combinations:

1. [1,5,20]
2. [5,20,60]
3. [10,20,60]

6.3.2 Point Predictions

Various configurations were tested in terms of the historic summations used, while the fluctuation predictions were typically 5 days. It's worth noting that the prediction horizon is configurable, and in the case of implementing a more complex trading strategy, one would likely choose to predict multiple points across the horizon in order to to develop a distribution. In this case, the MMS was kept purposefully simple, and so didn't warrant the more intensive implementation.

One point of concern, is that point predictions at 2 or 5 days in advance could open the process up to aliasing error, as it is unlikely to be predicting at the Nyquist frequency, and may end up fitting to the band ripple rather than the underlying signal. Conversely, 1 day predictions for closing price might offer a high degree of accuracy, but are not practical for trading due to market auction and bidding structures. The system implementation veered on the side of practicality in this case, and was configured for longer term predictions despite the risk of aliasing. While some of the configurations were run with 2 day predictions, it was seen that once costs were considered it was difficult for even the benchmark to produce desired P&L figures and so a 5 day prediction was generally implemented in subsequent tests.

6.3.3 Scaling

When testing the two limited scaling methodologies, standardization and normalization, it was found that the results for standardizing tended to be notably worse than normalization (discussed in 8.2). It's possible that outliers in price fluctuations and

changing variance through time (which is not captured effectively through the limited variation) resulted in a less informative representation and worse performance of the FNN models which can be prone to error maximisation. As a result of this, the limited normalization process was used for most of the tests run. It's worth noting that an implementation which incorporates processing the data through the Error Function in order to reduce the impact of outliers would likely prove a worthwhile endevour, and could be easily swapped in due to the systems modularity.

6.4 Data Segregation

Once processed, the dataset is split into 2 portions:

1 Training: The first set is used for the SAE training, as well as the initial SGD training on the predictive FFN network. This is the IS dataset and is used as what would typically be the historical training set.

2 Prediction: The second set is used only for the OGD training in order to generate returns from the predictive network. This is the OOS dataset and is used as what would typically be the testing / validation set.

The predictions from both the Training and the Prediction sets are used by Bailey's CSCV method to assess overfitting (the use of the CSCV technique negates the need for a hold-out portion of the dataset [1]), however only the OOS prediction returns are used in order to assess model P&L figures and performance.

In this case, a simple split point of 60%/40% was chosen for the Training and Prediction sets respectively. In a more restrictive case, where there is less data to work with, it would be advised to consider a method such as MinBTL as developed by Bailey et al. [82] and the estimated number of configurations to be tested in order to decide an appropriate split point.

6.5 Unsupervised Learning: SAE Training

The Training dataset is used to train the autoencoder network using either RBM pre-training or weight initialization algorithms and SGD training, as defined in 5.4. The Training portion of the dataset, as noted in 6.4, is itself split up into a training and validation set for the SAE. The efficacy of the SAE on the validation set is required in order to determine which networks to choose for subsequent steps in the process. The full testing process here will rely on a generated set of best SAE networks at each encoding layer size, to be used for FFN training and prediction (chosen by a minimum MSE score). The benefit of the modularised system is emphasised here, as the SAE training will not suffer from limitations due to backtesting considerations: any amount of configurations or processes can be tested for feature extraction without concern.

Once the SAE networks have been defined and chosen, they can be used to reprocess both the Training and Prediction datasets such that the input is encoded, and the output is as before. These encoded datasets can then be used for the following steps in the process.

In a productionized system, where data is updated daily, there would be a set period after which the SAE network would be retrained to reflect more recent data. This step wasn't deemed necessary at the outset of developing this process, but results

detailed in section 8.4 suggest that it is a necessary implementation step.

6.6 Supervised Learning: Prediction Network Training

Typically, the SAE and predictive network might be presented as one and the same, however in the process presented we are able to separate the two and optimise for feature extraction prior to training for predictions. Having done so, the datasets can have their input encoded, but retain the same output. The predictive network is then trained using the Training set, as detailed in 5.2.3. There's no requirement to validate this part of the training, so the dataset is not split into subportions as it was for SAE training.

Once the predictive network is trained, the OGD process is run through the encoded Prediction dataset in order to generate the predictions for the assets prices that the model produces - thus emulating what would have occured in a live environment.

6.7 Price Reconstruction

In the interest of more intepretable results for figures such as P&L, the prices fluctuations for both IS and OOS predictions are re-scaled back to their original price fluctuation predictions using the reverse scaling detailed in 4.1.3. These prices are then recostructed using 4.1.4, such that they represent a predicted future price of the asset which can then be used by the MMS. It's worth nothing that the price reconstruction applies the predicted fluctuation to the known original price, such that it would represent a system that is run daily, rather then a system that is making rolling predictions over a period of time.

6.8 Money Management Strategy

The MMS, as described in 5.7, takes a proof of concept approach to the system being investigated. It is a simple long strategy with a set liquidation point, and no notable refinement. The naive approach is taken purposefully here, so as not to bias the perspective of the system as a whole by the effects of an impactful trading strategy. A more effective trading strategy would incorporate a shorting side as well, though ultimately this isn't necessary in order to obtain a necessary indication of efficacy for the system (and conceptually, a consistently losing long strategy would be of interest in any case).

It is important, in the interest of effective optimization and development, that the pattern recognition in the prediction portion of the system is not tightly coupled with making it profitable. With this in mind, the modularity of the system is continued, with a distinct separation between the prediction signal and the MMS implementation which relies only on the predicted prices of each stock for a time point. The comparisons for the MMS are made against a benchmark of the same MMS, but with perfect information. This allows an indication for how effective the prediction system is (and need be) in order to generate comparable returns.

Can we learn to beat best stock paper ? Paper

An effective production trading strategy would be likely to include stop losses or pyramiding stop losses for effective P&L optimization. An ideal implementation, and possible area for further work, would include an MMS strategy where the bands for this are defined using a machine learning technique (rather than being chosen from domain knowledge or heuristics).

6.9 CSCV & PBO

The CSCV and PBO parts of the system are essentially just an implementation of the algorithm detailed in 5.6, though with a somewhat novel application to the predictive network and MMS combination. The CSCV process uses the IS and OOS returns from the MMS, which in turn used the prices from the predictions network, which then constitutes the M matrix used by the algorithm. Conceptually, the whole system comes into place here, as the results from the CSCV process are now indicative of not only backtest overfitting in the trading strategy, but also in the prediction network and without having to consider the impact of many configuration tests for feature extraction.

6.10 Process Diagram

The flow diagram below provides a graphical representation of this process from the raw market data up until the price prediction points (the MMS calculations and CSCV process are not included). The process may be repeated accordingly in terms of SAE structures that are desired for testing, such that the "Best SAE Network" is different for different encoding layer sizes, or data aggregations used (as was the case in many of the experiments which will be discussed in section 8).

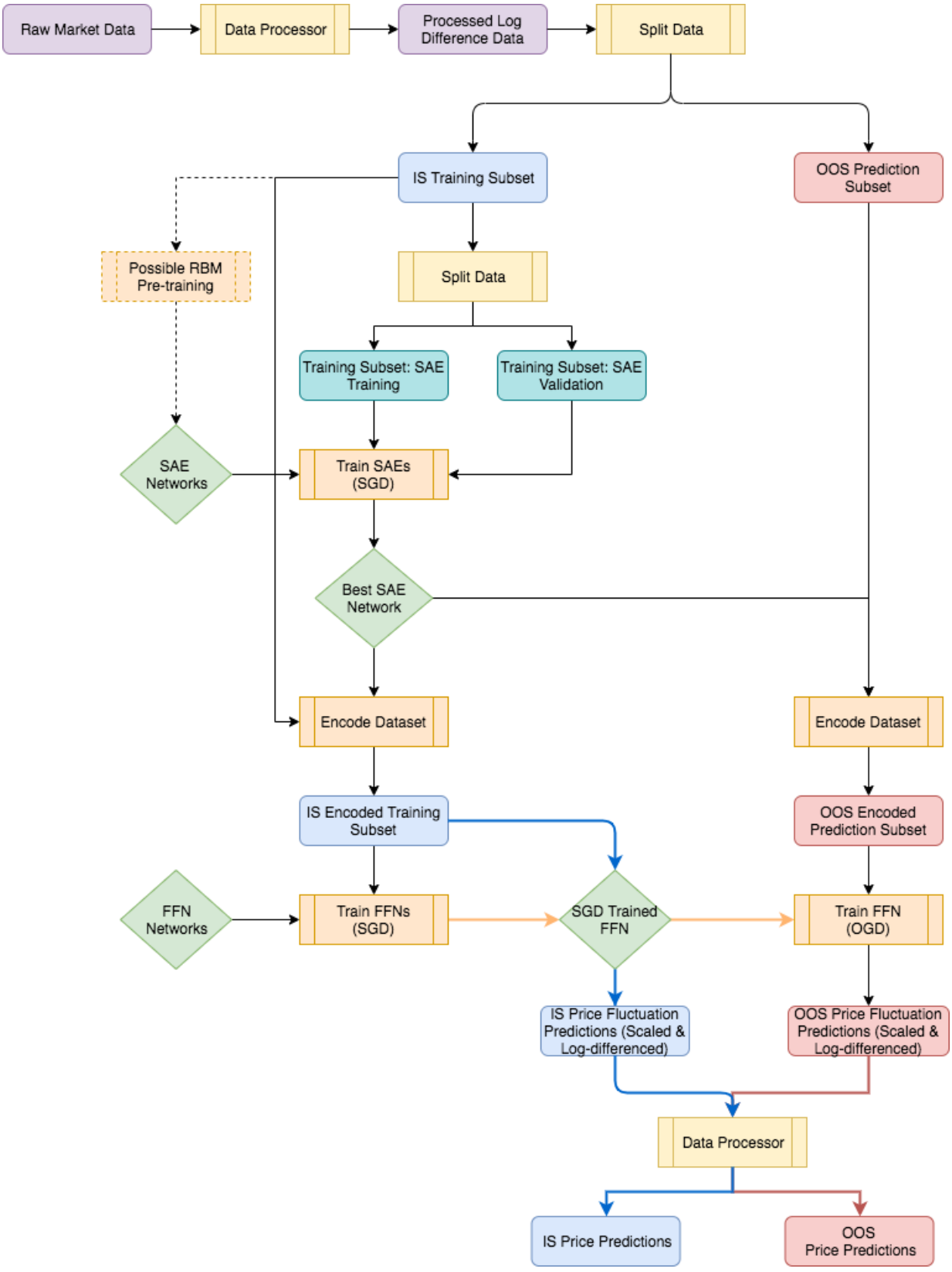


Figure 5: Overall Process Flow

7 Datasets Used

7.1 Synthetic Datasets

Data was generated and scaled, as per the methods detail in 4.2 and 4.1.2. The sets were generated with the following configurations, each for 5000 timesteps and with a 60/40 split on the Training and Prediction sets.

7.1.1 Synthetic6

The Synthetic6 dataset was configured to represent a combination of upwards, downwards and sideways Trends, each with high and low variance variations.

Trend Category	Variance Category	Trend Mean	Variance
Strong Upward	High	0.9	0.5
Strong Upward	Low	0.9	0.2
Upward	High	0.05	0.4
Upward	Low	0.05	0.1
Strong Downward	High	-0.8	0.55
Strong Downward	Low	-0.8	0.15

Table 1: Synthetic 6 Dataset Configuration

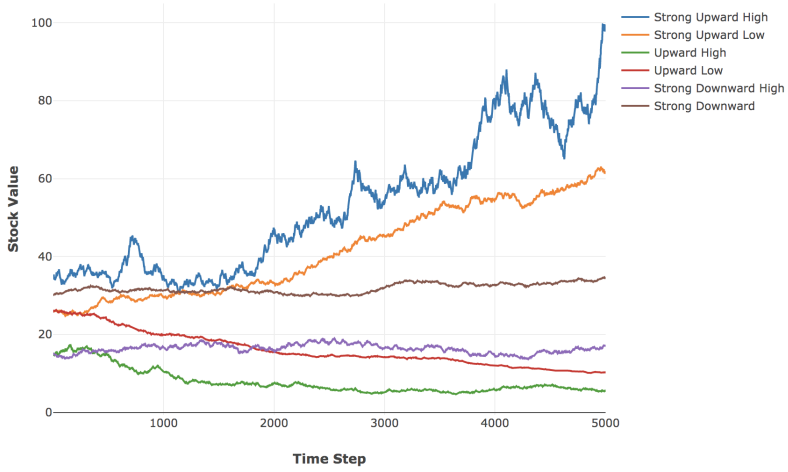


Figure 6: Synthetic 6 Prices

7.1.2 Synthetic10

The Synthetic10 dataset was configured to represent a wide array of behaviours, including very high (positive and negative) mean stocks, as well as much lower mean stocks (which may be more representative of typical behaviour). These were chosen to make sure the network learning is able to differentiate and correctly learn across different asset categories.

Trend Category	Variance Category	Trend Mean	Variance
Strong Upward	High	0.9	0.5
Strong Upward	Low	0.7	0.2
Upward	High	0.05	0.5
Upward	High	0.05	0.4
Upward	Low	0.04	0.1
Sideways	Low	0.02	0.15
Sideways	Low	0.01	0.05
Downwards	Low	-0.1	0.2
Strong Downward	Low	-0.4	0.15
Strong Downward	High	-0.8	0.55

Table 2: Synthetic 10 Dataset Configuration

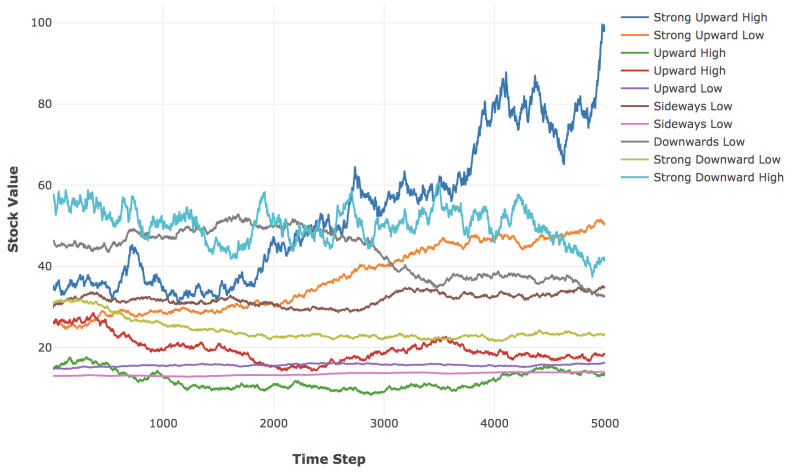


Figure 7: Synthetic 10 Prices

7.2 Actual Datasets

Several datasets have been used using JSE closing price relative data for 2003-2018 . They were processed the same way as the Synthetic sets, following the steps set out in 4 and with a 60/40 split on the Training/Prediction datasets.

add ref

7.2.1 Actual10

This is the primary closing price dataset that has been used through out the experiment process, which focused on choosing more prominent stocks from multiple sectors.

Code	Company	Sector
AGL	Anglo American	Resources
BIL	BHP Billigton	Resources
IMP	Impala Platinum Holdings	Resources
FSR	FirstRand Limited	Finance
SBK	Standard Bank	Finance
REM	Remgro Limited	Finance
INP	Investec	Finance
SNH	Steinhoff International Holdings	Retail
MTN	MTN	Communication Services
DDT	Dimension Data	Tech

Table 3: Actual 10 Dataset Configuration

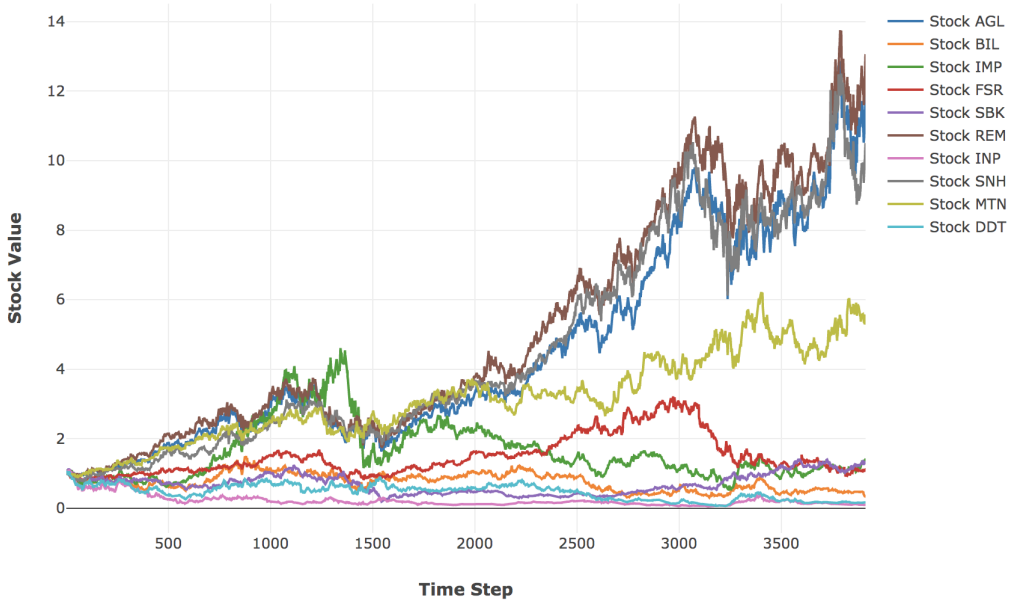


Figure 8: Actual 10 Prices

7.2.2 AGL

Using AGL data from JSE close price relatives.

7.2.3 AGL&ACL

Using AGL and ACL data from JSE close price relatives.

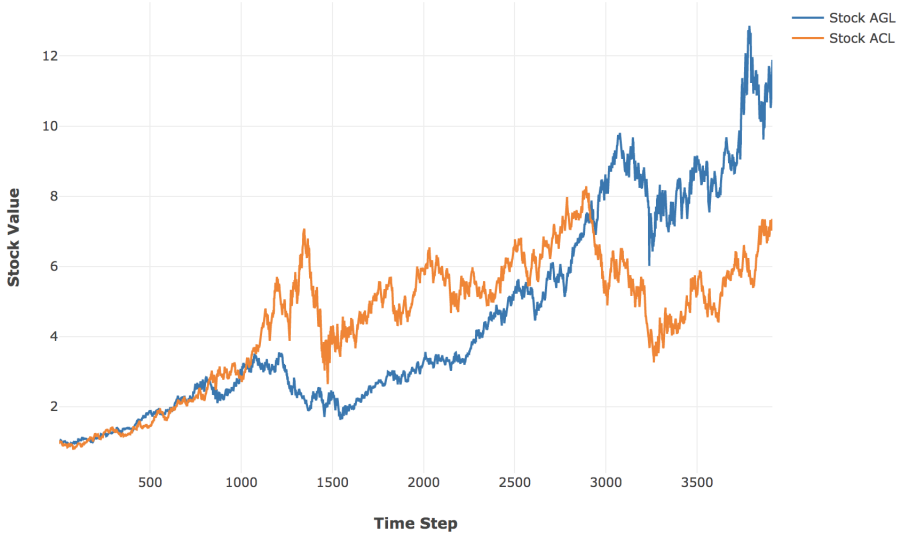


Figure 9: AGL & ACL Prices

7.2.4 Scaling10

Using the following assets from JSE close price relatives:

- ACL, AGL, AMS, AOD, BAW, BIL, BVT, CFR, CRH, DDT

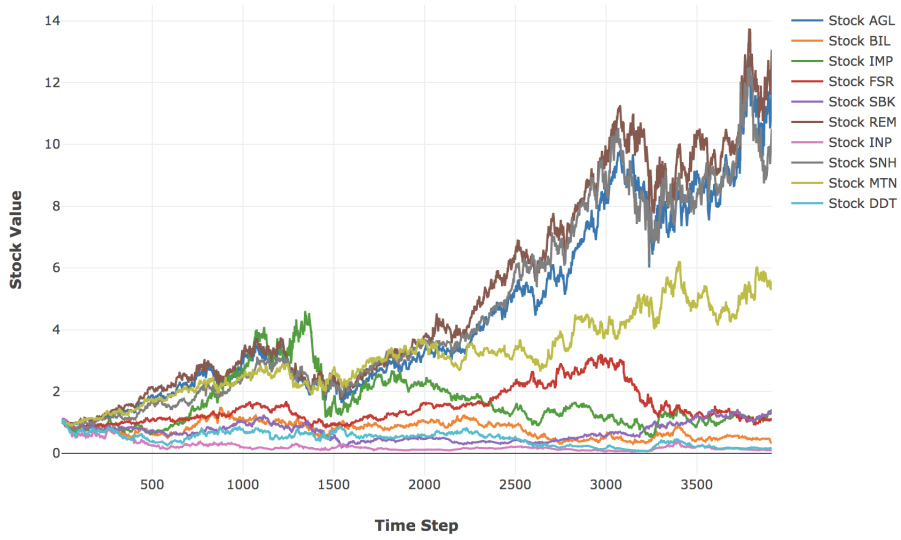


Figure 10: Scaling 10 Prices

8 Results

8.1 Introduction

The results section offers several key takeaways, mostly discussed in order of application (such that some findings will have been taken into account for later sections in this chapter). There are some general aspects worth keeping in mind throughout the result chapter:

1 Sample sizes for some configurations may differ greatly across results (particularly as results are aggregated across training sets and phases), though these are noted in the figures as well as the configurations in the appendix.

2 P&L is a relative unit measurement, rather than a particular currency (as noted in 4.3).

3 The MMS, as described in 5.7, only trades 1 stock unit at a time, and so any P&L figures are relative indications rather than absolute limits.

4 Results for synthetic data are compared against actual data when appropriate, such that it might guide when synthetic data can and should be used for parameter space exploration (as discussed in section 6.2).

8.1.1 Overview

1 Section 8.2 discusses the impacts of network activation functions and scaling choices and shows the benefits of Leaky ReLU activations and Normalized scaling, as well as the impacts of the limited scaling techniques.

2 Section 8.3 discusses the effects of various network weight methodologies. RBM pre-training is shown to be detrimental for financial series, and comparison are made of the variance based initializations with He-Adj showing the best all-rounded performance.

3 Section 8.4 discusses the effects of the SAE encoding performed, with feature reduction shown to be possible and effective but subject to changing dyamics over time.

4 Section 8.5 discusses the effects of network size, training parameters as well as the effects of learning optimizations on network performance.

5 Section 8.6 shows the negligble impact of extensive training on historical data, noting that reducing training epochs and datasets has limited effect, and that the online learning is most impactful for P&L. The effects of data aggregation windows on synthetic and actual data are also discussed.

6 Section 8.7 presents the PBO results for configurations run, and disusses the potential shortfalls and dynamics of the PBO methodology.

7 Section 8.8

add DSR when possible

8 Section 8.9 presents the general P&L results for experiment configurations and compares them to the benchmark.

9 Section 8.10 summarises the results presented, noting major differences between synthetic and actual data.

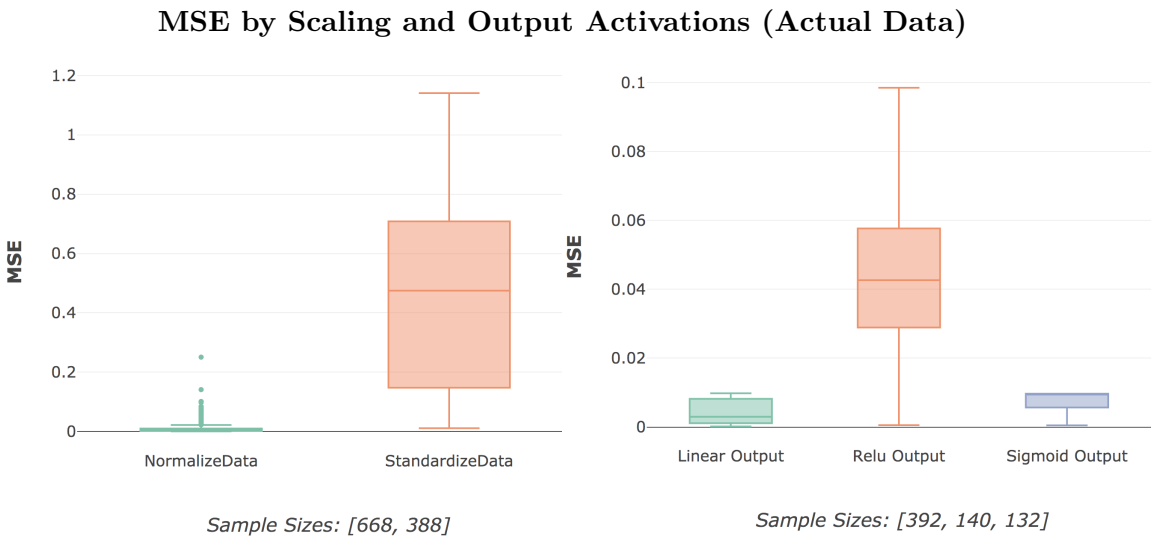
8.2 Linearity, Complexity and Structure of Data

8.2.1 GBM Generated Data

Geometric Brownian Motion (GBM), as discussed in 4.2, has been used to simulate synthetic datasets for the purposes of testing implementations and configurations. GBM is a popular choice for synthetic financial data as it is a Markov process, thus following a random walk and is generally consistent with the efficient market hypothesis in that the next price movements is conditionally independent of past movements. In line with this though, the series will exhibit a constant drift with price shocks according to it's stationary configuration, and changes in price that are not independent. The effect seen, and discussed more below, is that reconstruction of aggregated GBM data was able to performed quite effectively through linear activations, so long as the network (and encoding layers) were large enough to allow for sufficient transformations. Linear activations weren't tested further on actual data, as there is no reason to believe the time series would continue to exhibit the constant drift with non-independent price jumps over time. Further to this point, it is worth pointing out that as GBM are non ergodic series, one should be wary of considering ensemble based predictions with much confidence [86]. So while the GBM data was used for assessment of some prediction networks, these results should be considered contextually.

8.2.2 Effects of Activation Functions and Scaling

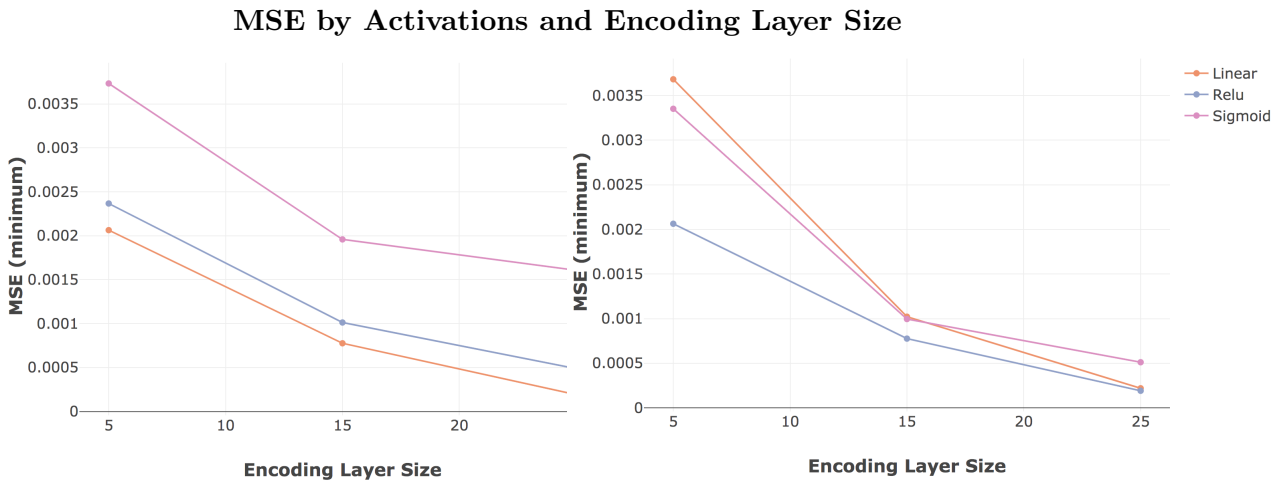
A range of activation functions (Linear, ReLU and Sigmoid) (as described in 5.2.2),as well as the scaling functions (as described in 4.1.2), were tested on actual data for SAE networks in order to determine the best configuration choices going forward. Figures 11 and 12 below show that Normalizing data offers far better performance than Standardizing, that Linear activations are best for the encoding and output layers, and that ReLU activations largely outperform Sigmoid or Linear activations for the hidden layers. This makes sense with the non-linear benefit at hidden layers but with less loss of error signal and information at the output and encoding layers. These results were used for subsequent parameter choices.



(a) Scaling Technique

(b) Output Activation

Figure 11: Dataset: Scaling10 dataset (7.2.4), Configuration 1 (10.2.1)
Figure (a) here shows SAE MSE performance according to the scaling techniques as described in 4.1.2. As discussed more fully in 6.3, the use of standardization for scaling the data doesn't allow for effective outlier treatments, resulting in the significantly worse performance seen here.
Figure (b) shows the effects of the output activation for configurations using Normalized scaling. The ReLU activations also seemed to result in very poor performance, most likely due to the loss of error signal in the output layer where it is most critical.

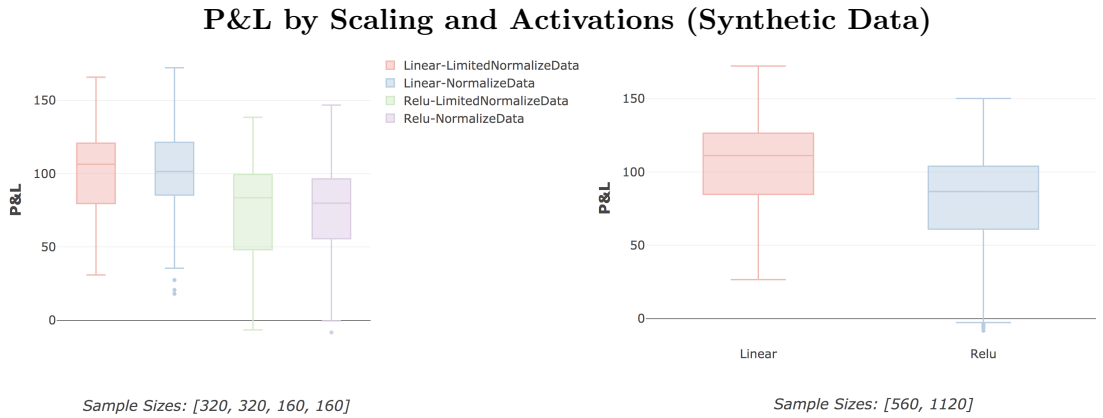


(a) Encoding Activations **(b) Hidden Activations**

Figure 12: Dataset: Scaling10 dataset (7.2.4), Configuration 2 (10.2.2)
Figure (a) shows the best (minimum) MSE scores for different encoding activations at the different encoding layer sizes (with a network input of 30). There is a consistent out performance of Linear activations in the encoding layer, which is expected from literature [33], and in line with reducing loss of error signal at critical points.
Figure (b) shows the best (minimum) MSE scores for different hidden layer activations at the different encoding layer sizes (with a network input of 30). At larger encoding layers, there is a competitive performance from fully linear networks, where they are less pressured to take advantage of non-linear effects. As the size of the encoding layer is reduced, the advantages of non-linear activations become more apparent, with outperformance by ReLU and Sigmoid activations. Sigmoid activations are known to suffer from learning slow down as a result of saturation and so have increased sensitivity to vanishing gradients [29]. SAE networks are deep by nature, and so it is not surprising that the Sigmoid activations are resulting in worse performance over the same training period when compared to ReLU activations.

8.2.3 Predictive FFN Activations and Scaling

In Figure 13, a comparison scaling techniques as well as of Linear and ReLU activations are made for synthetic data in both smaller and larger sized networks. In line with the characteristics of the synthetic data noted in 8.2.1, results for networks using a Linear Activation often outperformed the networks using non-linear activations such as ReLU or Leaky ReLU (activation functions are detailed in 5.2.2).



(a) Encoding Activations

(b) Hidden Activations

Figure 13: Dataset Synthetic6 (7.1.1)), Configuration 3 (10.2.3) & Configuration 4 (10.2.4)
Figure (a) shows the impact of the limited scaling technique (as described in 4.1.2) in comparison to the non-limited version. The implementation of this is not a choice when interested in simulating a real world implementation, but it is still of interest to note the impact of this issue. Additionally, the use of a Linear output layer has been shown to assist in reducing the impact of this effect.
Figure (b) offers a comparison of Linear and ReLU hidden layer activations on P&L, with the linear activations resulting in notably better P&L when compared to ReLU activations. Unlike the SAE networks, this persists even when the network size is decreased - this difference highlights the effects of GBM data and that it can be represented linearly, as per 8.2.1, whereas we would see actual stock data to benefit from a non-linear representation. Linear configurations for actual data were not run, as real stock data is not expected to follow patterns that are linear through time, thus offering little value.

8.2.4 Leaky ReLU vs ReLU

Leaky ReLU activations were implemented, and showed negligible effects for SAE, which can be seen in the Appendix 10.1.1. For the predictive network, the LeakyReLU did offer some more significant improvements, as seen in Figured 14 below, and so has been used as the hidden layer activation of choice for subsequent experiments.

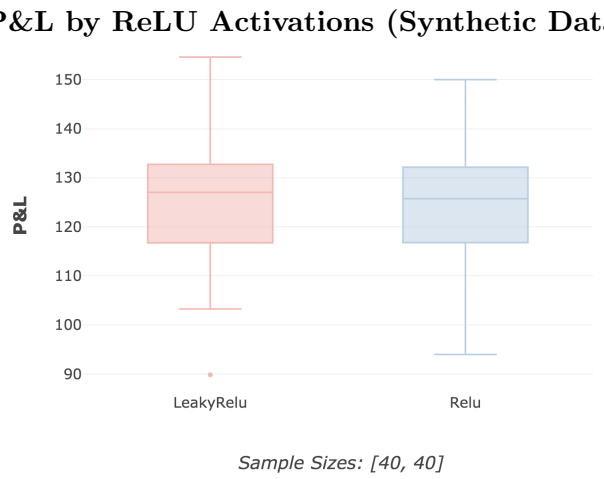


Figure 14: Dataset Synthetic6 (7.1.1) ; Configuration 5 (10.2.5)
The plot shows the FFN P&L, grouped by activation, showing some improvements from the Leaky ReLU activation. The effect is most noticeable in reducing the lower bounds of performance (10.8%), though has a clear benefit in the upper bounds as well (3.4%).

8.3 Weight Initialization Techniques

8.3.1 RBM Pretraining for Sigmoid Networks

While previously considered the best approach for training deep neural networks, the methodology of greedy layerwise RBM pre-training for Sigmoid SAE networks (as described in [33]) had detrimental effects on network performance, as seen below in Figure 15.

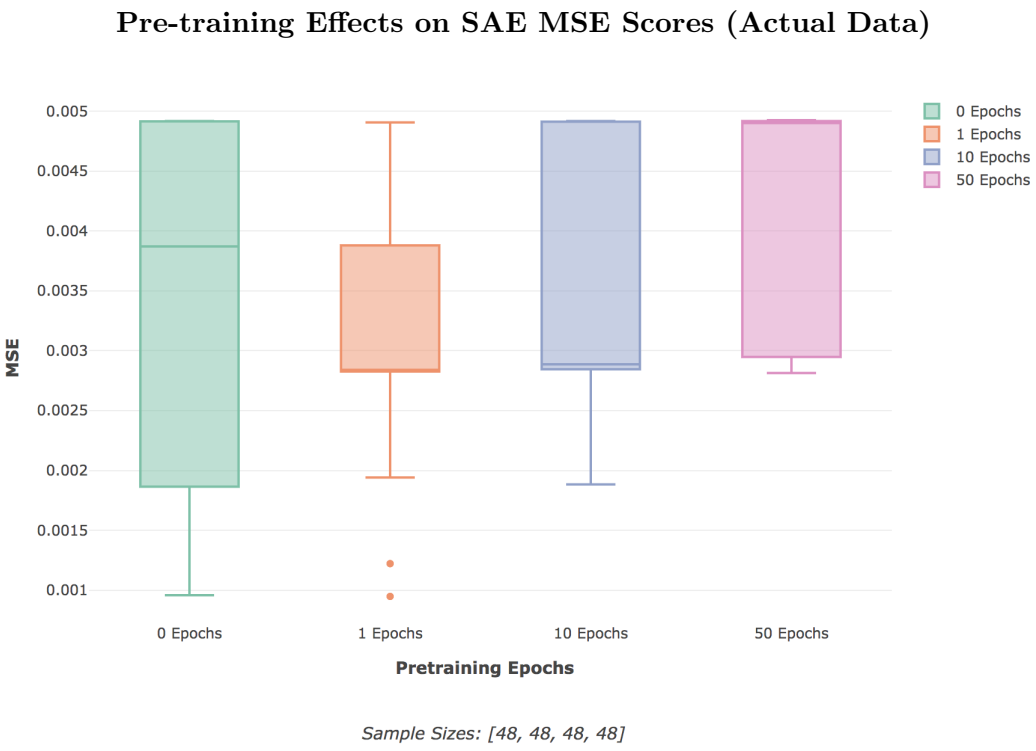


Figure 15: Dataset AGL&ACL (7.2.3); Configuration 6 (10.2.6)

The boxplots here show the summary of configuration performances, by minimum MSE achieved, grouped according to the number of pre-training epochs which the network had. There is an evident benefit to having no pre-training in this scenario as there is a clear decrease in performance as the number of epochs increase (the low value outliers that can be seen for the 1 epoch configuration were with learning rates which were low enough to approximate no epochs).

The RBM pre-training technique does assume that data is Independent and Identically Distributed (IID), and does ultimately traverse a different solution space and loss function. While it may be effective in some contexts, it was shown to result in a counterproductive exploration of the weight space for this non-IID dataset, and that ultimately the financial time series data is pathological for RBM pre-training. It may also emphasise the value of more recent data over historical data usage, as discussed more in Section 8.6[18].

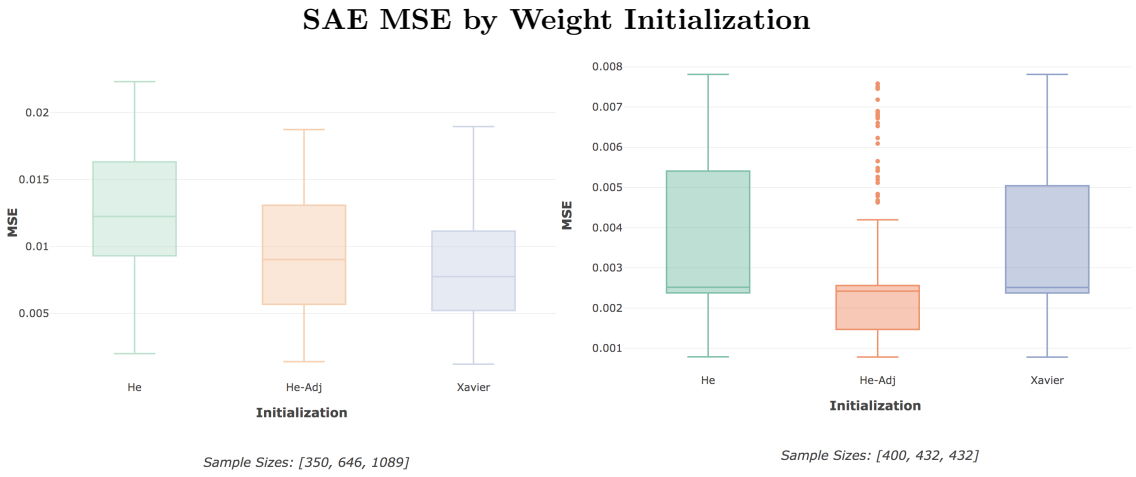
[18]The RBM greedy layerwise training implementation was effectively tested and validated on known datasets such as MNIST, example results for which can bee seen in Section 10.1.2 of the Appendix

8.3.1.1 Sigmoid Activation Functions

Due to the poor performance seen in Sigmoid function based SAE's here, as well as the poorer results when compared to ReLU activations (noted in 8.2.2), Sigmoid functions were largely excluded from further configuration testing.

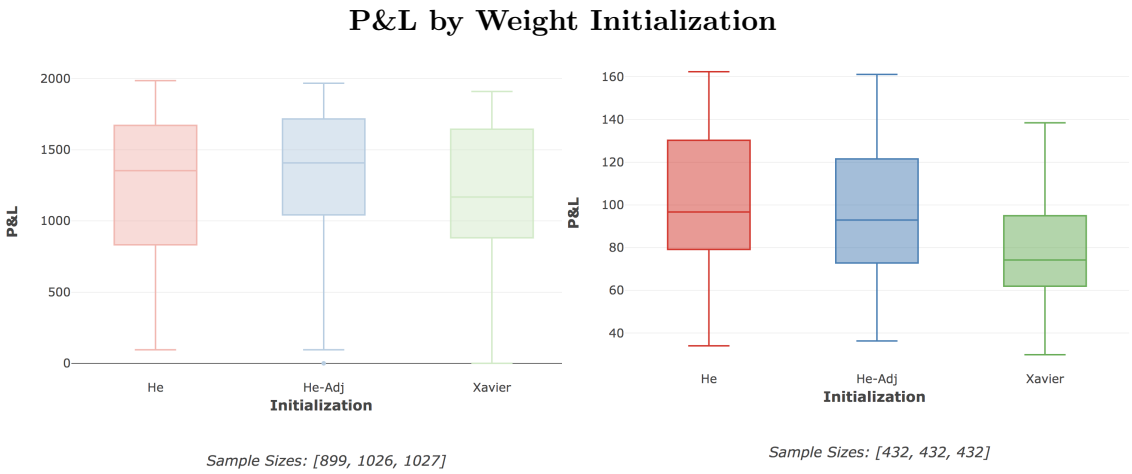
8.3.2 Variance Based Weight Initialization Techniques

More recent research has focused on the use of weight initialization using variance based methodologies, as discussed fully in 5.5. Our expectation, theoretically, is that the He initialization will generally outperform Xavier due to it being more appropriate for the ReLU activations being used. Additionally, in networks with varying layer sizes, we expect He-Adj and possibly Xavier to outperform He which is subject to imbalanced initializations. Ultimately, He-Adj as presented in 5.5, should present the best of both for an initialization that is suited to Leaky ReLU activations and the varying layer sizes found in SAE networks.



(a) Synthetic Data **(b) AGL**

Figure 16: Dataset: Synthetic10 (7.1.2), AGL 7.2.2, Configuration 7 (10.2.7) & Configuration 8 (10.2.8) Training SAE networks across the Synthetic10 and AGL datasets, we mostly see the expected patterns emerge. He-Adj consistently outperforms He due to being better suited to the network structures, as seen in the plots above. We also see that He-Adj clearly outperforms Xavier for AGL data, but has performance that is mostly the same (or even marginally worse) than Xavier for the synthetic dataset. Similar trends as AGL are seen when SAEs were trained for the full 10 asset dataset, which can be see in the Appendix, Section 10.1.2.



(a) Synthetic Data **(b) AGL**

Figure 17: Dataset: Synthetic10 (7.1.2), AGL 7.2.2, Configuration 9 (10.2.9) & Configuration 10 (10.2.10) Figure(a) shows P&L performance for synthetic data. The networks with more consistent layer sizes result in the performance between He and He-Adj becoming comparable, while both outperform Xavier (as expected, with ReLU activations). Figure (b) The P&L results for AGL data shows clear outperformance of Xavier by the He based initializations, but a less clear comparison betwen He and He-Adj.

The results here mostly conform to the theoretical expectations, though there are some inconsistencies. As discussed more fully in 8.2, the GBM data in synthetic datasets will have assets that are configured to certain mean and variances. As these movements are processed, scaled and aggregated over time through the data processing, the movements will generally be more similar and less complex than those of actual stock data which will show greater variance over time (this is shown more concretely in 8.6). The learning process then is not required to be able to pass through error signals that change as dynamically over time, and so there is less pressure on the initial starting weights. This consideration may account for the unexpected performance of the Xavier initialization in the Synthetic SAE networks. Further, some of the assumptions of these techniques, such as data being IID, may not apply in the first place, making their results less predictable. Due to the generally better or comparable performance of He-Adj, it was the only initialization chosen for further experiments.

8.4 Feature Selection

The efficacy of the feature selection performed by SAE networks, in so far as it affects ability to generate P&L, is displayed in the graphs below. The results are grouped by the size of the SAE encoding layer (i.e. the number of features that the input data was reduced to), and the OGD learning rate which produces a significant interaction in terms of P&L. The '0' encoding layer grouping indicates that no feature selection was done (i.e. there was no SAE network prior to the predictive FFN).

P&L By Feature Selection Size and OGD Learning Rate (Actual Data)

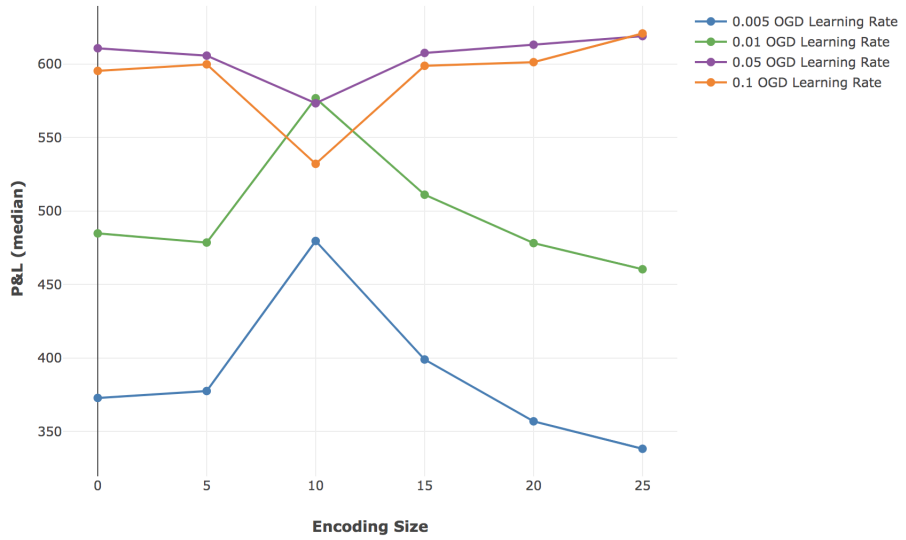


Figure 18: Dataset: Actual10 dataset (7.2.1), Configuration 11 (10.2.11)
The P&L results here show some interesting trends, and highlight the impact of changing dynamics over time. In the case when the OGD (OOS) learning rate is low, and the network is unable to effectively adapt to changing trends and dynamics, the SAE feature reduction is extremely beneficial and results in much higher P&L (as seen in for 0.005 and 0.01 learning rates). Once the OGD learning rate increases and the network is able to adapt, the SAE reduction becomes detrimental (as seen in for the 0.05 learning rate). This is of course amplified if the learning rate is overfitting (as seen for 0.01 learning rate).

An important aspect to note is that the SAE networks were only trained on the first portion of IS data, and not updated afterwards. An effective SAE feature reduction in this process is an optimization that is limited to a certain (earlier) time span. The better the optimisation is for IS data, the less generalisable it is OOS. This optimisation is a better alternative than none at all if the network is limited in it's ability to update itself, but becomes negligible or worse when the network starts updating effectively OOS. While feature reduction is clearly possible, it's also clear that features are not static across time.

Another noteworthy result is the performance of the SAE networks with an encoding layer of 5 nodes. For the, networks with a 0.05 learning rate, the performance is on par with no reduction or reduction to larger features, despite an 83% compression rate for the input data. It also appears to be operating in a more generalisable space compared to the 10 encoding reduction and maintains performance for the larger OGD learning rates.

There is a suggestion here that the optimal encoding size is related to the number of assets, or degrees of freedom (10 in this case), though this idea would need to be explored more. Naturally, future implementations would also need to include an effective, period updating of the SAE as the online learning phase progresses.

Figure 19: Dataset: Synthetic10 (7.1.2) ; Configurations 9 (10.2.9)
The trends in the synthetic data show that the optimal encoding size may be a little larger than the actual data, with P&L in these cases seeming to plateau after 15. This is however still an indication that feature reduction is both possible and effective, even with a 50% reduction in feature size.)

The results seen for synthetic data here offer a nice counterpoint to what is seen in the actual data. With the GBM stock data having constant characteristics throughout the whole dataset, there is no potential for the SAE to overfit to IS data and have a potentially detrimental effect as the learning rate increases (as was the case with actual data).

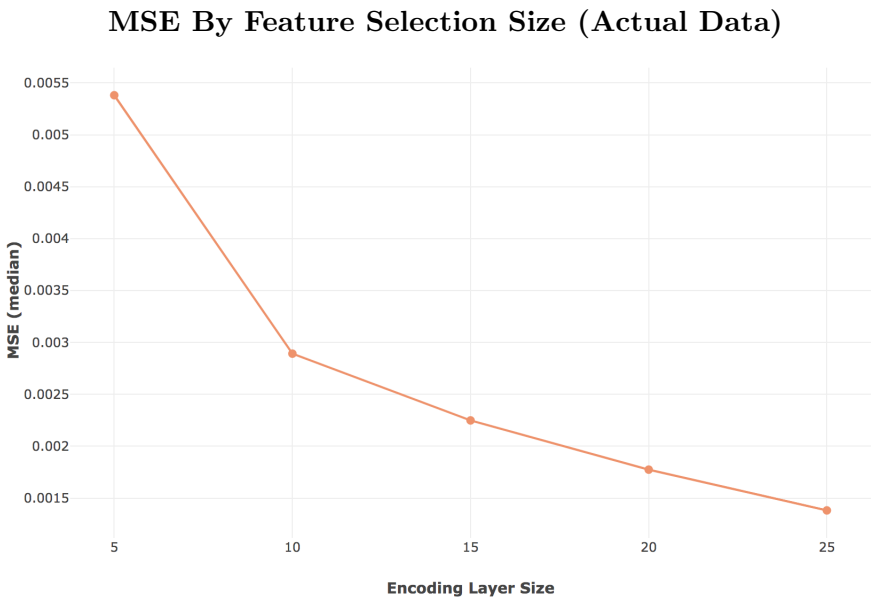
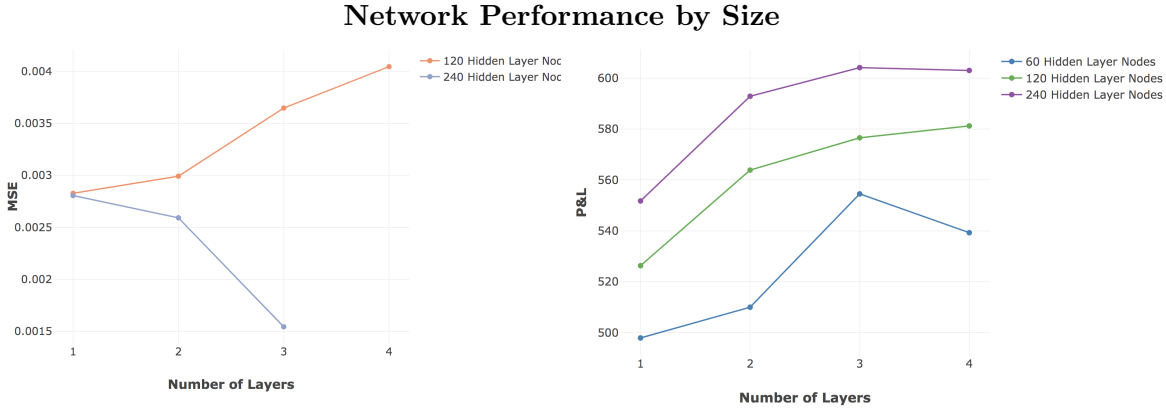


Figure 20: Dataset: Actual10 dataset (7.2.1), Configuration 12 (10.2.12)
The MSE scores here show that while feature selection and reduction is possible through the SAE networks, the ability to reproduce input does decrease monotonically with the encoding layer size. This is not of real concern though unless exact reproduction (with noise included) is of interest. Results for synthetic data were largely the same, which can be seen in Appendix section 10.1.3.

8.5 Network Structure and Training

8.5.1 Effects of Network Size

The effects of networks size on SAE MSE performance and the predictive P&L performance are shown in Figure 21 below. The boxplots showing data for different SAE and predictive networks, for both actual and synthetic data, can be seen in the appendix (section 10.1.4). The trends for both actual and synthetic data were largely the same, and so only actual data has been considered here.



(a) SAE MSE Scores

(b) Predictive P&L Scores

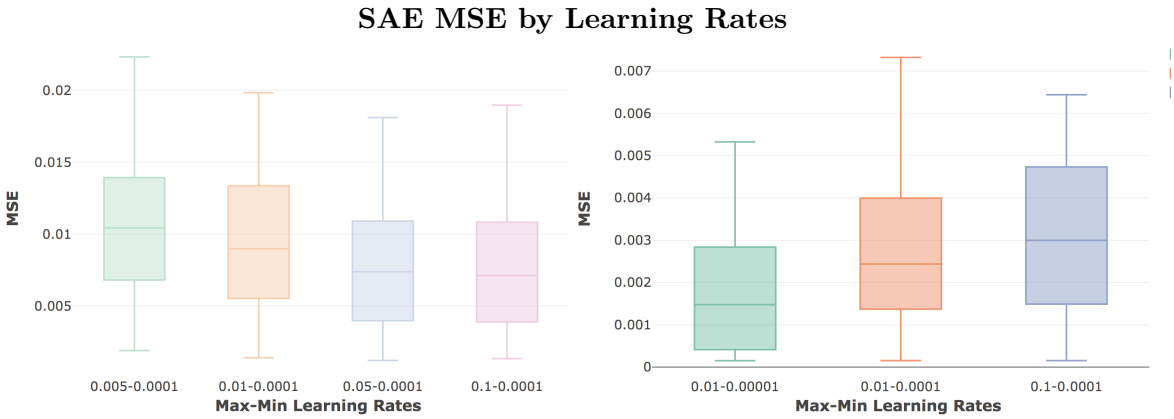
Figure 21: Dataset: Actual10 dataset (7.2.1), Configuration 12 (10.2.12) & Configuration 13 (10.2.13)
Figure (a) shows the median minimum MSE by SAE networks achieved with the indicated number of layers and layer nodes. It's worth noting that the number of layers here indicate the final SAE structure of N hidden layers, rather than the training structure of $2N+1$ hidden layers. Interestingly, the networks with 120 node layers seem to struggled with training, performance decreases as layers increased. This is to be expected if the learning parameters (e.g. SGD learning rate) are not ideal for the structure, the effects of which are amplified as network sizes increase. This is resolved in the larger 240 node networks which show significant improvement as more layers are added (as one would generally expect, subject to effective SGD).
Figure (b) shows the median OOS P&L achieved by predictive networks with the indicated number of layers and layer nodes. The behaviour is as expected here, where network performance generally increases with size. The exception is the four layer network of 60 nodes, where the training parameters were no longer effective in passing signal back (larger layer sizes are less likely to suffer from this, particularly with ReLU activations).

8.5.1.1 SAE Network Structures

It was interesting to note that the typically accepted SAE network structure, where layers decrease in size between the input layer and the encoding layer, resulted in worse performance when compared to the same number of layers with constant numbers of nodes. As detailed in section 8.3, the RBM based greedy layerwise pre-training proved to be ineffective, and so weight initialization with SGD training on the entire network was used instead. It is possible that a greedy layerwise training using these methods (rather than RBM initialization) may have shown increased performance in the decreasing layer size structure, but for SGD training it was clear that larger layers at all stages increased performance. The box plots showing these trends can be seen for actual and synthetic data in the appendix, section 10.1.4.

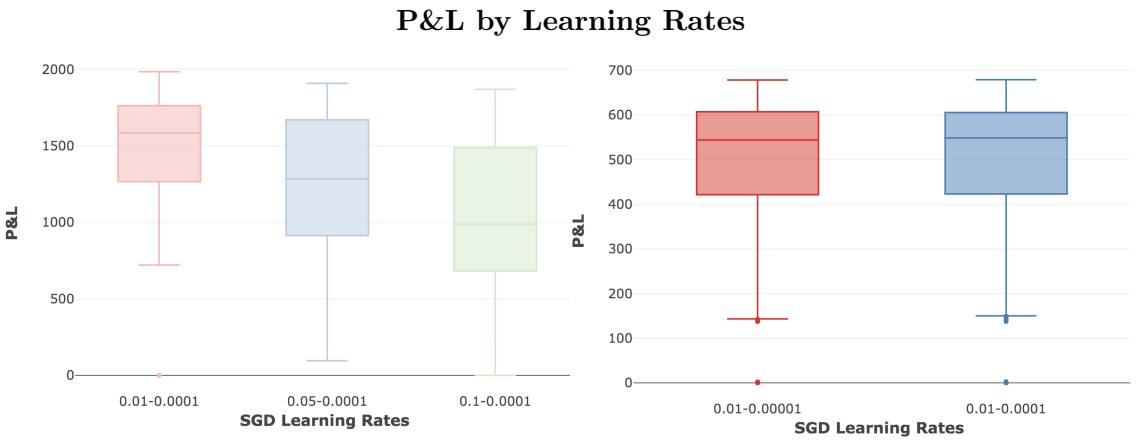
8.5.2 Effects of Learning Rates and Schedules

Learning rate schedules were implemented such that the learning rate would rise and fall through a set of values during a set number of epochs (or epoch cycle), as discussed in 5.2.6. Conceptually, this should allow a more effective traversal of the solution space, where larger learning rates are able to dislodge the configuration from saddle points, and smaller learning rates are able to optimize at a minima.



(a) Synthetic Data **(b) Actual Data**

Figure 22: Dataset: Synthetic10 dataset (7.1.2), Actual10 dataset (7.2.1), Configuration 7 (10.2.7) & Configuration 12 (10.2.12)
It is interesting to note that the effects here are quite different for Synthetic and Actual data, with networks for Synthetic data favouring learning rate ranges with a higher upper bound, and Actual data networks trending in favour of lower upper bound ranges. The synthetic data is structurally less complex, as discussed further in 8.2.1 and 8.6.3, and so the larger learning rates will allow for quicker optimisation. Conversely, SAE networks for actual data benefit from a more careful exploration of the solution space.



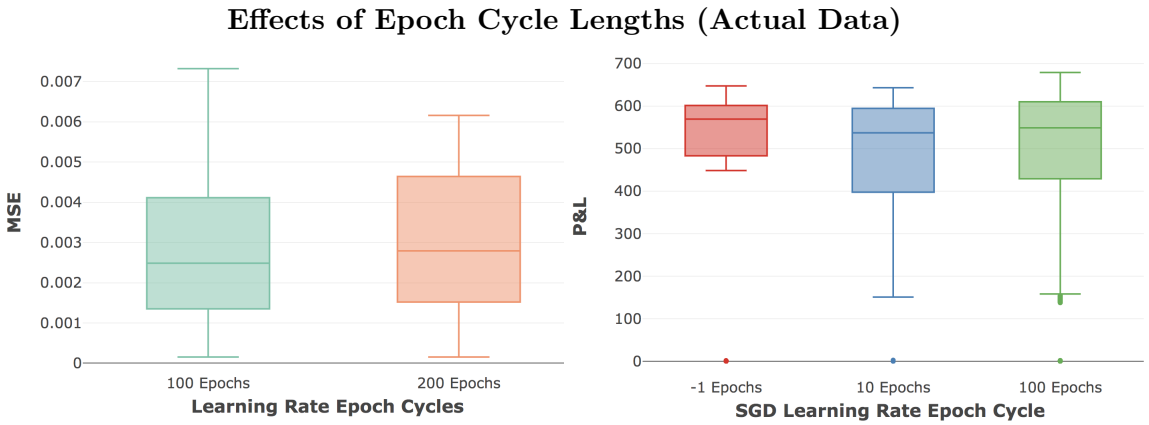
Sample Sizes: [984, 984, 984]

(a) Synthetic Data

Sample Sizes: [10368, 11664]

(b) Actual Data

Figure 23: Dataset: Synthetic10 dataset (7.1.2), Actual10 dataset (7.2.1), Configuration 9 (10.2.9) & Configuration 13 (10.2.13)
The learning rates for Actual data showin in Figure (b) here were chosen accordingly to the best rates seen in the SAE training. There's no substantial difference shown by changing the lower bound by a factor. The results for Synthetic data were interesting in that the networks had higher P&L when trainined with smaller learning rates, suggesting that the predictive task still warrants a more careful solution space exploration despite the less complex nature of the data.



Sample Sizes: [827, 133]

(a) MSE for SAE Networks

Sample Sizes: [648, 5400, 15984]

(b) P&L for Predictive Networks

Figure 24: Dataset: Actual10 dataset (7.2.1), Configuration 12 (10.2.12) & Configuration 13 (10.2.13)
Figure(a) shows the MSE scores for SAE networks, with the shorter epoch cycle offering the best (and the worst) results. Synthetic data for the SAE MSE scores behaved similarly, as seen in the appendix (section 10.1.4).
In Figure (b), for the predictive networks, P&L was highest in the 100 epoch cycle as well, where -1 indicates a constant learning rate rather than the cyclical pattern, and 10 epoch learning rates were used for configurations which only ran for 10 epochs. The configurations without learning epochs (-1) were in the second round of testing and had a more effective set of parameters elsewhere, hence the large difference in the lower bounds relative to the other configurations. Ultimately, the implementation of the learning rate cycle has offered a small but real improvement relative to the constant learning rate for P&L.

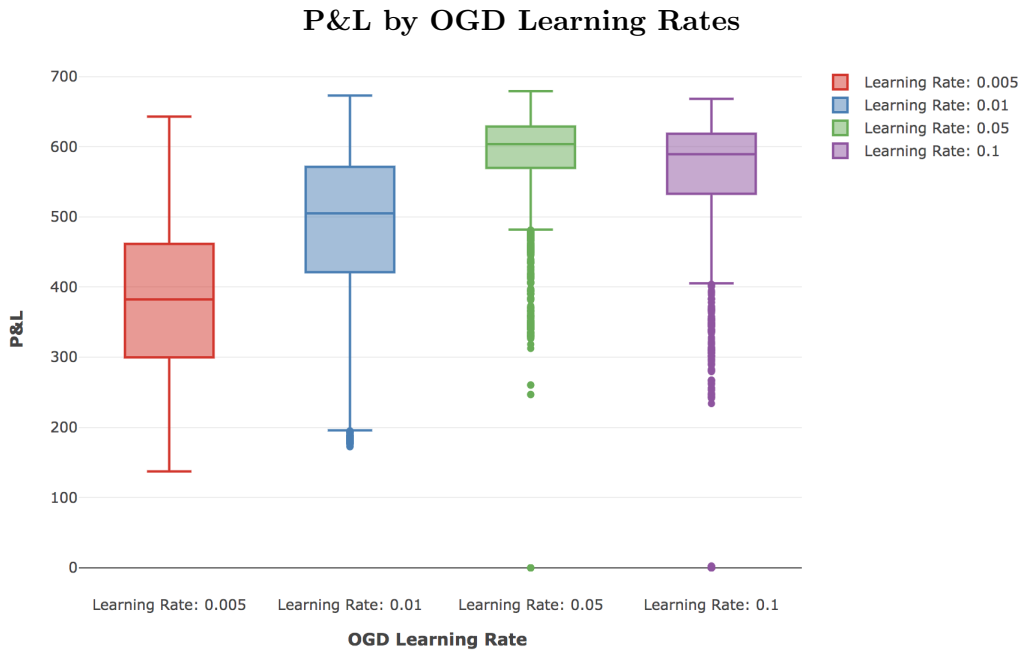
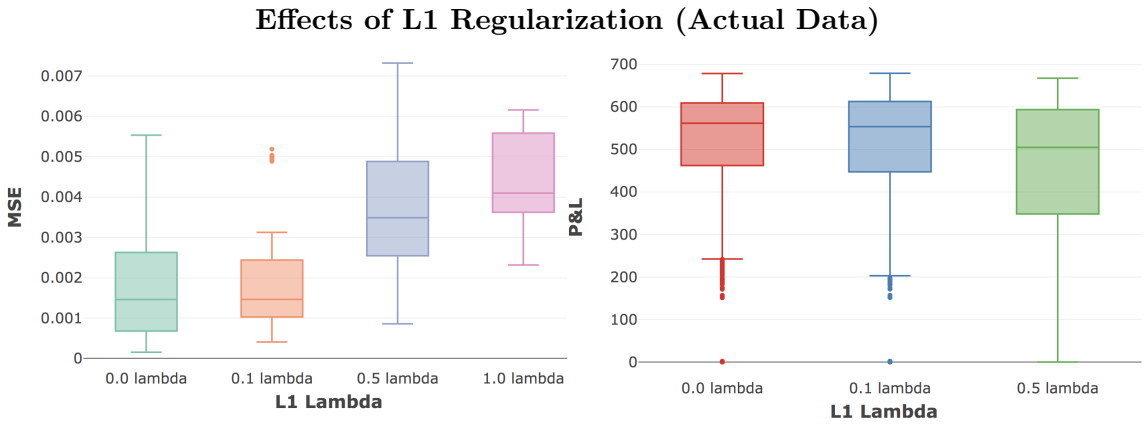


Figure 25: Dataset: Actual10 dataset (7.2.1), Configuration 13 (10.2.13)
The P&L performance for OGD learning rates is expectedly non-linear, with performance steadily increasing up until 0.05 where the network is adapting at the correct rate, and degrading thereafter as the network will overcorrect to noise and short term trends picked up in the online learning period. Networks trained on synthetic data showed similar trends (though were not trained past a turning point), and can be seen in the Appendix in Section 10.1.4.

8.5.3 Effects of Regularization



Sample Sizes: [451, 66, 410, 33]

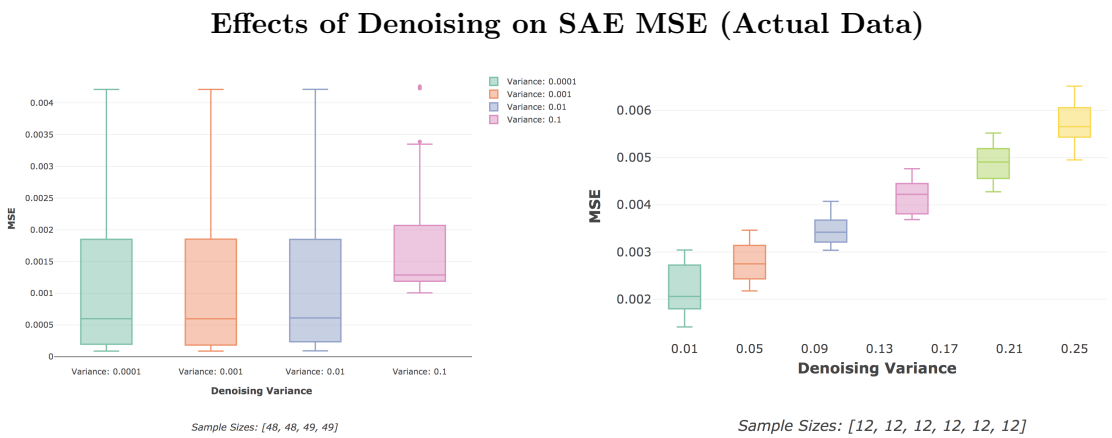
(a) MSE for SAE Networks

Sample Sizes: [8208, 6912, 6912]

(b) P&L for Predictive Networks (SGD)

Figure 26: Dataset: Actual10 dataset (7.2.1), Configurations 12 (10.2.12) & Configuration 13 (10.2.13) Both Figures (a) and (b) show increasingly detrimental performance impacts for SAE as well as predictive FFN networks as the rate of L1 regularization is increased. It is clear, in these cases, that the models were not overfitting in the first place. L1 regularization is expected to work well in instances of IID data, but for time series financial data, where the model isn't overfitting, it is just affecting model fidelity. The effect of regularization on SAE MSE ties in with results seen in sections such as RBM based pre-training (section 8.3). The effects of P&L from regularization during the SGD training phase were not expected to be largely beneficial, in light of the process, and the detrimental effect here suggests that it is only reducing the weight initialization like effect of SGD as noted in section 8.6.

8.5.4 Effects of Denoising



(a) Gaussian Denoising **(b) Masking Denoising**

Figure 27: Dataset: Scaling 10 (7.2.4) , Configuration 14 (10.2.14) & Configuration 15 (10.2.15)
Figure (a) shows the effects of gaussian denoising on SAE MSE scores, with scores becoming worse as the denoising increases. This may once again reflect the differences between IID datasets with a model that is likely to overfit in comparison to financial time series data. Figure (b), with masking denoising, was not expected to have a beneficial effect on SAE performance from a conceptual perspective, which the results are certainly in line with.

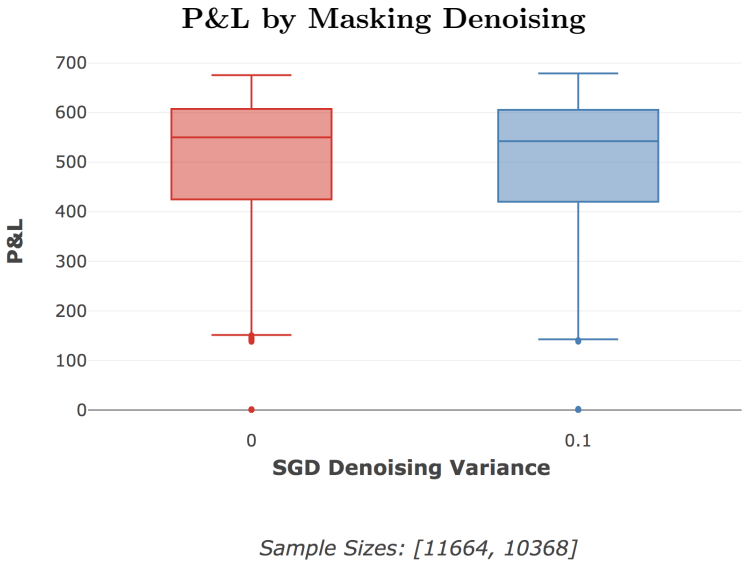


Figure 28: Dataset: Actual10 dataset (7.2.1), Configuration 13 (10.2.13)
Masking Denoising was tested on the predictive FFN networks in the hope that it might encourage the network to increase learning about the relationships between assets. The results for 10% demasking shown here were inconclusive as to whether this would be a beneficial optimization or not, and would warrant further efforts.

8.6 The Effects of Data Aggregation and Value of Historical Data

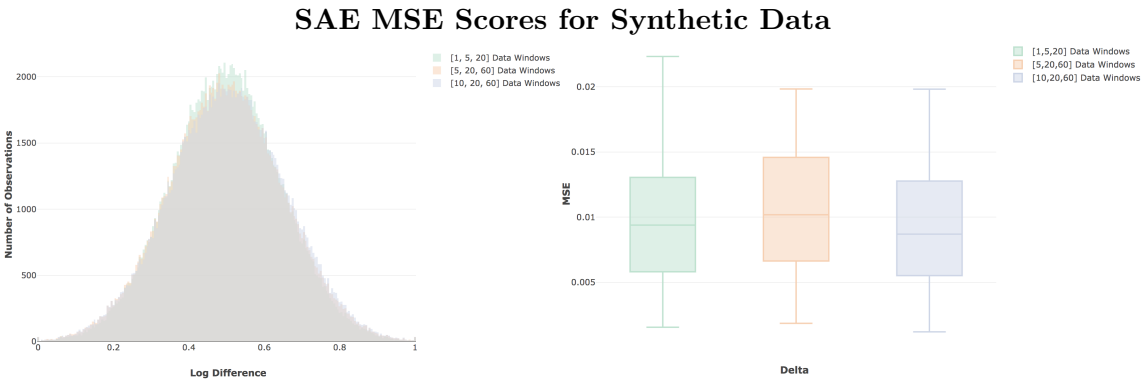
Input data was scaled to 3 different configurations across a series of tests in order to assess the effect of shorter and longer configurations on both SAE MSE performance, as well as the predictive predictive P&L results. The configurations tested (in trading day window periods) were:

1. [1, 5, 20]
2. [5, 20, 60]
3. [10, 20, 60]

By way of example using the 3rd configuration: the implementation of this is such that at each time point considered by a network the log difference changes for the past 10, 20 and 60 days are available for each asset. This is described more extensively in Chapter 4.

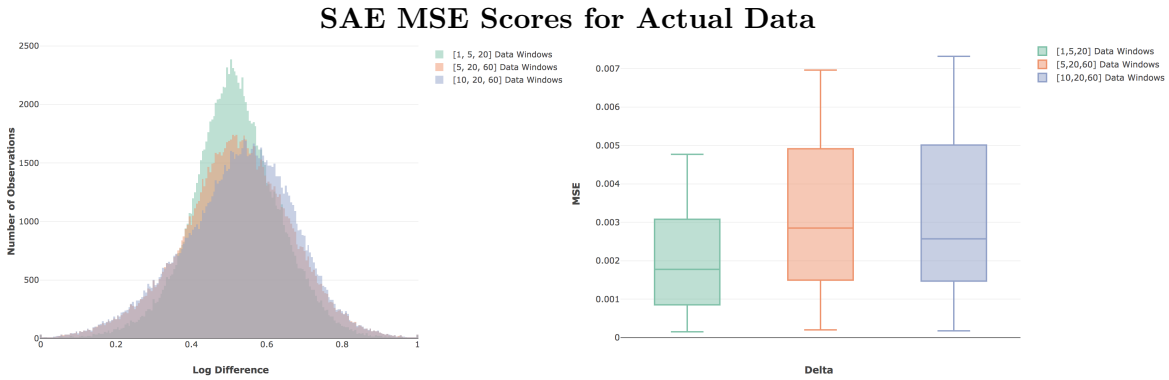
Sections 8.6.1 and 8.6.2 discuss the effects of data aggregation windows on SAE MSE and predictive P&L performance, showing that shorter windows are easier to replicate in SAE networks, and that P&L is improved through access to the most recent fluctuations (i.e. 1 day) and then longer medium term trends. Section 8.6.3 then discusses and validates the idea that extensive historical data is of limited use when having to make predictions for 1 week price changes in more recent data.

8.6.1 Data Aggregation and SAE MSE Scores



(a) Log Difference Distributions **(b) SAE MSE Scores**

Figure 29: Dataset: Synthetic10 dataset (7.1.2), Configuration 7 (10.2.7)
Figure (a) shows the distribution of values to be replicated by the SAE for synthetic data. Geometric Brownian Motion generates discrete changes that follow a log-normal distribution, the log difference and scaling of which (as per the data processing described in 4.1) result in normally distributed price changes. We see small differences in the distributions according to the data aggregation used, as one would expect, with slightly lower variances occuring for smaller data windows ($\sigma_{[1,5,20]} = 0.145$, $\sigma_{[5,20,60]} = 0.152$, $\sigma_{[10,20,60]} = 0.154$). There were some differences in SAE performance as indicated in Figure (b), though not to large extents and considering the very similar distribution of values, there isn't any expectation of seeing fundamental differences in the networks ability to compress and replicate them.



(a) Log Difference Distribution **(b) SAE MSE Scores**

Figure 30: Dataset: Actual10 dataset (7.2.1), Configuration 12 (10.2.12)
Figure (a) shows the distribution of values to be replicated by the SAE for actual data. The distributions are noticeably different from synthetic data, as there is no prior on the price change values, and so variances differ far more across the configurations ($\sigma_{[1,5,20]} = 0.118$, $\sigma_{[5,20,60]} = 0.146$, $\sigma_{[10,20,60]} = 0.150$). Smaller data windows are less likely to capture larger fundamental price changes, resulting in the lower variances. This in turn, makes for easier replication for the SAE networks due to less variety in the sample set, which is seen in Figure (b) with the noticeably better performance in the [1, 5, 20] configuration.

8.6.2 Data Aggregation and Predictive P&L Scores

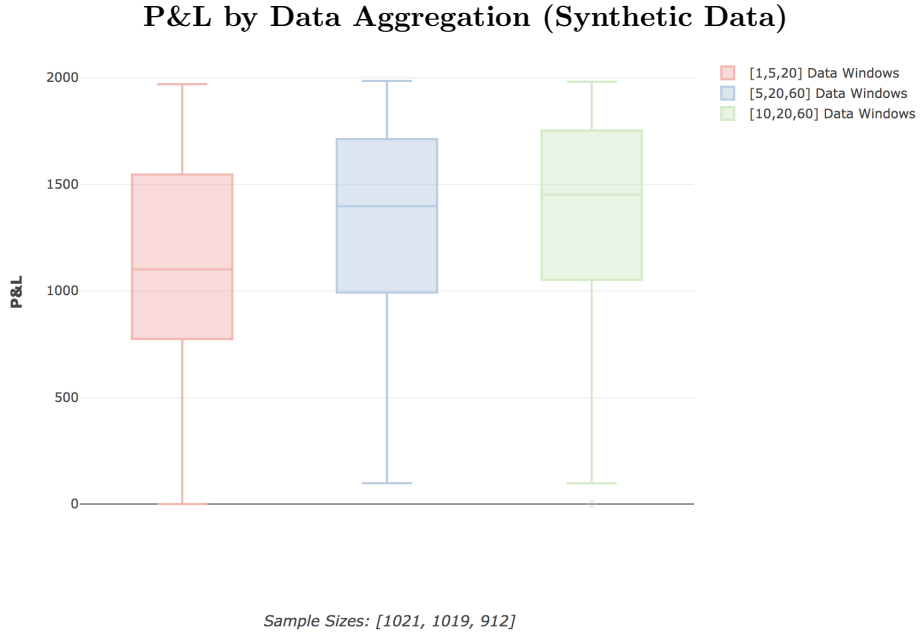


Figure 31: Dataset: Synthetic10 dataset (7.1.2), Configuration 9 (10.2.9)
The boxplots here show the P&L from the MMS according group by the data window configurations. There's a clear trend of P&L increasing as the length of the windows increase. Shorter term GBM data would be more likely to represent fluctuations, whereas the longer term windows will be more representative of the constant mean in the stocks, leading to easier predictive performance and higher P&L.

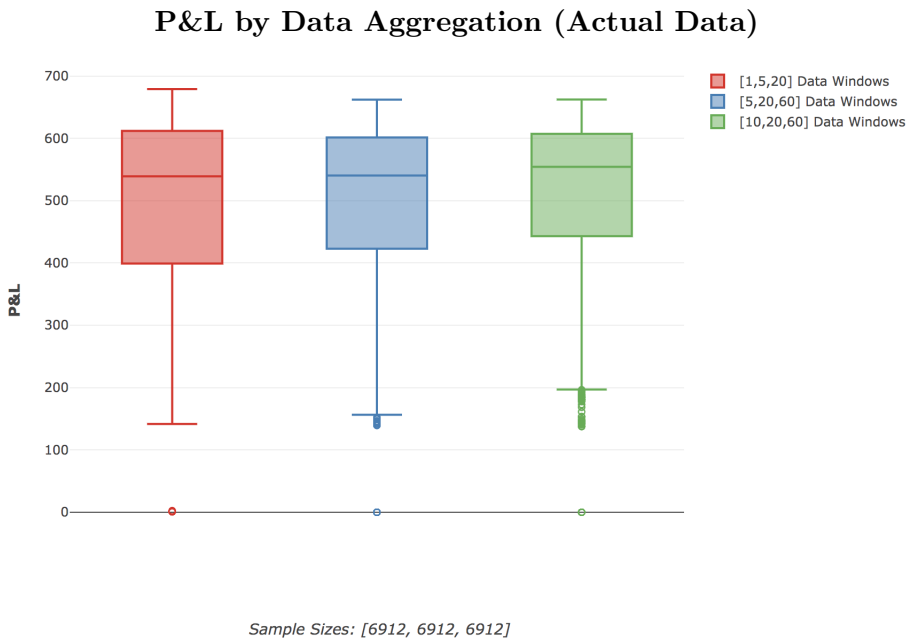


Figure 32: Dataset: Actual10 dataset (7.2.1), Configuration 13 (10.2.13)
The P&L by data window configurations for actual data show a more complex view. At a general level, predictive performance is increasing as the span of the windows increase. However, the highest P&L occurs for the shortest window configuration ([1, 5, 20]). The takeaway here is that both long term trends and short term fluctuations are useful for making short term predictions in price changes, but that a middle of the road configuration offers the worst of both dynamics.

8.6.3 Effects of IS Training and Historical Data

Experimental configuration sets were run to test the hypothesis that the amount of historical IS data available is of limited use, and that the real value is in broader current and cross sectional data. This idea is in line with the understanding that financial markets are inherently complex, adaptive and dynamic, as discussed more broadly in section 2.1. With this in mind, and especially so in light of a fundamentally changing macroeconomic landscape, there is limited reason to believe that the functions and relations that may have governed the asset prices 10-15 years ago would still be in effect today. The P&L results shown in Figures 33 and 34 validate this and show that extensive training on past data may be akin to pre-training network weights at best, and counterproductive in overfitting to dynamics that no longer exist at worst.

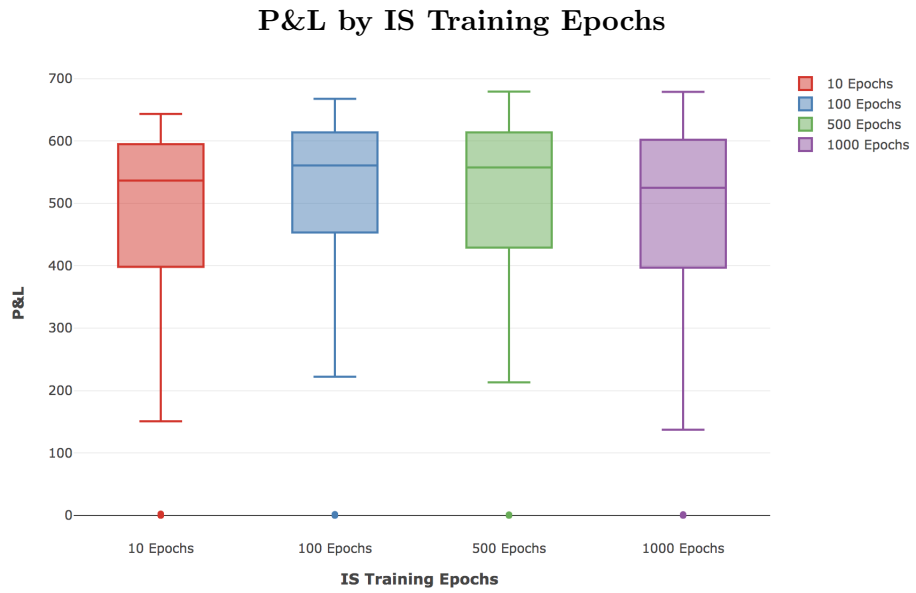


Figure 33: Dataset: Actual10 dataset (7.2.1), Configuration 13 (10.2.13)
The box plots above show P&L grouped by the number of epochs in the SGD IS training phase (i.e. the number of times the IS data was trained on). In this set of configurations, 100 Epochs offers the best overall performance, and further training to 500 or 1000 epochs degrades performance due to the network overfitting on the IS data. The results here are noteworthy as they show that the benefit of historical data is limited - having a network become better at learning return relationships from 10 years ago is not leading to increased P&L for more current data. The small difference between 10 and 100 Epochs further emphasises this point.

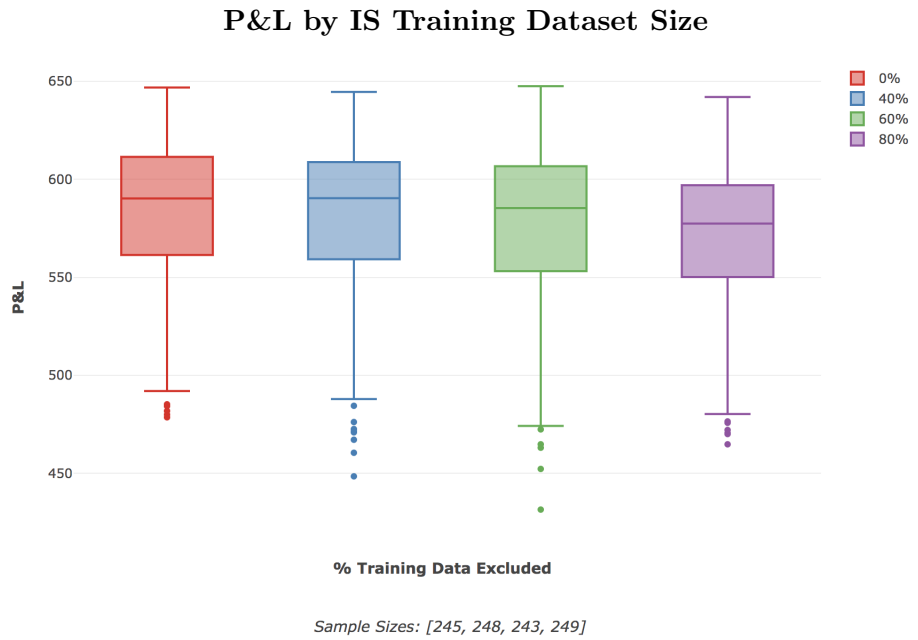


Figure 34: Dataset: Actual10 dataset (7.2.1), Configuration 16 (10.2.16) (some samples with 0 P&L were excluded for more effective visualization)
To further explore the effect of IS training on historical data, configurations were run with a percentage of the usual training data excluded, with the P&L results grouped above. The exclusion of up to 80% of the IS training data resulted in only a 2.2% drop in median P&L for those networks. The training in these instances were not adjusted to increase the number of epochs according to the size of IS data, and so the configurations with more data excluded were also in essence trained less. These results, combined with those in Figure 33 show the limited use in training on historical data, and hence the far greater value in current cross sectional information.

8.7 Probability of Backtest Overfitting

8.7.1 Concerns Regarding the PBO Calculation

The CSCV and PBO techniques, as detailed in 5.6, were developed by Bailey et al. [1] in order to offer a robust methodology of assessing whether a selected strategy is likely to have been justified through backtest overfitting, a common and problematic phenomenon (as discussed in 2.6). The crux of the methodology lies on the idea that for overfitting to occur, the strategies that deliver the best IS performance will systematically underperform in the OOS datasets (thus reflecting the model having overfit to the IS noise).

Their methodology offers several key benefits:

1 The CSCV methodology is generic, model-free and nonparametric, allowing it to arguably used in any model case

2 There is no requirement of a hold-out set, which removes potential credibility issues regarding whether the holdout set was treated appropriately or not

3 While the Sharpe Ratio is the typically chosen performance metric, the technique is generic enough to allow any performance measure to be used

4 The logit distribution developed through the assessment offers a useful view on the robustness of the strategies used and the nature of the PBO score

While the methodology is in substance a model free approach to assessing overfitting, the application in a machine learning context has highlighted some shortfalls and dynamics that are worth considering in its usage. Conceptually, overfitting occurs from model parameters being applied to the IS set which perform poorly OOS. For an online learning neural network though, the model parameters which are applied IS for prior training are not necessarily the same parameters that are used for the OOS results (i.e. one may not use the same learning rates for IS SGD training as for OOS OGD training). Further, the parameters of a neural network model are for the most part the weights of the network, which change and adapt during the online learning (OOS) phase. This is the primary strength of the model - while it may very well be overfit to the IS data, an effective learning implementation will result in the model adapting relatively quickly and so move out of the overfit solution space (evidence of IS overfitting can be seen in 8.6). Notably, this behaviour would not occur in many static financial econometric models that might typically be more closely associated with backtest overfitting issues.

The use of the logit metric in the CSCV method also creates some potentially problematic issues due to its basis in an ordinal ranking, and whether the best strategy in the IS set is higher than the median in the OOS set. The effect is that strategies which perform poorly both in the IS and OOS datasets are able to artificially bolster the ordinal position of the best strategy such that it is past the median point and does not result in an increased likelihood of overfitting. As the authors correctly noted, the addition of trials that are doomed to fail will bias results, and if configurations are obviously flawed then they shouldn't have been tried in the first place. In practice though, the issue is more pervasive than suggested. The combinatorial grid search approach as outlined in 6.1 results in certain parameter combinations that cause networks to perform very poorly. Individually, these parameters may work well in other combinations though, and so there would be no obvious reason to exclude them to begin with. In this sense, an honest search of the solution space where there is no prior

knowledge about what will work OOS may result in a PBO score that is lower than it should be. This dynamic is concerning in that it is easy to create poor performance models and so allows the PBO method to be used fraudulently, and is even more so in the case where it might occur without the researcher realising.

Further to this, the parameter space search methodology (section 6.1) also results in a lower likelihood of PBO due to the way of combining parameters across IS and OOS stages. By way of example, any configuration which performs well IS will have all possible OOS parameters tested in combination with it. While some of these combinations may result in poor performance, there will always be a combination of the best IS and best OOS parameter choices. This makes it unlikely that the best configurations will be past the median point for the logit calculation, resulting in a systematically low PBO.

Another potential implementation concern with CSCV and PBO is the choice of the S parameter, which is the number of splits for the data to be sampled into. As seen in Figure 35, the number of splits can have an overwhelming effect on the PBO score reported for the same data sets and results. This is not a problem if the implementation is done honestly and openly, but does allow for misuse and a potential inability to compare PBO scores. Further, while the authors do provide guidelines for choosing S such that the splits will be representative of trading periods, there is still no clearly defined method of choosing the time spans of these periods. The exponential growth in combinations to be tested as S increases also quickly results in reaching computational resource limits, which may further lead to the choice in S being made according to reasons that are not in line with prioritising the correct PBO score to report. These combination sizes can be seen below in Figure 36.

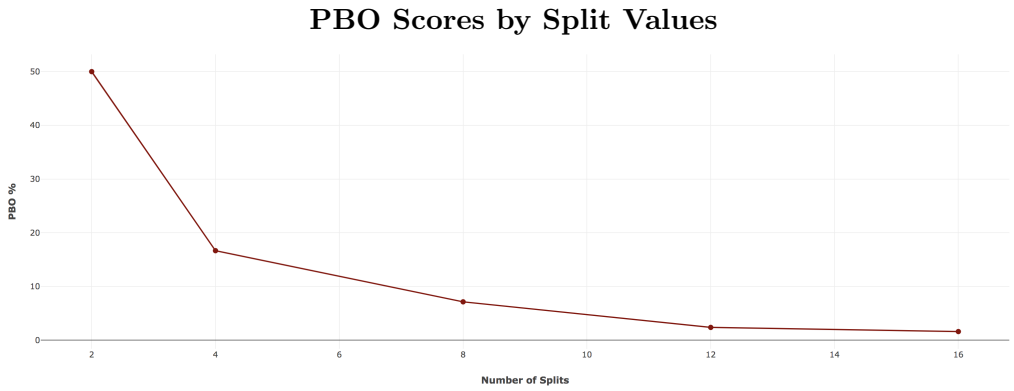


Figure 35:
The curve here shows the significant impact that the choice in the split parameters has for the PBO score calculated, with a monotonically decreasing score as S increases (these scores were for the same set of results).

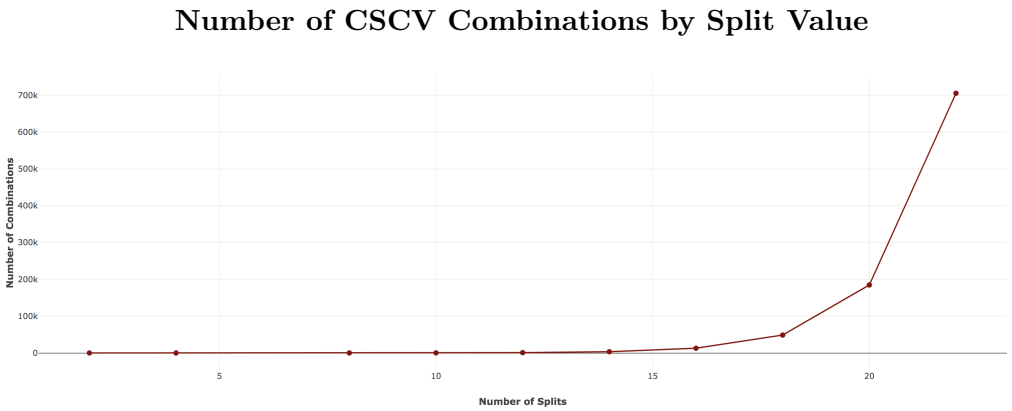


Figure 36:
The number of combinations that have to be procssed by the CSCV method according to the value of the split parameter (S) chosen

8.7.2 PBO Results

The full set of tests run (n = 22248) on the Actual10 (7.2.1) dataset had a final PBO of 1.6%, having been run with a split of 16. There were 15 years of data, making it a reasonable choice as the split parameter (which needs to be even). Ideally, the splits would represent shorter periods, but the exponential increase in computational time (as seen in Figure 36) made this impractical, and 16 also happens to be the typical choice recommended by the authors. The full logit distribution can be seen below in Figure 37.

correct vals if needed

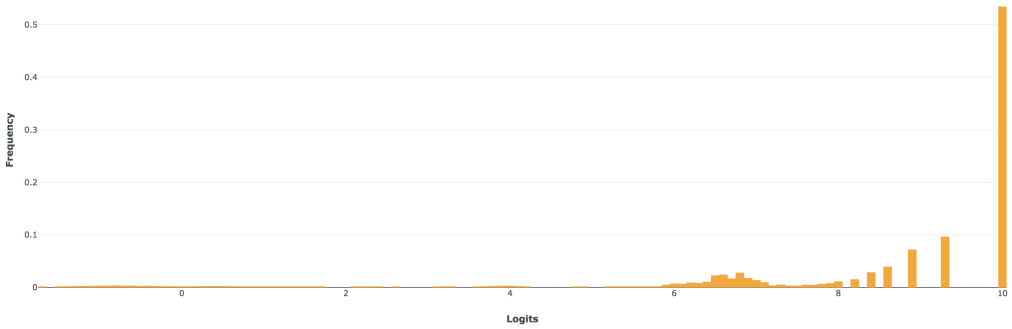


Figure 37: Dataset: Actual10 (7.2.1) ; Configurations)
The CSCV logit distribution for all configurations run, with a calculated PBO of 1.6%.

It is of interest to note the dynamics and contribution towards the PBO results. The configuration process went through 2 primary phases: an extremely broad combinatorial grid search, consisting of 20880 configurations; and a second much narrower search of 1512 configurations. Assessing only the configurations from the second phase, results in a PBO score of 6.3%, which is significantly higher than the overall PBO score, the logit distribution for which can be seen in Figure 38 below. The effect here highlights two important aspects of the PBO calculation:

1 The scores are much higher for the configurations which were picked more specifically after having already seen a large number of results, which is correctly indicative of increased likelihood to overfit.

2 However, the PBO score is not monotonically increasing with N, as one would expect. This is counterintuitive and is in line with the concerns raised in 8.7.1 regarding the effects of increasing configuration sample size.

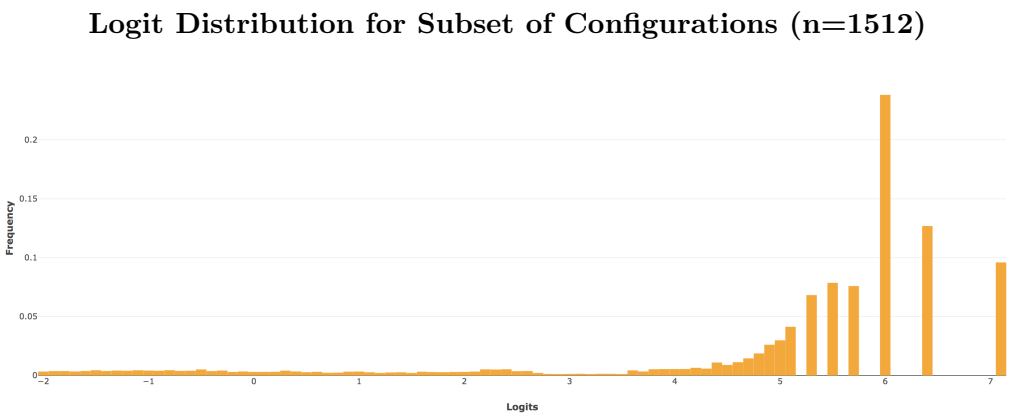


Figure 38: Dataset: Actual10 (7.2.1) ; Configurations)
The CSCV logit distribution for a narrower of configurations run, with a calculated PBO of 6.3%.

8.7.3 Framework Success

While the CSCV and PBO implementations do raise some concerns, the results here do also highlight the efficacy of the framework which has been implemented. The low PBO scores presented here show that the combinatorial search approach combined with an online machine learning model allows researchers to take on a broad exploration of the possible solution space while maintaining a low risk of backtest overfitting, and such that P&L results which are comparable to the benchmark are achieved, as seen below in section 8.9.

8.8 DSR Results

.

Note: The work here had not been presented in a final format, but rather so that it facilitates review as the 1.0 DSR value was unexpected.

The DSR process was run with:

- Maximum Clusters = 10
- Maximum Iterations = 10

The attached code in the paper was used to calculate the clusters, using an input of the correlation matrix for the return rates percentages of each strategy at each timpoint. The output was 2 clusters, and a DSR of 1.0. The intermediate values of the calculations, as well as the cluster configurations and comparisons have been included below.

Calculating the aSR values for each cluster:

$$\widehat{aSR}_k = \widehat{SR}_k \sqrt{Frequency_k} \tag{73}$$

$$\widehat{aSR}_1 = 0.70486\sqrt{365.25} = 13.47103 \tag{74}$$

$$\widehat{aSR}_2 = 0.70301\sqrt{365.25} = 13.43565 \tag{75}$$

Calculating the variance of the clusters:

$$\mathrm{E}\left[\mathrm{V}\left[\left\{\widehat{SR}_k\right\}\right]\right] = \frac{\mathrm{V}\left[\left\{\widehat{aSR}_k\right\}\right]}{\mathrm{Frequency}_{k^*}} \tag{76}$$

$$= \frac{\mathrm{Var}\left[\{13.47103, 13.43565\}\right]}{365.25} \tag{77}$$

$$= 8.569980392994096e-07 \tag{78}$$

$$SR^* = \sqrt{\mathrm{V}\left[\left\{\widehat{SR}_k\right\}\right]}\left((1-\gamma)Z^{-1}\left[1-\frac{1}{K}\right] + \gamma Z^{-1}\left[1-\frac{1}{Ke}\right]\right) \tag{79}$$

$$= \sqrt{8.56998e-07}\left((1-\gamma)Z^{-1}\left[1-\frac{1}{2}\right] + \gamma Z^{-1}\left[1-\frac{1}{2e}\right]\right) \tag{80}$$

$$= 0.00048115929261803515 \tag{81}$$

Best SR observed in full configuration set:

$$\widehat{SR} = 0.7286175492903578 \tag{82}$$

Calculating DSR:

$$\widehat{PSR}\left[SR^*\right] = Z\left[\frac{\left(\widehat{SR} - SR^*\right)\sqrt{T-1}}{\sqrt{1-\hat{\gamma}_3\widehat{SR} + \frac{\hat{\gamma}_4 - 1}{4}\widehat{SR}^2}}\right] \tag{83}$$

$$= Z \left[\frac{(0.7286 - 0.00048)\sqrt{1500 - 1}}{\sqrt{1 - 2.2772 * 0.7286 + \frac{12.3861-1}{4} 0.7286^2}} \right] \tag{84}$$

$$= Z \left[\frac{28.6566}{0.9353} \right] \tag{85}$$

$$= 1.0 \tag{86}$$

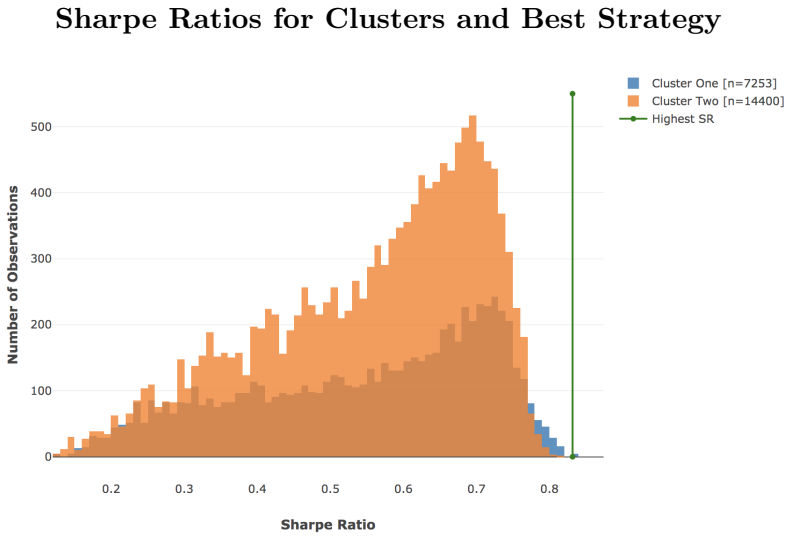


Figure 39:
The distribution of all sharpe ratios, grouped by cluster and a separate line for the best sharpe ratio.

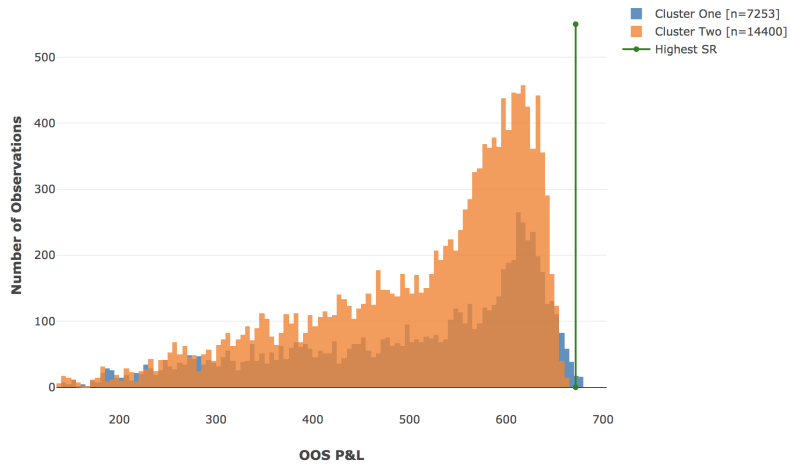


Figure 40:
The distribution of all oos P&L amounts, grouped by cluster and a separate line for the best strategy's P&L (chosen according to highest SR).

8.9 MMS Results

8.9.1 Summary of Experiment Results

Figures 41 and 42 below show the distributions of OOS P&L with and without trading costs being accounted for. As displayed there were a significant number of distributions which were within the 20%-30% range of the benchmark, which represents the P&L the strategy would have accumulated with perfect knowledge of future stock movements. While the costs do have a notable impact on the overall P&L distribution, it does not seem to indicate that the strategy would suffer from implementation limitations due to trading costs. The configurations represented by the '0' bar are networks which suffered from either exploding or vanishing gradients, and were not able to make sufficient predictions. The distribution of Sharpe Ratios are shown in Figure 43, and are expectedly following a similar pattern to the P&L distributions. The Sharpe Ratio's are not providing exceedingly good returns, though this is as representative of the basic trading strategy and selection of stocks used as any performance assessment of the underlying networks, as further emphasised by the distribution of correct trade percentages in Figure 44.

As noted earlier, the P&L reported on here is a relative unit measurement, rather than a particular currency (as discussed in 4.3). Additionally, the MMS, as described in 5.7, only trades 1 stock unit at a time and so any P&L figures are relative indications rather than absolute limits.

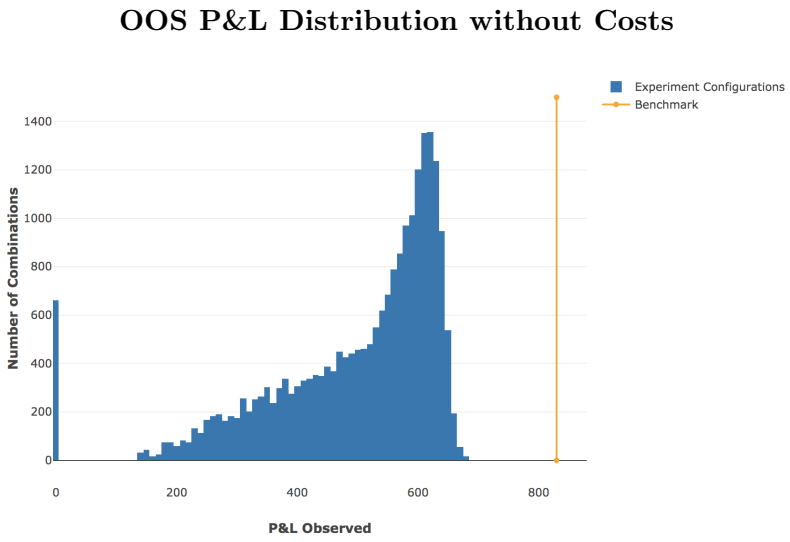


Figure 41: Dataset: Actual10 (7.2.1) ; Configurations)
The distribution of all OOS P&L values, with the benchmark P&L indicated in orange.

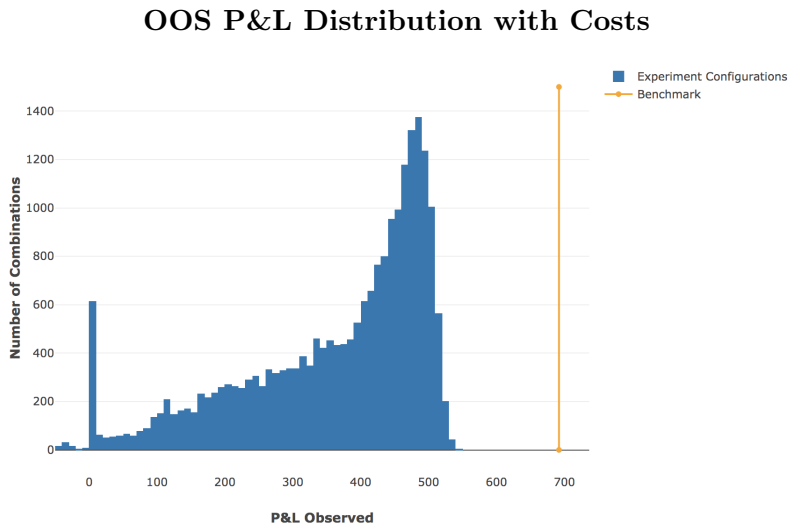


Figure 42: Dataset: Actual10 (7.2.1) ; Configurations)
The distribution of all OOS P&L values when trading costs are taken into account, with the benchmark P&L indicated in orange.

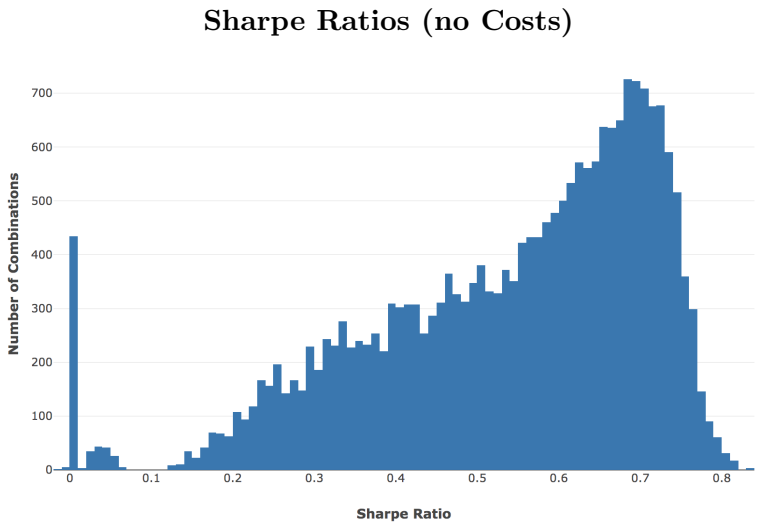


Figure 43: Dataset: Actual10 (7.2.1) ; Configurations)
The distribution of all strategies Sharpe Ratios, without costs taken into account.

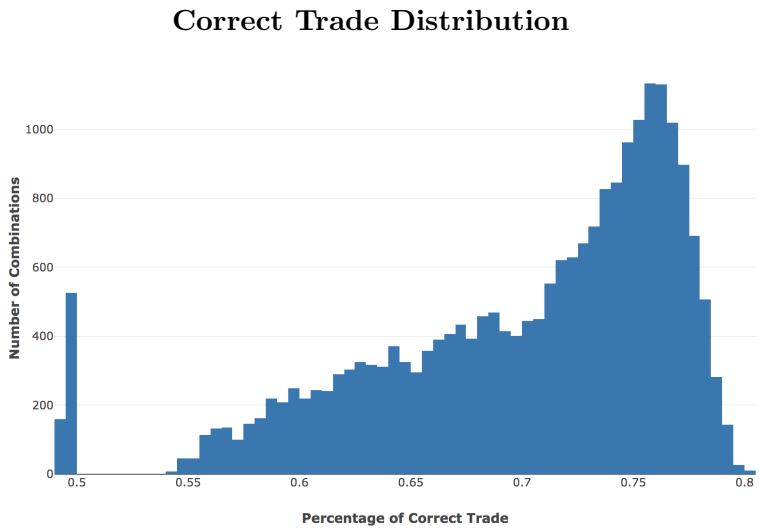


Figure 44: Dataset: Actual10 (7.2.1) ; Configurations)
The distribution here shows the number of configurations that resultd in the indicated percentage of correct trades.

8.9.2 Best Network Results

The best network configurations P&L performance relative to the benchmark is detailed beliw in Table 4, with performance within 22% or 28% of the benchmark (excluding and including costs, respectivel). Considering the benchmark represents perfect knowledge of future prices, this is a very promising delivery by the framework developed.

Measure	Best Network	Benchmark	Difference
Observed P&L	679.03	829.72	22.19%
Observed P&L with Costs	541.15	692.17	27.91%
Observed Return Rate	2.49%	3.04%	22.48%
Observed Return Rate with Costs	1.97%	2.53%	28.21%

Table 4: Best Network versus Benchmark

8.10 Results Summary

The results for training have been summarised below, especially insofar as they relate to differences observed between actual and synthetic data.

8.10.1 Framework Success and PBO

1 Section 8.7 and 8.9 show that the framework was able to generate models that deliver returns within a successful range of the benchmark, even with costs considered, and with a low likelihood of having succumbed to backtest overfitting as indicated by a rigorous assessment.

8.10.2 Historical Data

1 Section 8.6 shows that training on historical data in the SGD phase had only minor effects, both in terms of the amount of data and the duration of training on it. This validates the argument that historical financial data is of limited benefit in training predictive models, and emphasises the value of cross sectional current data. Section 8.4 builds on this by demonstrating the changing dynamics in feature reduction over time, and showing that even features will not be static over long periods.

8.10.3 Data Processing Considerations

1 Section 8.6 showed that while longer data window aggregations resulted in higher performance for synthetic data, actual data had better performance for shorter windows, with the results also suggesting that longer aggregation windows may improve P&L returns.

2 Section 8.2 showed that Normalizing data resulted in significantly better MSE scores on the SAE networks for actual data.

3 Section 8.2 also showed on Synthetic data that the Limited Normalization (10) did impact P&L, but within reasonable margins.

8.10.4 Weight Initializations

1 Section 8.3 showed that RBM based greey layerwise pre-training for SAE training was ineffective on actual financial time series data, and that variance based weight initializations generate better results.

2 Further, it was shown that the He-Adj weight initialization presented was the best all rounder for both Synthetic and Actual data, on both SAE and predictive networks.

8.10.5 Feature Selection

1 Section 8.4 showed that while feature selection was possible on actual financial time series data, the optimizations appear to be localized to certain time periods, and would need to be updated periodically in order to provide a benefit.

2 These effects were less prominent in synthetic data, where encoding efficacy simply plateaued after hitting a certain threshold.

8.10.6 Output Activations and Network Structure

1 Section 8.2 showed that linear encodings were more effective than ReLU or Sigmoid in the encoding and output layers, both for SAE and for predictive networks and on both synthetic and actual data.

2 In the hidden layers for both SAE and predictive networks, Linear activations performed better for synthetic data, but performed worse for actual datasets than ReLU and Sigmoid activations (particiularly in the scenario of smalled encoding layers).

3 Leaky ReLU was also shown to provide a marginal improvement for synthetic data in SAE and predictive networks when compare to ReLU for hidden layers.

4 Section 8.5 showed that for the most part, more layers of larger sizes results in better performance but that this is also subject to data and learning rates and should be experimented with.

8.10.7 Learning Optimizations

1 Section 8.5 showed L1 Regularization resulted in worse performance on actual data for both SAE and predictive networks.

2 Gaussian and Masking denoising was also shown to result in worse performance on SAE networks for actual data, while the effect of Masking denoising on actual predictive networks was inconclusive.

3 Learning rate cycles were shown offer some small improvements, though the ranges and epoch cycles need to be tailored appropriately. There did not appear to be strong relations for the best parameters when comparing actual and synthetic data.

3 OGD learning rate performances expectedly follow a curve, with underfitting and overfitting sitting either side of the optimal rate, which will change with the dataset.

9 Conclusion

and DSR?

A configurable system and framework, based on decoupled modules, has been presented which incorporates several notable techniques: deep learning neural networks for stock price fluctuation prediction, stacked auto encoders for the purpose of feature selection, and CSCV and PBO in order to assess the returns from the MMS and the likelihood that backtest overfitting has taken place.

The system presented is successful in achieving the aims set out for it and presents a cohesive view of how components interact and might be changed in a future implementation. Feedforward neural network training parameters are explored and presented, including network sizes, activation functions and learning optimizations. Stacked auto encoders, initialized using both RBM pre-training as well as variance based techniques, are presented and shown to be able to perform effective feature selection. An extensive comparison of synthetic and actual data, including data pre-processing, is covered such that it might guide further usage of the system. Lastly, the CSCV and PBO methodologies are implemented, considered, and shown that they are able to work for a novel implementation on machine learning models and provide rigorous assessment of results.

Further to this, a view on the phenomenology of financial markets has been developed and presented, based on the experiments results. The results presented for limited in-sample training of the networks (both by length of training and dataset size), the effects of feature selection developed in-sample, as well as out of sample training parameters all provide a clear notion that there is a positive but very limited benefit to training on long term historical financial time series. The results show that the value of a cross sectional view of the data has far more weight in delivering out of sample returns, which is particularly noteworthy in the context of a neural network with few structural model limitations.

Ultimately, the system has been presented and shown to not only deliver promising P&L figures for out of sample data, but has also done so while being able to incur a low risk of backtest overfitting.

9.1 Future Work

Several areas of the system have been highlighted as worthy of further work based on the results seen so far:

1 The updating of SAE models is the most likely area of further development, possibly based on rolling time windows in order to fully take advantage of recent data benefits, or using a bootstrapped dataset in order to add time based weights to observations.

2 While RBM pre-training was shown to be ineffective, it is possible that a greedy layerwise approach using variance based weight initialization could still be used in order to train more efficient SAE networks.

3 Data processing methods such as the Error Function coudl be implemented in order to better deali with outliers, which are often prominent in financial data.

4 Further exploration of using Masking denoising for price flucatuation prediction is warranted, in order to try and enforce correlation and relationship recognitition within the FNN.

5 A more extensive exploration of historical vs. current data would be particularly beneficial, especially with its potential application outside of just the presented system.

10 Appendix

10.1 Additional Results

10.1.1 Additional Results for Section 8.2.2 - Activation Functions and Scaling

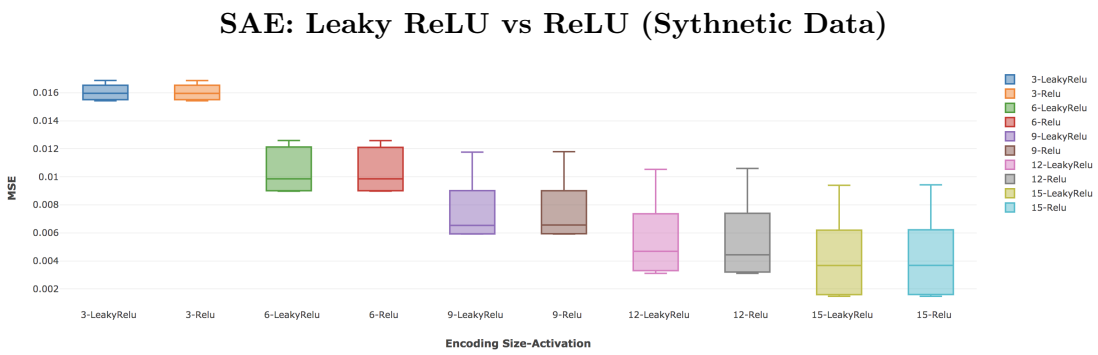


Figure 45: Dataset Synthetic6 (7.1.1) ; Configuration 17 (10.2.17)
The plot above shows the SAE MSE, grouped by encoding size and activations, showing a marginal increase in performance for Leaky ReLU activations.

10.1.2 Additional Results for Section 8.3 - Weight Initialization Techniques

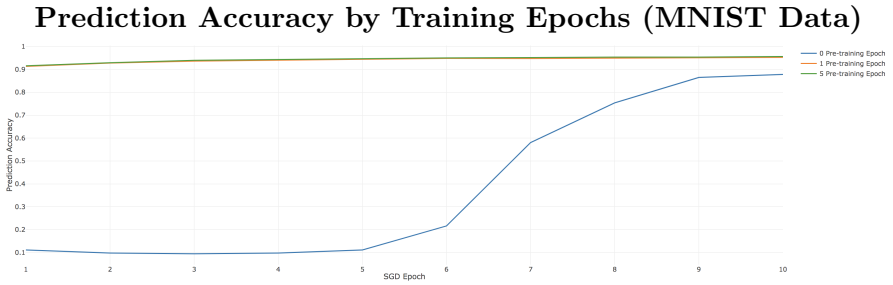


Figure 46: The plot above shows the prediction accuracy achieved on the MNIST dataset according to the number of pre-training epochs using the RBM greedy layerwise methodology as per section 5.3. There's a clear indication that pre-training allows the network to achieve much higher accuracy much quicker.

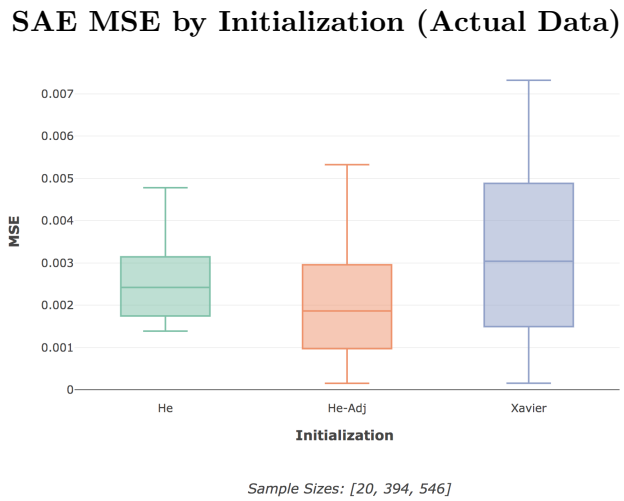


Figure 47: Dataset: Actual10 (7.2.1) ; Configuration 12 (10.2.12)
The SAE MSE results here continue the trend seen for AGL, where He-Adj offers the best performance. Both He and He-Adj suffered with larger learning rates, resulting in dying ReLU's and networks with null outputs, hence the smaller sample sizes. It's possible a more suitable set of learning rates would lead to better results, though the results presented in 8.3 lead to a favouring of He-Adj in any case.

10.1.3 Additional Results for Section 8.4 - Feature Selection

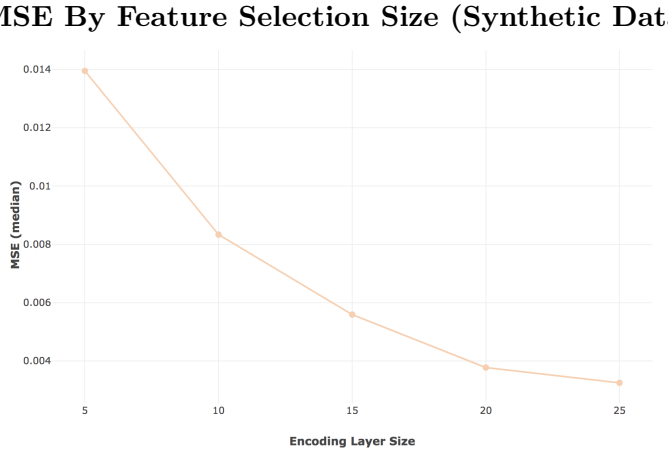


Figure 48: Dataset: Synthetic10 (7.1.2) ; Configuration 7 (10.2.7)
The MSE scores here show that while feature selection and reduction is possible through the SAE networks, the ability to reproduce input does decrease monotonically with the encoding layer size. This is not of real concern though unless exact reproduction (with noise included) is of interest.

10.1.4 Additional Results for Section 8.5 - Network Structure and Training

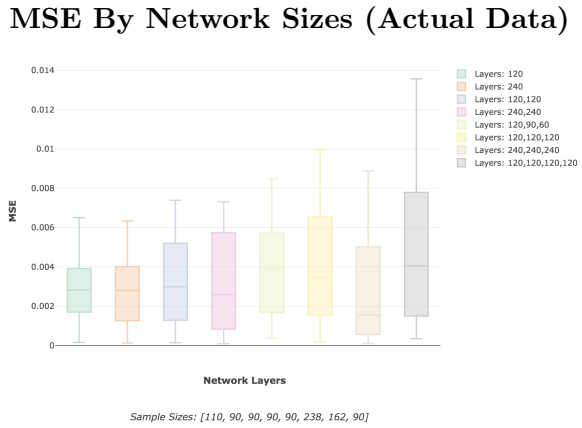


Figure 49: Dataset: Actual10 (7.2.1) ; Configuration 12 (10.2.12)
This figure shows the MSE achieved by SAE networks with the indicated sizes, showing a general increase in performance as both layers and layer sizes increase. The network sizes here indicate the final N layers for the SAE, rather than the $2N + 1$ that are present in training. Training parameters such as number of SGD epochs were not adjusted for network size, and so some smaller networks achieved higher performance as they had more accomodating training. The general trends should be focused on as the indicator for more tailored training. As noted in 8.5, the more typical structure of SAEs with descending layer sizes did not show better performance.

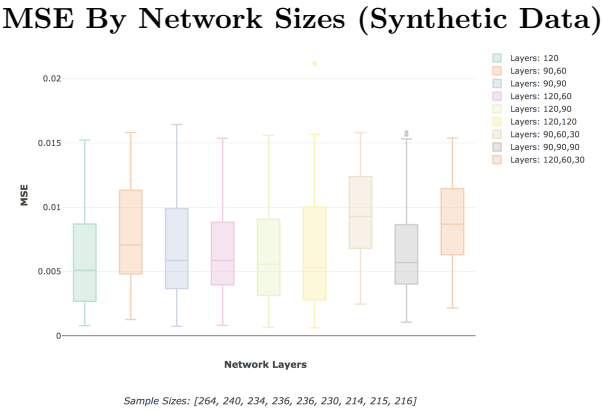


Figure 50: Dataset: Synthetic10 (7.1.2) ; Configuration 7 (10.2.7)
This figure shows the MSE achieved by networks with the indicated sizes, showing a general increase in performance as both layers and layer sizes increase. The network sizes here indicate the final N layers for the SAE, rather than the $2N+1$ that are present in training. Training parameters such as number of SGD epochs were not adjusted for network size, and so some smaller networks achieved higher performance as they had more accomodating training. The general trends should be focused on as the indicator for more tailored training. As noted in 8.5, the more typical structure of SAEs with descending layer sizes did not show better performance.

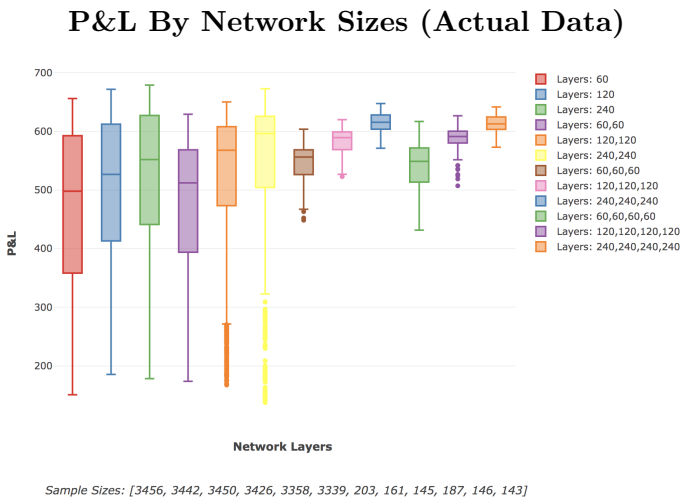


Figure 51: Dataset: Actual10 (7.2.1) ; Configuration 13 (10.2.13)
This figure shows the P&L achieved by networks with the indicated sizes, showing a general increase in performance as both layers and layer sizes increase. Networks that suffered from exploding ReLU's (and hence have approximately 0 P&L) have been excluded for the more effective visualization. Training parameters such as number of SGD epochs were not adjusted for network size, and so some smaller networks achieved higher performance as they had more accomodating training. The general trends should be focused on as the indicator for more tailored training.

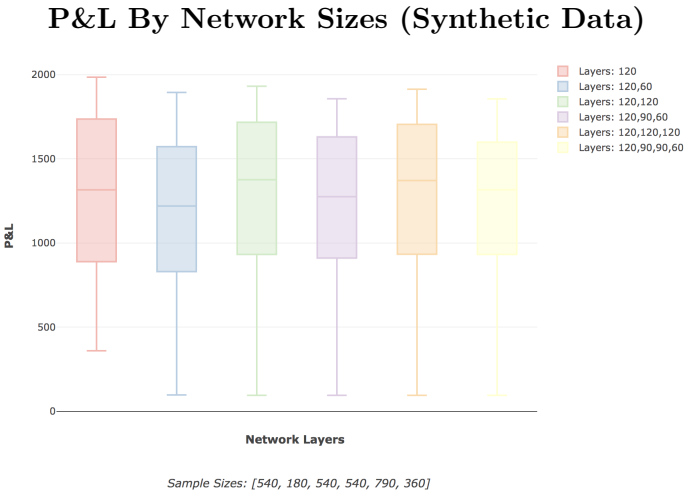


Figure 52: Dataset: Synthetic10 (7.1.2) ; Configuration 9 (10.2.9)
This figure shows the P&L achieved by networks with the indicated sizes, showing a general increase in performance as both layers and layer sizes increase. Networks that suffered from exploding ReLU's (and hence have approximately 0 P&L) have been excluded for the more effective visualization. Training parameters such as number of SGD epochs were not adjusted for network size, and so some smaller networks achieved higher performance as they had more accomodating training. The general trends should be focused on as the indicator for more tailored training.

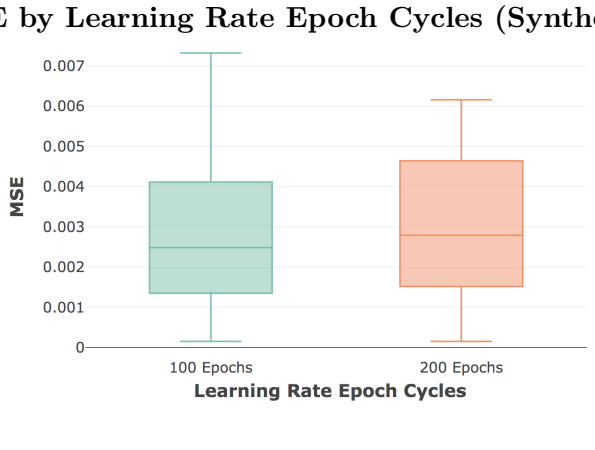


Figure 53: Dataset: Synthetic10 (7.1.2) ; Configuration 7 (10.2.7)
The SAE networks didn't show significant differences according to the epoch cycles. As discussed in sections 8.2.1 and 8.6.3, the structure of synthetic data for SAE replication is structurally less complex, and so learning optimizations are less likely to produce large differences.

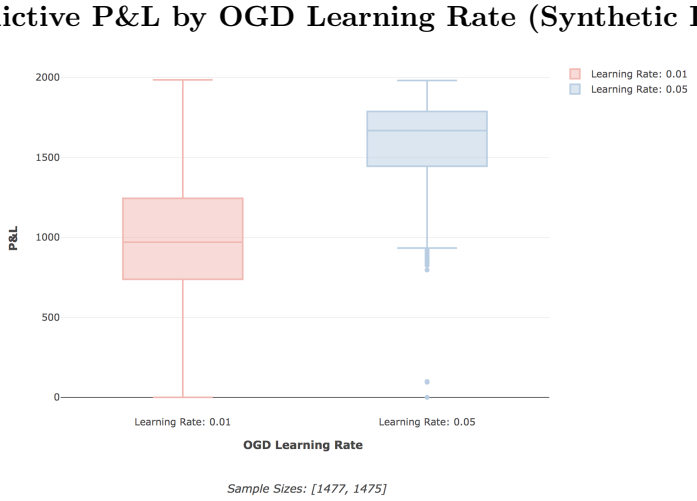


Figure 54: Dataset: Synthetic10 (7.1.2) ; Configuration 9 (10.2.9)
The networks trained on Synthetic data showed a sharp P&L increase as the OGD learning rate increased. It wasn't tested if they would experience the same turning point and begin to degrade as seen in networks for Actual data in 8.5.2

.

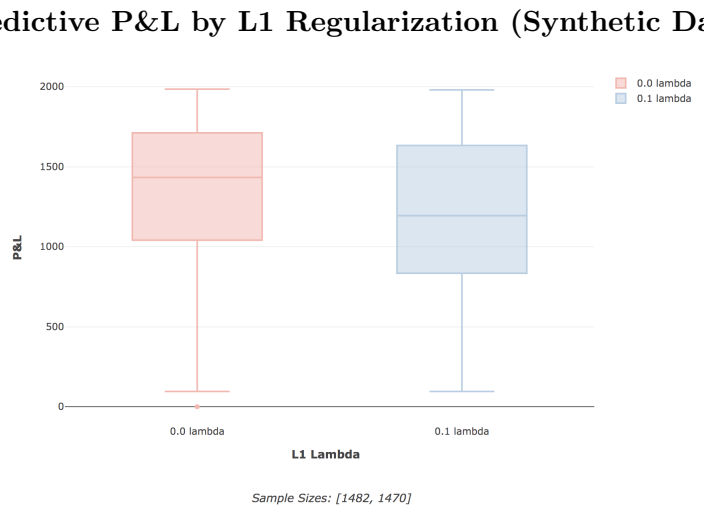


Figure 55: Dataset: Synthetic10 (7.1.2) ; Configuration 9 (10.2.9)

.

10.2 Configuration Sets Used

10.2.1 Configuration 1

- category: SAE
- sae_config_id: 0
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: NormalizeData;StandardizeData
- layer_sizes: (30,60,5); (30,60,60,5); (30,60,60,60,5); (30,120,5); (30,120,120,5); (30,120,120,120,5); (30,60,15); (30,60,60,15); (30,60,60,60,15); (30,120,15); (30,120,120,15); (30,120,120,120,15); (30,60,25); (30,60,60,25); (30,60,60,60,25); (30,120,25); (30,120,120,25); (30,120,120,120,25)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit
- encoding_activation: LinearActivation;SigmoidActivation;ReluActivation
- output_activation: LinearActivation;SigmoidActivation;ReluActivation
- learning_rate: 0.001;0.1;1.0;0.01;0.05;0.25;2.0
- minibatch_size: 20
- max_epochs: 500
- l1_lambda: 0.0
- min_learning_rate: 0.0
- epoch_cycle_max: 0
- ogd_learning_rate: 0.0

Samples: 1129

10.2.2 Configuration 2

- category: SAE
- sae_config_id: 0
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: NormalizeData
- layer_sizes: (30,60,5); (30,60,60,5); (30,60,60,60,5); (30,120,5); (30,120,120,5); (30,120,120,120,5); (30,60,15); (30,60,60,15); (30,60,60,60,15); (30,120,15); (30,120,120,15); (30,120,120,120,15); (30,60,25); (30,60,60,25); (30,60,60,60,25); (30,120,25); (30,120,120,25); (30,120,120,120,25)

- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit
- encoding_activation: LinearActivation;SigmoidActivation;ReluActivation
- output_activation: LinearActivation;SigmoidActivation;ReluActivation
- learning_rate: 0.001;0.1;1.0;0.01;0.05;0.25
- minibatch_size: 20
- max_epochs: 500
- l1_lambda: 0.0
- min_learning_rate: 0.0
- epoch_cycle_max: 0
- ogd_learning_rate: 0.0

Samples: 648

10.2.3 Configuration 3

- category: FFN
- sae_config_id: 3560;3574;3580;3590;3613;3620;3626;3638;3650;3671
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: NormalizeData;LimitedNormalizeData
- layer_sizes: (15,40,6); (12,40,6); (9,40,6); (6,40,6); (3,40,6); (15,40,40,40,6); (12,40,40,40,6); (9,40,40,40,6); (6,40,40,40,6); (3,40,40,40,6); (15,80,6);(12,80,6); (9,80,6); (6,80,6); (3,80,6); (15,80,80,80,6); (12,80,80,80,6); (9,80,80,80,6); (6,80,80,80,6); (3,80,80,80,6)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotNormalInit
- encoding_activation: ;LinearActivation
- output_activation: LinearActivation;ReluActivation
- learning_rate: 1.0e-5;0.0001;0.001;0.01
- minibatch_size: 20
- max_epochs: 500
- l1_lambda: 0.0
- min_learning_rate: 0.0
- epoch_cycle_max: 0
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.0001;0.001

Samples: 960

10.2.4 Configuration 4

- category: FFN
- sae_config_id: 3560;3574;3580;3590;3613;3620;3626;3638;3650;3671
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: NormalizeData;LimitedNormalizeData
- layer_sizes: (15,40,6); (12,40,6); (9,40,6); (6,40,6); (3,40,6); (15,40,40,40,6); (12,40,40,40,6); (9,40,40,40,6); (6,40,40,40,6); (3,40,40,40,6); (15,80,6);(12,80,6); (9,80,6); (6,80,6); (3,80,6); (15,80,80,80,6); (12,80,80,80,6); (9,80,80,80,6); (6,80,80,80,6); (3,80,80,80,6); (15,20,6); (12,20,6); (9,20,6); (6,20,6); (3,20,6); (15,20,20,6); (12,20,20,6); (9,20,20,6); (6,20,20,6); (3,20,20,6); (15,20,20,20,6); (12,20,20,20,6); (9,20,20,20,6); (6,20,20,20,6); (3,20,20,20,6)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotNormalInit
- encoding_activation: ;LinearActivation
- output_activation: LinearActivation;ReluActivation
- learning_rate: 1.0e-5;0.0001;0.001;0.01
- minibatch_size: 20
- max_epochs: 500
- l1_lambda: 0.0
- min_learning_rate: 0.0
- epoch_cycle_max: 0
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.0001;0.001

Samples: 1680

10.2.5 Configuration 5

- category: FFN
- sae_config_id: 8860;8872;8876;8888;8910;8800;8812;8816;8828;8850
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: LimitedNormalizeData

- layer_sizes: (15,40,6); (12,40,6); (9,40,6); (6,40,6); (3,40,6); (15,40,40,40,6); (12,40,40,40,6); (9,40,40,40,6); (6,40,40,40,6); (3,40,40,40,6); (15,80,6);(12,80,6); (9,80,6); (6,80,6); (3,80,6); (15,80,80,80,6); (12,80,80,80,6); (9,80,80,80,6); (6,80,80,80,6); (3,80,80,80,6)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotNormalInit
- encoding_activation: LinearActivation
- output_activation: LinearActivation
- learning_rate: 0.001;0.01
- minibatch_size: 20
- max_epochs: 500
- l1_lambda: 0.0
- min_learning_rate: 0.0
- epoch_cycle_max: 0
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.001

Samples: 80

10.2.6 Configuration 6

- category: SAE
- sae_config_id: 0
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: NormalizeData
- layer_sizes: (6,15,5); (6,15,15,5)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotNormalInit
- encoding_activation: LinearActivation;SigmoidActivation
- output_activation: SigmoidActivation
- learning_rate: 1.0;2.0;4.0
- minibatch_size: 20
- max_epochs: 100
- l1_lambda: 0.0
- min_learning_rate: 1.0
- epoch_cycle_max: 1
- ogd_learning_rate: 0.0

Samples: 192

10.2.7 Configuration 7

- category: SAE
- sae_config_id: 0
- steps: 5000
- deltas: 1,5,20;5,20,60;10,20,60
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 5
- scaling_function: LimitedNormalizeData
- layer_sizes: (30,120,60,25); (30,120,60,20); (30,120,60,15); (30,120,60,10); (30,120,60,5); (30,120,25); (30,120,20); (30,120,15); (30,120,10); (30,120,5); (30,120,120,25); (30,120,120,20); (30,120,120,15); (30,120,120,10); (30,120,120,5); (30,120,90,25); (30,120,90,20); (30,120,90,15); (30,120,90,10); (30,120,90,5); (30,90,60,25); (30,90,60,20); (30,90,60,15); (30,90,60,10); (30,90,60,5); (30,90,60,30,25); (30,90,60,30,20); (30,90,60,30,15); (30,90,60,30,10); (30,90,60,30,5); (30,120,60,30,25); (30,120,60,30,20); (30,120,60,30,15); (30,120,60,30,10); (30,120,60,30,5); (30,90,90,90,25); (30,90,90,90,20); (30,90,90,90,15); (30,90,90,90,10); (30,90,90,90,5);(30,90,90,25); (30,90,90,20); (30,90,90,15); (30,90,90,10); (30,90,90,5)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit;HeUniformInit;DCUniformInit
- encoding_activation: LinearActivation
- output_activation: LinearActivation
- learning_rate: 0.005;0.01;0.05;0.1
- minibatch_size: 20
- max_epochs: 400
- l1_lambda: 0.0
- min_learning_rate: 0.0001
- epoch_cycle_max: 100;300
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.0

Samples: 3266

10.2.8 Configuration 8

- category: SAE
- sae_config_id: 0
- steps: 5000
- deltas: 1,5,20;5,20,60;10,20,60
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 5

- scaling_function: LimitedNormalizeData
- layer_sizes: (3,12,6,2); (3,12,2); (3,12,12,2); (3,12,9,2); (3,9,6,2); (3,9,6,3,2); (3,12,6,3,2); (3,9,9,9,2); (3,9,9,2); (3,12,6,1); (3,12,1); (3,12,12,1); (3,12,9,1); (3,9,6,1); (3,9,6,3,1); (3,12,6,3,1); (3,9,9,9,1); (3,9,9,1)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit;HeUniformInit;DCUniformInit
- encoding_activation: LinearActivation
- output_activation: LinearActivation
- learning_rate: 0.005;0.01;0.05;0.1
- minibatch_size: 20
- max_epochs: 400
- l1_lambda: 0.0
- min_learning_rate: 0.0001
- epoch_cycle_max: 100;300
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.0

Samples: 1296

10.2.9 Configuration 9

- category: FFN
- sae_config_id: 18140;18914;18259;18311;19481;20662;18917;18260;18314;18766;18119;18191;19343;18344;18
- steps: 5000
- deltas: 1,5,20;5,20,60;10,20,60
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 5
- scaling_function: LimitedNormalizeData
- layer_sizes: (25,120,120,10); (20,120,120,10); (15,120,120,10); (10,120,120,10); (5,120,120,10); (25,120,120,120,10); (20,120,120,120,10); (15,120,120,120,10); (10,120,120,120,10); (5,120,120,120,10); (25,120,90,90,60,10); (20,120,90,90,60,10); (15,120,90,90,60,10); (10,120,90,90,60,10); (5,120,90,90,60,10); (25,120,90,60,10); (20,120,90,60,10); (15,120,90,60,10); (10,120,90,60,10); (5,120,90,60,10); (25,120,10); (20,120,10); (15,120,10); (10,120,10); (5,120,10); (25,120,60,10); (20,120,60,10); (15,120,60,10); (10,120,60,10); (5,120,60,10)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit;DCUniformInit;HeUniformInit
- encoding_activation: None
- output_activation: LinearActivation
- learning_rate: 0.01;0.05;0.1
- minibatch_size: 20
- max_epochs: 400

- l1_lambda: 0.0;0.1
- min_learning_rate: 0.0001
- epoch_cycle_max: 100
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.01;0.05

Samples: 3313

10.2.10 Configuration 10

- category: FFN
- sae_config_id: 25339;25778;25684;25846;25640;25767
- steps: 5000
- deltas: 1,5,20;5,20,60;10,20,60
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 5
- scaling_function: LimitedNormalizeData
- layer_sizes: (2,12,6,1); (1,12,6,1); (2,12,12,1); (1,12,12,1); (2,12,12,12,1); (1,12,12,12,1); (2,12,9,9,6,1); (1,12,9,9,6,1); (2,12,9,6,1); (1,12,9,6,1); (2,12,1); (1,12,1)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: HeUniformInit;XavierGlorotUniformInit;DCUniformInit
- encoding_activation: None
- output_activation: LinearActivation
- learning_rate: 0.01;0.05;0.1
- minibatch_size: 20
- max_epochs: 400
- l1_lambda: 0.0;0.1
- min_learning_rate: 0.0001
- epoch_cycle_max: 100
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.01;0.05

Samples: 1296

10.2.11 Configuration 11

- category: FFN
- sae_config_id: 1533;1497;1554;1639;147;1534;1468;1501;318;1284;1535;1553;1059;1508;333
- steps: 1
- deltas: 1,5,20;5,20,60;10,20,60

- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 5
- scaling_function: LimitedNormalizeData
- layer_sizes: (25,60,10); (20,60,10); (15,60,10); (10,60,10); (5,60,10); (25,120,10); (20,120,10); (15,120,10); (10,120,10); (5,120,10); (25,240,10); (20,240,10); (15,240,10); (10,240,10); (5,240,10); (25,120,120,10); (20,120,120,10); (15,120,120,10); (10,120,120,10); (5,120,120,10); (25,240,240,10); (20,240,240,10); (15,240,240,10); (10,240,240,10); (5,240,240,10); (25,60,60,10); (20,60,60,10); (15,60,60,10); (10,60,60,10); (5,60,60,10); (30,60,10); (30,120,10); (30,240,10); (30,60,60,10); (30,120,120,10); (30,240,240,10)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: DCUniformInit
- encoding_activation: None
- output_activation: LinearActivation
- learning_rate: 0.01
- minibatch_size: 32
- max_epochs: 100;500;1000;10
- l1_lambda: 0.0;0.1;0.5
- ifnull(tp.l2_lambda,0): 0
- min_learning_rate: 0.0001;1.0e-5
- epoch_cycle_max: 100;10
- is_denoising: 1
- denoising_variance: 0.0;0.1
- ogd_learning_rate: 0.005;0.01;0.05;0.1

Samples: 20737

10.2.12 Configuration 12

- category: SAE
- sae_config_id: 0
- steps: 1
- deltas: 1,5,20;5,20,60;10,20,60
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 5
- scaling_function: LimitedNormalizeData
- layer_sizes: (30,120,25); (30,120,20); (30,120,15); (30,120,10); (30,120,5); (30,120,120,25); (30,120,120,20); (30,120,120,15); (30,120,120,10); (30,120,120,5); (30,120,120,120,25); (30,120,120,120,20); (30,120,120,120,15); (30,120,120,120,10); (30,120,120,120,5); (30,120,120,120,120,25); (30,120,120,120,120,20); (30,120,120,120,120,15); (30,120,120,120,120,10); (30,120,120,120,120,5); (30,240,240,25); (30,240,240,20); (30,240,240,15); (30,240,240,10); (30,240,240,5); (30,240,240,240,25); (30,120,90,60,25); (30,120,90,60,20); (30,120,90,60,15); (30,120,90,60,10); (30,120,90,60,5); (30,240,25); (30,240,20); (30,240,15); (30,240,10); (30,240,5); (30,240,240,240,20); (30,240,240,240,15); (30,240,240,240,10); (30,240,240,240,5)

- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit;HeUniformInit;DCUniformInit
- encoding_activation: LinearActivation
- output_activation: LinearActivation
- learning_rate: 0.01;0.1
- minibatch_size: 32
- max_epochs: 2000;10000;20000
- l1_lambda: 0.5;0.0;0.1;1.0
- ifnull(tp.l2_lambda,0): 0
- min_learning_rate: 0.0001;1.0e-5
- epoch_cycle_max: 100;200
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.0

Samples: 1665

10.2.13 Configuration 13

- category: FFN
- sae_config_id: 1533;1497;1554;1639;147;1534;1468;1501;318;1284;1535;1553;1059;1508;333
- steps: 1
- deltas: 1,5,20;5,20,60;10,20,60
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 5
- scaling_function: LimitedNormalizeData
- layer_sizes: (25,60,10); (20,60,10); (15,60,10); (10,60,10); (5,60,10); (25,120,10); (20,120,10); (15,120,10); (10,120,10); (5,120,10); (25,240,10); (20,240,10); (15,240,10); (10,240,10); (5,240,10); (25,120,120,10); (20,120,120,10); (15,120,120,10); (10,120,120,10); (5,120,120,10); (25,240,240,10); (20,240,240,10); (15,240,240,10); (10,240,240,10); (5,240,240,10); (25,60,60,10); (20,60,60,10); (15,60,60,10); (10,60,60,10); (5,60,60,10); (30,60,10); (30,120,10); (30,240,10); (30,60,60,10); (30,120,120,10); (30,240,240,10); (10,60,60,60,10); (10,120,120,120,10); (10,240,240,240,10); (10,60,60,60,60,10); (10,120,120,120,120,10); (10,240,240,240,240,10)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: DCUniformInit
- encoding_activation: None
- output_activation: LinearActivation
- learning_rate: 0.01
- minibatch_size: 32
- max_epochs: 100;500;1000;10
- l1_lambda: 0.0;0.1;0.5

- ifnull(tp.l2_lambda,0): 0
- min_learning_rate: 0.0001;1.0e-5
- epoch_cycle_max: 100;10;-1
- is_denoising: 1;0
- denoising_variance: 0.0;0.1
- ogd_learning_rate: 0.005;0.01;0.05;0.1

Samples: 22033

10.2.14 Configuration 14

- category: SAE
- sae_config_id: 0
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: NormalizeData
- layer_sizes: (6,30,5); (6,30,30,5); (6,30,3); (6,30,30,3); (30,60,5); (30,60,60,5); (30,60,60,60,5); (30,120,5); (30,120,120,5); (30,120,120,120,5); (30,60,15); (30,60,60,15); (30,60,60,60,15); (30,120,15); (30,120,120,15); (30,120,120,120,15); (30,60,25); (30,60,60,25); (30,60,60,60,25); (30,120,25); (30,120,120,25); (30,120,120,120,25)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit
- encoding_activation: LinearActivation
- output_activation: LinearActivation
- learning_rate: 0.05;0.1
- minibatch_size: 20
- max_epochs: 1;1000
- l1_lambda: 0.0
- min_learning_rate: 0.0
- epoch_cycle_max: 0
- is_denoising: 1
- denoising_variance: 0.1;0.01;0.001;0.0001;1.0e-11
- ogd_learning_rate: 0.0

Samples: 243

10.2.15 Configuration 15

- category: SAE
- sae_config_id: 0
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: LimitedNormalizeData
- layer_sizes: (30,60,25); (30,60,60,25); (30,120,25); (30,120,120,25); (30,60,15); (30,60,60,15); (30,120,15); (30,120,120,15); (30,60,5); (30,60,60,5); (30,120,5); (30,120,120,5)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit
- encoding_activation: LinearActivation
- output_activation: LinearActivation
- learning_rate: 0.001
- minibatch_size: 20
- max_epochs: 500
- l1_lambda: 0.0
- min_learning_rate: 0.0
- epoch_cycle_max: 0
- is_denoising: 1
- denoising_variance: 0.01;0.05;0.1;0.15;0.2;0.25
- ogd_learning_rate: 0.0

Samples: 73

10.2.16 Configuration 16

- category: FFN
- sae_config_id: 318;1508;1639
- steps: 1
- deltas: 5,20,60;10,20,60;1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 5
- scaling_function: LimitedNormalizeData
- layer_sizes: (10,60,60,60,10); (10,120,120,120,10); (10,240,240,240,10); (10,60,60,60,60,10); (10,120,120,120,120,10); (10,240,240,240,240,10)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: DCUniformInit

- encoding_activation: None
- output_activation: LinearActivation
- learning_rate: 0.01
- minibatch_size: 32
- max_epochs: 100;500;10
- l1_lambda: 0.0
- ifnull(tp.l2_lambda,0): 0
- min_learning_rate: 0.0001
- epoch_cycle_max: -1;100;10
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.01;0.05;0.1

Samples: 1296

10.2.17 Configuration 17

- category: SAE
- sae_config_id: 0
- steps: 5000
- deltas: 1,5,20
- process_splits: 0.6
- training_splits: 0.8,1.0
- prediction_steps: 2
- scaling_function: LimitedNormalizeData
- layer_sizes: (18,40,15); (18,40,40,15); (18,40,40,40,15); (18,80,15); (18,80,80,15); (18,80,80,80,15); (18,40,12); (18,40,40,12); (18,40,40,40,12); (18,80,12); (18,80,80,12); (18,80,80,80,12); (18,40,9); (18,40,40,9); (18,40,40,40,9); (18,80,9); (18,80,80,9); (18,80,80,80,9); (18,40,6); (18,40,40,6); (18,40,40,40,6); (18,80,6); (18,80,80,6); (18,80,80,80,6); (18,40,3); (18,40,40,3); (18,40,40,40,3); (18,80,3); (18,80,80,3); (18,80,80,80,3)
- layer_activations: SigmoidActivation,ReluActivation,LinearActivation
- initialization: XavierGlorotUniformInit
- encoding_activation: LinearActivation
- output_activation: LinearActivation
- learning_rate: 0.001;0.01
- minibatch_size: 20
- max_epochs: 500
- l1_lambda: 0.0
- min_learning_rate: 0.0
- epoch_cycle_max: 0
- is_denoising: 0
- denoising_variance: 0.0
- ogd_learning_rate: 0.0

Samples: 120

11 References

[1] D. H. Bailey, J. Borwein, M. Lpez de Prado, and Q. J. Zhu, "The probability of backtest overfitting," *Journal of Computational Finance*, vol. 20, pp. 39–69, 4 2017. [Online]. Available: https://www.davidhbailey.com/dhbpapers/backtest-prob.pdf

[2] J. Murphy, "Technical analysis of financial markets," 01 1999.

[3] G. Griffioen, "Technical analysis in financial markets," *SSRN Electronic Journal*, 03 2003.

[4] M. N. Kahn, *Technical Analysis Plain and Simple: Charting the Markets in Your Language*. Financial Times Press, 2006.

[5] J. D. Schwager, *Getting Started in Technical Analysis*. Wiley, 1999.

[6] N. Johnson, G. Zhao, E. Hunsader, H. Qi, N. Johnson, J. Meng, and B. Tivnan, "Abrupt rise of new machine ecology beyond human response time," *Scientific reports*, vol. 3, p. 2627, 09 2013.

[7] B. Arthur, "Complexity in economics and financial markets," *Complexity*, vol. 1, pp. 20–25, 10 1995. [Online]. Available: https://onlinelibrary.wiley.com/doi/epdf/10.1002/cplx.6130010106

[8] J. Crutchfield, "Between order and chaos," *Nature Physics*, vol. 9, pp. 382–382, 06 2013.

[9] N. Packard, J. Crutchfield, and R. Shaw, "Geometry from a time series," *Phys. Rev. Lett.*, vol. 45, p. 712, 09 1980.

[10] F. Takens, "Detecting strange attractors in turbulence," *Dynamical Systems and Turbulance serial Lecture notes in Mathematics*, vol. 898, 01 1981.

[11] A. Skabar and I. Cloete, "Neural networks, financial trading and the efficient markets hypothesis," *ACSC '02: Proceedings of the Twenty-fifth Australasian Conference on Computer Science*, vol. 4, 02 2002.

[12] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 01 2015.

[13] A. Ivakhnenko, "Polynomial theory of complex systems. ieee trans syst man cybern," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 1, pp. 364 – 378, 11 1971.

[14] P. Werbos and P. John, "Beyond regression : new tools for prediction and analysis in the behavioral sciences /," 01 1974.

[15] H. Siegelmann, "Theoretical foundations of recurrent neural networks," 11 2019.

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[17] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, 01 2017.

[18] Y. Lecun, "A theoretical framework for back-propagation," 01 1992.

[19] P. Werbos, *Applications of advances in nonlinear sensitivity analysis*, 01 1970, vol. 38, pp. 762–770.

[20] D. Rumelhart, G. Hinton, and W. RJ, "Learning internal representations by error propagation," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, 07 1986.

[21] L. Cun, Y. Boser, B. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Back-propagation applied to handwritten zip-code recognition," *Neural Computation - NECO*, 01 1992.

[22] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," *30th International Conference on Machine Learning, ICML 2013*, 11 2012.

[23] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, 05 2015.

[24] Y. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," *NIPS*, vol. 27, 06 2014.

[25] R. Ge, F. Huang, C. Jin, and Y. Yuan, "Escaping from saddle points — online stochastic gradient for tensor decomposition," 03 2015. [Online]. Available: http://proceedings.mlr.press/v40/Ge15.pdf

[26] K. Hornik, "Multilayer feed-forward networks are universal approximators," *Neural Networks*, vol. 2, pp. 359–366, 1989.

[27] H. Wu, "Global stability analysis of a general class of discontinuous neural networks with linear growth activation functions," *Inf. Sci.*, vol. 179, pp. 3432–3441, 09 2009.

[28] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, 01 2010.

[29] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," *Proceedings of the 14th International Conference on Artificial Intelligence and Statisitics (AISTATS) 2011*, vol. 15, pp. 315–323, 01 2011.

[30] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," *Adv. Neural Inf. Process. Syst.*, vol. 19, pp. 153–160, 01 2007.

[31] G. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, pp. 1527–54, 08 2006.

[32] M. Ranzato, C. Poultney, S. Chopra, and Y. Lecun, "Efficient learning of sparse representations with an energy-based model," 01 2006.

[33] G. Hinton and R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science (New York, N.Y.)*, vol. 313, pp. 504–7, 08 2006.

[34] N. Roux and Y. Bengio, "1 representational power of restricted boltzmann machines and deep belief networks," *Neural computation*, vol. 20, pp. 1631–49, 07 2008.

[35] Y. Bengio and S. Bengio, "Modeling high-dimensional discrete data with multi-layer neural networks," 02 2001. [Online]. Available: http://papers.nips.cc/paper/1679-modeling-high-dimensional-discrete-data-with-multi-layer-neural-networks.pdf

[36] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. Lecun, "Pedestrian detection with unsupervised multi-stage feature learning," *Proceedings / CVPR, IEEE Computer Society Conference on Computer Vision and Pattern Recognition. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 12 2012.

[37] A. Krizhevsky, I. Sutskever, and G. Hinton, "Imagenet classification with deep convolutional neural networks," *Neural Information Processing Systems*, vol. 25, 01 2012.

[38] A. oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," 09 2016.

[39] D. Cirean, U. Meier, L. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural computation*, vol. 22, pp. 3207–20, 12 2010.

[40] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, pp. 1798–1828, 08 2013.

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *IEEE International Conference on Computer Vision (ICCV 2015)*, vol. 1502, 02 2015.

[42] R. Schaefer, "Subset selection in regression," *Technometrics*, vol. 34, 03 2012.

[43] D. Donoho, "High-dimensional data analysis: The curses and blessings of dimensionality," *AMS Math Challenges Lecture*, pp. 1–32, 01 2000.

[44] J. Fan and R. Li, "Statistical challenges with high dimensionality: Feature selection in knowledge discovery," *Proc. Madrid Int. Congress of Mathematicians*, vol. 3, 03 2006.

[45] J. Fan and Y. Fan, "High dimensional classification using features annealed independence rules," *Annals of statistics*, vol. 36, pp. 2605–2637, 02 2008.

[46] E. Fama, "The behavior of stock market price," *Journal of Business - J BUS*, vol. 38, 01 1965.

[47] M. Lngkvist, L. Karlsson, and A. Loutfi, "A review of unsupervised feature learning and deep learning for time-series modeling," *Pattern Recognition Letters*, vol. 42, 06 2014.

[48] G. Hinton, "Article training products of experts by minimizing contrastive divergence," *Neural computation*, vol. 14, pp. 1771–800, 09 2002.

[49] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *Journal of Machine Learning Research*, vol. 11, pp. 3371–3408, 12 2010.

[50] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, "Why does unsupervised pre-training help deep learning?" *Journal of Machine Learning Research*, vol. 11, pp. 625–660, 02 2010.

[51] L. Yisheng, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang, "Traffic flow prediction with big data: A deep learning approach," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, pp. 865–873, 01 2014.

[52] L. Takeuchi, "Applying deep learning to enhance momentum trading strategies in stocks," 2013.

[53] Y. Zhao, J. Li, and L. Yu, "A deep learning ensemble approach for crude oil price forecasting," *Energy Economics*, vol. 66, 06 2017.

[54] L. Troiano, E. Villa, and P. Kriplani, "On feature reduction using deep learning for trend prediction in finance," 04 2017.

[55] W. Bao, J. Yue, and Y. Rao, "A deep learning framework for financial time series using stacked autoencoders and long-short term memory," *PLoS ONE*, vol. 12, 07 2017.

[56] D. Hsu, "Time series compression based on adaptive piecewise recurrent autoencoder," 07 2017.

[57] S. Albers, "Online algorithms: a survey," *Mathematical Programming*, vol. 97, pp. 3–26, 05 2003. [Online]. Available: https://link.springer.com/content/pdf/10.1007%2Fs10107-003-0436-0.pdf

[58] L. Bottou and Y. Lecun, "Large scale online learning." *Advances in Neural Information Processing Systems 16*, 03 2004.

[59] Y. Lecun, L. Bottou, G. Orr, and K.-R. Mller, *Efficient BackProp*, 01 1998, vol. 1524, pp. 546–546.

[60] L. Bottou and N. Murata, *Stochastic Approximations and Efficient Learning, The Handbook of Brain Theory and Neural Networks*, 2nd ed. The MIT Press, 2019.

[61] S. Shalev-Shwartz, Y. Singer, N. Srebro, and A. Cotter, "Pegasos: Primal estimated sub-gradient solver for svm," *Math. Program.*, vol. 127, pp. 3–30, 03 2011.

[62] T. Zhang, "Solving large scale linear prediction problems using stochastic gradient descent algorithms," 01 2004.

[63] P. Tseng, "An incremental gradient(-projection) method with momentum term and adaptive stepsize rule," *SIAM Journal on Optimization*, vol. 8, 04 1999.

[64] P. Bartlett, E. Hazan, and A. Rakhlin, "Adaptive online gradient descent." *Advances in Neural Information Processing Systems 20 - Proceedings of the 2007 Conference*, 01 2007.

[65] J. Langford, L. Li, and T. Zhang, "Sparse online learning via truncated gradient," *Journal of Machine Learning Research*, vol. 10, 07 2008.

[66] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 07 2011.

[67] M. Zeiler, "Adadelta: An adaptive learning rate method," vol. 1212, 12 2012.

[68] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint*, vol. arXiv, 07 2012.

[69] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," *30th International Conference on Machine Learning, ICML 2013*, vol. 1302, 02 2013.

[70] S. Wang and C. Manning, "Fast dropout training," in *Proceedings of the 30th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, S. Dasgupta and D. McAllester, Eds., vol. 28, no. 2. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 118–126. [Online]. Available: http://proceedings.mlr.press/v28/wang13a.html

[71] L. Smith, "Cyclical learning rates for training neural networks," 03 2017, pp. 464–472.

[72] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," 08 2016.

[73] J. Ioannidis, "Why most published research findings are false," *CHANCE*, vol. 32, pp. 4–13, 01 2019.

[74] R. Mclean and J. Pontiff, "Does academic research destroy stock return predictability?" *The Journal of Finance*, vol. 71, 05 2013.

[75] F. Schorfheide and K. Wolpin, "On the use of holdout samples for model selection," *American Economic Review*, vol. 102, 05 2012.

[76] M. Lopez de Prado, "The future of empirical finance," *The Journal of Portfolio Management*, vol. 41, pp. 140–144, 07 2015.

[77] S. M. Weiss and C. A. Kulikowski, *Computer Systems That Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991.

[78] D. Hawkins, "The problem of overfitting," *Journal of chemical information and computer sciences*, vol. 44, pp. 1–12, 05 2004.

[79] P. Hansen, J. Nason, and A. Lunde, "The model confidence set," *Econometrica*, vol. 79, pp. 453–497, 03 2010.

[80] D. Aparicio and M. Lopez de Prado, "How hard is it to pick the right model? mcs and backtest overfitting," *Algorithmic Finance*, vol. 7, pp. 53–61, 01 2018. [Online]. Available: https://ssrn.com/abstract=3044740

[81] A. Lo, "The statistics of sharpe ratios," *Financial Analysts Journal*, vol. 58, 02 2003.

[82] D. H. Bailey, J. Borwein, M. Lopez de Prado, and Q. J. Zhu, "Pseudo-mathematics and financial charlatanism: The effects of backtest overfitting on out-of-sample performance," *Notices of the American Mathematical Society*, vol. 61, pp. 458–471, 4 2014. [Online]. Available: https://ssrn.com/abstract=2308659

[83] D. Bailey and M. Lopez de Prado, "The deflated sharpe ratio: Correcting for selection bias, backtest overfitting, and non-normality," *The Journal of Portfolio Management*, vol. 40, pp. 94–107, 09 2014.

[84] G. Hinton, "A practical guide to training restricted boltzmann machines[j]," *Momentum*, vol. 9, pp. 926–947, 01 2010.

[85] D. Wilcox and T. Gebbie, "Hierarchical causality in financial economics," *SSRN Electronic Journal*, 08 2014.

[86] A. A. Ole Peters, *Ergodicity Economics*. London Mathematical Laboratory, 2018.