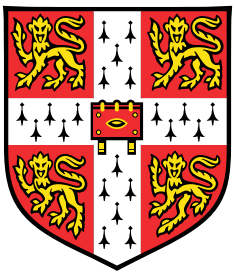
Development and applications of neural networks for economic forecasting



Joseph Fisher

Department of Economics
University of Cambridge

This dissertation is submitted for the degree of
Doctor of Philosophy

Trinity College

February 2020

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee

Joseph Fisher
February 2020

Development and applications of neural networks for economic forecasting

Joseph Fisher

Neural networks are one of a variety of machine learning models which are beginning to be widely used in economic forecasting applications. Despite this, there is relatively little understanding of the conditions in which neural networks provide accurate forecasts, the uncertainty bounds which can be put on such forecasts, and the most suitable network types and parameters for forecasting in the relatively small-sample settings encountered within economics. This thesis fits into a growing body of literature which aims to answer some of these questions. In Chapter 1, we present a detailed study of the accuracy of neural networks for forecasting financial volatility, and present a novel adaptation of networks for time series data. In Chapter 2, we present an adaptation to the output layer of neural networks which allows the generation of prediction intervals for forecasts, and present variants to the architecture which improve the accuracy of these prediction intervals. In Chapter 3, we focus on confidence intervals, and present the first simulation study of the suitability of bootstrapping for neural networks for generating confidence intervals with correct coverage. Finally, in Chapter 4, we focus on an alternate application of neural networks in forecasting; that of converting free-form text data into indices, and present a novel neural network architecture for improving the recognition of named entities (companies, people etc.) in text, a necessary first step in such forecasting.

Acknowledgements

I would like to acknowledge my supervisor, Professor Oliver Linton, for helpful feedback and comments throughout this thesis. In addition, I'd like to acknowledge Dr Debopam Bhattacharya, my advisor, Professor Alexey Onatskiy, Professor Dacheng Xiu, Dr Donald Robertson, Dr Joseph Barunik, Dr Weining Wang, Dr Ekaterina Shutova, Professor Hamish Lowe and Alexei De Boeck for helpful comments. Finally, I'd like to acknowledge the support of Dr Andreas Vlachos, the co-author of the work in Chapter 4, for welcoming me into the world of NLP!

Table of contents

List of figures

List of tables

Introduction

Neural networks are one of a variety of machine learning models which are beginning to be widely used in economic forecasting applications. Despite this, there is relatively little understanding of the conditions in which neural networks provide accurate forecasts, the uncertainty bounds which can be put on such forecasts, and the most suitable network types and parameters for forecasting in the relatively small-sample settings encountered within economics. This thesis fits into a growing body of literature which aims to answer some of these questions. In Chapter 1, we present a detailed study of the accuracy of neural networks for forecasting financial volatility, and present a novel adaptation of networks for time series data. In Chapter 2, we present an adaptation to the output layer of neural networks which allows the generation of prediction intervals for forecasts, and present variants to the architecture which improve the accuracy of these prediction intervals. In Chapter 3, we focus on confidence intervals, and present the first simulation study of the suitability of bootstrapping for neural networks for generating confidence intervals with correct coverage. Finally, in Chapter 4, we focus on an alternate application of neural networks in forecasting; that of converting free-form text data into indices, and present a novel neural network architecture for improving the recognition of named entities (companies, people etc.) in text, a necessary first step in such forecasting.

In this introduction, we first give an overview of the applications within economics which are being found for machine learning, before summarizing the development and use cases of neural networks, the particular machine learning model on which this research focuses. Finally, we give a summary of each of the four chapters in turn.

Machine learning in economics

Machine learning models have been developed to produce accurate predictions of a target variable, y, given a set of explanatory variables, x. The models, of which neural networks are one commonly used example, are flexible enough to approximate a large variety of non-linear functions. As such, given a sufficiently large dataset, they can be optimized to produce out-of-sample prediction which is often more accurate than linear models. That said, as most clearly expressed in Mullainathan and Spiess [2017], prediction is a fundamentally different problem from estimation of model parameters, β, as necessary for causal analysis.

Indeed, there is no guarantee than an accurate predictive model for y yields consistent model parameter estimates.

Despite this, there has been a gradual development of literature within econometrics which seeks to find areas in which the predictive power of machine learning techniques can be sensibly integrated. Overviews of such work can be found in Mol et al. [2017] (2017), Varian [2014], Kleinberg et al. [2015] and Mullainathan and Spiess [2017] amongst others. Broadly speaking, applications fall into two categories. The first involves isolating a predictive element within a broader causal framework. For example, Brian Lee and Stuart [2010] use machine learning methods to estimate the propensity score, Athey and Imbens [2016] use regression trees to estimate heterogeneous treatment effects, Chernozhukov et al. [2016] allows machine learning methods to be used for high-dimensional controls in a two-stage process, and Jason Hartford and Taddy [2017] use neural networks in the first stage of instrumental variables regression.

A second wave of applications involve using machine learning methods directly in prediction problems, where accuracy is more important than parameter estimation. Time series forecasting is an obvious example here, with two of many examples being Xiong et al. [2015] and Donaldson and Kamstra [1997], which use neural networks to forecast stock volatility.[1] Aside from forecasting, there are a growing range of policy problems for which accurate prediction is important. Kleinberg et al. [2015] uses the example of predicting the life expectancy of potential candidates for major operations, to ensure that treatment is effectively allocated. Kleinberg et al. [2017] find that machine learning algorithms can make more accurate decisions than judges in deciding whether or not to send suspects to jail while they await trial. Two further examples of pure prediction problems include Dana Chadler and List [2011], which predicts the youths most at risk of gun crime for targeting social interventions, and Jonah Rockoff and Staiger [2011], which aims to predict which teachers will be the most effective prior to a hiring decision.

Finally, there is a growing body of literature which aims to generate indices from alternative data sources such as images or text, often enabled through machine learning pre-processing of the raw data. For example, in Ying et al. [2019], the authors process satellite images to measure of the rate of property development on previously unused ground, which they propose as an indicator of economic activity. Ranco et al. [2015]; Nisar and Yeung [2018] investigate the relationship between measures of sentiment extracted from Twitter data and stock returns. Loughran and McDonald [2011] and Tetlock [2007] also try to forecast stock market patterns, but instead use news articles as the data source, whilst Baker et al. [2016]

[1] Given machine learning models tend to only out-perform more simple structures in applications where there is a relatively large dataset, forecasting applications tend to focus on financial indices

and Nyman et al. [2018] focus instead on estimating policy uncertainty and macro sentiment from news. Machine learning methods, and in particular neural networks, are well suited to the pre-processing stage of such projects; the final chapter fits into this literature, pushing the accuracy of entity recognition in text data sources.

Neural Networks

Neural networks at the most basic level are a non-linear transformation of an input variable, achieved via a set of learned weights and activation functions. In recent years, there has been a huge resurgence in interest in neural networks, as increased computational resources have allowed the training of significantly larger networks on larger datasets, resulting in massively improved performance in natural language processing, image processing and reinforcement learning. Alongside an increase in network size, the other key driver of progress has been innovations in network architecture, from Long Short-Term Memory (LSTM) networks for time series data (Hochreiter and Schmidhuber [1997]) to convolutional neural networks (CNNs) for images (Krizhevsky et al. [2017]) and Transformer networks (Vaswani et al. [2017]) for text data.

The flexibility of neural networks, and the scope for innovation to the network architecture, have led to significant and rapid increases in accuracy on a range of prediction tasks. However, understanding of the theoretical foundations which govern network performance has in some sense been left behind; optimization of the network weights for a particular problem remains something of a dark art, and the generation of accurate confidence and prediction intervals for networks remains to a large extent ignored. Whilst potentially not necessary for tasks such as image recognition, estimates of the uncertainty bounds on network predictions are generally desirable/essential in the applications of neural networks in economics, be it in the first stage of an instrumental variables setup, Jason Hartford and Taddy [2017], or in a time series prediction task, Donaldson and Kamstra [1997]. The practical side of achieving good network convergence, whilst not the focus of this work, is a recurring theme throughout, with all results relying on a careful hyperparameter search, whilst the accuracy of confidence and prediction intervals is the focus of Chapters 2 and 3 respectively.

Chapter 1: Forecasting financial volatility with neural networks

In Chapter 1, we aim to demonstrate that neural networks, if carefully optimized, are able to provide accurate forecasts even in the relatively small-sample settings encountered within economics and finance. We provide the first comparison of the performance of basic neural networks, convolutional neural networks and recurrent neural networks in forecasting financial

volatility. We compare out-of-sample forecasts across a range of financial indices, error metrics and forecast horizons, using realized volatility as a proxy for true volatility. The neural network models are shown to outperform popular models which have emerged from the financial volatility literature, including GARCH and HAR (Heterogeneous AR), with recurrent neural networks consistently providing the best out-of-sample errors. In addition, we propose a novel moving-average adaptation of the three variants of neural networks, and demonstrate that this adaptation consistently improves forecasting performance.

Chapter 2: Improving prediction intervals for neural networks using the quantile loss function

One of the commonly cited reasons for avoiding the use of neural networks in forecasting applications is that there are limited methods of generating accurate prediction intervals for forecasts. In Chapter 2, we aim to address this problem by focusing on improving the accuracy of prediction intervals for neural network based forecasts. We propose the use of the quantile loss function for generating such prediction intervals, and show that prediction intervals generated in this fashion significantly outperform previous approaches in terms of accuracy, both in simulated and real-world datasets. In addition, we propose three novel architecture variants for predicting the quantiles; isolated training of weights for each quantile output layer, modelling the quantile in an additive fashion from the mean prediction, and residual connections to the output layers. In each case, we show that the accuracy of the prediction intervals is increased, with their combination yielding substantially more reliable prediction intervals than previous approaches.

Chapter 3: Does bootstrapping neural networks work?

Chapter 3 focuses instead on confidence intervals around the network's mean prediction. To approximate these intervals, bootstrapping is a common approach. Early work established the validity of this method asymptotically, under a set of restrictive assumptions. However, due to a limitation in compute power when the initial work was developed, there remains very little understanding of how bootstrapping performs in limited sample-size settings in practice. In particular, there is no evidence that bootstrapping is able to approximate the uncertainty introduced by random weight initialization and sample splitting, two areas which cannot be accounted for in closed-form estimators. We present the first in-depth simulation study to answer this question, and show that broadly speaking, bootstrapping is effective at providing correctly sized standard errors, with coverage of the true function converging towards the correct levels as sample sizes increase. We extend the analysis to Recurrent Neural Networks

(RNNs), presenting the first experiments with the block bootstrap for neural networks, and show that the results from the I.I.D. case generally carry over, although convergence to correct coverage is slower. Finally, we include details of a novel computational architecture, which speeds up the training of multiple small neural networks on a GPU by a factor of 40 or more, allowing efficient application of bootstrapping in practice.

Chapter 4: Improving nested named entity recognition for more accurate sentiment indices

Finally, in Chapter 4, we focus on an alternate use of neural networks in economic forecasting; that of converting raw text data, such as news articles, companies 10-K filings and central bank reports, into sentiment indices which can be used for tasks such as measuring uncertainty, forecasting volatility or analyzing the impact of central bank communications. Current methods of building the indices rely on simple string matching, and sentiment-related word lists, both of which are inherently noisy. We suggest that more accurate measurements can be achieved by processing text with neural networks, and develop a novel neural network architecture for nested named entity recognition, the first stage in such an approach. Our architecture first merges tokens and/or entities into entities forming nested structures, and then labels each of them independently. Unlike previous work, our approach predicts real-valued instead of discrete segmentation structures, which allow it to combine word and nested entity embeddings while maintaining differentiability, allowing processing of articles 60 times faster than previous methods. We evaluate the accuracy of our approach using the ACE 2005 Corpus, where it achieves state-of-the-art F1 of 74.6, further improved with contextual embeddings (BERT) to 82.4, an overall improvement of close to 8 F1 points over previous approaches.

Such improvements in the accuracy with which we can identify entities in raw text data convert into less noisy extraction of companies when building sentiment indices, and ultimately more accurate and useful indices.

Bibliography

Athey, S. and Imbens, G. (2016). Recursive partitioning for heterogeneous causal effects: Table 1. *Proceedings of the National Academy of Sciences*, 113:7353–7360.

Baker, S. R., Bloom, N., and Davis, S. J. (2016). Measuring Economic Policy Uncertainty*. *The Quarterly Journal of Economics*, 131(4):1593–1636.

Brian Lee, J. L. and Stuart, E. (2010). Improving propensity score weighting using machine learning. *Statistics in Medicine 29 (3)*, pages 337–348.

Chernozhukov, V., Chetverikov, D., Demirer, M., Duflo, E., Hansen, C., and Newey, W. (2016). Double machine learning for treatment and causal parameters. *arXiv:1608.00060*.

Dana Chadler, S. L. and List, J. (2011). Predicting and preventing shootings among at-risk youth. *American Economic Review 101 (3)*, pages 288–292.

Donaldson, R. and Kamstra, M. (1997). An artificial neural network-garch model for international stock return volatility. *Journal of Empirical Finance*, 4(1):17 – 46.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

Jason Hartford, Greg Lewis, K. L.-B. and Taddy, M. (2017). Counterfactual prediction with deep instrumental variables networks. *Proceedings of the 34th International Conference on machine learning 70*, pages 1414–1423.

Jonah Rockoff, Brian Jacob, T. K. and Staiger, D. (2011). Can you recognize an effective teacher when you recruit one? *Education Finance and Policy 6 (1)*, pages 43–74.

Kleinberg, J., Lakkaraju, H., Leskovec, J., Ludwig, J., and Mullainathan, S. (2017). Human decisions and machine predictions. *NBER Working Paper 23180*.

Kleinberg, J., Ludwig, J., Mullainathan, S., and Obermeyer, Z. (2015). Prediction policy problems. *American Economic Review*, 105(5):491–95.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90.

Loughran, T. and McDonald, B. (2011). When is a liability not a liability? textual analysis, dictionaries, and 10-ks. *The Journal of Finance*, 66(1):35–65.

Mol, C. D., Gautier, E., Giannone, D., Mullainathan, S., Reichlin, L., van Dijk, H., and Wooldridge, J. (2017). *Big data in Economics: Evolution or Revolution?*, pages 612–632.

Mullainathan, S. and Spiess, J. (2017). Machine learning: an applied econometric approach. *Journal of Economic Perspectives 32 (2)*, pages 87–106.

Nisar, T. and Yeung, M. (2018). Twitter as a tool for forecasting stock market movements: A short-window event study. *The Journal of Finance and Data Science*, 4.

Nyman, R., Kapadia, S., Tuckett, D., Gregory, D., Ormerod, P., and Smith, R. (2018). News and narratives in financial systems: exploiting big data for systemic risk assessment. Bank of England working papers 704, Bank of England.

Ranco, G., Aleksovski, D., Caldarelli, G., and Mozetic, I. (2015). Investigating the relations between twitter sentiment and stock prices. *CoRR*, abs/1506.02431.

Tetlock, P. C. (2007). Giving content to investor sentiment: The role of media in the stock market. *The Journal of Finance*, 62(3):1139–1168.

Varian, H. (2014). Big data: New tricks for econometrics. *The Journal of Economic Perspectives*, 28:3–28.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

Xiong, R., Nichols, E., and Shen, Y. (2015). Deep learning stock volatilities with Google domestic trends. *arXiv, https:// arxiv.org/ abs/ 1512.04916*.

Ying, Q., Hansen, M. C., Sun, L., Wang, L., and Steininger, M. (2019). Satellite-detected gain in built-up area as a leading economic indicator. *Environmental Research Letters*, 14(11):114015.

Chapter 1

Forecasting financial volatility with neural networks

We provide the first comparison of the performance of feedforward neural networks, convolutional neural networks and recurrent neural networks in forecasting financial volatility. We compare out-of-sample forecasts across a range of financial indices, error metrics and forecast horizons, using realized volatility as a proxy for true volatility. The neural network models are shown to outperform popular models which have emerged from the financial volatility literature, including GARCH and HAR (Heterogeneous AR), with recurrent neural networks consistently providing the best out-of-sample errors. In addition, we propose a novel moving-average adaptation of the three variants of neural network, and demonstrate that this adaptation consistently improves forecasting performance.

1.1 Introduction

Volatility forecasting is a crucial task in asset pricing and risk management for investors, regulators and financial intermediaries. A wide range of autoregressive conditional heteroskedasticity (ARCH) and stochastic volatility models have been developed for this task in the econometric literature since the seminal paper of Engle [1982]. Comparison of approaches is made difficult by the latent nature of volatility, and in recent years, realized volatility (RVol) measures based on intraday data have emerged as the preferred proxy to the true process. Within this framework, ARCH-type models have been to some extent superseded by specifications which are estimated using realized volatility directly. The heterogeneous AR (HAR) model of Corsi [2009] has consistently demonstrated strong empirical performance amongst such models, and as such will provide the baseline for comparison of models in this study.

Advances in machine learning, and in particular neural networks, have attracted a significant amount of attention in the last few years. State-of-the-art results have been achieved in image recognition, voice recognition and language translation, amongst other tasks, through the development of neural network architectures and increased computing power. Although feedforward neural networks (NN) have held a small presence in the financial time series

forecasting literature for many years, the area has failed to gain any significant traction. Studies have tended to be small in scale, have failed to take advantage of modern developments and best practices from machine learning, and their results presented relative to models which have long been superseded in academic econometrics. This paper aims to compensate for some of these shortcomings, by providing a comprehensive comparison of modern neural network variants with models which are prevalent in econometrics, in the domain of financial indices volatility forecasting. In particular, we aim to assess whether carefully optimized neural networks are able to provide competitive forecast accuracy even in the relatively small-sample setting[1] of financial time series, and as such should be seen as a useful tool in the econometrician's forecasting toolbox.

In addition to assessing whether neural networks can be competitive with traditional econometric methods, we also aim to answer the question of which neural network architecture is most suited to financial time series forecasting. To do so, we present a comparison of the out-of-sample forecasting ability of three neural network variants. Firstly, feedforward neural networks (NNs), which have existed in statistics for many years, and have made occasional appearances in econometrics. Secondly, convolutional neural networks (CNNs), which have gained prominence in the last few years in the field of image recognition. Finally, recurrent neural networks (RNNs), which are widely used for natural language models. Comparison is made across ten financial indices, including the DJIA, S&P 500 and FTSE, and across a range of forecasting horizons, ranging from 1-day-ahead to 30-day-ahead. The main error metric used is quasi-likelihood (QL) loss, although mean squared error (MSE) statistics are also reported in places. All three neural network based models are shown to provide consistently lower error rates than the baselines, with RNNs generally performing the best.

Finally, we propose a novel adaptation to each type of neural network which involves allowing the networks to use the error from the previous steps prediction as an explanatory variable, in the style of MA models. This is novel across time series applications of neural networks in all fields, and involves a modification to the application of gradient descent in finding co-efficient values. It is demonstrated that this adaptation improves out-of-forecast results across all three network types.

1.1.1 Motivations

Machine learning, a subset of the field of "artificial intelligence", has existed as an area of research for many decades. The discipline is generally concerned with producing accurate prediction tools in large data settings. For example, the archetypal machine learning problem

[1] The sample sizes in this paper, of a few thousand observations, are small relative to the datasets of millions of images/text documents/tweets etc. on which neural networks have achieved remarkable success recently.

is classifying hand written digits automatically, which is achieved by training an algorithm on many thousands of pre-labelled images of digits. In such settings the accuracy of the prediction is paramount, and there is little to no value in establishing the relative causal influence of each pixel on the prediction. As such, the methods which have developed are often very effective at prediction, but complex to interpret (or "black boxes"). In particular, there is generally no causal mechanism between the explanatory variables and the output identified.

This explains why the crossover between mainstream econometrics and machine learning has remained limited. One area of econometrics in which accurate prediction (often based on large datasets) is key is financial time series forecasting. It is often also the case in forecasting applications that causal interpretation is of limited importance. If we are a practitioner forecasting a stock's volatility based on lags of returns, we care only about the accuracy of that forecast for choosing our portfolio. Although the causal effect of each specific lag may be of academic interest, is is of no practical concern as we do not have the ability to manipulate the results by changing the observations of returns. Consequently, there have been a steady stream of experimental papers which apply machine learning methods to forecasting financial time series. The work of Donaldson and Kamstra [1997], in which they modify a GARCH model in a semi-parametric fashion using a feedforward neural network is one example.

Despite work such as this, which has achieved a mixed degree of success, the use of machine learning methods for forecasting problems remains rare amongst econometricians. There are a range of potential reasons for this, some of which we hope to address in this paper.

Significantly, existing literature fails to provide a comparison of the best-performing methods from econometrics and machine learning concurrently. The majority of work either uses antiquated machine learning methods which have become almost entirely redundant in the recent literature, or compares modern machine learning methods to outdated econometric models. In the econometric volatility-forecasting literature, RVol has emerged as the dominant proxy for true volatility, and models which forecast directly using lags of realized variance (RV) (such as the HAR), as the dominant specifications. An appraisal of machine learning methods therefore needs to be undertaken in the RVol framework. The majority of existing papers, such as Donaldson and Kamstra [1997] and Xiong et al. [2015] use squared returns or high/low price based proxies for volatility.

On the machine learning side there has been a significant development and refinement of neural network models in recent years, and in particular the processes used to optimize them. The feedforward NN has long been largely redundant in the literature, having been replaced by variants of the CNN and the RNN. Despite this, the vast majority of research comparing econometric and machine learning techniques uses the feedforward NN only, and

often fails to take advantage of modern best practices for optimization. Vortelinos [2015] is one of the closest papers to our own, in that a comparison between HAR and a range of machine learning methods is undertaken in a realized volatility framework. Unfortunately, the author uses only the feedforward NN, restricts the size of the network arbitrarily, and uses an optimization method which is unable to prevent overfitting. Liu et al. [2018] also forecasts realized volatility with the HAR model and neural networks, but limit their analysis to recurrent neural networks only. Donaldson and Kamstra [1997] and Miranda and Burgess [1997] are two other examples of a wide range of literature which is limited to using only the feedforward NN. In concurrent work, Bucci [2020] provides a comparison of feedforward NNs and recurrent NNs with the auto-regressive fractionally integrated moving average (ARFIMA) family of models for forecasting realized volatility, with a focus on a setting with a large number of additional explanatory variables.

One of the key motivations for this work is to provide the first thorough comparison of the potential of the RNN and the CNN (the two dominant neural network types in modern machine learning) in a financial time series forecasting application. Convolutional neural networks were developed for image recognition tasks, as the structure of the network allows the same patterns to be picked up in multiple areas of the input vector, enabling recognition of an object in a picture regardless of its precise position. Despite the majority of the work with CNNs being image-based, there have been a number of adaptations of the model for forecasting time series, such as Yang et al. [2015]. In a time series setting, the structure allows the same features in the lags to be picked up regardless of the lag number at which they occur. This paper represents the first experimentation with CNNs for volatility forecasting.

The RNN has a dynamic structure, originally developed for handling variable length inputs, such as words in a sentence. RNNs now dominate the majority of natural language based tasks, such as translation. In this sense, they are more obviously suited to handling time series data. For language-based tasks, it is important to store/remember information from previous words or sentences. For example, if we are trying to predict the final word of a sentence, it is important for us to utilise information in the first few words. In a volatility forecasting exercise, the RNN is consequently well structured to make use of information in previous lags of realized volatility. Given this intuition, there has been wider experimentation with RNNs for time series forecasting. Xiong et al. [2015] is one example which uses a modern version of the RNN to forecast S&P 500 volatility. However, our work represents the first thorough application of modern variants of RNNs in a financial time series setting across a range of forecasting tasks and indices, and in particular to forecasting realized volatility.

The method for optimizing the weights of the neural network models we consider, based on the splitting of the data into separate train, validation and test segments, is well established

in the machine learning literature. However, there has been relatively little work establishing if this method is appropriate for forecasting of financial time series, and in particular volatility. In particular, the method requires an adequately large number of observations, and consistency of the data generating function over time. A final motivation of this paper is to investigate whether or not these conditions are satisfied in the case of realized volatility forecasting, and develop techniques through which we can apply the machine learning methodology successfully for this type of time series.

1.2 Models

The models which are compared in this paper include three variants of neural network, and three “baseline" models, which are chosen to provide a representation of the performance of the most commonly used models from the volatility forecasting literature within academic econometrics.

1.2.1 Baseline Models

Realized Volatility

The unobserved nature of conditional variance makes efficient comparison of competing models difficult. When evaluating models' performance, it is hard to identify if the difference in forecasting ability is due to the noisiness of the proxy, or an improvement in the model. Traditionally, squared returns were used as a proxy for true volatility. This resulted in poor out-of-sample performance of many ARCH-variants, leading to questions about the validity of the class of models as a whole. Andersen and Bollerslev [1998] addressed many of these concerns by using a proxy for volatility based on intra-day returns. They demonstrate that with the reduction in the noise of the proxy, ARCH-type models can indeed provide good out-of-sample forecasts. This result has been confirmed by other studies, including Hansen and Lunde [2005], in which the conclusion from an out-of-sample comparison of 330 different volatility models is that the best models do not provide a significantly better forecast than GARCH(1,1). This motivates the inclusion of the GARCH(1,1) model as one of our baselines for comparison.

Since Andersen and Bollerslev [1998] realized variance measures based on intra-day data have become the dominant proxy for the latent process. We use data from the Oxford-Man Institute of Quantitative Finance (Heber et al. [2009]), which provides a range of high-frequency based measures. The theory behind the measure used here, Realized Variance, was

established by Andersen et al. [2003] and Barndorff-Nielsen and Shephard [2002]. It is defined as:

$$RV_t = \sum x_{j,t}^2$$

where

$$x_{j,t} = X_{t_{j,t}} - X_{t_{j-1,t}}$$

and $t_{j,t}$ are the times of trades or quotes on the t_{th} day. If prices are observed without noise, then as $min_j|t_{j,t} - t_{j-1,t}| \to 0$ the measure consistently estimates the quadratic variation of the price on the t_{th} day.

Microstructure noise is a practical consideration, and the measure quoted in this paper is based on a subset of 5-minute returns data. As a subset is used, it is possible to average across many estimators each using different subsets, in line with Hansen and Lunde [2006]. This is done to the maximum degree possible in the measure of realized variance used here.

GARCH

The class of ARCH models, and particularly the generalized ARCH (GARCH) variant of Bollerslev [1986], have played a central part in volatility forecasting since the seminal paper of Engle [1982]. ARCH models, in their standard form, use the information in daily returns to estimate volatility. The GARCH (1,1) model is a common starting point for comparisons of models, in which the volatility process is defined as follows:

$$h_{t+1} = \gamma + \alpha r_t^2 + \beta h_t \tag{1.1}$$

where h_{t+1} is the forecast of next period's volatility, and r_t^2 is this period's squared returns. The model is symmetric, in that the sign of returns does not matter, and exhibits mean reversion if the restriction is imposed that $\alpha + \beta < 1$.

The observation that volatility tends to increase by a greater amount when past returns are negative has led to a number of non-symmetric adaptations of GARCH. In a comparative study of the forecasting performance of models within the ARCH class, Brownlees et al. [2011], the authors find that the Threshold ARCH (TARCH) model of Glosten et al. [1993] consistently provides the best out-of-sample forecasts of the whole range of ARCH models, and we therefore include it as our second comparison model. The volatility process is modelled as:

$$h_{t+1} = \gamma + (\alpha + \delta \mathbf{1}_{r_t<c}) r_t^2 + \beta h_t \tag{1.2}$$

where **1** is an indicator variable which takes a value of one when returns are less than the constant c, and zero otherwise. In this paper, we follow the standard approach of setting c equal to 0, which allows the model to react differently in periods of negative and positive returns.

Heterogeneous AR

Alongside the GARCH-variants based on returns, a number of models have been developed which use realized volatility measures directly to forecast future values. This started with Andersen et al. [2003], in which the authors proposed the use of auto-regressive fractionally integrated moving average (ARFIMA) models for capturing the highly persistent nature of realized volatility. Since that point, the approximate long-memory Heterogeneous AR (HAR) model of Corsi [2009] has, despite its simplicity, arguably become the dominant specification for financial volatility forecasting in a realized volatility setting. It is defined as:

$$RV_t = \beta_0 + \beta_1 RV_{t-1} + \beta_2 RV_{t-1|t-5} + \beta_3 RV_{t-1|t-22} + u_t \tag{1.3}$$

where

$$RV_{t-p|t-q} = \frac{1}{q+1-p} \sum_{i=p}^{q} RV_{t-i}$$

The choice of the average weekly lag and the average monthly lag is able to approximate the long-memory dynamic which is observed in the data.

For the GARCH, TARCH and HAR baselines used in this paper, all coefficients are estimated using maximum likelihood.

1.2.2 Neural Networks

Feedforward Neural Network (NN)

Feedforward neural networks are the most common form of network. They are defined by each node in each layer being connected to all other nodes in the same layer, and by the input to the network being of a fixed dimension. As a result, they are generally used with i.i.d. data, as the network is equally capable of attending to any input, regardless of its position in the input sequence. That said, this does not preclude feedforward networks from being used for time series data, as we experiment with in this paper; the input sequence must simply be

split into equally sized chunks. The popularity of the feedforward network across a wide range of applications stems from its flexibility; the number of layers can range in practice from one to many hundreds and the number of nodes in each layer from two or three to thousands. As a result, the network can be used to approximate very simple functions (with a small number of layers/nodes) or incredibly complex functions such as images. We discuss in more detail the mechanism for choosing an appropriate number of layers/nodes below.

The feedforward network is a non-linear function, which takes an input $x \in X$ (a vector or matrix of explanatory variables) and returns an output Y, which may be probability distribution over output classes or a real-valued vector. In this paper, the output of the network is a scalar, $y \in \mathbb{R}$, which will be our forecast of volatility for the next time step.

Given an input, x, the output, $h_{\theta,l}(x) \in \mathbb{R}^{d_l}$ of the l_{th} layer of the network is defined as

$$h_{\theta,l}(x) = g_l(W_l h_{\theta,l-1}(x) + b_l) \qquad l = 1, .., L$$

where $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ are referred to as the "weights" of the network and $b_l \in \mathbb{R}^{d_l}$ is referred to as the "bias". The input to layer one of the network is simply our vector of explanatory variables, $h_{\theta,0} = x$. The function $g^l(z)$ is a non-linear transformation, which is applied to all outputs of the layer individually, $g^l(z) = (\sigma(z_1), \sigma(z_2), ..., \sigma(z_{d_l}))$ with $z_1, z_2, ..., z_{d_l} \in \mathbb{R}$. The choice of σ is typically the sigmoid function, tanh or recitified linear units(ReLu).[2] The sigmoid function is used in this paper.

The final (output) layer of the network is simply a linear regression with or without[3] a non-linearity, which produces our estimate of volatility for the next period, $f_\theta(x) \in \mathbb{R}$:

$$f_\theta(x) = W_o h_{\theta,L}(x) + b_o$$

.

where $W_o \in \mathbb{R}^{d_L}$ and $b_o \in \mathbb{R}$.

The weights of the network, $\theta = (W_1, ..., W_L, b_1, ..., b_L, W_o, b_o)$ are learned through gradient descent.[4]

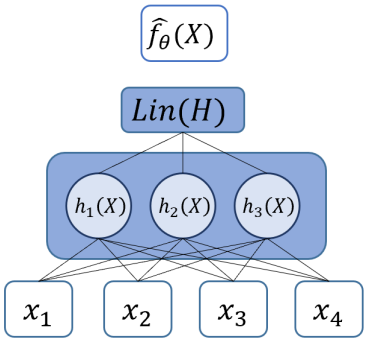
Diagrammatically, a single-layer NN can be represented as follows:

[2]The ReLU is defined as $f(x) = max(0, x)$ and is a popular choice of non-linearity as it avoids the gradient descent algorithm getting "stuck" in areas of the non-linear function with a very shallow gradient.

[3]We don't apply a non-linearity to the final layer in our work, so as not to limit the range of values the network can predict as the next timestep's volatility.

[4]See later section for explanation of the gradient descent algorithm.

Figure 1.1: NN with four inputs and one hidden layer of three nodes



$$h_1(X) = \sigma(b_1 + w_{(1,1)}x_1 + w_{(1,2)}x_2 + w_{(1,3)}x_3 + w_{(1,4)}x_4)$$
$$h_2(X) = \sigma(b_2 + w_{(2,1)}x_1 + w_{(2,2)}x_2 + w_{(2,3)}x_3 + w_{(2,4)}x_4)$$
$$h_3(X) = \sigma(b_3 + w_{(3,1)}x_1 + w_{(3,2)}x_2 + w_{(3,3)}x_3 + w_{(4,4)}x_4)$$

$$\hat{f}_\theta(X) = Lin(H) = b_o + w_{(o,1)}h_1(X) + w_{(o,2)}h_2(X) + w_{(o,3)}h_3(X)$$

In Figure 1.1, $d_1 = 3$ and $w_{1,2}$ refers to the second weight in the first node of the hidden layer. We have four weights and a bias term in each of the three hidden nodes, giving a total of fifteen weights in the hidden layer. We then have a further three weights in the output layer, with one bias term, giving a total of nineteen weights to be optimized. The complexity of the functions which can be approximated by this model increase as both the number of layers and the number of nodes in these layers grow.

Convolutional Neural Network (CNN)

Convolutional neural networks draw their name from applying the **same** set of weights (termed a “filter") across different areas of the input matrix (the filters “convolve" across the input matrix). This lies behind their success in image recognition; a filter which is tuned to be able to locate a cats head will be applied across the entirety of the input image, and as a result can identify the head regardless of its location in the image. Given image data is uncommon in the financial econometrics literature, the use of convolutional networks for the original task they were designed for is relatively rare, although Doering et al. [2017] provides one example, where the status of an order book is represented as an image, which is transformed by a CNN into a feature vector which can be input to a feedforward network.

That being said, there is nothing which precludes convolutional networks being used directly on time series, as demonstrated in Yang et al. [2015]. The intuition here is that there may be certain patterns of data in the time series which foreshadow a rise/fall in the series, but at an uncertain point in the future. In other words, a set pattern may be a useful feature in predicting a rise in the series, but we do not know whether this rise will occur at t+1, t+2 or t+5.

To define the network, consider an input vector $x \in \mathbb{R}^{d_x}$, which is formed of lags of one explanatory variable only. The outputs of filter j, $h_j(x) \in \mathbb{R}^{d_x - s_j + 1}$ (where s_j is the filter size of filter j) are defined as:

$$h_j(x) = W_j x_{(i:i+s_j-1)} + b_j$$

where $x_{1:3}$ denotes the first to third elements (inclusive) of the input vector x, $W_j \in \mathbb{R}^{s_j}$ and $b_j \in \mathbb{R}$. In words, the same weight vector, W_j, is multiplied by each set of adjacent s values in the input vector x. The outputs of each filter are then passed to a max-pooling layer, $g_j(x) \in \mathbb{R}$:

$$g_j(x) = max(h_j(x)) \qquad j = 1, ..., J$$

The output layer of the network $h_\theta(x) \in \mathbb{R}$ is simply of linear regression form, as with the feedforward NN:

$$f(x) = W_o g(x) + b_o$$

where $g(x) \in \mathbb{R}^J, W_o \in \mathbb{R}^J$ and $b_o \in \mathbb{R}$

In practice, we can incorporate multiple explanatory variables into the model, with a different set of filters applied to each explanatory variable.[5]. This simply increases the dimensions of the final linear regression, as does applying filters of different sizes to the same explanatory variable.

In Figure 1.2 $d_x = 4$ and $J = 2$, with one filter of size two and the other of size three. This results in two variables in the output linear regression. The key difference to the feedforward NN is that the same weights are applied to each section of the input vector. For example, $w_{1,1}$ appears in each of the first three equations.

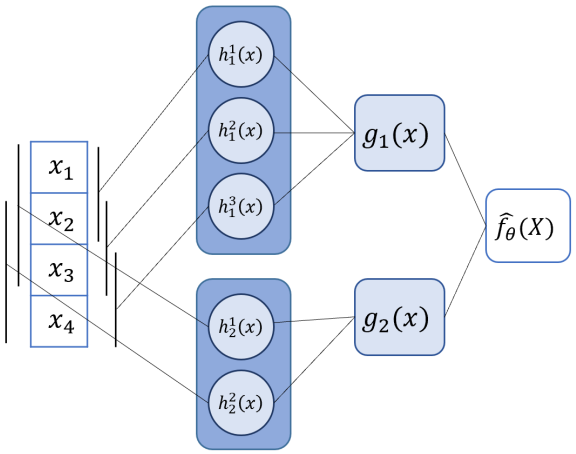
Recurrent Neural Network (RNN)

Recurrent neural networks differ from the previous two models in that they feature a dynamic feedback loop between the output of one layer and the input of the next. This gives RNNs two important features; firstly they can handle sequences of variable input length, and secondly they output at timestep t is able to depend on all previous inputs/timesteps (this does not imply that the model makes use of all previous timesteps when making a prediction, but rather is able to learn the appropriate number of previous steps to use during training) .[6] These two features mean recurrent neural networks have become particularly commonly used

[5] It is theoretically possible for the same filter to be applied across explanatory variables, although we choose not to include this variant here, as there is little theoretical justification that this would provide useful insight.

[6] In practice, training recurrent neural networks over many hundreds of timesteps results in some practical challenges, discussed below alongside the introduction of the LSTM architecture.

Figure 1.2: CNN with one explanatory variable and two filters



$$h_1^1(x) = b_1 + w_{1,1}x_1 + w_{1,2}x_2$$
$$h_1^2(x) = b_1 + w_{1,1}x_2 + w_{1,2}x_3$$
$$h_1^3(x) = b_1 + w_{1,1}x_3 + w_{1,2}x_4$$

$$h_2^1(x) = b_2 + w_{2,1}x_1 + w_{2,2}x_2 + w_{2,3}x_3$$
$$h_2^2(x) = b_2 + w_{2,1}x_2 + w_{2,2}x_3 + w_{2,3}x_4$$

$$g_1(x) = max(h_1^1(x), h_1^2(x), h_1^3(x))$$
$$g_2(x) = max(h_2^1(x), h_2^2(x))$$

$$f(x) = b_o + w_{o,1}g_1(x) + w_{o,2}g_2(x)$$

in the field of Natural Language Processing (NLP). Text or speech data requires an ability to handle variations in sequence length (paragraphs/sentences are composed of different numbers of words), and the meaning of a word often depends on the prior words/sentences of a document/recording, sometimes many words/sentences in the past. These two features also mean recurrent neural networks are a natural fit for time series data, with a growing body of literature within finance and econometrics, such as Bucci [2020] and Liu et al. [2018], applying them to forecasting problems.

To introduce the architecture of a recurrent neural network, consider a time series, $\{x_1, ..., x_t, ..., x_T\}$. In the basic RNN, the output of the network at time t, $h_t(x_{\hat{t}}) \in \mathbb{R}^{D_h}$ (where $x_{\hat{t}}$ refers to $(x_t, x_{t-1}, ..., x_1)$ and D_h is the dimension of the hidden layer of the network) can be described as:

$$h_t(x_{\hat{t}}) = g(W_h h_{t-1}(x_{\hat{(t-1)}}) + W_x x_t + b_h)$$

where $W_h \in \mathbb{R}^{D_h \times D_h}, W_x \in \mathbb{R}^{D_h \times d_x}$ and $b \in \mathbb{R}^{D_h}$. The weights matrices, W_h and W_x are not time dependent, with the same weights matrices being used at each timestep. The output $h_t(x_{\hat{t}})$ is termed the "hidden state" of the network, and can be thought of as holding information from previous timesteps. This information is then combined with the new input, x_t, to give our prediction at this timestep. The hidden state needs to be initialised at timestep zero, with the usual choice of a vector of zeros being used in this paper.

The remainder of the network is identical in format to the feedforward NN. The function $g(z)$ is a non-linear transformation, which is applied to all outputs of the layer individually, $g(z) = (\sigma(z_1), \sigma(z_2), ..., \sigma(z_{D_h}))$. The output layer is a linear regression without a non-linearity, which produces our estimate of volatility for the next period, $f_\theta(x) \in \mathbb{R}$:

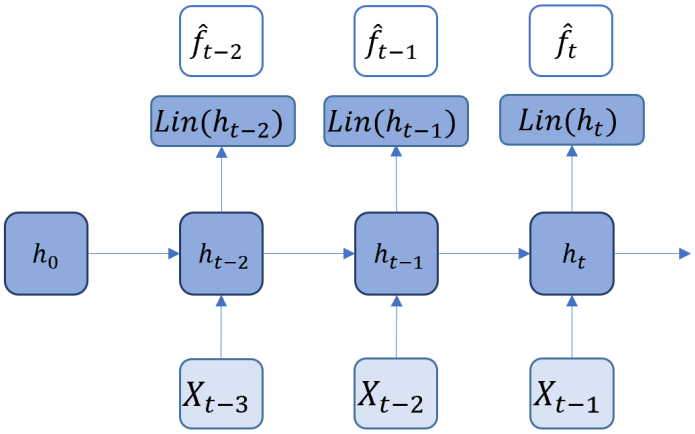
$$f(x_{\hat{t}}) = W_o h_t(x_{\hat{t}}) + b_o$$

.

where $W_o \in \mathbb{R}^{D_h}$ and $b_o \in \mathbb{R}$.

The RNN is best visualized "rolled out", although the process is in reality a continuous cycle of arbitrary length.

Figure 1.3: RNN rolled out over three timesteps



Theoretically, the RNN is able use information from every previous timestep in the current prediction, even as t grows very large. In practice, the use of gradient descent for learning the coefficient values combined with the structure of the network prohibits this. As we are using the same weights matrices at every timestep, the gradients of the cost function relative to the values of these matrices will be multiplied together many times. If the gradient is large in value this leads to exploding gradients, or if it is small to vanishing gradients. During the training phase, we therefore have to limit the number of previous timesteps which the algorithm can use information from, generally to ten or under.

In order to try and overcome this problem[7], a number of key adaptations to the RNN have been proposed which we make use of in this paper. The basic structure (where prediction is

[7]In natural language processing application, for which RNNs were developed, it is often useful to use information from words at the start of long sentences, or even in previous sentences/paragraphs. In this application, it is less of a problem, as lags greater than ten are unlikely to yield information that is particularly useful.

made using the hidden state from previous timesteps, and the new input at this timestep, x_t), remains the same, with the adaptations being made to the method by which we combine these two sources of information. Hochreiter and Schmidhuber [1997] proposed the long-short term memory (LSTM) cell. The idea is to keep some “memory" of inputs from time-steps in the past, and for the input at the current step to determine the extent to which this memory influences the current prediction. The equations which decide how much information of each source is used or passed on are termed “gates". In the LSTM, we define:

$$Input\,gate : i_t = \sigma(W_i x_t + U_i h_{t-1})$$
$$Forget\,gate : f_t = \sigma(W_f x_t + U_f h_{t-1})$$
$$Output\,gate : u_t = \sigma(W_c x_t + U_c h_{t-1})$$
$$New\,memory\,cell : \tilde{c}_t = tanh(W_c x_t + U_c h_{t-1})$$

$$Final\,memory\,cell : c_t = f_t c_{t-1} + i_t \tilde{c}_t$$
$$Final\,hidden\,state : h_t = u_t tanh(c_t)$$

where σ is a sigmoid function.

A more recent simplification of the above cell, developed for machine language translation, is the Gated Recurrent Unit (GRU) of Cho et al. [2014].

$$Update\,gate : u_t = \sigma(W_u x_t + U_u h_{t-1})$$
$$Reset\,gate : r_t = \sigma(W_r x_t + U_r h_{t-1})$$
$$New\,memory\,content : \tilde{h}_t = tanh(W_h x_t + r_t U_h h_{t-1})$$

$$Final\,hidden\,state : h_t = u_t h_{t-1} + (1 - u_t)\tilde{h}_t$$

The intuition behind the gate mechanism is clearer in the case of the GRU cell. If the reset gate is close to zero, then the model will drop past information from the memory cell which is not relevant to prediction at the current step or to any future prediction. The update gate, u_t, controls how much the memory state should matter for the current prediction only. This diminishes the exploding/vanishing gradient problem of the recurrent neural network, allowing training of the network with a greater number of previous timesteps (lags). If the

number of lags grows very large, it is also possible to approximate time-variant effects with an LSTM, provided their is a consistent signal in the training data that allows the network to identify timesteps at which these effects occur.

The LSTM and GRU cells significantly increase the complexity of the function which can be approximated by the RNN, and as such have both been incredibly successful in the field of natural language processing. This comes at the cost of an increase in the number of coefficient values which need to be learnt through gradient descent, and greater obscurity of the prediction mechanism.

Network size and the bias/variance tradeoff

One of the fundamental reasons for the popularity of all three neural network variants which we experiment with in this paper is the flexibility over their size, and hence over the how complex a function $f(x)$, they can approximate. The size of a network is generally measured by the number of learnable parameters (weights); this in turn depends on some key hyperparameter choices the researcher must make. For a feedforward neural network, these are the number of layers and the number of hidden nodes in each layer. For a convolutional neural network, we have the number of filters, the dimensions of each filter, and the number of layers. Finally, for a recurrent neural network, we have the cell architecture (LSTM, GRU etc.), the number of nodes within each cell, and the number of layers. In each case, different choices of these hyperparameters can result in a network with only a few weights, to one with millions, or even billions of weights, Brown et al. [2020].

As with many families of machine learning models, the bias-variance tradeoff determines the correct choice of network size. If the network size is too small, the network may be unable to approximate the true function $f(x)$, even if optimized perfectly, resulting in a high level of bias. However, if $f(x)$ is instead a relatively simple function, and the network size is large, then the model is likely to fit random noise ($\epsilon(x)$) in the training set, resulting in a high variance across samples.

In light of this, the researcher must make a decision over the optimum network size for their particular application. In practice, this can be thought of as being split into two steps. Firstly, the researcher will use their prior about the complexity of the underlying function $f(x)$ and the variance of $\epsilon(x)$, alongside the amount of training data available, to estimate a sensible range of network sizes within which experimentation can take place. This is necessary, as the total set of network sizes is infinite, and it is impractical to experiment with all options. The range of options we choose for the forecasting of realized volatility are discussed in Section 1.3.3 below. Secondly, within this range of options, it is good practice to perform a hyperparameter selection step, where a number of different network sizes/number of nodes

etc. are experimented with. The out-of-sample performance of each of these network sizes can be monitored on the validation dataset, hopefully giving an intuition as to the optimum bias-variance tradeoff for this problem. Importantly, the validation dataset should no longer be considered as true “out-of-sample" data once this process has been completed (especially if the number of different hyperparameter options experimented with is very large), and the final evaluation of the model should be on a separate test dataset, which has been held out until this point.

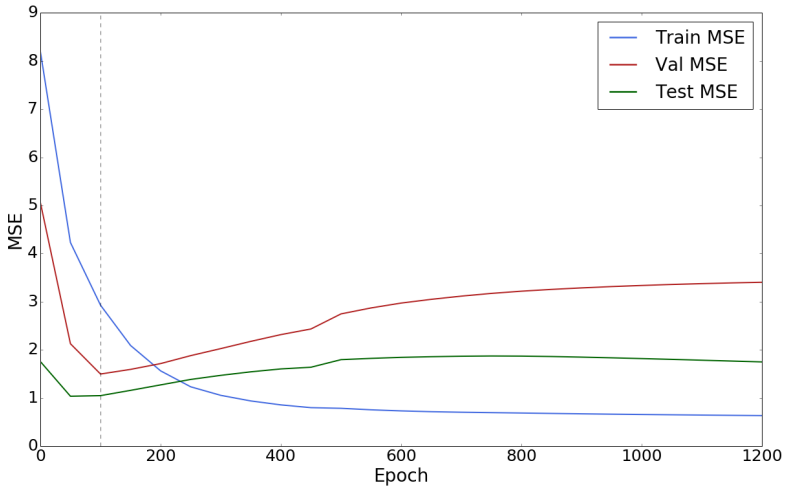
This pattern is followed in all chapters of this work, with a description of the relative sizes of the train, validation and test sets in Section 1.3 below, and of the hyperparameter options we experiment with in this paper in Section 1.3.3, as well as full details in the Appendices.

Gradient Descent

In all of the neural network models, the weights of the network are optimized using gradient descent. This is referred to as “training the model" in the machine learning literature. In theory, it would be possible to select coefficient values using maximum likelihood, although the compute time may be very large as you increase the dimensions of the weights. The problem with this approach is that the neural networks are able to approximate a vast array of complex non-linear functions. As a result, applying maximum likelihood would result in very low in-sample error, but likely very poor out-of-sample performance (“overfitting the training data"). Overfitting remains a potential problem when using gradient descent, especially if the algorithm initiates in a section of the loss function which is close to the global optimum. The most common practice for avoiding this, employed in this paper, is to split the data into a “train" set, a “validation" set and a “test" set. Gradient descent is performed on the train set, and the value of the loss function calculated for the validation set every hundred or so steps. We store the validation errors and network weights at each stage during training, and restore the weight values to those at which we achieved the best validation error once training is complete. There are a number of alternative methods to avoid overfitting, including dropout and regularisation. For simplicity, we limit the approach to early stopping in this paper, with experimentation with the other options left to future work.

This is demonstrated in Figure 1.4. Until epoch 100, the MSE of all three data sets is falling, with the model learning sensible weight values through gradient descent. Beyond epoch 100, the model begins to overfit the train data set, as seen in the continual reduction in the train MSE, but increase in both the test and val errors. If early-stopping were employed here, the final choice of network weights would be those where the val MSE achieves a minimum, demonstrated with the vertical dotted line.

Figure 1.4: Demonstration of overfitting, with a feedforward neural network predicting CAC 40 one-day-ahead volatility



Note: In the above graph, we can see that the test MSE actually begins to fall again from epoch 500 (epoch refers to the number of passes of the entire train dataset that have been completed) onwards, whereas the val MSE continues to rise. This highlights a potential problem with the use of validation data; if the data is not stationary, then a low error on the validation data may not correspond to a low error on the test data. This is discussed in more detail later.

The second advantage of a separate validation set is that we can compare models with different parameters (e.g. different number of hidden layers), and choose as our final model that which achieves the lowest validation error rate across all experiments. We can then evaluate the single chosen model on the test set, giving an out-of-sample forecast accuracy. Although we have not used the validation set during gradient descent, and therefore have not overfitted this set directly, if we are performing a large number of experiments we are at risk of finding a model which by chance performs particularly well on the validation set (or "data-snooping", as described in White [2000]), and therefore still need a completely separate test set for a true indication of how our model will perform out-of-sample.

This requires the choice of a loss function, $L_\theta(\hat{y}, y)$. The choice of loss functions for evaluating the out-of-sample performance of our models is discussed in more detail later. It is important to note that the loss function used to optimize the weights of the network does not have to be the same loss function through which we evaluate the performance of the network. For example, the results section in this paper includes the presentation of the mean squared error (MSE) out-of-sample loss for networks which have been trained using the quasi-likelihood (QL) loss.

In sudo-code, the basic gradient descent algorithm works as follows:

randomly initiate the weights of the model, θ

for epoch in number of epochs:

  for batch in number of batches:

    evaluate the gradient of the loss function $L_\theta(\hat{y}, y)$ with respect to the weights, ∇_θ

    update the weight values: $\theta_{new} = \theta_{old} - \alpha \nabla_\theta L_\theta(\hat{y}, y)$

restore weights to values at epoch with lowest validation error

where α is the "learning rate", and a "batch" refers to a segment of the training set, typically in the region of 50 to 100 observations. Many of the advances in machine learning over the last few years have been in the form of improved methods of gradient descent. There are a significant number of parameters which can be adjusted to optimize the algorithm, including the learning rate, the random initialization strategy and the loss function. An additional contribution of this paper is an exploration of the parameter values most suitable for applying the three neural networks to financial time series.

Validity of the cross validation method for forecasting time series

In order for the primary method of cross-validation used in machine learning applications (that of splitting the data into a train, validation and test set) to work, we require that the three separate data sets are generated by a consistent underlying function. In the time series setting, the train set is likely to consist of the earliest period of data, with the validation second, and the test set the most recent observations.[8] Consistency of the function generating the data is consequently reliant on stationarity of the series.

In line with previous studies, such as Engle and Patton [2007], we find that the realized variance data easily rejects the null of non-stationarity in unit root tests (we use the Augmented Dickey-Fuller test), despite the persistence of the process. This is a promising baseline result for the use of the validation method proposed. However, rejection of the null of a unit-root does not ensure that the underlying function which we hope to approximate is consistent across time in all dimensions. There is an increasing literature demonstrating the potential of models which employ time-variant coefficients to characterise the volatility process, since

[8]This is by no means required, and any ordering of the three sets, and indeed more complex combinations where the sets are split across time, is possible. However, given that in applications we are interested in forecasting future values, it makes sense to have the test set last, as this gives the best indication of the likely accuracy of these forecasts.

the early overview of Hastie and Tibshirani [1993]. In the approach we employ, the weights of the network are learnt using gradient descent on the train observations only, which occur a minimum of two years prior to the test data (the validation set sits in between) and a maximum of thirteen years prior. The model has no capacity to incorporate time-variant effects in this framework, weighting all of the train observations equally, regardless of position in the set. If such effects dominate the generative process, then the machine learning models are likely to produce poor predictions. One of the simplest methods of varying coefficients over time is simply to re-estimate the model at each timestep, using only recent data. Given that the HAR model does not require a validation data set, it is common to estimate the coefficients using the prior 1000 observations to the prediction step, which we conduct as an additional comparison to our models.

With realized volatility data, there are clear periods of persistently higher observations, with one obvious example being the financial crisis around 2009. It is likely in such periods that there may be temporal changes to the generative model, due to exogenous variables, beyond that which is captured by long memory type processes. There is also consistently higher variance of realized volatility itself during such periods. If the train, validation or test datasets is situated in such a period, there is a considerable risk that the machine learning methodology will produce bad forecasts. For example, if the validation data set was drawn from an era of financial turbulence, then the model which we choose based on its low error rate in this period will be unlikely to produce good out-of-sample forecasts during more stable periods.

Given that we are considering temporary changes to the model, or to the noise of the observations, the probability of any of the data sets being dominated by such periods clearly falls as we increase the size of each individual set. In a non-time series setting, it is clearly beneficial to have as large a dataset as possible for this reason, and to enable the models to pick up complex non-linear dynamics. With time series analysis using machine learning, this has to be balanced against the potential for changes in the generative model over time. The performance of the neural networks depends to a significant extent on balancing these factors, and a key contribution of this paper is an analysis of whether or not this is feasible in a realized volatility setting.

Moving Average Adaptations

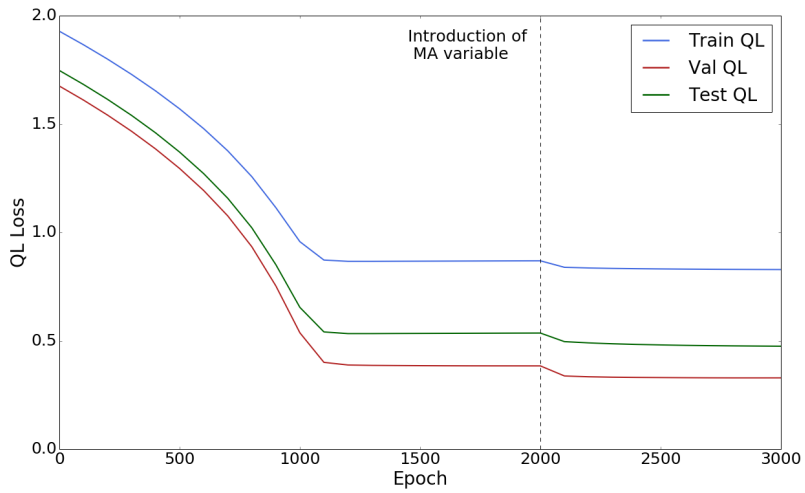
Research focused on improving the performance of neural networks is largely constrained to the fields of machine learning and artificial intelligence, and consequently to problems such as image recognition and natural language processing. Relatively, there has been very little work developing neural networks specifically for time series forecasting, despite a range

of papers evaluating their performance in this context. We present and experiment with a novel adaptation in which the residual from the neural network's prediction in the previous timestep is used as an input to the next steps prediction, in the style of a moving-average model. Models in which this adaption is included are followed by "MA" in the results section.

Evaluation of such models on the validation and test sets is straightforward, and follows in the usual fashion from evaluation of the basic models. During the "training"[9] phase, some changes to the gradient descent method need to be made.

During the initial stages of training, the values of the weights are updating relatively fast, resulting in the training loss falling quickly, as can be seen in the steep gradient up to around epoch 1200 in Figure 1.5. This, in turn, results in the values of the residuals changing frequently. As the residuals are now used as explanatory variables/features in the model, we are attempting to optimize a function which is continuously changing. This in the majority of cases results in the gradient descent method failing, with convergence not occurring. In order to overcome this problem, we propose constraining the gradients of the loss function with respect to the moving average weights to zero during the initial period of training. This allows the values of the other network weights, and hence those of the residuals, to stabilise. Once this has occurred, we allow the MA-gradients to be used by the algorithm.

Figure 1.5: Training of CNN-MA to predict one-day-ahead realized volatility of the Hang Seng



This is demonstrated in Figure 1.5. Initially the model is only allowed to use the lags of realized variance as explanatory variables. Once the coefficient values for these have settled close to an equilibrium, we begin to get a consistent value of the quasi-likelihood (QL) loss and residuals. At epoch 2000, the constraint on the MA-gradients is lifted, and the network

[9]In machine learning, "training" refers to tuning the weights of the model using gradient descent, or similar algorithms.

weights related to these are updated by the algorithm, resulting in a further reduction in the QL loss. In practice, we experiment with the lifting of the restriction at a set point in the training (epoch 2000, as in Figure 1.5), and with a more flexible approach where it is lifted once the rate of change of the validation loss has slowed sufficiently. The second approach would correspond to around epoch 1200 in Figure 1.5, though may be earlier or later for a different data/parameter set.

Secondly, in order to generate the correct residual from the network at the previous timestep, we have to reduce the batch size to one. Computationally, this slows the training by over an order of magnitude or more, making experimentation with such models infeasible. As a workaround, we continue with batches of the usual size during the training phase, and simply use as our estimates for the residuals in the current epoch the saved residuals from the previous epoch. As we change the network weights a number of times in each epoch, the residuals we save and use are not precisely correct. However, given that the magnitude with which we update the weights is very small in one epoch,[10] the values are close enough to their true values to facilitate training by this method.

1.3 Data and Empirical Methodology

1.3.1 Data

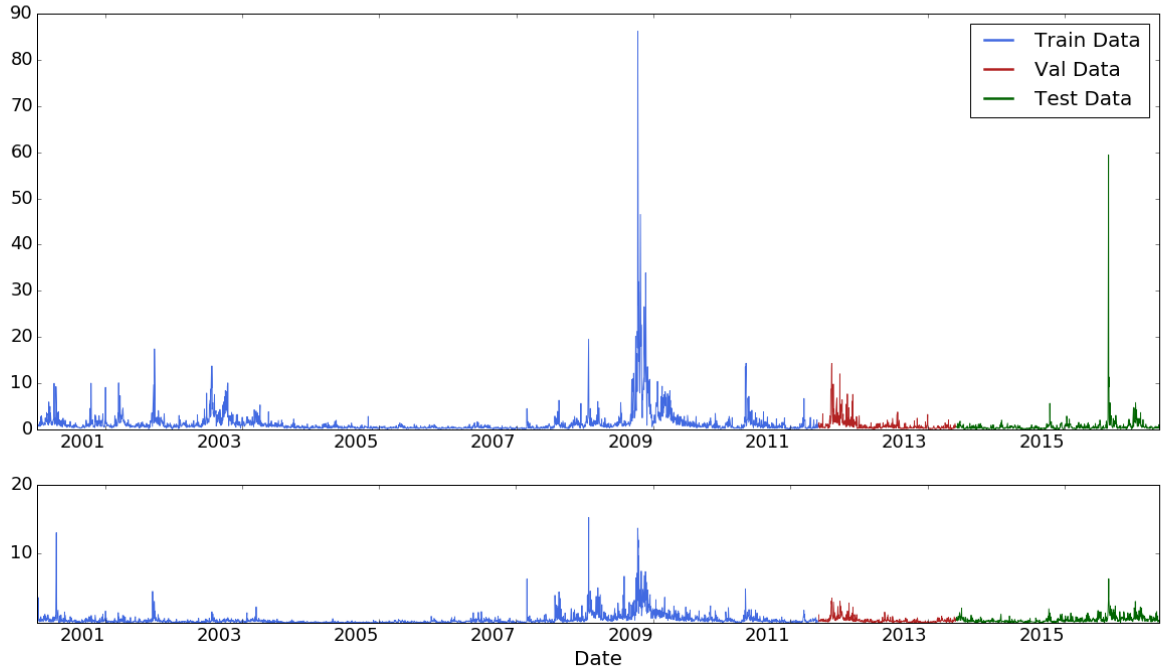
We conduct our empirical evaluation of the models using returns and realized variance data from ten major stock indices; the DJIA, Nasdaq, S&P 500, Russel 2000, CAC 40, FTSE 100, DAX, All Ordinaries, Hang Seng and Nikkei 225. Data is sourced from the Oxford-Man Institute Realized Library, Heber et al. [2009]. Our sample consists of approximately 4085 trading days per index, with slight variations due to differences in the number of days per year the indices are open for trading. It begins on the January 03, 2000, and finishes on May 20, 2016.

As discussed previously, we need to split the data into three sets for the neural networks. Our train set consists of approximately 2084 observations, running from January 03, 2000 to June 01, 2011. The validation set consists of around 502 observations per index, and runs from June 02, 2011 to 31 May, 2013. Finally, the test set is composed of around 749 observations, from June 03, 2013 to May 20, 2016.

The comparison models (GARCH, TARCH and HAR) are estimated using maximum likelihood, and therefore do not require a validation data set. For the majority of results listed

[10] This is ensured by the introduction of the MA features once the model has converged, as discussed above, and the fact that full training usually takes around a thousand to two thousand epochs at the learning rates we employ.

Figure 1.6: DJIA (top) and All Ordinaries (bottom) Realized Variance



Note: The figure illustrates the realized variance data which we use as our proxy for true volatility, for the DJIA and All Ordinaries indices. Both indices experience periods of high volatility, most notably around the financial crisis in 2009, although the scale of the volatility is far higher for the DJIA. The DJIA also experiences periods of volatility in the early 2000s, some of which don't appear for the All Ordinaries index.

in this paper, the estimation is carried out using only the train set, but results reported for all three sets. The validation and test sets can both be regarded as out-of-sample forecasts. The maximimum likelihood method of estimation holds a potential advantage over gradient descent, in that we can use data closer to the out-of-sample test set for our estimation. Brownlees et al. [2011] find little evidence to suggest that using an estimation window immediately before our test window improves results. We check this result, including an estimation of the HAR model using a rolling window of 1000 observations,[11] and reach a similar conclusion. The out-of-sample estimates are included in the primary results table, but not beyond this point.

1.3.2 Loss functions

There exist a large number of loss functions which have been used for comparing out-of-sample forecasts in the volatility setting. Patton [2011] identifies two loss functions which are robust

[11] In other words, for each individual observation in the test set, we compute our coefficient estimates by maximum likelihood using the previous thousand days of data. We consequently report only the validation and test errors, under the name, "HAR (Rolling)".

in the sense of asymptotically generating the same rankings of different models regardless of the proxy being used. This holds under a few minimal regularity conditions, and as long as the proxy is unbiased, as is the case for realized variance, used in this paper. The two functions are the mean squared error (MSE), based on the squared errors (SE):

$$SE(RV_t, f_t) = (RV_t - f_t)^2$$

and the quasi-likelihood (QL) loss,

$$QL(RV_t, f_t) = log(f_t) + \frac{RV_t}{f_t}$$

where f_t is our forecast for time t, and RV_t is the realized variance proxy for true volatility. Following Brownlees et al. [2011] we choose QL loss as our primary function for evaluation. As MSE depends on additive errors, as opposed to multiplicative errors in the case of QL loss, it becomes dominated by a few observations from periods of extremely high volatility. The QL loss avoids this, as well as being iid under the null that the forecasting model is correctly specified, as described in Brownlees et al. [2011].

The choice of loss function plays an additional key role in the training of machine learning models, as we are calculating the derivatives of the network weights with respect to the loss function during gradient descent. It is important to note, as mentioned previously, that the loss function which we use for carrying out gradient descent does not have to be the same loss function used to evaluate performance. Our empirical investigation includes out-of-sample MSE results for models trained using QL. The investigation of which loss function is most suitable for training neural networks in a financial time series setting is a supplementary contribution of the work.

1.3.3 Empirical Methodology

The focus of the empirical work in this paper is out-of-sample forecasting, and we aim to provide a comprehensive comparison of the model's performances across a range of forecasting scenarios. An analysis of in-sample performance, as carried out in much of the volatility forecasting literature, is not included. This follows from the earlier discussion about the neural network's potential to overfit the training data set, which renders such an analysis uninteresting.

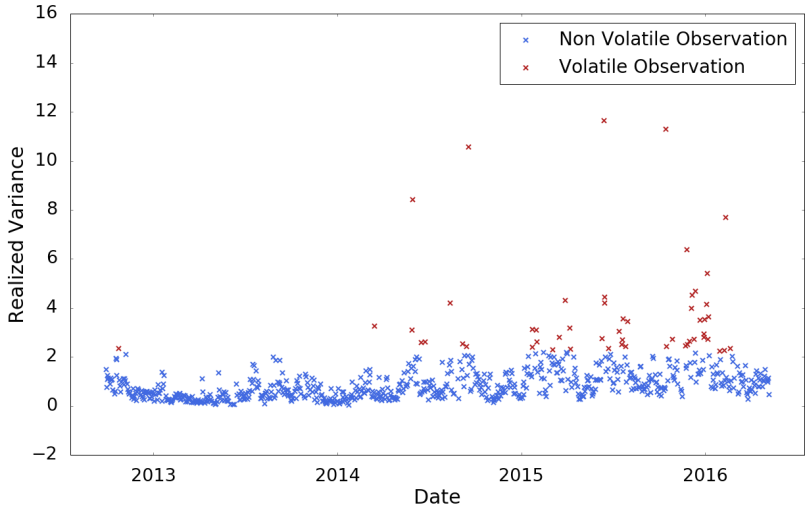
All the neural network models are built using Google's open source machine learning software library, Tensorflow.[12]

[12]https://www.tensorflow.org/

Forecasting Scenarios

The primary results focus on one-day-ahead forecasting (in line with the existing literature), and use the QL loss function for both the training of the networks and the analysis of their performance. We then investigate the potential to use the MSE loss function to train the networks, to provide insight into the differences in performance from employing gradient descent with an additive as opposed to multiplicative function. Next, we compare the out-of-sample performance in volatile and non-volatile periods, an application of particular interest to practitioners in risk analytics. We define a volatile observation as one which is over one standard deviation greater than the mean.[13] This yields a split of volatile/non-volatile observations as shown in Figure 1.7.

Figure 1.7: DAX realized variance test data (left) and number of volatile and non-volatile observations for each index (right)



	Vol.	Non Vol.
DJIA	12	736
Nas	44	704
S&P	23	725
Russ	76	671
CAC	56	701
FTSE	47	703
DAX	54	694
AllO	73	659
HanS	40	690
Nikk	38	682

Note: The volatile/non-volatile split illustrated is based on observations being larger/smaller than the mean plus one standard deviation

Finally, we look at two-, five- and thirty-day-ahead forecasts. There are two methods for multi-step-ahead prediction which are common in the literature. Firstly, rolling prediction, where the forecast for two days ahead is computed by first evaluating the one-day-ahead forecast, and then treating this value as an observation with which to predict a further day ahead. Alternatively, direct prediction would use the q_{th} lag or greater for a q-step-ahead prediction when estimating the coefficient values. The prediction for q-steps-ahead can then be provided directly by the model. The advantage of the latter approach is that it is simple

[13]Results in which we class a volatile observation as 0.5 or 1.5 standard deviations greater than the mean are included in the appendix

to include additional explanatory variables in the model,[14] which is likely to be common in almost all practical applications of volatility forecasting models. As such, we take the direct approach in this paper. The disadvantage of this choice is that it entails significant extra computation, as a new set of coefficient values has to be estimated for each forecast horizon. Given that we do not observe the true realized variance until multiple timesteps ahead, the moving-average adaptations are not compatible with either forecasting approach, so we exclude them for this section. As a baseline for comparison, we use a rolling prediction version of the HAR model.[15]

For each forecasting scenario we experiment with the three baseline models and four variants of each of the neural network types, giving twelve neural network models in total. For each network type, we begin with a model which uses only lags of realized variance as explanatory variables. These models are denoted “NN", “CNN" and “RNN" in the results tables. We then add lags of returns to the explanatory variables vector. Although the scale of returns will be correlated with the realized variance data, the sign adds new information, potentially allowing different predictions following negative and positive returns. Such models are denoted “NN (Ret)", “CNN (Ret)" and “RNN (Ret)" in the results. We then introduce the moving-average variant of each model. The basic moving average models are denoted “NN-MA", “CNN-MA" and “RNN-MA", and use lags of realized variance and the errors from previous timesteps prediction as explanatory variables. These models are trained with the approach described at the end of Section 2.2. Finally, we add lags of returns to the moving-average models, with the resulting networks denoted “NN-MA (Ret)", “CNN-MA (Ret)" and “RNN-MA (Ret)". Table 1.1 summarizes the explanatory variables used in each variant.

[14] With a rolling forecast, we would also need to provide a forecast for the explanatory variables up to one day before the forecast horizon. A random walk model for the explanatory variables is often used in practice.

[15] Rolling prediction is the standard baseline method in the literature, as direct prediction is not compatible with ARCH based models, which have a moving-average type component. As the HAR model provides more accurate one-step-ahead forecasts than the ARCH-based models, it is extremely likely to provide more accurate rolling forecasts, so we limit the comparison to the one model to keep the results succinct.

Table 1.1: Key of explanatory variables included in each neural network model variant

	Explanatory Variables
NN	Realized Variance
NN (Ret)	Realized Variance, Returns
NN-MA	Realized Variance, Residuals
NN-MA (Ret)	Realized Variance, Residuals, Returns
CNN	Realized Variance
CNN (Ret)	Realized Variance, Returns
CNN-MA	Realized Variance, Residuals
CNN-MA (Ret)	Realized Variance, Residuals, Returns
RNN	Realized Variance
RNN (Ret)	Realized Variance, Returns
RNN-MA	Realized Variance, Residuals
RNN-MA (Ret)	Realized Variance, Residuals, Returns

Estimation Strategy

The coefficients of the baseline models (GARCH, TARCH and HAR) are estimated using maximum likelihood on the train data. The only exception to this is the rolling HAR estimation, for which a new set of coefficients are estimated by maximum likelihood for each prediction, from the previous 1000 observations.

The neural network models are estimated using gradient descent. This requires the network weights to be randomly initialised, which immediately adds variance to the performance. If the algorithm begins in a disadvantageous section of the function to be optimized, convergence may be very slow or towards a local minimum which is significantly different from the global minimum. This is a common occurrence in practice with the training of neural networks for financial time series, given the large amounts of noise in the data. In order to minimize the risk of this, two approaches are taken.

Firstly, if the network fails to converge at all or converges very slowly during the first few hundred epochs, we restart the training with a new random initialization of the weights. Secondly, we experiment with a variety of parameters for the initialization and gradient descent method which affect the speed and quality of convergence. These include the learning rate, the random initialization function, and the initialization function's standard deviation or range. Given that it is possible to employ a different initialization strategy for different elements/layers of the network, the parameter space is significant.

There also exist a large number of adaptations of the gradient descent optimization algorithm which have emerged from the machine learning literature, including Adagrad, the Adam optimizer and Momentum. We carried out initial experimentation with the three

algorithms listed here, but found the gains in performance/speed to be inconsistent with the volatility dataset. A more thorough investigation of the relative advantages/disadvantages of these algorithms when forecasting financial time series is left to further work.

Aside from the initialization strategy, each of the neural network variants has a number of inidividual size/structure parameters which can be adjusted. Common across the three are the number of dimensions of the input vector (the number of realized variance lags, returns lags, and additional explanatory variable lags) and the number of layers in the network. For the feedforward NN, we can also change the number of nodes in each layer. For the CNN, we can adjust the number of filters for each explanatory variable, as well as the size of these filters. Finally, for the RNN, we can adjust the type of hidden cell (Basic RNN, LSTM or GRU), as well as the size of these cells, and the number of steps we use for the training of the network.[16] When the size/structure options are combined with the initialization options, the dimensions of the parameter space become very large. We balance exploration of this space with compute time, and aim to provide an experimentation strategy which gives a realistic idea of how these networks would perform in practical applications. Given that our main focus is a comparison of one-day-ahead forecasts, we explore a larger parameter space for these models, and a comparitively smaller number of options for the multi-step-ahead forecasts. Table 1.2 lists the total number of different parameter options explored for each forecasting scenario.[17]

Table 1.2: Total number of different parameter sets used per forecasting application

	QL one-day-ahead	MSE one-day-ahead	QL multi-step-ahead
NN	300	296	48
NN (Ret)	300	296	64
NN-MA	300	296	—
NN-MA (Ret)	300	296	—
CNN	440	320	54
CNN (Ret)	440	320	54
CNN-MA	440	640	—
CNN-MA (Ret)	440	640	—
RNN	144	144	54
RNN (Ret)	144	144	54
RNN-MA	144	144	—
RNN-MA (Ret)	144	144	—

[16] The ability of the RNNs to make use of information from many timesteps previous has to be balanced with the potential for exploding or vanishing gradients

[17] There are a large number of parameter options for which the gradient descent algorithm will never begin to converge. Examples include a very high standard deviation of the function used to randomly initiate the weights, or a learning rate which is far too high or low. Preliminary experimentation on the DJIA was conducted to find sensible ranges for these key parameters. The main study then explores the parameter space within these ranges, and for all the indices.

1.4 Results and Discussion

1.4.1 One-day-ahead forecasting using QL Loss

Baseline models

We begin the discussion with an overview of the performance of the baseline models in forecasting one-day-ahead realized volatility. Table 1.3 displays the out-of-sample QL loss for all of the comparison models when making one-day-ahead predictions. The performance of the baseline models is consistent with previous studies, including Corsi [2009] and Brownlees et al. [2011]. TARCH outperforms GARCH across nine out of the ten indices, demonstrating the well-documented tendency of volatility to react asymetrically to returns. HAR, which uses the realized variance data directly, provides lower forecast errors to TARCH in eight cases, with an equal error in the case of the DJIA. This supports the establishment of Corsi's HAR model as one of the central models in the realized variance based volatility forecasting literature.

Neural Network Models

Overall, the performance of the neural networks is strong, with the vast majority of the variants of each of the three network types outperforming the baseline models. For each of the ten indices, the best out-of-sample error is provided by one of the network models. The RNN consistently provides the best forecasts of the three network variants, followed by the feedforward NN. The CNN provides broadly similar errors to the feedforward NN, though on average performs slightly worse.

Restricting our attention to the basic versions of the network models which use only lags of realized variance for prediction (denoted "NN", "CNN" and "RNN" in the results table), we can see that they outperform the baseline models across the majority of the indices. This is an interesting result, as it suggests there is information in the recent lags of realized variance which is useful for forecasting purposes beyond that captured by the HAR structure, and which the networks are effective in picking up.

The networks are also effective in utilising information in the returns data. Across all of the three network variants, the inclusion of lags of returns in the explanatory variables vector (denoted "(Ret)")) reduces the forecasting errors consistently, and in many cases by a sizeable margin. It is likely that this is the result of the networks capturing the asymmetry of volatility's reaction to the sign of returns.

Table 1.3: Out-of-sample (test) QL loss for one-day-ahead predictions

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
GARCH	0.353	0.287	0.248	0.483	0.825	0.016	0.899	0.220	0.478	0.753
TARCH	0.315	0.262	0.204	0.488	0.814	0.003	0.896	0.201	0.474	0.720
HAR	0.315	0.197	0.176	0.206	0.809	0.019	0.891	0.164	0.454	0.697
NN	0.303	0.164	0.162	0.192	0.810	-0.010	0.884	0.168	0.455	0.702
NN (Ret)	0.275	0.153	0.143	0.186	0.799	-0.014	0.872	0.160	0.456	0.707
NN-MA	0.287	0.163	0.148	0.198	0.808	-0.012	0.894	0.164	0.455	0.702
NN-MA (Ret)	0.258	0.160	0.136	0.190	0.799	-0.019	0.871	0.156	0.455	0.752
CNN	0.311	0.175	0.160	0.212	0.815	-0.009	0.888	0.177	0.457	0.712
CNN (Ret)	0.288	0.163	0.160	0.198	0.808	-0.013	0.880	0.162	0.462	0.730
CNN-MA	0.303	0.171	0.167	0.205	0.814	-0.013	0.880	0.175	0.458	0.716
CNN-MA (Ret)	0.286	0.160	0.152	0.199	0.807	-0.016	**0.869**	0.162	0.461	0.706
RNN	0.301	0.157	0.155	0.196	0.813	-0.012	0.883	0.165	0.449	0.795
RNN (Ret)	**0.245**	0.152	0.106	0.186	0.797	-0.018	0.870	0.159	0.455	0.698
RNN-MA	0.285	0.163	0.157	0.203	0.806	-0.014	0.881	0.164	0.454	0.704
RNN-MA (Ret)	0.263	**0.147**	**0.105**	**0.181**	**0.781**	**-0.026**	0.870	**0.155**	**0.451**	**0.685**

Note: The results reported in the table are for neural networks which have been trained using the QL loss. That is, it is the derivative of the QL loss function with respect to the weights which is used to update the weights in the gradient descent algorithm.

The moving average models also consistently outperform the basic network variants. A moving-average model can react very quickly to changes in exogenous variables which temporarily shift an index to a higher or lower level, and we suggest that it is this increased flexibility that explains the improved performance.

Overall the “RNN-MA (Ret)" model is the strongest variant considered, producing the lowest or equal lowest out-of-sample error for eight of the ten indices.

As discussed, in order for the three-data-set methodology[18] to work in a time series setting we need the train, validation and test datasets to be generated by a consistent process, and to be representative of each other. This ensures that when we fit the process by gradient descent in the training phase a lower train error will translate to a lower validation error, until the point at which we begin to overfit the train data. Consistency across the validation and test datasets meanwhile ensures that when we pick the best model based on the lowest validation error, the model provides a good out-of-sample fit to the test data.

[18]In which we split the data into a train set, used to perform gradient descent, a validation set, for early-stopping and model selection, and a test set for out-of-sample evaluation.

Table 1.4: Average and standard deviation of the realized variance data in the train, validation and test sets

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
Average Train RV	1.40	1.68	1.43	1.32	1.60	1.06	2.09	0.49	1.11	1.27
Average Val RV	0.98	0.74	1.11	1.12	1.60	0.70	1.69	0.42	0.55	0.67
Average Test RV	0.66	0.54	0.58	0.52	0.96	0.43	1.05	0.50	0.65	0.97
Stdev Train RV	3.09	2.71	2.96	2.44	2.67	1.94	3.52	0.93	1.85	1.85
Stdev Val RV	1.57	1.17	1.77	2.02	2.06	1.11	2.34	0.45	0.60	1.15
Stdev Test RV	2.30	0.93	1.52	0.55	1.12	0.50	1.18	0.50	0.81	1.47

It is clear from Table 1.4 and Figure 1.6 that, at a minimum, the scale and variance of volatility are not consistent across our three datasets for the majority of the indices. The train set contains the period of turbulence surrounding the financial crisis, resulting in a higher average realized variance across nine of the ten indices, and also higher variance of volatility itself. The years spanning 2011 to 2013 from which the validation set is drawn also experienced financial turbulence, albeit on a smaller scale, with uncertainty in Europe from the aftermath of the crisis playing out.

Table 1.5: Train, validation and test QL losses for one-step-ahead prediction

		DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
GARCH	Tr	0.887	1.142	0.915	1.130	1.081	0.618	1.299	-0.032	0.855	1.016
	V	0.674	0.487	0.793	0.891	1.234	0.350	1.202	0.031	0.360	0.480
	Te	0.353	0.287	0.248	0.483	0.825	0.016	0.899	0.220	0.478	0.753
TARCH	Tr	0.870	1.129	0.896	1.109	1.062	0.609	1.276	-0.047	0.856	1.004
	V	0.631	0.484	0.744	0.853	1.203	0.354	1.194	0.024	0.371	0.483
	Te	0.315	0.262	0.204	0.488	0.814	0.003	0.896	0.201	0.474	0.720
HAR	Tr	0.854	1.041	0.865	0.918	1.049	0.596	1.260	-0.101	0.817	0.965
	V	0.649	0.408	0.768	0.684	1.170	0.311	1.156	-0.045	0.320	0.461
	Te	0.315	0.197	0.176	0.206	0.809	0.019	0.891	0.164	0.454	0.697
RNN	Tr	0.900	1.067	0.927	0.939	1.061	0.613	1.292	-0.062	0.846	1.108
	V	0.643	0.384	0.767	0.672	1.165	0.286	1.157	-0.051	0.305	0.382
	Te	0.301	0.157	0.155	0.196	0.813	-0.012	0.883	0.165	0.449	0.795
RNN (Ret)	Tr	0.867	1.096	0.864	0.924	1.051	0.594	1.285	-0.096	0.837	1.031
	V	0.616	0.377	0.740	0.657	1.159	0.280	1.148	-0.059	0.306	0.447
	Te	0.245	0.152	0.106	0.186	0.797	-0.018	0.870	0.159	0.455	0.698
RNN-MA	Tr	0.862	1.039	0.905	0.916	1.050	0.576	1.260	-0.089	0.820	0.965
	V	0.640	0.383	0.763	0.668	1.165	0.284	1.149	-0.050	0.306	0.460
	Te	0.285	0.163	0.157	0.203	0.806	-0.014	0.881	0.164	0.454	0.704
RNN-MA (Ret)	Tr	0.837	1.027	0.847	0.903	1.039	0.570	1.250	-0.122	0.834	0.962
	V	0.613	0.376	0.737	0.655	1.160	0.278	1.143	-0.057	0.303	0.454
	Te	0.263	0.147	0.105	0.181	0.781	-0.026	0.870	0.155	0.451	0.685

Note: “Tr" refers to the training data set, “V" to the validation set, and “Te" to the test (out-of-sample) set. In the case of the baseline models, both the validation and test sets can be regarded as out-of-sample, as maximum-likelihood estimation is based only on the train data.

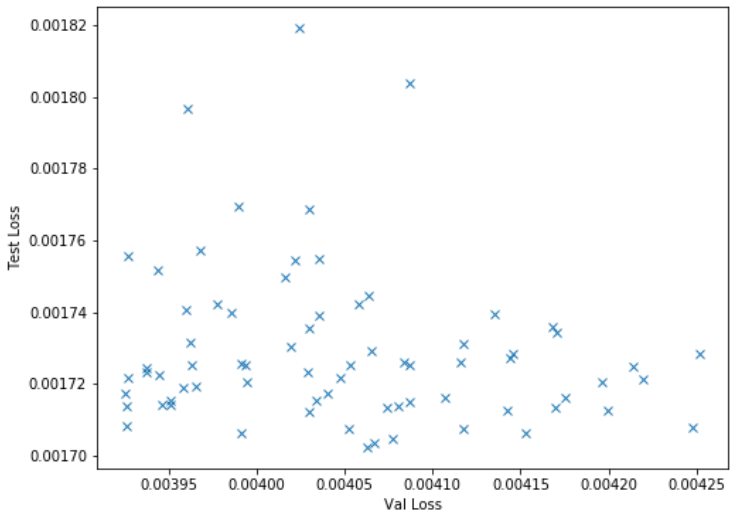
A potential method of mitigating such differences is the choice of the loss function used to train the network. The losses against which we calculate our gradients from an additive function such as the MSE will be dominated by volatile observations. If the training data set is particularly volatile overall, the model is likely to learn weight values which err on the side of over-prediction, to mitigate the effects of large observations. The QL loss function, on the other hand, is a multiplicative loss function, and also places a penalty on prediction of higher values, through the natural log function. As such, the model is likely to learn weights which produce accurate forecasts in periods of low volatility, and consequently transfer well across the three data sets.

To gain an intuition of this effect, we present the errors for our baseline models and the RNN variants on each of the three data sets in Table 1.5. The full table, including the NN and CNN results is included in the appendix, as Table A.1.

It is clear from an overview of Table 1.5 that the three-data-set method used in conjunction with the QL loss function is effective in mitigating the differences between the datasets. For any particular index, models which produce a low train QL are likely to also produce a low validation QL, which in turn translates to a better out-of-sample (test) error. This consistency of the relative errors across the datasets is a crucial observation, as it validates the use of the machine learning methodology in a volatility forecasting framework.

As discussed, this consistency is not guaranteed, and relies on there being sufficient predictability in the time series as well as stationarity. To emphasize this point, we also tried optimizing neural networks for prediction of the equity premium on the famous Goyal/Welch stock returns dataset, Welch and Goyal [2007]. The literature is inconclusive on the predictability in this dataset, but recent papers have suggested neural networks are able to outperform previous approaches, Feng et al. [2018].

Figure 1.8: Correlation between neural network validation and test set error for equity premium forecasts using the Goyal/Welch dataset



Although we do not present a full analysis of our prediction results in this thesis, we broadly concluded that neural networks are not suitable for this dataset, due to the lack of consistency of train, validation and test set errors. This is shown in Figure 1.8, which presents a scatterplot of the validation and test losses for a variety of feedforward neural networks with different hyperparameters. There is a clear lack of positive correlation between the two losses, indicating that models chosen using validation loss, as necessary for overfitting, will not necessarily produce accurate test set predictions of the equity premium.

1.4.2 One-day-ahead prediction using the MSE

To further explore the significance of the choice of loss function used for gradient descent, we experiment with training (and evaluating) models using the MSE.[19] The results are displayed in Table 1.6.

The first observation of note is that the HAR model loses its dominance over GARCH and TARCH, outperforming both for only five of the ten indices. TARCH meanwhile retains its relative advantage over GARCH. Given that the MSE is potentially dominated by volatile observations, this suggests that the TARCH model's ability to increase the scale of forecasts in periods of negative returns is playing an important role.

For the neural networks, the ordering of performance remains consistent, with the RNN on average providing the lowest out-of-sample errors, and the CNN the highest. The addition of the returns data continues to provide more accurate forecasts, as do the MA adaptations, although in the latter case the relative performance is very volatile. However, the consistency of the RNN's advantage over the other networks and the baseline models is significantly reduced. For example, in the case of the DJIA and AllO, the NN (Ret) model produces the best forecast, whereas TARCH is successful for the CAC.

To investigate the lack of consistency in the networks results, it is useful to analyse the errors achieved on the train and validation data sets. This is shown in Table 1.7 for the CNN,

[19] That is, during gradient descent, we are calculating the gradient of the MSE with respect to each individual weight when updating the weights. We also evaluate the out-of-sample performance using the MSE. Other things equal, we would expect that a model which has been trained using a particular loss function to perform well when evaluated using that same loss function, as the weights have been adjusted in the direction that minimizes the function.

Table 1.6: Out-of-sample (test) MSE for one-day-ahead predictions

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
GARCH	4.970	0.790	2.090	0.560	1.010	0.200	1.130	0.230	0.590	1.850
TARCH	4.840	0.740	1.980	0.620	**0.970**	0.190	1.130	0.200	0.550	1.620
HAR	5.290	0.710	2.110	0.180	1.000	**0.190**	1.160	0.160	0.490	1.700
NN	4.887	0.681	1.975	0.214	1.201	0.242	1.578	0.166	0.553	2.392
NN (Ret)	**4.495**	0.645	1.855	0.172	1.169	0.208	1.271	**0.141**	0.644	2.071
NN-MA	4.800	0.680	2.138	0.203	1.130	0.224	1.388	0.162	0.527	2.529
NN-MA (Ret)	4.748	0.675	**1.787**	0.164	1.050	0.205	1.246	0.146	0.583	1.944
CNN	4.818	0.668	2.189	0.206	1.156	0.209	1.274	0.160	0.584	2.500
CNN (Ret)	5.435	0.659	1.907	0.261	1.030	0.201	1.247	0.155	0.549	2.501
CNN-MA	4.939	0.658	2.200	0.201	1.067	0.203	1.267	0.159	0.523	2.374
CNN-MA (Ret)	5.021	0.643	2.007	0.273	1.073	0.213	1.461	0.160	0.615	1.939
RNN	4.826	0.738	1.978	0.185	1.138	0.204	1.393	0.161	0.507	1.624
RNN (Ret)	4.803	**0.642**	1.920	0.177	1.217	0.194	1.135	0.157	0.459	1.566
RNN-MA	4.828	0.658	2.003	0.227	1.020	0.206	1.492	0.223	0.518	2.095
RNN-MA (Ret)	4.658	0.669	1.951	**0.159**	1.133	0.195	**1.060**	0.153	**0.441**	**1.487**

with the full results for all models in Table A.2. Focusing on the DJIA results highlighted in blue, we note that the CNN achieves a higher train and validation error than the CNN-MA (Ret). However, the latter model produces a higher out-of-sample forecast (which is also significantly higher than the baseline models). The same can be said of the values highlighted in red for the DAX, and of many other comparisons across the table.

There are also inconsistencies between the train and validation errors. The values highlighted in green are an example of this, with the CNN-MA (Ret) model achieving a lower training error than the CNN-MA model, but a higher validation error.

Table 1.7: Train, validation and test MSE for one-step-ahead prediction (CNN only)

		DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
CNN	Tr	4.379	3.375	3.580	2.589	3.188	2.136	5.576	0.487	3.027	1.554
	V	1.373	0.518	1.599	1.765	1.576	0.525	1.858	0.120	0.298	0.796
	Te	4.818	0.668	2.189	0.206	1.156	0.209	1.274	0.160	0.584	2.500
CNN (Ret)	Tr	3.134	2.720	3.037	2.144	3.084	2.119	4.522	0.413	1.980	1.508
	V	1.087	0.417	1.313	1.363	1.496	0.471	1.632	0.114	0.352	0.872
	Te	5.435	0.659	1.907	0.261	1.030	0.201	1.247	0.155	0.549	2.501
CNN-MA	Tr	4.283	3.377	3.532	2.540	3.292	2.216	5.571	0.499	1.929	1.530
	V	1.347	0.516	1.579	1.766	1.595	0.520	1.857	0.119	0.295	0.792
	Te	4.939	0.658	2.200	0.201	1.067	0.203	1.267	0.159	0.523	2.374
CNN-MA (Ret)	Tr	3.133	2.729	2.991	2.610	3.000	2.130	4.657	0.436	1.409	1.248
	V	1.124	0.410	1.275	1.345	1.458	0.463	1.681	0.113	0.315	0.866
	Te	5.021	0.643	2.007	0.273	1.073	0.213	1.461	0.160	0.615	1.939

As discussed, inconsistencies such as those highlighted may result from structural changes to the underlying process generating the time series, or temporary periods of financial turbulence dominating a particular data set. Regardless of the cause, it is detrimental to the ability of the machine learning methodology employed here to produce accurate out-of-sample forecasts. To see this, note that in applications it is likely that the analyst would train a variety of networks, and choose between them based on the validation error, as the test error would of course not be available. Employing this strategy with the CNN models above, we would choose the CNN (Ret) model for forecasting the DJIA, as it achieves the lowest validation error, of 1.087. Over time, this would produce a forecast error of 5.435, which is significantly higher than the basic GARCH model.

1.4.3 Volatile and non-volatile observations

To further understand the superior performance of the neural networks it is of interest to break down the test data set into volatile and non-volatile observations. This is also an area of interest to practitioners, who need to be able to provide accurate risk analytics in both turbulent and calm periods. We define a volatile observation as one which is greater than the mean of the test data set plus one standard deviation. The appendix also includes results for the mean plus 0.5 times the standard deviation, and the more extreme case of the mean plus 1.5 times the standard deviation. The conclusions reached based on these three separate cases are broadly synonymous.

Table 1.8 displays the QL loss for the volatile observations. The performance of the TARCH model is of immediate note. It outperforms Corsi's HAR model across nine of the ten indices, and provides the best performance of any model in eight cases. This matches with the earlier observation that the TARCH's relative performance is better when using the MSE loss than it is the QL loss. The MSE places greater emphasis on volatile observations, which the asymmetric TARCH model appears effective at forecasting.

The neural networks advantage over the baseline models has disappeared, as has the relative ordering of the different types of network, with all three providing roughly the same level of accuracy. The models which include the returns data as an explanatory variable

Table 1.8: Out-of-sample (test) QL loss for volatile observations

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
GARCH	6.466	2.385	3.822	1.590	2.938	2.067	3.005	1.650	2.979	**3.583**
TARCH	**5.316**	2.281	**3.258**	**1.579**	**2.793**	1.952	**2.906**	**1.626**	**2.868**	3.672
HAR	6.818	2.284	3.902	1.797	2.839	1.951	2.962	1.879	2.904	4.184
NN	6.905	2.276	3.998	1.815	2.907	2.003	3.035	1.914	3.145	4.294
NN (Ret)	5.853	2.237	3.473	1.763	2.878	1.975	2.982	1.927	3.106	4.340
NN-MA	6.423	2.281	3.779	1.786	2.896	2.008	3.034	1.856	3.113	4.248
NN-MA (Ret)	5.630	2.251	3.602	1.743	2.866	1.970	2.995	1.875	3.142	4.958
CNN	7.442	2.291	3.978	1.758	2.872	1.994	3.021	1.904	3.185	4.456
CNN (Ret)	6.531	2.253	3.772	1.746	2.917	1.960	3.017	1.907	3.221	3.899
CNN-MA	7.001	2.311	4.025	1.809	2.957	1.990	2.991	1.882	3.171	4.156
CNN-MA (Ret)	6.595	2.227	3.814	1.751	2.961	1.934	2.941	1.911	3.112	4.035
RNN	7.125	2.250	3.824	1.776	2.913	2.031	3.038	1.874	3.017	6.110
RNN (Ret)	5.508	2.213	3.286	1.741	2.816	1.902	2.925	1.814	3.097	4.380
RNN-MA	6.708	2.400	3.609	1.825	2.886	2.014	3.039	1.850	3.094	4.325
RNN-MA (Ret)	6.704	**2.199**	3.422	1.807	2.846	**1.892**	2.967	1.825	3.268	4.193

Note: Volatile observations are classed as those which are greater than the mean plus one standard deviation. The number of observations classed as volatile for each index is listed in Figure 1.7.

have a clear advantage over the basic models, which suggests that the networks are able to incorporate some of the same information as the TARCH model, and allow a greater reaction to negative returns. The average performance of the more complex network models is in a sense unsurprising. The majority of the data points on which it is trained fall in non-volatile times, and given also that the QL loss downplays the importance of volatile observations, we would expect the weights of the model to converge to values which perform best in such times. It would be of interest to investigate if a more concrete asymmetric structure imposed on the models in the style of TARCH would improve forecasting performance, as there appears to be some limitation to the extent the networks can adapt the same weights for volatile and non-volatile periods.

Given the volatile results, it is clear that the neural networks performance advantage must be for non-volatile observations, and this is shown clearly in Table 1.9. The table accentuates the ordering of models observed in the main results, Table 1.3. An alternative perspective on the superior performance of the networks for the non-volatile observations is that there may be a lower level of noise in calm periods, allowing the networks to pick up more complex, non-linear patterns in the series. In volatile times, observations are likely to be driven to a large extent by exogenous factors such as unexpected news stories, adding a

Table 1.9: Out-of-sample (test) QL loss for non-volatile observations

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
GARCH	0.254	0.156	0.135	0.358	0.656	-0.121	0.736	0.062	0.333	0.595
TARCH	0.234	0.136	0.107	0.364	0.656	-0.128	0.740	0.043	0.336	0.556
HAR	0.209	0.067	0.058	0.026	0.647	-0.110	0.730	-0.026	0.313	0.497
NN	0.195	0.032	0.041	0.009	0.642	-0.144	0.717	-0.026	0.299	0.497
NN (Ret)	0.184	0.023	0.037	0.008	0.634	-0.147	0.708	**-0.036**	0.302	0.499
NN-MA	0.187	0.030	0.033	0.019	0.642	-0.147	0.728	-0.023	0.301	0.500
NN-MA (Ret)	0.171	0.029	0.026	0.014	0.635	-0.152	**0.705**	-0.034	0.299	0.512
CNN	0.195	0.043	0.039	0.037	0.651	-0.143	0.722	-0.014	0.299	0.498
CNN (Ret)	0.186	0.033	0.046	0.023	0.639	-0.144	0.714	-0.031	0.303	0.548
CNN-MA	0.194	0.037	0.044	0.024	0.643	-0.147	0.716	-0.014	0.301	0.518
CNN-MA (Ret)	0.184	0.030	0.036	0.024	0.635	-0.146	0.708	-0.032	0.308	0.515
RNN	0.190	0.026	0.037	0.017	0.645	-0.149	0.715	-0.025	0.301	0.491
RNN (Ret)	0.158	0.022	0.005	0.009	0.635	-0.147	0.709	-0.025	0.302	0.488
RNN-MA	0.180	0.023	0.048	0.019	0.641	-0.149	0.713	-0.022	0.301	0.497
RNN-MA (Ret)	**0.158**	**0.019**	**0.000**	**-0.002**	**0.632**	**-0.154**	0.707	-0.030	**0.294**	**0.485**

Note: Non-volatile observations are classed as those which are less than the mean plus one standard deviation. The number of observations classed as non-volatile for each index is listed in Figure 1.7.

significant amount of noise. In the presence of large amounts of noise, more parsimonious models generally perform better, which is consistent with the releative ordering of the models in this sub-section.

1.4.4 Multi-step-ahead forecasting

Multi-step-ahead prediction is important for practitioners, so we extend the analysis of the networks performance to this two-, five- and thirty-day-ahead forecasts. As discussed, we experiment with direct prediction using the neural networks, to allow the use of additional explanatory variables (here just returns) in a straightforward fashion. Table 1.10 displays the out-of-sample results.

At each of the three forecast horizons the network models outperform the rolling-HAR baseline model, suggesting that there is value in direct prediction, and that the networks are able to pick up useful information, even at long horizons. For the two-day-ahead forecasts, the relative ordering of the network's predictions is identical to that for the one-day-ahead forecasts, with the RNN providing the lowest errors, the NN second and the CNN third. For all networks, the returns data continues to provide a consistent improvement in performance.

Table 1.10: Two-, five- and thirty-day-ahead out-of-sample (test) QL loss

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
2-days-ahead										
HAR	0.388	0.272	0.259	0.242	0.841	0.056	0.925	0.180	0.485	0.752
NN	0.364	0.224	0.221	0.226	0.834	0.013	0.906	0.180	0.482	0.757
NN (Ret)	0.331	0.218	0.200	0.225	0.826	**0.010**	**0.901**	**0.175**	0.484	**0.748**
CNN	0.355	0.228	0.249	0.238	0.847	0.016	0.915	0.187	0.492	0.769
CNN (Ret)	0.367	0.218	0.231	0.238	0.838	0.011	0.916	0.184	0.494	0.771
RNN	0.347	0.218	0.222	0.226	0.830	0.013	0.911	0.181	**0.480**	0.756
RNN (Ret)	**0.313**	**0.210**	**0.191**	**0.217**	**0.823**	**0.010**	**0.901**	**0.175**	0.485	0.791
5-days-ahead										
HAR	0.509	0.353	0.380	0.300	0.882	0.105	0.965	0.214	0.528	0.797
NN	0.482	0.317	0.355	**0.285**	0.870	0.057	**0.936**	0.216	0.513	0.781
NN (Ret)	0.486	0.318	**0.346**	0.292	0.871	0.056	0.945	0.211	0.520	**0.760**
CNN	0.498	**0.305**	0.371	0.299	0.880	0.057	0.942	0.212	0.531	0.808
CNN (Ret)	**0.481**	0.311	0.352	0.306	0.877	**0.054**	0.951	**0.209**	0.532	0.787
RNN	0.515	0.317	0.385	0.290	0.875	0.056	**0.936**	0.216	0.513	0.828
RNN (Ret)	0.525	0.320	0.389	0.287	**0.868**	0.060	0.951	0.218	**0.510**	0.777
30-days-ahead										
HAR	0.664	0.576	0.568	0.489	1.001	0.288	1.088	0.257	0.642	0.970
NN	**0.603**	0.493	0.504	0.455	**0.988**	0.177	1.053	0.236	0.532	**0.894**
NN (Ret)	0.624	0.521	**0.500**	0.454	0.998	**0.161**	**1.049**	**0.233**	0.540	0.902
CNN	0.629	**0.460**	0.547	0.445	1.017	0.188	1.051	0.245	0.545	0.924
CNN (Ret)	0.613	0.528	0.589	0.471	1.021	0.192	1.109	0.267	0.538	0.899
RNN	0.620	0.490	0.654	0.456	1.053	0.236	1.066	0.266	0.542	0.946
RNN (Ret)	0.608	0.574	0.511	**0.445**	1.004	0.178	1.070	0.236	**0.524**	0.931

As we move to the five- and thirty-day horizons the consistent difference between the networks disappears, as does the advantage of the returns-based models. Indeed, for the thirty-day-ahead predictions, the feedforward NN (denoted "NN") is arguably the most consistent model, although the differences are small. These results should not be surprising. As we move to forecasts at further distances in the future, the relationship between observed values of the series and the future values is clearly going to be less consistent and structured. The probability of there being a complex non-linear relationship between five lags of realized variance observed a month previous and the current value is low, and as such, a simple model is likely to be sufficient to approximate the relationship, which all the network structures appear capable of.

In light of the previous discussion regarding the importance of consistency between the train, validation and test sets, it is of interest to observe whether this is maintained over longer horizons.

Table 1.11: Train, validation and test QL for thirty-step-ahead prediction

		DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
HAR	Tr	1.163	1.314	1.189	1.139	1.346	0.882	1.558	0.093	1.009	1.191
	V	0.899	0.731	1.008	0.974	1.406	0.609	1.443	0.062	0.537	0.641
	Te	0.664	0.576	0.568	0.489	1.001	0.288	1.088	0.257	0.642	0.970
NN	Tr	1.129	1.264	1.165	1.142	1.267	0.821	1.489	0.070	0.929	1.165
	V	0.859	0.618	0.970	0.906	1.393	0.563	1.416	0.107	0.444	0.745
	Te	0.603	0.493	0.504	0.455	0.988	0.177	1.053	0.236	0.532	0.894
NN (Ret)	Tr	1.115	1.253	1.141	1.137	1.291	0.803	1.484	0.062	0.928	1.158
	V	0.861	0.602	0.968	0.906	1.395	0.563	1.425	0.106	0.443	0.731
	Te	0.624	0.521	0.500	0.454	0.998	0.161	1.049	0.233	0.540	0.902
CNN	Tr	1.127	1.340	1.130	1.160	1.286	0.808	1.517	0.039	0.931	1.200
	V	0.844	0.592	0.953	0.885	1.397	0.539	1.432	0.111	0.449	0.742
	Te	0.629	0.460	0.547	0.445	1.017	0.188	1.051	0.245	0.545	0.924
CNN (Ret)	Tr	1.106	1.246	1.110	1.112	1.310	0.810	1.532	0.055	0.923	1.181
	V	0.856	0.578	0.953	0.876	1.392	0.548	1.415	0.108	0.446	0.749
	Te	0.613	0.528	0.589	0.471	1.021	0.192	1.109	0.267	0.538	0.899
RNN	Tr	1.207	1.354	1.736	1.150	1.795	0.949	1.571	-0.463	1.011	1.160
	V	0.848	0.569	0.928	0.857	1.339	0.516	1.405	0.068	0.431	0.665
	Te	0.620	0.490	0.654	0.456	1.053	0.236	1.066	0.266	0.542	0.946
RNN (Ret)	Tr	1.134	1.313	1.172	1.143	1.280	0.906	1.522	0.110	0.965	1.171
	V	0.847	0.571	0.952	0.868	1.379	0.539	1.408	0.107	0.434	0.675
	Te	0.608	0.574	0.511	0.445	1.004	0.178	1.070	0.236	0.524	0.931

Table 1.11 shows that at the thirty-day horizon the consistency is close to breaking down.[20] Comparing the validation errors of the HAR model to those of the neural networks, we can see that the networks consistently provide a lower validation error, which in turn results in a lower test error on average. This supports the idea that the machine learning methodology remains just about feasible at these horizons. However there are plenty of individual examples such as those values highlighted in red and blue, where the consistency of the ordering of the train, validation and test errors breaks down. Given that the relationship between current observations and observations a month previous is likely to be weak regardless, it is impressive that this relationship appears to be at least somewhat consistent across a fifteen year time period. In this sense, this final results table lends weight to the observation that volatility exhibits long-memory in a consistent fashion across time.

1.4.5 Optimal Parameter Choice

Given that this is the first study to conduct a wide range of experiments with neural networks in a financial time series setting, it may be of interest to practitioners to include a brief discussion of the effect of different parameter options on performance, which we do here. We include in the appendix the parameter options which achieved the lowest validation error for the one-step-ahead prediction with the QL loss for the RNN, as this was the best performing network. We also include the best parameter options for the NN-MA (Ret) and the CNN-MA (Ret), to give an indication of sensible ranges for each of the different parameters available for each network type.[21]

In general, the initialization of the networks and the learning rate proved the most important factor in providing a low error. If the standard deviation of the initialization function was too large, then the error in the first epoch would be very high, and the gradient descent algorithm would often not converge at all. However, if the standard deviation was too small, then many of the network weights would initiate at values extremely close to each

[20] The equivalent tables for the two-day-ahead and five-day-ahead forecasts are included in Tables A.3 and A.4 respectively.

[21] We do not include the full set of best parameter options for each network across each of the different forecasting tasks to keep the appendix reasonably sized. If readers are interested in these options, they can be obtained through contact with the author: jhjf2@cam.ac.uk

other, and often converge even closer during training, again giving poor performance. The latter point is of particular importance if the size of the network is very small, and small networks tended to work best if the standard deviation of the initialization function is much greater than for large networks.

The choice of learning rate is also very important for convergence, with networks often failing to converge at all if a learning rate which is too high or too low is chosen. Initial experimentation showed that a learning rate in the region of 0.00001 to 0.00009 worked effectively for the feedforward NN, whilst the range 0.0001 to 0.0009 was sensible for the CNN, and a larger rate of 0.001 to 0.009 performed well for the RNN. Convergence in far fewer epochs could be achieved by employing higher learning rates, but this tended to reduce performance, as a less precise minimum could be found. The learning rates here are specific to the gradient descent algorithm, with different optimization functions performing best with different learning rates. Preliminary experiments with the AdaGrad, Momentum and Adam optimizers found little difference in the performance with the small network sizes used here, so all results reported use the basic gradient descent algorithm.

The size elements of the network were generally not particularly important in defining if the gradient descent algorithm began to converge, but did effect the level of the minimum found. For the CNN, the size parameters (see Table A.9) were generally chosen to be quite large. This allowed the networks to approximate a sufficiently complicated array of functions, despite the simplicity imposed by the *max* operator. For the feedforward NN, a wide variety of network sizes performed well for different indices, with hidden layer sizes ranging between three and one hundred each providing the lowest error on occasion. The dimension of the RNN cells was generally kept small (under ten).

For the RNN, the most interesting parameter choice is the type of hidden cell, which could be selected from the basic RNN cell, the LSTM cell, or the GRU cell. Table A.5 displays the best performing parameter options for the one-step-ahead forecasting using the QL loss. The optimal choice of cell-type varies for each different type of network, but in general, the more complex LSTM and GRU cells provide the best predictions. This pattern continues for the optimum parameter choices for the multi-step-ahead forecasts.

1.5 Conclusions

We have presented the first comprehensive comparison of the out-of-sample performance of a range of neural network models in forecasting realized volatility. Compared with baseline models commonly used in the econometric literature the neural networks are shown to perform strongly, with recurrent neural networks consistently proving the best of the models considered. The machine learning cross-validation approach based on three separate datasets has been shown to be viable with daily realized volatility data. The choice of loss function used during gradient descent is significant to the latter result, with the QL loss providing consistently more accurate forecasts than the MSE. We have also presented a novel moving average adaptation to the neural networks, which has been shown to improve the out-of-sample errors. Overall, the results clearly demonstrate the potential of modern machine learning methods to provide accurate forecasts, suggesting that the area should continue to be explored by econometricians.

There remains significant work to be done to fully exploit the potential of machine learning for financial time series prediction. This paper largely applies methods which have been designed for use in image or natural language tasks directly to time series forecasting. The performance of the networks would undoubtedly be improved by further tailoring to the financial time series setting, beyond the moving average adaptation included here. In addition, the methodology at present does not include the generation of confidence intervals. There is some work on modelling uncertainty in the machine learning literature, and this would need to be developed in the time series setting to fully establish neural networks as a viable forecasting method.

More broadly, the “black-box" nature of the methods presented here, and indeed many machine learning methods, severely limits the range of applications in econometrics to which they can be applied. There remains significant work to be done on improving the interpretability of the models, and perhaps even in developing them for causal analysis.

Bibliography

Andersen, T. G. and Bollerslev, T. (1998). Answering the skeptics: Yes, standard volatility models do provide accurate forecasts. *International Economic Review*, 39(4):885–905.

Andersen, T. G., Bollerslev, T., Diebold, F. X., and Labys, P. (2003). Modeling and forecasting realized volatility. *Econometrica*, 71(2):579–625.

Barndorff-Nielsen, O. E. and Shephard, N. (2002). Econometric analysis of realized volatility and its use in estimating stochastic volatility models. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 64(2):253–280.

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31(3):307 – 327.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

Brownlees, C., Engle, R., and Kelly, B. (2011). A practical guide to volatility forecasting through calm and storm. *Journal of Risk*, 14.

Bucci, A. (2020). Realized Volatility Forecasting with Neural Networks. *Journal of Financial Econometrics*.

Cho, K., van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar. Association for Computational Linguistics.

Corsi, F. (2009). A Simple Approximate Long-Memory Model of Realized Volatility. *Journal of Financial Econometrics*, 7(2):174–196.

Doering, J., Fairbank, M., and Markose, S. (2017). Convolutional neural networks applied to high-frequency market microstructure forecasting. In *2017 9th Computer Science and Electronic Engineering (CEEC)*, pages 31–36.

Donaldson, R. and Kamstra, M. (1997). An artificial neural network-garch model for international stock return volatility. *Journal of Empirical Finance*, 4(1):17 – 46.

Engle, R. F. (1982). Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, 50(4):987–1007.

Engle, R. F. and Patton, A. J. (2007). 2 - what good is a volatility model?*. In Knight, J. and Satchell, S., editors, *Forecasting Volatility in the Financial Markets (Third Edition)*, Quantitative Finance, pages 47 – 63. Butterworth-Heinemann, Oxford, third edition edition.

Feng, G., He, J., and Polson, N. G. (2018). Deep learning for predicting asset returns.

Glosten, L. R., Jagannathan, R., and Runkle, D. E. (1993). On the relation between the expected value and the volatility of the nominal excess return on stocks. *The Journal of Finance*, 48(5):1779–1801.

Hansen, P. R. and Lunde, A. (2005). A forecast comparison of volatility models: does anything beat a garch(1,1)? *Journal of Applied Econometrics*, 20(7):873–889.

Hansen, P. R. and Lunde, A. (2006). Realized variance and market microstructure noise. *Journal of Business & Economic Statistics*, 24(2):208–218.

Hastie, T. and Tibshirani, R. (1993). Varying-coefficient models. *Journal of the Royal Statistical Society. Series B (Methodological)*, 55(4):757–796.

Heber, Gerd, Lunde, A., Shephard, N., and Sheppard, K. (2009). Oxford-man institute's realized library version:0.2. *Oxford-Man Institute, University of Oxford*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

Liu, F., Pantelous, A. A., and von Mettenheim, H.-J. (2018). Forecasting and trading high frequency volatility on large indices. *Quantitative Finance*, 18(5):737–748.

Miranda, F. G. and Burgess, N. (1997). Modelling market volatilities: the neural network perspective. *The European Journal of Finance*.

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

Vortelinos, D. (2015). Forecasting realized volatility: Har against principal components combining, neural networks and garch. *Research in International Business and Finance*.

Welch, I. and Goyal, A. (2007). A Comprehensive Look at The Empirical Performance of Equity Premium Prediction. *The Review of Financial Studies*, 21(4):1455–1508.

White, H. (2000). A reality check for data snooping. *Econometrica*, 68(5):1097–1126.

Xiong, R., Nichols, E., and Shen, Y. (2015). Deep learning stock volatilities with Google domestic trends. *arXiv, https://arxiv.org/abs/1512.04916*.

Yang, J.-B., Nhut, N., San, P., li, X., and Shonali, P. (2015). Deep convolutional neural networks on multichannel time series for human activity recognition. *IJCAI*.

Chapter 2

Improving prediction intervals for neural networks using the quantile loss function

We propose the use of the quantile loss function for generating prediction intervals for neural networks. We show that prediction intervals generated in this fashion significantly outperform previous approaches in terms of accuracy, both in simulated and real-world datasets. In addition, we propose three novel architecture variants for predicting the quantiles; isolated training of weights for each quantile output layer, modelling the quantile in an additive fashion from the mean prediction, and residual connections to the output layers. In each case, we show that the accuracy of the prediction intervals is increased, with their combination yielding substantially more reliable prediction intervals than previous approaches. We perform experiments with both feedforward neural networks (NNs) on i.i.d. data, and with recurrent neural networks (RNNs) on time series data, demonstrating that our results are robust across choices of network and data structure.

2.1 Introduction

Neural networks are becoming increasingly popular in finance and economics for prediction tasks, particularly in applications with large datasets. A common criticism of their application, however, is that prediction intervals are not included, or if they are, that there is little understanding of the most accurate method of generating prediction intervals in limited sample settings.

In this paper, we propose the use of the quantile loss function for generating prediction intervals for both feedforward and recurrent neural networks, and present the first detailed examination of the accuracy of these intervals. We provide results both in simulation, and in an empirical study using volatility data, which demonstrate that prediction intervals generated using the quantile loss function are significantly more accurate than previous approaches. These results are consistent across both feedforward neural networks and recurrent neural networks, and with i.i.d. and time series data.

In addition, we experiment with three novel variants of network architecture. Firstly, we propose freezing of the main network weights used to predict the mean function, and training of independent weights for each quantile output layer. This ensures that the accuracy of the mean prediction is not adversely impacted, and stabilizes the quantile predictions, particularly for the outer quantiles. Secondly, we experiment with modelling the quantile outputs in terms of their difference from the prediction of the mean/median, stabilizing results in small sample settings. Finally, we propose the use of a residual connection, which allows each quantile output layer to access the input data directly, further improving accuracy.

We conduct a detailed simulation study of each of the above variants across multiple data generating functions, noise distributions, and sample sizes, and show both that prediction intervals generated using quantile output layers are more accurate than previous approaches, and that all three of the proposed adaptations to the architecture improve accuracy further. We consolidate these conclusions with experiments on real-world datasets, showing that prediction intervals for financial time series are improved by each of the above contributions.

2.1.1 Existing Literature

Quantile Neural Networks

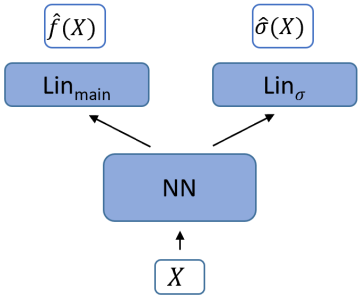
The term “regression quantiles" was introduced in the seminal paper of Koenker and Bassett [1978], which presented the quantile loss function for regression, and established the asymptotic properties. The first combination of the quantile loss function with neural networks was presented in Taylor [2000]. The authors showed the weights of a neural network could be

optimized using the quantile loss function, with an appropriate regularization term to prevent overfitting added. The resulting framework avoided the requirement for a distributional assumption on the volatility process, and was shown to outperform GARCH-based quantile estimates. Since that point, quantile neural networks have been used in energy load forecasting, He et al. [2016] and Zhang et al. [2018], financial volatility, Pradeepkumar and Ravi [2017], wind power, He and Li [2018] and a range of other indices.

Prediction Intervals for Neural Networks

The most common approach to generating prediction intervals for neural networks was introduced in Nix and Weigend [1994]. Their approach modifies a traditional feedforward network to have two outputs. The usual prediction of the mean, $\hat{f}(x)$, and an additional estimate of the variance, $\hat{\sigma}(x)$. This is demonstrated in Figure 2.1.

Figure 2.1: Nix and Weigend adaptation of NN architecture for prediction intervals



If we make the assumption of normally distributed errors, the target probability distribution is the standard:

$$P(y_i|x_i, \mathcal{N}) = \frac{1}{\sqrt{2\pi\sigma^2(x_i)}} exp(\frac{-[y_i - f(x_i)]^2}{2\sigma^2(x_i)})$$

The weights of the network can then be optimized (using any optimization procedure) to maximize the log likelihood. This network structure is a sensible first step towards prediction intervals, but has a number of potential drawbacks.

Most obviously, the assumption of gaussian errors may not always hold. Given that neural networks are generally applied to large datasets, with the intention of limiting assumptions about the structure of the underlying true function, $f(x)$, it is preferable to avoid assumptions on the distribution of the errors also. Secondly, with non-iid data, such as in the empirical example in this paper, the assumption is void, limiting the potential application of the same model to Recurrent Neural Networks. Finally, the Nix and Weigend method places a restraint on the loss function that can be used to optimize the network. In many applications, researchers have preferred loss functions (mean squared error, mean average error, quasi-likelihood etc.) which perform optimally[1] on a specific dataset. This limitation potentially explains why prediction intervals continue to be a rarity in the literature.

Prediction intervals with Quantile Neural Networks

Despite the extension from quantile neural networks with a single output, as introduced in Taylor [2000], to quantile neural networks with multiple outputs, which can generate prediction intervals, being straightforward, the literature has been slow in developing this method. In concurrent work to this, Rodrigues and Pereira [2018] have also suggested a multi-output neural network to jointly model the mean and two or more of the quantiles. They demonstrate the effectiveness of the model using two datasets from the transportation domain, and find that the multi-learning approach surprisingly improves their estimation of the mean, with the quantile outputs acting as additional regularization. We lay out the differences between our work and Rodrigues and Pereira [2018] clearly in the contribution section below. A further integration of the quantile loss function with neural networks is Yan et al. [2018], which focuses on predicting a single quantile only, as opposed to a prediction interval around the mean prediction.

2.1.2 Contribution

The contribution of this paper is to present a novel alternative to Nix and Weigend's method, making use of the quantile loss function. In doing so, we avoid the three limitations of

[1] As measured by the accuracy of the mean prediction, $\hat{f}(x)$

their work. There is no need for a distributional assumption on the errors, the network can be optimized with any loss function of the researchers choice[2], and the method is equally applicable to forecasting of time series with RNNs.

We note that Rodrigues and Pereira [2018] concurrently propose a related approach, jointly modelling the median and quantile outputs for data in the transportation sector. However, in addition to their work, we present and evaluate several novel variants of the proposed architecture.

- Separate optimization of the main network weights (using the mean prediction loss function) and each quantile output layer's weights (using their individual quantile loss functions)
- The estimation of the difference between the mean prediction and each quantile, as opposed to the quantile value directly.
- The use of a residual connection in the network architecture, allowing the quantile nodes of the network to attend to the feature vector directly.

Through both a simulation study and evaulation on financial returns datasets, we show that the use of the quantile loss functions for generating prediction intervals results in consistently more accurate intervals than the Nix and Weigend approach, and furthermore that each of the above three architecture innovations improve accuracy further, both for feedforward neural networks and recurrent neural networks.

The paper proceeds as follows. In Section 2.2, we present the two neural network architecture which we use for experimentation, the feedforward neural network and the recurrent neural network. We also present the three adaptations to the architecture introduced above, and discuss the optimization procedure. Section 2.3 explains the experimental setup, both for the simulation study and the empirical datasets. Section 2.4 then presents the results of the simulation study, and Section 2.5 the results on the empirical datasets. Finally, Section 2.6 concludes.

[2]The optimization of the main network weights is completely independent of the error estimates in our preferred method, ensuring that there is no loss of accuracy of the mean prediction.

2.2 Network Architectures

The main focus of this paper is on the **output** layers of neural networks, which we utilize for producing estimates of the quantiles. However, we wish to demonstrate that the improvements generated by the innovations in the output layers are independent of the main neural network architecture. To this end, we perform experiments with the two most commonly used architectures in econometric/financial applications. The feedforward neural network (NN), and the recurrent neural network (RNN), with the latter applicable to forecasting time series. We describe the two respective architectures below.

2.2.1 Feedforward Neural Networks

A feedforward neural network takes an input X_i (a vector or matrix of explanatory variables) and returns a scalar or vector, $\hat{Y}_i \in \mathbb{R}^{d_y}$.[3]

Given an input, x, the output, $h_{\theta,l}(x) \in \mathbb{R}^{d_l}$ of the l_{th} layer of the network is defined as

$$h_{\theta,l}(x) = g_l(W_l h_{\theta,l-1}(x) + b_l) \qquad l = 1, .., L$$

where $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ are referred to as the "weights" of the network and $b_l \in \mathbb{R}^{d_l}$ is referred to as the "bias". The input to layer one of the network is a vector of explanatory variables, $h_{\theta,0} = x$. The function $g^l(z)$ is a non-linear transformation, which is applied to all outputs of the layer individually, $g^l(z) = (\sigma(z_1), \sigma(z_2), ..., \sigma(z_{d_l}))$ with $z_1, z_2, ..., z_{d_l} \in \mathbb{R}$. The choice of σ is typically the sigmoid function, tanh or recitified linear unit (RELU).[4]

The final (output) layer of the network is a linear regression without a non-linearity, producing the networks prediction, $m_\theta(x) \in \mathbb{R}^{d_y}$:

$$m_\theta(x) = W_o h_{\theta,L}(x) + b_o$$

.

[3] $\hat{Y}_i$ can also represent a probability distribution, as with logistic regression, though we limit the analysis to scalar values of $\hat{Y}_i$ in this paper, as is more common in economic prediction models.

[4] The RELU is defined as $f(x) = max(0, x)$

where $W_o \in \mathbb{R}^{d_L}$ and $b_o \in \mathbb{R}$.

Figure 2.2: Feedforward Neural Network with a single hidden layer, and four inputs (features)

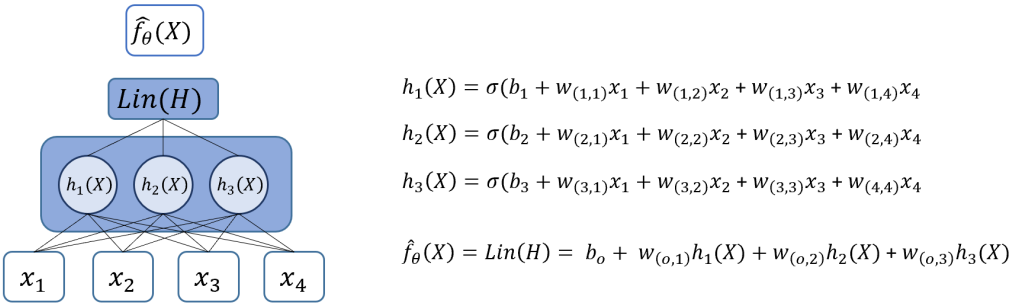


Figure 2.2 shows a feedforward NN with a feature vector X_i of dimension 4, and a single hidden layer with 3 nodes. The activation of the illustrated network is the sigmoid function.

The weights in the network, $W = (W_1, ..., W_L, b_1, ..., b_L, W_o, b_o)$ can be optimized using a range of different algorithms, described in Section 2.2.3 below.

2.2.2 Recurrent Neural Networks

Consider a time series, $\{x_1, ..., x_t, ..., x_T\}$. In the basic RNN, the output of the network at time t, $h_t(x_{\hat{t}}) \in \mathbb{R}^{D_h}$ (where $x_{\hat{t}}$ refers to $(x_t, x_{t-1}, ..., x_1)$ and D_h is the dimension of the hidden layer of the network) can be described as:

$$h_t(x_{\hat{t}}) = g(W_h h_{t-1}(x_{\hat{(t-1)}}) + W_x x_t + b_h)$$

where $W_h \in \mathbb{R}^{D_h \times D_h}, W_x \in \mathbb{R}^{D_h \times d_x}$ and $b \in \mathbb{R}^{D_h}$. The weights matrices, W_h and W_x are not time dependent, with the same weights matrices being used at each timestep. The output $h_t(x_{\hat{t}})$ is termed the "hidden state" of the network, and can be thought of as holding information from previous timesteps. This information is then combined with the new input, x_t, to give our prediction at this timestep. The hidden state needs to be initialised at timestep zero, with the usual choice of a vector of zeros being used in this paper.

The remainder of the network is identical in format to the basic NN. The function $g(z)$ is a non-linear transformation, which is applied to all outputs of the layer individually,

$g(z) = (\sigma(z_1), \sigma(z_2), ..., \sigma(z_{D_h}))$. The output layer is a linear regression without a non-linearity, which produces our estimate of volatility for the next period, $f_\theta(x) \in \mathbb{R}$:

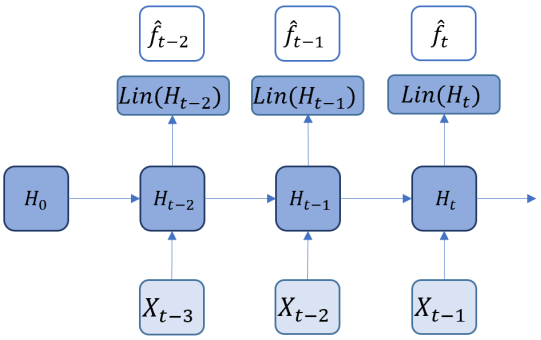
$$f(x_{\hat{t}}) = W_o h_t(x_{\hat{t}}) + b_o$$

.

where $W_o \in \mathbb{R}^{D_h}$ and $b_o \in \mathbb{R}$.

Diagrammatically, the RNN is best represented in “rolled out" form, although the process is in reality a continuous cycle of arbitrary length.

Figure 2.3: Single layer Recurrent Neural Network rolled out over three timesteps



At each point in time, the network makes a predictions for the next value in the time series based on the previous observation, x_{t-1}, and the hidden state of the network. It is possible to stack multiple layers of RNN cells, with the inputs to the next layer generally being the hidden states of the previous layer, as opposed to the inputs, x_t. There have been multiple extensions of the architecture of the recurrent neural network described above, such as the Long Short-Term Memory network of Hochreiter and Schmidhuber [1997] and the Gated Recurrent Unit of Cho et al. [2014]. However, in this paper we limit our experiments to the basic version only, to compare our innovations in the output layers in a consistent fashion.

2.2.3 Optimization

Optimizing of both network architectures described above is identical. We first randomly initiate the weights of the network. In this paper, we initiate the weights from a normal

distribution with mean 0 and standard deviation 0.05. We then define a loss function L(X). We experiment with a range of different loss functions, each of which is described below in Section 2.2.4. Regardless of which loss function is used, the optimization process remains identical. We iterate through batches of features (X) and targets (Y), and differentiate the loss function with respect to each of the weights of the network. We then update each of the weights based on this gradient. There are a wide variety of increasingly complex algorithms which have been suggested for taking this update step, including stochastic gradient descent (SGD), the Momentum optimizer Sutskever et al. [2013] and the Adam optimizer Kingma and Ba [2014]. In this work, we use SGD, which generally provides stable convergence in small network settings, and which is described as follows:

randomly initiate the weights of the model, W
for epoch in number of epochs:
    for batch in number of batches:
        evaluate the gradient of the loss function $L(X)$ with respect to the weights, ∇_W
        update the weight values: $W_{new} = W_{old} - \alpha \nabla_W L(W)$
restore network weights to epoch with lowest validation loss

Throughout the process, we monitor the loss on the validation set, and save the model weights every time it decreases. At the end of training, we reset the model weights to the point at which the lowest validation loss was achieved. Monitoring the validation loss in this fashion helps to prevent the network from overfitting the training data, even in the cases of simple data generating functions, $f(x)$ and large models.

An important addition to the optimization procedure described above is decaying the learning rate, α. During initial training, a relatively large value of α allows the weights to converge quickly towards their optimum values. However, it is often beneficial to gradually decrease the learning rate later in training, to allow smaller steps down the gradient, often improving the final validation loss. There are a range of potential functions for generating the learning rate decay. In this paper, we use a simple decay approach, in which the learning rate is halved every n epochs. The value of n is set to 4000 for this paper.

There are a number of other important hyperparameters which must be set during the optimization process. These include the learning rate, α, the batch size, and the number of

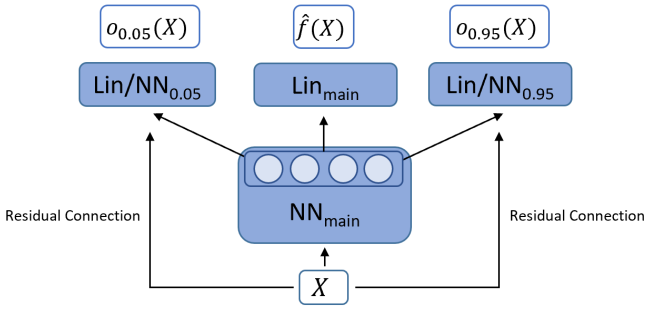
training epochs[5]. The values used for each of these hyperparameters, and the methods used to choose them, are discussed in Section 2.3.1.

2.2.4 Output Layers

Above, we have described the two basic network architectures of the feedforward neural network (NN) and recurrent neural network (RNN) with which we perform experiments. Next we turn our attention to the main focus of this paper: modifications of the **output layers** of the networks, which allow us to produce prediction intervals. The method of Nix and Weigend, in which they modified the architecture to output an estimate of the mean and conditional variance of the target variable (described in the introduction), was the first such approach. Here, we present a series of alternatives, all of which utilize the quantile loss function. We present the modifications to the outputs layers on top of a feedforward neural network. However, exactly the same modifications are possible with the RNN, with the only difference being that estimates of the quantiles are based on past values of the time series, as opposed to a static feature vector X. We perform experiments on the performance of each output layer variation on both feedforward NNs and RNNs.

Estimation of a prediction interval is achieved by directly predicting the conditional quantiles of the target variable, via adding to the main architecture an additional output layer for each quantile we wish to predict. This is illustrated in Figure 2.4, which shows the architecture used to predict the mean with the 5th and 95th quantiles (giving a 90% prediction interval). The final **hidden** layer of the NN is denoted as the light blue circles.

Figure 2.4: Prediction network with quantile outputs



As in the feedforward network previously, the mean prediction, $\hat{f}(x)$, is generated by a linear layer, LIN_{main}, which takes as input the final hidden layer of the main neural network, NN_{main}. We optimize the weights of the output layer LIN_{main} using gradient descent on the mean squared error (MSE) loss function:

[5] An epoch refers to iterating through the entirety of the training dataset a single time.

$$L_f = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{f}(X_i))^2$$

Then, for each quantile, we add an additional output layer, denoted as $LIN_{0.05}$ for the 5th quantile and $LIN_{0.95}$ for the 95th quantile. Each of these output layers also takes as input the final hidden layer of NN_{main}, and outputs a scalar, denoted $o_{0.05}$ for the 5th quantile. The scalar $o_{0.05}$ is then transformed into a prediction for the 5th quantile, denoted $\hat{q}_{0.05}$; we experiment with various transformations (the simplest of which is simply setting $\hat{q}_{0.05} = o_{0.05}$), all of which are described in Section 2.2.4 below. The residual connections in Figure 2.4 are described in Section 2.2.4.

In order to optimize the quantile output layers to give predictions of the different quantiles of the distribution, we use the quantile loss function, denoted as $L_{0.95}$ for the 95th quantile, and visualized in Figure 2.5:

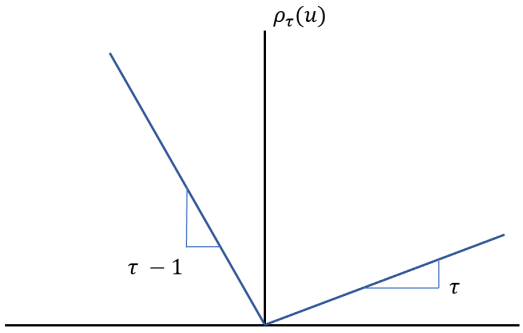
$$L_{0.95} = \frac{1}{N}\sum_{i}^{N}\rho_{0.95}(Y_i - \hat{q}_{0.95}(X_i)) \tag{2.1}$$

$$\rho_\tau = \begin{cases} \tau u & \text{if } u \geq 0 \\ (\tau - 1)u & \text{if } u < 0 \end{cases}$$

In other words, we optimize the weights of each separate quantile output layer using gradient descent against the relevant quantile loss function. For example, in Figure 2.4, we would optimize the weights of $LIN_{0.05}$ using the quantile loss function for the 5th quantile. There is no limit (other than computational expense) to the number of additional quantile outputs layers we can add to the network, allowing estimation of the full distribution if required.

Intuitively, estimation of the quantile loss function is appropriate for providing the prediction intervals we are targeting; for a given τ, the function is minimized where $\tau\%$ of the data lies below the prediction $\hat{q}_\tau$, and $1 - \tau\%$ of the data above $\hat{q}_\tau$.

Figure 2.5: Tilted quantile function



We experiment with three aspects of the described architecture in the experiments, which we describe in turn below.

Variant 1: Joint optimization of NN weights

As described in Section 2.2.3, the weights of any neural network architecture are optimized by differentiating a loss function with respect to each of the individual weights, given some features, X, and targets, Y. Given that we now have multiple loss functions (one for the mean prediction, and an additional one for each of the quantile predictions), we can choose which loss function to optimize each of the weights of the network against. As described above, the weights of output layers are all optimized against their relevant individual loss function (either the MSE loss for the mean prediction, or a quantile loss). However, there are a number of options for optimizing the weights of the main neural network (NN_{main} in Figure 2.4).

In the simplest case, denoted by the label *joint*, we add all of the losses together, and optimize the weights of the main network against this loss.

$$Loss_{main} = \alpha_1 L_f + \alpha_2 L_{0.95} + \alpha_2 L_{0.05}$$

The weights α_i can be tuned to prioritize the mean or quantile predictions. If we include output layers for more than two quantiles, these can simply be added to the loss function. This is the loss function used in Rodrigues and Pereira [2018], although they use an alternative network architecture, based on Convolutional LSTM layers.

An alternative to this approach, which is novel to this work, is to optimize the weights of the main network, NN_{main}, against the mean prediction only.

$$Loss_{main} = L_f$$

The quantile loss functions, $L_{0.95}$ and $L_{0.05}$, are therefore used only to optimize the weights of their individual output layers respectively. Training of the quantile output layers can be done concurrently with the main network, or completed once the weights of NN_{main} have been optimized and "frozen". In practice, we find the latter provides smoother convergence, and adds little to the overall training time, as all quantile outputs can be optimized simultaneously.

We denote this approach using the label *sep* in the results sections. An additional (potential) advantage of this method is that the mean prediction is left unaltered (i.e. the accuracy of the mean prediction will be independent of the quantile predictions, and the number of quantile outputs). This allows the researcher to generate prediction intervals without compromising their main results, and even to calculate the intervals for a previously trained neural network.

Variant 2: Additive quantile predictions

In Figure 2.4, we denoted the output for the prediction of the τth quantile as $o_\tau(X)$. In this section, we discuss two different methods for converting this output into a prediction for the τth quantile, $\hat{q}_\tau(X)$.

The simplest approach is to simply use $o_\tau(X)$ as the prediction. That is:

$$\hat{q}_\tau(X) = o_\tau(X)$$

This is the approach used in Rodrigues and Pereira [2018]. It keeps each quantile prediction independent of the mean prediction. However, one of the key potential problems with predicting quantiles in limited sample size settings (particularly in the case where we are predicting multiple quantiles which are close to each other) is the possibility of overlapping quantiles.

In order to limit the possibility of overlapping quantiles, and taking advantage of the inherent structure of quantiles[6], we propose a simple alternative approach, whereby quantile estimates are generated additively from the mean prediction or other quantiles. That is:

$$q_\tau = \begin{cases} \hat{f}(X_i) + o_\tau & \text{if } \tau \geq 0.5 \\ \hat{f}(X_i) - o_\tau & \text{if } \tau < 0.5 \end{cases}$$

The constraint that $o_\tau > 0$ ensures that the quantile predictions cannot overlap with the mean/median prediction, and can be enforced either during training or in post-processing with the ReLU activation function.[7] If multiple quantiles are being predicted, the outer quantiles can be modelled as the sum of the inner quantile plus the quantile prediction:

[6] I.e. we know that $q_{0.05} \leq q_{0.10} \leq q_{0.15}$ etc.

[7] In our experiments, we did not observe overlapping quantiles with the additive approach, and as a result it was not necessary to include the ReLU activation in our architecture.

$$q_{0.95} = q_{0.90} + o_{0.95}$$

However, in this paper we model all quantile outputs relative to the mean prediction, simulating the usecase where a single prediction interval is being generated.

We denote results using the additive approach with an $\_a$ after the initial word. For example, if we are optimizing the weights of the main network using the joint loss function, and using additive quantiles, we denote the results $joint\_a$. For cases where we set the quantile output equal to o_τ we leave the first word unchanged (i.e. simply denote results with *joint* for the joint optimization case, or *sep* for the separate case). For a summary of all of the labels for different variations, see Table 2.1 below.

The additive approach, despite being a very small adjustment to the architecture, simplifies the predictions of the quantiles significantly in some cases. For example, in the case of homoskedastic noise, the quantile outputs can simply be constant. This can be easily achieved by the network by setting all of the weights in each quantile output layer to 0.0, and the bias term to the correct level.

Variant 3: Residual connections

The third and final aspect of the architecture which we experiment with is the input to the quantile output layers. The baseline case, discussed above, uses the weights of the final hidden layer of the main network, NN_{main} as the input to each quantile output layer. In the case of the final hidden layer of the network being dimension 3, we would calculate o_τ as follows:

$$o_\tau(X) = b_\tau + w_{(\tau,1)}h_1(X) + w_{(\tau,2)}h_2(X) + w(\tau,3)h_3(X)$$

That is, the inputs to each of the quantile output layers are identical to the inputs to the mean output layer.

As an additional variation, we experiment with passing the input vector, X, to each of the quantile layers independently, as an additional feature. For the case where we have a final hidden layer of dimension 3, and a feature vector, X, of dimension 2, the τth quantile output is calculated as follows:

$$o_\tau(X) = b_\tau + w_{(\tau,1)}h_1(X) + w_{(\tau,2)}h_2(X) + w(\tau,3)h_3(X) + w_{(\tau,x1)}x_1 + w_{(\tau,x_2)}x_2$$

We term this a “residual" connection, following He et al. [2015]. Residual connections were originally introduced as a method of training extremely deep neural networks (those with many hidden layers). Although we are not experimenting with very deep neural networks in this paper, we suggest that residual connections may still be useful for quantile outputs,

as they allow each output to directly exploit information in the feature vector X which may not be useful for the mean prediction (and therefore may be factored out by the weights of NN_{main}).

For example, if the noise is heteroskedastic, prediction of the mean does not require modelling the level of heteroskedasticity in the main layers of the network. A residual connection ensures that the quantile outputs can still model this scenario. This is particularly applicable in the case where we optimize the weights of NN_{main} using just the mean function (results denoted *sep*).

We denote results which use a residual connection with $\_r$ added to the end of the experiment string. For example, in the case where we jointly optimize the main network weights, use additive quantile outputs, and add a residual connection, is denoted $joint\_a\_r$. This, and all result labels, are illustrated in Table 2.1 below.

Table 2.1: Variants of network architectures for prediction of quantiles

	Joint optimization of all NN weights	Additive quantile predictions	Residual connections
joint	Yes	No	No
joint_ a	Yes	Yes	No
joint_ a_ r	Yes	Yes	Yes
sep	No	No	No
sep_ a	No	Yes	No
sep_ a_ r	No	Yes	Yes

In total, then, we have six variants of architectures including quantile output layers, on which we perform experiments. In addition, for the feedforward neural network, we baseline our results using the Nix and Weigend approach, denoted *nw* in the results tables.

2.3 Experimental Setup

In order to evaluate our proposed innovations to the network's output layers, we present two sets of experiments. The first are simulation based. We define a range of data generating functions, $f(x)$, draw observations based on these functions and a noise generating function, $\epsilon(x)$, and fit each of the network variants to the resulting datasets. Given that we know the true function, $f(x)$, and the noise generating function, $\epsilon(x)$, we can evaluate our model variants against the true conditional quantiles.

The second set of experiments are based on real datasets, specifically returns and volatility data. In this case, we split the data into a train, validation and test dataset, and fit a neural

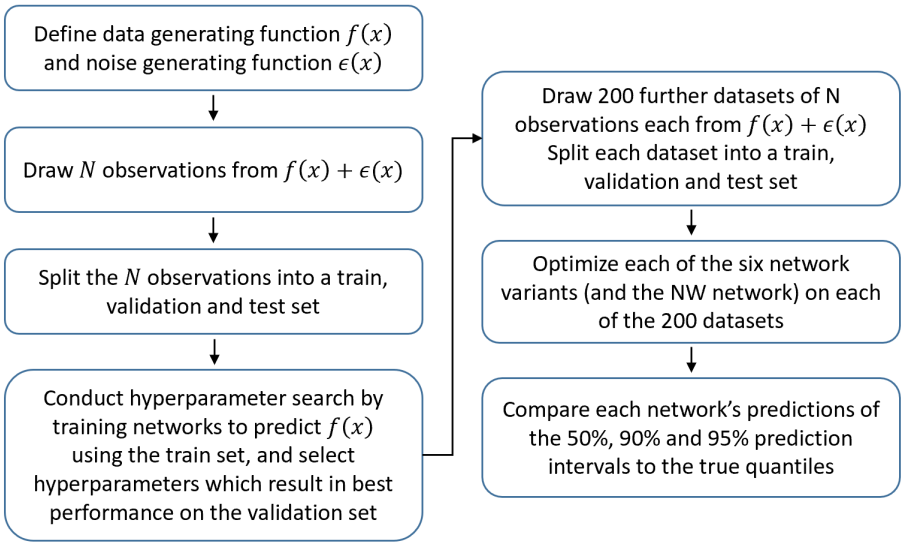
network using the train and validation data. We then evaluate the accuracy of the fitted prediction intervals on the test dataset.

2.3.1 Simulation

As described above, the aim of this section of experiments is to evaluate the accuracy of the prediction intervals generated by each model variant introduced in Section 2.2.4 on simulated data. Using simulated data allows us to evaluate the generated prediction intervals against the true quantiles, across a range of data generating functions.

The process is visualized in Figure 2.6. We define a data generating function, $f(x)$, and a noise generating function, $\epsilon(x)$. Next we draw observations based on these functions, giving a dataset of size N. We split this dataset into a train, test and validation set. We then conduct a hyperparameter search, in which we train a neural network to predict $f(x)$ only (i.e. **not** outputting prediction intervals as well at this stage).[8] We select the hyperparameter set which achieves the lowest loss on the validation set. Next, we draw a further D ($D = 200$ in all cases) datasets from $f(x) + \epsilon(x)$, each of size N. Alongside each of these datasets, we draw an additional out-of-sample dataset of size K, with K set to 300 in all experiments.[9] Using the chosen hyperparameter set, we train networks with each of the six model variants described in Table 2.1 on each of the D draws. In addition, we optimize a network based on the Nix and Weigend approach.

Figure 2.6: Summary of simulation steps



[8] See Section 2.3.1 below for details of the hyperparameter selection procedure.

[9] We could use the test datasplit as out-of-sample data, but in the case where $N = 200$, the test split consists of just 30 observations. Given we have the ability to simulate data, it makes sense to draw a larger out-of-sample set to evaluate results on.

In total then, for each dataset and each draw, we have seven different estimates of prediction intervals. We compare the accuracy of each of these variants against the true conditional quantiles, q_τ, for $\tau = (0.025, 0.05, 0.25, 0.75, 0.95, 0.975)$, using the out-of-sample dataset of size S. This corresponds to the outputs we would get for 50%, 90% and 95% prediction intervals.

The range of data generating functions, noise generating functions, sample sizes and model selection hyperparameters are described in the following sections.

I.I.D. Data Generating Functions (DGFs)

For the feedforward neural network, we perform experiments on five separate data generating functions, described in order of increasing complexity. For each of the functions, $f(X)$, we generate observations as follows:

$$Y_i = f(X_i) + \epsilon_i(X_i)$$

The noise is generated by the function $\epsilon_i(X_i)$. In addition to the five data generating functions, we experiment with four separate noise-generating functions, giving a total of 20 combinations. This allows us to understand the performance of the prediction intervals when the quantiles are generated by a variety of increasingly complex functions.

DGF1:
The first DGF, included as a baseline, has a simple linear structure.

$$f(X_i) = \alpha + \beta X_i$$

The coefficients, α and β, are drawn from $Unif([-1, 1])$, and there are three explanatory features (dimension $X_i = 3$)

DGF2:
The second DGF includes interaction and squared terms. The feedforward network should still be capable of fitting $f(X)$ perfectly (given enough data).

$$f(X_i) = \alpha + \beta X_i + \gamma X_i X_i'$$

α, β and γ are drawn from $Unif([-1, 1])$, with the dimension of X_i being 3.

DGF3:
The third DGF is based on the sin function. The dimension of X_i is 1.

$$f(X_i) = \frac{3}{2} sin(3X_i) + X_i$$

DGF4:

For the fourth DGF, we increase the dimensionality of the features vector, X_i, to 3. We therefore have multiple non-linear additive effects generated by the *sin* function, as well as linear effects with respect to each explanatory variable.

$$f(X_i) = \beta sin(\gamma X_i) + \delta X_i$$

α, β and γ are drawn from $Unif([-1, 1])$.

DGF5:

This final variant tests the performance of the models when the data generating function is more complicated than the network selected by the researcher. This ensures that the bias is non-zero.

We generate data from a feedforward neural network which is larger than any of the parameter options we try during the model selection stage.

$$f(X_i) = NN(X_i)$$

$$\epsilon_i \sim \mathcal{N}(0, 0.1^2)$$

We use three hidden layers in the network, each of size 50, and initiate the weights randomly from $\mathcal{N}(0, 0.5^2)$.

I.I.D. Noise generating functions

Given we are primarily interested in understanding the performance of each neural network variant in modelling the quantile generating functions, we evaluate the models on four separate noise generating functions, $\epsilon_i(X_i)$.

NGF1:

For the baseline case, we generate observations from the normal distribution, with a standard deviation of 2.0.

$$\epsilon_i(X_i) \sim \mathcal{N}(0, 2^2)$$

NGF2:

Next, we consider the case of heteroskedastic noise, with the standard deviation increasing linearly with $X_{i,1}$.

$$\epsilon_i(X_i) \sim \mathcal{N}(0, 0.2X_{i,1}^2)$$

That is, if the dimension of the feature vector, X_i, is greater than 1, the distribution of the noise term varies only with respect to the first feature. Given that we generate the sample of X_i from $Unif([0.0, 4.0])$, the average size of the noise terms remains similar to the homoskedastic case.

NGF3:

In the third variation, we keep the heteroskedasticity with respect to $X_{i,1}$, but generate the errors from a skewed normal distribution, with skewness of 1.0.

$$\epsilon_i(X_i) \sim SkewN(0, 0.2X_{i,1}^2)$$

Given that the quantile outputs of the neural network are generated individually, they should be capable of approximating the skewed quantiles precisely. This is not the case for the Nix/Weigend method, which is only able to model symmetric noise distributions.

In total then, we have five i.i.d. models and three i.i.d. noise generating functions, giving fifteen simulation combinations in total. For each model/noise combination, we train seven different feedforward NNs, corresponding to each of the seven quantile output layers described in Section 2.2.4, allowing a comparison of results across a broad range of data-generating functions.

Time Series Models

To evaluate the performance of the recurrent neural network, we generate data from four time series models. The first two, an AR process and and ARMA process, have homoskedastic noise. The second two are based on the ARCH, Engle [1982] and GARCH, Bollerslev [1986], models, allowing comparison of performance of the prediction intervals under heteroskedasticity. Unlike in the i.i.d. case above, where we simulate with combinations of data generating functions and noise generating functions, in the time series case we present each DGF and NGF together.

TGF1:

As a baseline, we generate data from an AR(1) process, with homoskedastic noise. Each of the quantile model variants should be capable of modelling this process perfectly, given enough observations.

$$y_t = \phi y_{t-1} + \epsilon_t$$

$$\epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

To ensure stationarity of the process, we set ϕ_1 equal to 0.9. The noise, ϵ_t, is generated from the normal distribution, with $\sigma = 0.2$.

TGF2:

As the second time series data generating function, we use an ARMA(3, 3) model, again keeping the noise homoskedastic.

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \phi_3 y_{t-3} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \theta_3 \epsilon_{t-3} + \epsilon_t$$

$$\epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

We draw $phi_{1,2,3}$ and $\theta_{1,2,3}$ from $Unif([-1, 1])$. As in TGF1, the noise is homoskedastic, with $\sigma = 0.2$.

TGF3:

For the second two time series generating functions, we introduce conditional heteroskedasticity, using the ARCH and GARCH models. In each case, the basic series is as AR(1) model, and the noise is modelled as an ARCH(1) process.

$$y_t = \phi y_{t-1} + \epsilon_t$$

$$\epsilon_t = \sigma_t z_t$$

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

$$z_t \sim \mathcal{N}(0, \sigma^2)$$

We set $\phi = 0.9$, $\alpha_0 = 0.7$ and $\alpha_1 = 0.6$, ensuring persistence in the magnitude of the error terms. z_t is a strong white noise process, with $\sigma = 0.2$.

TGF4:

For the final time series generating function, we increase the complexity of the heteroskedasticity, modelling the variance of the error term using a GARCH(1, 1) model.

$$y_t = \phi y_{t-1} + \epsilon_t$$

$$\epsilon_t = \sigma_t z_t$$

$$\sigma_t^2 = \alpha_0 + \alpha_1 \epsilon_{t-1}^2 + \beta_1 \sigma_{t-1}^2$$

$$z_t \sim \mathcal{N}(0, \sigma^2)$$

The coefficients of each model are identical to those in TGF3, with the addition of β_1, which we set to 0.4.

Sample size

Given that in practical applications in economics and particularly in finance, sample sizes are relatively small, and noise high, we wish to investigate the performance of each model variant at generating accurate prediction intervals across a range of sample sizes. For each model variant, data-generating function, and noise-generating function, we conduct experiments with four different sample sizes.

$$N = 200, 600, 1000, 2000$$

With the more complicated data-generating functions (especially when combined with the non-homoskedastic noise-generating functions), the small sample size of 200 will ensure that the networks are not able to discover $f(x)$ precisely, giving an idea of performance of each method in the presence of bias.

Note that the sample size of, for example, $N = 200$, refers to the data size before splitting into train, validation and test datasets. We use a train/validation/test ratio of 0.7/0.15/0.15 for all experiments. As a result, the smallest train set we experiment with has 140 observations, for which we have a validation and test set size of 30 observations.

Hyperparameter Selection

For each data generating function, and for each sample size, we carry out a hyperparameter selection procedure. The aim of this procedure is to find a set of hyperparameters which results in a good performance on out of sample data. The hyperparameter options we experiment with are shown in Tables 2.2 and 2.3, for the neural network and recurrent neural network respectively.

Table 2.2: Neural network hyperparameter options

Hyperparameter	Options
Learning rate	[0.01, 0.1]
Network size	[(10,), (10, 10), (30,), (30, 30, 30), (50, 50), (100,)]
Activation function	[Sigmoid, Tanh]
Weight initialization	[(truncated_normal, 0.1), (truncated_normal, 0.01)]

In Table 2.2, the “learning rate" refers to size of the gradient descent update, described in Section 2.2.3. The “network size" gives the number of hidden layers and the number of nodes in each of these layers. For example, a network size of (10, 10) refers to two hidden layers,

each of size 10. The “activation function" is the non-linearity applied after each hidden layer. Finally, the “weight initialization" refers to the distribution from which we randomly initialize the weights of the network prior to training. We use a truncated normal distribution with a standard deviation of 0.1 or 0.01, and values greater than two standard deviations away from zero resampled. There are also a number of additional hyperparameter options (generally less significant for ensuring accurate performance), which we hold fixed. These are described in Appendix B.1.

Table 2.3: Recurrent neural network hyperparameter options

Hyperparameter	Options
Number of steps	[3, 5, 10]
Layers	[1, 2]
Hidden size	[10, 20]
Activation function	[Relu, Sigmoid]
Learning rate	[0.01, 0.1, 0.001, 0.0001]
Target lags	[(1,), (1, 2), (1, 2, 3)]

Table 2.3 lists the options which we experiment with for the RNN. The “number of steps" refers to how many previous timesteps we unroll the network and backpropagate through during optimization. “Layers" refers to the number of hidden layers, and “hidden size" is the number of nodes in each of these layers. The “target lags" is the number of previous timesteps observations, $y_{(t-1,\ldots,t-n)}$, which we use as features at each timestep. As with the feedforward neural network, there are a number of additional hyperparameters which we hold constant, described in Appendix B.1.

The hyperparameter selections works as follows. We simulate a single train and validation set from the data generating function, with the same number of observations which we will evaluate on (i.e. [200, 600, 1000, 2000]). For each possible combination of the hyperparameters described above, we optimize a network on the train sample to predict the mean of the output distribution, and record the loss on the held-out validation sample. We then choose the set of hyperparameters which achieves the lowest loss on the validation sample. This set of hyperparameters is then used to train and compare networks with all combinations of output layers, described in Section 2.2.4.

2.3.2 Empirical Experiments

The aim of the empirical experiments is to complement the simulation study by providing results on real-world datasets. Given that in this case we do not have access to the true

quantiles, we instead split each dataset into a train, validation and test set. After optimization using the train and validation data, we calculate the quantile loss(es) on the test dataset.

Datasets

We use returns and volatility time series provided the Oxford Mann Institute's Realized Library (Heber et al. [2009]). For both time series, the frequency is daily, and we take the period from 03/01/2000 to 04/12/2017, giving around 4300 observations per index. We use returns of the following 18 indices: DJIA, Nasdaq 100, S&P 500, Russel 2000, CAC 40, FTSE 100, DAX, All Ordinaries, Hang Seng, Nikkei 225, KOSPI Composite Index, AEX Index, Swiss Market Index, IBEX 35, IPC Mexico, Bovespa Index, Euro STOXX 50, FTSE MIB.

The returns data is daily close-to-close percentage returns. We fit the main neural network (the mean prediction) to attempt to predict the returns directly, and as such, do not expect any significant predictability, Bossaerts and Hillion [2015]; Goyal and Welch [2002]; Welch and Goyal [2007]. However, we are interested in the accuracy of the network's estimates of the quantiles of the distribution, where we expect to see a greater degree of information, Cenesizoglu and Timmermann [2008].

As a secondary set of experiments, we train the main network to estimate a proxy for the realized volatility of each index. Andersen and Bollerslev [1998] introduced realized variance measures based on intra-day data which have become the dominant proxy for the latent volatility process; here, we use the measure "Realized Variance", developed by Andersen et al. [2003] and Barndorff-Nielsen and Shephard [2002] and defined as:

$$RV_t = \sum x_{j,t}^2$$

where

$$x_{j,t} = X_{t_{j,t}} - X_{t_{j-1,t}}$$

and $t_{j,t}$ are the times of trades or quotes on the t_{th} day. If prices are observed without noise, then as $min_j|t_{j,t} - t_{j-1,t}| \to 0$ the measure consistently estimates the quadratic variation of the price on the t_{th} day. To mitigate microstructure noise, the measure is based on a subset of 5-minute returns data.

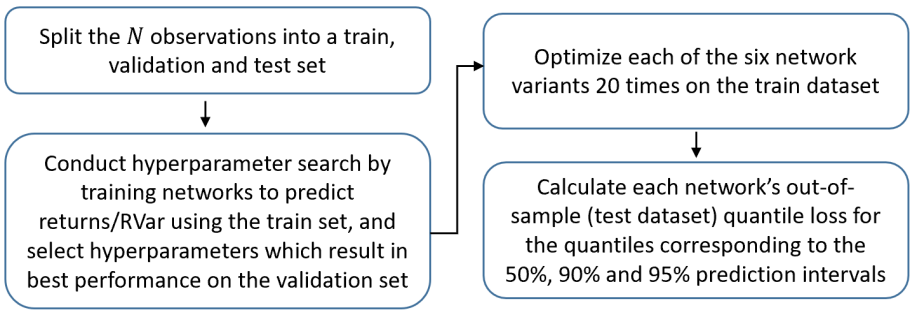
In this latter case, the quantiles become estimates of the volatility of volatility, in which there remains some interest in providing accurate predictions, Huang et al. [2019].

Training

For the empirical experiments, the experimental procedure is summarized in Figure 2.7. We first split each index into a train, validation and test set, using the ratios 0.7, 0.15, and 0.15 respectively (which corresponds to approximate dataset sizes of 3000, 650, 650). We then

carry out a hyperparameter selection step, using the same set of candidate hyperparameters as in the simulation experiments, described in Section 2.3.1. For each set of hyperparameters, the network is optimized on the training data, and the set of parameters which achieves the lowest loss on the validation data[10] selected.

Figure 2.7: Summary of empirical experimental steps



We then optimize 20 networks with this set of hyperparameters on the training data of the index in question, with early-stopping based on the validation data, and report the quantile losses achieved across a set of quantiles on the test data, which has been held out until this point.

2.4 Simulation Results

2.4.1 Description of statistics

The aim of the simulation is to measure the performance of each network variant in predicting the true prediction interval. Given that we know the data generating function, $f(x)$, and the noise generating function, $\epsilon(x)$, we can calculate the quantiles precisely. In the case where $\epsilon_i(X_i) \sim \mathcal{N}(0, \sigma^2)$ (i.e. we have homoskedastic noise) the 0.95 quantile is simply:

$$q(x)_{0.95} = f(x) + \sigma Z_{95}$$

We make equivalent calculations for the 0.025, 0.05, 0.25, 0.75 and 0.975 quantiles. In the cases of heteroskedastic noise generating functions, we calculate the quantiles at the X values in the out-of-sample set. We can then compare the estimates of the networks of each of these quantiles with the true values. Recall that for each dataset, we also drew an out-of-sample set of size K ($K = 300$ in practice). For each simulated dataset, we calculate the predictions of each network at each point in K, and then use the mean average error to compare these

[10] We use the loss on the mean prediction task (returns and realized variance respectively) as opposed to the quantile losses when selecting hyperparameters.

estimates with the true percentiles. In the case of the 0.95 quantile, for the s_{th} simulated dataset:

$$d(x_{ks})_{0.95} =\mid q(x_{ks})_{0.95} - \hat{q}(x_{ks})_{0.95} \mid$$

$$L^s_{0.95} = \frac{1}{K}\sum_{k=1}^{K} d(x_{ks})_{0.95}$$

where $d(x_{ks})_{0.95}$ denotes the absolute difference for the 95th percentile in the s_{th} simulated dataset, for the k_{th} observation.

Note that an accurate prediction of the quantile from the networks in the case of additive quantile models requires an accurate estimate of the mean and/or median, as well as the percentiles, though we focus in this paper on the accuracy of the quantile estimates only.

The result we present in the results tables is simply the average of the MAE across all simulation draws, S:

$$L_{0.95} = \frac{1}{S}\sum_{s=1}^{S} L^s_{0.95} \tag{2.2}$$

We also calculate two measures of the variance of the networks' predictions. Firstly, we present the standard deviation of the error, $d(x)_{0.95}$ across all observations in a dataset, and all repetitions, S.

$$\sigma^r_{0.95} = \sqrt{\frac{1}{S}\frac{1}{K}\sum_{s=1}^{S}\sum_{k=1}^{K}(d(x_{ks})_{0.95} - \bar{d}(x)_{0.95})^2} \tag{2.3}$$

This gives an indication of the consistency of each network variant's predictions across repetitions, and across different input features, x_i, which is of particular interest for the heteroskedastic noise generating functions. We refer to this measure as SD1.

Secondly, we present the standard deviation across the repetitions, S.

$$\sigma^s_{0.95} = \sqrt{\frac{1}{S}\sum_{s=1}^{S}(L^s_{0.95} - \bar{L}_{0.95})^2} \tag{2.4}$$

This isolates the variation in performance across samples. This is of particular relevance for neural network optimization, as different network architectures may converge less consistently (or not at all) across different samples relative to others. We refer to this standard deviation as SD2.

2.4.2 Description of Tables

We present Tables in two formats. Firstly, for the feedforward neural network and data generating function 3 only, we show the results in full in Tables 2.4, 2.5 and 2.6. For each of the other data generating functions, we present the full results Tables in Appendix B.2 only. Table 2.4 displays the results for this data generating function with homoskedastic noise (NGF1), Table 2.5 presents the results with heteroskedastic noise (NGF2), and Table 2.6 shows the results with skewed noise (NGF3). Within each table, each row corresponds to one network variant, which were summarized in Table 2.1. The top line of the column headers represents the sample size, ranging from 200 to 2000. The second line of the column headers shows the quantiles we are predicting. We group the results of symmetric quantiles together. For example, the 0.025 quantile is grouped with the 0.975 quantile, with that column representing the accuracy of a prediction of the 95% prediction interval. We display results for three prediction intervals in total, corresponding to the 95%, 90% and 50% intervals.

Within each table cell, the first (unbracketed) number gives the mean average error of the predictions for this quantile. This corresponds to Equation 2.2 above, and is referred to as the MAE from this point onwards. Our main conclusions are based on this result. The second number, in rounded brackets, gives the standard deviation of the errors across both observations and repetitions. This corresponds to Equation 2.3 above, and is denoted SD1. The final number, in square brackets, gives the standard deviation of the error across samples, corresponding to Equation 2.4 above, and is referred to as SD2. Within each column, we present the lowest results for each of the MAE, SD1 and SD2 in bold, giving a clear indication of the best performing network variant across each measure.

The second style of results table allows broader conclusions across data generating functions. In Table 2.7, for example, we show combined results for each data generating function. These Tables show the average rank of each network variant (each row) across the different simulations[11]. A rank of 1.00 would denote that this network variant performed best out of all network variants in **all** simulations. A rank of 8.00 would denote that this network variant performed worst out of all variants across **all** simulations.

The second number, in circular brackets, is the average distance between each network variant and the best performing network variant for this sample size and prediction interval. A value of 0.0 would denote that this network was always the best-ranked variant. A low number denotes that this network variant consistently ranks best, or if not, is at least very close to the best network in terms of accuracy of the prediction interval. A high value indicates that this network consistently outputs significantly less accurate prediction intervals than the best-performing variant.

[11] i.e. across each data generating function and each noise generating function, for a total of 15 repetitions

2.4.3 Main Results (FeedForward Neural Networks)

There are four main conclusions for the feedforward neural network, based on Tables 2.4, 2.5, 2.6 and 2.7.

Result 1: Networks which use the quantile loss function to generate prediction interval estimates consistently outperform the Nix and Weigend [1994] approach

This is most clearly seen in Table 2.7, in which the Nix Weigend (nw) network consistently ranks lowest, with an average ranking across all simulations, prediction intervals and sample sizes of 7.33. In practice, we found that the "nw" network would often fail to converge to a good equilibrium at all, with optimization of the log-likelihood loss function produces less consistent gradient updates than the quantile methods. This can be seen in Table 2.4, where the "nw" networks often don't converge to a more accurate prediction interval even as the sample size increases. The standard deviation across samples (in square brackets) is consistently larger than all other network variants, illustrating this effect. For some samples, the method converges well, giving accurate prediction intervals, but in others, it barely converges at all, giving the large standard deviation. This effect could likely be mitigated to some extent with careful tuning of hyperparameters for each sample. However, the broad conclusion is that the Nix Weigend approach produces less accurate and more volatile prediction interval estimates than the quantile approaches.

Result 2: "Joint" optimization is outperformed by "sep" optimization

That is, "joint" models, in which the weights of the main network are optimized using both the quantile loss functions and the mean prediction loss, perform consistently worse than "sep" models, in which we optimize the weights of the main network using the mean prediction loss only, and optimize the quantile output layer weights using their individual quantile losses. In Table 2.7, we can see that the highest ranking model is always one of the three "sep" models. The "sep" models are also more consistent in their convergence to strong prediction intervals, with their deviation from the best-performing model (in circular brackets) also consistently lower. This can also be seen in Tables 2.4, 2.5 and 2.6, with the standard deviation across samples (SD2), in square brackets, lower for the "sep" models on average than for the "joint" models.

This conclusion is to some extent unsurprising. It is a common pattern across neural network optimization that training the same weights with multiple loss functions[12] makes optimization more difficult, as the weights are being pulled in separate directions concurrently. The "sep" approach avoids this problem, by allocating each weight in the network to a

[12]The same patter is observed in the literature on Generative Adversarial Networks (see Goodfellow et al. [2014])and Multi-Task training (see Zhang and Yang [2017].

single loss function only for optimization; the main network weights are optimized using the mean prediction loss only, and each quantile output layer's weights are optimized using their individual quantile loss function only. The introduction of this style of training is one of the key novel contributions of this work, with the technique improving accuracy of the prediction models over Rodrigues and Pereira [2018], where they present the "joint" approach only.

Result 3: Additive prediction intervals outperform independent prediction intervals

The third main conclusion is that modelling prediction intervals as the mean prediction plus/minus a value, as opposed to independently, stabilizes the prediction interval estimates across all experiments, and in the vast majority, improves their accuracy as well. These results are denoted with an "_a", and we compare in each case (joint and sep) to the baseline for this method. In other words, we compare "joint_a" with "joint", and "sep_a" with "sep". In Table 2.7, we can see that the additive results for both "joint" and "sep" are almost always lower than the baselines, for both the ranks and differences from the best prediction. This effect is particularly prominent for the 0.025 and 0.975 quantile predictions, and for the lower sample sizes. Predicting these quantiles with a low sample size is particularly difficult, as there is limited information in the tails. The mean prediction is likely to be more accurate, and this provides a firmer base prediction which the additive methods can take advantage of, whereas the baseline approach has to predict the quantile independently. This is the second key novel contribution of this work.

Result 4: Residual connections stabilize convergence and improve results on average

The fourth conclusion is that a residual connection further improves the accuracy and consistency of predictions in the majority of cases. This can be seen both in Tables 2.4, 2.5 and 2.6, where the residual results (denoted with an "_r") generally provide the most accurate predictions and in Table 2.7. In Table 2.7, although the residual results do not always rank highest, they consistently give the lowest deviation from the best model (in circular brackets). That is, even if they are not the best model, they are very close to it in terms of accuracy. Or put another way, adding the residual connection very rarely hurts performance in any significant fashion.

We can see that for the heteroskedastic and skewed noise, the addition of a residual connection is more important. This is an expected results; the weights of the main network are optimized partially (joint) or entirely (sep) to predict the mean. Whether or not the noise is homoskedastic or symmetric is not significant for the mean prediction, and the main network weights may consequently ignore that information entirely. Adding a residual connection allows the quantile estimates to attend to this information directly, improving

accuracy. This is the first time that the use of a residual connection has been explored for generating prediction intervals, and represents the third novel contribution of this work.

Putting the above conclusions together, it is clear that the best performing network overall is the separately optimized output layers, with both the additive approach and a residual connection, denoted "sep_a_r". It achieves an average rank of 2.43 in Table 2.7, and is the lowest ranking model across six of the twelve sample and prediction interval sizes. More conclusively, it achieves an average deviation from the best performing model of just 0.06, compared to 0.29 for its closest competitor. In other words, even if it doesn't rank first, it is very close in terms of accuracy to the best prediction. Across every combination of sample sizes and prediction intervals, it achieves the lowest deviation from the best network, by a large margin. As a result, it is recommended as a safe option for producing accurate prediction intervals, regardless of sample size, prediction interval size, and complexity of the data generating and noise generating functions.

Main Conclusions (Recurrent Neural Networks)

The results for the recurrent neural networks are presented in Table 2.8, which gives a summary of the ranks across all experiments, with the full tables provided in Appendix B.2. The main conclusions from Table 2.8 largely consolidates main results 2, 3, and 4 from above. [13].

Result 2: "Joint" optimization is outperformed by "sep" optimization

As with the feedforward network, the "sep" models, which train the weights of the main network using the mean prediction loss only, outperform the "joint" optimization case in all but one of the columns (corresponding to dataset size and prediction interval). The one exception to this is with a sample size of 200, when predicting the 95% prediction interval, where the "joint_a" model ranks highest on average. However, we note that despite its higher average rank, it's average difference to the best-performing model of 1.15 is significantly higher than the "sep_a" model, which has an average difference of just 0.06. This indicates that although the "joint" model performs top in many cases, it is far less consistent than the "sep" models, and often converges to a poor equilibrium. This is a conclusion that is mirrored across all columns, with the average deviation from the best-performing model (shown in circular brackets) lower for all 36[14] of the "sep" models than their "joint" counterparts.

[13] We do not evaluate the Nix Weigend method on the time series data, as it relies on i.i.d. data, so main result 1 is based only on the feedforward network

[14] 12 columns with three separate results for each of the "joint" and "sep" model types in each column

Result 3: Additive prediction intervals outperform independent prediction intervals

Result 3, that additive prediction intervals perform more strongly than independent intervals, is also strongly supported in the RNN case. In Table 2.8, the "joint_a" and "sep_a" models outperform their baseline counterparts ("joint" and "sep" respectively) in the large majority of the cases. As is the case with the above conclusion, this result is more strongly seen in the average deviation from the best-performing model than it is in the average ranks. The difference is especially exaggerated for the outer quantiles (the 0.025 and 0.975 quantiles), where the additive approach stabilizes training dramatically. With a sample size of 200 we see a drop for the "sep" model from 1.41 to 0.02, and with a sample size of 600 from 2.36 to 0.04. This conclusion continues to hold even for the larger sample sizes. As with the feedforward network then, the introduction of additive quantiles significantly stabilizes training of the prediction intervals, and very rarely impacts performance in a negative way.

Result 4: Residual connections stabilize convergence and improve results on average

Result 4, that residual connections improve performance and stabilize training, is less clearly supported for the recurrent neural network. In only 11 of 24 cases does the residual and additive model (denoted _a_r) outperform the additive case (denoted _a) in terms of relative rank, with three ties. The story is similar for the average deviation from the mean, with the residual method only improving results in 11 cases, and doing worse in 13.

This result is to some extent expected. Whilst a feedforward neural network extracts information from multiple inputs through stacked layers (meaning there is potentially a separation of multiple layers between the input data and the output layers, which the residual connection bypasses), a recurrent neural network consolidates information by updating a vector representation of the time series at each point in time. As a result, recurrent neural networks generally perform better with fewer layers (in our experiments, networks with a single layer performed better in 85% of cases during the model selection step). This diminishes the need for a residual connection, which provides direct access to the input data for the output layer, as the output layer is close to the input data regardless.

Additional conclusions

As an additional result, we consider the performance of the individual models, denoted "ind", in which a completely separate network is trained for each quantile prediction, with no shared information. This method consistently outperforms the Nix and Weigend [1994] approach, and performs in general on par with the "joint" method, with no residual or additive adaptations. However, it is clear from Table 2.7 that this approach generates less accurate prediction

intervals than the “sep" method. It is particularly weak for the outer quantiles (0.025 and 0.975), due to the scarcity of information in the tails. This mirrors the conclusion from the additive approach, where for outer quantiles, utilizing information from the mean prediction, directly or indirectly, proves useful in stabilizing results. Given that training a separate neural network for each quantile prediction is also significantly more computationally expensive, the broad conclusion is that it is best to stick with the combined approaches.

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	1.894 (1.052) [0.556]	1.593 (0.900) [0.467]	0.683 (0.462) [0.191]	2.441 (1.060) [0.423]	2.049 (0.924) [0.358]	0.847 (0.556) [0.160]	2.433 (0.928) [0.287]	2.042 (0.819) [0.245]	0.846 (0.531) [0.121]	2.365 (0.863) [0.276]	1.985 (0.766) [0.235]	0.821 (0.513) [0.111]
ind	1.026 (0.717) [0.129]	0.986 (0.684) [0.115]	0.703 (0.520) [0.224]	0.998 (0.720) **[0.063]**	0.973 (0.685) **[0.051]**	0.597 (0.496) [0.237]	0.997 (0.718) **[0.072]**	0.965 (0.681) **[0.074]**	0.541 (0.473) [0.221]	0.994 (0.720) **[0.038]**	0.963 (0.679) **[0.027]**	0.469 (0.428) [0.171]
joint	0.693 (0.513) [0.164]	0.607 (0.463) [0.156]	0.461 (0.372) [0.119]	0.643 (0.504) [0.178]	0.556 (0.482) [0.190]	0.473 (0.409) [0.168]	0.568 (0.432) [0.111]	0.466 (0.386) [0.081]	0.384 (0.334) **[0.032]**	0.557 (0.413) [0.097]	0.458 (0.380) [0.064]	0.395 (0.338) **[0.019]**
joint_a	0.646 (0.543) [0.241]	0.612 (0.530) [0.242]	0.537 (0.448) [0.224]	0.590 (0.554) [0.263]	0.569 (0.543) [0.263]	0.521 (0.446) [0.228]	0.479 (0.460) [0.192]	0.456 (0.434) [0.172]	0.423 (0.379) [0.160]	0.497 (0.468) [0.199]	0.476 (0.454) [0.192]	0.440 (0.388) [0.166]
joint_a_r	0.688 (0.581) [0.260]	0.647 (0.553) [0.254]	0.561 (0.465) [0.230]	0.617 (0.574) [0.272]	0.593 (0.555) [0.267]	0.537 (0.448) [0.228]	0.499 (0.486) [0.227]	0.476 (0.467) [0.218]	0.436 (0.390) [0.188]	0.532 (0.509) [0.240]	0.508 (0.488) [0.227]	0.467 (0.409) [0.197]
sep	0.655 (0.507) [0.179]	0.569 (0.451) [0.172]	0.392 (0.306) [0.091]	0.593 (0.474) [0.183]	0.464 (0.391) [0.146]	0.329 (0.304) [0.096]	0.532 (0.437) [0.172]	0.387 (0.351) [0.123]	0.290 (0.284) [0.102]	0.459 (0.394) [0.159]	0.335 (0.317) [0.108]	0.252 (0.260) [0.109]
sep_a	0.459 (0.361) [0.108]	0.423 (0.339) [0.094]	**0.372** (0.294) **[0.075]**	0.386 (0.335) [0.092]	0.364 (0.323) [0.093]	0.335 (0.294) [0.097]	**0.322** (0.297) [0.111]	**0.304 (0.283)** [0.106]	**0.279 (0.260)** [0.100]	0.272 (0.259) [0.111]	0.253 (0.245) [0.108]	0.235 (0.221) [0.102]
sep_a_r	**0.456 (0.354) [0.104]**	**0.420 (0.332) [0.085]**	0.373 **(0.292)** [0.090]	**0.376 (0.320)** [0.093]	**0.356 (0.310)** [0.094]	**0.323 (0.283) [0.095]**	0.333 **(0.296)** [0.111]	0.311 (0.286) [0.108]	0.286 (0.261) [0.100]	**0.254 (0.239)** [0.105]	**0.241 (0.230)** [0.102]	**0.221 (0.208)** [0.098]

Table 2.4: Results for DGF3, with NGF1 (homoskedastic noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	0.911 (0.781) [0.244]	0.767 (0.660) [0.204]	0.332 (0.297) [0.083]	1.057 (0.610) [0.205]	0.887 (0.516) [0.172]	0.366 (0.233) [0.072]	1.029 (0.557) [0.136]	0.864 (0.470) [0.115]	0.355 (0.212) [0.050]	1.099 (0.550) [0.093]	0.923 (0.464) [0.079]	0.379 (0.205) [0.036]
ind	0.852 (0.683) [0.194]	0.620 (0.592) [0.226]	0.217 (0.250) [0.073]	0.706 (0.626) [0.177]	0.377 (0.428) [0.130]	0.130 (0.165) [0.044]	0.663 (0.593) [0.123]	0.322 (0.413) [0.080]	0.106 (0.134) [0.031]	0.615 (0.580) [0.075]	0.288 (0.399) [0.056]	0.090 (0.115) [0.024]
joint	0.300 (0.285) [0.106]	0.232 (0.233) [0.078]	0.177 (0.183) [0.063]	0.190 (0.182) [0.055]	0.138 (0.135) [0.029]	0.106 (0.101) [0.019]	0.154 (0.153) [0.040]	0.113 (0.112) [0.023]	0.085 (0.079) **[0.013]**	0.142 (0.145) [0.036]	0.100 (0.112) [0.016]	0.083 (0.091) [0.013]
joint_a	0.218 (0.205) [0.069]	0.195 (0.192) [0.057]	0.157 (0.145) [0.044]	0.142 (0.133) [0.037]	0.126 (0.123) [0.029]	0.109 (0.107) [0.022]	0.109 (0.101) **[0.022]**	0.099 (0.094) **[0.018]**	0.084 (0.082) [0.016]	0.094 (0.101) **[0.015]**	0.088 (0.097) **[0.013]**	0.079 (0.089) [0.012]
joint_a_r	0.223 (0.214) [0.075]	0.201 (0.198) [0.064]	0.162 (0.151) [0.047]	0.146 (0.135) [0.040]	0.130 (0.123) [0.034]	0.109 (0.102) [0.023]	0.110 (0.104) [0.025]	0.100 (0.096) [0.022]	0.083 (0.080) [0.016]	0.097 (0.105) [0.017]	0.089 (0.098) [0.014]	0.078 **(0.084)** [0.010]
sep	0.463 (0.483) [0.241]	0.307 (0.347) [0.185]	0.169 (0.165) [0.055]	0.269 (0.308) [0.169]	0.148 (0.156) [0.072]	0.096 (0.094) [0.016]	0.175 (0.178) [0.062]	0.109 (0.115) [0.038]	0.078 (0.080) [0.015]	0.157 (0.149) [0.038]	0.096 (0.101) [0.018]	0.075 (0.086) [0.011]
sep_a	0.198 (0.178) **[0.044]**	0.178 (0.162) [0.043]	0.145 (0.129) [0.044]	0.142 (0.138) [0.039]	0.118 (0.117) [0.033]	**0.095** (0.096) [0.018]	0.113 (0.114) [0.039]	0.093 (0.094) [0.028]	**0.075** (0.077) **[0.013]**	0.092 (0.102) [0.023]	**0.080** (0.092) [0.014]	**0.069** (0.086) **[0.009]**
sep_a_r	**0.184 (0.163)** [0.046]	**0.167 (0.146) [0.038]**	**0.144 (0.128) [0.038]**	**0.129 (0.119) [0.029]**	**0.114 (0.109) [0.023]**	**0.095 (0.093) [0.015]**	**0.102 (0.099)** [0.029]	**0.091 (0.089)** [0.021]	**0.075 (0.076) [0.013]**	**0.089 (0.093)** [0.020]	0.081 **(0.090)** [0.017]	**0.069** (0.085) [0.010]

Table 2.5: Results for DGF3 with NGF2 (heteroskedastic noise)

	200 (0.025, 0.975)	200 (0.05, 0.95)	200 (0.25, 0.75)	600 (0.025, 0.975)	600 (0.05, 0.95)	600 (0.25, 0.75)	1000 (0.025, 0.975)	1000 (0.05, 0.95)	1000 (0.25, 0.75)	2000 (0.025, 0.975)	2000 (0.05, 0.95)	2000 (0.25, 0.75)
nw	1.667 (0.783) [0.336]	1.400 (0.673) [0.289]	0.583 (0.362) [0.152]	0.859 (0.520) [0.173]	0.722 (0.439) [0.145]	0.299 (0.199) [0.061]	0.894 (0.480) [0.145]	0.751 (0.406) [0.122]	0.310 (0.185) [0.055]	0.882 (0.444) [0.084]	0.742 (0.375) [0.070]	0.306 (0.170) [0.035]
ind	1.147 (0.860) [0.111]	1.065 (0.829) [0.107]	0.471 (0.555) [0.307]	0.767 (0.672) [0.181]	0.561 (0.596) [0.191]	0.151 (0.256) [0.060]	0.735 (0.639) [0.156]	0.474 (0.538) [0.160]	0.137 (0.265) [0.065]	0.704 (0.637) [0.149]	0.436 (0.523) [0.157]	0.107 (0.211) [0.051]
joint	0.790 (0.674) [0.199]	0.555 (0.528) [0.239]	0.228 (0.324) [0.230]	0.173 (0.182) [0.062]	0.130 (0.140) [0.036]	0.093 (0.092) [0.020]	0.141 (0.146) [0.042]	0.103 (0.110) [0.024]	0.081 (0.078) [0.015]	0.115 (0.127) [0.038]	0.083 (0.097) [0.018]	0.069 (0.079) [0.012]
joint_a	0.359 (0.487) [0.326]	0.322 (0.464) [0.321]	0.294 (0.385) [0.275]	0.125 (0.119) [0.032]	0.106 (0.106) [0.027]	0.088 (0.083) [0.020]	0.107 (0.106) [0.026]	0.092 (0.095) [0.019]	0.079 (0.078) [0.016]	0.085 (0.092) [0.019]	0.074 (0.088) [0.015]	0.066 (0.079) [0.011]
joint_a_r	0.404 (0.563) [0.376]	0.371 (0.532) [0.362]	0.324 (0.415) [0.292]	0.125 (0.124) [0.036]	0.108 (0.110) [0.030]	0.089 (0.082) [0.020]	0.105 (0.103) [0.023]	0.093 (0.094) [0.018]	0.079 (0.075) [0.015]	0.088 (0.099) [0.021]	0.078 (0.093) [0.016]	0.069 (0.079) [0.013]
sep	0.959 (0.772) [0.144]	0.666 (0.617) [0.259]	0.176 (0.142) [0.031]	0.141 (0.142) [0.050]	0.108 (0.109) [0.035]	0.080 (0.076) [0.016]	0.113 (0.111) [0.036]	0.090 (0.090) [0.020]	0.070 (0.069) **[0.009]**	0.091 (0.103) [0.029]	0.074 (0.083) [0.015]	0.059 (0.074) [0.008]
sep_a	0.208 **(0.189)** [0.066]	**0.192 (0.168) [0.049]**	0.161 **(0.130)** [0.027]	0.108 (0.093) [0.025]	0.094 (0.083) [0.018]	0.077 **(0.071)** [0.016]	0.092 (0.081) [0.021]	0.077 **(0.073)** [0.014]	**0.066 (0.066) [0.009]**	0.077 (0.082) [0.017]	0.065 (0.078) [0.010]	**0.056 (0.072)** [0.008]
sep_a_r	**0.207 (0.189) [0.060]**	0.193 (0.170) [0.050]	**0.159** (0.131) **[0.025]**	**0.091 (0.080) [0.017]**	**0.085 (0.078) [0.015]**	**0.074 (0.071) [0.015]**	**0.079 (0.077) [0.016]**	**0.073 (0.073) [0.013]**	**0.066** (0.068) [0.010]	**0.066 (0.077) [0.011]**	**0.060 (0.076) [0.009]**	**0.056 (0.072) [0.007]**

Table 2.6: Results for DGF3 with NGF3 (skewed noise)

	200 (0.025, 0.975)	200 (0.05, 0.95)	200 (0.25, 0.75)	600 (0.025, 0.975)	600 (0.05, 0.95)	600 (0.25, 0.75)	1000 (0.025, 0.975)	1000 (0.05, 0.95)	1000 (0.25, 0.75)	2000 (0.025, 0.975)	2000 (0.05, 0.95)	2000 (0.25, 0.75)	Average
nw	7.07 (2.55)	7.33 (2.55)	7.67 (1.57)	7.00 (3.64)	7.13 (3.63)	7.40 (2.24)	7.13 (4.42)	7.13 (4.27)	7.73 (2.48)	7.33 (5.89)	7.40 (5.93)	7.60 (3.45)	7.33 (3.55)
ind	5.67 (0.96)	5.53 (0.84)	4.67 (0.38)	6.00 (1.63)	6.27 (1.59)	4.80 (0.67)	5.67 (1.55)	5.60 (1.10)	5.40 (0.35)	6.13 (2.97)	5.47 (2.96)	4.80 (1.04)	5.50 (1.34)
joint	6.20 (0.94)	5.87 (0.77)	4.53 (0.29)	5.73 (1.21)	5.60 (1.25)	5.07 (0.87)	6.20 (0.94)	6.27 (0.76)	5.53 (0.37)	5.27 (1.74)	5.13 (1.75)	5.27 (1.00)	5.56 (0.99)
joint_a	4.13 (0.50)	4.33 (0.48)	4.87 (0.36)	3.87 (0.83)	4.33 (0.90)	5.33 (0.68)	4.27 (0.56)	4.47 (0.52)	5.40 (0.33)	4.07 (1.04)	4.47 (1.09)	5.13 (0.73)	4.56 (0.67)
joint_a_r	4.40 (0.42)	4.60 (0.40)	4.87 (0.29)	4.53 (0.78)	4.60 (0.80)	5.53 (0.56)	4.33 (0.44)	4.87 (0.40)	4.80 (0.24)	4.20 (0.65)	4.73 (0.61)	5.00 (0.37)	4.70 (0.50)
sep	3.67 (0.61)	3.40 (0.46)	**2.87** (0.14)	4.40 (0.60)	3.67 (0.44)	**2.53** (0.17)	3.93 (0.55)	3.33 (0.42)	2.87 (0.15)	4.20 (1.05)	3.80 (0.86)	3.20 (0.40)	3.49 (0.49)
sep_a	**2.20** (0.20)	**2.20** (0.20)	3.47 (0.14)	**2.20** (0.30)	**2.13** (0.31)	2.80 (0.17)	**2.20** (0.33)	2.20 (0.32)	2.20 (0.13)	2.47 (0.55)	2.60 (0.56)	2.60 (0.31)	2.44 (0.29)
sep_a_r	2.67 **(0.10)**	2.73 **(0.10)**	3.07 **(0.09)**	2.27 **(0.07)**	2.27 **(0.06)**	**2.53 (0.04)**	2.27 **(0.07)**	**2.13 (0.06)**	**2.07 (0.03)**	**2.33 (0.06)**	**2.40 (0.05)**	**2.40 (0.02)**	**2.43 (0.06)**

Table 2.7: Ranks

	200 (0.025, 0.975)	200 (0.05, 0.95)	200 (0.25, 0.75)	600 (0.025, 0.975)	600 (0.05, 0.95)	600 (0.25, 0.75)	1000 (0.025, 0.975)	1000 (0.05, 0.95)	1000 (0.25, 0.75)	2000 (0.025, 0.975)	2000 (0.05, 0.95)	2000 (0.25, 0.75)	Average
ind	6.75 (2.17)	6.75 (2.09)	4.00 (1.20)	6.75 (3.74)	6.75 (2.45)	4.00 (1.00)	6.50 (0.66)	6.50 (0.36)	4.50 (0.21)	6.75 (4.50)	7.00 (3.97)	4.50 (0.84)	5.90 (1.93)
joint	4.25 (1.94)	3.75 (1.88)	4.25 (1.52)	3.75 (2.53)	3.50 (2.19)	4.75 (2.15)	5.00 (0.50)	4.50 (0.26)	5.50 (0.26)	4.00 (2.56)	3.50 (1.68)	6.75 (1.19)	4.46 (1.56)
joint_a	**2.50** (1.15)	3.25 (1.29)	4.75 (1.18)	3.25 (1.45)	2.75 (1.39)	5.75 (1.47)	4.00 (0.20)	5.00 (0.20)	6.00 (0.29)	3.25 (0.63)	4.25 (0.57)	5.50 (0.63)	4.19 (0.87)
joint_a_r	3.50 (1.18)	4.00 (1.29)	4.50 (1.13)	4.00 (1.46)	4.50 (1.43)	6.00 (1.53)	3.75 (0.21)	4.25 (0.18)	5.25 (0.23)	3.25 (0.45)	3.75 (0.34)	4.75 (0.28)	4.29 (0.81)
sep	4.50 (1.41)	4.50 (1.27)	**3.00** (0.76)	5.75 (2.36)	5.75 (1.60)	3.00 (0.60)	5.00 (0.27)	4.25 (0.12)	**2.25** (0.04)	4.75 (2.29)	4.75 (0.92)	2.75 (0.10)	4.19 (0.98)
sep_a	3.75 (0.06)	3.50 (0.10)	4.25 (0.15)	**2.25 (0.04)**	**2.00 (0.04)**	**1.75 (0.02)**	2.25 (0.04)	**1.50 (0.01)**	**2.25 (0.02)**	**2.75 (0.05)**	**2.25 (0.02)**	2.00 (0.04)	2.54 (0.05)
sep_a_r	2.75 **(0.02)**	**2.25 (0.03)**	3.25 **(0.07)**	**2.25** (0.07)	2.75 (0.06)	2.75 (0.03)	**1.50 (0.01)**	2.00 (0.02)	**2.25 (0.02)**	3.25 (0.08)	2.50 (0.03)	**1.75 (0.02)**	**2.44 (0.04)**

Table 2.8: RNN Ranks

2.5 Empirical Results

2.5.1 Description of statistics

For both the returns and realized variance predictions, we present three statistics. Firstly, the out-of-sample (test dataset) quantile loss, as described in Equation 2.1, evaluated at each of the six quantiles [0.025, 0.05, 0.25, 0.75, 0.95, 0.975]. For each index, Y, we average the quantile loss across all repetitions, R, with R set to 20 for all experiments.

$$L^Y_{0.95} = \frac{1}{R}\sum_{r=1}^{R} L^Y_{r,0.95}$$

We then average this result across all eighteen indices:

$$L_{0.95} = \frac{1}{Y}\sum_{y=1}^{Y} L^y_{0.95}$$

As with the simualation results, we average the results for the quantiles corresponding to the 75%, 90% and 95% prediction intervals together.[15] The quantile loss results are presented in the first three columns of Tables 2.9, 2.10, 2.11 and 2.12.

For each out-of-sample quantile loss, we also present the standard deviation across repetitions, R, which, as with the quantile loss, we then average across indices:

$$\sigma^Y_{0.95} = \sqrt{\frac{1}{R}\sum_{r=1}^{R}(L^Y_{r,0.95} - \bar{L}^Y_{0.95})^2}$$

$$\sigma_{0.95} = \frac{1}{Y}\sum_{y=1}^{Y} \sigma^y_{0.95}$$

We present the the value of σ_q in brackets in each table cell to the right of the values of L_q.

Finally, to evaluate the consistency of each network variant across the eighteen indices[16] we order the quantile loss of each network variant for each index, and report the rank, z, of each variant in this ordered list:

$$z^y_{0.95} = rank(sorted([L^{y,ind}_{0.95}, L^{y,joint}_{0.95}, L^{y,joint\_a}_{0.95}...))$$

where $L^{y,ind}_{0.95}$ is the quantile loss of the "ind" network variant for the 95th quantile and index y. We then average these ranks across all indices, and in the usual pairs of quantiles

[15] For example, we average $L_{0.05}$ and $L_{0.95}$ for the 90% interval).

[16] We wish to ensure that the average results described above are not driven by a single network variant performing very strongly on a few indices, but weakly on average.

corresponding to prediction intervals. A rank of 1.0 indicates the network variant performed best for all indices for this quantile, with a rank of 7.0 indicating it performed worst in all cases.

2.5.2 Main Results

As the conclusions from the empirical experiments for the feedforward neural network and recurrent neural network largely mirror each other, we discuss them jointly. Instead, we split the discussion along data lines, presenting the Realized Variance results first, in Table 2.9 for the NN and Table 2.10, which largely consolidate the simulation results, and the Returns results second, in Tables 2.11 and 2.12, which are more mixed.

Realized Variance Predictions

The predictions of the realized variance, and the associated quantiles of the realized variance indices, support Results 2, 3, and 4 from the simulation experiments, presented in Section 2.4.3.

Result 2: “Joint" optimization is outperformed by “sep" optimization

For both the feedforward NN in Table 2.9 and RNN in Table 2.10 we see a substantial improvement in the quantile loss when moving from joint optimization of the network weights to the separate approach, in which the main weights are optimized for the mean prediction only. This result is consistent for the Realized Variance predictions for both network types, and across all three prediction intervals, and can be seen in both the lower quantile loss values in the first three columns, and the lower average ranks in the last three columns. We also see a consistently lower standard deviation of results for the “sep" models, reflecting that convergence of the networks is smoother when each network parameter is optimized against one loss function only.

Table 2.9: Test dataset quantile loss and ranks for predictions of Realized Variance with a feedforward neural network

	Quantile Loss			**Ranks**		
	(0.025, 0.975)	**(0.05, 0.95)**	**(0.25, 0.75)**	**(0.025,0.975)**	**(0.05,0.95)**	**(0.25,0.75)**
ind	0.082 (0.063)	0.110 (0.073)	0.204 (0.057)	6.0	5.4	4.0
joint	0.122 (0.072)	0.157 (0.090)	0.267 (0.080)	5.7	6.4	6.2
joint_a	0.117 (0.067)	0.141 (0.072)	0.271 (0.075)	4.9	5.3	6.8
joint_a_r	0.111 (0.063)	0.133 (0.067)	0.251 (0.071)	4.4	4.1	5.1
sep	0.074 (0.055)	0.093 (0.059)	0.173 (0.045)	3.6	3.3	**1.9**
sep_a	0.064 (0.037)	0.082 (0.046)	0.169 (0.035)	2.2	1.9	2.2
sep_a_r	**0.057** (0.033)	**0.079** (0.045)	**0.168** (0.037)	**1.2**	**1.6**	**1.9**

Result 3: Additive prediction intervals outperform independent prediction intervals

The results for Realized Variance also support Result 3 from the simulation, that the additive approach, in which the prediction of each quantile is modelled as the mean prediction plus/minus an offset, provides more accurate prediction intervals than the traditional approach in which we predict each quantile directly. In 11 of the 12 cases where we move from a baseline model ("joint" or "sep") to an additive model ("joint_" or "sep_a"), we see a drop in both the quantile loss and the average rank. There is one exception to this result, with the average quantile loss for the (0.25, 0.75) quantiles increasing slightly from 0.267 to 0.271 in Table 2.9 when we move from the "joint" model to the "joint_a" model. We also see an associated increase in the average rank for these quantiles, from 6.2 to 6.8, indicating that this result is not driven by a single index. The less significant improvement of the additive approach for narrower prediction intervals is to some extent intuitive; the predictions of quantiles closer to the median are likely to be more similar to the network's prediction of the mean, and as a consequence there is less to be gained from the additional flexibility of the additive approach.

The advantage of the additive models is less clearly evident for the recurrent neural network in Table 2.10 than it is for the feedforward network in Table 2.9. This is likely due to the superior results for the RNN generally (we see consistently lower quantile loss scores for the RNN across all model variants) meaning there is less unexploited information in the quantiles for the additive approach to take advantage of.

Result 4: Residual connections stabilize convergence and improve results on average

The evidence to support Result 4, that residual connections (denoted by "_r") generally improve accuracy and reduce variance, is similar, with a more pronounced effect being seen in Table 2.9 for the NN than in Table 2.10 for the RNN. We also see a generally stronger improvement when moving from the "joint_a" to the "joint_a_r" models than we do when moving from the "sep_a" to the "sep_a_r" models.

Table 2.10: Test dataset quantile loss and ranks for predictions of Realized Variance with a recurrent neural network

	Quantile Loss			Ranks		
	(0.025, 0.975)	**(0.05, 0.95)**	**(0.25, 0.75)**	**(0.025,0.975)**	**(0.05,0.95)**	**(0.25,0.75)**
ind	0.060 (0.041)	0.083 (0.048)	**0.168** (0.040)	3.3	3.0	2.0
joint	0.101 (0.068)	0.130 (0.078)	0.248 (0.075)	6.6	6.7	6.9
joint_a	0.098 (0.061)	0.125 (0.069)	0.240 (0.066)	6.2	6.1	6.0
joint_a_r	0.098 (0.054)	0.117 (0.059)	0.197 (0.043)	4.9	4.8	4.1
sep	0.059 (0.042)	0.081 (0.048)	0.171 (0.041)	2.6	2.6	3.0
sep_a	0.058 (0.040)	0.081 (0.047)	0.170 (0.039)	**2.2**	2.5	**2.9**
sep_a_r	**0.057** (0.039)	**0.080** (0.046)	0.170 (0.039)	**2.2**	**2.4**	3.1

Returns Predictions

The predictions of returns, presented for the feedforward neural network in Table 2.11 and for the recurrent neural network in Table 2.12 provide less clear conclusions across network variants. Although the "sep_a_r" model does perform best across all six columns in Table 2.11, its advantage over the second best model is always very marginal. For the recurrent neural network, in Table 2.12, we actually see the "ind" model (in which a separate neural network is optimized for each individual quantile prediction) performing best on average, especially at the more extreme quantiles. We also see a less obvious advantage for the additive models (denoted with an "_a") over the baselines, alongside no obvious difference between the models with a residual connection (denoted with an "_r").

Table 2.11: Quantile Loss and Ranks for predictions of Returns with a feedforward neural network

	Quantile Loss			Ranks		
	(0.025, 0.975)	**(0.05, 0.95)**	**(0.25, 0.75)**	**(0.025,0.975)**	**(0.05,0.95)**	**(0.25,0.75)**
ind	0.0650 (0.0042)	0.1067 (0.0063)	0.2877 (0.0090)	3.3	3.5	3.4
joint	0.0905 (0.0123)	0.1232 (0.0122)	0.2909 (0.0101)	4.8	4.5	5.5
joint_a	0.0973 (0.0106)	0.1286 (0.0106)	0.2912 (0.0097)	5.6	5.2	5.6
joint_a_r	0.0974 (0.0109)	0.1285 (0.0109)	0.2908 (0.0098)	5.6	5.4	4.8
sep	0.0649 (0.0042)	0.1066 (0.0065)	0.2876 (0.0089)	2.9	3.1	2.9
sep_a	0.0649 (0.0044)	0.1066 (0.0068)	0.2877 (0.0092)	3.1	3.4	3.2
sep_a_r	**0.0647** (0.0044)	**0.1064** (0.0062)	**0.2875** (0.0088)	**2.8**	**2.8**	**2.5**

The intuitive explanation for this shift in results is the difficulty in predicting daily returns. The forecastability of returns has been questioned in the literature, Bossaerts and Hillion [2015]; Goyal and Welch [2002], with positive results tending to be isolated to longer term forecasts (monthly, quarterly or annual returns), Welch and Goyal [2007]. Although we are reporting results for the predictions of the quantiles only (i.e. we are not concerned with which network, if any, is able to forecast the returns the most accurately), it remains the case the

all the “joint" and “sep" models are based around a neural network which has been optimized to predict returns. As the noise-to-signal ratio for this prediction problem is very high, the main network weights will likely converge poorly. Despite there being more predictability in the quantiles, the quantile predictions then have to compensate for the noisy mean network, which in turn harms their performance.

Instead, for the “ind" models, in which we train a separate neural network for each quantile, we ignore the mean function (returns) entirely. As a result, we get a cleaner convergence of the models to a better equilibrium, and more accurate forecasts on average (most obviously the case for the recurrent neural network in Table 2.12).

Table 2.12: Quantile Loss and Ranks for predictions of Returns with a recurrent neural network

	Quantile Loss			**Ranks**		
	(0.025, 0.975)	**(0.05, 0.95)**	**(0.25, 0.75)**	**(0.025,0.975)**	**(0.05,0.95)**	**(0.25,0.75)**
ind	0.0640 (0.0057)	**0.1019** (0.0077)	**0.2861** (0.0100)	**2.2**	**1.6**	2.7
joint	0.0828 (0.0134)	0.1200 (0.0123)	0.2916 (0.0106)	5.8	6.1	6.2
joint_a	0.0769 (0.0102)	0.1155 (0.0100)	0.2909 (0.0103)	5.3	5.4	5.2
joint_a_r	0.0779 (0.0100)	0.1159 (0.0099)	0.2909 (0.0106)	5.3	5.3	4.9
sep	0.0638 (0.0042)	0.1058 (0.0065)	0.2867 (0.0094)	3.2	3.2	2.8
sep_a	0.0638 (0.0046)	0.1058 (0.0070)	0.2868 (0.0096)	3.4	3.4	**2.5**
sep_a_r	**0.0636** (0.0047)	0.1058 (0.0068)	0.2873 (0.0094)	2.8	2.9	3.7

2.6 Summary

A common criticism of neural network forecasts is that they provide no prediction or confidence intervals. In this paper, we have proposed the use of the quantile loss function for generating prediction intervals for feedforward and recurrent neural networks. In addition, we have presented three novel network variants specifically for prediction intervals. Firstly, an approach in which the weights of the main network are optimized as normal for the mean prediction task, and the weights of each quantile output layer are optimized independently. Secondly, the modelling of each quantile additively relative to the mean prediction, and thirdly, the use of a residual connection, allowing each quantile output layer to attend to the original feature matrix.

Through a combination of a simulation study and an evaluation on financial time series, we have demonstrated that prediction intervals generated via the quantile loss function overcome many of the shortcomings of previous approaches in the literature which rely on an assumption of normality, and outperform these approaches in terms of accuracy. We have also shown that all three of our proposed network variants improve accuracy on average. We caveat this final result in the case of prediction problems, such as of returns, where there is more predictability

in the tails of the distribution than for the mean, and suggest that in such cases optimizing an individual neural network for each quantile may be preferable.

Although this paper represents a step towards more accurate prediction intervals for neural networks, we emphasize that there remains considerable work to understand the theory behind network convergence, and to put bounds on the accuracy of predictions, both for the mean and the quantiles of the distribution.

Bibliography

Andersen, T. G. and Bollerslev, T. (1998). Answering the skeptics: Yes, standard volatility models do provide accurate forecasts. *International Economic Review*, 39(4):885–905.

Andersen, T. G., Bollerslev, T., Diebold, F. X., and Labys, P. (2003). Modeling and forecasting realized volatility. *Econometrica*, 71(2):579–625.

Barndorff-Nielsen, O. E. and Shephard, N. (2002). Econometric analysis of realized volatility and its use in estimating stochastic volatility models. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 64(2):253–280.

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, 31(3):307 – 327.

Bossaerts, P. and Hillion, P. (2015). Implementing Statistical Criteria to Select Return Forecasting Models: What Do We Learn? *The Review of Financial Studies*, 12(2):405–428.

Cenesizoglu, T. and Timmermann, A. (2008). Is the distribution of stock returns predictable? *SSRN Electronic Journal*.

Cho, K., van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, pages 103–111, Doha, Qatar. Association for Computational Linguistics.

Engle, R. (1982). Autoregressive conditional heteroskedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, 50(4):987–1007.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2672–2680.

Goyal, A. and Welch, I. (2002). Predicting the equity premium with dividend ratios. Working Paper 8788, National Bureau of Economic Research.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

He, Y. and Li, H. (2018). Probability density forecasting of wind power using quantile regression neural network and kernel density estimation. *Energy conversion and management*, 164:374–384.

He, Y., Xu, Q., Wan, J., and Yang, S. (2016). Short-term power load probability density forecasting based on quantile regression neural network and triangle kernel function. *Energy*, 114:498–512.

Heber, Gerd, Lunde, A., Shephard, N., and Sheppard, K. (2009). Oxford-man institute's realized library version:0.2. *Oxford-Man Institute, University of Oxford*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

Huang, D., Schlag, C., Shaliastovich, I., and Thimme, J. (2019). Volatility-of-volatility risk. *Journal of Financial and Quantitative Analysis*, 54(6):2423–2452.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

Koenker, R. and Bassett, G. (1978). Regression quantiles. *Econometrica*, 46(1):33–50.

Nix, D. and Weigend, A. (1994). Estimating the mean and variance of the target probability distribution. *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, pages 55–60.

Pradeepkumar, D. and Ravi, V. (2017). Forecasting financial time series volatility using particle swarm optimization trained quantile regression neural network. *Applied Soft Computing*, 58:35–52.

Rodrigues, F. and Pereira, F. (2018). Beyond expectation: Deep joint mean and quantile regression for spatio-temporal problems. *arXiv:1808.08798v1 [stat.ML]*.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages 1139–1147.

Taylor, J. W. (2000). A quantile regression neural network approach to estimating the conditional density of multiperiod returns. *Journal of Forecasting*, 19(4):299–311.

Welch, I. and Goyal, A. (2007). A Comprehensive Look at The Empirical Performance of Equity Premium Prediction. *The Review of Financial Studies*, 21(4):1455–1508.

Yan, X., Zhang, W., Ma, L., Liu, W., and Wu, Q. (2018). Parsimonious quantile regression of financial asset tail dynamics via sequential learning. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31*, pages 1575–1585. Curran Associates, Inc.

Zhang, W., Quan, H., and Srinivasan, D. (2018). An improved quantile regression neural network for probabilistic load forecasting. *IEEE Transactions on Smart Grid*, PP:1–1.

Zhang, Y. and Yang, Q. (2017). A survey on multi-task learning. *CoRR*, abs/1707.08114.

Chapter 3

Does bootstrapping neural networks work?

Neural networks are becoming an increasingly popular model for prediction in a wide variety of settings. To approximate standard errors for the network's predictions, bootstrapping is a common approach. Early work established the validity of this method asymptotically, under a set of restrictive assumptions. However, due to a limitation in compute power when the initial work was developed, there remains very little understanding of how bootstrapping performs in limited sample-size settings in practice. In particular, there is no evidence that bootstrapping is able to approximate the uncertainty introduced by random weight initialization and sample splitting, two areas which cannot be accounted for in closed-form estimators. We present the first in-depth simulation study to answer this question, and show that broadly speaking, bootstrapping is effective at providing correctly sized standard errors, with coverage of the true function converging towards the correct levels as sample sizes increase. We extend the analysis to Recurrent Neural Networks (RNNs), presenting the first experiments with the block bootstrap for neural networks, and show that the results from the I.I.D. case generally carry over, although convergence to correct coverage is slower. Finally, we include details of a novel computational architecture, which speeds up the training of multiple small neural networks on a GPU by a factor of 40 or more, allowing efficient application of bootstrapping in practice.

3.1 Introduction

Neural networks and sample uncertainty

When presented with a prediction problem, it is common practice in machine learning to try a range of different models, and select the one which achieves the lowest error on a held-out dataset. Neural networks are just one of the options available. However, they have gained significant notoriety[1] in recent years for their ability to approximate extremely complicated functions given a large enough dataset, leading to significant breakthroughs in the fields of image, speech and text recognition. Although the networks used for these applications are incredibly large, involving millions of weights, the researcher has the flexibility to set the size, and indeed structure, of the networks. As such, they are also effective in smaller scale settings, and have been used successfully in a number of the papers mentioned above, including Xiong

[1]This notoriety has also led to a massive surge in research related to neural networks, including novel network architectures, and advanced optimization techniques. Much of this research can be taken advantage of in the slightly smaller data settings in which economists tend to work.

et al. [2015] and Jason Hartford and Taddy [2017]. There is also a vast applied literature outside of economics, including health, finance and computer science, which makes use of neural networks in their various guises.

Given their widespread application, the importance of establishing accurate estimates of uncertainty around network's predictions is self-evident to an econometric audience. This is especially true if networks may be used for policy decisions, in healthcare or elsewhere, but is equally applicable in the forecasting literature. Despite this, there has been relatively little work to characterize either sample uncertainty (including confidence intervals) for the predictions from a neural net, or indeed prediction intervals.[2] An early comparative study of the accuracy of different methods for producing error estimates for neural networks is provided in Tibshirani [1996]. The paper finds that bootstrapping outperforms the closed form "sandwich" and "delta" methods, as it can capture the additional variability in network predictions due to the random starting weights. Pass [1992] also includes a small simulation study on bootstrapping neural networks. However, due to limitations in compute power at the time of these papers[3], the size of the simulations are limited, with only between 30 and 50 bootstrap draws taken. There is also no consideration of model selection effects.

Franke and Neumann [2000] is the most comprehensive coverage of bootstrapping neural networks to date, and adds some theoretical validity to the earlier works. Consistency results are established for bootstrap estimates of the distribution of network weight estimates. The paper also includes a slightly larger scale simulation study, which demonstrates the effectiveness of the theoretical results. Since that point a large number of papers, including Dybowski and Roberts [2001] and Heskes [1997], have made use of bootstrapping for estimating the distribution of neural network predictions, yet extensions to the core understanding of the theory or accuracy of the method have been limited.

Contribution

This paper aims to provide a comprehensive understanding of how bootstrapping neural networks performs in practice. In its basic form, this simply involves extending the scale of earlier simulation studies, to a size enabled by significant advances in compute power. This is further enabled by a novel training architecture for neural networks, allowing optimization of 40 networks in the time it previously took to optimize one. We also present the first results extending the application of bootstrapping to recurrent neural networks (for time series data).

On a secondary level, we expand the analysis to include two techniques central to modern day neural network optimization, which are not covered by the theoretical results. Firstly,

[2] A prediction interval is wider than a confidence interval, as it includes an estimate of the noise/irreducible error at that point.

[3] 1995 and 1992 respectively.

we consider the effects of model selection. In practice, a model selection[4] step is essential for providing accurate neural network predictions. This paper is the first to investigate the potential effects this has on the distribution of a networks predictions, and bootstrapping's ability to account for this. Secondly, effective neural network optimization requires splitting of the data into separate train and validation sets to avoid overfitting. The impact of this on the distribution of predictions is unknown, and we provide results to demonstrate if bootstrapping is effective in capturing it.

As additional contributions, we provide results indicating the relative importance of the three sources of variance in neural network predictions; sample uncertainty, sample splitting, and random weight initialization.[5] This has significant implications for the most effective method of reducing the variance of network's predictions, through ensemble techniques. We also provide the first comparison for neural networks of the pairs and wild bootstrap approaches for I.I.D. data, and of the simple and overlapping block bootstrap methods for time series data.

3.2 Experimental Setup

Consider the setting where we have an independent and identically distributed set of observations $(X_i, Y_i), i = 1, ..., N$, where X_i is of dimension d, with unknown distribution function F. We are interested in modelling the conditional expectation of Y_i, given $X_i = x$:

$$f(x) = E[Y_i \mid X_i = x]$$

Approximation of $f(x)$ is done with a feedforward neural network, as described below. We describe the optimal network function, $m_0(x)$, as the best approximation to $f(x)$ which is possible with a neural network for a given level of noise and sample size N. We wish to approximate the sampling distribution of estimates of the optimal network function, $\hat{m} - m_0$.

Note that the optimal network function, $m_0(x)$ will differ systematically from the true conditional expectation function, $f(x)$, if the sample size is small, and the optimal network includes bias.[6] However, as N increases, $m_0(x)$ will converge to $f(x)$. In large sample settings, we can then regard estimated confidence bounds as approximating confidence bounds around the true conditional expectation function.

[4]Model selection will be discussed in more detail later, but refers to choice of network size, number of layers, learning rate etc.

[5]See section 3.2.3

[6]Neural networks aim to balance the bias/variance trade-off to achieve optimal out-of-sample prediction, so non-zero bias is expected in small sample settings.

3.2.1 Example Model

We introduce an example model, which is used throughout the paper for graphical illustration of the methodology. In the experiments, this model is just one of five models used to test performance of bootstrapping feedforward neural networks, with a further three models used for recurrent neural nets.

To allow plotting of the results, we take a one-dimensional input vector, X, $X_i \sim Unif([2, 6])$ and a true function of the form:

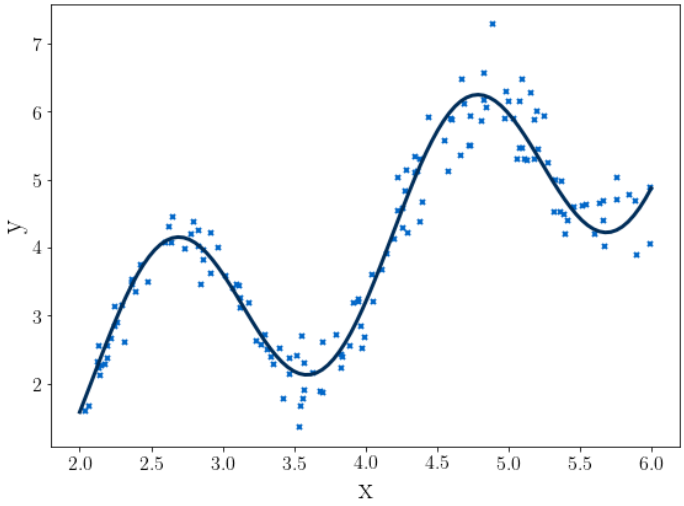
$$f(X_i) = \frac{3}{2} sin(3X_i) + X_i$$

We then generate 150 observations from the following formula:

$$Y_i = f(X_i) + \epsilon_i(X_i)$$

where $\epsilon_i \sim \mathcal{N}(0, (0.1X_i)^2)$. That is, we have heteroskedasticity, with the variance of the random error increasing linearly with X. An example dataset drawn from this function is illustrated in Figure 3.1.

Figure 3.1: The example function, with a dataset of 150 points



The remaining models used in the experiments are introduced below, in Section 3.2.7.

3.2.2 Neural Networks

Feedforward Neural Networks

A feedforward neural network takes an input X_i (a vector or matrix of explanatory variables) and returns a scalar or vector, $\hat{Y}_i \in \mathbb{R}^{d_y}$.[7]

Given an input, x, the output, $h_{\theta,l}(x) \in \mathbb{R}^{d_l}$ of the l_{th} layer of the network is defined as

$$h_{\theta,l}(x) = g_l(W_l h_{\theta,l-1}(x) + b_l) \qquad l = 1, .., L$$

where $W_l \in \mathbb{R}^{d_l \times d_{l-1}}$ are referred to as the "weights" of the network and $b_l \in \mathbb{R}^{d_l}$ is referred to as the "bias". The input to layer one of the network is a vector of explanatory variables, $h_{\theta,0} = x$. The function $g^l(z)$ is a non-linear transformation, which is applied to all outputs of the layer individually, $g^l(z) = (\sigma(z_1), \sigma(z_2), ..., \sigma(z_{d_l}))$ with $z_1, z_2, ..., z_{d_l} \in \mathbb{R}$. The choice of σ is typically the sigmoid function, tanh or recitified linear unit (RELU).[8]

The final (output) layer of the network is a linear regression without a non-linearity, producing the networks prediction, $m_\theta(x) \in \mathbb{R}^{d_y}$:

$$m_\theta(x) = W_o h_{\theta,L}(x) + b_o$$

.

where $W_o \in \mathbb{R}^{d_L}$ and $b_o \in \mathbb{R}$.

Figure 3.2: Feedforward Neural Network with a single hidden layer, and four inputs (features)

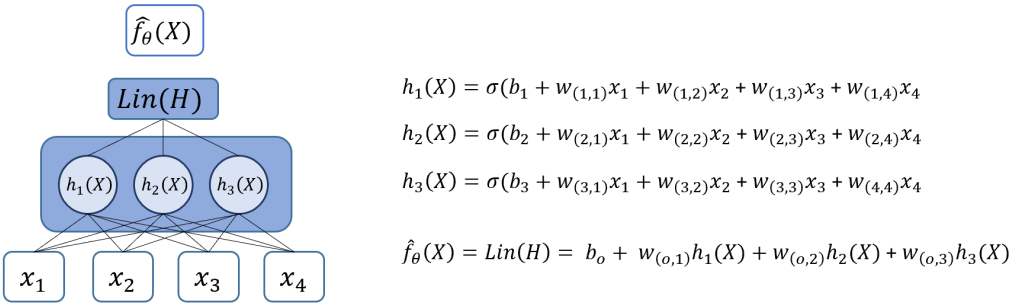


Figure 3.2 shows a feedforward NN with a feature vector X_i of dimension 4, and a single hidden layer. The activation of the illustrated network is the sigmoid function.

The weights of the network, $\theta = (W_1, ..., W_L, b_1, ..., b_L, W_o, b_o)$ can be optimized using a range of different algorithms, including gradient descent and quasi-Newton methods.

[7] $\hat{Y}_i$ can also represent a probability distribution, as with logistic regression, though we limit the analysis to scalar values of $\hat{Y}_i$ in this paper, as is more common in economic prediction models.

[8] The RELU is defined as $f(x) = max(0, x)$

Recurrent Neural Networks

Consider a time series, $\{x_1, ..., x_t, ..., x_T\}$. In the basic RNN, the output of the network at time t, $h_t(x_{\hat{t}}) \in \mathbb{R}^{D_h}$ (where $x_{\hat{t}}$ refers to $(x_t, x_{t-1}, ..., x_1)$ and D_h is the dimension of the hidden layer of the network) can be described as:

$$h_t(x_{\hat{t}}) = g(W_h h_{t-1}(x_{\hat{(t-1)}}) + W_x x_t + b_h)$$

where $W_h \in \mathbb{R}^{D_h \times D_h}, W_x \in \mathbb{R}^{D_h \times d_x}$ and $b \in \mathbb{R}^{D_h}$. The weights matrices, W_h and W_x are not time dependent, with the same weights matrices being used at each timestep. The output $h_t(x_{\hat{t}})$ is termed the "hidden state" of the network, and can be thought of as holding information from previous timesteps. This information is then combined with the new input, x_t, to give our prediction at this timestep. The hidden state needs to be initialised at timestep zero, with the usual choice of a vector of zeros being used in this paper.

The remainder of the network is identical in format to the basic NN. The function $g(z)$ is a non-linear transformation, which is applied to all outputs of the layer individually, $g(z) = (\sigma(z_1), \sigma(z_2), ..., \sigma(z_{D_h}))$. The output layer is a linear regression without a non-linearity, which produces our estimate of volatility for the next period, $f_\theta(x) \in \mathbb{R}$:

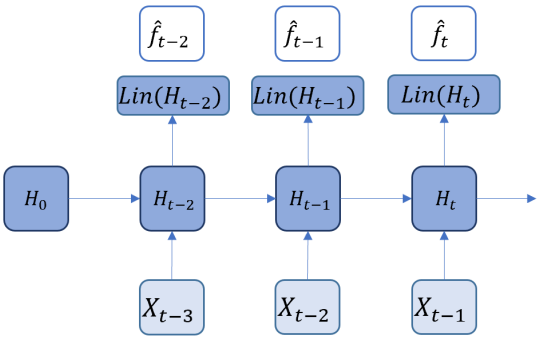
$$f(x_{\hat{t}}) = W_o h_t(x_{\hat{t}}) + b_o$$

.

where $W_o \in \mathbb{R}^{D_h}$ and $b_o \in \mathbb{R}$.

Diagrammatically, the RNN is best represented in "rolled out" form, although the process is in reality a continuous cycle of arbitrary length.

Figure 3.3: Single layer Recurrent Neural Network rolled out over three timesteps



At each point in time, the network makes a predictions for the next value in the time series based on the previous observation, x_{t-1}, and the hidden state of the network. It is possible to stack multiple layers of RNN cells, with the inputs to the next layer generally being the hidden states of the previous layer, as opposed to the inputs, x_t. Residual connections,

introduced by Kim et al. [2017], whereby the higher hidden layers have access to the inputs as well as the output of the previous hidden layer, are also commonly used in practice.

Given that the process is recurrent, theoretically, the RNN is able use information from every previous timestep in the current prediction, even as t grows very large. In practice, the use of gradient descent for learning the optimum weight values combined with the structure of the network prohibits this. As we are using the same weights matrices at every timestep, the gradients of the cost function relative to the values of these matrices will be multiplied together many times. If the gradient is large in value this leads to exploding gradients, or if it is small to vanishing gradients. During the training phase, we therefore have to limit the number of previous timesteps which the algorithm can use information from, generally to ten or under.

In order to try and overcome this problem, Hochreiter and Schmidhuber [1997] introduced a key innovation to the RNN cell, termed the Long Short-Term Memory Network, or LSTM. The basic structure (where prediction is made using the hidden state from previous timesteps, and the new input at this timestep, x_t), remains the same, with the adaptations being made to the method by which we combine these two sources of information. The idea is to keep some "memory" of inputs from time-steps in the past, and for the input at the current step to determine the extent to which this memory influences the current prediction. The equations which decide how much information of each source is used or passed on are termed "gates". In the LSTM, we define:

$$
\begin{aligned}
&Input\,gate : i_t = \sigma(W_i x_t + U_i h_{t-1}) \\
&Forget\,gate : f_t = \sigma(W_f x_t + U_f h_{t-1}) \\
&Output\,gate : u_t = \sigma(W_c x_t + U_c h_{t-1}) \\
&New\,memory\,cell : \tilde{c}_t = tanh(W_c x_t + U_c h_{t-1})
\end{aligned}
$$

$$
\begin{aligned}
&Final\,memory\,cell : c_t = f_t c_{t-1} + i_t \tilde{c}_t \\
&Final\,hidden\,state : h_t = u_t tanh(c_t)
\end{aligned}
$$

where σ is a sigmoid function. Given that the LSTM is used more frequently than the simple RNN in practice, we include simulations for this more complex cell structure as well, presenting the first results which explore the use of bootstrapping for LSTMs.

Optimization

Optimization of neural network is an iterative procedure, based on the gradient of the loss function with respect to the network weights. Broadly speaking, optimization algorithms

can be divided into two groups: those which use only the first order derivative of the loss function to calculate the update step (which we will call “gradient descent" based methods), and “quasi-Newton" methods, which use both the first and second order derivatives. In practice, gradient descent based methods are more common, and are used throughout this paper, although robustness results are included in the appendix for “quasi-newton" methods. To avoid overfitting, the dataset is split into three: a training set (75% of the data in this paper), a validation set (15%) and a test set (15%). Gradient descent is performed on the training set, and the out-of-sample error monitored on the validation set. Once training is complete, the model weights are restored to their values at the epoch which achieved the lowest validation loss. As the validation set is also used for model selection, the final estimate of the accuracy of the network is based on the test set. This error is calculated only once, after all optimization and model selection steps.

The basic gradient descent algorithm can be described in pseudo-code as follows:

randomly initialize the weights of the model, θ
for epoch in number of epochs:
    for batch in number of batches:
        evaluate the gradient of the loss function $L(\theta)$ with respect to
        the weights, ∇_θ
        update the weight values: $\theta_{new} = \theta_{old} - \alpha \nabla_\theta C(\theta)$
restore model weights to epoch with lowest validation loss

The choice of loss function is open to the researcher. The most commonly used for scalar outputs is the mean squared error, a trend which we follow in this paper:

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} [Y_i - \hat{m}(X_i)]^2$$

3.2.3 Sample uncertainty for neural networks

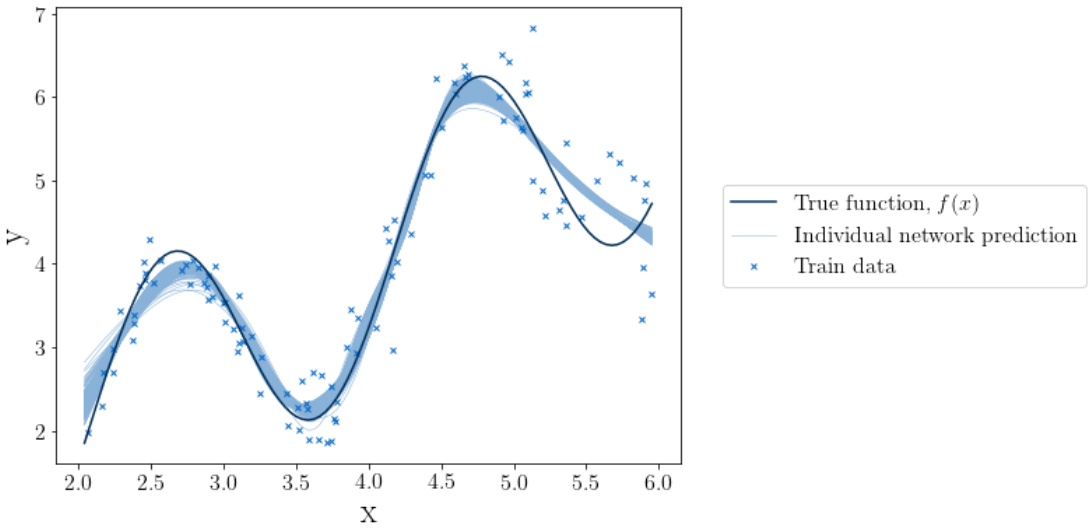
Due to the uncertainty introduced by random weight initialization and model selection, a closed form solution for the sampling distribution does not exist. Instead, we approximate the correct sampling distribution by defining a known population generating function, $f(X)$, and a noise generating function, $\epsilon(X)$. This allows us to take draws of the target variable, Y_i, using the following formula

The combination of sample uncertainty with the optimization procedure for neural networks produces three distinct sources of variability in the predictions. In order for bootstrap estimates of confidence intervals to be correct, all three must be considered.

1. Network weights initialization

As noted in Tibshirani [1996], the necessary random initialization of the network weights θ before optimization begins, combined with the non-convexity and potential for local minima in the loss function, introduces variability into network predictions. That is, even if we optimize a neural network on precisely the same training data, with an identical model structure, the weights will converge to marginally different values each repetition, due to the random initialization. To demonstrate this effect, we train 500 different networks on exactly the same dataset, using exactly the same network structure. Figure 3.4 displays the results.

Figure 3.4: Networks with identical parameters and data, optimized with different random initialization of the network weights

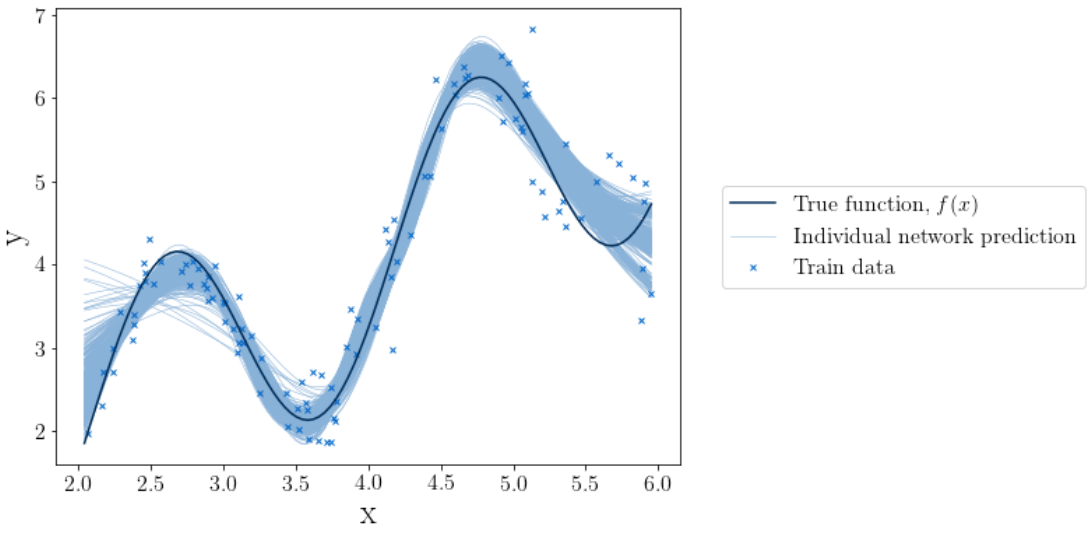


As can be seen, there is significant variation in the networks predictions, particularly at the turning points of the function. This variability is one of the key reasons why closed form solutions for neural network confidence intervals underestimate observed variation, as found in Tibshirani [1996]. The results section includes an analysis of the potential size of this error, which will clearly vary considerably based on the sample size, complexity of the data generating function, and network parameters (in particular, initializing the weights of the network from a distribution with a larger variance will, in general, increase the variance of the final predictions). As the bootstrap procedure includes a different random initialization for each repetition, re-sampling incorporates this effect, providing more accurate intervals.

2. Sample splitting

The second source of uncertainty occurs due to the random splitting of the data into train and validation sets for optimization of the network. With a fixed dataset, different random splits of the data will result in different model selection, and network weights. To demonstrate this effect, Figure 3.5 shows the predictions of 500 networks trained on the same dataset, but with a new random train/validation split before each optimization.

Figure 3.5: Networks with identical parameters and data, optimized with different random splits of the data into train and validation sets



Note that the variation displayed in Figure 3.5 includes both the effects of random weight initialization and sample splitting.[9] To gain an approximate understanding of the relative importance of the two effects, we can subtract the variance at each point due to random weight initialization from the variance due to the combination. The results section includes comment on the relative sizes in practice, and supports the conclusion of LeBaron and Weigend [1994] that sample splitting introduces more variability than random weight initialization, especially in datasets in which the signal to noise ratio is low.

The impact of sample splitting is not accounted for in the theoretical literature, or previous empirical work, and it is consequently unclear if bootstrapping can account for it. Intuitively, by sampling with replacement from the full dataset (i.e. both the training and validation sets combined), and then re-drawing the train and validation sets from the new dataset for each bootstrap repetition, sample splitting uncertainty is incorporated. Evidence for the

[9] Although it is technically possible to initialize the network each time with fixed weights, this would not accurately separate the two effects. As we are optimising the network on a different train dataset, the initial gradients and local minima will be different regardless.

importance of this approach relative to simply bootstrapping the train dataset are included in the robustness section.

3. Sample uncertainty

The third source of uncertainty is the standard sampling variation, which bootstrap is designed to replicate. We do not observe the full population dataset, but rather a single sample from it, (X_i, Y_i) $i = 1, ..., N$. Estimating sample uncertainty refers to attempting to measure how much our model's results would vary if we were to observe a different sample.

In the case of neural networks, all three of the above sources of uncertainty act in combination. The researcher observes only one sample from the true population, and in general, will split the data into a train and validation set just once. If only a single neural network is trained, they similarly observe only one realization of random weight initialization. Through bootstrapping, we hope to estimate the impact of all three sources of uncertainty accurately. The main aim of this paper is to assess whether this is possible in practice.

Model Selection

When optimizing a neural network for prediction on a new dataset, a model selection step is required. A range of network parameters will be experimented with, potentially including but not limited to the optimization algorithm, the number of hidden nodes, the activation function, and the number of layers. The best performing network for the problem is generally then chosen based on performance on a validation data set. As noted in Efron and Hastie [2016], model selection can have a significant impact on the final distribution of predictions. For each hypothetical realization of a dataset, (X_i, Y_i) $i = 1, ..., N$, sample split, and random initialization, the model selection step may result in a different choice of parameters. To illustrate this effect, we take five hundred draws with N = 50 from the example population generating function, introduced in Figure 3.1. We perform a model selection procedure on each of these draws, varying the learning rate, number of hidden nodes, number of layers and activation function, giving a total of 54 different potential parameter combinations.

Table 3.1: Parameters chosen through model selection procedure

activation_fn	learning_rate	num_hidden_nodes	num_layers	**Count**
relu	0.10	10	3	**168**
sigmoid	0.10	50	1	**49**
sigmoid	0.10	50	2	**48**
relu	0.05	10	3	**38**
sigmoid	0.10	50	3	**35**
sigmoid	0.10	10	1	**22**
sigmoid	0.05	50	1	**21**
sigmoid	0.05	50	2	**19**

The first four columns of Table 3.1 indicate the set of parameter options which were chosen by the model selection procedure, and the final column the number of times this particular set was chosen in 500 repetitions. It is clear that the optimum parameters selected are not consistent across different population draws, with a total of 29 out of the 54 options being selected at least once.[10]

The size of the network parameter set included in the model selection step, and hence the associated uncertainty introduced, will vary for each researcher/project. As such, it is impossible to account for analytically, and has consequently been ignored in the literature. We provide the first results which demonstrate the potential size of this effect, and suggest potential adjustments to account for it.

3.2.4 Baselines

Due to the uncertainty introduced by random weight initialization and model selection, a closed form solution for the sampling distribution does not exist. Instead, we approximate the correct sampling distribution by defining a known population generating function, $f(X)$, and a noise generating function, $\epsilon(X)$. This allows us to take draws of the target variable, Y_i, using the following formula:

$$Y_i = f(X_i) + \epsilon_i(X_i)$$

[10]We show only the top 8 sets of options and their counts in Table 3.1

To approximate the true sampling distribution, we generate P datasets of size N from this formula. This is of course infaesible in reality, where the true function $f(X)$ is unknown - the researcher would only observe a single dataset. We then optimize a neural network on each of the P datasets, and record the predictions of the network at an evenly spaced grid of X values, denoted $\hat{m}^p(X_j), p = 1, ..., P$.

In pseudo-code:

For p in range P:

Generate N pairs of observations of X_i and ϵ_i from defined distributions

$e.g. \qquad X_i \sim Unif([0,4]), \qquad \epsilon_i \sim N(0,1)$

Use these pairs to generate N observations Y_i, using the population generating formula:

$$Y_i = f(X_i) + \epsilon_i(X_i)$$

Optimize a neural network using the generated dataset (potentially including a model selection procedure). Store the predictions of the p_{th} network at points on a uniformly spaced grid, $X_j, j = 1, .., J$, denoted $\hat{m}^p(X_j)$.

Estimate quantities of interest based on the stored network functions. For example, we can estimate the "true" standard error at point X_j:

$$se_j^{true} = \left\{ \frac{1}{P-1} \sum_{j=1}^{P} [\hat{m}^p(X_j) - \hat{m}^*(X_j)]^2 \right\}^{1/2} \text{ where } \hat{m}^*(X_j) = \frac{1}{P} \sum_{p=1}^{P} \hat{m}^p(X_j)$$

In practice, as has been discussed, the researcher will often perform a model selection procedure before optimizing a neural network. To account for the additional uncertainty which this introduces, we can perform model selection for each of the P datasets drawn. We denote such baseline results with the superscript $true\_ms$. To isolate the size of the uncertainty

introduced by model selection, we produce a second set of baseline results, in which model selection is done just once on the first dataset drawn, and the same model parameters[11] then used for all remaining $P-1$ draws. We denote these results with the superscript *true*.

We can then estimate the size of the increase in the standard error at point j due to model selection using the formula:

$$ms\_effect = se_j^{true\_ms} - se_j^{true}$$

3.2.5 Bootstrapping

Bootstrapping, introduced by Efron [1979], generates error estimates by sampling many pseudo-datasets, and re-estimating the model on each of these samples. It is based on the principle that we can approximate the true distribution from which our observations (X_i, Y_i) are drawn, by the observed empirical distribution. Samples drawn with replacement from the empirical distribution, should then approximate samples from the true distribution. By training a network on each of the bootstrap samples, we can, in theory, generate confidence intervals for the networks predictions.

I.I.D. Bootstrap Methods

A range of different algorithms have been developed for I.I.D. data based on this principle. Given that, in practice, the majority of applications of neural networks are in cases where heteroskedasticity is likely, we focus on two algorithms which are robust to this: the “pairs" bootstrap, and the “wild bootstrap".

The algorithms are summarized below. In each case, B refers to the number of bootstrap samples generated. Firstly, the pairs bootstrap:

[11] Learning rate, network size, activation function etc.

For b in range B:

Generate a sample of size N, by sampling with replacement from the N data points observations, $\{(X_1, Y_1), (X_2, Y_2), ..., (X_N, Y_N)\}$. Denote the bth sample drawn as: $\{(X_1^{*b}, Y_1^{*b}), (X_2^{*b}, Y_2^{*b}), ..., (X_N^{*b}, Y_N^{*b})\}$

Split the bootstrap sample randomly into a separate train and validation data set

Optimize a neural network using the train and val sets. Store the predictions of the bth network at points on a uniformly spaced grid, $X_j, j = 1, .., J$, denoted $\hat{m}^{*b}(X_j)$.

Estimate quantities of interest based on the stored network functions. For example, we can estimate the standard error at point X_j:

$$se_j^{pairs} = \left\{ \frac{1}{B-1} \sum_1^B [\hat{m}^{*b}(X_j) - \hat{m}^*(X_j)]^2 \right\}^{1/2} \text{ where } \hat{m}^*(X_j) = \frac{1}{B} \sum_{b=1}^B \hat{m}^{*b}(X_j)$$

We denote results derived through the pairs bootstrap approach with the superscript $pairs$.

The wild bootstrap, introduced in Wu [1986], and discussed in the context of nonparametric nonlinear regression in Hardle [1990] and Hardle and Bowman [1979] also allows for heteroskedasticity. In the pairs approach, it is extremely likely that the bootstrap samples will contain duplicate values. The wild bootstrap differs in this regard; through introducing randomly generated noise, each bootstrap sample should contain unique values only. It is described as follows:

Optimize a uniformly consistent estimator for $f(x)$. In this case, we use a neural network optimally chosen for the sample size using a model selection procedure, denoted $\hat{m}(x)$

Approximate the random noise variables, ϵ_i as: $\hat{\epsilon}_i = Y_i - \hat{m}(X_i), \quad i = 1, ..., N$

Store these with their corresponding X vectors, creating the set

$S = \{(X_1, \hat{\epsilon}_1), (X_2, \hat{\epsilon}_2), ..., (X_N, \hat{\epsilon}_N)\}$

For b in range B:

Generate a sample of size N pairs of X vectors and residuals, by sampling with replacement from S, giving $\{(X_1^{*b}, \hat{\epsilon}_1^{*b}), (X_2^{*b}, \hat{\epsilon}_2^{*b}), ..., (X_N^{*b}, \hat{\epsilon}_N^{*b})\}$

Generate N i.i.d. random variables with mean 0 and variance 1, $\mu_1, ..., \mu_N$

Define the bootstrap outputs as $Y_i^{*b} = \hat{m}(X_i^{*b}) + \mu_i \hat{\epsilon}_i^{*b} \quad i = 1, ..., N$

Split the bootstrap sample randomly into a separate train and validation data set

Optimize a neural network using the train and val sets. Store the predictions of the bth network at points on an evenly spaced grid $X_j, j = 1, ..., J$, denoted $\hat{m}^{*b}(X_j)$.

Estimate quantities of interest based on the stored network functions, e.g. se_j^{wild}

Note that by multiplying the residuals by a random variable with mean 0 and variance 1, we alter neither their mean nor their variance. By keeping the residuals in pairs with their corresponding X vector, we allow for dependence of ϵ_i on X_i, or heteroskedasticity. We denote results derived through the wild bootstrap approach with the superscript $wild$.

Time Series Bootstrap Methods

For time series datasets, in which serial correlation and heteroskedasticity is possible, the block bootstrap approach was developed separately by Hall [1985], Kunsch [1989] and Carlstein [1986]. For the block bootstrap, continuous blocks are sampled from the dataset, and tiled back together, maintaining the dependency structure of the time series within each block.

We perform experiments with two versions of the block bootstrap. The first is the "simple block" method, in which the dataset is divided into N/L (where L is the block length) non-overlapping segments. The second is the "overlapping block" method of Kunsch [1989], in which the data is divided into $N - L + 1$ blocks, with a separate block starting at each timestep.

Simple Block:

Split data into N/L non-overlapping blocks, each of length L

Overlapping Block:

Split data into $N - L + 1$ overlapping blocks, each of length L

For b in range B:

Generate a sample of length N by drawing N/L blocks with replacement, and merging the blocks in the order drawn

Split the bootstrap sample into a separate train and validation data set (the last v observations are used as the val set)

Optimize a recurrent network using the train and val sets. Store the predictions of the bth network at points on an evenly spaced grid $X_j, j = 1, ..., J$, denoted $\hat{m}^{*b}(X_j)$.

Estimate quantities of interest based on the stored network functions, e.g. $se_j^{block\_s}$

We denote results with the simple block method using superscript *block_ s*, and the overlapping block with superscript *block_ o*.

3.2.6 Experimental Summary

To estimate the "true" effect of sample uncertainty, we define a population generating function, and draw P datasets of size N from this function. On each of these separate datasets, we optimize a neural network, with and without a model selection step. The distribution of predictions from each of these networks gives us an estimate of the three sources of uncertainty combined. For example, we can calculate the standard error of network predictions at point X_j, denoted se_j^{true} without model selection and $se_j^{true\_ms}$ with.

We then wish to test whether bootstrapping (using just a single dataset - as the researcher would have access to in reality) can generate the same estimates of sampling uncertainty. To do that, we take a single dataset, and bootstrap it B times. We train a neural net on each of the separate bootstrap draws, and from these networks we generate estimates of the standard error at the same point X_j, denoted se_j^{pairs} and se_j^{wild} for the pairs and wild bootstrap methods respectively. We then compare the estimates of the bootstrap standard errors to the true standard errors.[12] If bootstrapping is effective in capturing the three sources of uncertainty for neural networks, the estimates should be identical.

[12] Or indeed, make any comparison between the distributions generated by the population resampling and the bootstrap methods, including 95% confidence intervals, etc.

Table 3.2: Descriptions of different estimates of sample uncertainty generated in experimental procedure

Superscript	Description	Standard error at point j notation
$true\_ms$	"True" estimate derived from P datasets drawn from population generating function, with model selection applied	$se_j^{true\_ms}$
$true$	"True" estimate derived from P datasets drawn from population generating function, without model selection	se_j^{true}
$pairs$	Pairs bootstrap estimate derived from B bootstrap datasets drawn with replacement from a single dataset	se_j^{pairs}
$wild$	Wild bootstrap estimate derived from B bootstrap datasets drawn with replacement from a single dataset	se_j^{wild}
$block_s$	Simple block bootstrap estimate derived from B bootstrap datasets drawn with replacement from a single dataset	$se_j^{block_s}$
$block_o$	Overlapping block bootstrap estimate derived from B bootstrap datasets drawn with replacement from a single dataset	$se_j^{block_o}$
$random$	Estimate derived from re-training the same network with the same dataset R times, with different random initialization of the network weights each time	se_j^{random}

The only estimate which we have not introduced so far is that denoted *random*. This estimate is generated by training an identical neural network on the same dataset R times, with the only variation introduced by re-initiating the weights of the network randomly before each optimization. This serves to illustrate the variation in predictions which is introduced by the random weight initialization only. Although this would not be used by a researcher to estimate the true sampling uncertainty, it gives an indication of the extent to which theoretical estimates of sample uncertainty for neural networks can be misleading, as they cannot account for this effect.

3.2.7 Data generating functions

We conduct experiments with five I.I.D. functions, for evaluating bootstrapping of the feedforward neural network and denoted M1 to M5, and with three time series functions, for the recurrent neural network, denoted T1 to T3.

I.I.D. data generating functions

In addition to evaluating the performance of bootstrapping on the sample model introduced in Section 3.2.1, we perform experiments on 4 additional models, described in order of increasing complexity. In each case, we generate observations from the data generating function, $f(X)$, as follows:

$$Y_i = f(X_i) + \epsilon_i(X_i)$$

We describe the noise generating function $\epsilon(X_i)$ for each model separately, as the experiments contain a mixture of homoskedastic and heteroskedastic noise terms.

M1:

The first model is a simple linear model with homoskedastic noise. It is included as a baseline, to test the performance of bootstrapping neural networks, where the network is capable of overfitting $f(X)$.

$$f(X_i) = \alpha + \beta X_i$$

$$\epsilon_i \sim \mathcal{N}(0, 1.0^2)$$

The coefficients, α and β, are drawn from $Unif([-1, 1])$, and there are three explanatory features (dimension $X_i = 3$)

M2:

The second model, **M2**, includes interaction and squared terms. The feedforward network should still be capable of fitting $f(X)$ perfectly (given enough data). The noise is heteroskedastic with respect to the first explanatory variable, $X_{(i,1)}$.

$$f(X_i) = \alpha + \beta X_i + \gamma X_i X_i'$$

$$\epsilon_i \sim \mathcal{N}(0, 0.1X_{(i,1)}^2)$$

α, β and γ are drawn from $Unif([-1, 1])$, with the dimension of X_i being 3.

M3:

The third model, **M3**, is the sample model used for graphical illustration throughout the paper. The dimension of X_i is 1, and the noise is heteroskedastic with respect to the single explanatory feature.

$$f(X_i) = \frac{3}{2} sin(3X_i) + X_i$$

$$\epsilon_i \sim \mathcal{N}(0, (0.1X_{(i,1)}^2)$$

M4:

For the fourth model, we increase the dimensionality of the features vector, X_i, to 3. We therefore have multiple non-linear effects generated by the *sin* function, as well as linear effects with respect to each explanatory variable. The noise is heteroskedastic with respect to the first explanatory variable, $X_{(1,i)}$

$$f(X_i) = \beta sin(\gamma X_i) + \delta X_i$$

$$\epsilon_i \sim \mathcal{N}(0, (0.1X_{(i,1)}^2)$$

α, β and γ are drawn from $Unif([-1, 1])$.

M5:

The final model is designed to assess the performance of bootstrapping neural networks when the data generating function is more complicated than the network selected by the researcher. This ensures that the bias is non-zero. In practice, this can occur for two reasons. Firstly, and the most likely, is that the sample size is not large enough relative to the signal/noise ratio for a network to fit the true function. Secondly, the researcher may fail to perform an extensive model selection step, and choose a model architecture which is insufficiently large for the problem at hand. In both of these scenarios, we still wish to test whether bootstrapping is able to approximate the sampling uncertainty.

To ensure the data generating function, $f(X)$, is more complicated than the chosen network, we simply generate data from a feedforward neural network which is larger than any of the options we try during the model selection stage.

$$f(X_i) = NN(X_i)$$

$$\epsilon_i \sim \mathcal{N}(0, 0.1^2)$$

We use three hidden layers in the network, each of size 50, and initialize the weights randomly from $\mathcal{N}(0, 0.5^2)$. The noise is homoskedastic.

Time series data generating functions

T1:

The first time series model is a simple AR(1) process with homoskedastic noise. The neural network should be able to comfortably fit this function with very low bias, even in the small sample settings.

$$y_t = \phi y_{t-1} + \epsilon_t$$

$$\epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

ϕ_1 is set to 0.9 and the noise, ϵ_t, is generated from the normal distribution, with $\sigma = 0.2$.

T2:

For the second time series function, we use an ARMA(3, 3) model, again keeping the noise homoskedastic.

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \phi_3 y_{t-3} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \theta_3 \epsilon_{t-3} + \epsilon_t$$

$$\epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

We draw $phi_{1,2,3}$ and $\theta_{1,2,3}$ from $Unif([-1, 1])$. As in TGF1, the noise is homoskedastic, with $\sigma = 0.2$.

T3

As with M5 in the i.i.d. case, the final model aims at evaluating the effectiveness of bootstrapping in cases where the neural network is unable to fit the true function precisely. We use a recurrent neural network to generate the data, which is purposefully larger than any of the network architecture options available at model selection stage.

$$y_t = RNN(y_{t-1}, ..., y_{t-5}) + \epsilon_t$$

$$\epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

We set the number of hidden layers in the RNN to 3, and the number of nodes in each layer to 50. The weights of the network are initialized randomly from $\mathcal{N}(0, 0.5^2)$, and σ is set to 0.2.

3.2.8 Sample Sizes

For each model, to examine the impact of different sample sizes, we perform experiments with three dataset sizes, 150, 500 and 2000. We denote the models with each dataset size as e.g. $M1_{150}$, $M1_{500}$ and $M1_{2000}$ respectively for model 1. The smaller dataset sizes are of particular interest for the more complicated data generating functions, where we aim to test the performance of bootstrapping in situations where bias is non-zero.

3.3 Results

In assessing the effectiveness of the bootstrap, it is important to note (as developed theoretically in Franke and Neumann [2000]), that the bootstrap estimates should converge to the distribution of the estimation error $\hat{m}(x) - m_0(x)$, where $\hat{m}(x)$ is the prediction of the neural network on a given dataset, and $m_0(x)$ is the optimal network function in a predictive sense. Hence we generate approximate confidence bands for the optimal network function, $m_0(x)$, not the true function, $f(x)$. The difference between $f(x)$ and $m_0(x)$ arises due to the existence of bias in machine learning models, which will be non-zero for finite sample sizes.

In the first section of the results, Section 3.3.1, we investigate the ability of the bootstrap procedures to approximate the distribution of $\hat{m}(x) - m_0(x)$. That is, are the bootstrap distributions generated close to the "$true$" and "$true\_ms$" distributions, based on simulations.

Secondly, in Section 3.3.2, we present the coverage of the true function, $f(x)$, from confidence bands generated by bootstrapping. This includes an analysis of the size of bias in limited sample settings.

In Section 3.3.3, we present results on the additional variation in network predictions introduced due to model selection (hyperparameter search). Section 3.3.4 then provides a comparison of the variance introduced by random weight initialization, sample splitting and sampling uncertainty, providing an insight into the performance of closed form estimators of uncertainty. Finally, Section 3.3.5 highlights a potential failure case for bootstrapping of neural networks which was observed during experimentation.

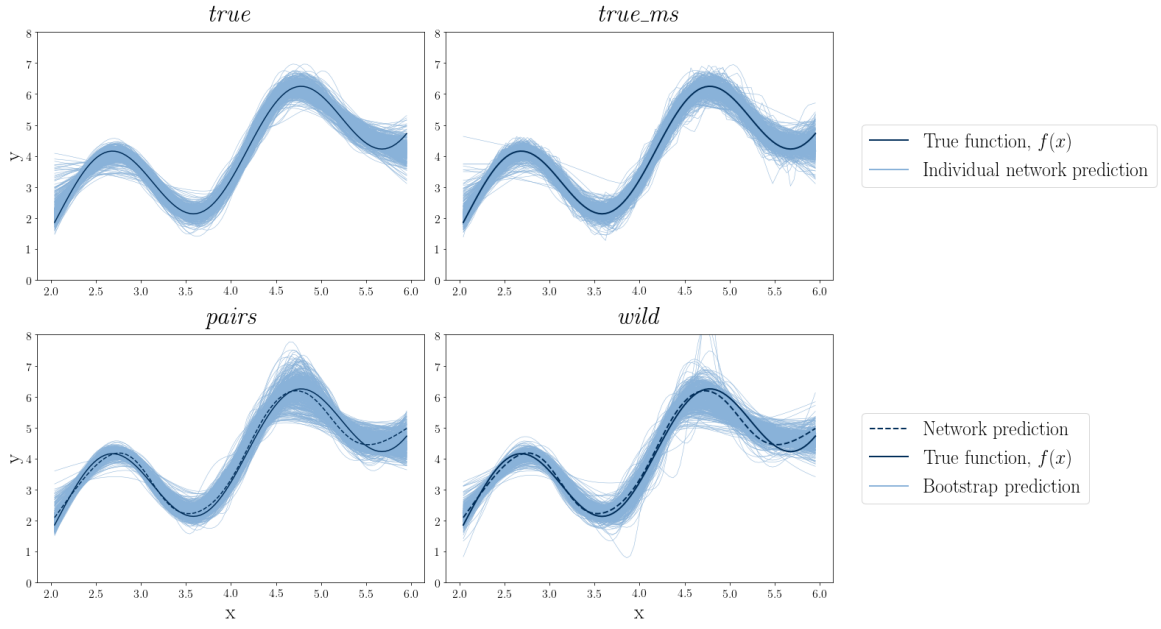
3.3.1 Approximating the distribution of the optimal network model

To begin our analysis of the results, we look at whether the distributions generated by bootstrapping are able to approximate the distribution of the estimation error $\hat{m}(x) - m_0(x)$. In terms of our experimental notation, we are comparing the results of the "true" and "true_ms" distributions with the bootstrap distributions.

This is visualized in Figure 3.6 for the example data generating function **M3**. In the top left panel, we show the true function, $f(x)$ in dark blue, and the predictions $\hat{m}(x)$ for P (P

= 500) neural networks, each optimized on a different draw from the true data generating function. In other words, this panel shows the distribution we are trying to approximate via bootstrapping, when we only have access to one draw from the population. Note that the same set of model hyperparameters, chosen in a model selection step for the first sample of data only, are used in each case. In the top right panel, we show the equivalent predictions, but with a separate model selection step for each data sample. That is, for P iterations, we draw a new sample from the true data generating function, and carry out a model selection step on this individual sample (i.e. P model selection steps in total). This introduces the additional uncertainty due to model selection, and as such the distribution is expected to be wider in the "true_ms" case as opposed to the "true" case. Ideally we would like bootstrapping to approximate the "true_ms" model, which includes all sources of uncertainty. However, for the next two results sections, we focus on comparing the bootstrap distributions to the "true" distributions, and return to the effect of model selection in Section 3.3.3.

Figure 3.6: Illustration of distributions of predictions generated by each method



In the bottom two panels of Figure 3.6, we show the bootstrap approximations to these target distributions. For each of these panels, we have taken a single sample out of the P

samples used to generate the “true" and “true_ms" results, and carried out a model selection step on this sample (the dashed blue line shows the best performing networks predictions of $f(x)$ using the whole sample). We then bootstrapped the sample B times (B = P = 500), and plot the predictions of $f(x)$ of a neural network optimized on each of the B samples. The aim is that the distribution of results generated at each point on x is equal for the bootstrap cases and the “true" cases. On the left hand side we show the distribution for the “pairs" bootstrap method, and on the right hand side for the “wild" method.

Note that we are interested in whether or not the shapes of the bootstrap distributions match the “true" distributions, as opposed to their mean. The single dataset used for the bootstrapping could be any of those which were used to generate the *true_ms* and *true* estimates in Figure 3.6. If the particular realization of the random noise terms in the dataset are above average, then we may be basing our bootstrap estimates around one of the highest realizations in the *true* graph - this is the very nature of the sampling uncertainty we are trying to account for. As long as the distribution generated by the bootstrap method has the correct variance, then the true function should still fall within the confidence interval.

Aside from odd behaviour at the tails of the function,[13] the bootstrap methods appear to approximate the level of sample uncertainty observed by the true simulations reasonably well in this example case. This includes dealing with the heteroskedasticity effectively - the width of the distributions generated increases towards the right side of the charts, where the variance of the noise term increases. To generalize this analysis to all sample sizes and models, we analyse estimates of the standard error at different input values, x_j, and across repetitions. Initially, we assume that the distributions are symmetric.[14] We can consequently estimate the standard error at point x_j as follows (the example given being for the *pairs* distribution):

$$se_j^{pairs} = \left\{ \frac{1}{B-1} \sum_{j=1}^{B} [\hat{m}^{*b}(x_j) - \hat{m}^*(x_j)]^2 \right\}^{1/2} \text{ where } \hat{m}^*(x_j) = \frac{1}{B} \sum_{b=1}^{B} \hat{m}^{*b}(x_j)$$

[13] This is due to the optimization procedure smoothing trading off some bias at the extremes of the distribution for lower variance throughout, an effect discussed in more detail at the end of the results section.

[14] In the case of the example function based on the sin curve, we know the distribution of the noise term $\epsilon(x)$ is indeed symmetric, so the associated distribution should also be centered on the true function.

We can then compare the bootstrap estimates of the standard error with the “true" estimates:

$$norm\_diff_j = \frac{se_j^{true} - se_j^{pairs}}{se_j^{true}}$$

In a stylized sense, this corresponds to comparing whether the distribution at a single value of x in Figure 3.6 has equal variance for the bootstrap and “true" models. We calculate this value on an evenly spaced grid of x_j values of size 100, and repeat the bootstrapping procedure on 100 separate datasets, to ensure accuracy of the results, giving a total of 10,000 normalized differences for each model and sample size. Values greater than 0 correspond to points at which the bootstrap method has underestimated the size of the standard error, and values less than 0 correspond to overestimation. As the differences are normalized, a value of 0.2 would correspond to a 20% underestimation of the true standard error. The vertical dotted line corresponds to the mean of the differences, and the dashed line to the median.

Figure 3.7: Normalized differences between the standard errors generated by the bootstrap estimates and the *true* estimates

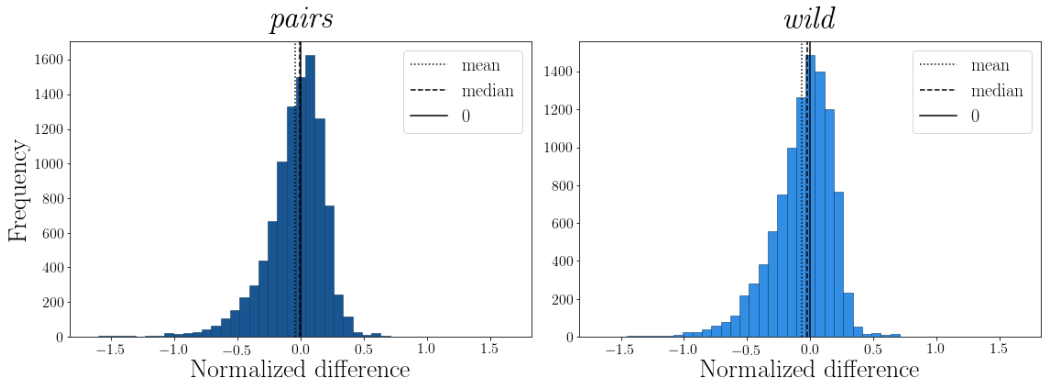


Figure 3.7 shows the distribution of these differences for both the pairs and wild bootstrap approaches, using the example data generating function **M3**. In both cases, the two distributions are centered around zero; that is, the two bootstrap methods tend to overestimate the level of uncertainty as much as they underestimate it.

Table 3.3 displays the full results, for all i.i.d. models and sample sizes. It shows the ratio of bootstrap estimates of the standard error which fall within different percentages of the

true value for the standard error. For example, for **M1** and a sample size of 500, 96.4% of the *pairs* estimates of the standard error fall within 25% of the *true* value, and 99.6% fall within 50%. For each model and sample size, we also present the mean of the differences, corresponding to the vertical dotted line if Figure 3.7.

Table 3.3: Ratio of normalized differences falling within percentages of "true" standard error for Feedforward Neural Network

		Sample size: 150		**Sample size: 500**		**Sample size: 2000**	
		pairs	**wild**	**pairs**	**wild**	**pairs**	**wild**
	mean	*-0.088*	*-0.027*	*-0.014*	*0.217*	*0.002*	*0.087*
	10%	0.488	0.472	0.778	0.200	0.970	0.684
M1	25%	0.834	0.860	0.964	0.584	0.993	0.898
	50%	0.990	0.990	0.996	0.998	1.000	1.000
	90%	1.000	1.000	1.000	1.000	1.000	1.000
	mean	*0.084*	*-0.95*	*-0.053*	*0.169*	*-0.005*	*0.047*
	10%	0.176	0.152	0.492	0.385	0.710	0.688
M2	25%	0.582	0.615	0.839	0.821	0.965	0.962
	50%	0.810	0.829	0.960	0.962	1.000	1.000
	90%	0.930	0.941	0.982	0.993	1.000	1.000
	mean	*-0.072*	*0.111*	*-0.049*	*0.104*	*0.006*	*0.132*
	10%	0.281	0.186	0.404	0.341	0.692	0.308
M3	25%	0.602	0.442	0.796	0.724	0.974	0.856
	50%	0.866	0.856	0.944	0.942	0.999	1.000
	90%	0.971	0.978	0.992	0.991	1.000	1.000
	mean	*0.048*	*0.092*	*-0.012*	*-0.045*	*0.001*	*0.007*
	10%	0.193	0.168	0.382	0.308	0.683	0.613
M4	25%	0.464	0.372	0.693	0.579	0.934	0.933
	50%	0.774	0.699	0.928	0.945	0.971	0.980
	90%	0.950	0.961	0.989	0.992	0.994	0.992
	mean	*-0.250*	*-0.231*	*-0.108*	*-0.094*	*-0.007*	*0.035*
	10%	0.005	0.005	0.184	0.108	0.521	0.501
M5	25%	0.190	0.107	0.452	0.386	0.870	0.848
	50%	0.199	0.157	0.774	0.721	0.937	0.944
	90%	0.272	0.231	0.920	0.923	0.990	0.995

The first broad conclusion from Table 3.3 is that bootstrapping of a single data sample is generally able to approximate the "true" standard errors, with the distributions of differences generally being centred around zero, and converging towards zero as sample sizes grow. Focusing on the "pairs" bootstrapping method initially, and taking $M3$ as an example, we

can see that the mean of the differences in standard error estimates converges from -0.072 with a sample size of 150 to 0.006 with a sample size of 2000, with a similar pattern of convergence towards zero seen across all 5 models. We can also see that the percentage of differences falling within 10%, 25%, 50% and 90% of the "true" standard error converges for all five models, with at least half of differences falling with 10% of the true estimate with a sample size of 2000 for all models. The evidence broadly points to the conclusion that the "pairs" bootstrap approach is effective at generating estimates of the random initialization and sampling uncertainty for feedforward networks.

The performance of the "wild" bootstrapping approach is slightly more erratic. Although the proportion of observations lying within each percentage of the true difference does converge towards 1.0, the *mean* values vary substantially, and are often larger in magnitude than the corresponding mean values for the "pairs" approach. For example, for **M1**, with a sample size of 500, the "wild" method has a mean difference of 0.217 (compared to -0.014 for "pairs"), indicating that the estimated standard errors are consistently too small. In other cases, such as for **M2** with a sample size of 150, the "wild" method has a *mean* difference of -0.95, indicating consistently too large standard error estimates. We suggest that the "wild" method is more erratic due to its reliance on the single neural network which is optimized on the sample data, before the residuals from this network's predictions are re-sampled. If this network does not converge to a strong estimate of $f(x)$, then all residuals sampled will be correspondingly skewed, and as a result all bootstrap networks may converge poorly. The stochastic nature of neural network optimization, and the corresponding lack of understanding of methods for ensuring smooth convergence, suggest that the "pairs" approach may be more robust, due to its non-reliance on a single network's convergence.

The results for **M5**, the data-generating function based on a neural network which is larger in size than any architecture available at model selection stage, are of note. For a sample size of 150, the *mean* values are large and negative for both bootstrap approaches, indicating consistently too wide confidence intervals. The proportion of differences with 90% of the "true" standard error are also very low, at only 0.272 and 0.231, indicating that bootstrapping does not appear to be effective in this case. This is due to the optimization

of the bootstrapped networks breaking down for this small sample size more regularly than the optimization of the networks trained on the population re-samples, an effect which we demonstrate in more detail in Section 3.3.5 below. Despite this, for the larger sample sizes, the estimates of the standard error converge to stronger values, indicating that the bootstrap approach is able to generate sensible distributions, even in the presence of bias (i.e. even if the confidence intervals are not centered on $f(x)$, as discussed in Section 3.3.2 below).

Table 3.4: Ratio of normalized differences falling within percentages of true standard error for Recurrent Neural Network

		Sample size: 150		**Sample size: 500**		**Sample size: 2000**	
		$block_s$	$block_o$	$block_s$	$block_o$	$block_s$	$block_o$
	mean	*0.383*	*0.377*	*-0.003*	*0.002*	*0.055*	*0.065*
	10%	0.142	0.138	0.447	0.463	0.547	0.583
T1	25%	0.350	0.358	0.835	0.875	0.895	0.919
	50%	0.650	0.654	0.996	0.999	0.999	0.999
	90%	1.000	1.000	1.000	1.000	1.000	1.000
	mean	*-0.121*	*0.086*	*0.007*	*-0.007*	*-0.003*	*0.002*
	10%	0.085	0.087	0.403	0.467	0.587	0.602
T2	25%	0.428	0.509	0.798	0.804	0.901	0.911
	50%	0.533	0.611	0.976	0.970	0.992	0.992
	90%	0.965	1.000	1.000	1.000	1.000	1.000
	mean	*-0.238*	*-0.348*	*0.015*	*-0.071*	*-0.002*	*0.004*
	10%	0.001	0.003	0.328	0.301	0.568	0.608
T3	25%	0.240	0.208	0.677	0.702	0.878	0.872
	50%	0.437	0.418	0.903	0.899	0.956	0.966
	90%	0.886	0.891	0.991	1.000	1.000	1.000

We present the same results, for the time series generating functions **T1** to **T3** in Table 3.4. The results broadly support the conclusions from the feedforward case in Table 3.3, that the differences converge to zero at larger sample sizes. However, particularly for the smaller sample sizes for the RNN, we see relatively large values for the *mean* difference for all all three data generating functions. We suggest that this is due to the relative complexity of even small, single-layer recurrent neural networks, in which each prediction is able to attend to all prior observations in the time series, making overfitting very easy in small samples. We don't see any consistent differences between the results for the "simple" block bootstrap approach

($block_s$) and the “overlapping" approach, ($block_o$), with each method providing estimates of the standard error with a similar level of accuracy.

3.3.2 Coverage of the true function $f(x)$

Next, we examine the ability of the bootstrap methods to provide accurate confidence intervals for the true function, $f(x)$. To generate a 95% interval, we take the 2.5th and 97.5th percentiles of the bootstrap distributions. For these estimates to provide correct confidence intervals for $f(x)$, two things must be correct. Firstly, the distribution of the bootstrap estimates at each point x_j must match that generated by the true sampling uncertainty. We showed that this is broadly the case in the section above, with the width of the bootstrap distributions converging to the width of the “true" distributions. Secondly, the neural networks estimates of the true function, $\hat{f}(x)$, must be non-biased. We know that in small-sample settings, neural networks are optimally trading off bias and variance for accurate predictions, and that bias will be non-zero. However, as the sample sizes grow larger, we would expect the bias to converge to zero, and as a result the coverage of bootstrapped estimates of confidence intervals to converge to the correct levels.

To illustrate the coverage metric we are going to present, we show confidence intervals for **M3** in Figure 3.8, for three different samples (i.e. 3 out of the 100 repetitions of bootstrapping which we carry out for each data generating function). Note that the intervals are generated around a single network prediction for this sample, shown by the dashed dark blue line. As a result, the intervals are not necessarily centered around $f(x)$, but should cover it in approximately 95% of repetitions.

As the sample size grows, we expect the width of the intervals to decrease, as shown in Figure 3.9. The coverage of the true function should still remain at the correct percentage level.

To evaluate coverage for all sample sizes and data generating functions, we generate intervals for each bootstrap repetition at 100 evenly spaced values of x_j for each function (i.e. for **M3** illustrated above we would calculate the intervals at 100 values of x_j between 2.0 and 6.0). We then present the percentage of times the true function $f(x)$ lies within these

Figure 3.8: Example 95% confidence intervals generated from pairs bootstrapping for **M3** with a sample size of 150

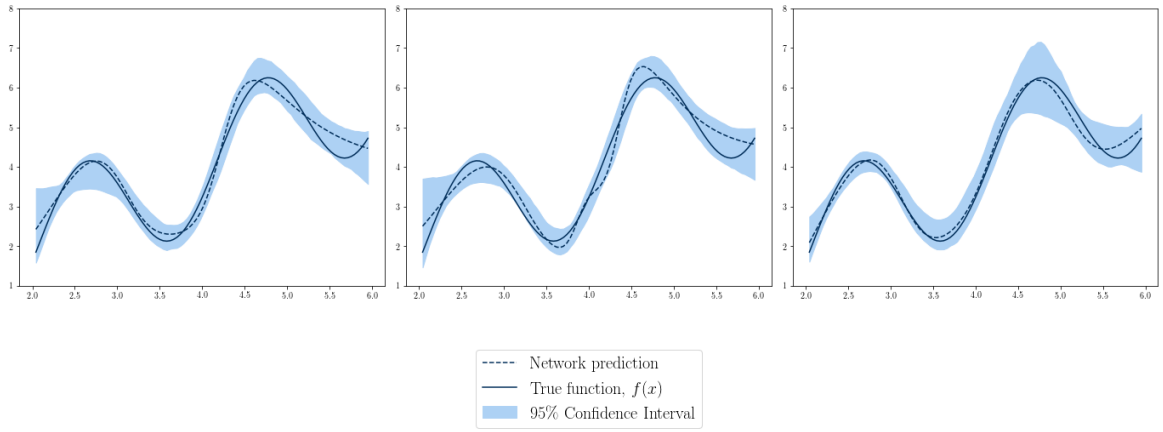
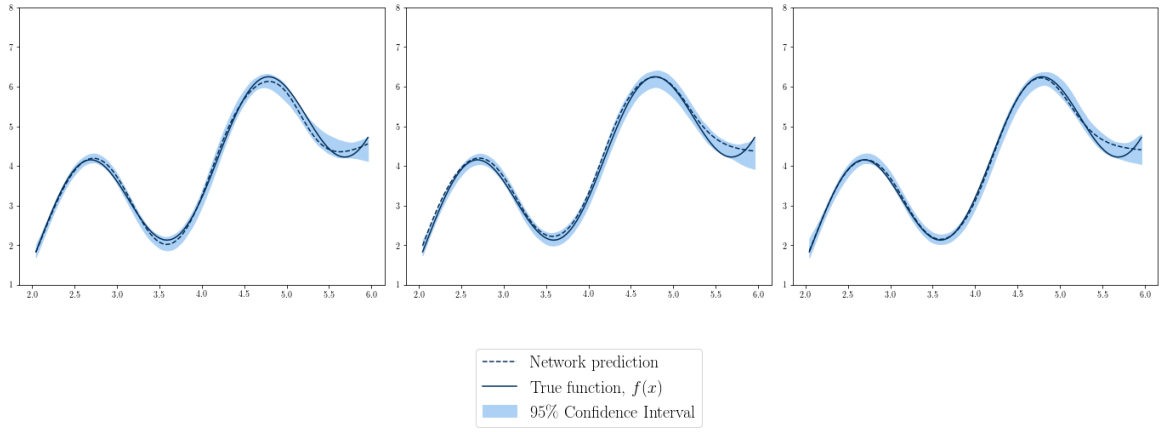


Figure 3.9: Example 95% confidence intervals generated from pairs bootstrapping for **M3** with a sample size of 500



intervals, and average across all repetitions. Table 3.5 presents the results for the feedforward neural network.

Table 3.5: Coverage of true function $f(x)$ from 90% and 95% confidence interval generated by bootstrapping for FeedForward Neural Network

		Sample size: 150		**Sample size: 500**		**Sample size: 2000**	
		pairs	**wild**	**pairs**	**wild**	**pairs**	**wild**
M1	90%	0.980	0.850	0.923	0.704	0.894	0.879
	95%	1.000	0.924	1.000	0.828	0.965	0.906
M2	90%	0.948	0.698	0.870	0.760	0.906	0.863
	95%	0.973	0.781	0.941	0.844	0.956	0.914
M3	90%	0.727	0.573	0.938	0.677	0.899	0.715
	95%	0.803	0.659	0.964	0.745	0.941	0.798
M4	90%	0.945	0.867	0.939	0.848	0.923	0.858
	95%	0.987	0.932	0.983	0.884	0.969	0.903
M5	90%	0.968	0.912	0.946	0.849	0.804	0.762
	95%	1.000	0.978	1.000	0.918	0.856	0.845

We begin by analysing the results for the “pairs" approach. For each of the models **M1** to **M4**, the coverage converges towards the correct level for the larger sample sizes, indicating that both the width of the confidence intervals is correct, and that bias is converging to zero. The exception is **M5**, which over-covers for a sample size of 150, but under-covers for a sample size of 2000. The small-sample case is discussed in Section 3.3.5 below. The large-sample case is in fact an expected result. By generating the data from a neural network which is larger in size than any available during model selection, we ensured the existence of non-zero bias. As a result, although the width of the bootstrap confidence interavals appears to converge to the correct level, the coverage of the true function remains too low even in large samples.

As with the distribution results, the performance of the “wild" bootstrap approach is considerably worse than the “pairs" approach, with the confidence intervals converging at a slower rate to the correct coverage, and consistently under-covering the true function. This paper is the first to compare the “pairs" and “wild" approach to bootstrapping neural networks, and we present as an additional result that the “pairs" approach is a more robust method.

Table 3.6: Coverage of true function $f(x)$ from 90% and 95% confidence interval generated by bootstrapping for Recurrent Neural Network

		Sample size: 150		**Sample size: 500**		**Sample size: 2000**	
		$block_s$	$block_o$	$block_s$	$block_o$	$block_s$	$block_o$
T1	90%	0.622	0.588	0.745	0.790	0.870	0.872
	95%	0.678	0.642	0.809	0.843	0.926	0.928
T2	90%	0.598	0.605	0.721	0.722	0.867	0.876
	95%	0.676	0.724	0.812	0.843	0.918	0.943
T3	90%	0.021	0.057	0.456	0.477	0.870	0.877
	95%	0.043	0.128	0.573	0.602	0.925	0.937

Table 3.6 presents the equivalent coverage results for the recurrent neural network. As with the feedforward case, the coverage levels broadly converge towards the correct levels. However, for the smaller sample sizes, the coverage is consistently too low, potentially indicating a higher level of bias for the recurrent models. As discussed previously, we suggest this is due to the flexibility of the recurrent neural networks, which can attend to all previous values in the time series. As a result, to prevent overfitting the optimization if often stopped early in the process, where a high level of bias may still exist.

3.3.3 Effect of model selection

As introduced in Section 3.2.3, there is an additional source of uncertainty for neural networks, ignored by the literature on confidence intervals to date, which arises due to the requirement for a model selection stage, in which we choose a set of network hyperparameters suitable for the dataset in question. To account for this, we introduced the “true_ms" baseline, in which for each new draw from the population generating function, we carry out a separate model selection stage.

Due to the computational cost of the “true_ms" baseline - we have to optimize multiple neural networks for each draw from the population generating function - we limited our experiments to two models, **M3** and **M5**. For each model and the three sample sizes, we calculate the following quantity, representing the average difference in the size of the standard errors of the “true_ms" model vs. the “true" model:

$$ms\_diff = \frac{1}{J}\sum_{j=1}^{J}\frac{(se_j^{true\_ms} - se_j^{true})}{se_j^{true\_ms}}$$

The results are presented in Table 3.7.

Table 3.7: Average difference between standard error estimates between the “true_ms" model and the “true" model for feedforward neural networks

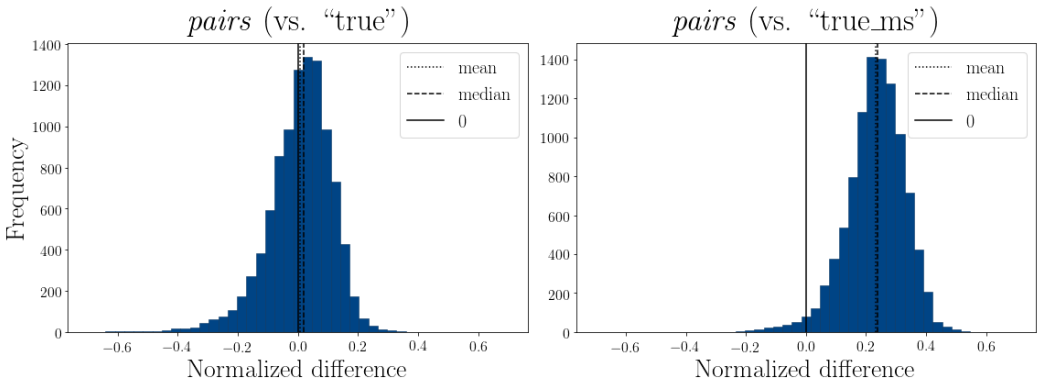
	Sample size: 150	**Sample size: 500**	**Sample size: 2000**
M3	0.129	0.168	0.225
M5	0.096	0.182	0.193

We emphasize that these results are indicative of the general size of the effect only; clearly the effect of the model selection step may vary considerably based on both the number of hyperparameters experimented with, and the distributions of these hyperparameters, combined with the dataset in question. However, it is of interest that in our simple experiments the model selection step results in standard error estimates which are up to 22% larger than those without.

As we have shown in Section 3.3.1, the bootstrap standard errors converge to the “true" baseline, without model selection. As a consequence, bootstrapping provides standard errors which are consistently smaller than the “true_ms" baseline. This is illustrated in Figure 3.10, for **M3** and a sample size of 2000. On the left hand side, we have the distribution of differences between the “pairs" bootstrap standard errors and the “true" standard errors, as shown previously on the left hand side of Figure 3.7. On the right, we have the comparison of the “pairs" standard errors to the “true_ms" standard errors. The mean of the differences is now just above 0.2, indicating that the bootstrap standard errors are consistently around 20% too small relative to “true_ms".

One potential method of dealing with this would be to carry out a model selection step for every bootstrap draw. However, to do so would be computationally extremely expensive, increasing the number of networks which need to be trained linearly in the number of parameter options we are testing (in our experiments this would result in training 54 * 500 = 27000

Figure 3.10: Normalized differences between the standard errors generated by the bootstrap estimates and the *true_ms* estimates



networks per iterations, instead of 500). In practice, it is extremely unlikely any researcher will perform this on top of bootstrapping their networks.

3.3.4 Comparison of neural network variance from different sources

As a supplementary result, we present the relative amount of variance introduced through random initialization ("random") of the network weights, re-splitting of the train, validation and test data, and population re-sampling. As with the impact of model selection, these results should be taken as indicative of the relevant size of each effect only. In particular, the effect of random weight initialization will depend heavily on the distribution from which the network weights are initialized. For example, if we randomly initialize the weights prior to training from $\mathcal{N}(0, 0.5^2)$ we would clearly expect a smaller size of this effect than if we initialize from $\mathcal{N}(0, 5.0^2)$.

That said, it is still of interest to give indicative sizes for each effect, and as such, in Table 3.8 we present the average size of the standard deviation for the "random", "split" and "true" models, normalized by the size of the "true" standard deviation.

Table 3.8: Average standard deviation resulting from random initialization, sample splitting and re-sampling for sample size of 500

	random	**split**	**true**
M1	0.421	0.771	1.000
M2	0.398	0.661	1.000
M3	0.325	0.579	1.000
M4	0.302	0.639	1.000
M5	0.258	0.539	1.000

Note that the three results nest each other; the “random" result contains the variation from random initialization only, whereas the “split" result contains the variation from random initialization **and** sample splitting, and the “true" result contains the variation from both of the prior two sources **and** population re-sampling. As such, it is not possible to isolate each effect precisely. However, it is clearly the case that both the impact of random initialization and of sample splitting is non-negligible. This supports the results in Tibshirani [1996], that closed-form estimators for neural network standard errors necessarily underestimate the true standard error, as they cannot account for the impact of random weight initialization or sample splitting. Our results indicate, on the other hand, that bootstrapping is able to adequately cope with both of these effects in a very intuitive fashion (we are able to re-initialize randomly and re-sample the validation data for each bootstrap draw).

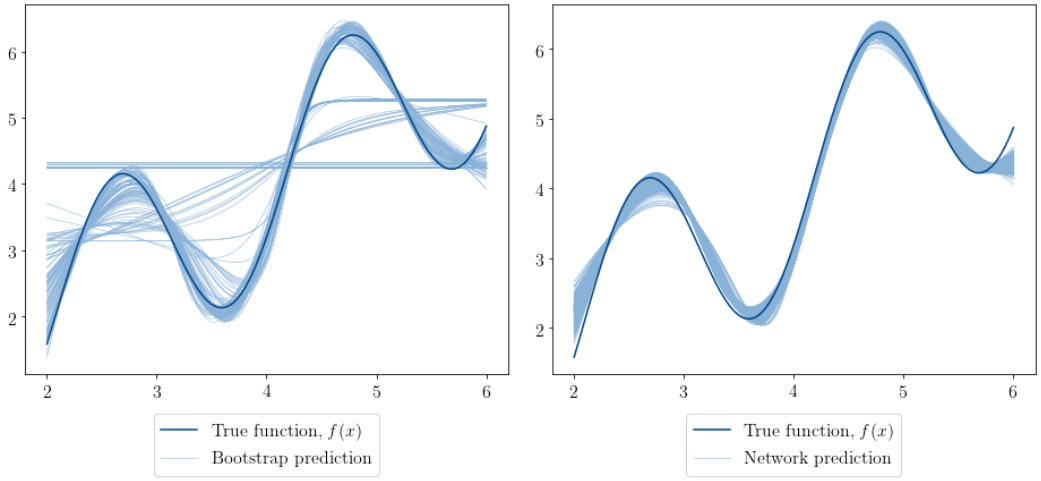
3.3.5 Analysis of failure cases

In Section 3.3.1, we noted that for **M5**, and a sample size of 150, the bootstrap estimates provided estimates of the standard error which were consistently too large, with a mean difference to the “true" results of -0.250, and with only 0.5% of standard error estimates within 10% of the “true" standard errors. Here, we present an explanation of this result, and re-produce it with **M3**, to illustrate the effect. Optimization of neural networks is notoriously fragile, and for the results in this paper we carry out a careful model selection step for each dataset and population draw. However, with small sample sizes, and bootstrapping, there is less variation in the bootstrap draws (in the sense that there are repeated draws of the same data points), than in the full sample. In some cases, this can lead to the neural networks

trained on the full sample converging more consistently than the neural networks trained on the bootstrap samples.

To illustrate this effect, we take the example data generating function, **M3**. As with all experiments, we initialize the network weights prior to training of the bootstrap samples and population samples from $\mathcal{N}(0, \sigma^2)$, and then train the "pairs" model based on 500 bootstrap draws, and the "true" baseline using 500 samples from the data generating function. We then increase the value of σ (increasing the initialization variance leads to worse network optimization beyond a certain level), until we get the effect shown in Figure 3.11. That is, the convergence of the bootstrap networks begins to break down for some samples (left hand panel), but we still get smooth convergence for the population re-samples (right hand panel).

Figure 3.11: Reproduction of failure of convergence for bootstrap estimates for **M3**



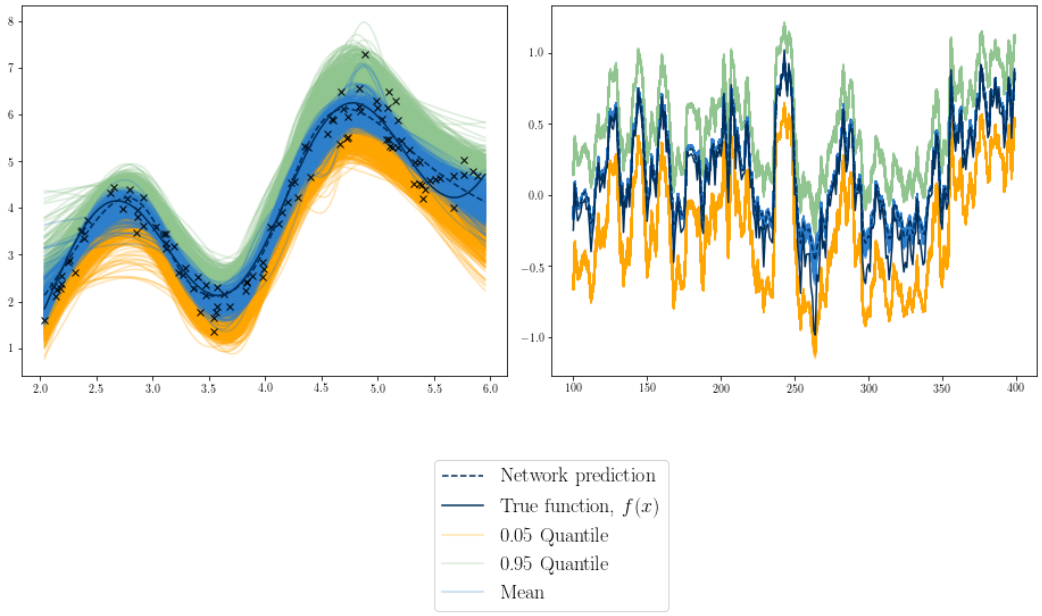
If we continue the process of increasing the value of σ, the convergence of the networks trained on the population samples will also begin to break down. However, for a short interval, because there is more information (i.e. less repeated samples) in the population samples than there is in the bootstrap samples, the former converges whilst the latter breaks down. This gives confidence intervals for the bootstrap approach which are substantially too large, as we observed with **M5** in the main results section. However, such cases are easily diagnosable, in that we observe substantially larger validation losses for some of the bootstrap networks than others, and in such cases, a more robust set of hyper-parameters can be employed.

3.4 Jointly generating confidence and prediction intervals

To tie together the previous two chapters, we provide the first demonstration of the joint generation of both confidence and prediction intervals for feedforward and recurrent neural networks. For generation of the quantiles, we take the best performing output layer variant from Chapter 2, in which the quantile output layers are optimized separately from the main network, a residual connection is used, and for which the quantile predictions are estimated as the mean/median prediction plus or minus the quantile output layer (i.e. the model denoted *sep_a_r* in Chapter 2).

We then perform the same bootstrapping procedures as outlined in this chapter, but with the quantile output layers included. For demonstration purposes, we use **M3** for the feedforward neural network, and **T1** for the feedforward network, with the results illustrated for a sample size of 100 in Figure 3.12. By combining the quantile output layers with bootstrapping, we get a distribution of predictions around both the mean prediction (denoted in blue in Figure 3.12), the 0.95 quantile (denoted in green in Figure 3.12), and the 0.05 quantile (denoted in orange).

Figure 3.12: Bootstrapping of multi-quantile neural network for example data generating functions and a sample size of 100



Generation of the confidence interval proceeds in the same fashion as this paper, and is illustrated by the dark blue shaded region in Figure 3.13, for the feedforward neural network optimized on **M1** and a sample size of $N = 100$ on the left hand side and $N = 300$ on the right.

However, we now also have a distribution of predictions around each of the quantiles. This could, in theory, be used to generate a confidence interval around each quantile. However, if we are interested in generating a prediction interval for the data generating function, we now have two options. First, we could take the mean of the distribution of predictions around the quantile. A 90% prediction interval generated in this fashion gives the coverage at sample sizes of 30, 100, 200 and 300 as presented in Table 3.9 for the feedforward neural network, and in Table 3.10 for the recurrent neural network. As before, results are averaged over 100 repetitions.

Table 3.9: Coverage of 90% prediction intervals from feedforward NN for **M3**

	30	100	200	300
Mean	0.77	0.846	0.879	0.896
90th Percentile	0.951	0.978	0.977	0.966

As we would expect, the coverage converges towards the correct level as the sample size increases, with the method giving a coverage of 0.896 for a sample size of 300 in the feedforward case and of 0.902 in the recurrent case. However, at lower sample sizes, the prediction intervals generated by taking the mean of the quantile predictions consistently under-cover, increasing gradually from a coverage of just 0.77 for a sample size of 30 in the feedforward case, and from 0.823 in the recurrent case. This result stems from the distribution of the noise term, $\epsilon(x)$. For any $\epsilon(x)$ which has lower density in the tails than around the mean (such as is the case for the normally distributed errors in these examples), a prediction interval generated by taking the mean of the quantile distributions will under-cover. Although the quantile predictions may be distributed symmetrically around the true quantile, a prediction that is too close to the mean (resulting in too narrow a prediction interval) will miss a greater

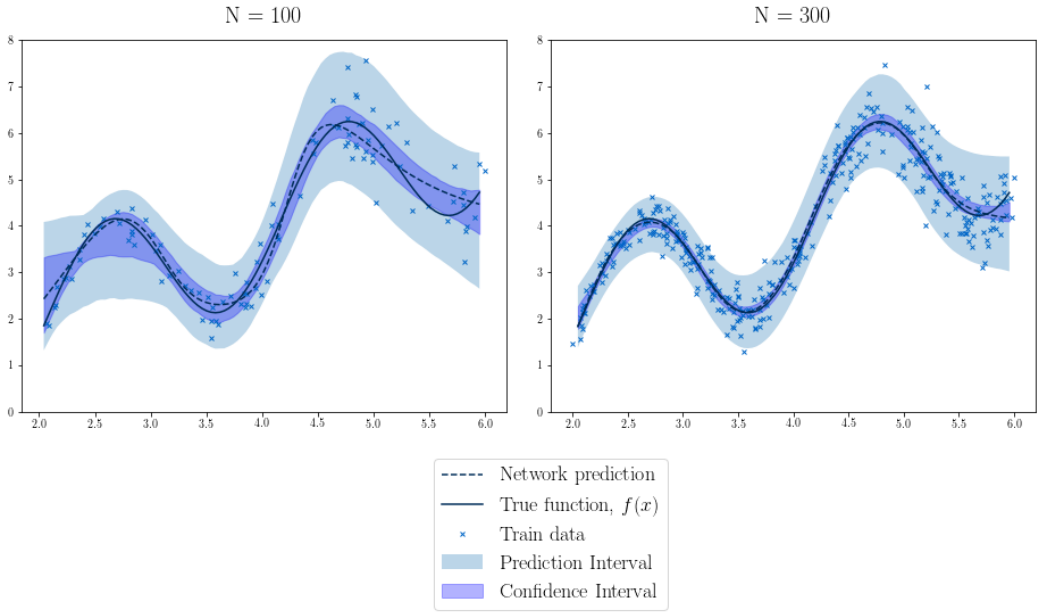
number of realized observations, $y_i(x)$, than one which is too far from the mean (too wide a prediction interval), as the density of $\epsilon(x)$ is greater closer to the mean than in the tails.

Table 3.10: Coverage of 90% prediction intervals from recurrent NN for **T1**

	30	100	200	300
Mean	0.823	0.867	0.893	0.902
90th Percentile	0.876	0.931	0.946	0.926

As an alternative, one could take as the prediction of the quantile the pth percentile of the distribution of quantile predictions. In this case, we choose the percentile that is furthest from the mean, to give a conservative prediction interval. For example, when generating the prediction of the 0.95 quantile in Figure 3.12, we take the 90th percentile of the distribution denoted in green. Conversely, for our estimation of the 0.05 quantile, we take the 10th percentile of the distribution denoted in orange. We present the coverage of prediction intervals generated in this fashion in Tables 3.9 and 3.10, denoted “90th Percentile". The coverage is now conservative (we over-cover) even at larger sample sizes, with a coverage of 0.966 for the feedforward NN and 0.926 for the RNN with a sample size of 300. However, this may be desirable from the researchers perspective especially with smaller sample sizes or complex data generating functions. A more precise coverage at lower sample sizes could be obtained by estimation of the gradient of the density of $y(x)$ around the quantiles (through generation of multiple quantile predictions around each quantile), which we leave to future work. Figure 3.13 presents the confidence (dark blue shaded region) and conservative prediction intervals (light blue shaded region) for the example data generating function, and sample sizes of $N = 100$ and $N = 300$.

Figure 3.13: Confidence and conservative predictions intervals generated jointly by feedforward neural network for example data generating function



3.5 Computational efficiency when bootstrapping

As an additional contribution, we introduce a novel computational architecture for training multiple neural networks on a GPU concurrently, which reduces the time to train a mid-sized network by up to a factor of 40, and smaller networks by considerably more. This innovation facilitated the experiments presented in this paper, reducing compute time on a larger server to a few days, from a few months.

The effect relies on the fact that modern GPUs are incredibly efficient relative to a CPU at processing very large matrices. This makes them many orders of magnitude faster at training very large networks, such as those with millions of weights used in image analysis. However, the time to transfer data to/from a GPU is significantly greater than transfer to/from the CPU. This means that training a very small neural network is often not particularly efficient with a GPU. A small[15] with 3 layers of size 100 trains in about 4 seconds on a personal computers CPU, compared to 5 seconds on the GPU (in this case, an Nvidia GeForce 1080).

[15]"Small" here is relative to neural networks used commmonly in machine learning applications, which can have many millions of weights.

However, by joining many small networks together during training, we can pass them to the GPU as a series of large matrix calculations, significantly improving the time it takes to train an individual network. Table 3.11 demonstrates that for this network size, we can reduce the time taken to optimize one network on the GPU from 2.42 seconds down to 0.07 seconds. This is a significant computational gain, and aids in making bootstrapping of neural networks a computationally viable practice for the researcher to generate confidence intervals.

Table 3.11: Training time for neural networks when concatenated together for optimization

	Total time (s)		Time per network (s)		GPU utilization (%)
Number of Networks	CPU	GPU (1080)	CPU	GPU (1080)	GPU (1080)
1	3.47	2.42	3.47	2.42	NaN
2	7.02	2.60	3.51	1.30	NaN
5	13.15	2.65	2.63	0.53	NaN
10	22.58	2.86	2.26	0.29	NaN
25	48.14	3.88	1.93	0.16	59.00
50	90.42	5.79	1.81	0.12	65.67
100	175.79	9.19	1.76	0.09	74.33
150	274.63	12.68	1.83	0.08	76.22
200	348.57	16.07	1.74	0.08	80.00
500	NaN	35.26	NaN	0.07	84.77

To achieve the efficiency gains recorded, three network architecture points must be combined.

1. The data must be passed to the GPU in a single matrix of dimensions [num_bootstraps, batch_size, num_features] (for a single feedforward network the dimensions would be [batch_size, num_features])

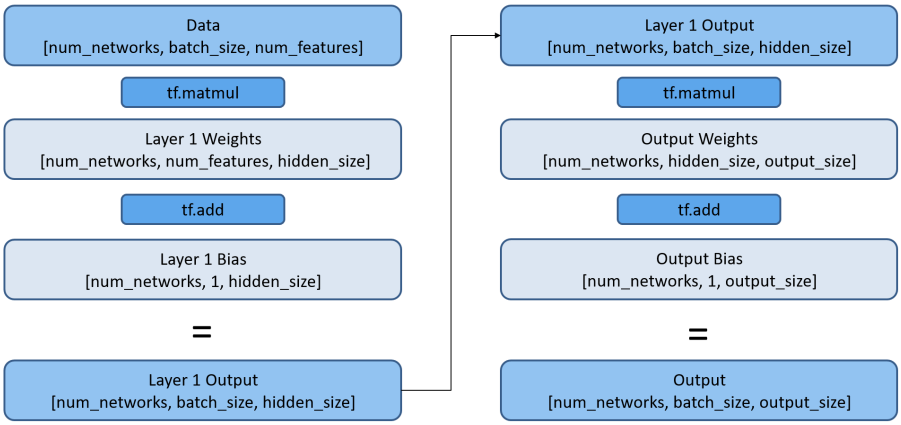
2. The weights of the network are created in size [num_bootstraps, input_size, output_size] matrices (as opposed to [input_size, output_size] for a standard network), and multiplied with the inputs to the layers in a single matmul op, which will broadcast efficiently.

3. All the networks concurrently with a single scalar loss function. To achieve the same gradient updates as if all networks had been optimized individually, the gradients must be scaled by num_bootstraps in the optimization step:

$$\theta_{new} = \theta_{old} - \alpha \frac{\nabla_\theta C(\theta)}{B}$$

The structure required for a single layer network is demonstrated in Figure 3.14.

Figure 3.14: Network structure for efficient training on GPU



This improvement can be applied to both the feedforward neural network and the recurrent neural network, providing similar computational improvements in both cases.

3.6 Summary

We have presented the largest simulation study assessing the performance of bootstrapping for neural networks to date, and the first simulation study assessing the performance of block bootstrapping for recurrent neural networks. We show that, broadly speaking, bootstrapping is able to account for the uncertainty introduced by random weight initialization, sample splitting and population re-sampling, providing standard errors which converge towards the correct levels, and correct coverage in large samples. We also present the first results indicating the potential size of the additional uncertainty introduced my model selection, and show that this can add significant additional variation to results, which is not covered by bootstrapping or closed form estimators.

In addition, we have presented the first comparison of the performance of the wild bootstrap and pairs bootstrap approaches for I.I.D. data, and of the simple block and overlapping block

approaches for time series data. In the former case, we show that the pairs approach performs substantially better, whilst the performance of the two block methods is similar. We also presented experimental results indicating the potential size of the additional uncertainty which can be introduced by a model selection step, which has been ignored by the literature today, and demonstrate that it can lead to distributions which are up to 25% wider. Finally, we presented a novel computational method for speeding up the training of multiple small neural networks, achieving up to a 40 times speed increase.

There still remains considerable work to be done in understanding the convergence of neural networks, and putting theoretical bounds on this convergence. In Section 3.3.5 we architected a failure of convergence for the bootstrap samples, leading to a breakdown of the standard error estimates. Such cases demonstrate the continued fragility of neural network convergence, and we emphasise that the results in this paper are prone to breakdown if the correct set of hyper-parameters for each dataset are not selected in a careful model selection step. Improving the robustness of convergence and greater understanding of the theoretical properties necessary for smooth convergence would greatly benefit the ability for practitioners to provide accurate confidence intervals for network predictions.

Bibliography

Carlstein, E. (1986). The use of subseries values for estimating the variance of a general statistic from a stationary sequence. *The Annals of Statistics*, 14(3):1171–1179.

Dybowski, R. and Roberts, S. (2001). *Confidence intervals and prediction intervals for feedforward neural networks*, pages 298–326.

Efron, B. (1979). Bootstrap methods: another look at the jackknife. *Annals of Statistics 7 (1)*, pages 1–26.

Efron, B. and Hastie, T. (2016). *Computer age statistical inference: algorithms, evidence and data science*. New York: Springer.

Franke, J. and Neumann, M. (2000). Bootstrapping neural networks. *Neural computation*, 12:1929–1949.

Hall, P. (1985). Resampling a coverage pattern. *Stochastic Processes and their Applications*, 20(2):231–246.

Hardle, W. (1990). *Applied Nonparametric Regression*. Econometric Society Monographs.

Hardle, W. and Bowman, A. (1979). Bootstrapping in nonparametric regression: local adaptive smoothing and confidence bands. *Journal of American Statistical Association 83 (401)*, pages 102–110.

Heskes, T. (1997). Practical confidence and prediction intervals. *Advances in Neural Information Processing Systems 9*, pages 176–182.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

Jason Hartford, Greg Lewis, K. L.-B. and Taddy, M. (2017). Counterfactual prediction with deep instrumental variables networks. *Proceedings of the 34th International Conference on machine learning 70*, pages 1414–1423.

Kim, J., El-Khamy, M., and Lee, J. (2017). Residual LSTM: design of a deep recurrent architecture for distant speech recognition. *CoRR*, abs/1701.03360.

Kunsch, H. R. (1989). The jackknife and the bootstrap for general stationary observations. *The Annals of Statistics*, 17(3):1217–1241.

LeBaron, B. and Weigend, A. (1994). Evaluating neural network predictors by bootstrapping. *Proceedings of the International Conference on Neural Information Processing*, 2:1207–1212.

Pass, G. (1992). Assessing and improving neural network predictions by the bootstrap algorithm. *Advances in neural information processing systems (NIPS)*, 5:196–203.

Tibshirani, R. (1996). A comparison of some error estimates for neural network models. *Neural Computation*, 8(2):152–163.

Wu, J. (1986). Jackknife, bootstrap and other resampling methods in regression analysis (with discussions). *Annals of Statistics 14 (4)*, pages 1261–1295.

Xiong, R., Nichols, E., and Shen, Y. (2015). Deep learning stock volatilities with Google domestic trends. *arXiv, https://arxiv.org/abs/1512.04916*.

Chapter 4

Improving nested Named Entity Recognition for more accurate sentiment indices

Joint work with Dr. Andreas Vlachos

There is a growing body of literature which aims to convert raw text data, such as news articles, companies 10-K filings and central bank reports, into sentiment indices which can be used for tasks such as measuring uncertainty, forecasting volatility or analyzing the impact of central bank communications. Current methods of building the indices rely on simple string matching, and sentiment-related word lists, both of which are inherently noisy. We suggest that more accurate measurements can be achieved by processing text with neural networks, and develop a novel neural network architecture for nested named entity recognition, the first stage in such an approach. Our architecture first merges tokens and/or entities into entities forming nested structures, and then labels each of them independently. We evaluate the accuracy of our approach for nested named entity recognition using the ACE 2005 Corpus, where it achieves state-of-the-art F1 of 74.6, further improved with contextual embeddings (BERT) to 82.4, an overall improvement of close to 8 F1 points over previous approaches trained on the same data. Additionally we compare it against BiLSTM-CRFs, the dominant approach for flat NER structures, demonstrating that its ability to predict nested structures does not impact performance in simpler cases.[1]

[1]Code available at https://github.com/fishjh2/merge_label

4.1 Introduction

Given the increasing quantity of raw text data available and easily accessible online, there is a growing interest in the finance and economic policy-making literatures in methods which can convert such data into indices, which can be then be used for forecasting financial variables, or monitoring the state of the financial sector or economy. We suggest that the common approach of building such indices, based on string-matching and pre-defined word lists, can be improved upon by processing the text with neural networks, allowing assignment of individually relevant sentiment-scores to each company, country, institution or person in a news corpus. Such an approach relies on accurately identifying entities in the text, a task referred to as "named entity recognition" in the natural language processing (NLP) literature. This paper focuses on improving both the accuracy and the speed of this part of the pipeline, ultimately allowing development of more accurate indices from a large news article corpus.

Applications

The first area text-based sentiment indices have become common is in attempting to predict asset prices. In Loughran and McDonald [2011], the authors define a set of "positive" and "negative" words specifically for financial news[2], and using a sample of 10-Ks from 1994 to 2008, build a sentiment index using these word list. They then investigate the links between this index and returns, volume, volatility and unexpected earnings in a series of regressions, finding mixed results in terms of forecasting, but in all cases that their financial-specific word-lists outperform their generic counterparts. Tetlock [2007] uses a sentiment measure of the market as a whole extracted from the Wall Street Journal column, "Abreast of the Market". The measure is generated based on counts in each article of words in the General Inquirer's Harvard IV-4 "psychosocialdictionary". They find a positive link between extremes of their measure (periods of very high or very low pessimism) and market trading volume,

[2]They find that a generic list of "positive" and "negative" words is not suitable for financial news, with up to three quarters of words in the generic "negative" list not having negative implications in a financial setting (such as "liability".

and that periods of high pessimism in the journal tend to be followed by a drop in prices and a reversion to fundamentals.

In Tetlock et al. [2008], the authors expand their previous method to all Wall Street Journal and Dow Jones News Service stories about S&P 500 firms from 1980 to 2004. This allows them to construct a firm-specific sentiment index, again using the lists of positive and negative words from the General Inquirer, as opposed to the single market-wide index previously. They find their index has forecasting power for low earnings, and that stock prices under-react to sentiment information, with both effects strongest when limiting their analysis to news articles which discuss firm's fundamentals (this highlights the potential noisiness of the measure - an article which mentions "Microsoft" in the title but then goes on to discuss a feature in Microsoft Office is unlikely to be significantly correlated with the stock price). In a slightly different market, Soo [2013] again uses positive and negative word lists to build sentiment indices of local housing markets, based on local newspaper articles. They find their index forecasts house prices two years in advance, as well as volume of trading.

In a macro setting, Baker et al. [2016] have produced an index of economic policy uncertainty (EPU). To build this index, they take articles from of the largest U.S. newspapers over a period beginning in 1900, and is based on the relative frequency of articles which contain at least one term from each of the following three lists:

1. economic, economy
2. uncertain, uncertainty
3. congress, deficit, federal reserve, legislation, regulation, white house

They show that the index spikes around presidential elections (particularly if closely contested), the Gulf Wars, 9/11 and the 2007 financial crisis, amongst other turbulent periods, and that spikes foreshadow drops in investment and employment at both a firm and macro level. Nyman et al. [2018] takes a similar approach to build the "Relative Sentiment Shift" index in the UK, based on counts of words related to uncertainty, and using a combination of Bank of England daily reports, Reuters' newswire articles and broker research reports.[3]

[3]The word list is specifically motivated by social-psychological research into action under uncertainty.

The incorporation of central bank reports itself builds on a growing literature which aims to monitor the sentiment which central banks are expressing in reports over time, including Sturm and De Haan [2011] and Blinder et al. [2008], with a focus both on greater understanding of the impact of central bank communications on the economy, and in the case of Nyman et al. [2018] on forecasting volatility in financial markets. Whilst the debate over the robustness of forecasts based on these methods is ongoing, both due to their relative recency (we only have access to a relatively short period of reliable online news articles, bank reports etc.) and due to the inherent endogeneity of the problem, it is clear from the growth of the literature and the interest from central banks that such sentiment indices based on raw text will continue to be used, at the very least to monitor existing conditions in markets. [4]

Limitations of current approaches

Common to all of the above approaches is the method used to construct the sentiment index from the raw text, which is based on simple string matching. This applies to both deciding which articles are referring to which companies, and in assigning sentiment. For the sentiment index itself, a set of "word lists" are first created, either manually using financial knowledge Loughran and McDonald [2011] or drawing inspiration from the psychological literature Nyman et al. [2018]. The index is then generated by selecting a set of articles, which can be specific to a firm, industry or region, based on a string-match of the company name, and then counting the occurrence of words in each of the separate sentiment lists (commonly positive or negative). The problem with this approach is that it is very prone to incorporating noise into the index, through a range of potential avenues.

Firstly, a single article can mention a large number of separate companies/institutions, such as this example excerpt from Reuters:

> *"Stockholm-listed shares .OMXSPI dropped 0.4%, while Norwegian shares .OSEAX were hit by salmon producers Mowi and SalMar after the companies said the U.S. Department of Justice had issued subpoenas. The top gainer on the STOXX 600 was*

[4] Negative news by its nature almost always follows negative events, so isolating a causal effect from news sentiment to asset prices etc. is necessarily difficult.

Danish software company Simcorp (SIM.CO), up 3.8% after better-than-expected results."

Even in this small excerpt, multiple companies are referenced, including with both a positive and negative sentiment. Count-based measures cannot account for this, instead attributing the same sentiment score to all companies in the article. Using our approach to named entity recognition (introduced below), we sampled a set of 500 articles related to financial news from the Reuters newswire corpus, and found an average of 8.3 different companies mentioned per article, all of which will receive the same sentiment score for that article using the above approaches.[5]

Secondly, company names regularly overlap with non-industrial terms, such as in the headline

"Amazon rainforest close to irreversible tipping point'"

where the term "Amazon" is clearly a reference to the rainforest as opposed to the company. String-matching methods cannot account for this, and would assign the sentiment of the article to the organization.

Thirdly, the presence of negative/positive terms does not necessarily infer negativity/positivity. In the headline

"Tesla TSLA defies analysts expectations of disastrous quarter with earnings report"

the sentiment with respect to Tesla is clearly positive, despite the presence of the negative term "disastrous". There are also a range of other potential avenues for noise to enter existing methods, including coreferences to an organization (an article about IBM may refer to "the computing giant" after the first few sentences, rather than use the exact company name), and misspellings.

Proposed methodology

Given the rapid progress in natural language processing in tasks such as Named Entity Recognition (identifying spans of text in an article which refer to entities, and labelling

[5]Note that one simple method to mitigate some of this effect would be to split each article into sentences, and count only positive/negative terms appearing in the same sentence as an organization.

them with the type, such as “Company", “Country" etc.), sentiment analysis (detecting the sentiment of a section of text) and coreference resolution (detecting coreferences to an entity, such as when the word “he" refers to a person mentioned elsewhere in the text), it is of interest to see if the level of noise in financial/macro sentiment indices can be reduced, improving their accuracy both for forecasting and as a monitor of current market conditions. The approach we suggest in this paper looks like the following:

To begin with, entities in the text are identified, using named entity recognition (NER) and coreference resolution, Lee et al. [2018]. Spans labelled as countries or organizations are then assigned to their precise entities using entity-linking techniques Raiman and Raiman [2018], and an individual measure of the sentiment for that entity in each sentence is generated using aspect-based sentiment analysis Rietzler et al. [2019].

Named Entity Recognition

In this paper, we focus on the first step, of named entity recognition. In particular, we focus on the task of Nested NER, which focuses on recognizing and classifying entities that can be nested within each other, such as “United Kingdom” and “The Prime Minister of the United Kingdom” in Figure 4.1. Such entity structures, while very commonly occurring, cannot be handled by the predominant variant of NER models [McCallum and Li, 2003; Lample et al., 2016], which can only tag non-overlapping entities (i.e. the model would only identify “the United Kingdom" in this example, and would miss the larger entity “the Prime Minister of the United Kingdom"). Despite the focus of this paper being restricted to the nested NER task, the output of the architecture we propose is an “entity embedding" (a vector of dimension 300) for each entity (country, person etc.) mentioned in the text. In this paper, we utilise these embeddings to provide a prediction of the “type" of each entity (Person, Location, City etc.) - but looking ahead, they provide a simple mechanism entity linking, or predicting the sentiment associated with each entity.

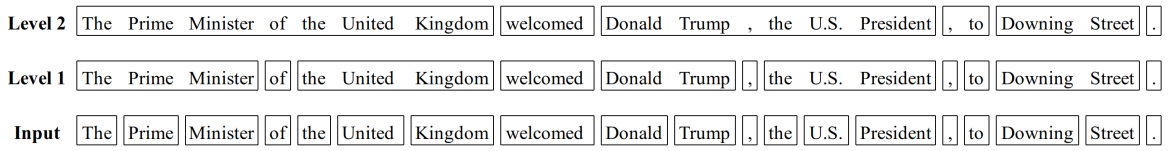
A number of approaches have been proposed for nested NER. Lu and Roth [2015] introduced a hypergraph representation which can represent overlapping mentions, which was further improved by Muis and Lu [2017], by assigning tags between each pair of consecutive

words, preventing the model from learning spurious structures (overlapping entity structures which are gramatically impossible). More recently, Katiyar and Cardie [2018] built on this approach, adapting an LSTM Hochreiter and Schmidhuber [1997] to learn the hypergraph directly, and Wang and Lu [2018] introduced a segmental hypergraph approach, which is able to incorporate a larger number of span based features, by encoding each span with an LSTM. However, to date, accuracy on this problem is relatively low, with a state-of-the-art F1 score of just 75 on the ACE 2005 corpus. On top of this, there is a dramatic drop in speed of the Nested NER algorithms relative to their flat NER counterparts, meaning processing large volumnes of news articles is computationally costly.

Our proposed approach decomposes nested NER into two stages. The input to the model is a series of “tokens"; generally these tokens are individual words as shown by the “Input" layer of Figure 4.1, though rare words may be split into two separate tokens. In the first step, the vector representations of these tokens are merged into entities (Level 1 in Figure 4.1). That is, we go from having separate vectors for “the", “United" and “Kingdom", to having a single vector representing this entire entity. These entities are then merged with other tokens or entities in higher levels. As opposed to being hard decisions, these merges are encoded as real-valued decisions (between 0.0 and 1.0), which enables a parameterized combination of word embeddings into entity embeddings at different levels. Once we have an entity embedding (vector) for each entity on each level, the embeddings can be used to make a prediction of the type (Person, Location etc.) of each entity identified. The model itself consists of feedforward neural network layers and is fully differentiable, thus it is straightforward to train with backpropagation.

In contrast to other methods, we only provide predictions of the entity type of the set of entity spans identified (as opposed to evaluating all possible combinations of spans of N tokens or less as in Wang and Lu [2018] and then taking those with the highest score). This results in faster decoding (decoding requires simply a single forward pass of the network).

Figure 4.1: Trained model's representation of nested entities, after thresholding the merge values, M (see section 2.1). Note that the merging of ", to" is a mistake by the model.



To test our approach on nested NER, we evaluate it on the ACE 2005 corpus (LDC2006T06) where it achieves a state-of-the-art F1 score of 74.6. This is further improved with contextual embeddings [Devlin et al., 2018] to 82.4, an overall improvement of close to 8 F1 points against the previous best approach trained on the same data, Wang and Lu [2018]. Our approach is also 60 times faster than its closest competitor, an important improvement for enabling generation of sentiment indices from a large corpus of news articles or documents. Additionally, we compare it against BiLSTM-CRFsHuang et al. [2015], the dominant flat NER paradigm, on Ontonotes (LDC2013T19) and demonstrate that its ability to predict nested structures does not impact performance in flat NER tasks as it achieves comparable results to the state of the art on this dataset.

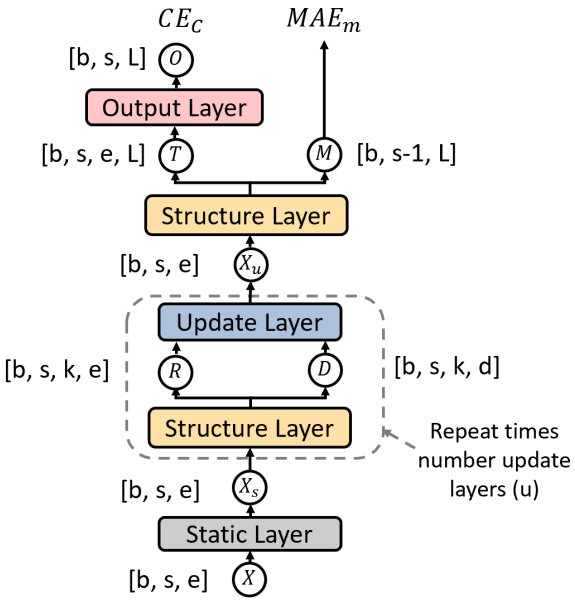
4.2 Network Architecture

4.2.1 Overview

The model decomposes nested NER into two stages. Firstly, it identifies the boundaries of the named entities at all levels of nesting (i.e. it identifies which tokens should be merged together to form a single entity, and which should be kept separate). These boundaries are defined by the tensor M in Figure 4.2, which is composed of real values between 0 and 1 (these real values are used to infer discrete split/merge decisions at test time, giving the nested structure of entities shown in Figure 4.1). We refer to this as predicting the "structure" of the NER output for the sentence. Secondly, given this structure, it produces embeddings (a single vector representation) for each entity, by combining the embeddings of smaller entities/tokens from previous levels (i.e. there will be an embedding for each rectangle in Figure 4.1). These

entity embeddings are used to label the entities identified, and could be used for downstream tasks including entity linking, coreference resolution and entity-specific sentiment scores.

Figure 4.2: Model architecture overview



An overview of the architecture used to predict the structure and labels is shown in Figure 4.2. The dimensions of each tensor are shown in square brackets in the figure. The input tensor, X, holds the token embeddings of dimension e, for every token in the input of sequence length, s. The first dimension, b, is the batch size. The **Static Layer** updates the token embeddings using contextual information, giving tensor X_s of the same dimension, $[b, s, e]$. That is, within the static layer, each token embedding is updated based on the embeddings of the tokens either side of it (the "context").

Next, for u repetitions, we go through a series of building the structure using the **Structure Layer**, and then use this structure to continue updating the individual token embeddings using the context in the **Update Layer**, giving an output X_u.

The updated token embeddings X_u are passed through the Structure Layer one last time, to give the final entity embeddings, T and structure, M. A feedforward **Output Layer** then gives the predictions of the label of each entity. For predictions of the sentiment of each entity, an additional output layer could be added to the network.

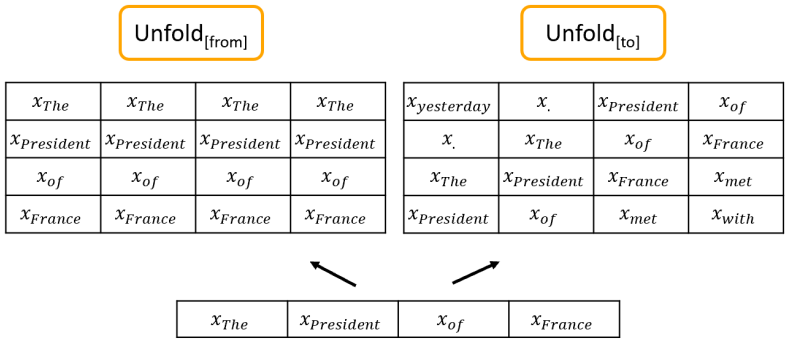
The structure is represented by the tensor M, of dimensions $[b, s-1, L]$. M holds, for every pair of adjacent words ($s-1$ given input length s) and every output level (L levels), a value between 0 and 1. A value close to 0 denotes that the two (adjacent) tokens/entities from the previous level are likely to be merged on this level to form an entity; nested entities emerge when entities from lower levels are used. Note that for each individual application of the Structure Layer, we are building multiple levels (L) of nested entities. That is, within each Structure Layer there is a loop of length L. By building the structure before the Update Layer, the updates to the token embeddings can utilize information about which entities each token is in, as well as neighbouring entities, as opposed to just using information about neighbouring tokens. As an example, the embedding of "the United Kingdom" on Level 1 of Figure 4.1 can use the information that there is an entity "the Prime Minister" two steps to its left in the sentence, and hence update its own embedding so that it becomes merged with the tokens to its left in Level 2.

4.2.2 Preliminaries

Before analysing each of the main layers of the network, we introduce two architectural building blocks, which are used multiple times throughout the architecture. The first one is the **Unfold operators**. Given that we process whole news articles in one batch (often giving a sequence_length (s) of 500 or greater) we do not allow each token in the sequence to consider every other token. Instead, we define a kernel of size k around each token, similar to convolutional neural networks [Kim, 2014], allowing it to consider the k/2 prior tokens and the k/2 following tokens.

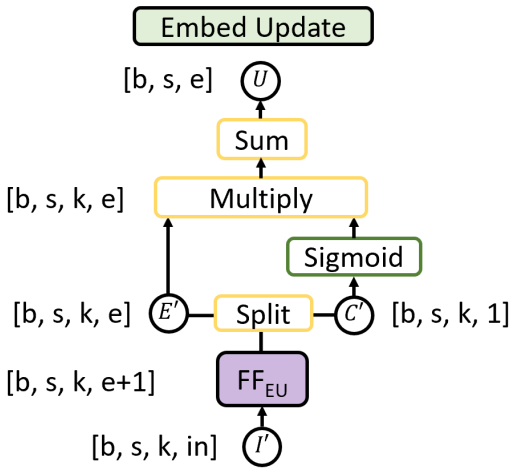
The unfold operators create kernels transforming tensors holding the word embeddings of shape [b, s, e] to shape [b, s, k, e]. $\text{unfold}_{[\text{from}]}$ simply tiles the embedding x of each token k times, and $\text{unfold}_{[\text{to}]}$ generates the k/2 token embeddings either side, as shown in Figure 4.3, for a kernel size k of 4. The first row of the $\text{unfold}_{[\text{to}]}$ tensor holds the two tokens before and the two tokens after the word "The", the second row the two before and after "President" etc. As we process whole articles, the unfold operators allow tokens to consider tokens from previous/following sentences.

Figure 4.3: Unfold Operators for the passage "... yesterday. The President of France met with ...". Each row in the matrices corresponds to the words "The", "President", "of" and "France" (top to bottom).



The second building block is the **Embed Update layer**, shown in Figure 4.4. This layer is used to update embeddings within the model using their context, and as such, can be thought of as equivalent in function to the residual update mechanism in Transformer Vaswani et al. [2017]. It is used in each of the Static Layer, Update Layer and Structure Layer from the main network architecture in Figure 4.2.

Figure 4.4: Embed Update layer



It takes an input I' of size $[b, s, k, in]$, formed using the unfold ops described above, where the last dimension in varies depending on the point in the architecture at which the layer is used. It passes this input through the feedforward NN FF_{EU}, giving an output of dimension $[b, s, k, e+1]$ (the network broadcasts over the last three dimensions of the input tensor). The

output is split into two. Firstly, a tensor E' of shape $[b, s, k, e]$, which holds, for each word in the sequence, k predictions of an updated word vector based on the k/2 words either side. Secondly, a weighting tensor C' of shape $[b, s, k, 1]$, which is scaled between 0 and 1 using the sigmoid function, and denotes how "confident" each of the k predictions is about its update to the word embedding. This works similar to an attention mechanism, allowing each token to focus on updates from the most relevant neighbouring tokens.[6] The output, U is then a weighted average of E':

$$U = sum_2(sigmoid(C') * E')$$

where sum_2 denotes summing across the second dimension of size k. U therefore has dimensions $[b, s, e]$ and contains the updated embedding for each word.

During training we initialize the weights of the network using the identity function. As a result, the default behaviour of FF_{EU} prior to training is to pass on the word embedding unchanged, which is then updated during via backpropagation. An example of the effect of the identity initialization is provided in the supplementary materials.

4.2.3 Static Layer

The static layer is a simple preliminary layer to update the embeddings for each token based on contextual information (i.e. based on the tokens either side of it), and as such, is very similar to a Transformer Vaswani et al. [2017] layer. This transforms the token embeddings from "static" embeddings, which are identical regardless of the context, to "contextual" embeddings, which represent the meaning of the word in the context of the sentence it appears. This is of particular importance for words such as "Washington", which can have multiple meanings (i.e. as the US capital with NER type "City", or as "George Washington", with NER type "Person"). This is of significance, as it overcomes one of the key limitations of the "string matching" techniques employed by previous approaches in the literature - we are now able

[6] The difference being that the weightings are generated using a sigmoid rather than a softmax layer, allowing the attention values to be close to one for multiple tokens.

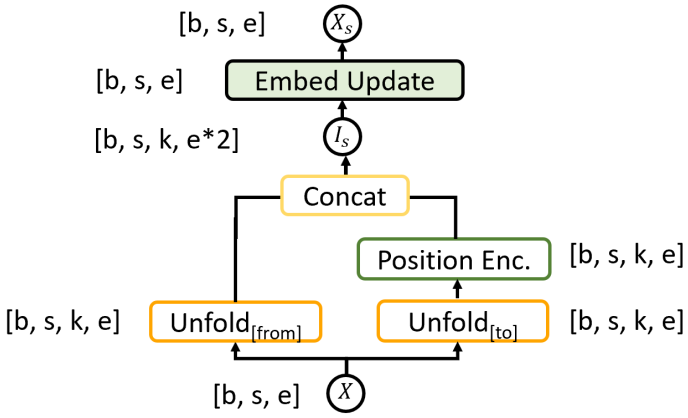
(theoretically) to differentiate entities beyond their string form, allowing more precise entity linking and sentiment indices.

To achieve this, following the unfold ops, a positional encoding (which communicates to the model the position in the sequence of each word) P of dimension e (we use a learned encoding) is added, giving tensor I_s:

$$I_s = concat(Unfold_{[from]}(X), Unfold_{[to]}(X) + P)$$

I_s is then passed through the Embed Update layer. In our experiments, we use a single static layer. There is no merging of embeddings into entities in the static layer.

Figure 4.5: Static Layer



4.2.4 Structure Layer

The Structure Layer is responsible for three tasks. Firstly, deciding which token embeddings should be merged at each level, expressed as real values between 0 and 1, and denoted M. Secondly, given these merge values M, deciding how the separate token embeddings should be combined in order to give the embeddings for each entity, T. Finally, for each token and entity, providing directional vectors D to the $k/2$ tokens either side, which are used to update each token embedding in the Update Layer based on its context. Intuitively, the directional vectors D can be thought of as encoding relations between entities - such as the relation

between an organization and its leader, or that between a country and its capital city (see Section 4.6.2 for an analysis of these relation embeddings).

In total then, we go within the structure layer from having a single embedding (vector) representation of each token (word), to having a single embedding of each **entity**. Having a single coherent representation of each entity allows a single coherent prediction of the type of each entity (rather than having to make separate predictions for each individual token and then combining), and in future work could be used for making a single prediction of the sentiment of each entity, or its coreferent spans.

Figure 4.6: Calculation of merging weight, directions and entities in Structure Layer

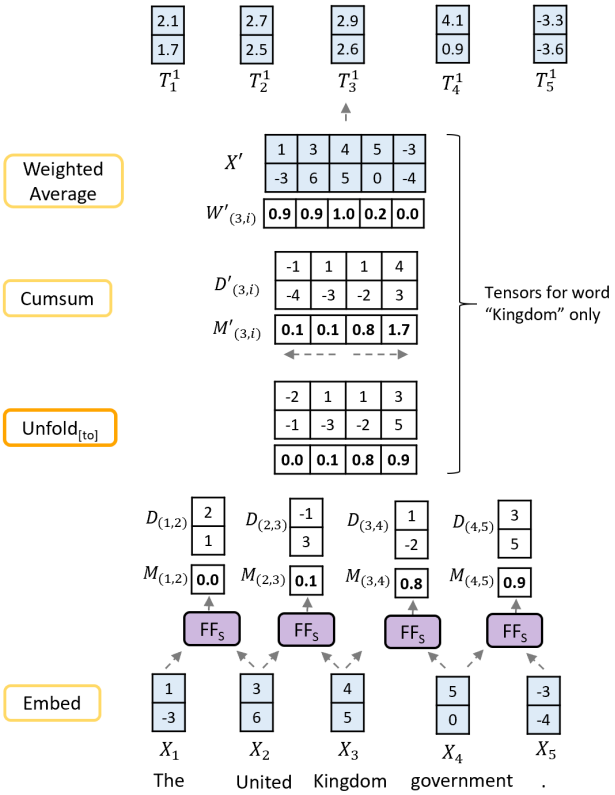


Figure 4.6 shows a minimal example of the calculation of D, M and T, with word embedding and directional vector dimensions $e = d = 2$, and kernel size, $k = 4$. We pass the embeddings (X) of each pair of adjacent words through a feedforward NN FF_S to give directions D [b, s-1, d] and merge values M [b, s-1, 1] between each pair. If FF_S predicts $M_{(1,2)}$ to be close to 0, this indicates that tokens 1 and 2 are part of the same entity on this

level. The $\text{unfold}_{[\text{to}]}$ op gives, for each word (we show only the unfolded tensors for the word "Kingdom" in Figure 4.6 for simplicity), D and M for pairs of words up to k/2 either side.

By taking both the left and right cumulative sum (cumsum) of the resulting two tensors from the center out (see grey dashed arrows in Figure 4.6 for direction of the two cumsum ops), we get directional vectors and merge values from the word "Kingdom" to the words before and after it in the phrase, $D'_{3,i}$ and $M'_{3,i}$ for $i = (1, 2, 4, 5)$. Note that we take the inverse of vectors $D_{(1,2)}$ and $D_{(2,3)}$ prior to the cumsum, as we are interested in the directions **from** the token "Kingdom" backwards to the tokens "United" and "The". The values $M'_{3,i}$ are converted to weights W' of dimension [b, s, k, 1] using the formula $W' = max(0, 1 - M')$[7], with the max operation ensuring the model puts a weight of zero on tokens in separate entities (see the reduction of the value of 1.7 in M' in Figure 4.6 to a weighting of 0.0). The weights are normalized to sum to 1, and multiplied with the unfolded token embeddings X' to give the entity embeddings T, of dimension [b, s, e]

$$T = \frac{W'}{sum_2(W')} * X'$$

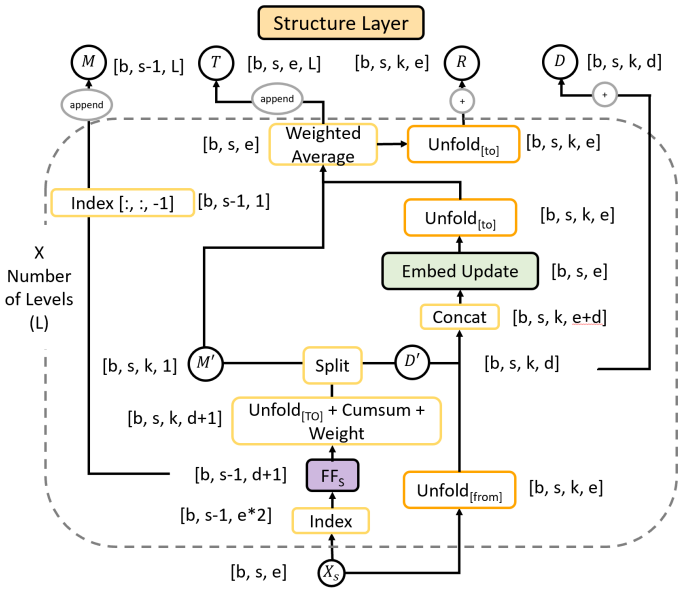
Consequently, the embeddings at the end of level 1 for the words "The", "United" and "Kingdom" (T^1_1, T^1_2 and T^1_3 respectively) are all now close to equal, and all have been formed from a weighted average of the three separate token embeddings. If $M_{(1,2)}$ and $M_{(2,3)}$ were precisely zero, and $M_{(3,4)}$ was precisely 1.0, then all three would be identical. In addition, on higher levels, the directions from other words to each of these three tokens will also be identical. In other words, the use of "directions"[8] allows the network to represent entities as a single embedding in a fully differentiable fashion, whilst keeping the sequence length constant.

Figure 4.6 shows just a single level from within the Structure Layer. The embeddings T are then passed onto the next level, allowing progressively larger entities to be formed by combining smaller entities from the previous levels.

[7] We use the notation D' to denote the unfolded version of tensor D, i.e. $D' = Unfold_{[to]}(D)$

[8] We use the term "directions" as we inverse the vectors to get the reverse direction, and cumsum them to get directions between tokens multiple steps away.

Figure 4.7: Structure Layer



The full architecture of the Structure Layer is shown in Figure 4.7. The main difference to Figure 4.6 is the additional use of Embed Update Layer, to decide how individual token/entity embeddings are combined together into a single entity. The reason for this is that if we are joining the words “The", “United" and “Kingdom" into a single entity, it makes sense that the joint vector should be based largely on the embeddings of “United" and “Kingdom", as “The" should add little information.The embeddings are unfolded (using the unfold$_{[\text{from}]}$ op) to shape $[b, s, k, e]$ and concatenated with the directions between words, D', to give the tensor of shape $[b, s, k, e + d]$. This is passed through the Embed Update layer, giving, for each word, a weighted and updated embedding, ready to be combined into a single entity (for unimportant words like “The", this embedding will have been reduced to close to zero, whereas for more important words in the entity, such as “United" or “Kingdom", the embeddings will have larger magnitudes). We use this tensor in place of tensor X in Figure 4.6, and multiply with the weights W' to give the new entity embeddings, T.

There are four separate outputs from the Structure Layer. The first, denoted by $\textcircled{T}$, is the entity embeddings from each of the levels concatenated together, giving a tensor of size [b, s, e, L]. The second output, $\textcircled{R}$, is a weighted average of the embeddings from different layers, of shape $[b, s, k, e]$. This will be used in the place of the unfold$_{[\text{to}]}$ tensor described above

as an input the the Update Layer. It holds, for each token in the sequence, embeddings of entities up to $k/2$ tokens either side. The third output, (D), will also be used by the Update Layer. It holds the directions of each token/entity to the $k/2$ tokens/entities either side. It is formed using the cumsum op, as shown in Figure 4.6. Finally, the fourth output, (M), stores the merge values for every level. It is used in the loss function, to directly incentivize the correct merge decisions at the correct levels.

4.2.5 Update Layer

The Update Layer is responsible for updating the **individual** token vectors, using the contextual information derived from outputs (R) and (D) of the Structure Layer. In other words, it serves the same purpose as the Static Layer, in that each embedding of each token is moved further away from its "static" representation (same for every occurrence of the token regardless of context) towards a "contextual" representation. For words with multiple meanings, such as the previously used example of "Washington", this moves our method further away from string-matching based techniques, and towards entity representations that allow a different prediction of the entity type dependent on the meaning of the tokens in the context of the paragraph.

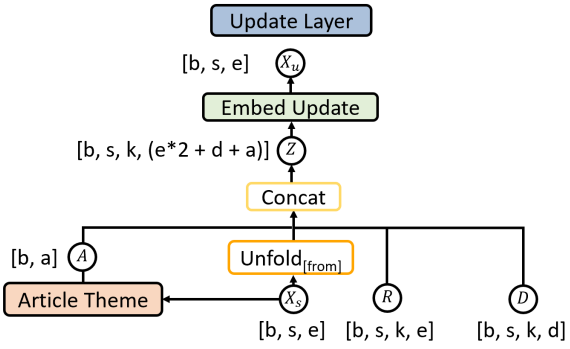
To do this, the update layer concatenates the two outputs of the structure layer together, along with the output of the unfold$_{[\text{from}]}$ op, X'_s, and with an article theme embedding A tensor, giving tensor Z of dimension [b, s, k, (e*2 + d + a)]. The article theme embedding is formed by passing every word in the article through a feedforward NN, and taking a weighted average of the outputs, giving a tensor of dimension $[b, a]$. This is then tiled[9] to dimension $[b, s, k, a]$, giving tensor A. A allows the network to adjust its contextual understanding of each token based on whether the article is on finance, sports, etc. Z is then passed through an Embed Update layer, giving an output X_u of shape $[b, s, e]$.

$$X_u = Embed\_Update(concat(X'_s, R, D, A))$$

[9]Tiling refers to simply repeating the tensor across **both** the sequence length s **and** kernel size k dimensions

We therefore update each word vector using four pieces of information. The original word embedding, a direction to a different token/entity, the embedding of that different token/entity, and the article theme.

Figure 4.8: Update Layer



The use of directional vectors D in the Update Layer can be thought of as an alternative to the positional encodings in Transformer Vaswani et al. [2017]. That is, the directional vectors give each token the positional information of the other tokens in the paragraph (for the token “United" in Figure 4.1 this corresponds to knowing that the token “welcomed" is one step to the right, the entity “Donald Trump" is two steps to the right etc.).

4.3 Implementation Details

4.3.1 Data Preprocessing

ACE 2005

ACE 2005 is a human annotated corpus for the task of nested named entity recognition. It is composed of around 180K tokens, with 7 distinct entity labels. The corpus labels include nested entities, allowing us to compare our model to the nested NER literature. The dataset is not pre-tokenized, so we carry out sentence and word tokenization using NLTK[10].

[10]NLTK is an open source library for NLP, which includes simple methods for splitting paragraphs of text into individual sentences, and the sentences into individual tokens/words; https://www.nltk.org/

OntoNotes

OntoNotes v5.0 is the largest corpus available for the basic NER task (where there is just a single layer of entities labelled), and is comprised of around 1.3M tokens, and 19 different entity labels. Although the labelling of the entities is not nested in OntoNotes, the corpus also includes labels for all noun phrases, which we train the network to identify concurrently. For training, we copy entities which are not contained within a larger nested entity onto higher levels, as shown in Figure 4.9.

Figure 4.9: OntoNotes Labelling

	OntoNotes Labels				**Model Targets**		
	L1	L2	L3		L1	L2	L3
Theresa	B-PER	O	O	Theresa	B-PER	B-PER	B-PER
May	I-PER	O	O	May	I-PER	I-PER	I-PER
visited	O	O	O	visited	O	O	O
Eastern	B-LOC	O	B-NP	Eastern	B-LOC	B-LOC	B-NP
Europe	I-LOC	O	I-NP	Europe	I-LOC	I-LOC	I-NP
and	O	O	I-NP	and	O	O	I-NP
the	O	B-NP	I-NP	the	O	B-NP	I-NP
USA	B-GPE	I-NP	I-NP	USA	B-GPE	I-NP	I-NP

Labelling

Both datasets denote the tokens that should be merged into a single entity using the “BIO" style of labelling, where a “B-" denotes the first token in the entity, an “I-" represents any middle/end tokens, and “O" is used for labelling tokens that are not part of an entity. This is demonstrated in Figure **??**. For both datasets, during training, we replace all “B-" labels with their corresponding “I-" label. At evaluation, all predictions which are the first word in a merged entity have the “B-" added back on. As the trained model’s merging weights, M, can take any value between 0 and 1, we have to set a cutoff at eval time when deciding which words are in the same entity. We perform a grid search over cutoff values using the dev set, with a value of 0.75 proving optimal.

4.3.2 Loss function

The model is trained to predict the correct merge decisions, held in the tensor M of dimension [b, s-1, L] and the correct entity class labels (Person, City etc.) given these decisions, C. The merge decisions are trained directly using the mean absolute error (MAE):

$$MAE_M = \frac{sum(|M - \hat{M}|)}{(b * s * L)}$$

This is then weighted by a scalar w_M, and added to the usual Cross Entropy (CE) loss from the predictions of the classes, CE_C, giving a final loss function of the form:

$$Loss = (w_M * MAE_M) + CE_C$$

In experiments we set the weight on the merge loss, w_M to 0.5.

4.3.3 Evaluation

Following previous literature, for both the ACE and OntoNotes datasets, we use a strict F1 measure, where an entity is only considered correct if both the label and the span are correct.

ACE 2005

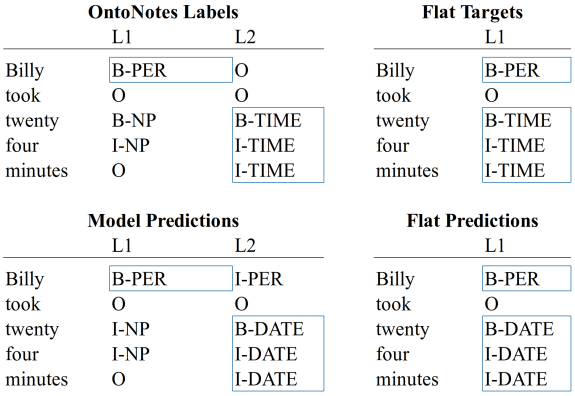
For the ACE corpus, the default metric in the literature Wang et al. [2018]; Ju et al. [2018]; Wang and Lu [2018] does not include sequential ordering of nested entities (as many architectures do not have a concept of ordered nested outputs). As a result, an entity is considered correct if it is present in the target labels, regardless of which layer the model predicts it on.

OntoNotes

NER models evaluated on OntoNotes are trained to label the 19 entities, and not noun phrases (NP). To provide as fair as possible a comparison, we consequently flatten all labelled entities

into a single column. As 96.5% of labelled entities in OntoNotes do not contain a NP nested inside, this applies to only 3.5% of the dataset.

Figure 4.10: OntoNotes Targets



The method used to flatten the targets is shown in Figure 4.10. The OntoNotes labels include a named entity (TIME), in the second column, with the NP “twenty-four" minutes nested inside. Consequently, we take the model's prediction from the second column as our prediction for this entity. This provides a fair comparison to existing NER models, as all entities are included, and if anything, disadvantages our model, as it not only has to predict the correct entity, but do so on the correct level. That said, the NP labels provide additional information during training, which may give our model an advantage over flat NER models, which do not have access to these labels.

4.3.4 Training and HyperParameters

We performed a small amount of hyperparameter tuning across dropout, learning rate, distance embedding size d, and number of update layers u. We set dropout at 0.1, the learning rate to 0.0005, d to 200, and u to 3. For full hyperparameter details see the supplementary materials. The number of levels, L, is set to 3, with a kernel size k of 10 on the first level, 20 on the second, and 30 on the third (we increase the kernel size gradually for computational efficiency as first level entities are extremely unlikely to be composed of more than 10 tokens, whereas higher level nested entities may be larger). Training took around 10 hours for OntoNotes, and around 6 hours for ACE 2005, on an Nvidia 1080 Ti.

For experiments without language model (LM) embeddings, we used pretrained Glove embeddings Pennington et al. [2014] of dimension 300[11]. Following Strubell et al. [2017], we added a “CAP features" embedding of dimension 20, denoting if each word started with a capital letter, was all capital letters, or had no capital letters. For the experiments with LM embeddings, we used the implementations of the BERT Devlin et al. [2018] and ELMO Peters et al. [2018] models from the Flair Akbik et al. [2018] project[12]. BERT and ELMO are both large neural network based models, which have been trained to predict masked out words on large corpuses of web text; as masking out of a word does not require human labelling of the dataset, the corpus can be generated automatically, and as a result is not limited in size. Both models provide “contextual" embeddings; in other words, part of the work required of the Static Layer in our model is already done, providing the model with a “hot start". We do not finetune the BERT and ELMO models, but take their embeddings as given.

4.4 Results

4.4.1 ACE 2005

On the ACE 2005 corpus, we begin our analysis of our model's performance by comparing to models which do not use the POS tags as additional features, and which use non-contextual word embeddings. These are shown in the top section of Table 4.1. The previous state-of-the-art F1 of 72.2 was set by Ju et al. [2018], using a series of stacked BiLSTM layers, with CRF decoders on top of each of them. Our model improves this result with an F1 of 74.6 (avg. over 5 runs with std. dev. of 0.4). This also brings the performance into line with Wang et al. [2018] and Wang and Lu [2018], which concatenate embeddings of POS tags with word embeddings as an additional input feature.

Given the recent success on many tasks using contextual word embeddings, we also evaluate performance using the output of pre-trained BERT Devlin et al. [2018] and ELMO Peters

[11] Glove embeddings are a set of pretrained “static" embeddings (single vector representation for each word regardless of context) which are a common starting point for many neural network models in the NLP literature.

[12] https://github.com/zalandoresearch/flair/

Table 4.1: ACE 2005

Model	**Pr.**	**Rec.**	**F1**
Multigraph + MS Muis and Lu [2017]	69.1	58.1	63.1
RNN + hyp Katiyar and Cardie [2018]	70.6	70.4	70.5
BiLSTM-CRF stacked Ju et al. [2018]	74.2	70.3	72.2
LSTM + forest [POS] Wang et al. [2018]	74.5	71.5	73.0
Segm. hyp [POS] Wang and Lu [2018]	76.8	72.3	74.5
Merge and Label	75.1	74.1	**74.6**
LM embeddings			
Merge and Label [ELMO]	79.7	78.0	78.9
Merge and Label [BERT]	82.7	82.1	**82.4**
LM + OntoNotes			
DyGIE Luan et al. [2019]			**82.9**

et al. [2018] models as input embeddings. This leads to a significant jump in performance to 78.9 with ELMO, and 82.4 with BERT (both avg. over 5 runs with 0.4 and 0.3 std. dev. respectively), an overall increase of 8 F1 points from the previous state-of-the-art. Finally, we report the concurrently published result of Luan et al. [2019], in which they use ELMO embeddings, and additional labelled data (used to train the coreference part of their model and the entity boundaries) from the larger OntoNotes dataset.

A secondary advantage of our architecture relative to those models which require construction of a hypergraph or CRF layer is its decoding speed, as decoding requires only a single forward pass of the network. As such it achieves a speed of 9468 words per second (w/s) on an Nvidia 1080 Ti GPU, relative to a reported speed of 157 w/s for the closest competitor model of Wang and Lu [2018], a sixty fold advantage.

4.4.2 OntoNotes

As mentioned previously, given the caveats that our model is trained to label all NPs as well as entities, and must also predict the correct layer of an entity, the results in Table 4.2 should be seen as indicative comparisons only. Using non-contextual embeddings, our model achieves a test F1 of 87.59. To our knowledge, this is the first time that a nested NER architecture

has performed comparably to BiLSTM-CRFs Huang et al. [2015] (which have dominated the named entity literature for the last few years) on a flat NER task.

Given the larger size of the OntoNotes dataset, we report results from a single iteration, as opposed to the average of 5 runs as in the case of ACE05.

Table 4.2: OntoNotes NER

Model	**F1**
BiLSTM-CRF Chiu and Nichols [2016]	86.28
ID-CNN Strubell et al. [2017]	86.84
BiLSTM-CRF Strubell et al. [2017]	86.99
Merge and Label	**87.59**
LM embeddings or extra data	
BiLSTM-CRF lex Ghaddar and Langlais [2018]	87.95
BiLSTM-CRF with CVT Clark et al. [2018]	88.81
Merge and Label [BERT]	89.20
BiLSTM-CRF Flair Akbik et al. [2018]	**89.71**

We also see a performance boost from using BERT embeddings, pushing the F1 up to 89.20. This falls slightly short of the state-of-the-art on this dataset, achieved using character-based Flair Akbik et al. [2018] contextual embeddings.

4.5 Ablations

To better understand the results, we conducted a small ablation study. The affect of including the **Static Layer** in the architecture is consistent across both datasets, yielding an improvement of around 2 F1 points; the updating of the token embeddings based on context seems to allow better merge decisions for each pair of tokens. Next, we look at the method used to update entity embeddings prior to combination into larger entities in the **Structure Layer**. In the described architecture, we use the Embed Update mechanism (see Figure 4.7), allowing embeddings to be changed dependent on which other embeddings they are about to be combined with. We see that this yields a significant improvement on both tasks of around 4 F1 points, relative to passing each embedding through a linear layer.

The inclusion of an "article theme" embedding, used in the **Update Layer**, has little effect on the ACE05 data. but gives a notable improvement for OntoNotes. Given that the distribution of types of articles is similar for both datasets, we suggest this is due to the larger size of the OntoNotes set allowing the model to learn an informative article theme embedding without overfitting.

Table 4.3: Architecture Ablations

	ACE05	**OntoNotes**
Static Layer		
with	74.6	87.59
without	73.1	85.22
Embed Combination		
Linear	70.2	83.96
Embed Update	74.6	87.59
Article Embedding		
with	74.5	87.59
without	74.6	85.60
Sentence boundaries		
with	70.8	86.30
without	74.6	87.59

Next, we investigate the impact of allowing the model to attend to tokens in neighbouring sentences (we use a set kernel size of 30, allowing each token to consider up to 15 tokens prior and 15 after, regardless of sentence boundaries). Ignoring sentence boundaries boosts the results on ACE05 by around 4 F1 points, whilst having a smaller affect on OntoNotes. We hypothesize that this is due to the ACE05 task requiring the labelling of pronominal entities, such as "he" and "it", which is not required for OntoNotes. The coreference needed to correctly label their type is likely to require context beyond the sentence.

4.6 Discussion

4.6.1 Entity Embeddings

As our architecture merges multi-word entities, it not only outputs vectors of each word, but also for all entities - the tensor T. To demonstrate this, Table 4.4 shows the ten closest

entity vectors in the OntoNotes test data to the phrases “the United Kingdom”, “Arab Foreign Ministers" and “Israeli Prime Minister Ehud Barak”.[13]

Table 4.4: Entity Embeddings Nearest Neighbours

the United Kingdom	**Arab Foreign Ministers**	**Israeli Prime Minister Ehud Barak**
the United States	Palestinian leaders	Italian President Francesco Cossiga
the Tanzania United Republic	Yemeni authorities	French Foreign Minister Hubert Vedrine
the Soviet Union	Palestinian security officials	Palestinian leader Yasser Arafat
the United Arab Emirates	Israeli officials	Iraqi leader Saddam Hussein
the Hungary Republic	Canadian auto workers	Likud opposition leader Ariel Sharon
Myanmar	Palestinian sources	UN Secretary General Kofi Annan
Shanghai	many Jewish voters	Russian President Vladimir Putin
China	Lebanese Christian lawmakers	Syrian Foreign Minister Faruq al - Shara
Syria	Israeli and Palestinian negotiators	PLO leader Arafat
the Kyrgystan Republic	A Canadian bank	Libyan leader Muammar Gaddafi

Given that the OntoNotes NER task considers countries and cities as GPE (Geo-Political Entities), the nearest neighbours in the left hand column are expected. The nearest neighbours of “Arab Foreign Ministers” and “Israeli Prime Minister Ehud Barak” are more interesting, as there is no label for groups of people or jobs for the task.[14] Despite this, the model produces good embedding-based representations of these complex higher level entities. This is a significant result; it demonstrates that our model has been effective in providing a meaningful representation of each entity in the input, which can provide the basis for the further tasks we have mentioned, including coreference resolution, entity linking and entity-level sentiment detection.

4.6.2 Directional Embeddings

The representation of the **relationship between** each pair of words/entities as a vector is primarily a mechanism used by the model to update the word/entity vectors. However, the resulting vectors, corresponding to output Ⓓ of the Structure Layer, may also provide useful information for downstream tasks such as knowledge base population.

[13]Note that we exclude from the 10 nearest neighbours identical entities from higher levels. I.e. if “the United Kingdom” is kept as a three token entity, and not merged into a larger entity on higher levels, we do not report the same phrase from all levels in the nearest neighbours.

[14]The phrase “Israeli Prime Minister Ehud Barak" would have “Israeli" labelled as NORP, and “Ehud Barak" labelled as PERSON in the OntoNotes corpus.

To demonstrate the directional embeddings, Table 4.5 shows the ten closest matches for the **direction** between “the president" and “the People’s Bank of China". The network has clearly picked up on the relationship of an employee to an organisation.

Table 4.5: Directional Embeddings Nearest Neighbours

the president	**→ the People’s Bank of China**
the chairman	→ the SEC
Vice Minister	→ the Ministry of Foreign Affairs
Chairman	→ the People’s Association of Taiwan
Deputy Chairman	→ the TBAD Women’s Division
Chairman	→ the KMT
Vice President	→ the Military Commission of the CCP
vice-chairman	→ the CCP
Associate Justices	→ the Supreme Court of the United States
Chief Editor	→ Taiwan’s contemporary monthly
General Secretary	→ the Communist Party of China

Table 4.5 also provides further examples of the network merging and providing intuitive embeddings for multi-word entities.

4.7 Conclusion

Current methods for building sentiment/uncertainty indices from raw text in finance and economics use very basic string-matching techniques, based on lists of words representing positive and negative concepts. We have suggested that such indices could be improved using more sophisticated natural language processing techniques, including identifying entities, coreferences and sentiment using neural networks. We have presented a novel neural network architecture for smoothly merging token embeddings in a sentence into entity embeddings, across multiple levels for the initial task in this pipeline of nested NER, setting a new state-of-the-art F1 score by close to 8 F1 points, whilst also being competitive at flat NER, and processing articles around 60 times faster than existing alternatives. Despite being trained only for NER, the architecture provides intuitive embeddings for a variety of multi-word entities, a step which we suggest could prove useful for a variety of the other tasks in the index-building pipeline, including entity linking, coreference resolution and aspect-based sentiment analysis. Future work will focus these latter stages of the text-processing pipeline,

and on building sentiment indices for individual companies, countries and sectors using the new approach.

Bibliography

Akbik, A., Blythe, D., and Vollgraf, R. (2018). Contextual string embeddings for sequence labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1638–1649. Association for Computational Linguistics.

Baker, S. R., Bloom, N., and Davis, S. J. (2016). Measuring Economic Policy Uncertainty*. *The Quarterly Journal of Economics*, 131(4):1593–1636.

Blinder, A. S., Ehrmann, M., Fratzscher, M., de Haan, J., and Jansen, D.-J. (2008). Central Bank Communication and Monetary Policy: A Survey of Theory and Evidence. (170).

Chiu, J. P. C. and Nichols, E. (2016). Named entity recognition with bidirectional lstm-cnns. *Transactions of the Association for Computational Linguistics*, 4:357–370.

Clark, K., Luong, M., Manning, C. D., and Le, Q. V. (2018). Semi-supervised sequence modeling with cross-view training. *EMNLP*.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Ghaddar, A. and Langlais, P. (2018). Robust lexical features for improved neural network named-entity recognition. *CoRR*, abs/1806.03489.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

Huang, Z., Xu, W., and Yu, K. (2015). Bidirectional LSTM-CRF models for sequence tagging. *CoRR*, abs/1508.01991.

Ju, M., Miwa, M., and Ananiadou, S. (2018). A neural layered model for nested named entity recognition. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1446–1459. Association for Computational Linguistics.

Katiyar, A. and Cardie, C. (2018). Nested named entity recognition revisited. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 861–871. Association for Computational Linguistics.

Kim, Y. (2014). Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., and Dyer, C. (2016). Neural architectures for named entity recognition. In *Proceedings of NAACL-HLT*, pages 260–270.

Lee, K., He, L., and Zettlemoyer, L. (2018). Higher-order coreference resolution with coarse-to-fine inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 687–692, New Orleans, Louisiana. Association for Computational Linguistics.

Loughran, T. and McDonald, B. (2011). When is a liability not a liability? textual analysis, dictionaries, and 10-ks. *The Journal of Finance*, 66(1):35–65.

Lu, W. and Roth, D. (2015). Joint mention extraction and classification with mention hypergraphs. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 857–867. Association for Computational Linguistics.

Luan, Y., Wadden, D., He, L., Shah, A., Ostendorf, M., and Hajishirzi, H. (2019). A general framework for information extraction using dynamic span graphs. *CoRR*, abs/1904.03296.

McCallum, A. and Li, W. (2003). Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 188–191. Association for Computational Linguistics.

Muis, A. O. and Lu, W. (2017). Labeling gaps between words: Recognizing overlapping mentions with mention separators. In *Proceedings of the 2017 Conference on Empirical*

Methods in Natural Language Processing, pages 2608–2618. Association for Computational Linguistics.

Nyman, R., Kapadia, S., Tuckett, D., Gregory, D., Ormerod, P., and Smith, R. (2018). News and narratives in financial systems: exploiting big data for systemic risk assessment. Bank of England working papers 704, Bank of England.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *In EMNLP*.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. In *Proc. of NAACL*.

Raiman, J. and Raiman, O. (2018). Deeptype: Multilingual entity linking by neural type system evolution. *CoRR*, abs/1802.01021.

Rietzler, A., Stabinger, S., Opitz, P., and Engl, S. (2019). Adapt or get left behind: Domain adaptation through bert language model finetuning for aspect-target sentiment classification.

Soo, C. (2013). Quantifying animal spirits: News media and sentiment in the housing market. *SSRN Electronic Journal*.

Strubell, E., Verga, P., Belanger, D., and McCallum, A. (2017). Fast and accurate sequence labeling with iterated dilated convolutions. *EMNLP*.

Sturm, J.-E. and De Haan, J. (2011). Does central bank communication really lead to better forecasts of policy decisions? new evidence based on a taylor rule model for the ecb. *Review of World Economics*, 147(1):41–58.

Tetlock, P. C. (2007). Giving content to investor sentiment: The role of media in the stock market. *The Journal of Finance*, 62(3):1139–1168.

Tetlock, P. C., Saar-Tsechansky, M., and Macsassky, S. (2008). More than words: Quantifying language to measure firms' fundamentals. *The Journal of Finance*, 63(3):1437–1467.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

Wang, B. and Lu, W. (2018). Neural segmental hypergraphs for overlapping mention recognition. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 204–214. Association for Computational Linguistics.

Wang, B., Lu, W., Wang, Y., and Jin, H. (2018). A neural transition-based model for nested mention recognition. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1011–1017. Association for Computational Linguistics.

Appendix A: Chapter 1

A.1 Further Results

Table A.1: Train, validation and test QL for one-step-ahead prediction (all models)

		DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
GARCH	Tr	0.887	1.142	0.915	1.130	1.081	0.618	1.299	-0.032	0.855	1.016
	V	0.674	0.487	0.793	0.891	1.234	0.350	1.202	0.031	0.360	0.480
	Te	0.353	0.287	0.248	0.483	0.825	0.016	0.899	0.220	0.478	0.753
TARCH	Tr	0.870	1.129	0.896	1.109	1.062	0.609	1.276	-0.047	0.856	1.004
	V	0.631	0.484	0.744	0.853	1.203	0.354	1.194	0.024	0.371	0.483
	Te	0.315	0.262	0.204	0.488	0.814	0.003	0.896	0.201	0.474	0.720
HAR	Tr	0.854	1.041	0.865	0.918	1.049	0.596	1.260	-0.101	0.817	0.965
	V	0.649	0.408	0.768	0.684	1.170	0.311	1.156	-0.045	0.320	0.461
	Te	0.315	0.197	0.176	0.206	0.809	0.019	0.891	0.164	0.454	0.697
NN	Tr	0.856	1.034	0.867	0.924	1.040	0.576	1.252	-0.101	0.816	0.963
	V	0.649	0.386	0.772	0.681	1.164	0.286	1.150	-0.050	0.310	0.459
	Te	0.303	0.164	0.162	0.192	0.810	-0.010	0.884	0.168	0.455	0.702
NN (Ret)	Tr	0.853	1.026	0.874	0.907	1.034	0.572	1.248	-0.110	0.817	0.975
	V	0.625	0.381	0.751	0.678	1.160	0.281	1.140	-0.053	0.309	0.466
	Te	0.275	0.153	0.143	0.186	0.799	-0.014	0.872	0.160	0.456	0.707
NN-MA	Tr	0.851	1.033	0.859	0.916	1.041	0.575	1.256	-0.104	0.814	0.964
	V	0.647	0.386	0.770	0.678	1.164	0.285	1.149	-0.047	0.308	0.458
	Te	0.287	0.163	0.148	0.198	0.808	-0.012	0.894	0.164	0.455	0.702
NN-MA (Ret)	Tr	0.836	1.027	0.862	0.904	1.035	0.571	1.250	-0.115	0.815	1.011
	V	0.623	0.378	0.752	0.671	1.160	0.281	1.142	-0.054	0.308	0.466
	Te	0.258	0.160	0.136	0.190	0.799	-0.019	0.871	0.156	0.455	0.752
CNN	Tr	0.852	1.039	0.861	0.921	1.047	0.581	1.261	-0.112	0.840	0.973
	V	0.647	0.389	0.774	0.681	1.172	0.292	1.153	-0.045	0.317	0.464
	Te	0.311	0.175	0.160	0.212	0.815	-0.009	0.888	0.177	0.457	0.712
CNN (Ret)	Tr	0.844	1.034	0.848	0.907	1.038	0.575	1.255	-0.113	0.815	0.980
	V	0.617	0.388	0.740	0.683	1.166	0.287	1.142	-0.057	0.316	0.463
	Te	0.288	0.163	0.160	0.198	0.808	-0.013	0.880	0.162	0.462	0.730
CNN-MA	Tr	0.850	1.035	0.867	0.913	1.040	0.579	1.257	-0.113	0.834	0.967
	V	0.638	0.385	0.766	0.676	1.168	0.289	1.147	-0.049	0.314	0.461
	Te	0.303	0.171	0.167	0.205	0.814	-0.013	0.880	0.175	0.458	0.716
CNN-MA (Ret)	Tr	0.847	1.029	0.862	0.899	1.037	0.578	1.250	-0.110	0.819	0.961
	V	0.618	0.382	0.743	0.678	1.161	0.288	1.140	-0.057	0.314	0.458
	Te	0.286	0.160	0.152	0.199	0.807	-0.016	0.869	0.162	0.461	0.706
RNN	Tr	0.900	1.067	0.927	0.939	1.061	0.613	1.292	-0.062	0.846	1.108
	V	0.643	0.384	0.767	0.672	1.165	0.286	1.157	-0.051	0.305	0.382
	Te	0.301	0.157	0.155	0.196	0.813	-0.012	0.883	0.165	0.449	0.795
RNN (Ret)	Tr	0.867	1.096	0.864	0.924	1.051	0.594	1.285	-0.096	0.837	1.031
	V	0.616	0.377	0.740	0.657	1.159	0.280	1.148	-0.059	0.306	0.447
	Te	0.245	0.152	0.106	0.186	0.797	-0.018	0.870	0.159	0.455	0.698
RNN-MA	Tr	0.862	1.039	0.905	0.916	1.050	0.576	1.260	-0.089	0.820	0.965
	V	0.640	0.383	0.763	0.668	1.165	0.284	1.149	-0.050	0.306	0.460
	Te	0.285	0.163	0.157	0.203	0.806	-0.014	0.881	0.164	0.454	0.704
RNN-MA (Ret)	Tr	0.837	1.027	0.847	0.903	1.039	0.570	1.250	-0.122	0.834	0.962
	V	0.613	0.376	0.737	0.655	1.160	0.278	1.143	-0.057	0.303	0.454
	Te	0.263	0.147	0.105	0.181	0.796	-0.026	0.870	0.155	0.451	0.685

Table A.2: Train, validation and test MSE for one-step-ahead prediction (all models)

		DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
GARCH	Tr	5.190	6.840	4.580	5.250	4.050	2.420	6.830	0.550	3.120	3.950
	V	1.640	1.020	2.060	3.670	2.420	0.800	2.940	0.210	0.320	0.860
	Te	4.970	0.790	2.090	0.560	1.010	0.200	1.130	0.230	0.590	1.850
TARCH	Tr	4.200	4.490	3.490	3.880	3.400	2.190	5.660	0.530	2.540	2.430
	V	1.430	0.910	1.710	2.490	2.070	0.790	2.690	0.190	0.330	0.970
	Te	4.840	0.740	1.980	0.620	0.970	0.190	1.130	0.200	0.550	1.620
HAR	Tr	4.480	3.430	3.830	2.790	3.550	2.250	5.630	0.460	1.930	1.560
	V	1.480	0.640	1.770	2.260	1.840	0.660	2.020	0.130	0.320	0.850
	Te	5.290	0.710	2.110	0.180	1.000	0.190	1.160	0.160	0.490	1.700
NN	Tr	6.132	2.767	2.260	2.548	3.166	2.047	3.279	0.464	1.724	1.196
	V	1.425	0.519	1.408	1.762	1.545	0.521	1.446	0.123	0.298	0.673
	Te	4.887	0.681	1.975	0.214	1.201	0.242	1.578	0.166	0.553	2.392
NN (Ret)	Tr	2.145	1.974	2.987	1.250	2.377	1.866	3.039	0.245	0.826	0.833
	V	1.295	0.353	1.430	0.984	1.244	0.398	1.414	0.110	0.297	0.690
	Te	4.495	0.645	1.855	0.172	1.169	0.208	1.271	0.141	0.644	2.071
NN-MA	Tr	4.582	3.179	2.666	2.679	2.635	1.816	3.425	0.397	2.934	1.113
	V	1.379	0.500	1.572	1.808	1.539	0.508	1.456	0.123	0.296	0.684
	Te	4.800	0.680	2.138	0.203	1.130	0.224	1.388	0.162	0.527	2.529
NN-MA (Ret)	Tr	3.078	2.193	2.205	1.079	2.545	1.846	3.549	0.241	1.116	0.854
	V	1.262	0.357	1.318	1.024	1.259	0.407	1.490	0.112	0.301	0.695
	Te	4.748	0.675	1.787	0.164	1.050	0.205	1.246	0.146	0.583	1.944
CNN	Tr	4.379	3.375	3.580	2.589	3.188	2.136	5.576	0.487	3.027	1.554
	V	1.373	0.518	1.599	1.765	1.576	0.525	1.858	0.120	0.298	0.796
	Te	4.818	0.668	2.189	0.206	1.156	0.209	1.274	0.160	0.584	2.500
CNN (Ret)	Tr	3.134	2.720	3.037	2.144	3.084	2.119	4.522	0.413	1.980	1.508
	V	1.087	0.417	1.313	1.363	1.496	0.471	1.632	0.114	0.352	0.872
	Te	5.435	0.659	1.907	0.261	1.030	0.201	1.247	0.155	0.549	2.501
CNN-MA	Tr	4.283	3.377	3.532	2.540	3.292	2.216	5.571	0.499	1.929	1.530
	V	1.347	0.516	1.579	1.766	1.595	0.520	1.857	0.119	0.295	0.792
	Te	4.939	0.658	2.200	0.201	1.067	0.203	1.267	0.159	0.523	2.374
CNN-MA (Ret)	Tr	3.133	2.729	2.991	2.610	3.000	2.130	4.657	0.436	1.409	1.248
	V	1.124	0.410	1.275	1.345	1.458	0.463	1.681	0.113	0.315	0.866
	Te	5.021	0.643	2.007	0.273	1.073	0.213	1.461	0.160	0.615	1.939
RNN	Tr	4.047	3.630	3.423	2.910	3.353	2.028	5.608	0.511	0.623	0.628
	V	1.373	0.538	1.624	1.937	1.525	0.511	1.591	0.124	0.287	0.224
	Te	4.826	0.738	1.978	0.185	1.138	0.204	1.393	0.161	0.507	1.624
RNN (Ret)	Tr	5.294	1.981	3.595	1.701	2.645	1.788	4.638	0.436	2.471	0.581
	V	1.243	0.352	1.351	1.074	1.096	0.333	1.283	0.112	0.287	0.226
	Te	4.803	0.642	1.920	0.177	1.217	0.194	1.135	0.157	0.459	1.566
RNN-MA	Tr	3.934	2.959	4.228	2.556	2.991	1.871	5.364	0.441	1.374	1.602
	V	1.258	0.438	1.528	1.750	1.309	0.331	1.504	0.118	0.283	0.807
	Te	4.828	0.658	2.003	0.227	1.020	0.206	1.492	0.223	0.518	2.095
RNN-MA (Ret)	Tr	4.714	1.989	2.271	1.451	2.159	1.736	3.950	0.386	2.207	1.181
	V	1.147	0.336	1.266	0.922	0.927	0.237	1.167	0.106	0.280	0.638
	Te	4.658	0.669	1.951	0.159	1.133	0.195	1.060	0.153	0.441	1.487

Table A.3: Train, validation and test QL for two-step-ahead prediction

		DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
HAR	Tr	0.880	1.075	0.893	0.937	1.086	0.633	1.304	-0.089	0.833	1.004
	V	0.672	0.452	0.787	0.715	1.200	0.345	1.185	-0.034	0.347	0.523
	Te	0.388	0.272	0.259	0.242	0.841	0.056	0.925	0.180	0.485	0.752
NN	Tr	0.879	1.059	0.887	0.933	1.068	0.601	1.287	-0.090	0.830	1.011
	V	0.665	0.417	0.783	0.708	1.193	0.308	1.170	-0.043	0.325	0.515
	Te	0.364	0.224	0.221	0.226	0.834	0.013	0.906	0.180	0.482	0.757
NN (Ret)	Tr	0.868	1.055	0.882	0.923	1.062	0.595	1.283	-0.086	0.826	1.017
	V	0.652	0.410	0.770	0.703	1.188	0.304	1.164	-0.050	0.326	0.525
	Te	0.331	0.218	0.200	0.225	0.826	0.010	0.901	0.175	0.484	0.748
CNN	Tr	0.875	1.064	0.893	0.935	1.075	0.610	1.294	-0.085	0.834	1.012
	V	0.670	0.422	0.793	0.719	1.202	0.314	1.177	-0.043	0.328	0.519
	Te	0.355	0.228	0.249	0.238	0.847	0.016	0.915	0.187	0.492	0.769
CNN (Ret)	Tr	0.869	1.062	0.883	0.932	1.069	0.607	1.293	-0.089	0.832	1.012
	V	0.650	0.421	0.770	0.717	1.201	0.310	1.178	-0.046	0.331	0.520
	Te	0.367	0.218	0.231	0.238	0.838	0.011	0.916	0.184	0.494	0.771
RNN	Tr	0.891	1.100	0.941	0.954	1.086	0.639	1.323	-0.511	0.893	1.036
	V	0.661	0.410	0.783	0.697	1.195	0.311	1.169	-0.047	0.321	0.405
	Te	0.347	0.218	0.222	0.226	0.830	0.013	0.911	0.181	0.480	0.756
RNN (Ret)	Tr	0.884	1.093	0.921	0.942	1.088	0.629	1.320	-0.083	0.860	1.104
	V	0.645	0.408	0.762	0.693	1.192	0.305	1.169	-0.053	0.325	0.510
	Te	0.313	0.210	0.191	0.217	0.823	0.010	0.901	0.175	0.485	0.791

Table A.4: Train, validation and test QL for five-step-ahead prediction

		DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
HAR	Tr	0.937	1.122	0.947	0.984	1.141	0.686	1.365	-0.045	0.870	1.059
	V	0.734	0.535	0.846	0.787	1.256	0.413	1.259	-0.009	0.384	0.570
	Te	0.509	0.353	0.380	0.300	0.882	0.105	0.965	0.214	0.528	0.797
NN	Tr	0.959	1.120	0.950	0.983	1.113	0.649	1.341	-0.048	0.860	1.048
	V	0.716	0.482	0.839	0.774	1.248	0.361	1.237	-0.014	0.339	0.580
	Te	0.482	0.317	0.355	0.285	0.870	0.057	0.936	0.216	0.513	0.781
NN (Ret)	Tr	0.914	1.103	0.933	0.986	1.112	0.649	1.341	-0.047	0.847	1.048
	V	0.718	0.480	0.839	0.771	1.247	0.362	1.240	-0.017	0.338	0.584
	Te	0.486	0.318	0.346	0.292	0.871	0.056	0.945	0.211	0.520	0.760
CNN	Tr	0.920	1.105	0.942	0.981	1.115	0.650	1.345	-0.051	0.846	1.058
	V	0.717	0.487	0.848	0.772	1.256	0.361	1.246	-0.015	0.341	0.585
	Te	0.498	0.305	0.371	0.299	0.880	0.057	0.942	0.212	0.531	0.808
CNN (Ret)	Tr	0.911	1.106	0.929	0.978	1.113	0.646	1.347	-0.061	0.844	1.057
	V	0.727	0.487	0.850	0.771	1.262	0.365	1.251	-0.017	0.343	0.584
	Te	0.481	0.311	0.352	0.306	0.877	0.054	0.951	0.209	0.532	0.787
RNN	Tr	0.987	1.193	0.995	1.007	1.126	0.682	1.390	-0.009	0.843	1.148
	V	0.706	0.471	0.831	0.761	1.247	0.353	1.228	-0.019	0.338	0.439
	Te	0.515	0.317	0.385	0.290	0.875	0.056	0.936	0.216	0.513	0.828
RNN (Ret)	Tr	0.941	1.162	0.976	0.991	1.140	0.675	1.388	-0.037	0.880	1.082
	V	0.707	0.467	0.823	0.756	1.243	0.349	1.231	-0.016	0.338	0.559
	Te	0.525	0.320	0.389	0.287	0.868	0.060	0.951	0.218	0.510	0.777

A.2 Parameter Options

Table A.5: Parameter options of the best-performing RNN models for one-day-ahead prediction

RNN

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
num_explanatory_vars	1	1	1	1	1	1	1	1	1	1
exp_var_1_name	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar
exp_var_1_lags	1	1	1	1	1	1	1	1	1	1
hidden_size	6	10	3	10	10	6	10	10	3	6
num_layers	1	1	1	1	1	2	1	2	1	1
num_steps	5	5	10	5	5	5	5	5	5	30
cell_type	LSTM	LSTM	LSTM	LSTM	B_RNN	LSTM	LSTM	LSTM	LSTM	B_RNN
final_layer_initializer	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no
initializer_stdev	0.1	1	0.1	1	1	0.1	0.1	1	0.1	0.1
batch_length	50	50	50	50	50	50	50	50	50	50
num_epochs	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000
learning_rate	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.001	0.0001
dropout_keep_prob	1	1	1	1	1	1	1	1	1	1
feature_scaling	False	False	False	False	False	False	False	False	False	False
loss_function	QL	QL	QL	QL	QL	QL	QL	QL	QL	QL
optimizer_choice	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De
decrease_learning_rate	False	False	False	False	False	False	False	False	False	False
early_stopping	True	True	True	True	True	True	True	True	True	True
early_stopping_number	600	600	600	600	600	600	600	600	600	600
give_up_number	5	5	5	5	5	5	5	5	5	5

RNN (Ret)

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
num_explanatory_vars	2	2	2	2	2	2	2	2	2	2
exp_var_1_name	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar
exp_var_2_name	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns
exp_var_1_lags	1	1	1	1	1	1	1	1	1	1
exp_var_2_lags	1	1	1	1	1	1	1	1	1	1
hidden_size	10	10	6	10	10	3	10	10	3	6
num_layers	1	1	2	1	1	1	2	2	1	2
num_steps	10	10	5	5	5	5	5	10	5	10
cell_type	GRU	GRU	GRU	LSTM	GRU	LSTM	LSTM	LSTM	LSTM	LSTM
final_layer_initializer	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no
initializer_stdev	0.1	0.1	1	0.1	1	1	1	0.1	0.1	0.1
batch_length	50	50	50	50	50	50	50	50	50	50
num_epochs	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000
learning_rate	0.005	0.005	0.005	0.001	0.005	0.005	0.005	0.001	0.005	0.001
dropout_keep_prob	1	1	1	1	1	1	1	1	1	1
feature_scaling	False	False	False	False	False	False	False	False	False	False
loss_function	QL	QL	QL	QL	QL	QL	QL	QL	QL	QL
optimizer_choice	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De
decrease_learning_rate	False	False	False	False	False	False	False	False	False	False
early_stopping	True	True	True	True	True	True	True	True	True	True
early_stopping_number	600	600	600	600	600	600	600	600	600	600
give_up_number	5	5	5	5	5	5	5	5	5	5

RNN-MA

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
num_explanatory_vars	2	2	2	2	2	2	2	2	2	2
exp_var_1_name	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar
exp_var_1_lags	1	1	1	1	1	1	1	1	1	1
num_MA_lags	1	1	1	1	1	1	1	1	1	1
hidden_size	3	3	10	10	10	10	6	6	10	6
num_layers	2	2	2	1	1	2	2	2	1	1
num_steps	5	10	10	5	5	10	5	5	10	5
cell_type	GRU	LSTM	B_RNN	B_RNN	B_RNN	B_RNN	GRU	LSTM	LSTM	B_RNN
final_layer_initializer	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no
initializer_stdev	1	0.1	0.1	0.1	0.1	1	0.1	1	0.1	1
batch_length	50	50	50	50	50	50	50	50	50	50
num_epochs	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000
learning_rate	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.005
dropout_keep_prob	1	1	1	1	1	1	1	1	1	1
feature_scaling	False	False	False	False	False	False	False	False	False	False
loss_function	QL	QL	QL	QL	QL	QL	QL	QL	QL	QL
optimizer_choice	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De
decrease_learning_rate	False	False	False	False	False	False	False	False	False	False
early_stopping	True	True	True	True	True	True	True	True	True	True
early_stopping_number	600	600	600	600	600	600	600	600	600	600
delay_MA_training	True	True	True	True	True	True	True	True	True	True
MA_start_epoch	Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex
give_up_number	5	5	10	5	5	5	5	5	5	5

RNN-MA (Ret)

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
num_explanatory_vars	2	2	2	2	2	2	2	2	2	2
exp_var_1_name	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar
exp_var_2_name	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns
exp_var_1_lags	1	1	1	1	1	1	1	1	1	1
exp_var_2_lags	1	1	1	1	1	1	1	1	1	1
num_MA_lags	1	1	1	1	1	1	1	1	1	1
hidden_size	6	6	10	10	6	6	10	10	6	6
num_layers	1	1	1	1	1	1	1	1	1	1
num_steps	5	5	5	10	10	5	5	10	10	5
cell_type	LSTM	LSTM	GRU	B_RNN	GRU	LSTM	LSTM	GRU	GRU	LSTM
final_layer_initializer	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no
initializer_stdev	1	1	1	1	0.1	1	1	1	0.1	0.1
batch_length	50	50	50	50	50	50	50	50	50	50
num_epochs	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000
learning_rate	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.005	0.005
dropout_keep_prob	1	1	1	1	1	1	1	1	1	1
feature_scaling	False	False	False	False	False	False	False	False	False	False
loss_function	QL	QL	QL	QL	QL	QL	QL	QL	QL	QL
optimizer_choice	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De
decrease_learning_rate	False	False	False	False	False	False	False	False	False	False
early_stopping	True	True	True	True	True	True	True	True	True	True
early_stopping_number	600	600	600	600	600	600	600	600	600	600
delay_MA_training	True	True	True	True	True	True	True	True	True	True
MA_start_epoch	Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex	Flex
give_up_number	5	5	5	5	5	5	5	5	5	5

Note: The table shows the parameters of the models which achieved the lowest validation error for each index, for one-step-ahead forecasts. The loss function used for training was the QL loss.

Table A.6: Explanation of the parameter options which can be chosen for the RNN

num_explanatory_vars	The number of explanatory variables used for this model. One refers to models which just use RV, and two adds returns
exp_var_1_name	The name of the first explanatory variable
exp_var_2_name	The name of the second explanatory variable
exp_var_1_lags	The number of lags of the first explanatory variable (always RV) used for prediction
exp_var_2_lags	The number of lags of the first explanatory variable (always Returns) used for prediction
num_MA_lags	The number of moving average lags used for prediction
hidden_size	The dimensions of the RNN cell
num_layers	The number of layers in the network.
num_steps	The number of previous steps for which the gradient of the loss function with respect to the weights is calculated when updating the weight values. This is, in effect, a limit on the number of lags the RNN can use for prediction.
cell_type	The type of cell in the RNN. Chosen from "B_RNN" (which refers to the basic RNN cell), "LSTM" (which refers to the long-short term memory cell) and "GRU" (which refers to the Gated Recurrent Network)
final_layer_initializer	The function from which we draw random values to initialise the final layer. "tr_no" refers to a trucated normal function, with values further than 2 standard deviations from the mean dropped and repicked. We always set a mean of 0.
initializer_stdev	The standard deviation of the function used for the initialization of the final layer
batch_length	The batch size which we use when training the network
num_epochs	The maximum number of epochs (an epoch is one cycle through all the data) used for training
learning_rate	The learning rate we apply during gradient descent
dropout_keep_prob	Dropout refers to a technique in which we randomly constrain a certain number of nodes to be zero at a training step, used to avoid overfitting. We do not include results with dropout applied in this paper, so the probability is set to 1.
feature_scaling	Feature scaling refers to scaling of the input features to lie between 0 and 1. This is useful if the explanatory variables are very different in average scale. We do not apply feature scaling in this work.
loss_function	The loss function used for training the network
optimizer_choice	The choice of optimization function. We train all networks using the basic gradient descent algorithm (Gr_De) in this paper
decrease_learning_rate	This refers to the option to decrease the learning rate during training, which can be used to try to improve the convergence. It is not utilised in the results reported in this paper
early_stopping	Whether or not gradient descent is stopped if the train error is no longer improving
early_stopping_number	The number of epochs in which the train error has to fail to improve before training is stopped
delay_MA_training	Whether or not we delay the training of the weights associated with the MA variables, to allow the values of the residuals to settle
MA_start_epoch	The epoch at which we start optimizing the network weights associated with the MA variables, used if "delay_MA_training" is true
give_up_number	The number of times we will restart the training of the network with the parameter values, if it fails to converge after the random initialization

Table A.7: Parameter options of the best-performing NN-MA (Ret) model for one-day-ahead prediction

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
num_explanatory_vars	2	2	2	2	2	2	2	2	2	2
exp_var_1_name	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar
exp_var_2_name	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns
exp_var_1_lags	3	7	7	3	5	3	5	3	3	3
exp_var_2_lags	3	3	3	3	3	3	3	3	3	3
num_MA_lags	2	2	2	2	2	2	2	2	2	2
num_layers	1	1	1	2	2	1	2	1	2	2
hidden1_units	30	30	30	10	10	3	30	100	10	100
hidden2_units	0	0	0	100	30	0	100	0	100	100
initialization_function	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no	tr_no
hidden_weights_stdev	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
final_layer_stdev	10	10	2	10	10	10	10	2	10	10
batch_size	100	100	100	100	100	100	100	100	100	100
num_epochs	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000
learning_rate	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004	0.00004
feature_scaling	False	False	False	False	False	False	False	False	False	False
loss_function	QL	QL	QL	QL	QL	QL	QL	QL	QL	QL
optimizer_choice	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De
decrease_learning_rate	False	False	False	False	False	False	False	False	False	False
early_stopping	True	True	True	True	True	True	True	True	True	True
early_stopping_number	600	600	600	600	600	600	600	600	600	600
delay_MA_training	True	True	True	True	True	True	True	True	True	True
MA_start_epoch	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000
give_up_number	5	5	5	5	5	5	5	5	5	5

Note: The table shows the parameters of the models which achieved the lowest validation error for each index, for one-step-ahead forecasts. The loss function used for training was the QL loss.

Table A.8: Explanation of the parameter options which can be chosen for the NN, if not included in the RNN options above

hidden1_units	The number of nodes in the first layer of the neural network
hidden2_units	The number of nodes in the second layer of the neural network
initialization_function	The function used to initialize the random weight values prior to training. We use the same function for the hidden and final layer in this paper, but allow the standard deviations to vary across these layers
hidden_weights_stdev	The initialization standard deviation for the hidden layers of the network.

Table A.9: Parameter options of the best-performing CNN-MA (Ret) model for one-day-ahead prediction

	DJIA	Nas	S&P	Russ	CAC	FTSE	DAX	AllO	HanS	Nikk
num_explanatory_vars	2	2	2	2	2	2	2	2	2	2
exp_var_1_name	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar	RVar
exp_var_2_name	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns	Returns
exp_var_1_lags	8	4	8	4	8	6	4	8	4	6
exp_var_2_lags	4	4	4	4	4	4	4	4	4	4
MA_lags	4	4	4	4	4	4	4	4	4	4
var_1_num_filters	200	200	200	200	100	200	200	200	30	5
var_2_num_filters	100	100	100	100	100	100	100	100	100	100
MA_num_filters	100	100	100	100	100	100	100	100	100	100
var_1_filter_sizes	[3, 5]	[2]	[2]	[2]	[3, 5]	[2]	[2]	[2]	[2, 3, 4]	[2, 3]
var_2_filter_sizes	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]
MA_filter_sizes	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]	[2, 3]
weights_stdev	0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.01	0.1	0.1
final_layer_initializer	uniform	uniform	uniform	uniform	uniform	uniform	uniform	uniform	uniform	uniform
initializer_range	(-0.5,0.5)	(-0.5,0.5)	(-0.5,0.5)	(-0.5,0.5)	(-0.5,0.5)	(-0.5,0.5)	(-0.5,0.5)	(-0.5,0.5)	(-0.5,0.5)	(-0.5,0.5)
batch_size	100	100	100	100	100	100	100	100	100	100
num_epochs	6000	6000	6000	6000	6000	6000	6000	6000	6000	6000
learning_rate	0.00007	0.0005	0.0005	0.0005	0.0005	0.0005	0.0005	0.0005	0.0005	0.0005
dropout_keep_prob	1	1	1	1	1	1	1	1	1	1
feature_scaling	False	False	False	False	False	False	False	False	False	False
loss_function	QL	QL	QL	QL	QL	QL	QL	QL	QL	QL
optimizer_choice	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De	Gr_De
decrease_learning_rate	False	False	False	False	False	False	False	False	False	False
early_stopping	True	True	True	True	True	True	True	True	True	True
early_stopping_number	600	600	600	600	600	600	600	600	600	600
delay_MA_training	True	True	True	True	True	True	True	True	True	True
MA_start_epoch	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000

Note: The table shows the parameters of the models which achieved the lowest validation error for each index, for one-step-ahead forecasts. The loss function used for training was the QL loss.

Table A.10: Explanation of the parameter options which can be chosen for the CNN, if not included in the RNN or NN options above

var1_num_filters	The number of filters of each filter size used for the first explanatory variable (RV). A filter refers to a set of linear regression weights.
var2_num_filters	The number of filters of each filter size used for the second explanatory variable (Returns)
MA_num_filters	The number of filters of each filter size used for the moving average explanatory variable
var_1_filter_sizes	A list of the sizes of filters to include for the first explanatory variable (RV). The filter size refers to the number of weights used in this particular linear regression.
var_2_filter_sizes	A list of the sizes of filters to include for the second explanatory variable (Returns)
MA_filter_sizes	A list of the sizes of filters to include for the moving average explanatory variable
final_layer_initializer	The initialization function for the final layer. “uniform" refers to the uniform function
initializer_range	The range for the initialization function, used if the function if the final_layer_initialiser is “uniform"

Appendix B: Chapter 2

B.1 Hyperparameter options

In Section 2.3.1, we listed the sets of parameters which we varied during the hyperparameter stage. Tables B.1 and B.2 below list the remaining hyperparameters, which were kept constant throughout all experiments.

Table B.1: Additional neural network hyperparameter options

Hyperparameter	Options
Optimizer	Stochastic Gradient Descent
Training steps	20000
Learning Rate Decay	half_every_4000_steps
Batch Size	50
Bias initialization	(constant, 0.1)
Feature scaling	Standard Deviation Scale
Target scaling	Standard Deviation Scale

In Table B.1, the “optimizer" refers to the optimization algorithm used to update the network parameters after backpropagation. Stochastic Gradient Descent is one of the simplest of the available algorithms, but tends to work well when the dataset is relatively small, and the noise levels high. The number of “training steps" denotes the number of times we update the network’s weights during training, using a single batch each time. A “learning rate decay" of half_every_4000_steps denotes that every 4000 training steps, we half the learning rate. Decaying the learning rate in this fashion generally improves convergence, as the network weights are able to adjust to finer regularities in the training data. The “batch size" is the number of observations we process in a single training step, and the “bias initialization" of (constant, 0.1) indicates that we initialize all bias values in the network to 0.1 prior to training. Finally, both features and targets are scaled prior to training using the following formula:

$$x = \frac{x - \mu(x)}{\sigma(x)}$$

Table B.2: Additional recurrent neural network hyperparameter options

Hyperparameter	Options
Optimizer	Stochastic Gradient Descent
Training steps	12000
Learning rate decay	half_every_4000_steps
Weight initialization	(truncated_normal, 0.05)
Batch size	2
Bias initialization	(constant, 0.1)
Feature scaling	Standard Deviation Scale
Target scaling	Standard Deviation Scale

Table B.2 lists the constant hyperparameter options for the recurrent neural network experiments. The only value not described above is the "weight initialization", which refers to the distribution from which we initialize the weights of the network prior to training. We draw the values from a normal distribution with mean 0 and standard deviation 0.05, discarding and redrawing values which fall further than two standard deviations from the mean.

B.2 Full Results Tables

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	3.800 (0.798) [0.476]	3.189 (0.777) [0.440]	1.327 (0.697) [0.327]	2.516 (0.710) [0.312]	2.112 (0.697) [0.282]	0.924 (0.589) [0.161]	2.099 (0.780) [0.388]	1.764 (0.764) [0.366]	0.858 (0.590) [0.215]	2.232 (0.817) [0.434]	1.875 (0.794) [0.395]	0.893 (0.608) [0.211]
ind	0.742 (0.552) **[0.276]**	0.680 (0.523) [0.265]	0.637 (0.472) [0.191]	0.512 (0.386) **[0.157]**	0.453 (0.350) **[0.136]**	0.390 (0.337) [0.186]	0.427 (0.328) **[0.152]**	0.367 (0.294) [0.139]	**0.244** (0.185) [0.078]	0.560 (0.393) **[0.066]**	0.518 (0.364) [0.072]	0.389 (0.302) [0.119]
joint	1.747 (1.530) [1.435]	1.401 (1.218) [1.107]	0.702 (0.537) [0.280]	2.241 (2.372) [2.323]	1.891 (1.967) [1.911]	0.874 (0.807) [0.674]	0.594 (0.634) [0.543]	0.464 (0.484) [0.395]	0.266 (0.219) [0.116]	1.099 (1.645) [1.592]	0.947 (1.396) [1.343]	0.465 (0.607) [0.522]
joint_a	1.383 (1.359) [1.267]	1.135 (1.066) [0.952]	0.681 (0.514) [0.251]	1.661 (2.078) [2.034]	1.401 (1.696) [1.644]	0.663 (0.646) [0.505]	0.631 (0.632) [0.555]	0.499 (0.452) [0.361]	0.267 (0.204) [0.095]	0.684 (1.369) [1.339]	0.576 (1.120) [1.087]	0.318 (0.455) [0.382]
joint_a_r	1.496 (1.314) [1.145]	1.218 (1.045) [0.869]	0.647 (0.499) [0.270]	2.025 (2.163) [2.109]	1.672 (1.771) [1.713]	0.687 (0.682) [0.557]	0.581 (0.508) [0.392]	0.445 (0.374) [0.252]	0.264 (0.200) [0.089]	0.669 (1.353) [1.325]	0.570 (1.137) [1.106]	0.273 (0.410) [0.346]
sep	0.716 (0.564) [0.331]	0.603 (0.475) [0.252]	0.503 **(0.390)** **[0.174]**	0.485 (0.375) [0.184]	0.421 (0.325) [0.155]	0.303 (0.248) [0.113]	0.421 (0.327) [0.153]	**0.340 (0.276)** **[0.133]**	**0.244 (0.184)** [0.076]	0.542 (0.382) [0.070]	0.477 (0.344) [0.092]	0.326 (0.258) [0.106]
sep_a	**0.709 (0.529)** [0.307]	**0.579 (0.448)** **[0.230]**	**0.490** (0.408) [0.227]	**0.400 (0.316)** [0.196]	**0.364 (0.278)** [0.164]	**0.280 (0.223)** **[0.103]**	**0.416 (0.322)** [0.161]	0.344 (0.279) [0.136]	0.245 (0.189) [0.083]	**0.187 (0.151)** [0.090]	**0.166 (0.134)** **[0.070]**	**0.153 (0.124)** [0.064]
sep_a_r	0.891 (0.740) [0.425]	0.775 (0.638) [0.334]	0.545 (0.446) [0.235]	0.527 (0.398) [0.190]	0.451 (0.356) [0.170]	0.303 (0.241) [0.107]	0.444 (0.342) [0.157]	0.367 (0.295) [0.139]	0.245 **(0.184)** **[0.075]**	0.199 (0.168) [0.102]	0.180 (0.148) [0.074]	0.161 (0.126) **[0.062]**

Table B.3: Results for DGF1 (linear model) with NGF1 (homoskedastic noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	0.380 (0.342) [0.096]	0.332 (0.298) [0.083]	0.207 (0.181) [0.050]	0.334 (0.283) [0.079]	0.283 (0.241) [0.068]	0.132 (0.119) [0.034]	0.382 (0.295) **[0.052]**	0.322 (0.253) [0.045]	0.143 (0.133) [0.026]	0.335 (0.247) [0.037]	0.282 (0.210) [0.033]	0.122 (0.100) [0.020]
ind	0.416 (0.423) [0.204]	0.276 (0.279) [0.109]	0.152 (0.139) [0.043]	0.214 (0.276) [0.127]	0.135 (0.170) [0.063]	0.076 (0.083) [0.026]	0.253 (0.345) [0.176]	0.143 (0.227) [0.075]	0.060 (0.077) **[0.017]**	0.125 (0.142) [0.040]	0.090 (0.086) [0.022]	0.066 (0.062) [0.013]
joint	0.582 (0.523) [0.264]	0.420 (0.385) [0.201]	0.183 (0.156) [0.059]	0.332 (0.408) [0.258]	0.231 (0.301) [0.196]	0.101 (0.093) [0.032]	0.493 (0.495) [0.248]	0.329 (0.346) [0.183]	0.093 (0.085) [0.037]	0.223 (0.272) [0.173]	0.142 (0.170) [0.105]	0.074 (0.064) [0.018]
joint_a	0.527 (0.408) [0.168]	0.399 (0.316) [0.131]	0.195 (0.163) [0.056]	0.299 (0.280) [0.163]	0.212 (0.205) [0.123]	0.109 (0.101) [0.040]	0.335 (0.260) [0.106]	0.222 (0.185) [0.097]	0.083 (0.076) [0.035]	0.163 (0.149) [0.078]	0.105 (0.090) [0.036]	0.075 (0.064) [0.019]
joint_a_r	**0.256 (0.223)** [0.102]	**0.191 (0.169)** **[0.073]**	**0.135 (0.122)** **[0.040]**	**0.123 (0.119)** [0.054]	**0.102 (0.097)** [0.039]	**0.075 (0.073)** [0.027]	**0.112 (0.103)** [0.053]	**0.077 (0.072)** **[0.035]**	**0.051 (0.048)** [0.018]	**0.087 (0.079)** [0.034]	**0.075 (0.064)** [0.020]	**0.061 (0.052)** [0.016]
sep	0.548 (0.420) [0.154]	0.424 (0.330) [0.119]	0.201 (0.184) [0.052]	0.356 (0.303) [0.147]	0.235 (0.206) [0.092]	0.107 (0.103) [0.029]	0.559 (0.393) [0.100]	0.391 (0.268) [0.067]	0.137 (0.108) [0.042]	0.158 (0.137) [0.053]	0.115 (0.096) [0.027]	0.070 (0.062) [0.015]
sep_a	0.535 (0.386) [0.108]	0.404 (0.309) [0.096]	0.212 (0.202) [0.067]	0.321 (0.270) [0.127]	0.220 (0.191) [0.080]	0.109 (0.105) [0.031]	0.536 (0.345) [0.064]	0.402 (0.267) [0.074]	0.140 (0.115) [0.048]	0.158 (0.140) [0.060]	0.114 (0.096) [0.030]	0.070 (0.064) [0.016]
sep_a_r	0.285 (0.260) **[0.095]**	0.229 (0.225) [0.088]	0.189 (0.191) [0.066]	0.142 (0.131) **[0.043]**	0.114 (0.110) **[0.034]**	0.086 (0.090) **[0.025]**	0.146 (0.138) [0.075]	0.106 (0.105) [0.056]	0.068 (0.064) [0.023]	0.097 (0.085) **[0.022]**	0.080 (0.073) **[0.016]**	0.062 (0.057) **[0.012]**

Table B.4: Results for DGF1 (linear model) with NGF2 (heteroskedastic noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	0.505 (0.471) [0.149]	0.426 (0.398) [0.126]	0.184 (0.176) [0.059]	0.535 (0.470) [0.282]	0.451 (0.402) [0.240]	0.195 (0.208) [0.130]	0.425 (0.336) [0.059]	0.359 (0.285) [0.048]	0.158 (0.134) [0.025]	0.549 (0.466) [0.268]	0.464 (0.400) [0.228]	0.204 (0.214) [0.134]
ind	0.344 (0.471) [0.339]	0.165 (0.196) [0.133]	**0.073 (0.066) [0.036]**	1.026 (0.822) [0.349]	0.923 (0.724) [0.275]	0.319 (0.437) [0.308]	0.145 (0.162) [0.057]	0.111 (0.116) [0.035]	0.081 (0.081) [0.015]	1.100 (0.892) [0.389]	0.971 (0.779) [0.314]	0.219 (0.417) [0.314]
joint	0.625 (0.592) [0.349]	0.390 (0.383) [0.232]	0.102 (0.091) [0.049]	0.887 (0.738) [0.352]	0.707 (0.618) [0.313]	0.377 (0.426) [0.270]	0.212 (0.265) [0.167]	0.131 (0.133) [0.068]	0.082 (0.074) [0.018]	0.862 (0.778) [0.408]	0.625 (0.580) [0.317]	0.237 (0.332) [0.234]
joint_a	0.381 (0.288) [0.126]	0.255 (0.203) [0.099]	0.113 (0.098) [0.057]	0.609 (0.532) [0.285]	0.491 (0.476) [0.278]	0.283 (0.358) [0.242]	0.174 (0.155) [0.078]	0.118 (0.104) [0.045]	0.083 (0.072) [0.018]	0.512 (0.381) [0.167]	0.373 (0.312) [0.169]	0.163 (0.232) [0.161]
joint_a_r	**0.180 (0.163) [0.092]**	**0.132 (0.116) [0.066]**	0.094 (0.080) [0.051]	0.477 (0.587) [0.405]	0.363 (0.503) [0.356]	0.219 (0.367) [0.269]	**0.100 (0.089)** [0.036]	**0.083 (0.074)** [0.024]	**0.068 (0.060)** [0.018]	0.190 (0.372) [0.285]	0.109 (0.264) [0.207]	0.036 (0.079) [0.061]
sep	0.370 (0.305) [0.156]	0.232 (0.187) [0.089]	0.095 (0.085) [0.044]	0.665 (0.609) [0.329]	0.362 (0.270) [0.100]	0.120 (0.093) [0.030]	0.162 (0.155) [0.074]	0.119 (0.109) [0.043]	0.077 (0.070) [0.016]	0.632 (0.560) [0.290]	0.355 (0.242) [0.079]	0.114 (0.083) [0.016]
sep_a	0.363 (0.278) [0.119]	0.272 (0.215) [0.097]	0.106 (0.100) [0.055]	0.428 (0.291) **[0.068]**	0.314 (0.212) [0.052]	0.123 (0.096) [0.030]	0.169 (0.153) [0.072]	0.118 (0.106) [0.040]	0.078 (0.071) [0.015]	0.433 (0.280) [0.057]	0.294 (0.195) [0.030]	0.116 (0.084) [0.017]
sep_a_r	0.190 (0.193) [0.117]	0.150 (0.160) [0.099]	0.093 (0.090) [0.051]	**0.136 (0.130)** [0.077]	**0.077 (0.072) [0.041]**	**0.042 (0.037) [0.021]**	0.104 (0.095) **[0.029]**	0.089 (0.082) **[0.020]**	0.072 (0.066) **[0.013]**	**0.058 (0.054) [0.029]**	**0.037 (0.034) [0.019]**	**0.023 (0.020) [0.010]**

Table B.5: Results for DGF1 (linear model) with NGF3 (skewed noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	2.463 (2.607) [2.532]	2.200 (2.247) [2.166]	1.506 (1.312) [1.194]	0.892 (0.636) [0.214]	0.824 (0.599) [0.161]	0.680 (0.505) **[0.051]**	0.666 (0.504) **[0.105]**	0.645 (0.490) **[0.084]**	0.603 (0.462) **[0.039]**	0.637 (0.466) **[0.054]**	0.632 (0.461) **[0.045]**	0.621 (0.449) **[0.022]**
ind	0.903 (0.712) [0.467]	0.765 (0.578) [0.292]	0.676 (0.492) **[0.177]**	0.607 (0.437) **[0.142]**	0.541 (0.408) **[0.137]**	0.463 (0.371) [0.148]	0.455 (0.355) [0.129]	0.407 (0.320) [0.101]	0.330 (0.264) [0.075]	0.340 (0.278) [0.104]	0.301 (0.245) [0.077]	0.238 (0.201) [0.044]
joint	0.809 (0.654) [0.416]	0.699 (0.546) [0.283]	**0.604 (0.469)** [0.211]	1.971 (2.174) [2.103]	1.656 (1.810) [1.730]	0.827 (0.785) [0.612]	0.953 (1.629) [1.586]	0.824 (1.361) [1.315]	0.468 (0.588) [0.498]	0.350 (0.292) [0.118]	0.307 (0.254) [0.089]	0.235 (0.201) [0.046]
joint_a	0.853 (0.672) [0.487]	0.818 (0.626) [0.426]	0.923 (0.704) [0.498]	1.930 (2.125) [2.059]	1.607 (1.751) [1.674]	0.762 (0.708) [0.512]	0.874 (1.550) [1.511]	0.773 (1.278) [1.233]	0.445 (0.513) [0.409]	0.343 **(0.271)** [0.110]	0.294 (0.239) [0.072]	0.243 (0.203) [0.046]
joint_a_r	1.153 (0.890) [0.489]	1.023 (0.781) [0.410]	0.957 (0.712) [0.493]	1.999 (2.099) [2.020]	1.633 (1.741) [1.660]	0.726 (0.701) [0.524]	1.153 (1.694) [1.641]	0.964 (1.395) [1.341]	0.488 (0.552) [0.445]	0.355 (0.293) [0.109]	0.321 (0.262) [0.077]	0.246 (0.208) [0.051]
sep	**0.772 (0.591) [0.269]**	**0.690 (0.507) [0.154]**	0.693 (0.529) [0.258]	0.611 (0.454) [0.176]	0.518 (0.415) [0.170]	0.376 (0.313) [0.125]	**0.408** (0.328) [0.140]	**0.336 (0.269)** [0.093]	**0.289 (0.232)** [0.066]	0.341 (0.287) [0.130]	0.285 **(0.228)** [0.066]	**0.226 (0.187)** [0.038]
sep_a	0.795 (0.644) [0.310]	**0.690** (0.573) [0.237]	0.675 (0.582) [0.283]	**0.516 (0.389)** [0.185]	**0.440 (0.360)** [0.169]	**0.340 (0.286)** [0.103]	0.412 **(0.317)** [0.139]	0.342 (0.272) [0.096]	0.294 (0.240) [0.076]	**0.338** (0.291) [0.139]	**0.275** (0.230) [0.078]	0.229 (0.188) [0.041]
sep_a_r	1.013 (0.773) [0.330]	0.827 (0.670) [0.271]	0.614 (0.538) [0.231]	0.580 (0.447) [0.175]	0.496 (0.386) [0.149]	0.379 (0.318) [0.107]	0.488 (0.370) [0.141]	0.395 (0.311) [0.100]	0.303 (0.249) [0.074]	0.373 (0.304) [0.131]	0.307 (0.248) [0.070]	0.238 (0.199) [0.044]

Table B.6: Results for DGF2 (interaction model) with NGF1 (homoskedastic noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	1.167 (0.627) [0.215]	0.981 (0.543) [0.183]	0.426 (0.297) [0.086]	0.961 (0.695) [0.099]	0.806 (0.588) [0.086]	0.335 (0.270) [0.049]	1.094 (0.698) [0.083]	0.918 (0.588) [0.071]	0.378 (0.258) [0.038]	1.107 (0.660) [0.067]	0.929 (0.556) [0.058]	0.382 (0.244) [0.033]
ind	0.836 (0.792) [0.373]	0.741 (0.679) [0.284]	0.470 (0.423) [0.191]	0.319 (0.279) [0.090]	0.263 (0.233) [0.054]	0.148 (0.160) [0.052]	0.292 (0.246) [0.042]	0.256 (0.218) [0.041]	0.134 (0.142) [0.059]	0.264 (0.214) [0.060]	0.215 (0.188) [0.059]	0.094 (0.109) [0.049]
joint	0.994 (0.869) [0.443]	0.861 (0.738) [0.344]	0.558 (0.454) [0.169]	0.217 (0.226) [0.116]	0.179 (0.185) [0.085]	0.128 (0.127) [0.037]	0.210 (0.195) [0.072]	0.178 (0.172) [0.070]	0.136 (0.142) [0.061]	0.131 (0.114) [0.036]	0.108 (0.097) [0.029]	0.085 (0.079) [0.028]
joint_a	0.860 (0.690) [0.323]	0.746 (0.619) [0.292]	0.519 (0.457) [0.208]	0.229 (0.227) [0.114]	0.193 (0.195) [0.087]	0.140 (0.143) [0.046]	0.173 (0.165) [0.074]	0.151 (0.150) [0.070]	0.124 (0.128) [0.056]	0.128 (0.116) [0.046]	0.109 (0.101) [0.040]	0.091 (0.090) [0.038]
joint_a_r	0.627 (0.599) [0.372]	0.536 (0.529) [0.329]	0.346 (0.349) [0.192]	0.200 (0.194) [0.081]	0.176 (0.173) [0.070]	0.138 (0.148) [0.052]	0.165 (0.154) [0.071]	0.144 (0.141) [0.066]	0.116 (0.118) [0.051]	0.129 (0.130) [0.068]	0.111 (0.118) [0.063]	0.095 (0.106) [0.052]
sep	0.815 (0.784) [0.388]	0.669 (0.632) [0.288]	0.387 (0.386) [0.200]	0.231 (0.213) [0.081]	0.179 (0.168) [0.055]	0.126 (0.121) **[0.029]**	0.183 (0.163) [0.054]	0.143 (0.130) [0.040]	0.098 (0.090) [0.023]	0.136 (0.117) [0.033]	0.111 (0.099) [0.027]	0.077 (0.074) [0.018]
sep_a	0.692 (0.597) [0.260]	0.583 (0.518) [0.226]	0.361 (0.359) [0.185]	0.223 (0.201) [0.072]	0.182 (0.168) [0.058]	0.129 (0.123) [0.033]	0.147 (0.124) [0.037]	**0.127** (0.109) [0.029]	0.096 **(0.084)** **[0.018]**	0.132 (0.112) [0.029]	0.108 (0.093) **[0.023]**	**0.075 (0.067)** **[0.017]**
sep_a_r	**0.416 (0.355)** **[0.128]**	**0.341 (0.297)** **[0.100]**	**0.228 (0.203)** **[0.056]**	**0.189 (0.172)** **[0.054]**	**0.164 (0.150)** **[0.046]**	**0.125 (0.120)** [0.031]	**0.144 (0.120)** **[0.032]**	0.128 **(0.108)** **[0.028]**	**0.095 (0.084)** [0.019]	**0.126 (0.105)** **[0.028]**	**0.103 (0.088)** **[0.023]**	0.078 (0.069) [0.019]

Table B.7: Results for DGF2 (interaction model) with NGF2 (heteroskedastic noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	0.999 (0.910) [0.206]	0.842 (0.769) [0.173]	0.368 (0.347) [0.076]	1.085 (0.704) [0.129]	0.913 (0.596) [0.108]	0.378 (0.271) [0.052]	1.044 (0.777) [0.093]	0.878 (0.654) [0.077]	0.363 (0.281) [0.037]	1.051 (0.713) [0.067]	0.884 (0.600) [0.054]	0.365 (0.257) [0.026]
ind	0.379 (0.339) [0.115]	0.312 (0.283) [0.074]	0.190 (0.189) [0.050]	0.302 (0.255) [0.065]	0.245 (0.214) [0.057]	0.122 (0.127) [0.046]	0.303 (0.258) [0.064]	0.255 (0.225) [0.050]	0.108 (0.126) [0.049]	0.304 (0.237) [0.040]	0.270 (0.212) [0.027]	0.199 (0.184) [0.034]
joint	0.314 (0.294) [0.120]	0.249 (0.239) [0.091]	0.187 (0.184) [0.059]	0.180 (0.180) [0.085]	0.152 (0.149) [0.062]	0.123 (0.125) [0.049]	0.182 (0.186) [0.085]	0.152 (0.165) [0.078]	0.121 (0.140) [0.065]	0.140 (0.154) [0.088]	0.120 (0.135) [0.075]	0.091 (0.111) [0.060]
joint_a	0.295 (0.260) [0.092]	0.244 (0.223) [0.075]	0.193 (0.187) [0.055]	0.172 (0.167) [0.075]	0.148 (0.144) [0.059]	0.121 (0.122) [0.046]	0.133 (0.136) [0.063]	0.120 (0.125) [0.057]	0.102 (0.110) [0.046]	0.131 (0.140) [0.080]	0.114 (0.124) [0.070]	0.086 (0.102) [0.055]
joint_a_r	0.275 (0.252) [0.085]	0.234 (0.220) [0.072]	0.191 (0.184) [0.051]	0.155 (0.146) [0.063]	0.136 (0.128) [0.050]	0.112 (0.110) [0.040]	0.135 (0.141) [0.070]	0.121 (0.129) [0.061]	0.104 (0.115) [0.052]	0.180 (0.187) [0.104]	0.163 (0.173) [0.093]	0.130 (0.152) [0.075]
sep	0.315 (0.300) [0.136]	0.231 (0.217) [0.081]	**0.162** (0.157) **[0.044]**	0.183 (0.167) [0.060]	0.146 (0.134) [0.044]	0.107 (0.102) [0.028]	0.153 (0.147) [0.051]	0.116 (0.114) [0.036]	0.088 (0.088) [0.027]	0.126 (0.127) [0.061]	0.098 (0.098) [0.044]	0.061 (0.060) [0.019]
sep_a	0.261 (0.234) [0.074]	0.218 (0.200) [0.063]	0.172 (0.158) [0.048]	0.174 (0.153) [0.048]	0.143 (0.127) [0.037]	0.107 **(0.097)** **[0.024]**	0.123 (0.109) [0.034]	0.106 (0.095) [0.028]	**0.084 (0.077)** [0.019]	0.109 (0.096) [0.033]	0.089 (0.081) [0.032]	0.061 (0.055) [0.017]
sep_a_r	**0.245 (0.222)** **[0.069]**	**0.207 (0.190)** **[0.061]**	0.164 **(0.156)** [0.047]	**0.152 (0.134)** **[0.039]**	**0.130 (0.117)** **[0.034]**	**0.104 (0.097)** **[0.024]**	**0.121 (0.105)** **[0.030]**	**0.105 (0.091)** **[0.024]**	0.085 **(0.077)** **[0.017]**	**0.096 (0.077)** **[0.019]**	**0.081 (0.067)** **[0.018]**	**0.058 (0.050)** **[0.010]**

Table B.8: Results for DGF2 (interaction model) with NGF3 (skewed noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	3.869 (2.460) [0.344]	3.451 (2.266) [0.281]	2.469 (1.754) **[0.104]**	3.872 (2.478) [0.229]	3.462 (2.298) [0.195]	2.525 (1.797) [0.106]	4.511 (2.715) [0.281]	3.944 (2.534) [0.244]	2.681 (1.850) [0.123]	3.820 (2.446) [0.257]	3.413 (2.265) [0.232]	2.489 (1.745) [0.131]
ind	1.388 (1.146) [0.515]	1.129 (0.915) [0.388]	0.839 (0.636) [0.175]	1.012 (0.796) [0.298]	0.878 (0.667) [0.202]	0.691 (0.500) [0.065]	0.865 (0.746) [0.351]	0.726 (0.546) [0.147]	0.613 (0.442) [0.059]	1.095 (1.232) [0.801]	0.748 (0.697) [0.393]	0.631 (0.423) **[0.022]**
joint	1.503 (1.587) [1.135]	1.230 (1.361) [0.952]	0.872 (0.819) [0.439]	1.034 (0.867) [0.408]	0.882 (0.726) [0.308]	0.684 (0.498) **[0.064]**	1.210 (0.979) [0.418]	0.820 (0.650) [0.248]	0.609 (0.443) [0.072]	1.818 (2.103) [1.549]	1.574 (1.878) [1.343]	1.241 (1.369) [0.868]
joint_a	**0.991** (0.796) [0.321]	**0.879 (0.674) [0.201]**	0.800 (0.604) [0.134]	0.820 (0.622) [0.155]	0.759 (0.575) [0.122]	0.687 (0.507) [0.069]	0.663 (0.489) [0.116]	0.659 (0.484) [0.109]	0.674 (0.494) [0.127]	1.762 (2.240) [1.745]	1.585 (1.990) [1.464]	1.210 (1.355) [0.869]
joint_a_r	1.210 (0.948) [0.365]	1.033 (0.789) [0.234]	0.814 (0.612) [0.142]	0.904 (0.695) [0.209]	0.826 (0.625) [0.146]	0.689 (0.502) [0.070]	0.775 (0.589) [0.166]	0.708 (0.525) [0.117]	0.670 (0.495) [0.132]	1.498 (2.072) [1.703]	1.305 (1.790) [1.389]	0.959 (1.078) [0.701]
sep	0.993 **(0.769) [0.258]**	0.892 (0.691) [0.205]	**0.787 (0.591)** [0.132]	**0.762 (0.568) [0.140]**	0.717 (0.533) **[0.101]**	0.648 (0.477) [0.086]	0.738 (0.566) [0.179]	0.687 (0.517) [0.143]	0.614 (0.444) [0.063]	0.794 (0.970) [0.656]	0.761 (0.942) [0.643]	0.705 (0.837) [0.553]
sep_a	1.038 (0.810) [0.325]	0.891 (0.701) [0.226]	0.833 (0.643) [0.216]	0.767 (0.572) [0.143]	**0.709 (0.532)** [0.112]	**0.643 (0.474)** [0.092]	**0.630 (0.463) [0.071]**	**0.612 (0.448) [0.066]**	**0.598 (0.435) [0.049]**	0.721 (0.887) [0.604]	0.698 (0.860) [0.585]	0.656 (0.766) [0.502]
sep_a_r	1.225 (0.960) [0.354]	1.034 (0.818) [0.259]	0.892 (0.698) [0.267]	0.816 (0.618) [0.152]	0.742 (0.566) [0.125]	0.648 (0.477) [0.089]	0.704 (0.529) [0.111]	0.662 (0.493) [0.095]	0.601 (0.436) **[0.049]**	**0.581 (0.420) [0.088]**	**0.556 (0.399) [0.090]**	**0.527 (0.371)** [0.086]

Table B.9: Results for DGF4 (mixture of sin functions) with NGF1 (homoskedastic noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	2.639 (1.830) [0.163]	2.579 (1.795) [0.135]	2.473 (1.700) [0.056]	2.682 (1.822) [0.061]	2.612 (1.806) [0.049]	2.472 (1.770) **[0.018]**	2.571 (1.807) [0.186]	2.507 (1.777) [0.156]	2.370 (1.721) [0.063]	2.331 (1.704) [0.266]	2.257 (1.669) [0.227]	2.107 (1.590) [0.098]
ind	1.046 (0.756) [0.235]	0.863 (0.629) [0.168]	0.551 (0.400) [0.090]	0.956 (0.681) [0.189]	0.724 (0.541) [0.169]	**0.453** (0.310) [0.043]	1.034 (0.712) [0.070]	0.882 (0.612) [0.059]	0.538 (0.397) [0.110]	0.904 (0.659) [0.199]	**0.584 (0.443)** [0.146]	**0.433 (0.280)** [0.015]
joint	1.248 (1.283) [0.921]	1.017 (1.018) [0.704]	0.533 (0.387) [0.084]	0.819 (0.568) [0.069]	0.698 (0.482) **[0.047]**	0.470 (0.316) [0.031]	1.214 (1.161) [0.820]	0.934 (0.865) [0.578]	0.494 (0.363) [0.101]	**0.801** (0.548) [0.065]	0.650 (0.460) [0.058]	0.446 (0.295) [0.014]
joint_a	0.919 (0.680) [0.259]	0.772 (0.559) [0.143]	0.531 (0.383) [0.075]	0.830 (0.583) [0.096]	0.703 (0.488) [0.065]	0.473 (0.321) [0.045]	0.911 (0.645) [0.185]	0.753 (0.535) [0.132]	0.502 (0.369) [0.101]	0.817 (0.553) [0.046]	0.674 (0.472) **[0.040]**	0.450 (0.300) **[0.013]**
joint_a_r	0.933 (0.680) [0.188]	0.783 (0.570) [0.126]	0.538 (0.388) [0.074]	0.847 (0.592) [0.074]	0.718 (0.499) [0.059]	0.470 (0.318) [0.034]	0.909 (0.640) [0.132]	0.766 (0.546) [0.126]	0.517 (0.379) [0.106]	0.825 (0.573) [0.113]	0.679 (0.486) [0.093]	0.464 (0.317) [0.053]
sep	0.869 (0.622) [0.149]	0.731 (0.521) [0.095]	0.493 **(0.345) [0.034]**	0.840 (0.585) [0.094]	0.704 (0.485) [0.049]	0.464 **(0.309)** [0.022]	0.827 (0.567) [0.072]	0.683 (0.466) [0.038]	0.432 (0.289) **[0.016]**	0.846 (0.571) [0.065]	0.690 (0.481) [0.047]	0.448 (0.299) [0.035]
sep_a	**0.800 (0.564) [0.095]**	**0.701 (0.497) [0.072]**	0.496 (0.355) [0.050]	**0.796 (0.552)** [0.060]	**0.685 (0.474)** [0.048]	0.460 **(0.309)** [0.030]	**0.789 (0.539) [0.046]**	**0.666 (0.454) [0.035]**	**0.429 (0.288) [0.016]**	0.809 **(0.543) [0.045]**	0.673 (0.468) [0.042]	0.440 (0.298) [0.039]
sep_a_r	0.816 (0.582) [0.103]	0.712 (0.510) [0.076]	**0.492** (0.350) [0.044]	0.808 (0.561) **[0.056]**	0.694 (0.481) **[0.047]**	0.464 **(0.309)** [0.024]	0.792 (0.544) [0.053]	0.667 (0.457) [0.038]	**0.429** (0.290) [0.019]	0.814 (0.547) [0.046]	0.678 (0.471) [0.043]	0.442 (0.300) [0.042]

Table B.10: Results for DGF4 (mixture of sin functions) with NGF2 (heteroskedastic noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	2.693 (1.832) [0.218]	2.575 (1.773) [0.180]	2.322 (1.647) [0.064]	2.764 (1.873) [0.250]	2.637 (1.817) [0.217]	2.380 (1.678) [0.104]	2.729 (1.832) [0.209]	2.596 (1.790) [0.187]	2.311 (1.698) [0.091]	2.710 (1.831) [0.275]	2.572 (1.786) [0.240]	2.286 (1.673) [0.103]
ind	1.047 (0.760) [0.234]	0.859 (0.624) [0.161]	0.542 (0.395) [0.101]	0.965 (0.699) [0.244]	0.751 (0.566) [0.204]	0.429 (0.292) [0.024]	1.227 (1.269) [0.892]	0.842 (0.588) [0.113]	0.448 **(0.299)** [0.042]	0.922 (0.686) [0.217]	0.584 (0.446) [0.141]	**0.409 (0.270)** [0.019]
joint	1.018 (0.949) [0.633]	0.790 (0.662) [0.359]	0.528 (0.385) [0.101]	0.773 (0.643) [0.328]	0.652 (0.534) [0.252]	0.451 (0.320) [0.066]	1.033 (1.063) [0.669]	0.920 (1.022) [0.664]	0.699 (0.898) [0.604]	0.682 (0.479) [0.084]	0.565 (0.398) [0.054]	0.414 (0.272) [0.012]
joint_a	0.948 (0.737) [0.355]	0.748 (0.564) [0.184]	0.546 (0.399) [0.105]	0.727 (0.523) [0.067]	0.618 (0.448) [0.046]	0.434 (0.298) **[0.023]**	0.968 (0.956) [0.545]	0.850 (0.925) [0.560]	0.686 (0.886) [0.592]	0.709 (0.488) [0.053]	0.589 (0.411) [0.043]	0.416 (0.277) [0.014]
joint_a_r	0.908 (0.690) [0.256]	0.748 (0.570) [0.187]	0.549 (0.399) [0.106]	0.740 (0.534) [0.089]	0.616 (0.449) [0.065]	0.437 (0.297) **[0.023]**	0.984 (0.861) [0.420]	0.876 (0.825) [0.441]	0.663 (0.740) [0.459]	**0.673 (0.466)** [0.067]	**0.556 (0.385)** [0.039]	0.414 (0.272) [0.012]
sep	0.759 (0.542) [0.114]	0.635 (0.458) [0.081]	**0.467 (0.328)** [0.049]	0.743 (0.529) [0.069]	0.622 (0.449) [0.051]	**0.427** (0.293) [0.038]	0.738 (0.518) [0.066]	0.623 (0.444) [0.042]	0.444 (0.300) [0.020]	0.747 (0.512) [0.053]	0.615 (0.429) [0.046]	0.423 (0.283) **[0.009]**
sep_a	**0.708 (0.509) [0.086]**	**0.626 (0.449) [0.069]**	0.471 **(0.328) [0.041]**	**0.711** (0.512) **[0.052]**	0.611 (0.443) **[0.037]**	0.433 (0.295) [0.026]	**0.719 (0.505) [0.051]**	0.616 (0.439) **[0.038]**	0.444 (0.300) **[0.019]**	0.723 (0.493) **[0.039]**	0.608 (0.420) **[0.029]**	0.420 (0.280) **[0.009]**
sep_a_r	0.728 (0.525) [0.098]	0.630 (0.457) [0.078]	0.471 (0.329) [0.045]	0.715 **(0.511) [0.052]**	**0.598 (0.432)** [0.044]	0.428 **(0.291)** [0.035]	0.721 (0.509) [0.054]	**0.613 (0.438)** [0.041]	**0.443** (0.300) **[0.019]**	0.697 (0.477) [0.053]	0.575 (0.398) [0.040]	0.412 (0.275) [0.026]

Table B.11: Results for DGF4 (mixture of sin functions) with NGF3 (skewed noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
nw	2.312 (0.993) [0.243]	1.947 (0.976) [0.220]	1.037 (0.725) **[0.107]**	1.941 (0.908) [0.192]	1.644 (0.881) [0.177]	0.923 (0.654) [0.101]	1.983 (0.908) [0.226]	1.678 (0.883) [0.214]	0.927 (0.659) [0.110]	1.894 (0.945) [0.212]	1.616 (0.906) [0.197]	0.944 (0.680) [0.109]
ind	0.811 (0.598) **[0.141]**	0.770 (0.569) **[0.121]**	0.699 (0.547) [0.187]	0.558 (0.434) [0.121]	0.550 (0.426) [0.114]	0.438 (0.338) [0.065]	0.798 (0.572) **[0.067]**	0.771 (0.549) **[0.038]**	0.723 (0.510) [0.054]	0.472 (0.364) [0.090]	0.409 (0.308) [0.057]	0.340 (0.253) [0.029]
joint	1.782 (1.342) [1.098]	1.466 (1.133) [0.849]	0.789 (0.640) [0.271]	0.594 (0.471) [0.146]	0.539 (0.425) [0.124]	0.408 (0.316) **[0.064]**	1.291 (1.222) [1.037]	1.146 (1.027) [0.814]	0.807 (0.599) [0.214]	0.388 (0.302) [0.079]	0.356 (0.268) [0.049]	0.319 **(0.235) [0.022]**
joint_a	1.528 (1.295) [1.083]	1.271 (1.078) [0.817]	0.751 (0.602) [0.236]	0.537 (0.423) [0.119]	0.501 (0.391) [0.100]	0.419 (0.325) [0.068]	1.112 (1.124) [0.953]	0.998 (0.949) [0.752]	0.716 (0.560) [0.218]	**0.354 (0.264) [0.038]**	0.344 (0.255) **[0.035]**	0.321 (0.237) [0.025]
joint_a_r	1.818 (1.243) [0.891]	1.510 (1.056) [0.663]	0.822 (0.627) [0.195]	0.716 (0.555) [0.136]	0.625 (0.486) [0.123]	0.468 (0.365) [0.080]	1.126 (1.273) [1.153]	0.930 (1.067) [0.940]	0.514 (0.513) [0.304]	0.457 (0.347) [0.063]	0.411 (0.310) [0.054]	0.345 (0.258) [0.032]
sep	0.740 **(0.573)** [0.212]	0.677 (0.531) [0.208]	**0.590 (0.468)** [0.166]	**0.514 (0.397) [0.099]**	**0.467 (0.362) [0.080]**	**0.405 (0.313)** [0.065]	0.706 (0.520) [0.108]	0.599 (0.456) [0.125]	0.419 (0.346) [0.128]	0.363 (0.277) [0.051]	**0.336 (0.253)** [0.037]	**0.318** (0.237) [0.023]
sep_a	**0.731** (0.580) [0.275]	**0.650 (0.510)** [0.209]	0.642 (0.490) [0.194]	0.525 (0.409) [0.110]	0.479 (0.372) [0.089]	0.410 (0.317) [0.067]	**0.371** (0.307) [0.110]	**0.363** (0.297) [0.102]	0.343 (0.280) [0.091]	0.366 (0.276) [0.051]	0.340 (0.255) **[0.035]**	0.320 (0.237) **[0.022]**
sep_a_r	0.980 (0.753) [0.276]	0.896 (0.685) [0.213]	0.698 (0.532) [0.185]	0.652 (0.502) [0.124]	0.589 (0.454) [0.114]	0.461 (0.356) [0.075]	0.382 **(0.296)** [0.077]	0.372 **(0.285)** [0.059]	**0.337 (0.258) [0.035]**	0.441 (0.335) [0.060]	0.397 (0.300) [0.048]	0.349 (0.263) [0.037]

Table B.12: Results for DGF5 (neural network model) with NGF1 (homoskedastic noise)

	200 (0.025, 0.975)	200 (0.05, 0.95)	200 (0.25, 0.75)	600 (0.025, 0.975)	600 (0.05, 0.95)	600 (0.25, 0.75)	1000 (0.025, 0.975)	1000 (0.05, 0.95)	1000 (0.25, 0.75)	2000 (0.025, 0.975)	2000 (0.05, 0.95)	2000 (0.25, 0.75)
nw	0.771 (0.574) [0.128]	0.683 (0.519) [0.126]	0.467 (0.381) [0.133]	0.957 (0.726) [0.227]	0.856 (0.663) [0.220]	0.611 (0.513) [0.217]	0.858 (0.658) [0.221]	0.761 (0.602) [0.216]	0.523 (0.465) [0.222]	0.914 (0.738) [0.218]	0.818 (0.679) [0.214]	0.591 (0.530) [0.230]
ind	0.815 (0.579) [0.099]	0.710 (0.512) [0.093]	**0.382** (0.295) [0.041]	0.824 (0.597) [0.104]	0.659 (0.475) [0.079]	**0.331 (0.248)** [0.027]	0.798 (0.566) [0.067]	0.638 (0.450) [0.042]	0.330 (0.245) [0.010]	0.733 (0.518) [0.049]	0.584 (0.413) [0.051]	**0.285 (0.223)** [0.036]
joint	0.802 (0.574) [0.090]	0.704 (0.510) [0.092]	0.386 (0.300) [0.042]	0.759 (0.542) [0.111]	0.631 (0.448) [0.083]	0.334 **(0.248)** **[0.025]**	0.763 (0.541) [0.076]	0.613 (0.428) [0.037]	0.328 (0.242) **[0.008]**	0.712 (0.508) [0.089]	0.584 (0.414) [0.066]	0.302 (0.231) [0.018]
joint_a	0.797 (0.573) [0.092]	0.702 (0.514) [0.089]	0.388 (0.301) [0.040]	0.729 (0.503) [0.059]	0.611 (0.421) [0.049]	0.335 **(0.248)** [0.026]	0.728 (0.509) [0.044]	0.601 (0.418) [0.026]	0.329 (0.244) **[0.008]**	0.685 (0.472) [0.034]	0.565 (0.387) [0.025]	0.300 (0.229) [0.017]
joint_a_r	0.820 (0.595) [0.094]	0.716 (0.524) [0.091]	0.400 (0.313) [0.051]	0.771 (0.541) [0.064]	0.644 (0.453) [0.056]	0.341 (0.256) [0.029]	0.767 (0.542) [0.053]	0.622 (0.437) [0.032]	0.330 (0.246) [0.009]	0.708 (0.496) [0.040]	0.580 (0.404) [0.032]	0.303 (0.233) [0.018]
sep	0.775 (0.549) [0.085]	0.679 **(0.486)** [0.080]	**0.382 (0.294)** **[0.034]**	0.698 (0.472) **[0.042]**	**0.589 (0.400)** **[0.034]**	0.337 (0.252) [0.029]	0.706 (0.488) **[0.038]**	0.584 (0.403) [0.026]	**0.317 (0.234)** [0.011]	0.662 **(0.450)** [0.030]	**0.549 (0.373)** [0.020]	0.292 **(0.223)** **[0.016]**
sep_a	**0.769 (0.547)** **[0.073]**	**0.671** (0.487) [0.069]	0.403 (0.313) [0.046]	**0.697 (0.471)** **[0.042]**	**0.589** (0.401) **[0.034]**	0.337 (0.253) [0.030]	**0.701 (0.483)** **[0.038]**	**0.583 (0.402)** **[0.025]**	0.319 (0.235) [0.013]	**0.661** (0.451) **[0.028]**	**0.549** (0.375) **[0.019]**	0.293 (0.224) **[0.016]**
sep_a_r	0.788 (0.570) [0.075]	0.684 (0.501) **[0.066]**	0.408 (0.321) [0.048]	0.719 (0.491) [0.044]	0.616 (0.425) [0.036]	0.346 (0.260) [0.029]	0.712 (0.491) **[0.038]**	0.599 (0.415) [0.026]	0.321 (0.239) [0.013]	0.669 (0.459) [0.032]	0.559 (0.386) [0.026]	0.296 (0.227) [0.018]

Table B.13: Results for DGF5 (neural network model) with NGF2 (heteroskedastic noise)

	200 (0.025, 0.975)	200 (0.05, 0.95)	200 (0.25, 0.75)	600 (0.025, 0.975)	600 (0.05, 0.95)	600 (0.25, 0.75)	1000 (0.025, 0.975)	1000 (0.05, 0.95)	1000 (0.25, 0.75)	2000 (0.025, 0.975)	2000 (0.05, 0.95)	2000 (0.25, 0.75)
nw	0.680 (0.584) [0.280]	0.605 (0.529) [0.249]	0.438 (0.382) [0.175]	**0.584 (0.431)** **[0.142]**	**0.515 (0.388)** **[0.131]**	0.353 (0.273) [0.074]	**0.535 (0.401)** [0.137]	**0.472 (0.359)** [0.121]	0.322 (0.250) [0.063]	**0.419 (0.336)** [0.112]	**0.374** (0.302) [0.095]	0.272 (0.220) [0.049]
ind	0.767 (0.623) [0.238]	0.630 (0.518) [0.203]	0.366 (0.287) [0.072]	0.767 (0.601) [0.225]	0.592 (0.467) [0.172]	0.348 (0.273) [0.075]	0.713 (0.564) [0.206]	0.580 (0.460) [0.175]	0.348 (0.273) [0.065]	0.652 (0.519) [0.178]	0.545 (0.434) [0.157]	0.291 (0.236) [0.066]
joint	0.699 (0.556) [0.209]	0.582 (0.465) [0.171]	**0.365** (0.285) [0.066]	0.803 (0.610) [0.212]	0.588 (0.456) [0.163]	0.351 (0.272) **[0.070]**	0.677 (0.538) [0.209]	0.563 (0.444) [0.171]	0.339 (0.264) [0.065]	0.585 (0.467) [0.166]	0.473 (0.384) [0.146]	0.293 (0.237) [0.069]
joint_a	0.635 (0.484) [0.157]	0.547 (0.421) [0.134]	**0.365 (0.284)** **[0.061]**	0.642 (0.497) [0.191]	0.549 (0.428) [0.164]	0.358 (0.277) [0.074]	0.655 (0.511) [0.200]	0.556 (0.434) [0.170]	0.340 (0.265) [0.065]	0.577 (0.451) [0.150]	0.469 (0.373) [0.133]	0.289 (0.233) [0.066]
joint_a_r	0.672 (0.525) [0.169]	0.596 (0.473) [0.155]	0.388 (0.306) [0.072]	0.692 (0.538) [0.190]	0.576 (0.456) [0.169]	0.358 (0.280) [0.077]	0.678 (0.526) [0.201]	0.567 (0.446) [0.173]	0.344 (0.269) [0.066]	0.575 (0.446) [0.142]	0.462 (0.366) [0.127]	0.284 (0.229) [0.062]
sep	**0.586 (0.441)** **[0.135]**	**0.521 (0.399)** **[0.120]**	0.378 (0.297) [0.066]	0.665 (0.520) [0.203]	0.543 (0.427) [0.165]	**0.337 (0.263)** **[0.070]**	0.617 (0.469) [0.156]	0.509 (0.386) [0.127]	**0.310** (0.241) [0.056]	0.528 (0.411) [0.130]	0.400 (0.306) [0.084]	**0.248 (0.193)** [0.028]
sep_a	0.614 (0.470) [0.137]	0.542 (0.421) [0.126]	0.388 (0.309) [0.078]	0.622 (0.478) [0.180]	0.526 (0.406) [0.152]	0.340 (0.264) [0.072]	0.607 (0.448) **[0.135]**	0.507 (0.377) **[0.115]**	**0.310 (0.239)** **[0.051]**	0.514 (0.370) [0.071]	0.402 (0.298) [0.061]	0.253 (0.196) [0.024]
sep_a_r	0.639 (0.492) [0.143]	0.568 (0.445) [0.134]	0.392 (0.311) [0.078]	0.643 (0.494) [0.170]	0.551 (0.432) [0.157]	0.340 (0.267) [0.074]	0.620 (0.462) [0.147]	0.510 (0.387) [0.128]	**0.310** (0.242) [0.056]	0.519 (0.373) **[0.068]**	0.397 **(0.292)** **[0.053]**	0.252 (0.195) **[0.022]**

Table B.14: Results for DGF5 (neural network model) with NGF3 (skewed noise)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
ind	0.102 (0.106) [0.049]	0.089 (0.098) [0.050]	0.067 (0.060) [0.023]	0.087 (0.119) [0.035]	0.070 (0.086) [0.024]	**0.062 (0.069)** [0.017]	0.069 (0.109) [0.067]	0.049 (0.069) [0.016]	0.046 (0.062) **[0.015]**	0.058 (0.085) [0.021]	0.045 (0.059) [0.011]	0.037 (0.040) [0.012]
joint	0.077 (0.071) [0.033]	**0.068 (0.064)** [0.029]	**0.066** (0.061) [0.028]	**0.065** (0.078) [0.021]	**0.060 (0.070)** [0.017]	0.063 (0.073) [0.019]	0.077 (0.117) [0.076]	0.052 (0.070) [0.029]	0.051 (0.062) [0.020]	**0.038 (0.042)** [0.010]	**0.037** (0.039) [0.011]	0.040 (0.043) [0.018]
joint_a	**0.074 (0.062) [0.025]**	0.073 (0.065) [0.030]	0.074 (0.067) [0.033]	0.068 (0.076) [0.021]	0.066 (0.074) [0.019]	0.069 (0.078) [0.024]	0.043 (0.050) [0.019]	0.044 (0.050) [0.019]	0.052 (0.056) [0.022]	0.040 (0.043) [0.016]	0.039 (0.041) [0.016]	0.039 (0.041) [0.017]
joint_a_r	0.077 (0.066) [0.030]	0.078 (0.070) [0.035]	0.078 (0.073) [0.035]	0.068 **(0.075)** [0.024]	0.066 (0.074) [0.023]	0.069 (0.086) [0.037]	0.045 (0.053) [0.022]	0.048 (0.052) [0.024]	0.055 (0.061) [0.025]	0.042 (0.043) [0.015]	0.040 (0.041) [0.017]	0.037 (0.040) [0.020]
sep	0.095 (0.099) [0.049]	0.082 (0.076) [0.030]	**0.066 (0.059) [0.022]**	0.088 (0.104) [0.027]	0.075 (0.087) [0.020]	0.068 (0.075) [0.016]	0.049 (0.064) [0.030]	0.044 (0.052) [0.018]	**0.041 (0.044) [0.015]**	0.043 (0.049) [0.012]	0.040 (0.045) [0.011]	0.036 (0.039) **[0.009]**
sep_a	0.082 (0.070) [0.027]	0.076 (0.069) [0.027]	0.077 (0.072) [0.036]	0.074 (0.080) **[0.016]**	0.068 (0.074) **[0.014]**	0.066 (0.071) **[0.015]**	0.040 (0.046) [0.015]	**0.041 (0.046) [0.015]**	0.042 (0.045) **[0.015]**	0.041 **(0.042) [0.009]**	**0.037 (0.038) [0.009]**	**0.034 (0.037)** [0.010]
sep_a_r	0.077 (0.068) [0.026]	0.072 (0.065) **[0.025]**	0.076 (0.071) [0.035]	0.073 (0.077) [0.021]	0.068 (0.071) [0.018]	0.068 (0.074) [0.023]	**0.037 (0.042) [0.012]**	0.042 **(0.046)** [0.017]	0.042 (0.046) **[0.015]**	0.045 (0.046) [0.011]	0.040 (0.039) [0.010]	0.037 (0.038) [0.013]

Table B.15: Recurrent Neural Network results for TGF1 (AR1 model)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
ind	0.187 (0.145) [0.039]	0.176 (0.137) [0.031]	0.149 (0.116) [0.017]	0.237 (0.173) [0.053]	0.189 (0.141) [0.030]	0.132 (0.100) [0.011]	0.258 (0.185) [0.024]	0.239 (0.177) [0.021]	0.187 (0.139) [0.014]	0.268 (0.191) [0.017]	0.249 (0.182) [0.012]	0.190 (0.148) [0.018]
joint	0.167 (0.129) [0.032]	**0.155 (0.120)** [0.022]	**0.145 (0.113)** [0.016]	0.157 (0.116) [0.023]	0.145 (0.108) **[0.015]**	0.129 (0.098) [0.010]	0.262 (0.191) [0.037]	0.232 (0.173) [0.030]	0.184 (0.135) [0.013]	0.277 (0.200) [0.034]	0.246 (0.183) [0.030]	0.192 (0.149) [0.016]
joint_a	**0.161 (0.124)** [0.023]	0.158 (0.122) [0.025]	0.152 (0.119) [0.022]	0.156 (0.114) [0.025]	0.144 (0.107) [0.019]	0.131 (0.099) [0.013]	0.216 (0.159) [0.029]	0.199 (0.148) [0.027]	0.172 (0.125) [0.018]	0.232 (0.169) [0.038]	0.213 (0.160) [0.035]	0.176 (0.138) [0.025]
joint_a_r	0.167 (0.129) [0.028]	0.159 (0.124) [0.024]	0.150 (0.117) [0.018]	0.162 (0.119) [0.028]	0.149 (0.110) [0.021]	0.130 (0.099) [0.012]	0.226 (0.166) [0.030]	0.208 (0.155) [0.026]	0.176 (0.128) [0.018]	0.245 (0.177) [0.036]	0.220 (0.165) [0.032]	0.181 (0.141) [0.023]
sep	0.168 (0.131) [0.022]	0.163 (0.127) [0.020]	0.155 (0.122) **[0.014]**	0.173 (0.133) [0.036]	0.151 (0.115) [0.025]	**0.125 (0.094) [0.007]**	0.187 (0.147) [0.044]	0.158 (0.122) [0.032]	0.136 (0.098) [0.022]	0.173 (0.133) [0.032]	0.147 (0.113) [0.025]	0.118 (0.089) [0.005]
sep_a	0.171 (0.136) [0.024]	0.167 (0.133) [0.021]	0.163 (0.129) [0.018]	**0.150 (0.111)** [0.021]	**0.143 (0.106)** [0.017]	**0.125 (0.094) [0.007]**	**0.145 (0.106) [0.019]**	**0.139 (0.100) [0.015]**	**0.129 (0.088) [0.010]**	**0.134 (0.098) [0.008]**	**0.129 (0.095) [0.007]**	**0.116 (0.086) [0.004]**
sep_a_r	0.168 (0.134) **[0.020]**	0.164 (0.131) **[0.018]**	0.161 (0.128) [0.022]	0.151 (0.112) **[0.019]**	0.144 (0.108) [0.016]	0.126 (0.096) [0.008]	0.150 (0.111) [0.025]	0.144 (0.106) [0.023]	0.134 (0.094) [0.018]	0.138 (0.103) [0.019]	0.131 (0.098) [0.016]	0.118 (0.087) [0.007]

Table B.16: Recurrent Neural Network results for TGF2 (ARMA model)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
ind	0.398 (0.293) [0.128]	0.339 (0.255) [0.100]	0.190 (0.204) [0.110]	0.084 (0.098) [0.066]	0.050 (0.046) [0.023]	0.037 (0.035) [0.019]	0.064 (0.108) [0.025]	0.051 (0.072) [0.016]	0.042 (0.050) [0.014]	0.391 (0.264) [0.054]	0.321 (0.232) [0.065]	0.064 (0.077) [0.047]
joint	0.358 (0.281) [0.126]	0.306 (0.266) [0.129]	0.234 (0.228) [0.116]	0.041 (0.037) [0.022]	0.037 (0.034) [0.020]	0.049 (0.044) [0.028]	0.043 (0.052) [0.019]	0.040 (0.046) [0.013]	0.041 (0.049) [0.013]	0.233 (0.198) [0.102]	0.140 (0.172) [0.116]	0.084 (0.112) [0.075]
joint_a	0.234 (0.253) [0.160]	0.220 (0.243) [0.153]	0.192 (0.215) [0.128]	0.040 (0.035) [0.020]	0.041 (0.038) [0.023]	0.048 (0.041) [0.022]	0.041 (0.045) [0.012]	0.041 (0.046) [0.014]	0.044 (0.048) [0.016]	0.060 (0.079) [0.056]	0.050 (0.040) [0.009]	0.046 (0.037) [0.007]
joint_a_r	0.246 (0.261) [0.163]	0.224 (0.254) [0.161]	0.182 (0.218) [0.134]	0.047 (0.040) [0.022]	0.045 (0.043) [0.026]	0.059 (0.055) [0.036]	0.039 (0.041) [0.012]	0.037 (0.040) [0.013]	0.038 (0.041) [0.015]	0.042 (0.047) [0.031]	0.032 (0.041) [0.028]	0.024 (0.032) [0.022]
sep	0.316 (0.252) [0.112]	0.246 (0.230) [0.121]	0.153 (0.187) [0.116]	0.047 (0.047) [0.026]	0.043 (0.044) [0.026]	0.036 (0.034) [0.019]	0.051 (0.064) [0.020]	0.043 (0.053) [0.016]	0.036 (0.038) [0.011]	0.217 (0.179) [0.079]	0.089 (0.097) [0.055]	0.022 (0.025) [0.007]
sep_a	0.066 (0.069) [0.030]	0.063 (0.068) [0.030]	0.065 (0.073) [0.037]	0.037 (0.033) [0.017]	**0.036** (0.033) [0.019]	0.035 **(0.033) [0.016]**	0.040 (0.041) [0.013]	**0.036** (0.038) **[0.009]**	0.035 (0.035) **[0.009]**	**0.023** (0.023) [0.009]	**0.021** (0.021) [0.006]	0.020 **(0.020) [0.005]**
sep_a_r	**0.065 (0.068) [0.029]**	**0.057 (0.063) [0.025]**	**0.056 (0.066) [0.033]**	**0.036 (0.032) [0.014]**	**0.036 (0.032) [0.017]**	**0.034 (0.033)** [0.017]	**0.037 (0.039) [0.010]**	**0.036 (0.036) [0.009]**	**0.034 (0.034) [0.009]**	**0.023 (0.022) [0.008]**	**0.021 (0.020) [0.005]**	**0.018 (0.020) [0.005]**

Table B.17: Recurrent Neural Network results for TGF3 (ARCH model)

	200 (0.025, 0.975)	**200 (0.05, 0.95)**	**200 (0.25, 0.75)**	**600 (0.025, 0.975)**	**600 (0.05, 0.95)**	**600 (0.25, 0.75)**	**1000 (0.025, 0.975)**	**1000 (0.05, 0.95)**	**1000 (0.25, 0.75)**	**2000 (0.025, 0.975)**	**2000 (0.05, 0.95)**	**2000 (0.25, 0.75)**
ind	0.336 (0.264) [0.111]	0.285 (0.263) [0.145]	0.202 (0.226) [0.138]	0.451 (0.316) [0.126]	0.315 (0.266) [0.145]	0.133 (0.196) [0.139]	0.067 (0.088) [0.034]	0.053 (0.057) [0.018]	**0.047** (0.044) [0.016]	0.059 (0.100) [0.020]	0.048 (0.073) [0.013]	0.042 (0.051) [0.012]
joint	0.350 (0.309) [0.176]	0.298 (0.296) [0.179]	0.234 (0.241) [0.138]	0.360 (0.318) [0.183]	0.308 (0.314) [0.199]	0.248 (0.262) [0.157]	**0.052** (0.050) [0.021]	**0.048** (0.045) [0.017]	0.054 (0.050) [0.023]	**0.043** (0.059) [0.014]	**0.043** (0.054) [0.012]	0.048 (0.060) [0.019]
joint_a	0.250 (0.284) [0.194]	0.232 (0.266) [0.179]	0.187 (0.220) [0.139]	0.217 (0.288) [0.208]	0.200 (0.267) [0.190]	0.171 (0.230) [0.157]	0.056 (0.051) [0.021]	0.055 (0.050) [0.021]	0.058 (0.053) [0.025]	0.047 (0.060) [0.013]	0.047 (0.058) [0.012]	0.051 (0.061) [0.015]
joint_a_r	0.238 (0.284) [0.199]	0.219 (0.267) [0.184]	0.183 (0.226) [0.150]	0.211 (0.283) [0.204]	0.201 (0.278) [0.199]	0.169 (0.241) [0.168]	0.054 (0.051) [0.023]	0.051 (0.048) [0.022]	0.050 (0.052) [0.028]	**0.043 (0.054)** [0.013]	**0.043** (0.051) [0.013]	0.045 (0.054) [0.020]
sep	0.203 (0.209) [0.119]	0.180 (0.207) [0.129]	0.133 (0.177) [0.117]	0.317 (0.268) [0.141]	0.219 (0.262) [0.183]	0.088 (0.167) [0.125]	0.059 (0.066) [0.027]	0.051 (0.051) [0.017]	0.048 **(0.043) [0.012]**	0.054 (0.068) [0.017]	0.047 (0.057) [0.013]	0.042 (0.048) [0.012]
sep_a	0.088 (0.085) [0.042]	0.078 (0.085) [0.043]	0.068 (0.076) [0.039]	**0.033 (0.031) [0.013]**	**0.032 (0.030) [0.013]**	**0.027 (0.028) [0.009]**	0.053 **(0.049) [0.016]**	**0.048 (0.043) [0.013]**	0.049 (0.045) [0.016]	0.048 (0.056) [0.012]	0.044 (0.051) **[0.010]**	0.042 (0.049) **[0.011]**
sep_a_r	**0.084 (0.072) [0.031]**	**0.072 (0.069) [0.028]**	**0.060 (0.061) [0.028]**	0.037 (0.035) [0.015]	0.034 (0.033) [0.014]	0.028 **(0.028)** [0.010]	**0.052 (0.049)** [0.017]	0.050 (0.047) [0.017]	0.048 (0.046) [0.017]	0.047 (0.055) **[0.011]**	**0.043 (0.050) [0.010]**	**0.040 (0.047)** [0.012]

Table B.18: Recurrent Neural Network results for TGF4 (GARCH model)

Appendix C: Chapter 4

C.1 HyperParameters

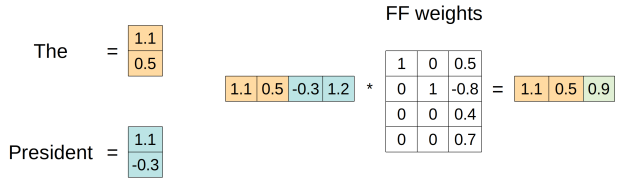
In addition to the hyperparameters recorded in the main paper, there are a large number of additional hyperparameters which we kept constant throughout experiments. The feedforward NN in the Static Layer, FF_s, has two hidden layers each of dimension 200. The NN in the Embed Update layer, FF_{EU} has two hidden layers, each of dimension 320. The output NN has one hidden layer of dimension 200. Aside from FF_{EU}, which is initialized using the identity function as described in Supplementary section C.2, all parameters of networks are initialized from the uniform distribution between -0.1 and 0.1. The article theme size, a, is set to 50. All network layers use the SELU activation function of **?**. The kernel size k for the Static Layer is set to 6, allowing each token to attend the 3 tokens either side.

On the OntoNotes Corpus, we train for 60 epochs, and half the learning rate every 12 epochs. On ACE 2005, we train for 150 epochs, and half the learning rate every 30 epochs. We train with a maximum batch dimension of 900 tokens. Articles longer than length 900 are split and processed in separate batches. We train using the Adam Optimizer, and, in addition to the dropout of 0.1, we apply a dropout to the Glove/LM embeddings of 0.2.

C.2 Identity initialization

Figure C.1 gives a minimum working example of identity initialization of FF_{EU}. The embedding for “The" is [1.1, 0.5], and that for “President" is [1.1, -0.3]. Through the unfold ops, we'll end up with the two embeddings concatenated together. Figure C.1 shows FF_{EU} as having just one layer with no activation function to demonstrate the effect of the identity initialization. The first two dimensions of the output are the embedding for “The" with no changes. The final output (in light green) is the weighting.

Figure C.1: Update mechanism



In reality, the zeros in the weights tensor are initialized to very small random numbers (we use a uniform initialization between -0.01 and 0.01), so that during training FF_{EU} learns to update the embedding for “The" using the information that it is one step before the word “President".

C.3 Formation of outputs R and D in Structure Layer

Outputs Ⓡ and Ⓓ of the **Structure Layer** have dimensions [b,s, k, e] and [b, s, k, d] respectively. These outputs are a weighted average of the directional and embedding outputs from the L levels of the structure layer. We use the weights, W', (see Figure 4.6) to form the weighted average:

$$D = \sum_{l=1}^{L} W'_l D_l$$

In the case of the weighted average for the embedding tensor, R, we use the weights from the next level.

$$R = \sum_{l=1}^{L} W'_{l+1} R_l$$

As a result, when updating, each token “sees" information from tokens/entities on other levels dependent on whether or not they are in the same entity. For the intuition behind this, we use the example phrase “The United Kingdom government" from Figure 4.6. The model should output merge values M which group the tokens “The United Kingdom" on the first level, and then group all the tokens on the second level. If this is the case, then for **the token “United"**, R and D will hold the embedding of/directions to the tokens “The" and “Kingdom" in their disaggregated (unmerged) form. However, for **the token “government"**, R and D will hold embeddings of/ directions to the combined entity “the United Kingdom" in each of the three slots for “The", “United" and “Kingdom". Because “government" is not in the same entity as “The United Kingdom" on the first level, it “sees" the aggregated embedding of this entity.

Intuitively, this allows the token “government" to update in the model based on the information that it has a country one step to the left of it, as opposed to having three separate tokens, one, two and three steps to the left respectively. Note that as with the entity merging, there are no hard decisions during training, with this effect based on the real valued merge tensor M, to allow differentiability.