DEGREE PROJECT IN MATHEMATICS,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

A Neural Networks Approach to Portfolio Choice

YOUNES DJEHICHE

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

A Neural Networks Approach to Portfolio Choice

YOUNES DJEHICHE

Degree Projects in Financial Mathematics (30 ECTS credits)
Degree Programme in Applied and Computational Mathematics
KTH Royal Institute of Technology year 2018
Supervisor at Aktie-Ansvar AB: Björn Löfdahl
Supervisor at KTH: Henrik Hult
Examiner at KTH: Henrik Hult

TRITA-SCI-GRU 2018:233
MAT-E 2018:41

Royal Institute of Technology
School of Engineering Sciences
KTH SCI
SE-100 44 Stockholm, Sweden
URL: www.kth.se/sci

Abstract

This study investigates a neural networks approach to portfolio choice. Linear regression models are extensively used for prediction. With the return as the output variable, one can come to understand its relation to the explanatory variables the linear regression is built upon. However, if the relationship between the output and input variables is non-linear, the linear regression model may not be a suitable choice. An Artificial Neural Network (ANN) is a non-linear statistical model that has been shown to be a "good" approximator of non-linear functions. In this study, two different ANN models are considered, Feed-forward Neural Networks (FNN) and Recurrent Neural Networks (RNN). Networks from these models are trained to predict monthly returns on asset data consisting of macroeconomic data and market data. The predicted returns are then used in a long-short portfolio strategy. The performance of these networks and their corresponding portfolios are then compared to a benchmark linear regression model. Metrics such as average hit-rate, mean squared prediction error, portfolio value and risk-adjusted returns are used to evaluate the model performances. The linear regression and the feed-forward model yielded good average hit-rates and mean squared-errors, but poor portfolio performances. The recurrent neural network models yielded worse average hit-rates and mean squared prediction errors, but had outstanding portfolio performances.

Några tillämpningar av neurala nätverk i portföljval

Sammanfattning

Den här studien undersöker portföljval med hjälp av neurala nätverk. Linjära regressionsmodeller används extensivt vid prediktion. Med avkastning som responsvariabel kan man ta reda på dess relation med förklaringsvariablerna som regressionmodellen är byggd på. Men, om förhållandet är icke-linjärt, kan en linjär regressionmodell vara opassande. Neurala nätverk är en icke-linjär statistisk modell som har visats vara en god skattare av icke-linjära funktioner. I den här studien kommer två olika neurala nätverksmodeller att undersökas, framåtkopplade nätverk och rekurrenta nätverk. Nätverk från dessa två modeller tränas för att prediktera månatlig avkastning för data på tillgångar som består av makroekonomisk data samt marknadsdata. De predikterade avkastningarna används sedan i en "long-short extended risk parity" portföljstrategi. Prestandan för nätverken samt deras respektive portföljer undersöks och jämförs med en refrensmodell som består av en linjär regression. Olika metriker, såsom genomsnittligt träffvärde, genomsnittligt kvadratiskt fel, portföljvärde och riskjusterad avkastning, används för att evaluera modellernas prestanda. Den linjära regressionsmodellen samt det framåtkopplade nätverket gav en god genomsnittligt träffvärde samt ett lågt genomsnittligt kvadratiskt prediktionsfel, men inte ett bra portföljvärde. De rekurrenta modellerna gav sämre genomsnittligt träffvärde samt ett lite högre genomsnittligt kvadratiskt fel, däremot presterade portföljen mycket bättre.

Acknowledgements

I would like to express my deep gratitude to my supervisor at Aktie-Ansvar AB, Björn Löfdahl for his involvement, patient guidance, insightful feedback, and interest in this project. Many thanks to Tobias Grelsson for his encouragements and support during the preparation of the thesis. I would also like to express my sincere appreciation to the CEO of Aktie-Ansvar AB, Sina Mostafavi, for granting me the opportunity to write my thesis at Aktie-Ansvar AB and all the assistance he has provided me. Finally, my grateful thanks are also extended to my supervisor at KTH, Royal Institute of Technology, Prof. Henrik Hult, for his comments and guidance throughout this project and in other courses during my time at KTH.

Contents

Chapter 1

Introduction

In the world of quantitative portfolio management, a systematic approach is often applied to construct asset portfolios by using different statistical models based on a variety of market data. When constructing portfolios of financial instruments, managers often rely on estimates of the conditional expectation of the instruments' future returns. Thus, it is imperative to develop statistical models that best predict the available data.

Linear regression models are extensively used for prediction, for example in the context of portfolio choice. With the price or the return as the output variable, one can come to understand the linear relation to the explanatory (or input) variables. However, if the relationship between the output and input variables is non-linear, the linear regression model may not be a suitable choice.

An *Artificial Neural Network* (ANN) is a non-linear statistical model that has been shown to be a "good" approximator of non-linear functions, a sort of statistical curve-fitting tool (see [13]). Originally, ANNs were designed to model the human brain, with the aim to emulate brain activity. For that reason, much of the terminology and structure is reminiscent of its origin. As the models have evolved, they can nowadays, in theory, approximate any function. For that reason, ANN are used in a variety of applications, such as prediction and forecasting (see [16]). It can be shown that the linear regression model is a special case of an ANN (see [16]), and thus it seems natural that the next step is to investigate how well ANNs perform when predicting future returns.

The main objective of this work is to investigate if neural network techniques can yield a better prediction power compared to linear regression in portfolio choice. In view of the Kolmogorov's universal approximation theorem (see [6]), neural network techniques should be able to give a better fit compared to the linear regression. However, we would like to know if they can also yield a better prediction given a certain set of data.

In this study, we use the dataset provided by Aktie-Ansvar AB. It consists of monthly returns of 13 different assets $A_1, \ldots, A_{13}$. The explanatory variables of each asset consist of macroeconomic data such as inflation, money supply and current account, and market data such as foreign exchange, yield curves and volatilities. The response variable is the corresponding monthly return.

The dataset is taken at the end of each month from January 31, 2004 to March 31, 2018 (a total of 171 data-points).

We will limit ourselves to only testing two different models of neural networks:

- Feed-forward Neural Network (FNN)
- Recurrent Neural Network (RNN)

as well as experimenting with a few hyperparameters related to each model, which we will elaborate on more thoroughly below. We will compare the prediction power of these models with that of the linear regression using various metrics, as well as their performance on a portfolio, which will be optimized using the obtained predictions.

This work is organized as follows. In Chapter 2, some background theory regarding neural networks and portfolio theory is presented. In Chapter 3, a more in-depth presentation of feed-forward network model is given. Chapter 4 contains a more in-depth presentation of the recurrent network model. In Chapter 5, the methodology for training and selecting the various neural network models is presented. In Chapter 6, the results are displayed, followed by a discussion and analysis. Finally, some conclusions and suggestions for future work are gathered in Chapter 7.

Chapter 2

Background

2.1 Artificial Neural Networks

2.1.1 The Artificial Neuron

The elementary building blocks of the human nervous system are called neurons. Similarly, the building blocks of ANNs are called neurons (or nodes) and are based on Rosenblatt's single-layer perceptron [16]. The neuron consists of a vector of multiple real-valued inputs $\mathbf{X} = (X_1, \ldots, X_r)^T$ and a single output Y. The connection between a input value X_i and an output value Y is indicated with a connection weight β_i. The output is then obtained by computing the activation value U as the sum of $\mathbf{X}$, with their respective weights in the vector $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_r)$, and a bias term β_0:

$$U = \beta_0 + \sum_{i=1}^{r} \beta_i X_i = \beta_0 + \mathbf{X}^T \boldsymbol{\beta},$$

and passing it through an *activation function* f,

$$Y = f(U) = f(\beta_0 + \mathbf{X}^T \boldsymbol{\beta}). \tag{2.1}$$

A visualization is presented in Figure 2.2. We note that selecting the identity function, $f(x) = x$, yields a multiple linear regression. Thus, linear regressions are a special case of neural networks.

The artificial neuron is a primary building block for all ANNs that we will use in this work.

2.1.2 Activation Functions

From the input to each neuron, an output is generated through a transfer function known as the *activation function* (see [6]). Non-linear activation functions are key parts of what gives a non-linear ANN the ability to model non-linear functions. These functions can *squash* an infinite input to a finite

output. That is, they map $\mathbb{R}$ to a finite interval. A common choice are the so-called *sigmoidal* functions, $\sigma(\cdot)$. Apart from their apparent "S-shape" when visualized in a plot, sigmoidal functions are functions that are monotonically increasing and constrained by a pair of horizontal asymptotes as $x \to \pm\infty$. If $\sigma(x) + \sigma(-x) = 1$, then the sigmoidal function is considered *symmetric*, and if $\sigma(x) + \sigma(-x) = 0$, then the sigmoidal function is considered *asymmetric* (see [16]). Examples of commonly used sigmoidal functions are given in Figure 2.1 below.

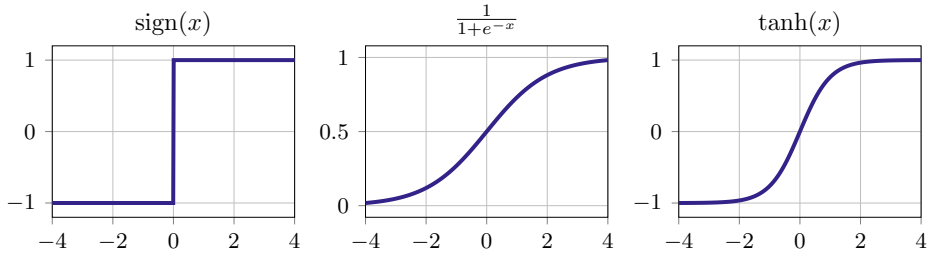


Figure 2.1. *Examples of commonly used sigmoidal activation functions ([16, 6]).*

It is worth noting that the hyperbolic tangent $\tanh : \mathbb{R} \to \mathbb{R}$, defined as

$$\tanh(x) := \frac{e^{2x} - 1}{e^{2x} + 1}, \tag{2.2}$$

is a linear transformation of the logistic function $\sigma : \mathbb{R} \to \mathbb{R}$, defined as

$$\sigma(x) := \frac{1}{1 + e^{-x}} \tag{2.3}$$

such that

$$\tanh(x) = 2\sigma(2x) - 1. \tag{2.4}$$

However, as shown in Figure 2.1, they generate two different output ranges. Thus, one can tailor the selection of activation functions based on the desired output. If, for example, the desired output is a probability (which take values between zero and one), then the logistic sigmoidal is the preferred choice. Furthermore, these functions are easily differentiable, a property that we will see is very useful to possess when training ANNs. Network training (or learning) is a process in which the connection weights of a network are adjusted in accordance with the input values (see [8] for further details).

2.1.3 Network Architecture

In general, the architecture of an ANN consists of multiple neurons (or nodes) that are connected with weights β_{ij}. Depending on how one connects the

neurons, one can obtain many different network structures. In a *fully connected* network, $\beta_{ij} \neq 0$, for all i, j. If there exists a $\beta_{ij} = 0$, the network is considered *partially connected*. In Figure 2.2 below, an example of the simplest of ANN, a single-layer perceptron, is visualized ([16]).

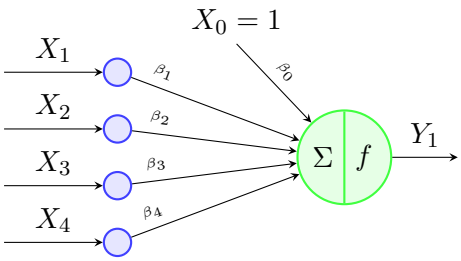


Figure 2.2. *A model of a single-layer perceptron with, $r = 4$ input variables and one output variable. The βs are weights attached to the connections between nodes, β_0 is a bias term, and f is the activation function. (source [16])*

There are several different models of ANNs. As mentioned in [6], the most popular ones can be categorized into Feed-forward Neural Networks (FNN) and Recurrent Neural Networks (RNN). The main difference between the two models is in terms of information flow. In FNNs, the signals travel from input to output, without any information going in between nodes in the same layer. However, in RNNs, the signals are travelling in both directions and between nodes in the same layer.

2.2 Learning Methods

The process of calibrating or fitting an ANN to data is often referred to as learning (or training). Algorithms are used to set weights and other network parameters. These algorithms are called *learning algorithms*. One complete run of a learning algorithm is called an *epoch* ([6]).

Typically, the learning methods are split into three categories:

- **Supervised learning**: This method is a closed-loop feedback system, where the network parameters are adjusted by minimizing the error function, which generally is some variation of the difference between the network output and the desired output. Supervised learning is used in e.g. regression ([6, 5]).
- **Unsupervised learning**: This method involves no target values. Instead, it attempts to draw information from the input data using correlation-detection to find patterns or features without a teacher. Unsupervised learning is used in e.g. clustering ([6, 5]).

- **Reinforcement learning**: This method specifies how an artificial agent should operate and learn from the given input data, using a set of rules aimed to maximize the reward. Reinforcement learning is used in e.g. artificial intelligence ([6, 5]).

2.3 Generalization

The goal of training ANNs is to be able to use the network on unseen data. This is called the *generalization capability* (or the prediction capability) of a network.

Overfitting happens when the network is overtrained for too many epochs or the network has too many parameters. The result may be acceptable for the training data, but when applying the network on new data it will yield poor results. This is due to the network fitting the noise in the data rather than the underlying signal, which is an indication of poor generalization capability. We end up with a bias-variance trade-off (or dilemma), where the requirements for the desirable small bias and small variance are conflicting. The best generalization performance is achieved by balancing bias and variance (see [6]).

There are several methods for controlling and regulating generalization. One way is to stop the training early, meaning you limit the number of epochs the network is trained. Another way is *regularization*, which in the context of supervised learning, modifies the error function by penalization to make the network prefer smaller connection weights, similar in principle to ridge-regression ([16, 6]).

2.4 Training Artificial Neural Networks

So far, the training of ANNs has consisted of passing forward an input set of data and receiving an output set. However, there is no guarantee that one epoch will yield optimal connection weights and a minimal prediction error. Adjusting the connection weights in the network can be made using a supervised learning approach. Since there exists a desired output for every input, the error can be computed. The error signal is then propagated backwards into network and the weights can be adjusted by a gradient-descent-based algorithm. Thus, a closed-loop control system is achieved, analogous to ones in automatic control (cf. [16, 6, 5]).

2.4.1 Loss Function

For supervised learning problems, a typical choice of error function is the squared error (SE)

$$\mathrm{SE} = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2, \tag{2.5}$$

where y_i is the actual output and the $\hat{y}_i$ is the predicted output from the network. The SE is computed each epoch and the learning process is ended when the error is sufficiently small or a failure criterion is met ([6]).

When combining the generalization technique of regularization with the error function (2.5), we get

$$E = \mathrm{SE} + \lambda_c E_c,$$

where E_c is a constraint term, which penalizes poor generalization, and λ_c is a positive valued penalization parameter that balances the trade-off between error minimization and smoothing ([6]).

In this thesis, we restrict our attention to the loss function $\mathcal{L}$ as

$$\mathcal{L}(\mathbf{W}) := \sum_{i=1}^{N} \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|^2 + \lambda_c \sum_{j=1}^{M} \mathbf{W}_j^2, \tag{2.6}$$

where $\mathbf{y}_i$ is the vector of actual outputs, $\hat{\mathbf{y}}_i$ is the vector of predicted outputs from the network, $(\lambda_c > 0)$ is the penalization parameter and $\mathbf{W}$ is the matrix of weights in the network. If $\hat{\mathbf{y}}$ is linear in $\mathbf{x}$, then $\mathcal{L}$ is the loss function for the well known ridge regression problem. To minimize the loss function, a gradient-descent procedure is usually applied.

2.4.2 Gradient-Descent Methods

The simplest algorithm for finding the nearest local minimum of a function, with a computable first derivative, is the *steepest descent method* ([16, 2]).

Algorithm 1 Steepest Descent

1: Select an initial estimate, x_0, for the minimum of $F(x)$.
2: Select a learning parameter, η.
3: **repeat** for $k = 0, 1, 2, \ldots$
4: set $p_k = -\nabla F(x_k)$
5: set $x_{k+1} = x_k + \eta p_k$
6: **until** $\|\nabla F(x_{k+1})\|$ is sufficiently small.

The learning parameter η specifies how large each step should be in the iterative process, i.e. how fast we should move toward a local minimum. As with many things in statistics, there is a trade-off with the selection of

η. If η is too large, the gradient will descend toward a local minimum at a rapid rate. However, this can cause oscillations which can overshoot the local minimum. If η is too small, the gradient will descend toward a local minimum slowly, and the computations can take a very long time ([16]).

Note that the steepest descent Algorithm 1 is not a particularly efficient minimization approach. This is because, although proceeding along a negative gradient works well for near-circular contours, the reality is that in many applications this may not be the case. Here, there is a need for more sophisticated methods (see [2]).

The literature for alternative gradient-descent algorithms is quite extensive, with possible alternative methods such as *Adam* (see [17]), *Adagrad* and *RMSprop* (see [21]) for optimizing the gradient-descent.

Gradient descent is a generic method for continuous optimization. If the objective function $F(x)$ is convex, then all local minima are global, meaning that the gradient descent method is guaranteed to find a global minimum. However, in the case where $F(x)$ is non-convex, the gradient-descent method will converge to a local minimum or a saddle point.

The reasons for selecting gradient descent methods in non-convex problems are:

1. Speed. Gradient descent methods are fast and heavily optimized algorithms are available.

2. A local minimum may be sufficient.

For most neural network configurations, except for the linear regression case, the loss function will not be convex in the weights.

Gradient methods are most efficiently computed using automatic differentiation.

2.4.3 Automatic Differentiation

As noted earlier, the function to which the gradient-descent method is applied has to be differentiable. In the context of ANNs, this means that the activation functions have to be differentiable. For that reason, the selection of activation functions is of great importance.

The way the differential of the function is computed is also of great importance. The methods for computing derivatives in computer programs can be classified into four categories (cf. [3]):

1. Manual derivation and coding in the results

2. Numerical differentiation using finite difference methods

3. Symbolic differentiation

4. Automatic differentiation

From [3], there are some downsides for many of these methods that are too important to ignore, especially when dealing with neural networks:

1. Manual differentiation is a time consuming and error prone endeavour.

2. Numerical differentiation is simple to implement, but can yield highly inaccurate results to due the rounding and truncation which introduces approximation errors. It is also costly to compute in many cases.

3. Symbolic differentiation tackles the weaknesses of both the manual and numerical methods, however it generally yields complex and cryptic expressions plagued with the problem of "expression swell".

The solution to the problems stated above is *automatic differentiation*.

Automatic differentiation consists of two modes, forward and reverse. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$, if the operation count to evaluate f is denoted by $\mathcal{O}(f)$, then the time it takes to compute a $m \times n$ Jacobian is $n \cdot c \cdot \mathcal{O}(f)$ using the forward mode and $m \cdot c \cdot \mathcal{O}(f)$ using the reverse mode, where c is a constant guaranteed to be $c < 6$ (see [10]). In the case of neural networks, scenarios where $n \gg m$ is what generally will occur. For that reason, only the reverse mode is presented in this thesis.

Reverse mode automatic differentiation is represented by the following formula:

$$\frac{\partial f}{\partial x} = \sum_{g \in N_f} \frac{\partial f}{\partial g} \frac{\partial g}{\partial x} \tag{2.7}$$

where N_f is parent nodes of the function node $f(g_1(g_2(\cdots g_n(x))))$.

A well known application of automatic differentiation is the backpropagation algorithm for feed-forward networks, which we will elaborate more on in Chapter 3.

2.5 Portfolio Choice

2.5.1 Long-Short Extended Risk Parity

The reason for predicting the returns, be it with a linear regression or with a neural network, is to aid in the process of selecting the best portfolio for the assets available. In this work, the portfolio selection method we will use is a modified version of the Long-Short Extended Risk Parity portfolio optimization method of [1]. This strategy looks to distribute the total portfolio risk (volatility) equally across the portfolio constituents.

$$\begin{aligned} \underset{\mathbf{w}_t}{\text{maximize}} \quad & \sum_{i=1}^{N_t} |\mu_t^i| \log |w_t^i| \\ \text{subject to} \quad & \sqrt{\mathbf{w}_t^T \boldsymbol{\Omega}_t \mathbf{w}_t} \leq \sigma_{\text{TGT}}, \\ & w_t^i > 0, \text{if } \mu_t^i > 0, \\ & w_t^i < 0, \text{if } \mu_t^i < 0, \end{aligned} \tag{2.8}$$

where, for asset i at time t, μ_t^i is the predicted return, w_t^i is the weight on the asset, $\boldsymbol{\Omega}_t$ is the dispersion matrix, N_t is the number of assets and σ_{TGT} is the volatility target.

In order to best use (2.8), an adjustment on the input vector of predicted returns $\boldsymbol{\mu}_t = (\mu_t^1, \ldots, \mu_t^{N_t})$ is made. We multiply it by

$$\text{sign}(\boldsymbol{\mu}_t) = (\text{sign}(\mu_t^1), \ldots, \text{sign}(\mu_t^{N_t})),$$

which leads to $\boldsymbol{\mu}_t$ having the same sign and the optimization problem becomes a bit easier to solve as the problem will only contain a one sided constraint on the weights. After the weights $\boldsymbol{w}_t$ are determined, they are readjusted with $\text{sign}(\boldsymbol{\mu}_t)$, allowing for long and short positions.

2.5.2 Portfolio Performance Measures

To compare the portfolios selected with the optimization method in eq. (2.8), we use different performance measures, which we review below.

Empirical Value-at-Risk

Let X be the value of a financial portfolio at time 1, say $X = V_1$, then the loss variable is $L = -X$. Consider samples, $L_1, \ldots, L_n$ of independent copies of L, then we estimate VaR at level p by

$$\widehat{\text{VaR}}_p(X) = L_{[np]+1,n}, \tag{2.9}$$

where we have sorted the samples of L as follows: $L_{1,n} \geq \cdots \geq L_{n,n}$. The bracket indicates the floor function ([15]).

Empirical expected shortfall

The empirical expected shortfall (ES) is simply obtained by inserting the empirical VaR into the definition of ES ([15]). We get the following:

$$\widehat{\text{ES}}_p(X) = \frac{1}{p}\left(\sum_{k=1}^{[np]} \frac{L_{k,n}}{n} + \left(p - \frac{[np]}{n}\right) L_{[np]+1,n} \right). \tag{2.10}$$

Sharpe Ratio

An often-used risk-adjusted performance measure for an investment with the return R is the Sharpe ratio S_{Sharpe} where

$$S_{\text{Sharpe}} = \frac{\mathbb{E}[R]}{\sqrt{\text{Var}(R)}}. \tag{2.11}$$

The ratio measures the excess return per unit of deviation in an investment asset [15]. This measure is to be used in relation to other Sharpe ratios and not independently. The higher the ratio is, the better.

Sortino Ratio

The Sortino ratio is a modification of the Sharpe ratio but uses downside deviation rather than standard deviation as the measure of risk, i.e. only those returns falling below a user-specified target are considered risky. The Sharpe ratio penalizes both upside and downside volatility equally, which may not be as desirable considering positive return is almost exclusively desired (cf. [20]).

The Sortino ratio is defined as

$$S_{\text{Sortino}} = \frac{R - \bar{R}}{\text{TDD}}, \tag{2.12}$$

where R is the return, $\bar{R}$ is the target return and TDD is the target downside deviation defined as

$$\text{TDD} = \sqrt{\frac{1}{N}\Sigma_{i=1}^{N}(\min(0, R_i - \bar{R}))^2},$$

where r_i is the i:th return, N is the total number of negative asset returns and T is the same target return as before. The definition is notably very similar to the standard deviation.

This measure is to be used in relation to other Sortino ratios and not independently. The higher the ratio, the better performance.

Chapter 3

Feed-forward Neural Networks

A *Feed-forward Neural Network* (FNN) consists of neurons connected with each other in only one direction, from input to output. In it, the neurons are organized in layers such that there is no connection between the neurons belonging to the same layer. A hidden layer is a computational layer of neurons that is neither part of the input nor the output neurons. The most common type of FNN is the multi-layer FNN, also known as the Multi-Layer Perceptron (MLP) ([6]).

3.1 Network Architecture

An MLP maps the input variables $\mathbf{X} = (X_1, \ldots, X_r)^T$ non-linearly to the output variables $\mathbf{Y} = (Y_1, \ldots, Y_s)^T$. The number of output variables depends on the goal under consideration. In the regression context, one output variable would be similar to a multiple regression, while two or more variables is equivalent to a multivariate regression ([16]).

An MLP with one hidden layer is called a "two-layer network". For N hidden layers, the MLP is called "(N+1)-layer network" ([16]). In Figure 3.1 below, a we present a model of a two-layer network.

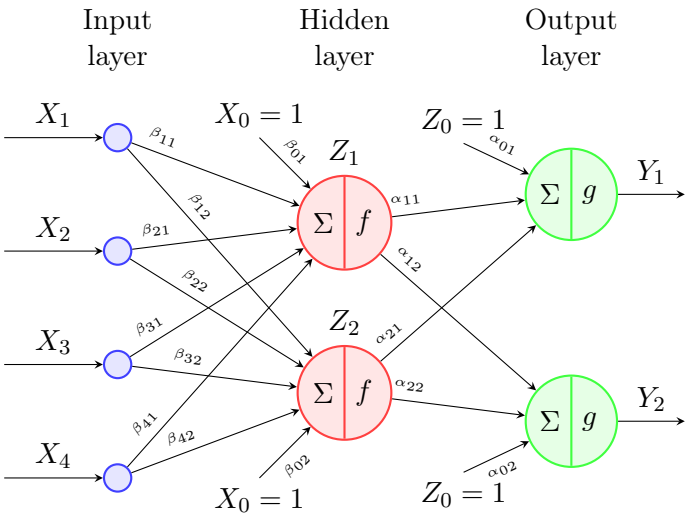


Figure 3.1. *A model of a multi-layer perceptron with one hidden layer, $r = 4$ neurons in the input layer, $s = 2$ neurons in the output layer and $t = 2$ neurons in the hidden layer. The αs and βs are weights attached to the connections between nodes, and f and g are activation functions (source [16]).*

3.1.1 Universal Approximation Theorem

Kolmogorov's universal approximation theorem is an important result used to motivate the usefulness of ANNs (see [6]). It shows that ANNs are a very powerful tool for the approximation of arbitrary continuous functions.

Theorem 3.1 *Any continuous real-valued function $f(x_1, \ldots, x_n)$ defined on $[0, 1]^n$, $n \geq 2$, can be represented in the form*

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{2n+1} h_j \left(\sum_{i=1}^{n} g_{ij}(x_i) \right) \tag{3.1}$$

where g_{ij} and h_i are continuous functions of one variable, and g_{ij} are monotonically increasing functions independent of f.

This means that it is theoretically possible for an FNN, with at least a single hidden layer, to approximate any continuous function, provided the network has a sufficient amount of hidden nodes ([16]).

3.1.2 Single Hidden Layer

Consider a two-layer network consisting of r input nodes $\mathbf{X} = (X_1, \ldots, X_r)^T$, s output nodes $\mathbf{Y} = (Y_1, \ldots, Y_s)^T$ and a single layer of t hidden nodes $\mathbf{Z} = (Z_1, \ldots, Z_t)^T$. Let β_{ij} be the weight of the connection $X_i \to Z_j$ with

bias β_{0j} and let α_{jk} be the weight of the connection $Z_j \to Y_k$ with bias α_{0k}. Set $U_j := \beta_{0j} + \mathbf{X}^T\boldsymbol{\beta_j}$, where $\boldsymbol{\beta_j} = (\beta_{1j}, \ldots, \beta_{rj})$ and $V_k := \alpha_{0k} + \boldsymbol{Z}^T\boldsymbol{\alpha_{jk}}$, where $\boldsymbol{\alpha_j} = (\alpha_{1k}, \ldots, \alpha_{tk})$. Then

$$Z_j = f_j(U_j), \quad j = 1, \ldots, t, \tag{3.2}$$

where $f_j(\cdot)$ is the activation function for the hidden layer and

$$\nu_k(\mathbf{X}) = g_k(V_k), \quad k = 1, \ldots, s, \tag{3.3}$$

where $g_k(\cdot)$ is the activation function for the output layer. Thus, we can express the value of the output node by combining (3.2) and (3.3) as

$$Y_k = \nu_k(\mathbf{X}) + \epsilon_k, \tag{3.4}$$

where ϵ_k is an error term that could be considered Gaussian with mean zero and variance σ_k^2 ([16]).

3.1.3 Multiple Hidden Layers

For N hidden layers, the $(N+1)$-layer network would be expressed, using matrix notation, in the following way:

$$\boldsymbol{\nu}(\mathbf{X}) = \mathbf{g}(\boldsymbol{\alpha}_0 + \mathbf{A}\mathbf{f}(\boldsymbol{\beta}_0 + \mathbf{B}\mathbf{X})), \tag{3.5}$$

where $\boldsymbol{\nu} = (\nu_1, \ldots \nu_s)^T$; $\mathbf{B} = (\beta_{ij})$ is a $(t \times r)$-matrix of weights between the input nodes; $\mathbf{B} = (\beta_{jk})$ is an $(s \times t)$-matrix of weights between the hidden layer and the output layer; $\boldsymbol{\beta_0} = (\beta_{01}, \ldots \beta_{0t})^T$ and $\boldsymbol{\alpha_0} = (\alpha_{01}, \ldots \alpha_{0k})^T$ are the bias vectors; $\mathbf{f} = (f_1, \ldots f_t)^T$ and $\mathbf{g} = (g_1, \ldots g_k)^T$ are the vectors of activation functions ([16]).

Similar to the single-layer perceptron, when the activation functions $f(\cdot)$ and $g(\cdot)$ are equal to the identity function, then, (3.3) collapses into a multivariate reduced rank regression ([16]).

3.2 Training Feedforward Neural Networks

3.2.1 The Backpropagation-of-Errors Algorithm

The industry standard for training FNNs is the backpropagation-of-errors (BP) algorithm. As mentioned earlier in Chapter 2 the BP-algorithm is essentially a special case of automatic differentiation and gradient-descent (see [16]).

The BP-algorithm efficiently computes the first derivatives of an error function with regards to the connection weights. Later, the derivatives are used in iterative gradient-descent methods to adjust the connection weights by

minimizing the chosen error function. In order to implement the algorithm on a FNN, the activation functions have to be continuous, nonlinear, monotonically increasing and differentiable ([16, 6]).

In the following part, for simplicity, we apply the BP-algorithm on the two-layer network visualized in Figure 3.1 following the instructions in [16]. The process can be applied for other kinds of ANNs.

The set of r input nodes is denoted by $\mathcal{I}$, the set of s output nodes is denoted by $\mathcal{K}$ and the set of t input nodes is denoted by $\mathcal{J}$. As such, $i \in \mathcal{I}$ indexes an input node, $k \in \mathcal{K}$ indexes an output node and $j \in \mathcal{J}$ indexes a hidden node. The current epoch is indexed by l, such that $l = 1, 2, \ldots n$.

Starting at the k-th output node, the error signal that has been propagated back after the forward sweep is denoted by

$$e_{l,k} = y_{l,k} - \hat{y}_{l,k}, \quad k \in \mathcal{K} \tag{3.6}$$

where $y_{l,k}$ is the desired output and $\hat{y}_{l,k}$ is the actual network output, at node k during epoch l.

The optimizing criterion, in this example, is the *Error Sum of Squares* (ESS), which is defined as

$$\mathrm{E}_l = \frac{1}{2}\sum_{k\in\mathcal{K}}(y_{l,k} - \hat{y}_{l,k})^2 = \frac{1}{2}\sum_{k\in\mathcal{K}} e_{l,k}^2. \tag{3.7}$$

The supervised learning problem is to minimize the MSE with regards to the connection weights in the network, in this case $\{\alpha_{jk}\}$ and $\{\beta_{ij}\}$.

We let

$$v_{l,k} = \sum_{j\in\mathcal{J}} \alpha_{l,jk} z_{l,j} = \alpha_{l,0k} + \mathbf{z}_l^T \boldsymbol{\alpha}_{l,k}, \quad k \in \mathcal{K}, \tag{3.8}$$

where $z_{l,0} = 1$, $\mathbf{z}_l = (z_{l,1}, \ldots, z_{l,t})^T$ and $\boldsymbol{\alpha}_l = (\alpha_{l,1}, \ldots, \alpha_{l,t})^T$. The output generated from the network is

$$\hat{y}_{l,k} = g_k(v_{l,k}), \quad k \in \mathcal{K}, \tag{3.9}$$

with $g_k(\cdot)$ being a differentiable activation function.

After every epoch, the weights $\alpha_{l,jk} = (\boldsymbol{\alpha}_{l,1}, \ldots, \boldsymbol{\alpha}_{l,s}) = (\alpha_{l,jk})$ are updated using the gradient-descent method. Letting $\boldsymbol{\alpha}_l$ be the ts vector of all the hidden-layer-to-output-layer weights at the l-th iteration, the update rule becomes

$$\boldsymbol{\alpha}_{l+1} = \boldsymbol{\alpha}_l + \Delta\boldsymbol{\alpha}_l, \tag{3.10}$$

where

$$\Delta\boldsymbol{\alpha}_l = -\eta\frac{\partial E_l}{\partial \boldsymbol{\alpha}_l} = \left(-\eta\frac{\partial E_l}{\partial \alpha_{l,jk}}\right) = (\Delta\alpha_{l,jk})\,. \tag{3.11}$$

Similar update rules applies to the bias term $\alpha_{l,0k}$ as well.

Applying the chain rule to (3.11) yields

$$\begin{aligned}\frac{\partial E_l}{\partial \alpha_{l,jk}} &= \frac{\partial E_l}{\partial e_{l,k}} \cdot \frac{\partial e_{l,k}}{\partial \hat{y}_{l,k}} \cdot \frac{\partial \hat{y}_{l,k}}{\partial v_{l,k}} \cdot \frac{\partial v_{l,k}}{\partial \alpha_{l,jk}} \\ &= e_{l,k} \cdot (-1) \cdot g'(v_{l,k}) \cdot z_{l,j}. \\ &= -e_{l,k} g'(\alpha_{l,0k} + \mathbf{z}_l^T \boldsymbol{\alpha}_{l,k},) z_{l,j}\end{aligned} \tag{3.12}$$

It is possible to express this in terms of the *sensitivity* (or *local gradient*) of the l-th epoch, at the k-th output node. Thus,

$$\frac{\partial E_l}{\partial \alpha_{l,jk}} = -\delta_{l,k} z_{l,j} \tag{3.13}$$

where

$$\delta_{l,k} := \frac{\partial E_l}{\partial \hat{y}_{l,k}} \cdot \frac{\partial \hat{y}_{l,k}}{\partial v_{l,k}} = e_{l,k} g'(v_{l,k}). \tag{3.14}$$

This means that the gradient-descent update for $\alpha_{l,jk}$ is

$$\alpha_{l+1,jk} = \alpha_{l,jk} - \eta \frac{\partial E_l}{\partial \alpha_{l,jk}} = \alpha_{l,jk} + \eta \delta_{l,k} z_{l,j}. \tag{3.15}$$

This process is now repeated for the connection weights between the i-th input node to the j-th hidden node.

For the l-epoch, we let

$$u_{l,j} = \sum_{i \in \mathcal{I}} \beta_{l,ij} x_{l,i} = \beta_{l,0j} + \mathbf{x}_l^T \boldsymbol{\beta}_{l,j}, \quad j \in \mathcal{J}, \tag{3.16}$$

where $x_{l,0} = 1$, $\mathbf{x}_l = (x_{l,1}, \ldots, x_{l,r})^T$ and $\boldsymbol{\beta}_{l,j} = (\beta_{l,1j}, \ldots, \beta_{l,rj})^T$.

The output generated from the network is

$$z_{l,j} = f_j(u_{l,j}), \quad j \in \mathcal{J}, \tag{3.17}$$

with $f_j(\cdot)$ being a differentiable activation function, at the j-th hidden node.

After every epoch, the weights $\beta_{l,ij}$ are updated using the gradient-descent method. Letting $\boldsymbol{\beta}_l$ be the rt vector of all the hidden-layer-to-output-layer weights at the l-th iteration, the update rule becomes

$$\boldsymbol{\beta}_{l+1} = \boldsymbol{\beta}_l + \Delta \boldsymbol{\beta}_l, \tag{3.18}$$

where

$$\Delta \boldsymbol{\beta}_l = -\eta \frac{\partial E_l}{\partial \boldsymbol{\beta}_l} = \left(-\eta \frac{\partial E_l}{\partial \beta_{l,ij}} \right) = (\Delta \beta_{l,ij}) \,. \tag{3.19}$$

Similar update rules applies to the bias term $\beta_{l,0j}$ as well.

Applying the chain rule to (3.19) yields

$$\frac{\partial E_l}{\partial \beta_{l,ij}} = \frac{\partial E_{l,j}}{\partial z_{l,j}} \cdot \frac{\partial z_{l,j}}{\partial u_{l,j}} \cdot \frac{\partial u_{l,j}}{\partial \beta_{l,ij}}, \tag{3.20}$$

where

$$\begin{aligned} \frac{\partial E_{l,j}}{\partial z_{l,j}} &= \sum_{k \in \mathcal{K}} e_{l,k} \cdot \frac{\partial e_{l,k}}{\partial z_{l,j}} \\ &= \sum_{k \in \mathcal{K}} e_{l,k} \cdot \frac{\partial e_{l,k}}{\partial v_{l,k}} \cdot \frac{\partial v_{l,k}}{\partial z_{l,j}} \\ &= -\sum_{k \in \mathcal{K}} e_{l,k} \cdot g'(v_{l,k}) \cdot \alpha_{l,jk} \\ &= -\sum_{k \in \mathcal{K}} \delta_{l,k} \alpha_{l,jk}. \end{aligned} \tag{3.21}$$

Thus, (3.20) becomes

$$\frac{\partial E_l}{\partial \beta_{l,ij}} = -\sum_{k \in \mathcal{K}} \delta_{l,k} \alpha_{l,jk} f'(\beta_{l,0j} + \mathbf{x}_l^T \boldsymbol{\beta}_{l,j}) x_{l,i}. \tag{3.22}$$

Similar to (3.14), we can set

$$\delta_{l,j} := f'(u_{l,j}) \sum_{k \in \mathcal{K}} \delta_{l,k} \alpha_{l,jk}. \tag{3.23}$$

This means that the gradient-descent update for $\beta_{l,ij}$ is

$$\beta_{l+1,ij} = \beta_{l,ij} - \eta \frac{\partial E_l}{\partial \beta_{l,ij}} = \beta_{l,ij} + \eta \delta_{l,j} x_{l,i}. \tag{3.24}$$

The training of a FNN consists of a *forward pass* and a *backpropagation pass*. After setting an error function and selecting the initial weights of the network, the backpropagation algorithm is used to compute the necessary corrections (3.15) and (3.24). The backpropagation algorithm reads:

Algorithm 2 Backpropagation

Initialize the connections weights $\boldsymbol{\beta}_0$ and $\boldsymbol{\alpha}_0$
Calculate the error function E.
for each epoch $l = 1, 2 \ldots, n$ **do**:
    Calculate the error function E_l.
    if the error E_l is less than a threshold **then return**
    end if
    for each input $x_{k,ij}$, $i = 1, 2, \ldots r$ **do**
        procedure FORWARD PASS(Inputs enter the node from the left and emerge from the right of the node.)
            Compute the output node using (3.17) and then (3.9).
        end procedure
        procedure BACKPROPAGATION PASS(The network is run in reverse order, layer by layer, starting at the output layer.)
            Calculate the error function E_l.
            Update the connections weights, between the output and the hidden layer that is to the left of it, using (3.15).
            Update the connections weights, between the hidden and input layer that is the left of it, using (3.24).
        end procedure
    end for
end for

This iterative process is repeated until some suitable stopping time (cf. [16, 6, 7]).

Chapter 4

Recurrent Neural Networks

ANNs are modeled after the human brain. But humans do not throw out all memory and start thinking from scratch every time. In fact, human thoughts have some persistence in the brain. The brain possesses a strongly recurrent connectivity. This is one of the shortcomings of FNN. It lacks a recollection functionality. As it has a static structure, going only from input to output, it cannot deal with sequential or temporal data. A proposed solution to these problems is the *Recurrent Neural Network* (RNN) (see [6]).

In principle, an RNN is capable of mapping the entire history of previous inputs to each output. This recollection functionality allows previous input data to persist in the network, which can thereby influence the output, similar to a human brain ([8]).

4.1 Network Architecture

When running temporal data through a neural network, one has to run the data for each time step through parallel neural networks as visualized in Figure 4.1.

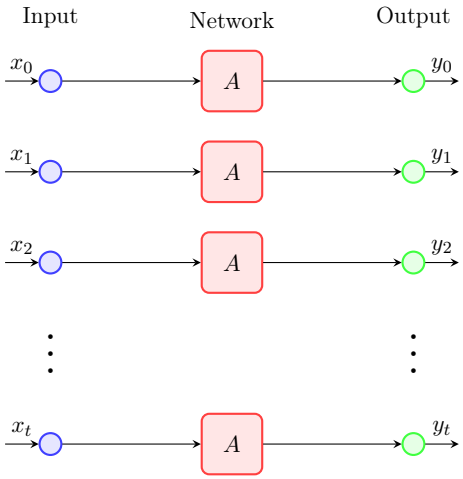


Figure 4.1. *A model of parallel MLPs, A, that looks at some input x_t and outputs a value x_t for $t = 0, 1, 2, \ldots, t$.*

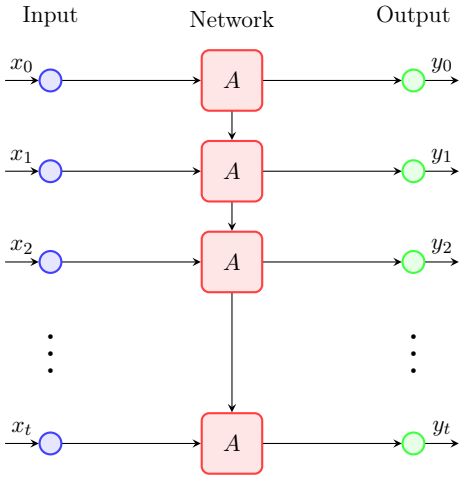


Figure 4.2. *Model of an unfolded recurrent neural network where A is a neural network, that looks at some input x_t and outputs a value y_t for $t = 0, 1, 2, \ldots, t$.*

In short, one could draw an RNN in the following way

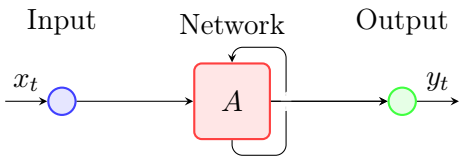


Figure 4.3. *A model of a recurrent neural network.*

All RNNs have the form of a chain of repeating modules (or blocks) of neural network, with each network passing information to the next. In basic RNN, this repeating module will have a very simple structure, such as a node with a single activation function. A visualization is presented in Figure 4.4. The visualization is based on a similar design presented in [19].

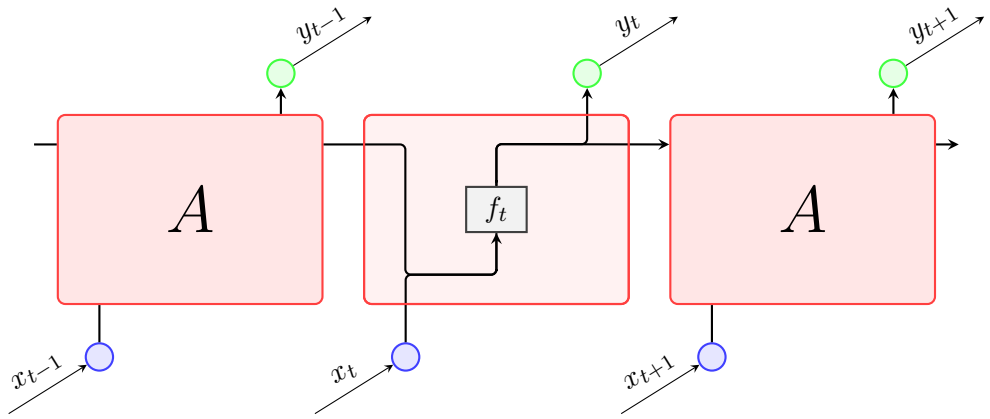


Figure 4.4. *Repeating module of a basic recurrent neural network.*

Using the notation in [9], e have an input sequence $x = (x_1, \ldots, x_t)$, a hidden vector sequence $h = (h_1, \ldots, h_t)$ and an output vector sequence $y = (y_1, \ldots, y_t)$, which the RNN computes. For time t, the RNN module has the following composition:

$$h_t = f_t(W_{ih}x_t + W_{hh}h_{t-1} + b_h) \tag{4.1}$$

$$y_t = g_t(W_{ho}h_t + b_o) \tag{4.2}$$

where the W terms denote weight matrices (e.g. W_{ih} is the input-hidden weight matrix), the b terms denote bias vectors (e.g. b_h is the hidden bias vector) and f_t is the hidden layer activation function.

Basic RNNs are not very useful in practice. The problem is typical of deep neural networks in that the gradients of a given input on the hidden layer, and therefore on the network output, either decays or blows up exponentially as it cycles around the network's recurrent connections. This effect is known as the *vanishing gradient problem* or the *exploding gradient problem* (see [8, 11]).
As a result, there are a few modified RNN models available. Among them the *Long Short-Term Memory Networks* (see [14]), and *Gated Recurrent Units* (see [18]).

4.2 Long Short-Term Memory Networks

Proposed in [14], the *Long Short-Term Memory* (LSTM) network architecture was explicitly designed to deal with the long-term dependency problem and to make it easy to remember information over long periods of time until it is needed ([11]).

The basic RNN module consisted of only an activation function. The LSTM module has a more complex structure. The architecture is presented in Figure 4.5.

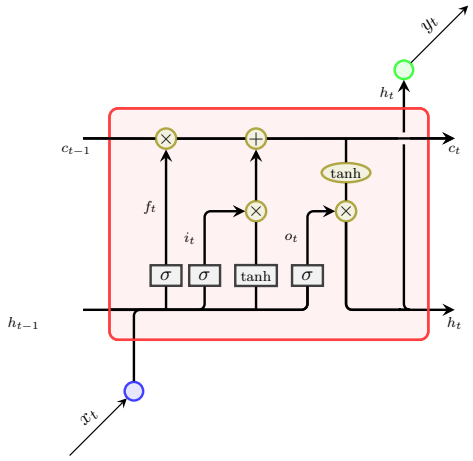


Figure 4.5. *Repeating module of a long short-term memory neural network.*

Instead of having a single activation function, the LSTM module has four (see [22, 11, 19]).

1. **Cell state**: The key feature is the cell state, C_t which remembers the information over time. Gates modulate the information flow, by regulating the amount of information that goes in to the cell state.

2. **Forget gate**: To decide what information should remain or be discarded from the cell state, a forget gate is used. It is a sigmoid which uses h_{t-1} and x_t, and returns a value between zero (forget) and one (remember).

3. **Input gate**: The LSTM module receives inputs from other parts of as well. The input gate, i_t is a sigmoid that decides which values are going to be updated.

4. **Output gate**: Lastly, a decision is made regarding what the LSTM module should output. This output is based on a filtered version of the cell state information. Firstly, a sigmoid activation function o_t decides which parts of the cell state the LSTM module will output. Then, the

cell state is passed through a tanh activation function and multiplied with the output of the sigmoid gate o_t. The result h_t is passed on to the rest of the network.

Following the implementation in [9], the components in the LSTM module have the following composition:

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_i) \tag{4.3}$$
$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_f) \tag{4.4}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tanh(W_{xc} + W_{hc}h_{t-1} + b_c) \tag{4.5}$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_t + b_o) \tag{4.6}$$
$$h_t = o_t \odot \tanh(c_t) \tag{4.7}$$

where σ is the logistic sigmoid function, and i, f, o and c are respectively the input gate, forget gate, output gate and cell state vector, all of which are the same size as the hidden vector h. The weight matrix subscripts have the obvious meaning, for example W_{hi} is the hidden-input gate matrix, W_{xo} is the input-output gate matrix etc. The weight matrices from the cell to gate vectors (e.g. W_{ci}) are diagonal, so element m in each gate vector only receives input from element m of the cell vector. For each gate, there is a bias terms b. The $\odot$ operator represents element-wise multiplication.

4.3 Gated Recurrent Units

A variant on the LSTM network is the Gated Recurrent Unit (GRU). Presented in [18], it is an increasingly popular simplified version of the LSTM network ([19]).

Similar to the LSTM network, a GRU tries to capture the long-term dependency using gating mechanisms. However, there are a few differences, most notably the lack of memory cell state c_t, that is the central feature of an LSTM module. Instead, the GRU has a reset gate, r_t, which determines how the combination of the previous memory and the new input should be, and an update gate, z_t, which determines how much of the previous memory the GRU should remember. In Figure 4.6, the network architecture of the GRU module is presented.

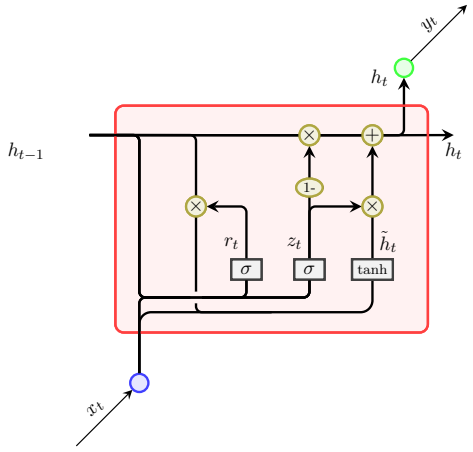


Figure 4.6. *Repeating module of a gated recurrent unit.*

Following the implementation in [23], the operations of a GRU are represented by the following equations:

$$h_t = (1 - z_t)h_{t-1} + z_t\tilde{h}_t \tag{4.8}$$
$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \tag{4.9}$$
$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1})) \tag{4.10}$$
$$r_t = \sigma(W_r x_t + U_r h_{t-1}) \tag{4.11}$$

where the vectors h_t is the output from GRU, z_t is the update gate, r_t is the reset gate and $\tilde{h}_t$ is the candidate output. The weight matrices in the GRU are W_h, W_z, W_r, U_h, U_z and U_r. The biases are omitted.

4.4 Training Recurrent Neural Networks

Similar to the FNN, the scheme used for training RNNs is also backpropagation. However, as RNNs have a temporal aspect, a modified version is required, namely the *Backpropagation Through Time* (BPTT)-algorithm (see [12]).

The BPTT-algorithm is simply BP-algorithm applied to an unrolled RNN, which, as mentioned before, becomes a deep-FNN. The key difference is that, for RNNs, the loss function depends on the activation of the hidden layer through both the output layer and the hidden layers at the next time-step. This is because the RNN share parameters across layers. All neural networks are just nested functions like $f(g(h(x)))$. The same chain rule applies to RNNs, with the difference between the FNN and RNN being the time element. The series of functions will only extend when adding a time element ([8, 12]).

Chapter 5

Methodology

In this chapter, we explain how the network selections and portfolio choice were conducted. We compare the prediction ability and performance of networks from three different models (single-layer FNN, basic RNN and GRU) with the benchmark model (linear regression). This is done on 13 financial assets $A_1, \ldots, A_{13}$, where we use the same network type and parameters on each asset A_i. Thus, after setting the hyperparameters of a specific network type, we train 13 networks for the 13 assets.

5.1 Data

For each asset A_i, there is a corresponding monthly return. We aggregate monthly returns into six month returns, $y_{i,t}$, and set this data as our response variable. There are also 16 corresponding input variables $x_{1,i,t}, \ldots, x_{16,i,t}$, for each asset A_i. These are proprietary explanatory variables that have been provided by Aktie-Ansvar AB. These variables are believed to best explain the predicted return. They consist of macroeconomic data such as inflation, money supply and current account, and market data such as foreign exchange, yield curves and volatilities. We call the explanatory variables "indicators".

Financial data are time series data, which means that the order the appear in is crucial and the next data point is dependent on the previous. Financial data is also a very limited commodity. The dataset we have at our disposal is taken at the end of each month from January 31, 2004 to March 31, 2018 (a total of 171 data-points).

To summarize, the data that goes in to the models are the explanatory variables $x_{1,i,t}, \ldots, x_{16,i,t}$. The networks will then yield the relation between response $y_{i,t}$ and the explanatory variables.

The implementation is done in Julia, a high-level, high-performance dynamic programming language for numerical computing (see [4]).

5.2 Training the Networks

5.2.1 Loss Function

The loss function $\mathcal{L}$ used for all networks is the squared error (SE) with added regularization

$$\mathcal{L}(\mathbf{W}) := \sum_{i=1}^{N} \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|^2 + \lambda_c \sum_{j=1}^{M} \mathbf{W}_j^2, \tag{5.1}$$

where $\mathbf{y}_i$ is the vector of actual outputs, $\hat{\mathbf{y}}_i$ is the vector of predicted outputs from the network, λ_c is the penalization parameter and $\mathbf{W}$ is the matrix of weights in the network.

5.2.2 Gradient-Descent Algorithm

The gradient-descent optimization algorithm we use is RMSprop (cf. [21]), which is defined as follows:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}\left[g^2\right]_t}} g_t \tag{5.2}$$

where g_t is the gradient of the loss function at time-step t, θ is the matrix of network weights and

$$\mathbb{E}\left[g^2\right]_t = (1-\rho)\mathbb{E}\left[g^2\right]_{t-1} + \rho g_t^2,$$

where ρ is the decay parameter, which we set to $\rho = 0.02$. The decay determines how much of the old information is retained and how much of the new information is absorbed.

5.2.3 Hyperparameters

A hyperparameter is a parameter whose value is set before the learning process begins. After much experimentation, we decided to vary the learning rate η, penalization factor λ_c, the number of epochs, and the network structure in terms of number of hidden layers and number of nodes. We have set the number of indicators and signals which determine the number of input nodes to 16 and output nodes to one.

5.2.4 Training the Networks

For this project, due to the time series nature of the data, we have decided to not split the data into the classical training, validation and test sets. Instead, we will proceed as follows: for each time step, we train the network on all data points available up to that time and use the next data point as a test set, that is, we will always try to predict one step ahead based on all the

available information up to that time. For each network, the training that we perform is called the *initial training*. For each time step thereafter, the training is referred to as *incremental training*. This means that we have 61 overlapping training sets and 60 non-overlapping test sets.

Initial Training

On each network model, we make an initial training on a set of training data according to the learning methods of each model as described earlier in the thesis. The initial training set consists of 105 time-steps, representing one financial cycle (around 8-10 years). In order for the models to be able to capture the signal in the data (instead of the noise), we need a "good" amount of data to train the models on. For that reason we use data points from 105 time-steps and not fewer. Furthermore, having data from an entire financial cycle increases the chance for exposure to upturns and downturns.

The data that goes in the model are initialized weights (using randomization) and the explanatory variables $x_{1,i,t}, \ldots, x_{16,i,t}$. The network then predicts a return $\hat{y}_{i,t}$ which we call a "signal". Then we compute the loss function and use backpropagation algorithm to then adjust the weights. This is repeated for each epoch (lap) until we decide it is time to stop.

The actual return $y_{i,t}$ and the predicted return $\hat{y}_{i,t}$ are returns with a return period of six months. The reason for using a time horizon of six months is that macroeconomic data typically describe long term occurrences as opposed to short time occurrences like one day or one month. This means that, for example, $\hat{y}_{i,t}$ will be the six month return from the month of January up to and including the month of June and $\hat{y}_{i,t+1}$ will the the six month return from the month of February up to and including the month of July. This procedure will reduce the amount of data we will have by six data points.

To evaluate the selection of model (i.e. to determine if the choice of hyperparameters is suitable), we look at the in-sample plot of each initial training run. An example is presented in Figure 5.1. From the plot we look at the resulting fit and change the hyperparameters accordingly. From the bias-variance trade-off, we get that if the fit is too good, the prediction ability of the network will probably be limited as the data contains a lot of noise. Furthermore, if the fit is simply a straight line (i.e. zero), then that is equivalent to not taking any position at all and we consider that to not be sufficient at all. What we look for is something in the middle between these two extreme cases, which is what we consider a "good enough" fit. This is adjusted by early stopping, meaning we select a number of epochs the network is trained on.

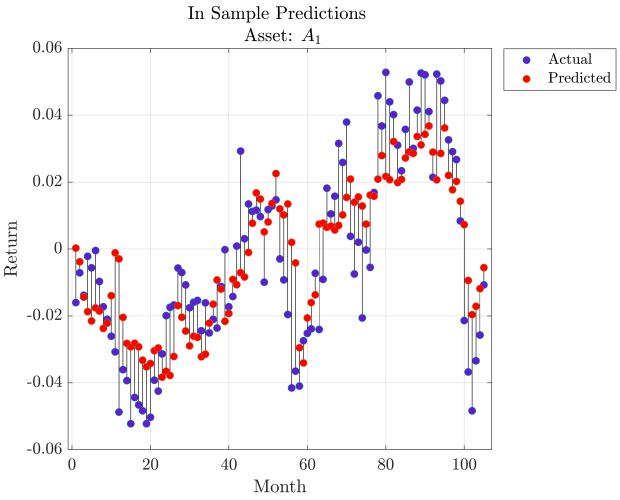


Figure 5.1. *An example of an in-sample plot*

Incremental Training

For each time-step, we add the actual return $y_{i,t}$ to the data set the network trains on, train the network again, and predict the next time-steps future return. This means that we test for one time step at a time. For each time step, the training set increases by one data point.

The weights obtained after each incremental training step are used as the initial guess for the next incremental training step. The reason for this is to speed up the training as it is more likely that the next steps weights will be closer to the previous steps weight than it is to randomized weights. The incremental training is performed on the remaining 60 data points that are left when using a return period of six months. The result is then analyzed using an out-of-sample plot which shows how well the network managed to predict the future return. An example is presented in Figure 5.2.

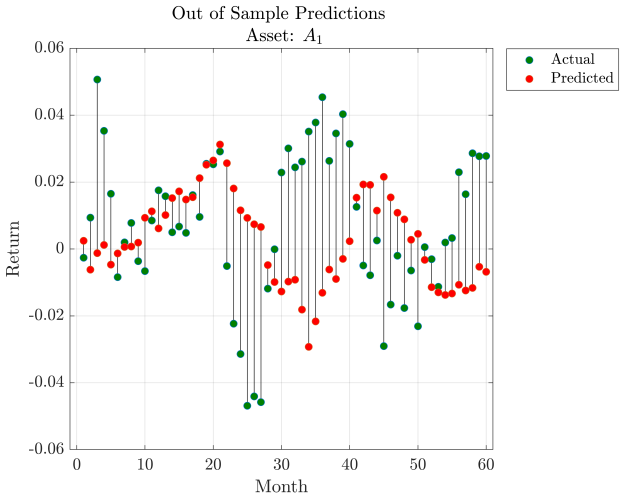


Figure 5.2. *An example of an out-of-sample plot*

What we look for in an out-of-sample plot is that the predicted returns are as close to the actual return as possible.

5.3 Evaluating the Networks

After having trained the networks and obtained the resulting predicted returns, we apply several metrics to determine the prediction ability of the networks and compare that to the prediction ability of the benchmark.

The hit-rate and the mean squared prediction error combined with the out-of-sample-plot are used to determine the prediction ability of the network.

5.3.1 Hit-Rate

The hit-rate is defined as the number of times the predicted signal's sign matches the actual signal's sign. For each of the 60 test sets, the hit-rate is computed and accumulated. For each step of the incremental training, we compute the average hit-rate. We will pay close attention to the final time step's average hit-rate.

The motivation for using the hit-rate is that often times, since the data is quite noisy, predicting the sign of the return may be sufficient when determining the position one will take on the asset. Furthermore, it is much easier to predict the sign than it is to predict the actual return. An example of a plot of the hit-rates is presented in Figure 5.3. A models average hit-rate is the final time-step's average hit-rate.

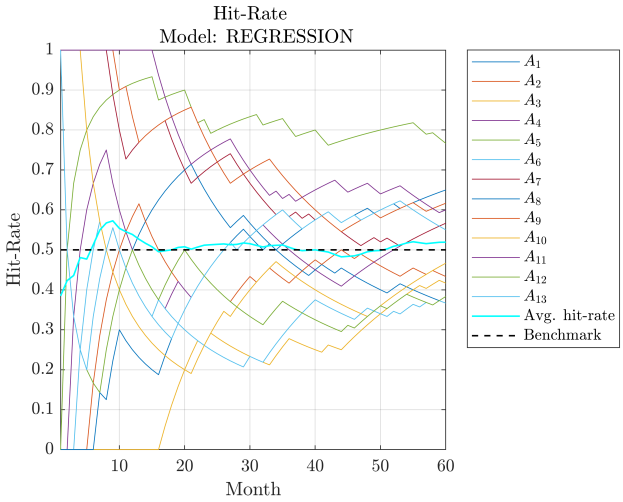


Figure 5.3. *Plot of the average hit-rate for each asset using the benchmark model.*

5.3.2 Mean Squared Prediction Error

The mean squared prediction error (MSPE) is defined as the average of the squared difference between the predicted signal and the actual signal at each time step.

$$\text{MSPE} = \frac{1}{N}\sum_{k=1}^{N}(y_{i,k} - \hat{y}_{i,k})^2, \tag{5.3}$$

where $\hat{y}_{i,k}$ is the predicted value of the signal at time step k and $y_{i,k}$ is the actual value of the signal at time step k. A models mean squared prediction error is the average of the final time-steps squared-errors

The mean squared prediction error determines how far the prediction is from the actual value. The hit-rate only determines whether the predicted sign is correct, but not how far the prediction is from the actual value.

5.4 Application on a Portfolio Strategy

When the best networks have been selected, we implement the results on a portfolio strategy and determine the value of the portfolio. That is, we use the predictions from the models to balance the portfolio using a selected optimization method. The performance of the portfolio is then used as a measure for the performance of the network with regards to its prediction ability over a time horizon equivalent to the length of the test set, which in this case is 60 months.

5.4.1 Portfolio Optimization Method

The portfolio optimization method used is an adjusted version of the Long-Short Extended Risk-Parity method presented in eq. (2.8). Here, the monthly volatility target is $\sigma_{\text{TGT}} = 2.89\%$, which translates to a yearly volatility of 10%, a realistic value.

5.4.2 Performance Metrics

The performance of the subsequent portfolios will be determined by looking at the value of the portfolio at the end of the time horizon, the maximum drawdown, the yearly return, the value-at-risk and expected shortfall at a 5% level and the Sharpe and Sortino ratios. Since we only possess 60 predicted monthly returns, the value-at-risk and expected shortfall measures are to be taken with caution due to the limited number of data used to compute them.

Chapter 6

Results

Trial and error yielded the following settings to the penalization term λ_c presented in Table 6.1.

Network	**Gate**	λ_c
Feed-forward	output-gate	0.0001
Recurrent	output-gate	0.0001
Gated-recurrent-unit	relevance-gate	0.0001
Gated-recurrent-unit	probablility-gate	0.0001
Gated-recurrent-unit	output-gate	0.0001

Table 6.1. *Penalization values λ_c for each gate in the networks.*

6.1 Benchmark Network

In this section, we present the results obtained from the benchmark model, which in this thesis is the linear regression model. It is a feed-forward network with no hidden layers and the identity function as the activation function.

6.1.1 Training the Network

In Table 6.2, the settings for the training of the network is presented.

Training stage	**Learning rate**	**Epochs**
Initial training	0.001	1500
Incremental training	0.0001	500

Table 6.2. *Settings for the training of the benchmark network.*

6.1.2 Prediction Performance

In Table 6.3, the prediction performance of the network is presented. In Figure 6.1, the hit-rate for each asset over time is presented.

Metric	Value
Average hit-rate	51.923%
MSPE	0.00132

Table 6.3. *Prediction performance of the benchmark network.*

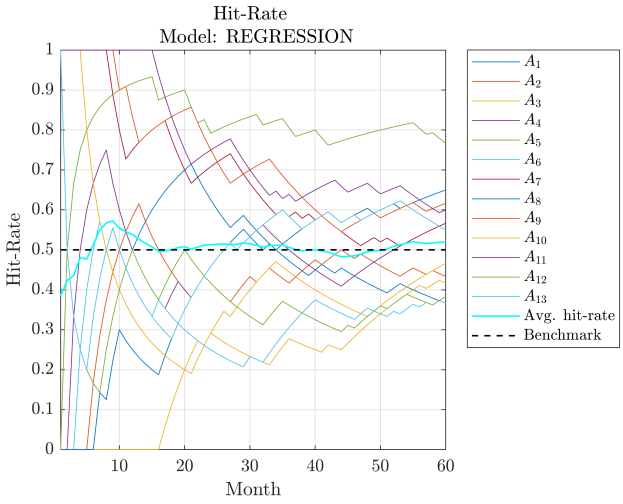


Figure 6.1. *Plot of the average hit rate for each asset using the benchmark model.*

6.1.3 Portfolio Performance

The performance of the portfolio built on the signals yielded from this model is presented in Figure 6.2 and Table 6.4.

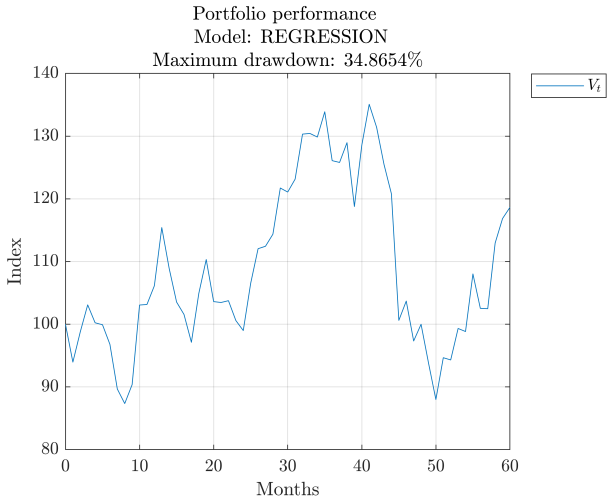


Figure 6.2. *Performance of the portfolio based on the benchmark model.*

Metric	Value
Portfolio value V_{60}	118.6444
Yearly return	3.4783%
Maximum drawdown	34.8654%
$\text{VaR}_{0.05}$	6.2485%
$\text{ES}_{0.05}$	10.6543%
Yearly Sharpe ratio	0.1788
Yearly Sortino ratio	1.0478

Table 6.4. *Performance and risk measures of the portfolio based on the benchmark model.*

6.2 Feed-Forward Network

In this section, we present the results obtained from the feed-forward network. The best feed-forward network we could train is a feed-forward with one hidden layer and four nodes with the *tanh* function as the activation function.

6.2.1 Training the Network

In Table 6.5, the settings for the training of the network is given.

Training stage	Learning rate	Epochs
Initial training	0.001	1500
Incremental training	0.0001	500

Table 6.5. *Settings for the training of the feed-forward network.*

6.2.2 Prediction Performance

In Table 6.6, the prediction performance of the network is presented. In Figure 6.3, the hit-rate for each asset over time is presented.

Metric	Value
Average hit-rate	50.897%
MSPE	0.00137

Table 6.6. *Prediction performance of the best feed-forward network we managed to train.*

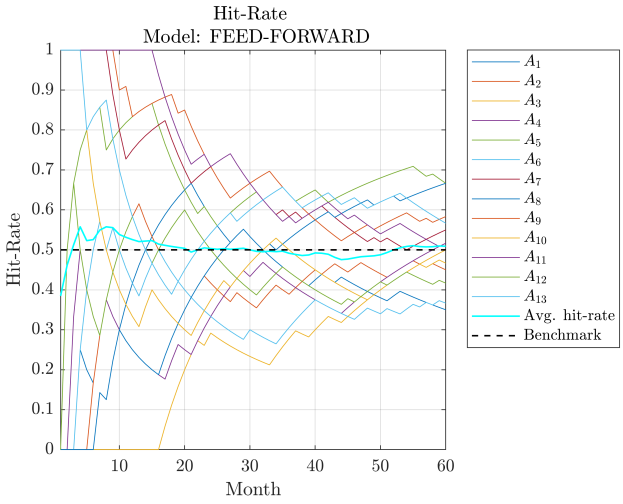


Figure 6.3. *Plot of the average hit rate for each asset using the best feed-forward model we managed to train.*

6.2.3 Portfolio Performance

The performance of the portfolio built on the signals yielded from this model is presented in Figure 6.4 and Table 6.7.

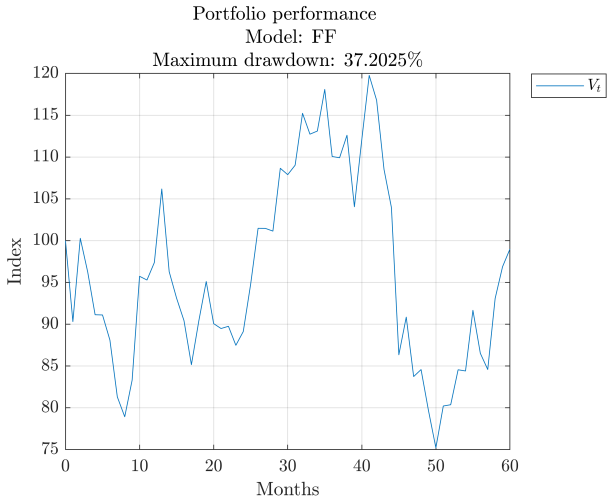


Figure 6.4. *Performance of the portfolio based on the best feed-forward model we managed to train.*

Metric	Value
Portfolio value V_{60}	99.0061
Yearly return	-0.1996%
Maximum drawdown	37.2025%
$\text{VaR}_{0.05}$	7.8255%
$\text{ES}_{0.05}$	11.9826%
Yearly Sharpe ratio	-0.009529
Yearly Sortino ratio	-0.05763

Table 6.7. *Performance and risk measures of the portfolio based on the best feed-forward model we managed to train.*

6.3 Recurrent Network

In this section, we present the results obtained from the recurrent network with a basic unit (which we will refer to as the recurrent network). The best recurrent network we could train is one with a single hidden layer with three nodes with the *tanh* function as the activation function.

6.3.1 Training the Network

In Table 6.8, the settings for the training of the network is presented.

Training stage	Learning rate	Epochs
Initial training	0.001	750
Incremental training	0.0001	1000

Table 6.8. *Settings for the training of the recurrent network.*

6.3.2 Prediction Performance

In Table 6.9, the prediction performance of the network is presented. In Figure 6.3, the hit-rate for each asset over time is presented.

Metric	Value
Average hit-rate	49.231%
MSPE	0.00429

Table 6.9. *Prediction performance of the best recurrent network we managed to train*

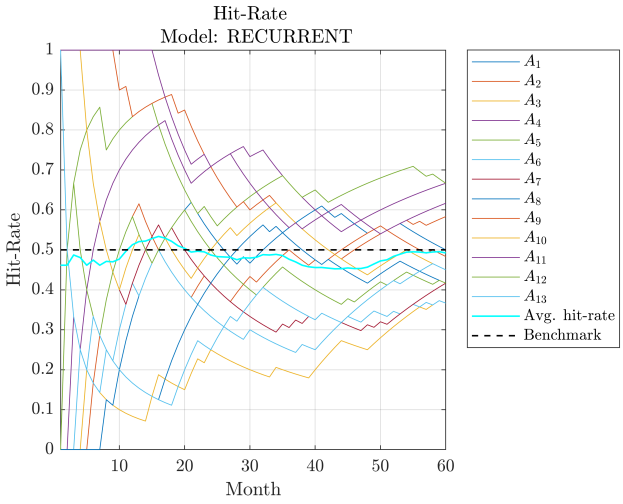


Figure 6.5. *Plot of the average hit rate for each asset using the best recurrent model we managed to train.*

6.3.3 Portfolio Performance

The performance of the portfolio built on the signals yielded from this model is presented in Figure 6.6 and Table 6.10.

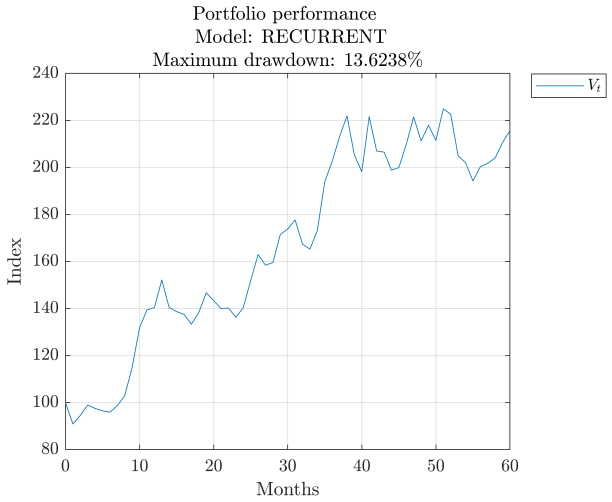


Figure 6.6. *Performance of the portfolio based on the best recurrent model we managed to train.*

Metric	Value
Portfolio value V_{60}	215.6644
Yearly return	16.6153%
Maximum drawdown	13.6238%
$\text{VaR}_{0.05}$	7.4765%
$\text{ES}_{0.05}$	8.2403%
Yearly Sharpe ratio	0.8572
Yearly Sortino ratio	5.2347

Table 6.10. *Performance and risk measures of the portfolio based on the best recurrent model we managed to train.*

6.4 Gated-Recurrent-Unit

In this section, we present the results obtained from the recurrent network. The best GRU we could train is one with a single layer with three nodes with the *tanh* function as the activation function in the output-gate and the logistic function as the activation function in the reset and update gates.

6.4.1 Training the Network

In Table 6.11, the settings for the training of the network is presented.

Training stage	Learning rate	Epochs
Initial training	0.001	750
Incremental training	0.0001	750

Table 6.11. *Settings for the training of the gate-recurrent-unit network*

6.4.2 Prediction Performance

In Table 6.9, the prediction performance of the network is presented. In Figure 6.3, the hit-rate for each asset over time is presented.

Metric	Value
Average hit-rate	49.231%
MSPE	0.00513

Table 6.12. *Prediction performance of the best gate-recurrent-unit network we managed to train*

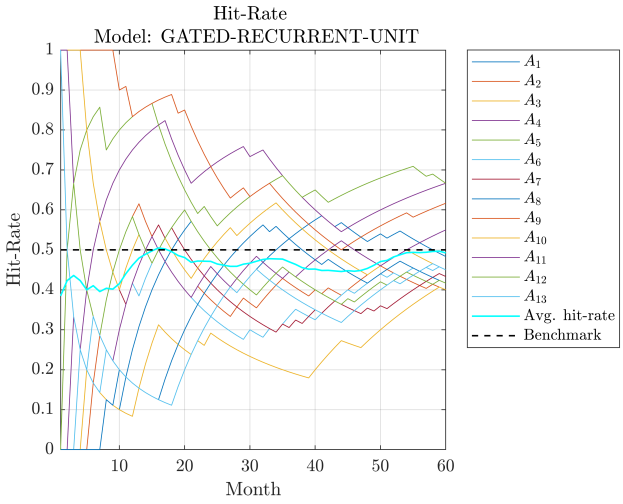


Figure 6.7. *Plot of the average hit rate for each asset using the best gate-recurrent-unit model we managed to train.*

6.4.3 Portfolio Performance

The performance of the portfolio built on the signals yielded from this model is presented in Figure 6.8 and Table 6.13.

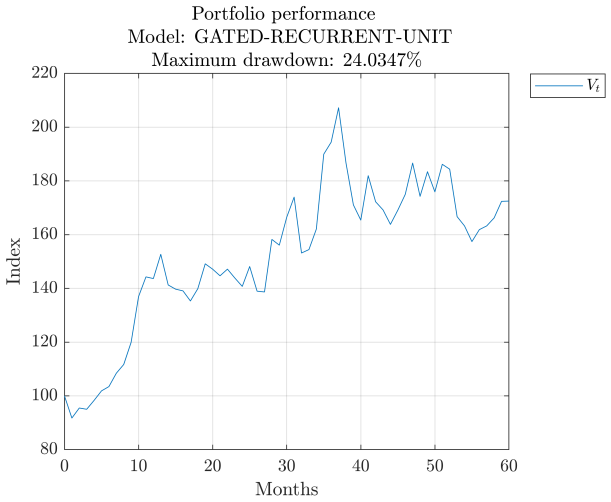


Figure 6.8. *Performance of the portfolio based on the best gate-recurrent-unit model we managed to train.*

Metric	Value
Portfolio value V_{60}	172.5088
Yearly return	11.5224%
Maximum drawdown	24.0347%
$\text{VaR}_{0.05}$	8.3692%
$\text{ES}_{0.05}$	10.4780%
Yearly Sharpe ratio	0.5400
Yearly Sortino ratio	3.1788

Table 6.13. *Performance and risk measures of the portfolio based on the best gate-recurrent-unit model we managed to train.*

6.5 Discussion

Based on the Sharpe and Sortino ratios, the models that yielded the best portfolio performance are the recurrent network models. They outperformed the benchmark regression model, not only in risk adjusted measures, but also in the portfolio value after 60 months as well as the yearly return. The clear worst model, in terms of portfolio performance, is the feed-forward model. We note that the portfolios based on the RNN and GRU networks performed a lot better compared to the portfolio built on the signals from the benchmark regression model.

When computing the correlation between the performances (i.e. the monthly returns) of the different models, a strong correlation is observed between the

regression and the FNN networks, as well as the RNN and GRU networks. The correlations are presented in Table 6.14 below.

Model	**REGRESSION**	**FNN**	**RNN**	**GRU**
REGRESSION	1	0.9516	0.4405	0.2149
FNN	0.9516	1	0.4721	0.2149
RNN	0.4405	0.4721	1	0.8224
GRU	0.2149	0.2149	0.8224	1

Table 6.14. *Correlation of the monthly returns from the different models.*

The correlation matrix in Table 6.14 confirms that there are two clusters of the models: the benchmark regression model and the FNN are in one group, while RNN and GRU is in another.

If we compare the regression model and the FNN, we note that they are only mappings $\mathbf{x} \to y$. RNN and GRU, however, depend on information from previous time-steps as well. Thus, the models grouping together is nothing out of the ordinary considering they are from two categories where one looks at the now and the other looks at history.

We believe that the reason that we observe the strong correlation between the regression model and the FNN model is that the data can be nearly linear, but the FNN is most likely overfitted. If the data is almost linear, then the regression and FNN should give the same fit, in the event that the FNN is not grossly overparametrized. And this phenomenon is also reflected in the performance of the portfolio. We note that the portfolio value at the time horizon of the FNN is 99.0 compared to the regression's 118.6. And this goes hand in hand with the subject of generalization (or prediction power) when it comes to neural networks. The resulting generalization capability of the network is poor because it has more parameters.

As mentioned earlier in the thesis, the regression model is a special case of the FNN. For this reason, it is not surprising they have a similar performance. Furthermore, RNN is a special case of the GRU network, and they perform similarly.

The reason for the disparity in performance between the recurrent network models and the feed-forward models could be due to the information retaining capability of the recurrent models.

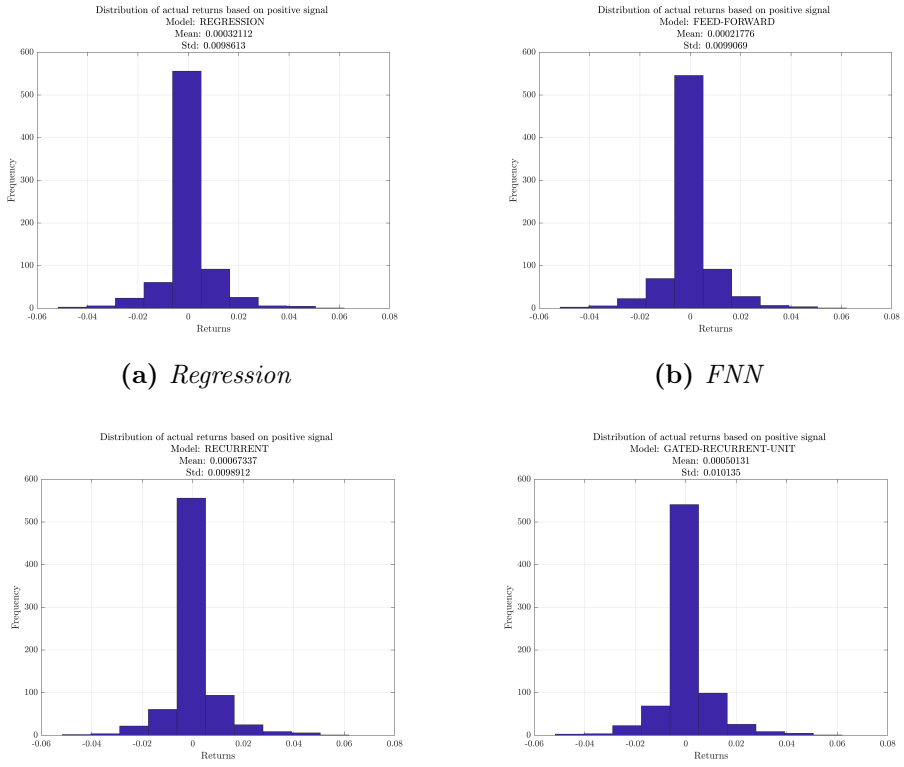
When looking at the selected metrics (average hit-rate and mean squared prediction error) for determining the prediction power of a network, both feed-forward models performed much better compared to their recurrent counterparts. However, the performance of the portfolio was worse for the feed-forward networks. This leads us to down to three possibilities: either

the average hit-rate and the mean squared prediction error are poor measures of prediction power over different model types or they are not suitable for recurrent network model types or the result is just a fluke stemming from a small data sample. The answer to the final hypothesis cannot easily be found without having access to more data. The classical approach of splitting the data into training, validation and test cannot easily be applied without violating the time-series nature of the data.

In order to analyze why the portfolios based on signals predicted using recurrent network models yielded a better performance, we look at the distribution of the actual monthly returns when the models predict positive and negative signals. From there, we compute the expected return and standard deviation for these two subsets. The results are presented in Figures 6.9 and 6.10 and in Table 6.15.

Model	**Regression**	**FNN**	**RNN**	**GRU**
Positive expected return	0.000321	0.000218	0.000673	0.000501
Negative expected return	-0.000389	-0.000286	-0.000745	-0.000572

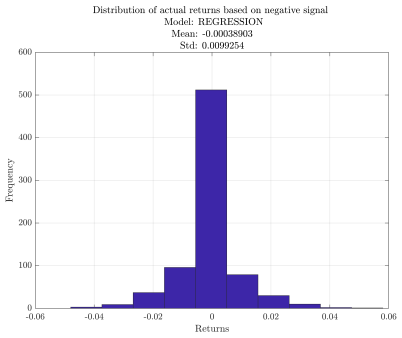
Table 6.15. *Expected return of the distribution of actual monthly returns based on positive predicted signal*



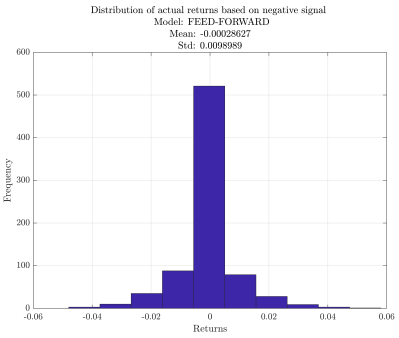
(a) *Regression* **(b)** *FNN*

(c) *RNN* **(d)** *GRU*

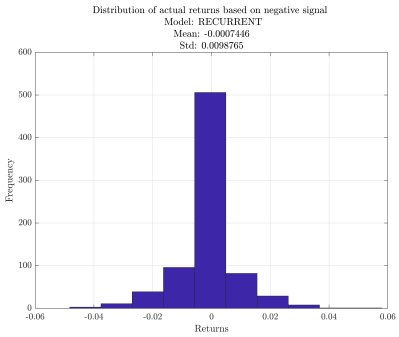
Figure 6.9. *The distribution of actual monthly returns based on positive signals.*



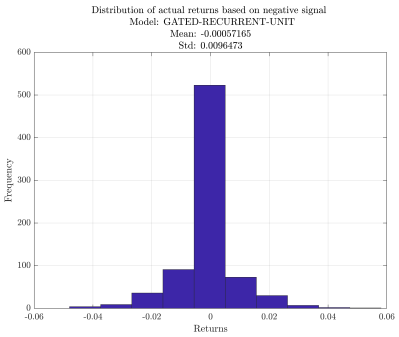
(a) *Regression*



(b) *FNN*



(c) *RNN*



(d) *GRU*

Figure 6.10. *The distribution of actual monthly returns based on negative signals.*

We note from the results presented in Table 6.15 that, even though the recurrent models have a worse hit-rate than the feed-forward models, when the model predicts a positive or negative signal, the expected monthly return has the same sign and is much larger compared to the feed-forward models. This result seems to explain why the portfolios based on signals from the recurrent models perform much better. It also seems to be a possible explanation for why the average hit-rate is not a sufficient measure.

Regarding the mean squared prediction error, an explanation for its insufficiency is seemingly a bit more straight-forward and not model dependent. Suppose that the actual return of an asset is 0% and the model predicts a positive return of 10%, then the portfolio optimization will take a long position in that asset. Compare that scenario to the one where the model predicts 5% positive return, but the actual return is -5%. In both scenarios, the average squared error and the average hit-rate will be the same. However, when trading on those positions, in the one case, the portfolio manager will not lose much since the actual return is 0%. In the other case, the loss will be much more severe since the actual return is -5%.

Chapter 7

Conclusions

In this work, we have tested and selected neural network models from two different network types:

- Feed-forward networks
- Recurrent networks

and compared their performance with a benchmark regression model.

We trained and tested the models on data provided by Aktie-Ansvar AB and predicted future returns. Furthermore, we applied the future returns in a portfolio optimization strategy and obtained results for each model.

Selecting the best model requires a metric that can be used to determine which model is the best. We have looked at a few metrics that we deemed relevant and interesting to scenario in hand:

- The average hit-rate.
- The mean squared prediction error.
- The Sharpe and Sortino ratios.
- The portfolio value.

Following the average hit-rate and mean squared-error, the best model is the benchmark regression model. Following the Sharpe and Sortino ratios as well as the portfolio value after 60 months, the best model is the recurrent neural network with a layer consisting of four hidden nodes with an activation function tanh.

Consequently, it is not easy to determine the best model as this heavily depends on which criteria are used for valuation.

Furthermore there is room to further selecting even better networks out of the model types presented in this thesis. This is because the solution space

is quite large and the problem is non-convex, leading to the existence of multiple local minima.

These results indicate that the neural networks approach to portfolio choice has the potential to surpass classical methods. However, there are still many open questions and plenty of research opportunities.

7.1 Future Work

In this work, we used gradient-descent methods to train the networks. One could potentially investigate the effect of non-convex optimization methods and algorithms.

We also used a limited number of neural network models and configurations. There are quite a few other network types that could be interesting to investigate, among them the LSTM-network.

Furthermore, a more thorough research regarding how to set the hyperparameters is very interesting since the current method is based on trial and error.

Bibliography

[1] N. Baltas, D. Jessop, C. Jones, and H. Zhang, *Global Quantitative Research Monographs Trend-Following meets Risk Parity*, tech. rep., UBS Limited, 2013.

[2] M. Bartholomew-Biggs, *Nonlinear Optimization with Engineering Applications*, Springer Optimization and Its Applications, Springer US, 2008.

[3] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul, *Automatic differentiation in machine learning: a survey*, CoRR, abs/1502.05767 (2015).

[4] J. Bezanson, S. Karpinski, V. Shah, and A. Edelman, *The Julia Language*. https://julialang.org/. Accessed: 2018-05-17.

[5] I. da Silva, D. Spatti, R. Flauzino, L. Liboni, and S. dos Reis Alves, *Artificial Neural Networks: A Practical Course*, Springer International Publishing, 2016.

[6] K. Du and M. Swamy, *Neural Networks and Statistical Learning*, SpringerLink : Bücher, Springer London, 2013.

[7] J. Feldman and R. Rojas, *Neural Networks: A Systematic Introduction*, Springer Berlin Heidelberg, 1996.

[8] A. Graves, *Supervised Sequence Labelling with Recurrent Neural Networks*, Studies in Computational Intelligence, Springer Berlin Heidelberg, 2012.

[9] A. Graves and N. Jaitly, *Towards end-to-end speech recognition with recurrent neural networks*, in Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32, ICML'14, JMLR.org, 2014, pp. II–1764–II–1772.

[10] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*, Other Titles in Applied Mathematics, Society for Industrial and Applied

Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2008.

[11] R. Grosse, *Lecture 15: Exploding and Vanishing Gradients*, University of Toronto Computer Science, 2017.

[12] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, *Memory-efficient backpropagation through time*, CoRR, abs/1606.03401 (2016).

[13] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*, Springer Series in Statistics, Springer New York, 2009.

[14] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, Neural Computation, 9 (1997), pp. 1735–1780.

[15] H. Hult, F. Lindskog, O. Hammarlid, and C. Rehn, *Risk and Portfolio Analysis: Principles and Methods*, Springer Series in Operations Research and Financial Engineering, Springer New York, 2012.

[16] A. Izenman, *Modern Multivariate Statistical Techniques: Regression, Classification, and Manifold Learning*, Springer Texts in Statistics, Springer New York, 2009.

[17] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, CoRR, abs/1412.6980 (2014).

[18] C. Kyunghyun, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, *Learning phrase representations using RNN encoder-decoder for statistical machine translation*, CoRR, abs/1406.1078 (2014).

[19] C. Olah, *Understanding LSTM Networks*. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[20] T. N. Rollinger and S. T. Hoffman, *Sortino:A 'Sharper' Ratio*, tech. rep., Red Rock Capital LLC, 2015.

[21] S. Ruder, *An overview of gradient descent optimization algorithms*, CoRR, abs/1609.04747 (2016).

[22] I. Sutskever, *Training Recurrent Neural Networks*, PhD thesis, University of Toronto, Department of Computer Science, 2013.

[23] K. Yao, T. Cohn, K. Vylomova, K. Duh, and C. Dyer, *Depth-gated Recurrent Neural Networks*, CoRR, abs/1508.03790 (2015).

TRITA -SCI-GRU 2018:233