Forecasting Time Series with Artificial Neural Networks

Master Thesis

P.E. Visscher

Scientific Computing Group
Mathematical Institute
Utrecht University

Supervised by
Prof. dr. R.H. Bisseling

Universiteit Utrecht

July 6, 2018

Abstract

Artificial Neural Networks (ANN) are used as universal approximators and are getting widely adopted in several working fields such as finance. One of the problems that can be addressed using ANN is the forecasting of time series. Time series forecasting is known to be a difficult problem, often requiring expert knowledge, and can be applied to problems including predicting stock value, sales forecasting, and inventory. We explore how the sparsity that occurs in trained ANNs can be used to generalize the network topology to any Directed Acyclic Graph (DAG), instead of running on a layer based architecture. Allowing the ANN to run on any DAG allows it to use the full capabilities of the input, and intermediate values. We show how both the Feed-forward Neural Network (FNN) and Recurrent Neural Network (RNN) topologies can be generalized to this variant, in both training and prediction. Finally we train these network architectures on benchmark problems and use them to forecast time series, where we demonstrate a powerful algorithm to predict the stock market values one day ahead.

Contents

1 Introduction

Time series forecasting is known to be a difficult problem and can be applied on problems such as: predicting stock value, forecasting sales, and managing inventory [Box+16]. In this thesis we will explore the theory and effectiveness of forecasting the future value of company stock in the stock market using Artificial Neural Networks (ANNs). Based on the existing theory of ANNs, we introduce a more generic model, which we will test on both benchmark problems and forecasting of time series. First we start with an introduction of what time series are, followed by an overview of popular time series forecasting methods. By doing so, we give the historical context of time series prediction. Finally, in Section 1.2, we describe why it is hard to use those existing methods in practice, and why ANNs do not suffer from the same problems.

Time series are a data type, indexed and ordered in time. A time series is either continuous or discrete. In this work we only consider discrete time series, with a fixed time interval h. We define $J = [1, \ldots, n]$ as the collection of all indices for n discrete time points. The discrete ordered set of time points T can be concisely defined as $T = h_0 + hJ$, where h_0 is the time offset of the time series, and h is the fixed time interval, with $\tau_t = h_0 + ht \in T$ the time corresponding to index $t \in J$. We can write the discrete time series as a function $\varkappa : T \to \mathbb{R}^m$ of time τ_t, but we simplify this notation to $z_t = \varkappa(\tau_t)$. When $m = 1$ we have a one-dimensional observation at each time step; we call this a univariate time series. For $m > 1$ each observation is a vector z_t; in this case we assume a multivariate process. For a univariate time series we use the vector $\boldsymbol{z}$ and for a multivariate time series we use the matrix Z.

Traditionally, a single time series is decomposed into multiple time series each describing a certain characteristic. We can decompose a time series z_t at time index t with:

- T_t, a trend component which describes the increasing or decreasing trend in the time series. Note that this component may be non-linear.
- S_t, describes the seasonality component, which is a periodic function.
- N_t, describes the noise component, which is the remainder of the

time series, after the other components have been accounted for.

These components can be modelled both additively and multiplicatively: $z_t = T_t + S_t + N_t$ and $\zeta_t = T_t \cdot S_t \cdot N_t$ respectively [Box+16]. Note that the multiplicative model may be described by the additive model with a logarithmic transformation

$$\begin{aligned} \ln \zeta_t &= \ln(T_t \cdot S_t \cdot N_t) \\ &= \ln T_t + \ln S_t + \ln N_t \\ &= \tilde{z}_t. \end{aligned}$$

This means that the additive model is generic enough, since a multiplicative representation can easily be transformed to it.

By using the characteristics of each of the components of the decomposition, we can model them separately, after which we can recombine them at a later stage. For example, the trend may be modelled with a linear regression, while the seasonality can be fit using a Fourier series. These statistical approximations are deterministic and we assume that the characteristics hold over time, which makes them less flexible than methods that do not make these assumptions [Har90].

1.1 Statistical Methods

Historically, time series forecasts were made using statistical methods, where early on methods like OLS were used. In 1927 Yule [Yul27] and in 1931 Walker [Wal31] defined the Autoregressive (AR) model approach. An improved version of this model called Autoregressive Moving-Average (ARMA) was popularised by Box and Jenkins [BJ70]. Originating from the Box-Jenkins method, the Generalized Autoregressive Conditional Heteroscedasticity (GARCH) method was defined by Bollerslev [Bol86].

Ordinary Least Squares

A simple method to forecast time series is by applying the OLS algorithm. We assume that our time series can be modelled as

$$\mathbf{z} = X\beta + \epsilon, \tag{1.1}$$

where $X \in \mathbb{R}^{n \times m}$ with full column rank m contains the exogenous variables, $\beta \in \mathbb{R}^m$ contains the model parameters, $\epsilon \in \mathbb{R}^n$ contains uncorrelated random errors with $\mathbb{E}[\epsilon] = 0$ and variance σ^2, and $\mathbf{z} \in \mathbb{R}^n$.

To use this model we have to fit it to our data, from which we will get the unbiased estimates $\hat{\beta}$ for the parameters β. In our fit we want to minimise the sum of squared residuals

$$\begin{aligned} S(\beta) &= \langle \epsilon, \epsilon \rangle \\ &= \|\mathbf{z} - X\beta\|_2^2. \end{aligned}$$

This can be done by letting $\hat{\beta}$ be our estimator such that

$$\hat{\beta} = (X^T X)^{-1} X^T \mathbf{z}, \tag{1.2}$$

see the proof in Appendix A. In practice we fit the $\hat{\beta}$ parameters using a linear system solver. For example let our model, where τ_t describes the training data, be given by

$$\mathbf{z}_t = \beta_1 + \beta_2 \tau_t + \epsilon_t$$

or in matrix notation

$$\mathbf{z} = \begin{pmatrix} 1 & \tau_1 \\ \vdots & \vdots \\ 1 & \tau_n \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} + \epsilon.$$

After fitting this model to the sample data, with the OLS estimator, we get parameters β_1 and β_2. From the estimator $\hat{\beta}$ we can predict values at other time points. These new time points are defined as $\hat{\tau} \in \mathbb{R}^k$. By choosing our own X matrix for the OLS model that incorporates these $\hat{\tau}$ values

$$X = \begin{pmatrix} 1 & \hat{\tau}_1 \\ \vdots & \vdots \\ 1 & \hat{\tau}_k \end{pmatrix},$$

we can make predictions by applying the $\hat{\beta}$ estimator. While OLS works very well to describe a linear trend, it fails to define the seasonality of a time series, since all this information will be hidden in the ϵ vector.

Polynomial Regression

We can modify the OLS approach to fit polynomial models. Again we assume that our time series can be modelled as

$$\mathbf{z} = X\beta + \epsilon.$$

While all parameters function the same as in normal OLS, we only have to redefine our X matrix. Let m be the dimension of the polynomial we want to fit to our data. For this m we let X be the Vandermonde matrix

$$X = \begin{pmatrix} 1 & \tau_1 & \tau_1^2 & \dots & \tau_1^m \\ 1 & \tau_2 & \tau_2^2 & \dots & \tau_2^m \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ 1 & \dots & \dots & \dots & \tau_n^m \end{pmatrix},$$

wherein we have $X_{i,j} = \tau_i^{j-1}$ for each element. The fitting method described in the OLS algorithm can be used here too. We extrapolate this polynomial by ensuring the matrix is defined as a Vandermonde

matrix. To properly make a regression using this model we need to know the dimension m of the polynomial beforehand. As demonstrated by Anscombe [Ans73] both over-fitting, and under-fitting are a real problem with model dimensionality.

Definition 1 (Strict Stationarity).

Let $J = [1, \ldots, n]$ be the index set for n time points. A time series $\mathbf{z} \in \mathbb{R}^n$ is strictly stationary if for all $t, k, k + t \in J$ the probability distribution for $z_1, \ldots, z_t$ is the same as for $z_{1+k}, \ldots, z_{t+k}$.

Definition 2 (Covariance).

Let $J = [1, \ldots, n]$ be the index set for n time points and $\mathbf{z} \in \mathbb{R}^n$ be a time series. The covariance function is defined as

$$\mathrm{Cov}(z_t, z_s) = \mathbb{E}\left[(z_t - \mathbb{E}\left[z_t\right])(z_s - \mathbb{E}\left[z_s\right])\right],$$

for all $t, s \in J$.

Definition 3 (Weak Stationarity).

Let $J = [1, \ldots, n]$ be the index set for n time points. A time series $\mathbf{z} \in \mathbb{R}^n$ is weakly stationary if a μ and σ exists such that

1. $\mathbb{E}\left[z_t\right] = \mu$ for all $t \in J$.
2. $\mathbb{E}\left[(z_t - \mu)^2\right] = \sigma^2 < \infty$ for all $t \in J$.
3. For all $k \in J$ there exists γ_k such that for all $t, t + k \in J$ it holds that $\mathrm{Cov}(z_t, z_{t+k}) = \gamma_k$.

Definition 4 (Discrete White Noise).

Let $J = [1, \ldots, n]$ be the index set for n time points. Discrete white noise $\{\epsilon_t, t \in J\}$ is identically and independently distributed (i.i.d) and there exists a σ such that

1. $\mathbb{E}\left[\epsilon_t\right] = 0$.
2. $\mathbb{E}\left[\epsilon_t^2\right] = \sigma^2$.
3. $\mathbb{E}\left[\epsilon_t \epsilon_s\right] = 0$, for $t \neq s$, and $t, s \in J$.

General Linear Processes

Wold showed in 1938 [Wol38] that every weakly stationary process with $\mathbb{E}[z_t] = 0$ can be written as a general linear process

$$z_t = \epsilon_t + \sum_{i=1}^{\infty} \psi_i \epsilon_{t-i}, \tag{1.3}$$

where ϵ_t is discrete white noise, and $\boldsymbol{\psi} = (\psi_1, \psi_2, \dots)$ are weights. Koopmans [Koo74, p. 254] showed that under certain conditions (such as spectral density boundedness of the white noise process) this can be rewritten as

$$z_t = \epsilon_t + \sum_{i=1}^{\infty} \pi_i z_{t-i}, \tag{1.4}$$

where again ϵ_t is discrete white noise, and $\pi_1, \pi_2, \dots$ are weights. The weights π_i and ψ_i for all i are related by an explicit bijection. The number of weights needed for both these models are infinite, and thus the models in these forms are not practical. Under the assumption that only a finite number of weights are non-zero we can derive new models.

Moving-average Model

The Moving-Average (MA) process uses a weighted sum of the previous q discrete white noise data points, or MA(q) in short. This is a special case of Equation (1.3) wherein only the first q weights of $\boldsymbol{\psi}$ are non-zero. The model is defined as

$$z_t = \mu + \epsilon_t - \sum_{i=1}^{q} \theta_i \epsilon_{t-i}, \tag{1.5}$$

where ϵ_t is discrete white noise, $\theta_1, \dots, \theta_q$ are weights, and μ is a translation constant (and in this case the mean). To fit this model to a time series we need to find the values for μ, $\theta_1, \dots, \theta_q$ and the σ^2 for the discrete white noise ϵ_t, thus we have $q + 2$ unknown parameters. A MA process is strictly stationary, just as the general linear process.

Autoregressive Model

In an AR model each element in the time series is a linear combination of previous elements, contrary to the weighted sum of discrete white noise data points MA uses. This is a special case of Equation (1.4), wherein only the first p weights are non-zero. The AR model of order p, AR(p), is defined as

$$z_t = \mu + \epsilon_t + \sum_{j=1}^{p} \phi_j z_{t-j}, \tag{1.6}$$

where $\phi_1, \dots, \phi_p$ are the model parameters, ϵ_i is discrete white noise, and μ is a translation constant. Again to fit this model we need to find the parameters μ, $\phi_1, \dots, \phi_p$ and the σ^2 of the discrete white noise ϵ_t, which are $p + 2$ unknown parameters. Unlike the MA model, the AR model is not necessarily weakly stationary. The process is only stationary when the

roots of the characteristic polynomial defined by the model fall outside the unit circle [AA13].

Autoregressive-moving-average

The AR and MA models can be improved by combining them. This new process is called ARMA, ARMA(p, q), and is given by

$$z_t = \mu + \epsilon_t - \sum_{i=1}^{q} \theta_i \epsilon_{t-i} + \sum_{j=1}^{p} \phi_j z_{t-j}, \tag{1.7}$$

where the parameters are the terms combined from the AR and MA models. We now have $p + q + 2$ unknown parameters. Since ARMA assumes that our model is stationary, an extension called Autoregressive Integrated Moving-Average (ARIMA) was developed which also works for non-stationary time series. ARIMA integrates the time series d times by differencing the values of a non-stationary time series to make it weakly stationary. ARMA and ARIMA can be extended to perform better on data with seasonal components, called Seasonal Autoregressive Moving-Average (SARMA) and Seasonal Autoregressive Integrated Moving-Average (SARIMA) respectively. These models include a differencing with a lag s, which is the seasonality period, to account for the periodic terms. Another generalisation of the ARMA model, named GARCH can be made. GARCH assumes that the discrete white noise variance in the model follows an ARMA model, whereas ARIMA assumes that this error variance is constant [Bol86]. AR, MA, ARMA, ARIMA, and GARCH are univariate models, but in practice we often want to forecast multivariate processes. For this purpose Vector Autoregression (VAR), Vector Autoregressive Moving-Average (VARMA), Vector Autoregressive Integrated Moving-Average (VARIMA), and Multivariate Generalized Autoregressive Conditional Heteroscedasticity (MGARCH) were developed. For every time series a new number of parameters has to be determined, and usually each time series has many parameters. This is the drawback of these type of processes, since parameter estimation for many time series may get infeasible due to the growing number of parameters needed [DP02].

1.2 Transition to Artificial Intelligence

The traditional methods presented here often require hand-crafted features, and expert knowledge of the field [Gam17]. They often require either strict-, or weak-stationarity in the data, which in practice often does not hold. This means that before we can apply traditional methods, we need to transform our data with techniques such as detrending algorithms, which introduce their own class of problems. It should also be noted that the ARIMA class of models only performs well on linear processes.

ANNs do not suffer from these problems since they are non-linear in nature and data driven. ANNs often outperform ARIMA models [Koh+96; KOW04; CL11] and even hybrids of ANNs and ARIMA models have been used [Día+08] to produce better results than either model alone. Although ANNs have been around since the 1950s and 1960s, only recently the cost of collecting data has become cheaper than analyzing it due to the advent of cloud computing and cheaper and larger storage hardware. The massive data collection that followed from it sparked renewed interest in data driven algorithms, of which ANNs is an example. ANNs learn to model processes based on example input and output values, and performance gets better when more data is used in the training of the models. This shifts the required expert knowledge, that was required with techniques such as ARIMA, from the working field of the data to knowledge of the algorithm. This inherently makes it cheaper and easier to make meaningful predictions from time series data.

2 Artificial Neural Networks

In this chapter we introduce the concepts of ANNs, and concisely discuss how they have evolved. We give the theory on two of the most common ANN architectures; the Feedforward Neural Network (FNN) and the Recurrent Neural Network (RNN). An ANN is a method for approximating an unknown function $y = f(x)$ that maps the input data $\mathbf{x}$ to the output data $\mathbf{y}$. An ANN consists of a directed network of perceptrons each of which itself maps an input data space to an output space. A perceptron is defined by a set of parameters and an activation function. These parameters can be 'learned' or 'trained' by solving an optimisation problem for a training set $\{x_i\}$ for which the output $\{y_i\}$ is known. The details of how the networks are trained are left to later chapters, see Chapter 4 and Chapter 5, where the learning algorithms are described in detail.

Single-layer Perceptron

The simplest form of ANN is the Single-layer perceptron. Rosenblatt developed these kind of perceptrons through the 1950s and the 1960s [Ros61]. Let $\mathbf{x} \in \mathbb{R}^n$ be the input vector, $\mathbf{w} \in \mathbb{R}^n$ the weights, and $b \in \mathbb{R}$ the bias of the perceptron. Now let the output of the perceptron be defined by thefollowing function

$$f(\mathbf{x}, \mathbf{w}, b) = \begin{cases} 0 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b \leq 0, \\ 1 & \text{if } \langle \mathbf{w}, \mathbf{x} \rangle + b > 0. \end{cases} \tag{2.1}$$

This perceptron instantly flips its value from 0 to 1 when $x \approx 0$. By making this function continuous we can apply algorithms that use the derivative to train the weights $\mathbf{w}$ and bias b. Initially this continuous perceptron was made by using the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.2}$$

The sigmoid here is called the activation function. The sigmoid perceptron, also called a sigmoid neuron, is now defined as

$$f(\mathbf{x}, \mathbf{w}, b) = \sigma(\langle \mathbf{x}, \mathbf{w} \rangle + b) = \frac{1}{1 + e^{-\langle \mathbf{w}, \mathbf{x} \rangle - b}}. \tag{2.3}$$

Feedforward Neural Network

The Single-layer perceptron can be extended to the multi-layer case, often called Multi Layer Perceptron (MLP), or Feedforward Neural Network (FNN), wherein we have multiple layers each consisting of multiple neurons. Information only flows in one direction in a FNN, and thus FNNs are acyclic. In this type of ANN we have the input layer, vector $\mathbf{x}$, middle layers, and finally an output layer. The layers in the middle are called hidden layers, as they are neither part of the input or output. Each hidden layer is fully connected with the previous layer, and the output layer is fully connected with the last hidden layer. Each hidden layer and output layer has a fixed number of neurons, which is a configurable parameter of the model. The goal of this network is to approximate a function f^*.

In an FNN, each neuron is connected to all the neurons in the next layer, which can be seen as a complete bipartite graph, or biclique. Each connection in this graph has its own weight, and thus we have a weighted biclique. Let d_l be the dimension of layer l, and $W_l \in \mathbb{R}^{d_{l+1} \times d_l}$ be the weight matrix where each element $(W_l)_{i,j}$ is the weight between neuron i on layer l and neuron j on layer $l+1$. Now let $\mathbf{b}_l \in R^{d_{l+1}}$ be the bias vector of layer l with each element $(\mathbf{b}_l)_i$ the bias of neuron i. Now the output $\mathbf{y}$ of the FNN, with hidden layers $l \in [1, \ldots, n-1]$ is defined as

$$\mathbf{x}_{l+1} = g_l(W_l \mathbf{x}_l + \mathbf{b}_l), \tag{2.4}$$

$$\mathbf{y} = \omega(\mathbf{x}_{n-1}) \tag{2.5}$$

where $\mathbf{x}_0$ is the input vector, $\mathbf{y}$ is the output vector, $g_l : \mathbb{R}^{d_l} \to \mathbb{R}^{d_l}$ is the activation function of layer l, and $\omega : \mathbb{R}^{d_n} \to \mathbb{R}^{d_n}$ the output function. An example of a FNN can be seen in Figure 2.1

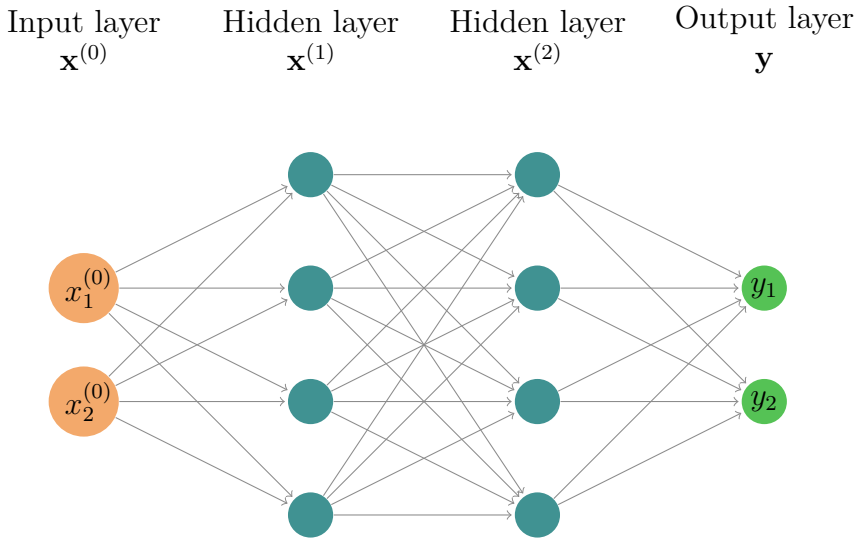


Figure 2.1: An example FNN with two 4-dimensional hidden layers, an input vector and output vector both of dimension 2.

Activation Functions

ANNs initially used the sigmoid function as activation function, see Equation (2.2), but practice shifted fast to the use of the hyperbolic tangent activation function $g(x) = \tanh(x) = 2\sigma(2x) - 1$ due to its stronger gradients around $x = 0$, bias avoidance and faster convergence in training [Hay04; LKS91]. The Rectified Linear Unit (ReLU) activation function $g(x) = \max(0, x)$ was popularised by Krizhevsky in [KSH12], where a convergence improvement of approximately 6 times over hyperbolic tangent activation functions was shown. ReLU has a large domain for which the neuron output has value 0, which means that gradient based learning methods cannot get information from these neurons. Using ReLU, neurons can irreversibly die by failing to get to a value wherein the neuron gets activated again. The ReLU activation function is generalized by redefining it as $g(x, \alpha_i) = \max(0, x) + \alpha_i \min(0, \alpha_i)$, where i refers to node i. The Leaky ReLU sets $\alpha_i \approx 0.01$ which reintroduces a small gradient [MHN13]. Another form makes α_i a trainable parameter, which results in the Parametric ReLu (PReLU) [He+15].

Theorem 1 (Kolmogorov's Theorem [Kol63]).

Any multivariate continuous function $f : \mathbb{R}^n \to \mathbb{R}$ can be represented as a superposition of one-dimensional functions such that

$$f(\mathbf{x}) = \sum_{j=0}^{2n} \phi_j \left(\sum_{i=1}^{n} \psi_{j,i}(x_i) \right), \tag{2.6}$$

where both $\phi_j : \mathbb{R} \to \mathbb{R}$ and $\psi_{j,i} : \mathbb{R} \to \mathbb{R}$ are continuous. While Kolmogorov's proof was inconstructive, Braun and Griebel finally proved this theorem in 2009 [BG09].

Theorem 2 (Universal Approximation Theorem [Csá01; Cyb89]).

Let ϕ be an arbitrary activation function. We let X be a compact set, defined as $X \subseteq \mathbb{R}^n$. The space of continuous functions on X is denoted by $C(X)$. Then given any $\epsilon > 0$ for all $f \in C(X)$, there exists a $m \in \mathbb{N}$, and $u_i, b_i \in \mathbb{R}$, and $\mathbf{w}_i \in \mathbb{R}^m$ where $i \in [1, \ldots, m]$ such that

$$F(x) = \sum_{i=1}^{m} u_i \phi(\langle \mathbf{w}_i, \mathbf{x} \rangle + b_i),$$

and $\|F(\mathbf{x}) - f(\mathbf{x})\| < \epsilon$, for all $\mathbf{x} \in X$.

Universal Approximator

Kolmogorov showed in 1957 [Kol63] by proving Theorem 1 that any multivariate continuous function $f : \mathbb{R}^n \to \mathbb{R}$ can be represented by $O(n^2)$ continuous one-dimensional functions. By building on this theorem, Theorem 2 proved that a single-layer FNN can approximate any

continuous function on a compact set. This theorem was extended to the multi-layer case [HSW89] by Hornik, Stinchcombe and White, and it showed the existence of a FNN for every compact continuous function in $\mathbb{R}^m$. A remarkable corollary of this theorem is that the working of the FNN is not dependent on the activation function ϕ, and that the number of neurons needed is finite. Unfortunately neither Theorem 1 nor Theorem 2 give direction on the learnability of the needed parameters. Both shallow and deep (a network with more than one layer) FNNs are universal approximators by Theorem 2, but deep ANNs are conjectured to have a significantly greater representational power than a shallow ANN. This conjecture is supported by recent research [MLP16; CSS16], that shows that deep networks require fewer parameters to represent the same function than shallow networks.

Recurrent Neural Network

In theory, a FNN due to Theorem 2 is capable of approximating every compact continuous function. In practice we need different architectures of the networks to effectively learn the needed capabilities. One of such architectures that is of particular interest for predicting time series is the RNN [WH86]. While the FNN architecture has to represent time explicitly with a fixed-length window, the RNN does this by representing time recursively [Mik+14]. This makes the RNN architecture particularly suitable to sequential data, since it is able to learn long-term dependencies in the data better. In fact it can be shown that for any computable function there exists a finite RNN that is able to compute it. Furthermore, there also exist finite sets of RNNs that are Turing complete and thus they can be used to implement all algorithms [SS95].

The idea in a RNN architecture is to let each time step share the same progressing internal state, and let the neural network have the data of not only the current time step, but also the previous. This can be, but is not limited to, a shared connection between hidden units, the output from a previous time step to the hidden units of the current time step. Or more generally, given the forward-propagation function f, parameters θ, the state $h_{(i)}$, and input $\mathbf{x}_{(i)}$ at time step i, a RNN often uses a representation in the form

$$\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, x_{(t)}, \theta). \tag{2.7}$$

RNNs are notoriously harder to train than FNNs with gradient based learning methods, which is explained by Theorem 3 [PMB13]. Improved versions of the RNN such as the Long Short-Term Memory (LSTM) networks [HS97] and Gated Recurrent Units (GRU) networks, were developed specifically to prevent the gradients from exploding or vanishing [Chu+14].

Theorem 3 (Exploding and vanishing gradients [PMB13]).

Let $\sigma : \mathbb{R}^d \mapsto \mathbb{R}^d$ be an activation function with a bounded derivative, and

$$\gamma = \sup\{\|\sigma'(\mathbf{x}_k)\|_2 , \mathbf{x}_k \in \mathbb{R}^d\}.$$

Let $t - k$ be the difference in time steps from time t. Now when the largest eigenvalue λ_1 of the recurrent weight matrix $\mathbf{R}$ satisfies $\lambda_1 < \gamma$, the gradient of the long term contributions converges to 0 exponentially fast with $O(\exp(t - k))$. When $\lambda_1 > \gamma$, the gradient of the long term contributions keeps increasing exponentially fast with $O(\exp(t - k))$.

Sparsity

Now that we have introduced the most common ANN architectures, we will try to create a generic architecture from these. A basis for this lies in the work of Bourely [BBC17] where the implementation of ANNs as sparse multipartite graphs was shown. We can take this form further by representing ANNs as Directed Acyclic Graphs (DAGs) without layers. A foundation of this idea was laid out by van der Lugt [Lug17], wherein the ANN was reprented as a graph and the computations were done using GraphBLAS. Another development in ANNs was made by applying a technique called skip connections [Lin+96]. This method adds direct connections between layers that are not consecutive, and are implemented using layers with identity activation functions. Skip connections were used by a Microsoft Research team that won the ImageNet 2015 competition [He+16].

In the next few chapters, we combine the idea of sparseness and skip connections to train the model and predict stock market time series. An ANN in this form will be able to learn its own topology, whereas the traditional layered ANN still requires expert knowledge of the machine learning field to define an effective architecture.

3 ANN as a Computational Graph

Traditionally, all ANNs are built with neurons separated by layers. This limits the topology of the ANN to only make connections directly between preceding and succeeding layers. Instead, we will allow the ANNs to operate any DAG, which removes this limitation. In this chapter we explain how the traditional architectures predict the output given an input, and how we can extend this to run on a DAG. Finally, we explain how we can obtain the gradient of an objective function with respect to the ANN parameters. This method is the basis for many learning algorithms, and will be used in Chapter 4 and Chapter 5 for the training of our more generic models.

3.1 Layered Architectures

Before we explain how we generalize the structures of both the FNN and RNN, we first have to understand how they calculate their output given an input vector. The FNN algorithm can be reduced to a simple recurrence relation. Let $\mathbf{x}_0$ be the input vector. In an FNN we have multiple layers. For every layer l we have a weight matrix $\mathbf{W}_l$ and a bias vector $\mathbf{b}_l$. By applying the element wise activation function g_l we define the output of the FNN as following recurrence relation

$$\mathbf{x}_{l+1} = g_l(W_l\mathbf{x}_l + \mathbf{b}_l), \tag{3.1}$$

where $\mathbf{x}_l$ the output of the previous layer. The more general topology of ANNs introduced by the RNN type of networks has connections that feed their input to the next calculation in the form of loop connections, as seen in Equation (3.2) which defines a RNN with a single layer. The RNN works on sequences with clearly defined time steps in the input.

$$\mathbf{h}_t = g(\mathbf{L}\mathbf{x}_t + \mathbf{R}\mathbf{h}_{t-1} + \mathbf{b}), \tag{3.2}$$

where $\mathbf{h}_t$ is the state, $\mathbf{x}_t$ is the input on time step t, $\mathbf{b}$ the bias vector, $\mathbf{R}$ the recurrent weight matrix, $\mathbf{L}$ the input weight matrix, and g the element-wise activation function [PMB13]. This allows the network to model the

relation between time steps, whereas the FNN does not explicitly learn this relation. This difference allows the RNN to use any sequence length, while the FNN has to be trained for a specific length.

Back-propagation

Most ANNs are trained using the back-propagation algorithm, that calculates the gradient of an objective function with respect to its parameters. The forward-propagation algorithm is the process of iteratively applying Equation (2.4) and Equation (2.5). From this we get the output $\hat{\mathbf{y}}$, which approximates the true output $\mathbf{y}$. The back-propagation algorithm was popularised by Rumelhart, Hinton and Williams in 1986 [RHW+88]. Back-propagation computes the gradient of an objective function $J(\theta, \hat{\mathbf{y}}, \mathbf{y})$ where θ are the learnable parameters. It works by propagating the errors in each layer back to the previous layer starting with the output. Given pairs of input and output $(\mathbf{x}, \mathbf{y})$ as data sample, and with Equation (2.4) and Equation (2.5), we can apply the chain rule iteratively for $i \in [1, \dots, l-1]$

$$\delta_0 = \nabla_{\hat{\mathbf{y}}} J(\theta, \hat{\mathbf{y}}, \mathbf{y}), \tag{3.3}$$

$$\delta_i = (W_{l-i})^T \left(\delta_{i-1} \odot g'_{l-i} \left(W_{l-i}\mathbf{x}_{l-i} + \mathbf{b}_{l-i}\right)\right), \tag{3.4}$$

$$\delta_l = \nabla_\theta J(\theta, \hat{\mathbf{y}}, \mathbf{y}), \tag{3.5}$$

where $\odot$ is the element-wise (Hadamard) product. By using gradient optimization methods such as Stochastic Gradient Descent (SGD), we can modify the parameters to find an optimal value of the objective function J. The naive back-propagation algorithm with n nodes runs in $O(n^2)$ operations. However, this formulation only works on networks without loop connections, and cannot be used directly on RNNs.

A common approach to obtain the gradients of RNNs is by using Back-propagation Through Time (BPTT), which is a modification of the back-propagation algorithm. BPTT allows the RNN to be defined as a DAG by using a process that is called unrolling. The unrolling process replicates the connections of the input, output and hidden units for an arbitrary number of time steps t, and forms a new graph by using these replicas as layers. This is done by substituting the recurrence relation

$$\mathbf{h}_t = g(\mathbf{L}\mathbf{x}_t + \mathbf{R}\mathbf{h}_{t-1} + \mathbf{b}), \tag{3.6}$$

directly with the previous computations, which gives us the following

$$\mathbf{h}_t = g(\mathbf{L}\mathbf{x}_t + \mathbf{R}g(\mathbf{L}\mathbf{x}_{t-1} + \mathbf{R}\,(\dots) + \mathbf{b}) + \mathbf{b}). \tag{3.7}$$

After unrolling the graph we are left with a DAG, which is trainable like a deep FNN [LBH15]. BPTT unrolls the network in the back-propagation algorithm, and uses the network as a normal FNN during the back-propagation. This algorithm allows us to simply calculate the gradients, and lets us use those gradients as if they were from a FNN. A visual example of the unrolling process can be seen in Figure 3.1.

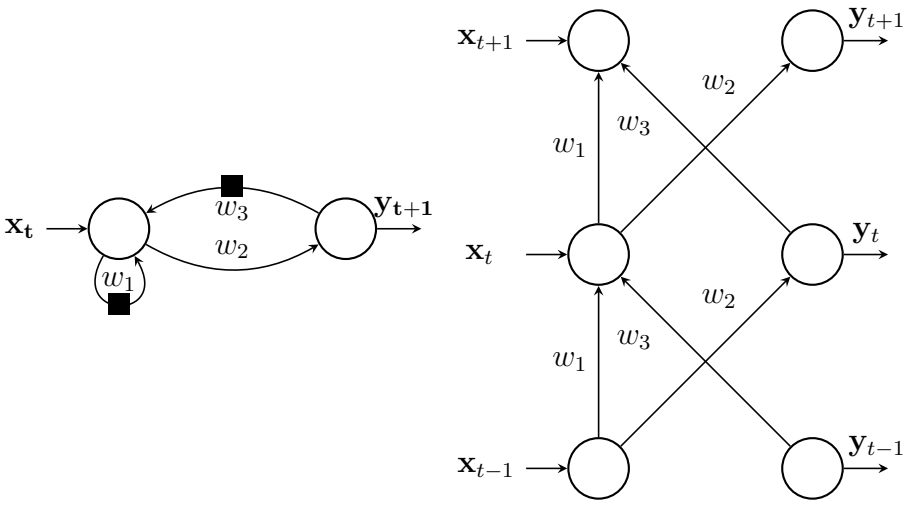


Figure 3.1: The input x_t at time step t propagates with weight w_2 directly to output y_{t+1}. However the first node is also connected to the previous output of that node by weight w_1, and previous output of the second node by weight w_3. (*Left*) An example of a RNN graph wherein the black squares indicate a delay of a single time step. (*Right*) The same RNN unrolled to include time steps $t-1, t, t+1$. The resulting graph is a DAG and can be trained by back-propagation, since all loops are now removed from the graph.

Definition 5 (Weighted Adjacency Matrix).

Let $G = \{V, E, w\}$ be a weighted and directed graph, possibly with self-loops, with n vertices in the vertex set V, where $v_1, \ldots, v_n \in V$, the edge set is E, and the weight function $w : E \mapsto \mathbb{R}\backslash\{0\}$ gives every edge $e \in E$ a weight. We use the notation $e_{i,j}$ for the edge $(v_i, v_j) \in E$, where the edge links node v_j to node v_i. We define the weighted adjacency matrix of the graph G as $\mathbf{A} \in \mathbb{R}^{n \times n}$ with

$$\mathbf{A}_{i,j} = \begin{cases} w(e_{i,j}) \text{ if } e_{i,j} \in E, \\ 0 \text{ otherwise .} \end{cases}$$

3.2 Lagrange Multipliers

Since we do not use the standard layer based topology, we need to define the back-propagation algorithm specifically for the topologies that run on general DAGs. We let the network have an input size s and an output size d and n hidden nodes. The first s nodes in the network are input nodes that have the identity function as activation function. We first

define the block that connects the s input nodes to the n hidden nodes as

$$\mathbf{I} = \begin{pmatrix} i_{1,1} & \cdots & i_{1,s} \\ \vdots & \ddots & \vdots \\ i_{n,1} & \cdots & i_{n,s} \end{pmatrix}.$$

The connections of hidden nodes are represented in a DAG, and input nodes are ordered according to partial DAG ordering, thus only allowing connections to previous nodes in this ordering. This results in a strictly lower triangular weighted adjacency matrix defined as

$$\mathbf{W} = \begin{pmatrix} 0 & 0 & \cdots & 0 & 0 \\ w_{2,1} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & & \\ w_{n-1,1} & w_{n-1,2} & \cdots & 0 & 0 \\ w_{n,1} & w_{n,2} & \cdots & w_{n,n-1} & 0 \end{pmatrix}.$$

Finally the connections between the hidden and output nodes are defined by the output matrix

$$\mathbf{O} = \begin{pmatrix} o_{1,1} & \cdots & o_{1,n} \\ \vdots & \ddots & \vdots \\ o_{d,1} & \cdots & o_{d,n}. \end{pmatrix}$$

Now let $\mathbf{L} \in \mathbb{R}^{(n+s+d)\times(n+s+d)}$ be the strictly lower triangular weighted adjacency matrix of a directed network graph, see Definition 5, describing n nodes. We define $\mathbf{L}$ as the following block matrix

$$\mathbf{L} = \begin{bmatrix} 0 & 0 & 0 \\ \mathbf{I} & \mathbf{W} & 0 \\ 0 & \mathbf{O} & 0 \end{bmatrix}.$$

In practice we do not need to store the top s rows, since these will always be zero. With this adjacency graph we propagate the values of the network with a new formulation

$$\begin{aligned} z_i &= \langle \mathbf{L}_{i,*}, \xi \rangle + b_i, \\ \xi_i &= g_i(z_i), \end{aligned}$$

where i is now indicating a node instead of a layer.

Similarly, we define the recurrent weights $\mathbf{S} \in \mathbb{R}^{n\times(n+s)}$ as the weighted adjacency matrix that connects the previous state to the current state. Now let $\mathbf{R} \in \mathbb{R}^{(n+s+d)\times(n+s+d)}$ be the weighted adjacency block matrix

$$\mathbf{R} = \begin{bmatrix} 0 & 0 \\ \mathbf{S} & 0 \end{bmatrix}.$$

The matrix $\mathbf{R}$ is allowed to connect the input, and hidden nodes of the previous time step of the recurrence relation to the current hidden nodes.

This allows us to extend the previous notation with a recurrence relation at time t (denoted with superscript (t)) for node i as

$$z_i^{(t)} = \langle \mathbf{L}_{i,*}, \xi^{(t)} \rangle + \langle \mathbf{R}_{i,*}, \xi^{(t-1)} \rangle + b_i,$$
$$\xi_i^{(t)} = g_i(z_i^{(t)}).$$

We formalize the computational graph as a DAG, where each node represents a variable (scalar, vector or matrix), which are connected through edges that represent operations (functions) [GBC16]. When we work with FNNs, we define $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ to be the sample set of m tuples with input $x_i \in \mathbb{R}^s$ and output $y_i \in \mathbb{R}^d$. And when we work with RNNs the sample set S is defined as $S = \{(t_1, x_1, y_1), \ldots, (t_m, x_m, y_m)\}$, with m tuples of input $x_i \in \mathbb{R}^{s \times t_i}$ and output $y_i \in \mathbb{R}^{d \times t_i}$. When $\mathbf{R}$ is defined as a zero matrix, the network is not connected to the previous state, and is equal to the non-recurrent version. In 1988, LeCun defined a theoretical framework [LeC+88] in which he derived the back-propagation algorithm as a constrained minimization problem. Let J be a function that has a global minimum when the network correctly predicts the output from S, with only the input given. This allows us to write the training of the network architecture as the following optimization problem

$$\min_{\mathbf{L},\mathbf{R},\mathbf{b}} J(\mathbf{L}, \mathbf{R}, \mathbf{b}, S)$$
$$\text{s.t. } z_i = f_i(\mathbf{z}),$$

where f_i is a smooth multivariate function of a subset of $\mathbf{z}$ that defines the relation between nodes in the computational graph, and $\mathbf{z}$ is the vector of intermediate values in the computational graph. The relations between z_i values are defined by a computational graph, and thus there are no cyclic constraints. Let $a+1$ be the length of the vector $\mathbf{z}$. We define $\mathbf{z}_{a+1}$ to be the output of the computational graph:

$$z_{a+1} = J(\mathbf{L}, \mathbf{R}, \mathbf{b}, S).$$

We can convert the previously given constrained optimization problem into an unconstrained problem by using a Lagrangian framework similar to the one specified by LeCun [LeC+88]. Now let $\mathbb{I} = [1, \ldots, a]$ be the indices of the intermediate values. The Lagrange function is now defined as

$$\mathscr{L}(\mathbf{L}, \mathbf{R}, \mathbf{b}, \mathbf{z}, \lambda) = z_{a+1} - \sum_{i=1}^{a+1} \lambda_i \left(z_i - f_i(\mathbf{z}) \right). \tag{3.8}$$

When the function $\mathscr{L}$ is in an optimum, the condition

$$\nabla \mathscr{L}(\mathbf{L}, \mathbf{R}, \mathbf{b}, \mathbf{z}, \lambda) = \mathbf{0} \tag{3.9}$$

holds while meeting the necessary constraints. (Unfortunately, this is also true for saddle points and local minima.) Due to this condition we need that for all i

$$\frac{\partial \mathscr{L}(\mathbf{L}, \mathbf{R}, \mathbf{b}, \mathbf{z}, \lambda)}{\partial \lambda_i} = \mathbf{0}, \tag{3.10}$$

which directly follows from Equation (3.8) when for all $i \in \mathbb{I}$

$$z_i = f_i(\mathbf{z}). \tag{3.11}$$

This constraint defines the forward-propagation of the network [GB10b]. We need the gradient with respect to the objective function to be 0 too:

$$\begin{aligned} 0 &= \frac{\partial \mathscr{L}(\mathbf{L}, \mathbf{R}, \mathbf{b}, \mathbf{z}, \lambda)}{\partial z_{a+1}} \\ &= \frac{\partial}{\partial z_{a+1}} \left(z_{a+1} - \sum_{j=1}^{a+1} \lambda_j (z_j - f_j(\mathbf{z})) \right) \\ &= 1 - \lambda_{a+1}. \end{aligned}$$

Thus we need $\lambda_{a+1} = 1$ to satisfy our condition. The back-propagation is obtained from the constraints of the training data input and node calculations. For all $i \in \mathbb{I}$ it must hold that

$$\nabla_{\mathbf{z}} \mathscr{L}(\mathbf{L}, \mathbf{R}, \mathbf{b}, \mathbf{z}, \lambda) = \frac{\partial \mathscr{L}(\mathbf{L}, \mathbf{R}, \mathbf{b}, \mathbf{z}, \lambda)}{\partial z_i} = \mathbf{0}. \tag{3.12}$$

Let $u(i) = O_i \subseteq \mathbb{I}$ be a map that maps the node index i to the indices of its output nodes, after which we can rewrite Equation (3.12) as

$$\begin{aligned} 0 &= \frac{\partial \mathscr{L}(\mathbf{L}, \mathbf{R}, \mathbf{b}, \mathbf{z}, \lambda)}{\partial z_i} \\ &= \frac{\partial}{\partial z_i} \left(z_{a+1} - \sum_{j \in \mathbb{I}} \lambda_j (z_j - f_j(\mathbf{z})) \right) \\ &= \frac{\partial}{\partial z_i} \left(- \sum_{j \in \mathbb{I}} \lambda_j (z_j - f_j(\mathbf{z})) \right) \\ &= -\lambda_i + \sum_{j \in \mathbb{I}} \lambda_j \frac{\partial f_j(\mathbf{z})}{\partial z_i} \\ &= -\lambda_i + \sum_{j \in u(j)} \lambda_j \frac{\partial f_j(\mathbf{z})}{\partial z_i}. \end{aligned}$$

From which follows that

$$\lambda_i = \sum_{j \in u(i)} \lambda_j \frac{\partial f_j(\mathbf{z})}{\partial z_i}, \tag{3.13}$$

which gives the derivative of the objective function J with respect to every node i. Since we defined both $\mathbf{L}$ and $\mathbf{R}$ as weighted adjacency matrices, we can easily obtain the ingoing connections of node i by taking the non-zero elements of the row, and the outgoing connections by taking the non-zero elements of the column of $\mathbf{L}$, and similarly for $\mathbf{R}$.

4 The Feed Forward Algorithm

In the previous chapter we gave a general definition of ANNs on DAGs. This chapter implements our equivalent of the FNN, which we will call Neural Graphs (NG), and omit the recurrent version for now. In Equation (3.13) we obtained the formula that defines the back-propagation algorithm for the required structure. To get an idea whether the more general graph structure works, we will test it against benchmark problems in Chapter 6. In this chapter we will define the building blocks we need to form the NG; the activation functions, and the objective function. With these building blocks we can define the NG algorithms; the forward-propagation, the backward-propagation, the optimization functions, and finally strategies to improve the NG. The workings of ANNs are highly dependent on the type of problem they have to solve. The benchmark problems we will solve in Chapter 6 are classification problems. This type of problem requires some extra theory that is also given in this chapter.

4.1 Building Blocks

Although we have discussed the architecture of the NG algorithm, we still have to fill in some of the variables. One such variable is which activation functions we use. In our algorithm we will use two separate activation functions; the LeakyRelu for hidden, and Softmax for output nodes, which both will be defined in the following section. We also have to define the objective function that has a global minimum when the network correctly predicts the output from the sample input.

4.1.1 Activation Functions

To make use of any sparsity in the weighted edges in the NGs we need to make sure that for every activation function $f : \mathbb{R} \mapsto \mathbb{R}$, and a small value ϵ we have $f(\epsilon) \approx f(-\epsilon) \approx 0$. Using these properties we will show that any weight that is smaller than ϵ will have no effect on the output of the activation function. For each node i the value is calculated by multiplying the output of previous nodes with a weight vector $\mathbf{w}_i$, and using the bias b_i and the activation function f_i. Now let Z_i be the set of indices for which the elements of vector $\mathbf{w}_i$ are 0, and E_i where the

elements are not equal to 0.

$$\begin{aligned} x_i &= f_i(\langle (x_1, \ldots, x_{i-1}), \mathbf{w}_i \rangle + b_i) \\ &= f_i \left(\sum_{k=1}^{i} x_k \cdot (\mathbf{w}_i)_k + b_i \right) \\ &= f_i \left(\sum_{k \in E_i} x_k \cdot (\mathbf{w}_i)_k + \sum_{k \in Z_i} x_k \cdot (\mathbf{w}_i)_k + b_i \right) \\ &= f_i \left(\sum_{k \in E_i} x_k \cdot (\mathbf{w}_i)_k + b_i \right). \end{aligned}$$

Due to this property we can both calculate the propagation through the edge in a dense calculation, or omit it completely in a sparse calculation.

LeakyRelu

For our hidden nodes we choose a LeakyRelu activation function, which is defined as

$$f(x, \alpha_i) = \max(0, x) + \alpha_i \min(0, \alpha_i),$$

where α_i could be turned into a trainable parameter for each node i. However, we choose a fixed $\alpha_i = 0.01$.

Softmax

Finally, we define the activation function for the categorical output nodes. Let o be the length of the output, and $y \in [1, \ldots, o]$ be the index of the label we want to predict. First we encode our output labels into a so-called one-hot vector, see Table 4.1, where the label i has element i of the vector equal to 1 and the rest of the vector 0. Now we want to calculate the conditional probability of the category $\hat{y}_i = P(y = i \mid \mathbf{x})$ which is the multinomial distribution.

category	integer	one-hot
red	1	100
yellow	2	010
green	3	001

Table 4.1: An example of integer encoding and one-hot encoding of categorical data.

Given a weight matrix $\mathbf{W}$, state $\mathbf{h}$ and bias $\mathbf{b}$, we assume that $\mathbf{z} = \mathbf{W}\mathbf{h} + \mathbf{b}$ is a vector of unnormalized log-space probabilities such that $z_i = \ln \hat{P}(y = i|\mathbf{x})$, where $\hat{P}$ is an unnormalized probability distribution [GBC16]. By exponentiating and normalizing we get the softmax function, see Equation (4.1). Given the output vector $\mathbf{z}$, the softmax function $\sigma : \mathbb{R}^o \mapsto [0, 1]^o$ calculates a probability distribution over a discrete

variable with n possible values [GBC16]. This makes it a good activation function for the output, since we want to classify from o different categories. The softmax function is defined as

$$p_j = \sigma(\mathbf{z})_j = \frac{\exp(z_j)}{\sum_{k=1}^{o} \exp(z_k)}. \tag{4.1}$$

One important property of the softmax function is its invariance to offsets:

$$\begin{aligned}
\sigma(\mathbf{z} + c)_j &= \frac{\exp(z_j + c)}{\sum_{k=1}^{o} \exp(\mathbf{z}_k + c)} \\
&= \frac{\exp(c)\exp(z_j)}{\exp(c)\sum_{k=1}^{o} \exp(z_k)} \\
&= \frac{\exp(z_j)}{\sum_{k=1}^{o} \exp(z_k)} \\
&= \sigma(\mathbf{z}).
\end{aligned}$$

In implementations we choose $c = -\max_i z_i$ to make the computations more stable. The derivative of the softmax function with respect to the $\mathbf{z}$ vector has two cases:

$$\frac{\partial p_j}{\partial z_i} = \begin{cases} p_i(1 - p_i), & \text{for } i = j, \\ -p_i p_j, & \text{for } i \neq j. \end{cases} \tag{4.2}$$

4.1.2 Objective Function

Since the benchmark problem is defined as a classification task, we can use a cross entropy objective function. Let $\hat{\mathbf{y}}$ be the calculated output which approximates the true output $\mathbf{y}$. Now the cross entropy function $J : \mathbb{R}^o \mapsto \mathbb{R}$ is defined as

$$J(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{o} y_j \ln \hat{y}_j,$$

and it calculates the expected number of bits required to identify random samples from the distribution of $\mathbf{y}$ when using an optimal coding scheme for $\hat{\mathbf{y}}$.

When the cross entropy function is used in combination with the output of the softmax function we obtain the following derivative given that $\mathbf{y}$ is a one-hot vector (note that the output of the softmax is substituted as

the predicted value p_j):

$$\begin{aligned}\frac{\partial J}{\partial z_i} &= -\sum_j^o y_j \frac{\partial \ln p_j}{\partial z_i} \\ &= -y_i(1-p_i) - \sum_{j\neq i} y_j \frac{1}{p_j}(-p_j p_i) \\ &= -y_i(1-p_i) + \sum_{j\neq i} y_j p_i \\ &= -y_i + y_i p_i + \sum_{j\neq i} y_j p_i \\ &= -y_i + \sum_{i=1}^o y_j p_i.\end{aligned}$$

We defined $\mathbf{y}$ to be a one-hot vector (only one element with value one, and the rest zero), thus this formula simplifies to

$$\frac{\partial J}{\partial z_i} = p_i - y_i. \tag{4.3}$$

We use this composite function of the softmax function together with the cross entropy function as the objective function for training.

4.2 Neural Graph Algorithms

Now that all building blocks are defined for the NG, we can define both the algorithms for forward-propagation and back-propagation through the network. These algorithms then can be used for the gradient optimization algorithms. Unfortunately, the optimization algorithms also have parameters, for which we will also need to specify methods of finding correct values.

4.2.1 The Forward-Propagation Algorithm

The first algorithm we have to specify is the forward-propagation algorithm, since the intermediate values are also needed for the back-propagation algorithm. The forward-propagation algorithm, see Algorithm 4.1, calculates the output and intermediate values of the NG. The function gives as output two vectors, ξ and $\mathbf{z}$, where the former is the output state for the corresponding node, and the latter holds the input state. The vectors ξ and $\mathbf{z}$ are calculated with the formulas we described earlier

$$\begin{aligned}z_i &= \langle \mathbf{L}_{i,*}, \xi \rangle + b_i, \\ \xi_i &= g_i(z_i).\end{aligned}$$

Algorithm 4.1. The forward-propagation algorithm.

input : n : input size,
m : number of hidden nodes,
o : output size,
ω : output function,
$\mathbf{x}$: input vector of length n,
$\mathbf{b}$: bias vector of length $n + m + o$,
$\mathbf{L}$: lower triangular weighted adjacency matrix,
output : $\mathbf{y}$: output vector of length o,
ξ : output state vector for all nodes with length $n + m + o$,
$\mathbf{z}$: input state vector for all nodes with length $n + m + o$.

function FORWARDPROPAGATE
let $\xi = \mathbf{0}$ of size $n + m + o$
let $\xi[: n] = \mathbf{x}$
let $\mathbf{z} = \mathbf{0}$ of size $n + m + o$.
let $\mathbf{z}[: n] = \mathbf{x}$

for $i = n$ **to** $n + m$ **do**
compute $x = \langle \mathbf{L}_{i,*}, \xi \rangle + b_i$.
let $\xi_i = f_i(x)$
let $z_i = x$

for $i = n + m + 1$ **to** $n + m + o$ **do**
compute $z_i = \langle \mathbf{L}_{i,*}, \xi \rangle + b_i$

compute $\xi[n + m + 1 : n + m + o] = \omega(\mathbf{z}[n + m + 1 : n + m + o])$
compute $\mathbf{y} = \xi[n + m + 1 : n + m + o]$
end function

The first n nodes are regarded as input nodes that have the same input and output. This means that we have to set the first n elements to the value of the input for both ξ and $\mathbf{z}$. The next m nodes are the hidden nodes. For every hidden node we calculate the affine transformation, which is the dot product of the incoming weights in that node and the output of previous nodes, transformed by the bias for that node. We store this affine transformation in the vector $\mathbf{z}$ and calculate the output of the function by substituting this affine transformation to the activation function, which in its turn gets stored in $\mathbf{z}$. Finally, we are left with the calculation of the output nodes. Again we calculate the affine transformation as input of these nodes, after which we use the entirety of these nodes to calculate the output function. These values are also stored in $\mathbf{z}$ and ξ in the corresponding elements. In our implementation we can use this

property to only store the section of the adjacency matrix that contributes to this part. This property also holds for the bias vector. As a result we can reduce the size of the adjacency matrix to $(m + o) \times (n + m + o)$ and the bias vector to $m + o$. If we only want to calculate the output of the NG we do not need to compute and store the $\mathbf{z}$ as it is only needed in the back-propagation algorithm.

4.2.2 Training the Neural Graph

Although we can calculate the output of the NG, we still have to find correct values for $\mathbf{L}$ and $\mathbf{b}$ before the output is meaningful. The NG is trained by giving it examples, for which we optimize the objective function. This means that we need to divide our data set into a training set and a validation set. The training data set is used to train the algorithm, and the validation data set is used to measure how well the trained model generalizes to examples it has not seen before.

Mini-batch methods

Instead of training the model on the complete training data set, we use a stochastic method. The training data set is divided into a given number of mini-batches of equal size. A larger batch gives a more accurate gradient of the objective function. This means that a smaller batch size introduces a noise to the objective function, which can offer a regularizing effect [WM03]. Algorithm 4.2 shows the general learning algorithm. The CalculateGradient method calculates the gradient of the NG given the parameters. Note that for the computation of the average of the gradient that is calculated for every mini-batch, each individual sample can be parallelized. After the average gradient is calculated, we calculate the new parameters by performing the UpdateParameters function. This function can be implemented using an optimization algorithm such as SGD. A single epoch is when the entire training data set is used for training once. Thus for every epoch the training algorithm retrains the network on the complete data set. Since we use iterative methods to update the parameters, we need more than one epoch to get respectable results from our training. The maximum number of epochs, which defines the end-condition of the training algorithm, is also passed to the learning algorithm.

Algorithm 4.2. The learning algorithm.

input : $h_{\max}$: the total number of epochs it will run,
s : the batch size,
S : list of input samples,
O : list of corresponding outputs,
θ : initial parameters,
output : θ : the learnt parameters.

let $i = 0$.
let $h = 0$.
let $s_{\text{total}} = \lfloor \text{SIZE}(S)/s \rfloor$
let $\nabla\theta = \mathbf{0}$

while $h < h_{\max}$ **do**
  calculate $\hat{h} = \lfloor i/s_{\text{total}} \rfloor$
  calculate $j = i \bmod s_{\text{total}}$
  if $i \neq j$ and $i > 0$ **then**
    CalculateAccuracy().
  end if
  calculate $k_{\text{start}} = s \cdot j$
  calculate $k_{\text{end}} = \min(s \cdot (j+1), \text{SIZE}(S))$
  for $k = k_{\text{start}}$ **to** k_{end} **do**
    calculate $\nabla\hat{\theta} = \text{CalculateGradient}(S[k], O[k], \theta)$
    calculate $\nabla\theta = \nabla\theta + \frac{1}{k_{\text{end}} - k_{\text{start}}} \nabla\hat{\theta}$
  end for
  calculate $\theta = \text{UpdateParameters}(\theta, \nabla\theta)$
  let $h = \hat{h}$
  let $i = i + 1$
end while

Back-propagation

The hard part of the training of a NG is calculating the gradient of the objective function. In Equation (3.13) we obtained the basic definition of the back-propagation algorithm using Lagrange multipliers:

$$\lambda_i = \sum_{j \in u(i)} \lambda_j \frac{\partial f_j(\mathbf{z})}{\partial z_i}.$$

Now we can expand this formula specifically for the problem we have defined. Since the gradient of a node is dependent on the value of the outgoing connections, we first need to do a forward-propagation to calculate the states of the nodes. The Lagrange multiplier for the objective function is defined as 1. Due to this, the Lagrange multipliers

of the output nodes can be calculated by calculating the derivative of the objective function with respect to output nodes such as was calculated in Equation (4.3):

$$\lambda_i = 1 \cdot \frac{\partial J}{\partial z_i} = p_i - y_i,$$

where i is the i-th output node. The weight gradient for all nodes i can be calculated from these Lagrange multipliers by using the chain rule:

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{L}_{*,i}} &= \frac{\partial J}{\partial z_i}\frac{\partial z_i}{\partial \mathbf{L}_{*,i}} \\ &= \lambda_i \left(\frac{\partial f_i}{\partial \mathbf{L}_{*,i}} \left(\mathbf{L}_{*,i}\xi + b_i\right)\right) \\ &= \lambda_i \xi f_i'(\mathbf{L}_{*,i}\xi + b_i),\end{aligned}$$

and for the bias gradient we get

$$\begin{aligned}\frac{\partial J}{\partial b_i} &= \frac{\partial J}{\partial z_i}\frac{\partial z_i}{\partial b_i} \\ &= \lambda_i \left(\frac{\partial f_i}{\partial b_i} \left(\mathbf{L}_{*,i}\xi + b_i\right)\right) \\ &= \lambda_i f_i'(\mathbf{L}_{*,i}\xi + b_i).\end{aligned}$$

The hidden nodes require a bit more work since we first need to calculate the Lagrange multiplier, before we can use it to calculate the gradients. We have a composite function of the affine transformation and the activation function, from which we get the Lagrange multipliers

$$\begin{aligned}\lambda_i &= \frac{\partial J}{\partial z_i} \\ &= \sum_{j=0}^{m+n+o} \lambda_j \frac{\partial f_i}{\partial z_i} \left(\mathbf{L}_{*,i}\xi + b_i\right) \\ &= \sum_{j=0}^{m+n+o} \lambda_j \mathbf{L}_{j,i}.\end{aligned}$$

Algorithm 4.3 puts these formulas together and calculates the complete gradient for the weights and the biases.

Algorithm 4.3. The back-propagation algorithm.

input : n : input size,
m : number of hidden nodes,
o : output size,
J : objective function,
ω : output function,
$\mathbf{x}$: input vector of length n,
$\hat{\mathbf{y}}$: true output vector of length o,
$\mathbf{b}$: bias vector of length $n + m + o$,
$\mathbf{L}$: lower triangular weighted adjacency matrix,
output : $\nabla\mathbf{b}$: the gradient of the bias vector,
$\nabla\mathbf{L}$: the gradient of the weight matrix.

compute $[\mathbf{y}, \xi, \mathbf{z}] = \textsc{ForwardPropagate}(n, m, o, \omega, \mathbf{x}, \mathbf{b}, \mathbf{L})$.

let $\lambda = \mathbf{0}$
let $\nabla\mathbf{b} = \mathbf{0}$
let $\nabla\mathbf{L} = \mathbf{0}$

compute $\mathbf{h} = J'(\xi[n + m : n + m + o], \hat{\mathbf{y}})$.
let $\lambda[n + m : n + m + o] = h$
let $\nabla\mathbf{b}[n + m : n + m + o] = h$

for $j = n + m + o$ **to** $n + m$ **do**
  for $\{i \in [1, n + m + o] | \mathbf{L}_{j,i} \neq 0\}$ **do**
    compute $\nabla\mathbf{L}_{j,i} = \lambda_j \xi_i$

for $j = n + m$ **to** n **do**
  let $\sigma = 0$.
  for $\{i \in [1, n + m + o] | \mathbf{L}_{i,j} \neq 0\}$ **do**
    compute $\sigma = \sigma + \lambda_i \mathbf{L}_{i,j}$

  compute $\lambda_j = \sigma f_i'(z_j)$

  for $\{i \in [1, n + m + o] | \mathbf{L}_{j,i} \neq 0\}$ **do**
    compute $\nabla\mathbf{L}_{j,i} = \lambda_j \xi_i$.

  let $\nabla b_j = \lambda_j$

4.2.3 Optimization Algorithms

The back-propagation algorithm calculates the gradients of the parameters of our model with respect to our chosen objective function. With these gradients we can update our initially chosen parameters in a direction that minimizes the objective function. For every sample we get a separate gradient, from which we calculate the average gradient in our mini-batch, see Algorithm 4.2. With this averaged gradient we update our parameters with an optimization algorithm. Let θ be the parameter we want to update, and $\nabla\theta$ the average gradient for this parameter. We define a parameter η as the learning rate, which controls how fast the algorithm learns. For these variables we now define several widely used optimization algorithms.

SGD

Intuitively, in this algorithm, we let the objective function be a surface defined by the parameters of the model. In SGD we drop a weight-less ball on this surface and let it roll to a minimum. The learning rate allows us to increase or decrease the accuracy of simulation, where a small learning rate will be more sensitive to local minima in the objective surface. The SGD algorithm is given in Algorithm 4.4

Algorithm 4.4. The SGD algorithm.

$$
\begin{aligned}
\textit{input}: \eta &: \text{ the learning rate,} \\
\theta &: \text{ the parameters of the objective function,} \\
J &: \text{ the objective function,} \\
\textit{output}: \theta &: \text{ the updated parameters.}
\end{aligned}
$$

Estimate gradient $\nabla\theta = \mathbb{E}\left[\nabla_\theta J\right]$
Calculate $\theta = \theta - \eta\nabla\theta$

Definition 6 (Convex Functions).

A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is called convex if for every $x, y \in \mathbb{R}^n$, and for all $\lambda \in [0, 1]$

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y).$$

Definition 7 (Lipschitz Continuous Functions).

A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is called Lipschitz continuous when there exists a $L > 0$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ such that

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L \|\mathbf{x} - \mathbf{y}\|_2 .$$

Let the function $f : \mathbb{R}^n \mapsto \mathbb{R}$ be convex and Lipschitz continuous, see Definition 6 and Definition 7. By expanding the function f around the point $\mathbf{x} \in \mathbb{R}^n$ we get for every $\mathbf{y} \in \mathbb{R}^n$ the following quadratic Taylor series approximation

$$\begin{aligned} f(\mathbf{y}) &\leq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \frac{1}{2}\nabla^2 f(\mathbf{x}) \|\mathbf{y} - \mathbf{x}\|_2^2 \\ &\leq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \frac{1}{2} L \|\mathbf{y} - \mathbf{x}\|_2^2 . \end{aligned}$$

Now we define $\hat{\mathbf{x}} = \mathbf{x} - \eta \nabla f(\mathbf{x})$, which is the SGD updated $\mathbf{x}$ vector. By combining these we get the following

$$\begin{aligned} f(\hat{\mathbf{x}}) &\leq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\hat{\mathbf{x}} - \mathbf{x}) + \frac{1}{2} L \|\hat{\mathbf{x}} - \mathbf{x}\|_2^2 \\ &\leq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{x} - \eta \nabla f(\mathbf{x}) - \mathbf{x}) + \frac{1}{2} L \|\mathbf{x} - \eta \nabla f(\mathbf{x}) - \mathbf{x}\|_2^2 \\ &\leq f(\mathbf{x}) - \nabla f(\mathbf{x})^T \eta \nabla f(\mathbf{x}) + \frac{1}{2} L \|\eta \nabla f(\mathbf{x})\|_2^2 \\ &\leq f(\mathbf{x}) - \eta \|\nabla f(\mathbf{x})\|_2^2 + \frac{1}{2}\eta^2 L \|\nabla f(\mathbf{x})\|_2^2 \\ &\leq f(\mathbf{x}) - (1 - \frac{1}{2}\eta L)\eta \|\nabla f(\mathbf{x})\|_2^2 . \end{aligned}$$

When $\eta \leq \frac{1}{L}$, the inequality simplifies to

$$f(\hat{\mathbf{x}}) \leq f(\mathbf{x}) - \frac{1}{2}\eta \|\nabla f(\mathbf{x})\|_2^2 .$$

This means that when $\nabla f(\mathbf{x}) > 0$, SGD decreases the value of the objective function. This results in SGD iterating to local minima in non-convex optimization problems. In practice we can alter the learning rate to make SGD more robust to these local minima by stepping over them.

Momentum

The most common method of increasing the algorithm's robustness, is by adding momentum to the movement of parameters. If we again see our optimization problem as a ball drop on an objective surface, momentum adds velocity to this ball. Instead of being captured by local minima, it

will retain its speed and move out of such minima again. This behaviour is introduced by adding the momentum parameter μ into the SGD formula. We first calculate the new velocity of the parameters, which is done by retaining a previous part of the velocity, controlled by μ, and adding the new part by using the gradient and learning rate. With this velocity we can calculate the new parameters as seen in Algorithm 4.5.

Algorithm 4.5. The SGD algorithm with momentum.

$$
\begin{aligned}
\textit{input}: \eta &: \text{ the learning rate,} \\
\theta &: \text{ the parameters of the objective function,} \\
v &: \text{ the initial velocity,} \\
\mu &: \text{ the momentum,} \\
J &: \text{ the objective function,} \\
\textit{output}: \theta &: \text{ the updated parameters.}
\end{aligned}
$$

Estimate gradient $\nabla\theta = \mathbb{E}\left[\nabla_\theta J\right]$
Update the velocity $\mathbf{v} = \mu\mathbf{v} - \eta\nabla\theta$
Calculate $\theta = \theta + \mathbf{v}$

A more popular method of calculating the velocity, called the Nestrov-Momentum, evaluates the gradient after we adjusted the parameters for the velocity, as seen in Algorithm 4.6. Since the gradient is evaluated after the velocity is applied, the Nestrov-Momentum can be seen as a correction factor [GBC16]. The Nestrov-Momentum has stronger convergence guarantees on convex optimization problems, and performs more consistently in practice [Sut+13].

Algorithm 4.6. The SGD algorithm with Nestrov-Momentum.

$$
\begin{aligned}
\textit{input}: \eta &: \text{ the learning rate,} \\
\theta &: \text{ the parameters of the objective function,} \\
\mu &: \text{ the momentum,} \\
J &: \text{ the objective function,} \\
\textit{output}: \theta &: \text{ the updated parameters.}
\end{aligned}
$$

Estimate gradient $\nabla\theta = \mathbb{E}\left[\nabla_\theta J\right]$
Update the velocity $\mathbf{v}_i = \mu\mathbf{v}_{i-1} - \eta\nabla\theta$
Calculate $\theta = \theta - \mu\mathbf{v}_{i-1} + (1+\mu)\mathbf{v}_i$

Adam

Another widely used extension to the SGD algorithm is the Adam optimization algorithm [KB14]. Adam makes use of previously seen parameter values and gradients to adaptively alter the momentum. It incorporates

the main advantages of two other algorithms: RMSProp [TH12] and AdaGrad [Zei12]. This results in an algorithm that alters the learning rate per-parameter, which improves problems with sparse gradients, and it takes the magnitude of the gradient into account which improves the stochastic properties of the algorithm. The algorithm calculates the exponential moving average of the gradient and the squared gradient of the parameters, for which the parameters ρ_1 and ρ_2 control the decay rates of the effective learning rate. The iterative algorithm is given in Algorithm 4.7.

Algorithm 4.7. The Adam algorithm.

$$
\begin{aligned}
\textit{input}: \eta &: \text{ the learning rate,} \\
\theta &: \text{ the parameters of the objective function,} \\
\mu &: \text{ the momentum,} \\
t &: \text{ the time-step,} \\
J &: \text{ the objective function,} \\
\textit{output}: \theta &: \text{ the updated parameters.}
\end{aligned}
$$

Estimate gradient $\nabla\theta = \mathbb{E}\left[\nabla_\theta J\right]$
Calculate $\mathbf{s}_i = \rho_1 \mathbf{s}_{i-1} + (1-\rho_1)\nabla\theta$
Calculate $\mathbf{r}_i = \rho_2 \mathbf{r}_{i-1} + (1-\rho_2)\langle\nabla\theta, \nabla\theta\rangle$
Calculate $\hat{\mathbf{s}}_i = \dfrac{\mathbf{s}_i}{1-\rho_1^t}$
Calculate $\hat{\mathbf{r}}_i = \dfrac{\mathbf{r}_i}{1-\rho_2^t}$
Calculate $\theta = \theta - \dfrac{\eta\hat{\mathbf{s}}_i}{\sqrt{\hat{\mathbf{r}}_i} + \epsilon}$

AMSGrad

Although Adam is still very popular, it can be proved that for any decay parameters $\rho_1, \rho_2 \in [0, 1[$ such that $\rho_1 < \sqrt{\rho_2}$ there is a stochastic convex optimization problem where Adam does not converge to an optimal solution [RKK18]. Reddi, Kale and Kumar show how the problems in the exponential smoothing can be fixed and introduce a new variant of the algorithm called AMSGrad. AMSGrad is presented in Algorithm 4.8.

Algorithm 4.8. The AMSGrad algorithm.

input : η : the learning rate,
θ : the parameters of the objective function,
μ : the momentum,
J : the objective function,
output : θ : the updated parameters.

Estimate gradient $\nabla\theta = \mathbb{E}\left[\nabla_\theta J\right]$
Calculate $\mathbf{s}_i = \rho_1 \mathbf{s}_{i-1} + (1-\rho_1)\nabla\theta$
Calculate $\mathbf{r}_i = \rho_2 \mathbf{r}_{i-1} + (1-\rho_2)\langle\nabla\theta, \nabla\theta\rangle$
Calculate $\hat{\mathbf{r}}_i = \max(\hat{\mathbf{r}}_{i-1}, \mathbf{r}_i)$
Calculate $\theta = \theta - \dfrac{\eta \mathbf{s}_i}{\sqrt{\hat{\mathbf{r}}_i} + \epsilon}$

4.2.4 Learning Rate

We until now, have regarded the learning rate η to be fixed. The learning rate parameter determines the accuracy of the optimization algorithms. This means that in practice a fixed learning rate will have trouble to converge to an optimal solution. A common approach to solve this problem is by using an annealing schedule for the learning rate, where the learning rate decreases every epoch. One of such annealing schedules is the step decay annealing algorithm, where the learning rate is reduced by a multiplying factor every few epochs. An annealing schedule uses an exponential back-off to model the learning rate. Let η_0 be the initial learning rate, t be the iteration or epoch count, and λ controls the decay rate. The schedule is then defined as

$$\eta = \eta_0 e^{-\lambda t}.$$

A recent improvement in learning rate schedules is called cyclic learning. The learning rate is often seen as the most import parameter of the learning algorithm, and thus finding a good learning rate schedule is important. Cyclic learning eliminates the need for empirically finding the best learning rate for a problem [Smi17]. Instead of monotonically decreasing the learning rate, we allow learning rate schedules to also increase the learning rate with a periodicity. The basic schedule for Cyclic Learning is the Triangular learning rate schedule, which is presented in Algorithm 4.9. This schedule specifies the learning rate that follows a repeating triangle, with the given lower and upper bound of the learning rate. First we introduce the periodicity of the function, using

$$c = t \mod 2s,$$

where t is the current epoch, and s is the half cycle value. We want to normalize this value between -1 and 1, so we first have to divide it by s.

Now we have $0 \leq \frac{c}{s} < 2$. We translate this to $-1 \leq \frac{c}{s} - 1 < 1$. By taking the absolute value of this value, we get a periodic triangular function between 0 and 1, calculated as

$$x = |\frac{t}{s} - 2c - 1|.$$

We want to start the the triangular function with the lower bound of the learning rate. This means that we flip the value by calculating $1 - x$. Now we use an affine transformation to specify the minimum and maximum values as the lower and upper bound of the learning rate

$$\eta = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \cdot (1 - x).$$

To make sure we converge, we extend this algorithm by adding a decay factor, see Algorithm 4.10. Now we have a learning rate schedule that is specified by a repeating triangle where the minimum and maximum use an exponential decay.

Algorithm 4.9. The Triangular Cyclic Learning algorithm.

input : $\eta_{\min}$: the lower bound of the learning rate,
$\eta_{\max}$: the upper bound of the learning rate,
t : the current epoch count,
s : the half cycle size,
output : η : the effective learning rate.

function TRIANGULAR($\eta_{\min}, \eta_{\max}, t, s$)
Calculate $c = t \mod 2s$
Calculate $x = |\frac{t}{s} - 2c - 1|$
Calculate $\eta = \eta_{\min} + (\eta_{\max} - \eta_{\min}) \cdot (1 - x)$
end function

Algorithm 4.10. The Exponential Range Cyclic Learning algorithm.

input : $\eta_{\min}$: the lower bound of the learning rate,
$\eta_{\max}$: the upper bound of the learning rate,
γ : the exponential decay factor,
t : the epoch count,
i : the iteration count,
s : the half cycle size,
output : η : the effective learning rate.

Calculate $\hat{\gamma} = \gamma^i$
Calculate $\eta =$ TRIANGULAR($\hat{\gamma} \cdot \eta_{\min}, \hat{\gamma} \cdot \eta_{\max}, t, s$)

Now that we have an algorithm that is robust to the chosen initial learning rate, we still need to find a proper initial minimum and maximum learning rate. Although Smith presented a method in [Smi17] to find the boundaries, we will specify a similar but different method to finding these values. In Algorithm 4.11 we give a learning rate schedule that exponentially increases the learning rate between two bounds. We run the normal learning algorithm, see Algorithm 4.2, but we record both the loss value and the effective learning rate. From these two values we can make a visualisation of how the learning rate has an impact on the objective function. First we difference all consecutive values, which gives the rate of change in the objective function over the used learning rates. When we smooth these values, using a smooth moving average, we get a plot similar to Figure 4.1. In this plot we can find the minimum objective rate of change, and take it as the upper-bound of the learning rate. A simple method to define the lower-bound is to multiply the upper-bound by a fraction such as 1/3.

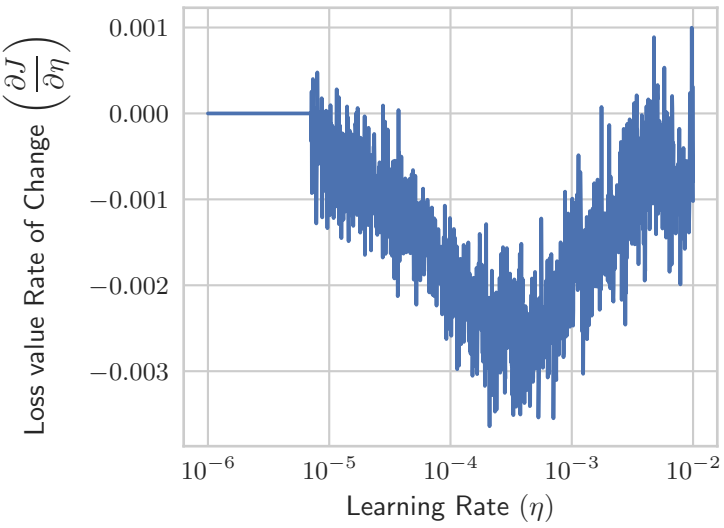


Figure 4.1: A learning rate versus rate of change of the objective function value plot for the MNIST data set.

Algorithm 4.11. The Rate Learning algorithm.

$$
\begin{aligned}
\textit{input} : \eta_{\min} &: \text{ the lower bound of the learning rate,} \\
\eta_{\max} &: \text{ the upper bound of the learning rate,} \\
\gamma &: \text{ the exponential decay factor,} \\
b &: \text{ the mini-batch size,} \\
i &: \text{ the iteration count,} \\
\textit{output} : \eta &: \text{ the effective learning rate.}
\end{aligned}
$$

Calculate $\alpha = \frac{1}{\eta_{\min}} \eta_{\max}^{1/b}$
Calculate $\eta = \eta_{\min} \cdot \alpha^i$

4.3 Regularization

Although we now have the means to learn and predict values with NGs, our main objective is to build a model that generalizes to unseen data. Due to the No Free Lunch theorem, see Theorem 4, generalization of machine learning models does not come naturally, and the model only infers rules that are likely to be true in the training data set. This means that we need to help the model generalize better, and prevent overfitting. An easy way to prevent overfitting is by regularizing the weight parameters, which ensures that we prefer smaller values for the weights. An added advantage is that this technique automatically improves the sparsity of the model, and thus reduces noise in the weights, and reduces the complexity of the model. To add the weight regularization, we simple modify the objective function J as

$$\hat{J}(\theta) = J(\theta) + \lambda \langle \theta, \theta \rangle,$$

with parameters θ, and regularization factor λ, that controls the penalty on the size of the parameters. This specific form of regularization is called L^2 regularization, since we use the L^2 norm. It is also possible to choose any L^p norm for regularization, but L^2 has proven to perform well in practice.

Theorem 4 (No Free Lunch Theorem).

"Averaged over all possible data-generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points." [GBC16] Proved by Wolpert in 1996 [Wol96].

Another regularization technique, introduced by Srivastava et al [Sri+14], is Dropout. Dropout tries to make the learning of features

in the data set more robust by deactivating nodes with a certain probability on each sample. When Dropout is applied to an ANN, convergence in training takes about 2-3 times longer in epochs, which is the result of the introduced noise, but the generalization of the model is improved. Dropout implicitly trains an ensemble of multiple graphs and averages the results. When a node is dropped-out, all incoming and outgoing edges are set to 0. This can be done by drawing from a Bernoulli distribution, with the given dropout chance, and multiplying the drawn value with the input of the node. The Bernoulli probability mass function is specified as

$$f(k,p) = \begin{cases} q = 1-p & \text{for } k = 0, \\ p & \text{for } k = 1, \end{cases}$$

where $k \in \{0,1\}$ and p is the probability. Let $D = (X_1, \ldots, X_n)$ be a vector drawn from the Bernoulli distribution. Since we defined our activation functions to output zero when the input is near zero, this also fixes the output connections. Srivastava suggests to modify the affine transformation and activation function in both the training and testing such that for the training we use

$$\xi_i = X_i f_i(\mathbf{L}_{*,i}\xi + b_i),$$

where p_i is the probability to drop the node i, f_i is the activation function, and b_i the bias of node i, $\mathbf{L}$ the weight matrix, and ξ the vector with the output of the previous nodes. During the test phase we need to rescale the output as well to

$$\xi_i = q_i f_i(\mathbf{L}_{*,i}\xi + b_i),$$

where $q_i = 1 - p_i$ is the probability to keep node i. The rescaling is done to compensate for the signal that is lost due to the dropped nodes. A simpler version to implement is called Inverted Dropout, which only scales the output on training, and leaves the test function intact. The training output is now defined as

$$\xi_i = \frac{1}{q_i} X_i f_i(\mathbf{L}_{*,i}\xi + b_i).$$

4.4 Weight Improvements

There are several ways of improving the performance of the NG, that target the weight definitions specifically. By using a specific initialization scheme, we can improve convergence. Another convergence gain lies in the calculation of the weights themselves, and finally we specify how we can make use of the sparseness introduced in the model.

Initialization

By initializing weights to certain values we can help the convergence of the optimization. The network should have a source of asymmetry, since otherwise the gradient of many nodes will be nearly the same, and thus the remaining values will stay near each other. We introduce asymmetry by drawing from a Gaussian distribution with mean zero, and variance one. Since the mean is zero, it is still easy for nodes to switch from a positive to a negative weight. We still need to make sure the variance of the output in nodes doesn't grow because there are an increasing number of connections with every node. To fix this we use He initialization [He+15]. He initialization is an initialization scheme, developed specifically for rectified linear units type of activations such as ReLu and LeakyRelu, which is based on the Xavier-Godot initialization scheme [GB10a]. He initialization assumes a Gaussian distribution with mean zero and variance

$$\text{Var}(\mathbf{w}_i) = \frac{2}{(1+\alpha_i^2)n_i},$$

where α_i is the parameter of the PRelu, n_i is the number of ingoing connections in node i.

Normalization

One of the most effective techniques to improve convergence of the optimization algorithms was introduced by Ioffe and Szegedy [IS15]. Their technique called Batch Normalization, reparametrizes the input of every node to mean zero and variance one over the mini-batch. Due to these operations, no gradient from the back-propagation will increase the mean or variance [GBC16]. Since the algorithm is dependent on the mini-batch samples, it is not feasible to use it in a recurrent setting, due to the variable length of sequences. Other techniques were developed to fix this problem, and one of such techniques, developed by Salimans and Kingma, is Weight Normalization [SK16]. Weight Normalization splits the direction and magnitude of the incoming weights in a node into a normalized vector and a scalar. For node i the weight vector $\mathbf{w}_i$ is parametrized as

$$\mathbf{w}_i = \frac{\mathbf{L}_{*,i}}{\|\mathbf{L}_{*,i}\|_2} g_i,$$

where g_i is the magnitude scalar for node i. Since we only change the formulation of the weights, we can easily add this to the back-propagation algorithm, where we obtain the following new gradients

$$\nabla g_i = \frac{\nabla \mathbf{w}_i \cdot \mathbf{L}_{*,i}}{\|\mathbf{L}_{*,i}\|_2},$$
$$\nabla \mathbf{L}_{*,i} = \frac{g_i \nabla \mathbf{w}_i}{\|\mathbf{L}_{*,i}\|_2} - \frac{g_i \nabla g_i}{\|\mathbf{L}_{*,i}\|_2^2} \mathbf{L}_{*,i}.$$

In practice Weight Normalization shows a performance near Batch Normalization, while it is computationally lighter. A large difference, however, is that it does not reparametrize the input of activations to zero mean and variance one, which makes the network more dependent on good initialization of the parameters.

Sparsification

Due to the sparseness in the model, we can accelerate the computations in the NG, by removing weights that have little effect on the output. First we assume we have a procedure NonZeros that returns an ascending sorted vector of non-zeros in a matrix, and a procedure DropValues that removes all absolute values that are smaller than a certain value. Using these functions, we define Algorithm 4.12, that calculates a bounded percentile value of the given weight matrix, and drops all values that are absolutely smaller than this value.

Algorithm 4.12. The Sparsify algorithm.

$$
\begin{aligned}
\textit{input} : \alpha : &\text{ the lower bound of the percentile value,} \\
p : &\text{ the percentile we remove} \\
\mathbf{W} : &\text{ a weighted adjacency matrix,} \\
\textit{output} : \eta : &\text{ the effective learning rate.}
\end{aligned}
$$

Let $\hat{W} = (|W_{i,j}|)$
Let $\mathbf{w} = \textsc{NonZeros}(\hat{W})$
Let $i = \lfloor p \cdot \textsc{Size}(\mathbf{w}) \rfloor$
Let $v = w_i$
Let $\hat{v} = \min(v, \alpha)$
Update $W = \textsc{DropValues}(W, \hat{v})$

5 The Recurrent Algorithm

The recurrent version of the NG, Recurrent Neural Graphs (RecurrentNG), follows from the forward version. We see the recurrent graph as an unrolled DAG, which allows us to use the forward algorithms with only a few changes. Instead of calculating the input of the current node from only the forward part, it is extended with the recurrent part

$$z_i^{(t)} = \langle \mathbf{L}_{i,*}, \xi^{(t)} \rangle + \langle \mathbf{R}_{i,*}, \xi^{(t-1)} \rangle + b_i,$$
$$\xi_i^{(t)} = g_i(z_i^{(t)}),$$

where $z_i^{(t)}$ is the input of node i at time-step t, $\mathbf{L}$ the forward weighted adjacency matrix and $\mathbf{R}$ the recurrent weighted adjacency matrix, $\xi^{(t)}$ the output vector of nodes at time-step t, and b_i the bias of node i. The algorithm is adjusted such that we input multiple time-steps, for which it calculates the final time-step output. The recurrent matrix may only make connections with the previous input and with hidden nodes, and is restricted from accessing the output of the previous time-step. In Algorithm 5.1, the RecurrentNG forward-propagation algorithm is presented; the red markings show the difference with Algorithm 4.1.

Algorithm 5.1. The RecurrentNG forward-propagation algorithm.

$input: n$: input size,
m : number of hidden nodes,
o : output size,
t_{total} : the total number of time-steps,
ω : output function,
$\mathbf{x}^{(t)}$: input vector of length n at timestep t,
$\mathbf{b}$: bias vector of length $n+m+o$,
$\mathbf{L}$: lower triangular weighted adjacency matrix,
$\mathbf{R}$: weighted adjacency matrix,
$output: \mathbf{y}^{(t)}$: output vector of length o at timestep t,
$\xi^{(t)}$: output state vector for all nodes with length $n+m+o$, at timestep t
$\mathbf{z}^{(t)}$: input state vector for all nodes with length $n+m+o$ at timestep t.

function ForwardPropagate
  let $\xi^{(0)} = \mathbf{0}$ of size $n+m+o$
  for $t = 1$ **to** t_{total} **do**
    let $\xi^{(t)} = \mathbf{0}$ of size $n+m+o$
    let $\xi^{(t)}[:n] = \mathbf{x}$

    let $\mathbf{z}^{(t)} = \mathbf{0}$ of size $n+m+o$.
    let $\mathbf{z}^{(t)}[:n] = \mathbf{x}$

    for $i = n$ **to** $n+m$ **do**
      compute $x = \langle \mathbf{L}_{i,*}, \xi^{(t)} \rangle + \langle \mathbf{R}_{i,*}, \xi^{(t-1)} \rangle + b_i$.
      Let $\xi_i^{(t)} = f_i(x)$
      Let $z_i^{(t)} = x$

    for $i = n+m$ **to** $n+m+o$ **do**
      compute $z_i^{(t)} = \langle \mathbf{L}_{i,*}, \xi^{(t)} \rangle + \langle \mathbf{R}_{i,*}, \xi^{(t-1)} \rangle + b_i$

    compute $\xi^{(t)}[n+m : n+m+o] = \omega(\mathbf{z}^{(t)}[n+m : n+m+o])$
    compute $\mathbf{y}^{(t)} = \xi^{(t)}[n+m : n+m+o]$
  end for
end function

Back-propagation

In the back-propagation we can follow the same method as we had for NG, with two differences: we need to compute a forward pass on all time steps first, and we need to modify the gradient and Lagrange multiplier formulas. Since the output of the graph is not connected via the recurrent graph, the gradient and Lagrange multipliers for these nodes are not modified.

The first change we need to accommodate is the gradient calculation for the $\mathbf{L}$ matrix which follows from recalculating the derivatives:

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{L}_{*,i}} &= \frac{\partial J}{\partial z_i^{(t)}} \frac{\partial z_i^{(t)}}{\partial \mathbf{L}_{*,i}} \\ &= \lambda_i^{(t)} \left(\frac{\partial f_i}{\partial \mathbf{L}_{*,i}} \left(\mathbf{L}_{*,i} \xi^{(t)} + \mathbf{R}_{*,i} \xi^{(t-1)} + b_i \right) \right) \\ &= \lambda_i^{(t)} \xi^{(t)} f_i'(\mathbf{L}_{*,i} \xi^{(t)} + \mathbf{R}_{*,i} \xi^{(t-1)} + b_i).\end{aligned}$$

Similarly we can calculate the derivatives of the $\mathbf{R}$, which follow the same structure

$$\begin{aligned}\frac{\partial J}{\partial \mathbf{R}_{*,i}} &= \frac{\partial J}{\partial z_i^{(t)}} \frac{\partial z_i^{(t)}}{\partial \mathbf{R}_{*,i}} \\ &= \lambda_i^{(t)} \left(\frac{\partial f_i}{\partial \mathbf{R}_{*,i}} \left(\mathbf{L}_{*,i} \xi^{(t)} + \mathbf{R}_{*,i} \xi^{(t-1)} + b_i \right) \right) \\ &= \lambda_i^{(t)} \xi^{(t-1)} f_i'(\mathbf{L}_{*,i} \xi^{(t)} + \mathbf{R}_{*,i} \xi^{(t-1)} + b_i).\end{aligned}$$

The derivative for the bias remains fairly untouched after accounting for the modified input of the activation function

$$\begin{aligned}\frac{\partial J}{\partial b_i} &= \frac{\partial J}{\partial z_i^{(t)}} \frac{\partial z_i^{(t)}}{\partial \mathbf{b}_i} \\ &= \lambda_i^{(t)} \left(\frac{\partial f_i}{\partial b_i} \left(\mathbf{L}_{*,i} \xi^{(t)} + \mathbf{R}_{*,i} \xi^{(t-1)} + b_i \right) \right) \\ &= \lambda_i^{(t)} f_i'(\mathbf{L}_{*,i} \xi^{(t)} + \mathbf{R}_{*,i} \xi^{(t-1)} + b_i).\end{aligned}$$

Finally we need to redefine the calculation of the Lagrange multiplier using Equation (3.13). Since the recurrent part is added, we need to add the corresponding terms to the summation of the outgoing connections in the node. This adds an extra dependency on the Lagrange multipliers from the next time-step. Substituting in the formula gives us

$$\begin{aligned}\lambda_i^{(t)} &= \frac{\partial J}{\partial z_i^{(t)}} \\ &= \sum_{j=0}^{m+n+o} \lambda_j^{(t)} \frac{\partial f_i}{\partial \mathbf{z}_i^{(t)}} \left(\mathbf{L}_{*,i} \xi^{(t)} + \mathbf{R}_{*,i} \xi^{(t-1)} + b_i \right) + \\ &\qquad \lambda_j^{(t+1)} \frac{\partial f_i}{\partial z_i^{(t)}} \left(\mathbf{L}_{*,i} \xi^{(t+1)} + \mathbf{R}_{*,i} \xi^{(t)} + b_i \right) \\ &= \sum_{j=0}^{m+n+o} \lambda_j^{(t)} \mathbf{L}_{j,i} + \lambda_j^{(t+1)} \mathbf{R}_{j,i}.\end{aligned}$$

When we reassemble the separate parts, we obtain the new back-propagation algorithm, which is given in Algorithm 5.2. Techniques such as dropout

and weight normalization are calculated separately for both $\mathbf{L}$ and $\mathbf{R}$ in our implementation.

Algorithm 5.2. The RecurrentNG back-propagation algorithm.

input : n : input size, m : number of hidden nodes,
o : output size, J : objective function, ω : output function,
t_{total} : the total number of time-steps,
$\mathbf{x}$: input vector of length n,
$\hat{\mathbf{y}}$: true output vector of length o,
$\mathbf{b}$: bias vector of length $n + m + o$,
$\mathbf{L}$: lower triangular weighted adjacency matrix,
output : $\nabla\mathbf{b}$: the gradient of the bias vector,
$\nabla\mathbf{L}$: the gradient of the weight matrix.

compute time steps $[\mathbf{y}, \xi, \mathbf{z}] =$ ForwardPropagate().
let $\nabla\mathbf{b} = \mathbf{0}$
let $\nabla\mathbf{L} = \mathbf{0}$
for $t = t_{\text{total}}$ **to** 1 **do**
  let $\lambda^{(t)} = \mathbf{0}$
  compute $\mathbf{h} = J'(\xi^{(t)}[n + m : n + m + o], \hat{\mathbf{y}}^{(t)})$.
  let $\lambda^{(t)}[n + m : n + m + o] = h$
  let $\nabla\mathbf{b}[n + m : n + m + o] = h$
  for $j = n + m + o$ **to** $n + m$ **do**
    for $\{i \in [1, n + m + o] | \mathbf{L}_{j,i} \neq 0\}$ **do**
      compute $\nabla\mathbf{L}_{j,i} = \nabla\mathbf{L}_{j,i} + \lambda_j^{(t)}\xi_i^{(t)}$

  for $j = n + m$ **to** n **do**
    let $\sigma = 0$.
    for $i = 1$ **to** $n + m + o$ **do**
      compute $\sigma = \sigma + \lambda_i^{(t)}\mathbf{L}_{i,j} + \lambda_i^{(t+1)}\mathbf{R}_{i,j}$

    compute $\lambda_j^{(t)} = \sigma f_i'\left(z_j^{(t)}\right)$
    for $\{i \in [1, n + m + o] | \mathbf{L}_{j,i} \neq 0\}$ **do**
      compute $\nabla\mathbf{L}_{j,i} = \nabla\mathbf{L}_{j,i} + \lambda_j^{(t)}\xi_i^{(t)}$.

    for $\{i \in [1, n + m + o] | \mathbf{R}_{j,i} \neq 0\}$ **do**
      compute $\nabla\mathbf{R}_{j,i} = \nabla\mathbf{R}_{j,i} + \lambda_j^{(t)}\xi_i^{(t-1)}$.

    let $\nabla b_j = \nabla b_j + \lambda_j^{(t)}$

6 Experiments

In this chapter we give both the experiments on the benchmark data sets, and the experiments on time series data. For the benchmark data sets we use two classification problems, and for the time series we first try sinusoid functions, after which we run the algorithms on stock market data.

6.1 Classification

We start this section by giving the definitions of two benchmark problems. The first problem is the classification of the MNIST data set. This data set contains handwritten digits of 0 to 9 in a binary format [LCB10]. There is a training set of 60000 examples, and a test set of 10000 examples. The data set is constructed from the larger data sets MNIST Special Database 1 and Special Database 3, where the former were handwritten digits collected from high-school students, and the latter from Census Bureau employees. The data is already preprocessed such that the digits are centred in a gray-scale image of 28 by 28 pixels. Some implementations use a technique called data augmentation, this creates variations of the original training images such that we have newer and harder images to train with [WP17]. However, we will not use these techniques since we only want to measure the performance of the NG, but do not aim to have better performance to more specialized algorithms. Due to the simplicity of using MNIST in benchmarking models, it has become increasingly popular. A drop-in replacement, Fashion-MNIST, that uses the exact same file format contains images from 70000 fashion products from 10 different categories [XRV17], is our second data set. Fashion-MNIST is more challenging than MNIST and thus holds more real-world value as a benchmark.

The implementation of the algorithm is done using C++, with the help of the linear algebra library Armadillo [San16]. Armadillo internally uses the Intel MKL library for both LAPACK and BLAS operations. The code for the forward and recurrent ANN can be found in Appendix C. With this implementation we test the performance on both the MNIST and Fashion-MNIST data sets.

MNIST

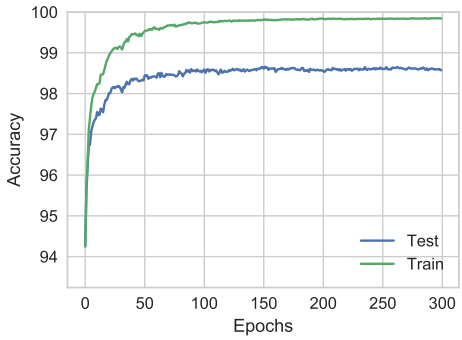
The MNIST data set represents a classification problem, so we use the softmax output function we described before with length 10. Before we give the samples to the network, we first normalize the input by subtracting the mean of the input values, and dividing by the variance of the data set. We run the NG algorithms with several configurations of number of nodes so we can examine the performance. In Figure 6.1 and Figure 6.2 we give the results of a training on the MNIST data set for 200 and 300 nodes respectively. In Appendix B are the results of more configurations. For each configuration, we empirically determined the hyper-parameters for optimal train accuracy and a small generalization error. For all configurations we notice that there is a substantial generalization error, of at least 1%. However, the training accuracy for both converge to nearly 100%: the highest observed test accuracy was 98.86% for 300 nodes and 98.66% for 200 nodes.

For comparison, the highest reported score on the MNIST database [YCC98][1] for non-convolution ANNs is reported on a deep convex net [DY11] was a 99.17% accuracy on the test database. The second largest accuracy, that does not use data augmentation, such as affine or elastic distortions, or ensemble methods, reports a 98.6% accuracy with 800 hidden nodes. Furthermore, we succeed in reducing the number of connections required to around 9.4% of the original for the 300 nodes and to around 21% for 200 nodes. This reduction also shows in the time required for each epoch, shaving a whole 6 seconds per epoch off from the time for the 200 node configuration and 13 seconds per epoch for the 300 node configuration. By using the sparsity of the problem, we can greatly decrease the training time. This sparsity is also clearly demonstrated in the spy plot of non-zero values for both configurations, wherein the connections with the input are only defined for the middle of the images where the written information is defined.

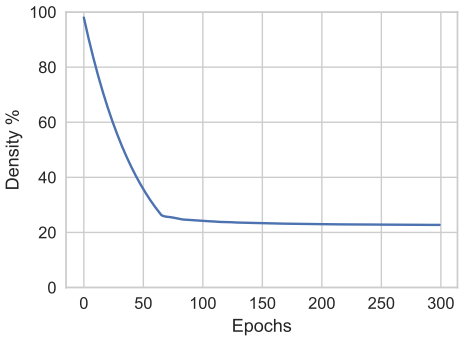
Fashion-MNIST

We take the best performing configuration on the MNIST data set and use that configuration on the Fashion-MNIST data set. The results can be seen in Figure 6.3. The network is reduced to around 27% of its original connections. The training accuracy reaches 98.6% while the test accuracy stays around 90.3%. This is competitive with convolutional architectures [XRV17]. We notice that the resulting spy plot is much denser than in the original MNIST data set, which indicates that more information of the input images is used.

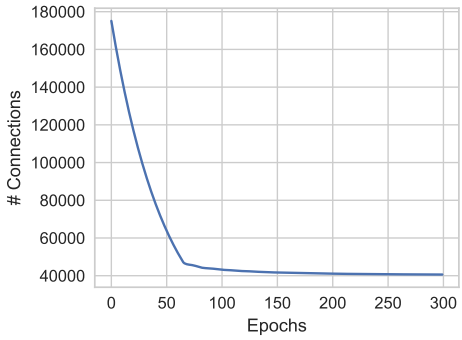
[1] Note that this database is not complete.



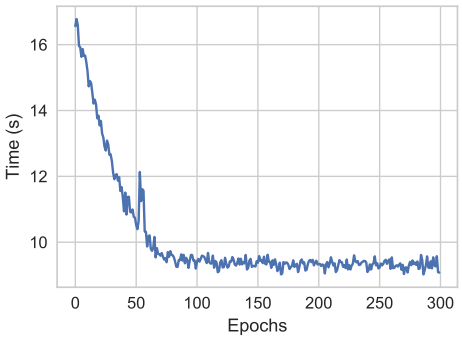
(a) The accuracy of the predictions for both training and test data.



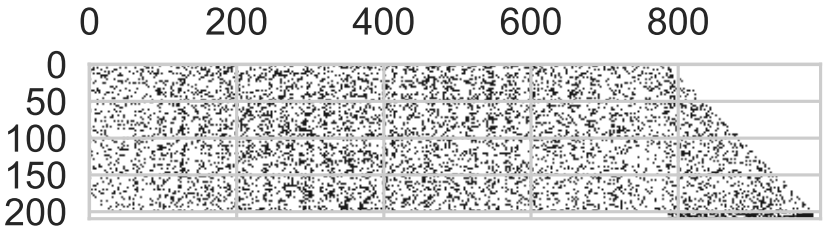
(b) The density of the sparse matrix compared to the initialization plotted over the number of epochs.



(c) The number of connections in each epoch.

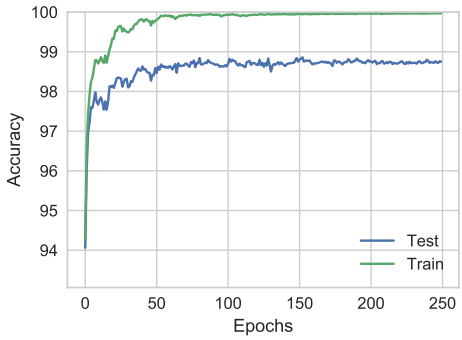


(d) The time each epoch took in seconds.

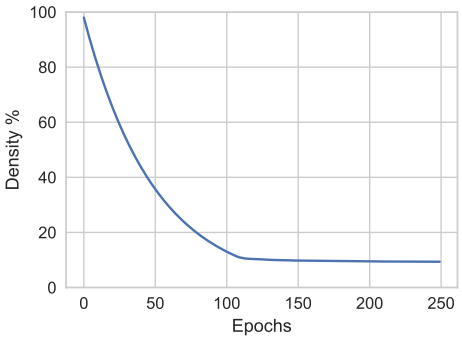


(e) The spy plot of the resulting weights.

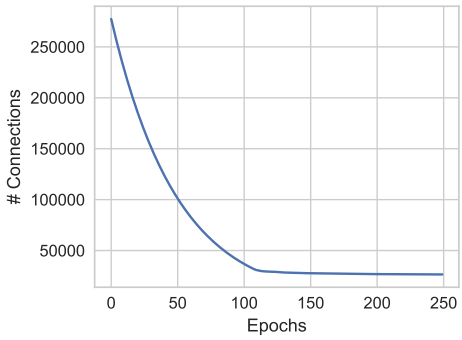
Figure 6.1: A NG training with 200 nodes on MNIST.



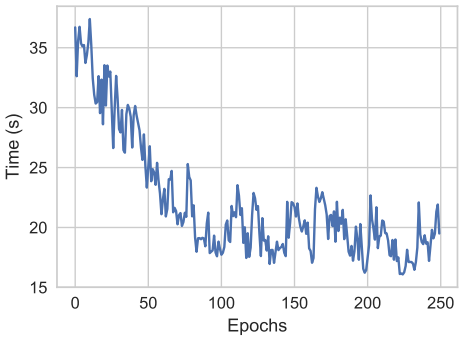
(a) The accuracy of the predictions for both training and test data.



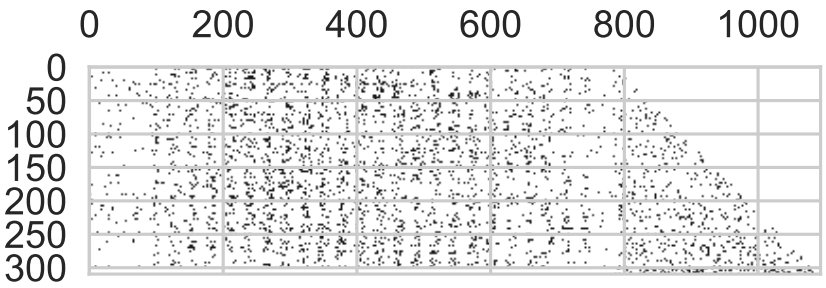
(b) The density of the sparse matrix compared to the initialization plotted over the number of epochs.



(c) The number of connections in each epoch.

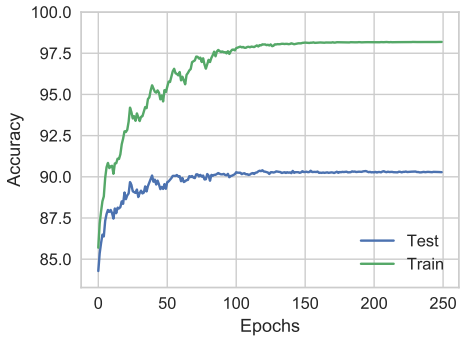


(d) The time each epoch took in seconds.

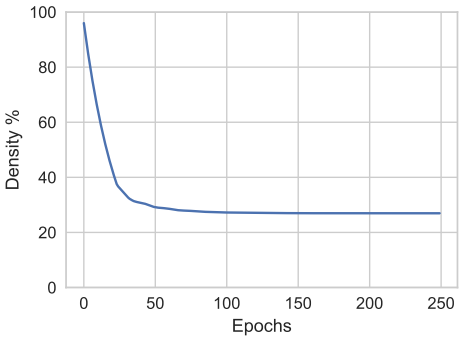


(e) The spy plot of the resulting weights.

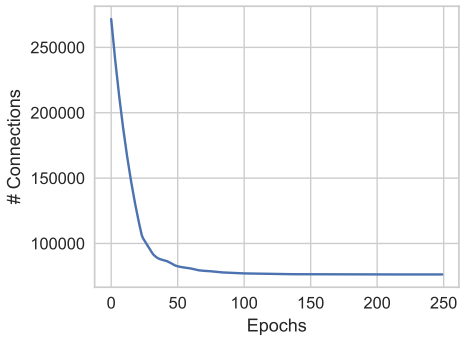
Figure 6.2: A NG training with 300 nodes on MNIST.



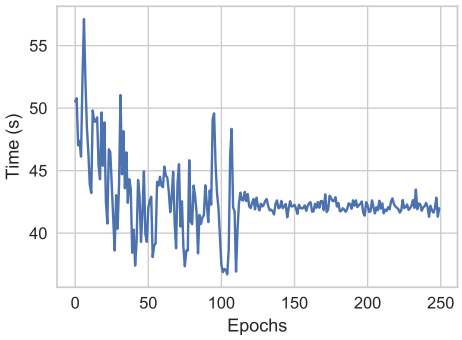
(a) The accuracy of the predictions for both training and test data.



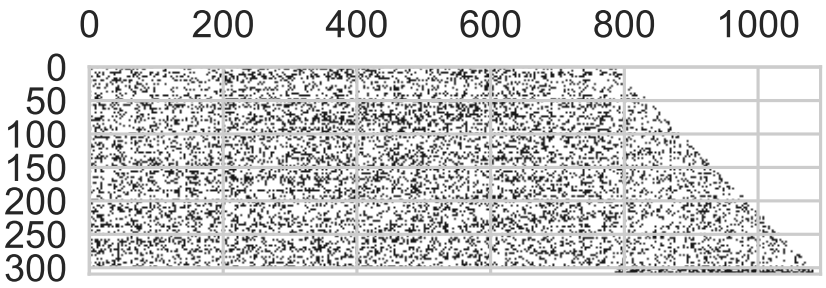
(b) The density of the sparse matrix compared to the initialization plotted over the number of epochs.



(c) The number of connections in each epoch.



(d) The time each epoch took in seconds.



(e) The spy plot of the resulting weights.

Figure 6.3: A NG training with 300 nodes on Fashion-MNIST

6.2 Time Series

Predicting time series is a completely different problem than the MNIST classification problem. We change the objective function to Sum Squared Error (SSE), which is defined as

$$\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

where y_i is the true output, and $\hat{y}_i$ the predicted output. To make sure the network performs as expected, we first try the NG algorithm on functions with a clearly defined input and output. For this we choose variations on a sinusoid function and try to predict its output with varying inputs.

6.2.1 Sinusoid

We first use the simple sine function, which defines the following continuous time series

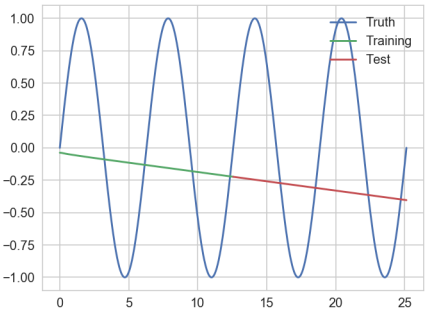
$$y = \sin(x).$$

We use the domain $x \in [0, 4\pi[$ as the training data set, and $x \in [4\pi, 8\pi[$ as the validation data set. For the first experiment we input the normalized values (divided by 4π), and try to predict the validation data set with only the given x values to obtain a discrete time series. The input domain is divided in 400 samples, and so is the validation domain, since it has the same length. We notice that over the training epochs, the algorithm learns to predict the correct y value for the given x. However, when we look at the out-of-sample prediction performance, we see that it does not learn the periodicity of the function. The resulting spy plot is dense in the first nodes, and gets sparser near the end.

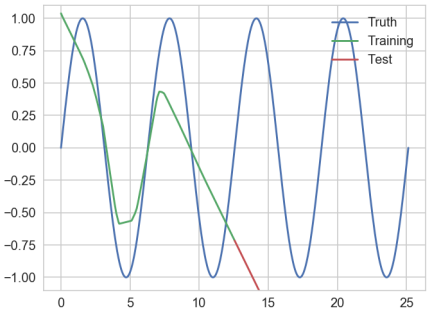
In the next experiment, we double the domain of both the training and test set to $x \in [0, 8\pi[$ and $x \in [8\pi, 16\pi[$ respectively. In Figure 6.5 we see that the increase of the domains did not make the network learn the periodicity of the function. In the last sinusoid experiment we define the function we want to predict to

$$y = \frac{\sin(x)}{60 + x}.$$

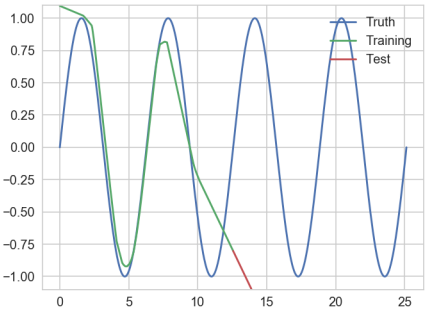
This function has a decreasing amplitude over time, which makes it harder for the network to learn. Instead of giving the values of x to the network, we now give the preceding 200 y values. This method is called a sliding window input. In Figure 6.6 we see that it now learns the periodicity correctly, but fails to properly predict the amplitude of the function. This can be improved by either increasing the number of samples the network can learn from, or by increasing the number of epochs.



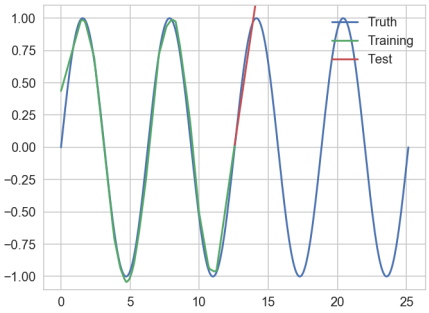
(a) Accuracy at epoch 1.



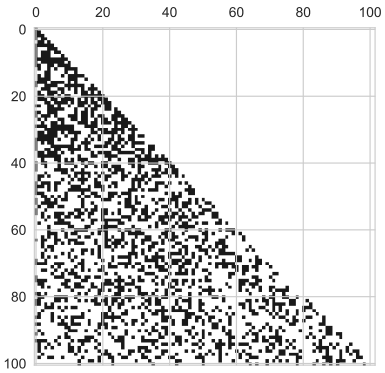
(b) Accuracy at epoch 217.



(c) Accuracy at epoch 577.

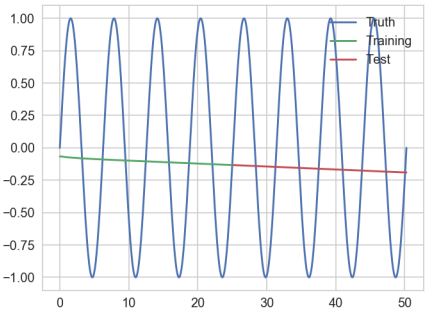


(d) Accuracy at epoch 1441.

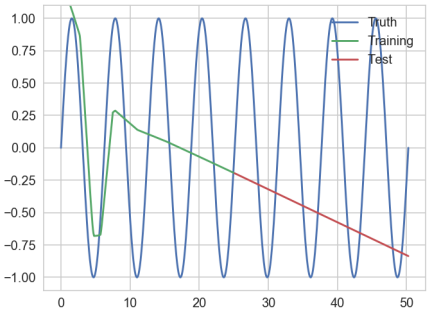


(e) The spy plot of the resulting weights.

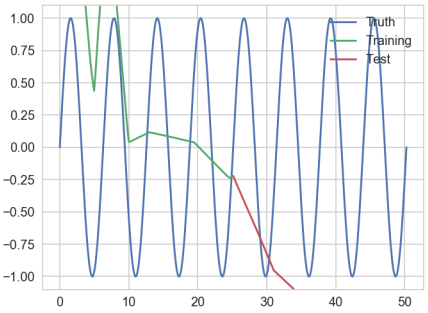
Figure 6.4: A NG training with 100 nodes on the sine wave function.



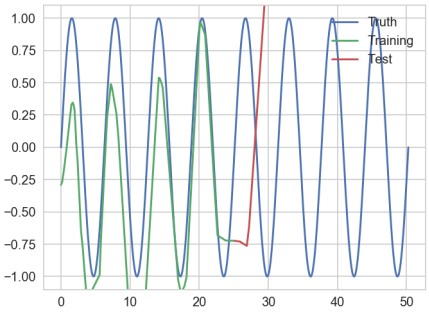
(a) Accuracy at epoch 1.



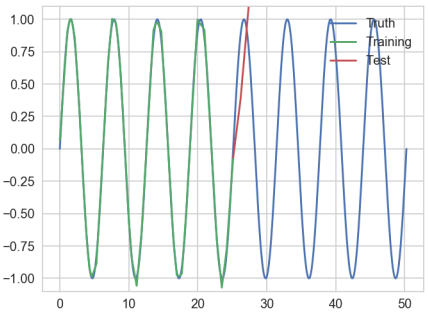
(b) Accuracy at epoch 1441.



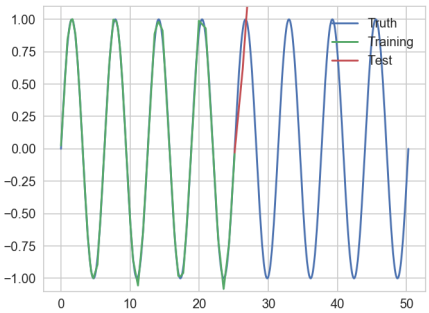
(c) Accuracy at epoch 2881.



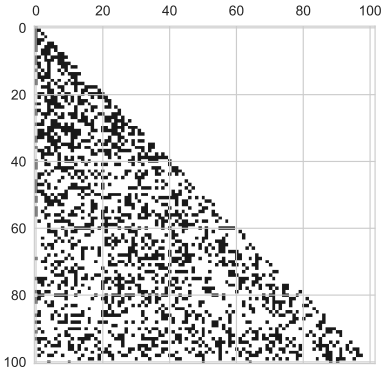
(d) Accuracy at epoch 4321.



(e) Accuracy at epoch 8641.

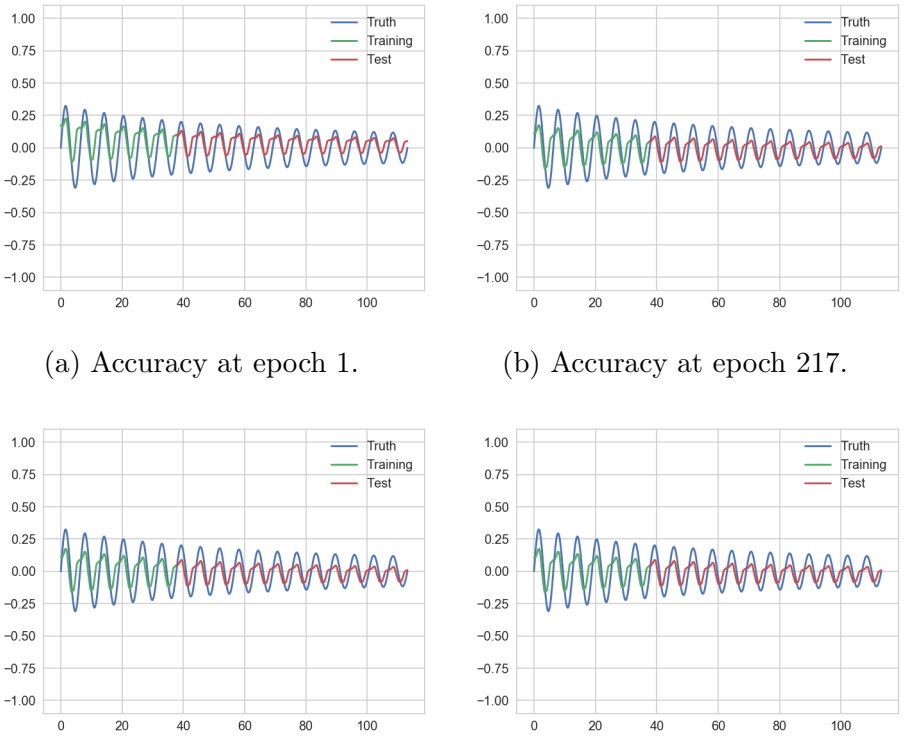


(f) Accuracy at epoch 19801.



(g) The spy plot of the resulting weights.

Figure 6.5: A NG training with 100 nodes on the sine wave function with a larger domain.



(a) Accuracy at epoch 1.

(b) Accuracy at epoch 217.

(c) Accuracy at epoch 577.

(d) Accuracy at epoch 1441.

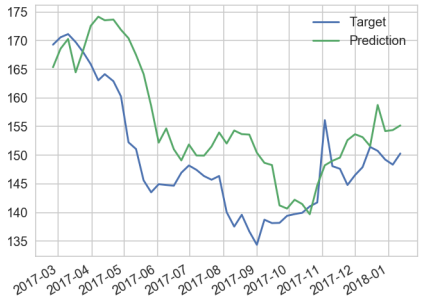
Figure 6.6: A NG training with 20 nodes on a sinusoid with a sliding window.

6.2.2 IBM Stock

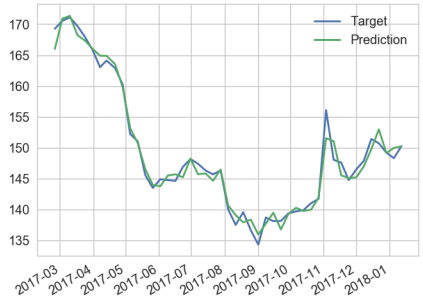
Now that we have an indication of the performance of our NG on time series data, we will use the algorithm to predict real stock market values. We use weekly data, which consist of four separate time series with: open, high, low, and close value for the whole week. The open value resembles the value of the stock when the market opens that week, close holds the value when the market closes in that week, high and low hold the maximum and minimum value respectively in that week. For our experiments we choose to use the IBM stock data, from 2000-06-02 to 2017-02-17 as training domain, 2017-02-24 to 2018-01-12 as validation domain, and 2018-01-19 to 2018-05-21 as test domain. Both the test and validation domain will remain as unseen data, where the validation domain is used to verify the generalization error, and the test domain as the prediction of values that are contemporary. First we train the NG algorithm with 25 nodes, and allow it to use a window of 19 previous data points. In Figure 6.7 we see that the algorithm is able to predict the training data very well, due to its high overlap (SSE of around 1.5). When we look at the validation data we see that over the epochs the predictions gets better, and eventually the algorithm is able to predict the opening value of the stock with a very good accuracy (SSE of around 1.9).

In Figure 6.8 we show the prediction of the test and validation data for close, high, and low. It is clear that the model is able to predict the open value much better than the other time series, which can be explained due to the value of the following open and close values having the same direction. However, they are not necessarily the same. In the other time series, the algorithm produces a value near the last value with a slight damping, which creates a lagged time series. While this type of prediction still holds a relevant value, it is not as useful as the prediction on the open time series. The next experiment will use the predictions of the previously predicted values as input in the new calculation. This means that instead of predicting the next value of the time series, we can also predict the following values to that. In Figure 6.9, we show the results of this experiment. It is clear that this is a much harder test case for the algorithm, and it only produces values that are approximate to the real values. What is interesting to note, is that the model captures some of the periodicity of the model, and the values do not differ too much from the true value.

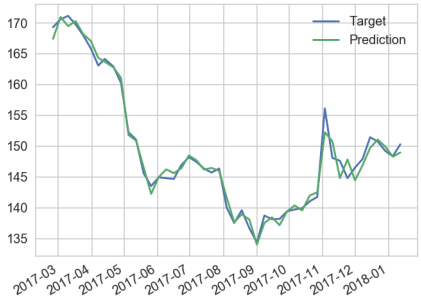
(a) The prediction of open training data at epoch 2809.



(b) The prediction of open validation data at epoch 1.

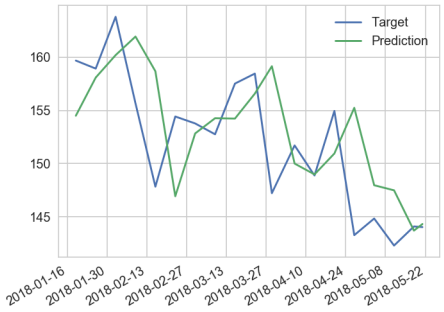


(c) The prediction of open validation data at epoch 1081.

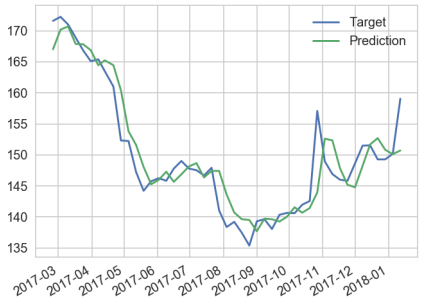


(d) The prediction of open validation data at epoch 2809.

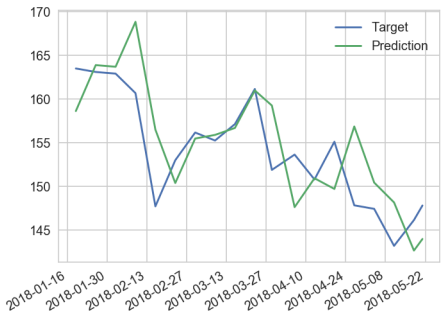
Figure 6.7: A NG training with 25 nodes on IBM stock with a sliding window of 19 previous data points.



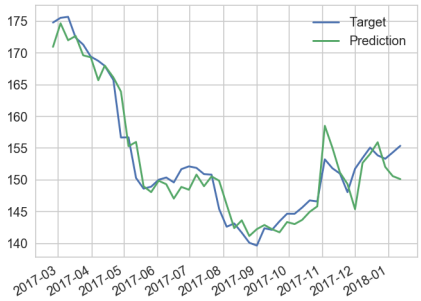
(a) The prediction of close test data at epoch 2809.



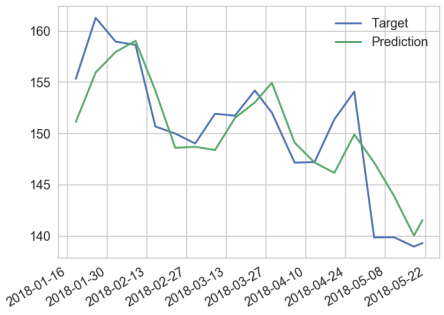
(b) The prediction of close validation data at epoch 2809.



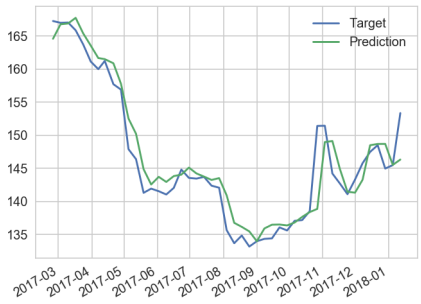
(c) The prediction of high test data at epoch 2809.



(d) The prediction of high validation data at epoch 2809.

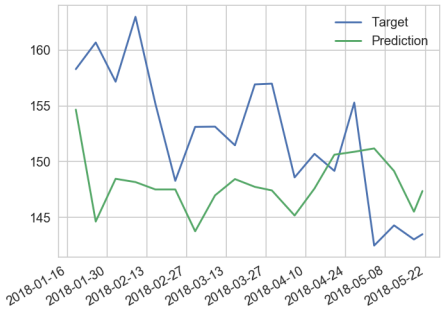


(e) The prediction of low test data at epoch 2809.

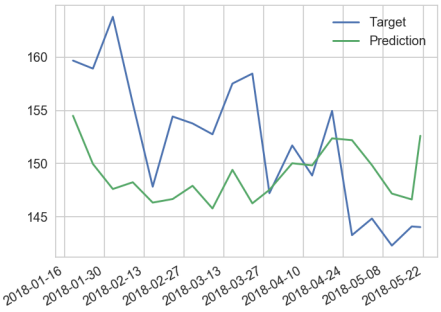


(f) The prediction of low validation data at epoch 2809.

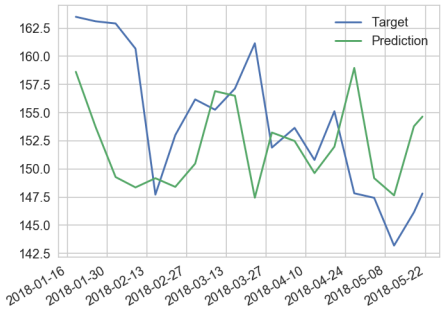
Figure 6.8: A NG training with 25 nodes on IBM stock with a sliding window of 19 previous data points.



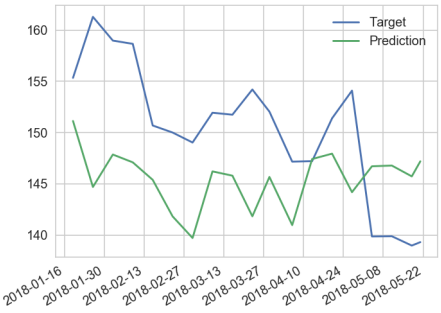
(a) The prediction of open test data at epoch 2809.



(b) The prediction of close test data at epoch 2809.



(c) The prediction of high test data at epoch 2809.



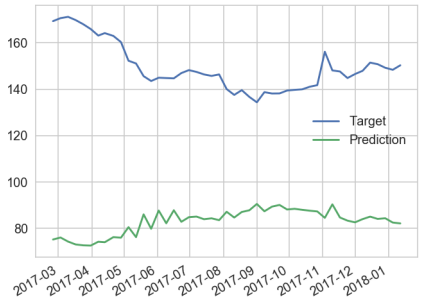
(d) The prediction of low test data at epoch 2809.

Figure 6.9: A NG training with 25 nodes on IBM stock with a sliding window of 19 previous data points, that uses its own output.

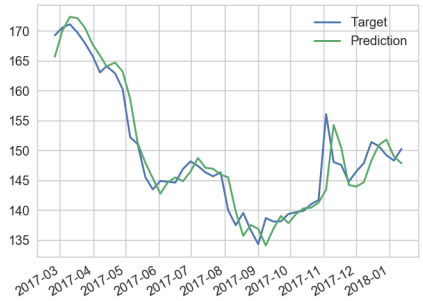
6.2.3 Recurrent IBM Stock

Finally, we train the recurrent version of the algorithm on the same problem. Instead of using the 19 values for all four time series as input, we use an input of only four data points over 19 time steps. In Figure 6.10 and Figure 6.11 we can see that the recurrent algorithm yields lower performance (SSE 2.8 on test, and 7.4 on validation data) and training is more unstable. Finally when we let the algorithm predict based on its own output we obtain the results as seen in Figure 6.12. We notice that this model converges to a trend, but does not yield a meaningful forecast. This problem of training RNNs with gradient methods can be explained by vanishing and exploding gradients [PMB13; BSF94], see Theorem 3. Solutions proposed to address these problems are LSTM networks [HS97; GSC99] and more recently GRU networks [Cho+14], but these topologies are hard to generalize to run on any DAG.

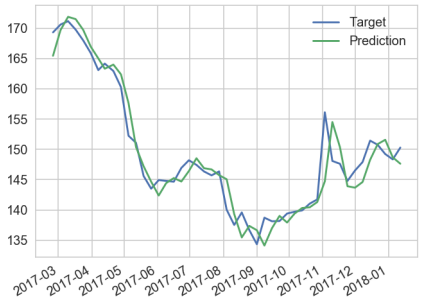
(a) The prediction of open training data at epoch 1369.



(b) The prediction of open validation data at epoch 1.

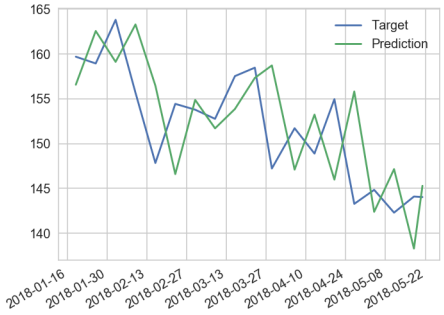


(c) The prediction of open validation data at epoch 505.

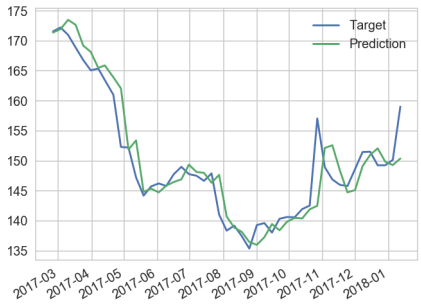


(d) The prediction of open validation data at epoch 1369.

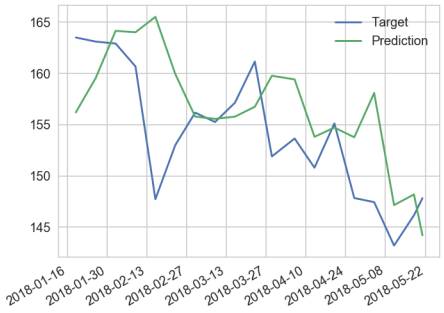
Figure 6.10: A RecurrentNG training with 5 nodes on IBM stock with 19 time steps.



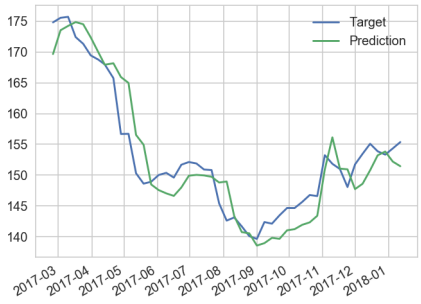
(a) The prediction of close test data at epoch 1368.



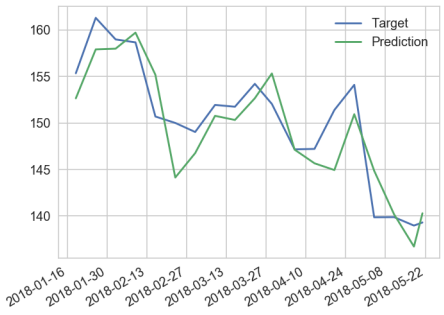
(b) The prediction of close validation data at epoch 1368.



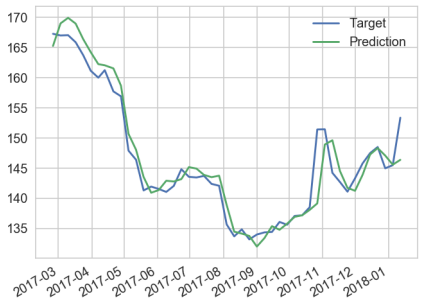
(c) The prediction of high test data at epoch 1368.



(d) The prediction of high validation data at epoch 1368.

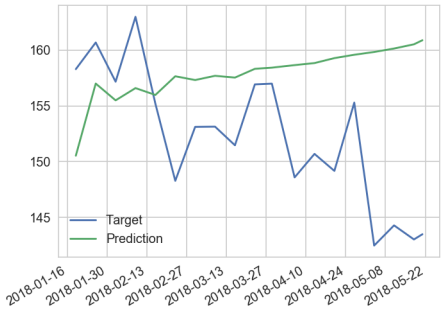


(e) The prediction of low test data at epoch 1368.

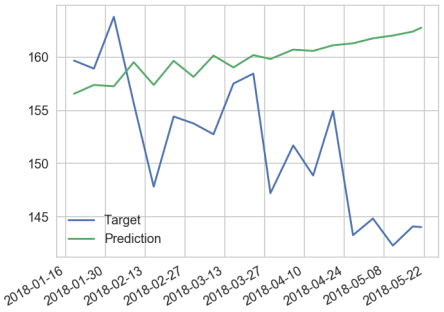


(f) The prediction of low validation data at epoch 1368.

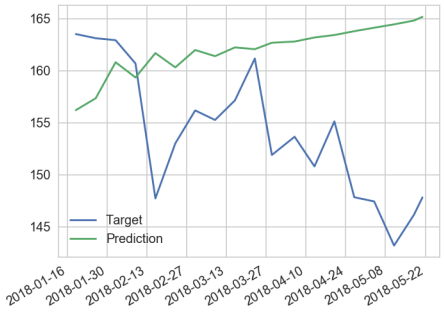
Figure 6.11: A RecurrentNG training with 5 nodes on IBM stock with 19 time steps.



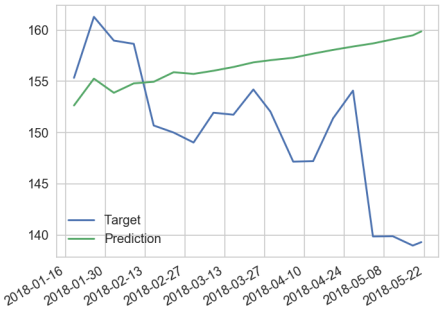
(a) The prediction of open test data at epoch 1368.



(b) The prediction of close test data at epoch 1368.



(c) The prediction of high test data at epoch 1368.



(d) The prediction of low test data at epoch 1368.

Figure 6.12: A RecurrentNG training with 5 nodes on IBM stock with 19 time steps, that uses its own output.

7 Conclusion

In this work we proposed a fundamental advancement of time series prediction through the use of ANNs. Due to this, we overcome the major problems in traditional methods, where it was needed to have expert knowledge of the time series you wanted to forecast. With NGs the model is based on the training data fed to the model, and not the knowledge of the person that makes the forecasts. This makes forecasting time series more flexible, easier to produce in large quantities, and more accurate. We showed how you can successfully use a generic DAG as a ANN for both prediction and training. Furthermore, the performance for the MNIST data set was competitive with more specialised networks for image classification, which was again shown on the Fashion-MNIST data set.

When we used the NG algorithm for time series data, we showed that it was able to learn to predict the open value of the IBM stock value nearly perfectly. While it had more trouble with the close, low and high values, they were still close to the true values. It is clear that the NG is able to predict multivariate time series, but the performance can still be improved in many ways. One of such improvements can be made by allowing it to see more stock symbols, and let it learn the relations between companies and general trends in the stock market.

Another improvement can be made by allowing it to forecast not only the next value of the time series, but by allowing it to predict a window. This should yield more accurate predictions over more time than just the next data point. The RecurrentNG algorithm proved to be less useful, since it was harder to train. Its biggest advantage over the NG algorithm is its capability to use any sequence length as input, whereas the NG has to be trained for specific lengths. In the traditional layered topology, the RNN usually has better performance due to having memory of previous calculations due to the stored state. This advantage, however, was less obvious in the RecurrentNG and NG algorithms, since they are allowed to connect to all previous outputs (in the same calculation) by default.

While the current learning algorithm for NG already make much use of the sparsity in connections, we have only removed the connections we were certain to have almost no effect on the results. More sparsity can be

introduced by making either the learning sparsification more greedy, or by developing an additional algorithm that improves the sparsity after the weights have been learned. This can be pushed even further, by increasing the number of nodes that the NG has access to when the objective function has converged. This can be done repeatedly until the performance of the network stops improving.

Acronyms

ANN Artificial Neural Network 1, 7–14, 19, 36, 43, 44, 58
AR Autoregressive 2, 5, 6
ARIMA Autoregressive Integrated Moving-Average 6, 7
ARMA Autoregressive Moving-Average 2, 6

BPTT Back-propagation Through Time 14

DAG Directed Acyclic Graph 12–17, 19, 39, 54, 58

FNN Feedforward Neural Network 8–11, 13, 14, 17, 19

GARCH Generalized Autoregressive Conditional Heteroscedasticity 2, 6
GRU Gated Recurrent Units 11, 54

LSTM Long Short-Term Memory 11, 54

MA Moving-Average 5, 6
MGARCH Multivariate Generalized Autoregressive Conditional Heteroscedasticity 6
MLP Multi Layer Perceptron 9

NG Neural Graphs iii, 19, 22, 24, 25, 35, 36, 38–40, 43–54, 58, 59, 69–71

OLS Ordinary Least Squares ii, 2, 3, 67

RecurrentNG Recurrent Neural Graphs 39, 40, 42, 55–58
ReLU Rectified Linear Unit 10
RNN Recurrent Neural Network 8, 11, 13–15, 17, 54, 58

SARIMA Seasonal Autoregressive Integrated Moving-Average 6
SARMA Seasonal Autoregressive Moving-Average 6
SGD Stochastic Gradient Descent 14, 24, 28–30
SSE Sum Squared Error 48

VAR Vector Autoregression 6
VARIMA Vector Autoregressive Integrated Moving-Average 6
VARMA Vector Autoregressive Moving-Average 6

Bibliography

[AA13] Adhikari, Ratnadip and Agrawal, RK. "An introductory study on time series modeling and forecasting". In: *arXiv preprint arXiv:1302.6613* (2013).

[Ans73] Anscombe, Francis J. "Graphs in statistical analysis". In: *The American Statistician* vol. 27. no. 1 (1973), pp. 17–21.

[BBC17] Bourely, Alfred, Boueri, John Patrick, and Choromonski, Krzysztof. "Sparse Neural Networks Topologies". In: *arXiv preprint arXiv:1706.05683* (2017).

[BG09] Braun, Jürgen and Griebel, Michael. "On a constructive proof of Kolmogorov's superposition theorem". In: *Constructive approximation* vol. 30. no. 3 (2009), p. 653.

[BJ70] Box, George EP and Jenkins, Gwilym M. *Time Series Analysis: Forecasting and Control*. 1st ed. Holden–Day, 1970.

[Bol86] Bollerslev, Tim. "Generalized autoregressive conditional heteroskedasticity". In: *Journal of econometrics* vol. 31. no. 3 (1986), pp. 307–327.

[Box+16] Box, George EP, Jenkins, Gwilym M, Reinsel, Gregory C, and Ljung, Greta M. *Time series analysis: forecasting and control*. 5th ed. John Wiley & Sons, 2016.

[BSF94] Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* vol. 5. no. 2 (1994), pp. 157–166.

[Cho+14] Cho, Kyunghyun et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* vol. abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[Chu+14] Chung, Junyoung, Gulcehre, Caglar, Cho, KyungHyun, and Bengio, Yoshua. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555* (2014).

[CL11] Chen, Ling and Lai, Xu. "Comparison between ARIMA and ANN models used in short-term wind speed forecasting". In: *Power and Energy Engineering Conference (APPEEC), 2011 Asia-Pacific*. IEEE. 2011, pp. 1–4.

[Csá01] Csáji, Balázs Csanád. "Approximation with artificial neural networks". In: *Faculty of Sciences, Eötvös Loránd University, Hungary* vol. 24 (2001), p. 48.

[CSS16] Cohen, Nadav, Sharir, Or, and Shashua, Amnon. "On the expressive power of deep learning: A tensor analysis". In: *Conference on Learning Theory*. 2016, pp. 698–728.

[Cyb89] Cybenko, George. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals, and Systems (MCSS)* vol. 2. no. 4 (1989), pp. 303–314.

[Día+08] Díaz-Robles, Luis A et al. "A hybrid ARIMA and artificial neural networks model to forecast particulate matter in urban areas: The case of Temuco, Chile". In: *Atmospheric Environment* vol. 42. no. 35 (2008), pp. 8331–8340.

[DP02] Dufour, Jean-Marie and Pelletier, Denis. "Linear methods for estimating VARMA models with a Macroeconomic application". In: *2002 Proceedings of the Business and Economic Statistics Section of the American Statistical Association, Washington, DC* (2002), pp. 2659–2664.

[DY11] Deng, Li and Yu, Dong. "Deep Convex Network: A Scalable Architecture for Speech Pattern Classification". In: *Interspeech*. International Speech Communication Association, Aug. 2011. URL: https://www.microsoft.com/en-us/research/publication/deep-convex-network-a-scalable-architecture-for-speech-pattern-classification/.

[Gam17] Gamboa, John Cristian Borges. "Deep Learning for Time-Series Analysis". In: *arXiv preprint arXiv:1701.01887* (2017).

[GB10a] Glorot, Xavier and Bengio, Yoshua. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[GB10b] Grubb, Alexander and Bagnell, J Andrew. "Boosted Backpropagation Learning for Training Deep Modular Networks." In: *ICML*. 2010, pp. 407–414.

[GBC16] Goodfellow, Ian, Bengio, Yoshua, and Courville, Aaron. *Deep learning*. MIT press, 2016.

[GSC99] Gers, Felix A, Schmidhuber, Jürgen, and Cummins, Fred. "Learning to forget: Continual prediction with LSTM". In: (1999).

[Har90] Harvey, Andrew C. *Forecasting, structural time series models and the Kalman filter*. Cambridge University Press, 1990.

[Hay04] Haykin, Simon. "A comprehensive foundation". In: *Neural Networks* vol. 2. no. 2004 (2004), p. 41.

[He+15] He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[He+16] He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[HS97] Hochreiter, Sepp and Schmidhuber, Jürgen. "Long short-term memory". In: *Neural computation* vol. 9. no. 8 (1997), pp. 1735–1780.

[HSW89] Hornik, Kurt, Stinchcombe, Maxwell, and White, Halbert. "Multilayer feedforward networks are universal approximators". In: *Neural networks* vol. 2. no. 5 (1989), pp. 359–366.

[IS15] Ioffe, Sergey and Szegedy, Christian. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[KB14] Kingma, Diederik and Ba, Jimmy. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[Koh+96] Kohzadi, Nowrouz, Boyd, Milton S., Kermanshahi, Bahman, and Kaastra, Iebeling. "A comparison of artificial neural network and time series models for forecasting commodity prices". In: *Neurocomputing* vol. 10. no. 2 (1996). Financial Applications, Part I, pp. 169–181. ISSN: 0925-2312. DOI: https://doi.org/10.1016/0925-2312(95)00020-8. URL: http://www.sciencedirect.com/science/article/pii/0925231295000208.

[Kol63] Kolmogorov, Andrei Nikolaevich. "On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition". In: *Translations American Mathematical Society* vol. 2. no. 28 (1963), pp. 55–59.

[Koo74] Koopmans, Lambert H. *The spectral analysis of time series*. Academic press, 1974.

[KOW04] Kihoro, J, Otieno, R, and Wafula, C. "Seasonal time series forecasting: A comparative study of ARIMA and ANN models". In: *AJST* vol. 5. no. 2 (2004).

[KSH12] Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[LBH15] LeCun, Yann, Bengio, Yoshua, and Hinton, Geoffrey. "Deep learning". In: *Nature* vol. 521. no. 7553 (2015), p. 436.

[LCB10] LeCun, Yann, Cortes, Corinna, and Burges, CJ. "MNIST handwritten digit database". In: *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist* vol. 2 (2010).

[LeC+88] LeCun, Yann, Touresky, D, Hinton, G, and Sejnowski, T. "A theoretical framework for back-propagation". In: *Proceedings of the 1988 connectionist models summer school*. CMU, Pittsburgh, Pa: Morgan Kaufmann. 1988, pp. 21–28.

[Lin+96] Lin, Tsungnan, Horne, Bill G, Tino, Peter, and Giles, C Lee. "Learning long-term dependencies in NARX recurrent neural networks". In: *IEEE Transactions on Neural Networks* vol. 7. no. 6 (1996), pp. 1329–1338.

[LKS91] LeCun, Yann, Kanter, Ido, and Solla, Sara A. "Second order properties of error surfaces: Learning time and generalization". In: *Advances in neural information processing systems*. 1991, pp. 918–924.

[Lug17] Lugt, B.J. van der. "Reductie van Neurale Netwerken met GraphBLAS". Bsc Thesis. Utrecht University, 2017.

[MHN13] Maas, Andrew L, Hannun, Awni Y, and Ng, Andrew Y. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. ICML*. Vol. 30. 1. 2013.

[Mik+14] Mikolov, Tomas et al. "Learning longer memory in recurrent neural networks". In: *arXiv preprint arXiv:1412.7753* (2014).

[MLP16] Mhaskar, Hrushikesh, Liao, Qianli, and Poggio, Tomaso A. "Learning Real and Boolean Functions: When Is Deep Better Than Shallow". In: *CoRR* vol. abs/1603.00988 (2016). arXiv: 1603.00988. URL: http://arxiv.org/abs/1603.00988.

[PMB13] Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. "On the difficulty of training recurrent neural networks". In: *International Conference on Machine Learning*. 2013, pp. 1310–1318.

[RHW+88] Rumelhart, David E, Hinton, Geoffrey E, Williams, Ronald J, et al. "Learning representations by back-propagating errors". In: *Cognitive modeling* vol. 5. no. 3 (1988), p. 1.

[RKK18] Reddi, Sashank J, Kale, Satyen, and Kumar, Sanjiv. "On the convergence of Adam and beyond". In: *International Conference on Learning Representations*. 2018.

[Ros61] Rosenblatt, Frank. *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Tech. rep. Cornell Aeronautical Lab inc Buffalo NY, 1961.

[San16] Sanderson, Conrad. *Armadillo C++ Linear Algebra Library*. June 2016. DOI: 10.5281/zenodo.55251. URL: https://doi.org/10.5281/zenodo.55251.

[SK16] Salimans, Tim and Kingma, Diederik P. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks". In: *Advances in Neural Information Processing Systems*. 2016, pp. 901–909.

[Smi17] Smith, Leslie N. "Cyclical learning rates for training neural networks". In: *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*. IEEE. 2017, pp. 464–472.

[Sri+14] Srivastava, Nitish et al. "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of machine learning research* vol. 15. no. 1 (2014), pp. 1929–1958.

[SS95] Siegelmann, Hava T and Sontag, Eduardo D. "On the computational power of neural nets". In: *Journal of computer and system sciences* vol. 50. no. 1 (1995), pp. 132–150.

[Sut+13] Sutskever, Ilya, Martens, James, Dahl, George, and Hinton, Geoffrey. "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning*. 2013, pp. 1139–1147.

[TH12] Tieleman, Tijmen and Hinton, Geoffrey. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* vol. 4. no. 2 (2012), pp. 26–31.

[Wal31] Walker, Gilbert. "On Periodicity in Series of Related Terms". In: *Proceedings of the Royal Society of London* vol. 131 (1931), pp. 518–532.

[WH86] Williams, DRGHR and Hinton, Geoffrey. "Learning representations by back-propagating errors". In: *Nature* vol. 323. no. 6088 (1986), pp. 533–538.

[WM03] Wilson, D Randall and Martinez, Tony R. "The general inefficiency of batch training for gradient descent learning". In: *Neural Networks* vol. 16. no. 10 (2003), pp. 1429–1451.

[Wol38] Wold, Herman. "A study in the analysis of stationary time series". PhD thesis. Almqvist & Wiksell, 1938.

[Wol96] Wolpert, David H. "The lack of a priori distinctions between learning algorithms". In: *Neural computation* vol. 8. no. 7 (1996), pp. 1341–1390.

[WP17] Wang, Jason and Perez, Luis. *The effectiveness of data augmentation in image classification using deep learning.* Tech. rep. Technical report, 2017.

[XRV17] Xiao, Han, Rasul, Kashif, and Vollgraf, Roland. "Fashion-MNIST: a novel image data set for benchmarking machine learning algorithms". In: *arXiv preprint arXiv:1708.07747* (2017).

[YCC98] Yann, LeCun, Corinna, Cortes, and Christopher, JB. "The MNIST database of handwritten digits". In: *URL http://yhann. lecun. com/exdb/mnist* (1998).

[Yul27] Yule, G Udny. "On a method of investigating periodicities in disturbed series, with special reference to Wolfer's sunspot numbers". In: *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character* vol. 226 (1927), pp. 267–298.

[Zei12] Zeiler, Matthew D. "ADADELTA: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012).

A OLS Estimator Optimality

We define the OLS model as

$$\mathbf{z} = X\beta + \epsilon, \tag{A.1}$$

where $X \in \mathbb{R}^{n \times m}$ with full column rank m containing the exogenous variables, $\beta \in \mathbb{R}^m$ containing the model parameters, $\epsilon \in \mathbb{R}^n$ contains uncorrelated random errors with $\mathbb{E}[\epsilon] = 0$ and variance σ^2, and $\mathbf{z} \in \mathbb{R}^n$. In our fit we want to minimise the sum of squared residuals

$$\begin{aligned} S(\beta) &= \langle \epsilon, \epsilon \rangle \\ &= \|\mathbf{z} - X\beta\|_2^2 . \end{aligned}$$

Now let the parameter $\hat{\beta}$ be our estimator such that

$$\hat{\beta} = (X^T X)^{-1} X^T \mathbf{z}. \tag{A.2}$$

X is full column rank, and thus $(X^T X)$ is invertible. Then

$$\begin{aligned} X^T(\mathbf{z} - X\hat{\beta}) &= X^T\mathbf{z} - (X^T X)(X^T X)^{-1} X^T \mathbf{z} \\ &= X^T\mathbf{z} - X^T\mathbf{z} = 0 \end{aligned}$$

To find the $\hat{\beta}$ that minimizes the sum of squared residuals, we need to take the partial derivative

$$\begin{aligned} \frac{\partial}{\partial \beta} S(\beta) &= \langle -X, \mathbf{z} - X\beta \rangle + \langle \mathbf{z} - X\beta, -X \rangle \\ &= -X^T(\mathbf{z} - X\beta) - (\mathbf{z} - X\beta)^T X \\ &= -X^T(\mathbf{z} - X\beta) - (X^T(\mathbf{z} - X\beta))^T . \end{aligned}$$

We showed that $X^T(\mathbf{z} - X\hat{\beta}) = 0$ for our chosen $\hat{\beta}$, thus for $\beta = \hat{\beta}$ we obtain

$$\frac{\partial}{\partial \beta} S(\beta) = 0,$$

thus we have an extremum for $\beta = \hat{\beta}$. Furthermore, since $S(\beta)$ is quadratic we have a local minimum. Now let $S(\beta)$ be the global minimal for some $\beta \neq \hat{\beta}$. Due to the minimum, the following property holds:

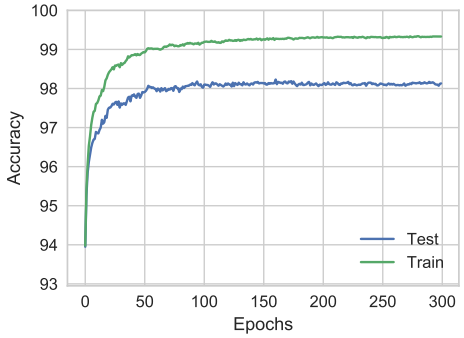
$$\frac{\partial}{\partial \beta} S(\beta) = 0,$$

and thus $X^T(\mathbf{z} - X\beta) = 0$. From this follows that $X^T X\beta = X^T \mathbf{z}$. Now we write $S(\beta)$ in terms of $S(\hat{\beta})$:

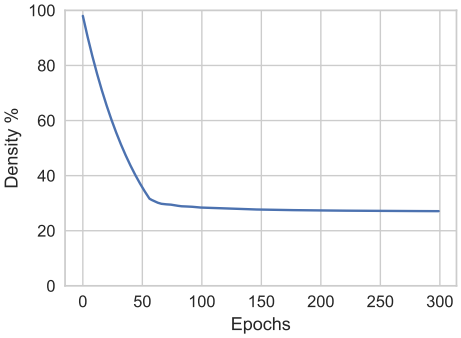
$$\begin{aligned} S(\beta) &= S(\hat{\beta}) + \hat{\beta}^T X^T X\hat{\beta} + \beta^T X^T X\beta - \mathbf{z}^T X\beta - \beta^T X^T \mathbf{z} \\ &= S(\hat{\beta}) + \hat{\beta}^T X^T X\hat{\beta} - \beta^T X^T X\beta \\ &= S(\hat{\beta}) + (\hat{\beta} - \beta)^T X^T X(\hat{\beta} - \beta). \end{aligned}$$

Since $X^T X$ is positive definite, we know that $(\hat{\beta} - \beta)^T X^T X(\hat{\beta} - \beta) \geq 0$ and $(\hat{\beta} - \beta) \notin \ker(X) = \{\mathbf{0}\}$. In this case we should have for $\beta \neq \hat{\beta}$ that $S(\hat{\beta}) < S(\beta)$, but $S(\beta)$ was already minimal, which is a contradiction. From this follows that $\beta = \hat{\beta}$ minimizes the sum of squared residuals. Now assume there is some other $\beta \neq \hat{\beta}$ hat that also minimizes $S(\beta)$. Again, since $(\hat{\beta} - \beta) \notin \ker(X)$, we have $(\hat{\beta} - \beta)^T X^T X(\hat{\beta} - \beta) > 0$ and thus $S(\hat{\beta}) < S(\beta)$, which leads to a contradiction. From this follows that the solution is unique.

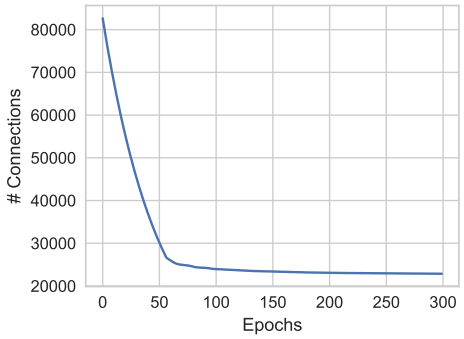
B NG MNIST Training



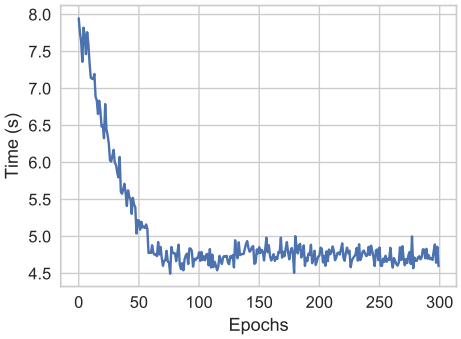
(a) The accuracy of the predictions for both training and test data.



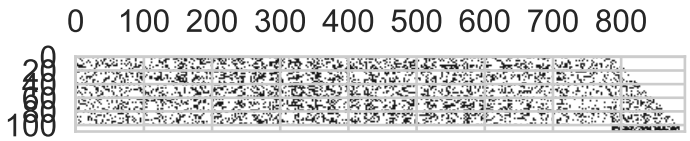
(b) The density compared to the initialization plotted over the number of epochs.



(c) The number of connections in each epoch.

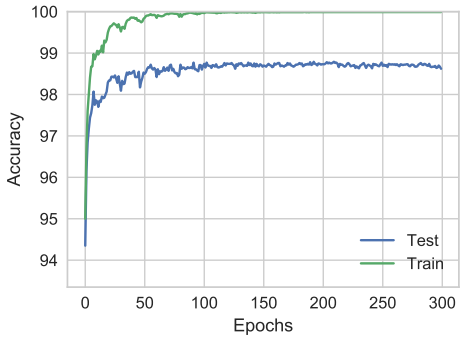


(d) The time each epoch took in seconds.

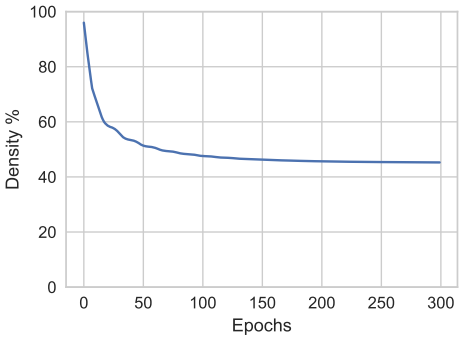


(e) The spy plot of the resulting weights.

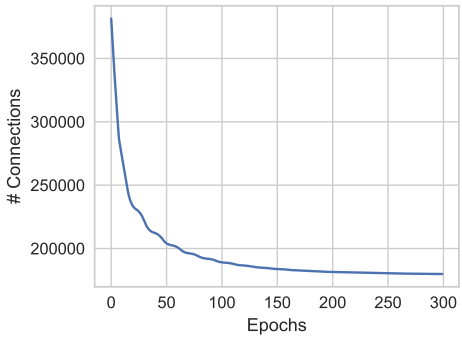
Figure B.1: A NG training with 100 nodes on MNIST



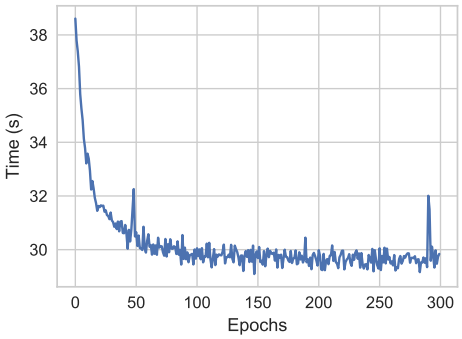
(a) The accuracy of the predictions for both training and test data.



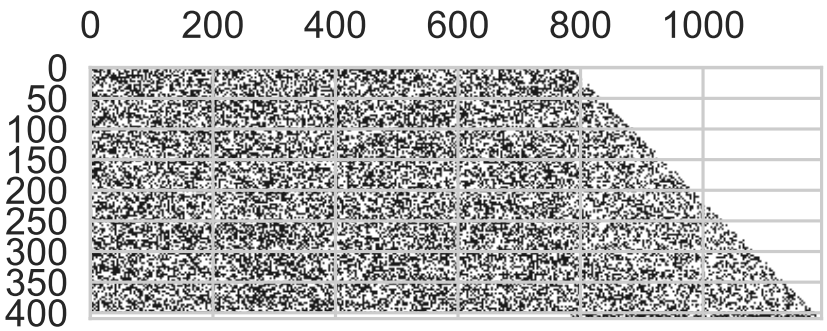
(b) The density compared to the initialization plotted over the number of epochs.



(c) The number of connections in each epoch.



(d) The time each epoch took in seconds.



(e) The spy plot of the resulting weights.

Figure B.2: A NG training with 400 nodes on MNIST

C Code

ng.h

```
#pragma once

template<typename tActivation = PReluActivation, typename tOutput
    ↪ = SoftMaxActivation, typename tMat = arma::fmat>
class NeuralGraph
{
public:

    struct TrainingState
    {
        tMat Ldf;
        arma::fvec biasdf;
        arma::fvec gaindf;
        arma::fvec dropout;
        bool hasDropout;
        float dropChance;
        float loss;
        size_t samples;

        /**
         * Constructor
         *
         * @param Ldf The L gradient matrix.
         * @param biasdf The bias gradient.
         * @param gaindf The gain gradient.
         * @param samples The sample count.
         */

        TrainingState(tMat Ldf, arma::fvec biasdf, arma::fvec gaindf, size_t
            ↪ samples)
            : Ldf(Ldf),
              biasdf(biasdf),
              gaindf(gaindf),
              samples(samples),
              hasDropout(true),
```

```
            dropChance(0.15f),
            loss(0.0f)
    {

    }

    /**
     * Copy constructor
     *
     * @param other The other trainingstate to copy from.
     */

    TrainingState(const TrainingState &other)
        : Ldf(other.Ldf),
          biasdf(other.biasdf),
          gaindf(other.gaindf),
          dropout(other.dropout),
          samples(other.samples),
          hasDropout(other.hasDropout),
          dropChance(other.dropChance),
          loss(other.loss)
    {

    }

    /**
     * Resets this object
     */

    void Reset()
    {
        Ldf.zeros();
        biasdf.zeros();
        gaindf.zeros();
        dropout.zeros();
        samples = 0;
        loss = 0.0f;
    }
};

/**
 * Constructor
 *
 * @param inputSize Size of the input.
 * @param outputSize Size of the output.
 * @param hiddenNodes The number of hidden nodes.
 */
```

```
NeuralGraph(std::size_t inputSize, std::size_t outputSize, std::size_t
    ↪ hiddenNodes)
    : mInputSize(inputSize),
      mOutputSize(outputSize),
      mHiddenNodes(hiddenNodes),
      mL(outputSize + hiddenNodes, inputSize + outputSize +
          ↪ hiddenNodes, arma::fill::zeros),
      mBias(outputSize + hiddenNodes, arma::fill::zeros),
      mGain(outputSize + hiddenNodes, arma::fill::ones),
      mLColNorm(outputSize + hiddenNodes, arma::fill::ones),
      mN(outputSize + hiddenNodes),
      mDensity(1.0f)
{

    mL.tail_cols(mN) = arma::trimatl(arma::randn<arma::fmat>(mN,
        ↪ mN), -1);
    mL.head_cols(mInputSize) = arma::randn<arma::fmat>(mN,
        ↪ mInputSize);
    mL.tail_rows(mOutputSize).cols(0, mInputSize).zeros();
    mL.tail_cols(mOutputSize).zeros();

    mL = mL.t();

    // He initialization
    for (arma::uword c = 0; c < mL.n_cols; ++c)
    {
        float connections = 0.0f;

        if (c < mHiddenNodes)
        {
            connections = c + mInputSize;
        }
        else
        {
            connections = mHiddenNodes;
        }

        mL.col(c) *= std::sqrt(2.0f / ((1 + std::pow(0.2f, 2)) *
            ↪ connections));
    }

    CacheIndices();

    mInitialNonZeros = arma::nonzeros(mL).eval().n_elem;
}

/**
 * Returns a valid default state
 *
```

```
 * @return A TrainingState.
 */

TrainingState DefaultState()
{
    return TrainingState(tMat(arma::size(mL), arma::fill::zeros), arma::
        ↪ fvec(mN, arma::fill::zeros), arma::fvec(mN,arma::fill::zeros),
        ↪ 0);
}

/**
 * Cache non-zero indices of the weight matrix, such that we can easily
 * iterate on those.
 */

void CacheIndices()
{
    tMat lmask = tMat(arma::spones(arma::sp_fmat(mL)));
    mLOutgoingIndices.resize(mL.n_rows);
    mLIngoingIndices.resize(mL.n_cols);

    for (size_t i = 0, size = mL.n_rows; i < size; ++i)
    {
        mLOutgoingIndices[i] = arma::find(lmask.row(i));
    }

    for (size_t i = 0, size = mL.n_cols; i < size; ++i)
    {
        mLIngoingIndices[i] = arma::find(lmask.col(i));

        mLColNorm(i) = arma::norm(mL.col(i));
    }
}

/**
 * Forward propagates the given input
 *
 * @param input The input vector.
 *
 * @return An arma::fvec of size output.
 */

arma::fvec Forward(arma::fvec input)
{
    auto xi = CalculateStates(input);
    return xi.tail_rows(mOutputSize);
}

/**
```

```
 * Calculates the states from the given input vector by forward-
     ↪ propagation
 *
 * @param [in,out] input The input vector.
 *
 * @return The calculated state.
 */

arma::fvec CalculateStates(arma::fvec &input)
{
    arma::fvec xi = arma::fvec(mL.n_rows, arma::fill::zeros);
    xi.head_rows(input.n_rows) = input;

    for (size_t i = 0, ti = mInputSize, end = mHiddenNodes; i < end;
        ↪ ++i, ++ti)
    {
        float x = AffineTransform(i, ti, xi);
        xi(ti) = tActivation::f(x);
    }

    // propagate weight to the output and calculate the activation
        ↪ separately
    for (size_t i = mHiddenNodes, ti = mInputSize + mHiddenNodes,
        ↪ end = mN; i < end; ++i, ++ti)
    {
        xi(ti) = AffineTransform(i, ti, xi);
    }

    xi.tail_rows(mOutputSize) = tOutput::f(xi.tail_rows(mOutputSize)
        ↪ );
    return xi;
}

/**
 * Calculates the states input and returns it completely (used for back-
     ↪ propagation) by forward-
 * propagation
 *
 * @param [in,out] state The state to capture statistics.
 * @param [in,out] input The input vector.
 *
 * @return The calculated intermediate states.
 */

std::tuple<arma::fvec, arma::fvec> CalculateStatesInput(TrainingState
    ↪ &state, arma::fvec &input)
{
    arma::fvec z = arma::fvec(mL.n_rows, arma::fill::zeros);
    z.head_rows(input.n_rows) = input;
```

```
    arma::fvec xi = arma::fvec(mL.n_rows, arma::fill::zeros);
    xi.head_rows(input.n_rows) = input;

    if (state.hasDropout)
    {
        z.head_rows(input.n_rows) %= state.dropout.head_rows(input.
            ↪ n_rows);
        xi.head_rows(input.n_rows) %= state.dropout.head_rows(
            ↪ input.n_rows);
    }

    for (size_t i = 0, ti = mInputSize, end = mHiddenNodes; i < end;
        ↪ ++i, ++ti)
    {
        float x = AffineTransform(i, ti, xi);


        if (state.hasDropout)
        {
            x *= state.dropout(i);
        }

        z(ti) = x;
        xi(ti) = tActivation::f(x);
    }

    // propagate weight to the output and calculate the activation
        ↪ separately
    for (size_t i = mHiddenNodes, ti = mInputSize + mHiddenNodes,
        ↪ end = mN; i < end; ++i, ++ti)
    {
        float x = AffineTransform(i, ti, xi);
        z(ti) = x;
    }

    xi.tail_rows(mOutputSize) = tOutput::f(z.tail_rows(mOutputSize));
        ↪
    return { xi, z.tail_rows(mN)};
}

/**
 * Performs an affine transform on the vector and calculates the weight
     ↪ normalization
 *
 * @param i Zero-based index of the node.
 * @param ti The translated node index (deals with non-square matrix).
 * @param xi The xi vector.
 *
 * @return The node output value.
```

```
 */

FORCEINLINE float AffineTransform(size_t i, size_t ti, const arma::
    ↪ fvec &xi)
{
    return arma::as_scalar(arma::dot(mL.unsafe_col(i).head(ti) /
        ↪ mLColNorm.at(i) * mGain(i), xi.head_rows(ti))) + mBias(i)
        ↪ ;
}

template< typename tObjective = CrossEntropy<SoftMaxActivation
    ↪ >>

/**
 * Adds a sample to the current training state
 *
 * @param [in,out] state The training state.
 * @param in The input vector.
 * @param out The true output vector.
 *
 * @return The value of the objective function.
 */

float AddSample(TrainingState &state, arma::fvec in, arma::fvec out)
{
    if (state.hasDropout)
    {

        state.dropout = arma::fvec(mInputSize + mHiddenNodes, arma
            ↪ ::fill::randu);
        float realDropChance = state.dropChance;

        for (arma::uword i = 0; i < state.dropout.n_rows; ++i)
        {
            state.dropout(i) = state.dropout(i) > realDropChance ? 1.0f
                ↪  / (1.0f - realDropChance) : 0.0f;
        }
    }

    auto[xi, z] = CalculateStatesInput(state, in);

    arma::fvec lagrange(arma::size(z), arma::fill::zeros);

    arma::fvec dflogsum = tObjective::df(xi.tail_rows(mOutputSize), z.
        ↪ tail_rows(mOutputSize), out);

    lagrange.tail_rows(mOutputSize) = dflogsum;
    state.biasdf.tail_rows(mOutputSize) += dflogsum;
```

```
    for (size_t j = mN − 1, end = mN − mOutputSize; j >= end; −−j
        ↪ )
    {

        const float nv = mLColNorm(j);

        for (const auto &i : mLIngoingIndices[j])
        {
            float nablaW = lagrange.at(j) ∗ xi(i);
            float nablag = nablaW ∗ mL.at(i, j) / nv;
            state.gaindf(j) += nablag;

            state.Ldf(i, j) += mGain.at(j) / nv ∗ (nablaW − mL.at(i, j)
                ↪  / nv ∗ nablag);
        }
    }

    for (std::ptrdiff_t j = mHiddenNodes − 1, jt = mHiddenNodes +
        ↪ mInputSize − 1; j >= 0; −−j, −−jt)
    {
        float lagrangej = 0.0f;

        for (const auto &i : mLOutgoingIndices[jt])
        {
            lagrangej += lagrange.at(i) ∗ (mL.at(jt, i) / mLColNorm.at
                ↪ (i) ∗ mGain.at(i)) ∗ state.dropout(i);
        }

        lagrange(j) = tActivation::df(z(j)) ∗ lagrangej;

        const float nv = mLColNorm(j);

        for (const auto &i : mLIngoingIndices[j])
        {
            float nablaW = lagrange.at(j) ∗ xi(i);
            float nablag = nablaW ∗ mL.at(i, j) / nv;
            state.gaindf.at(j) += nablag;

            state.Ldf.at(i, j) += mGain.at(j) / nv ∗ (nablaW − mL.at(i,
                ↪  j) / nv ∗ nablag);
        }

        state.biasdf.at(j) += lagrange.at(j);
    }

    ++state.samples;

    return tObjective::f(xi.tail_rows(mOutputSize), out);
}
```

```
/**
 * Sparsifies the weight matrix based on the element values
 *
 * @param minimum The minimum value that is considered still for
     ↪ truncation.
 * @param maxPercentile The maximum percentile to delete.
 *
 * @return The density percentage.
 */

float Sparsify(float minimum, float maxPercentile)
{
    tMat W = GetWeight();

    const arma::fvec nonzeros = arma::sort(arma::abs(arma::nonzeros(
        ↪ W)));
    const float percentile = std::min(nonzeros((size_t)(maxPercentile *
        ↪  nonzeros.n_elem)), minimum);

    mL.elem(arma::find(arma::abs(W) < percentile)).zeros();
    CacheIndices();

    mDensity = (float)arma::nonzeros(mL).eval().n_elem / (float)(
        ↪ mInitialNonZeros);
    return mDensity;
}

/**
 * Gets the weight matrix from the weight normalization
 *
 * @return The weight matrix.
 */

tMat GetWeight()
{
    tMat W = mL;

    for (arma::uword c = 0; c < W.n_cols; ++c)
    {
        W.unsafe_col(c) *= mGain(c) / mLColNorm(c);
    }

    return W;
}

/**
 * Gets the density percentage
 *
```

```
     * @return A float.
     */

    float Density()
    {
        return mDensity;
    }

    /**
     * Gets the number of connections
     *
     * @return A float.
     */

    float Connections()
    {
        return mDensity * mInitialNonZeros;
    }

    std::size_t mInputSize;
    std::size_t mOutputSize;
    std::size_t mHiddenNodes;
    std::size_t mN;

    tMat mL;
    std::vector<arma::uvec> mLOutgoingIndices;
    std::vector<arma::uvec> mLIngoingIndices;
    arma::fvec mLColNorm;
    arma::fvec mBias;
    arma::fvec mGain;

    float mDensity;

    std::size_t mInitialNonZeros;
};
```