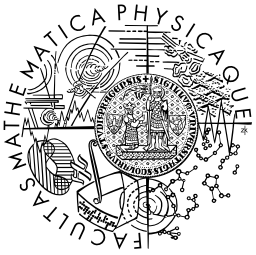


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Tomáš Witzany

Deep neural networks and their application for economic data processing

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	doc. RNDr. Iveta Mrázová, CSc.
Study programme:	Informatics
Study branch:	Theoretical Informatics

Prague 2017

V první řadě bych poděkoval především Doc. RNDr. Ivetě Mrázové, CSc. za cenné rady a přípomínky, za trpělivost a průběžnou kontrolu mých výsledků při vedení mé diplomové práce. Dále bych rád poděkoval Národní Gridové Infrastruktuře Metacentrum za přístup k výpočetním a úložným zařízením, pod programem "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042).

V neposlední řadě chci poděkovat svým nejbližším, především rodičům, za poskytování motivace a podpory při psaní této práce i během celého mého studia.

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Hluboké neuronové sítě a jejich využití při zpracování ekonomických dat

Autor: Bc. Tomáš Witzany

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Doc. RNDr. Iveta Mrázová, CSc., Katedra teoretické informatiky a matematické logiky

Abstrakt: Analýza makroekonomických časových řad je klíčová pro informovanost rozhodnutí politiků na národní úrovni. Analýza ekonomických údajů má bohatou historii a zejména v oblasti modelování nelineárních závislostí zůstává mnoho otevřených otázek. K moderním nástrojům pro analýzu časových řad patří mimo jiné metody strojového učení. Z těchto metod neuronové sítě patří k jedné z nejpoužívanějších, jak modelovat nelineární závislosti. Cíl této práce spočívá ve studiu hlubokých neuronových sítí, analýze jejich vlastností a posouzení jejich kvalit pro řešení úloh, například prognózu vývoje HDP nebo klastrování zemí. Použité modely zahrnují vrstevnaté neuronové sítě, LSTM sítě, konvoluční sítě a Kohonenovy mapy. K analýze a testování studovaných modelů byla použita historická data poskytovaná Organizací spojených národů a Světovou bankou. Tato data zahrnují historii makroekonomického vývoje přes 190 různých zemí za posledních padesát let. Vzhledem k vysokým časovým nárokům na testování modelů jsme využili služeb výpočetního centra MetaCentrum.

Klíčová slova: neuronové sítě, mlp, lstm, cnn, som, makroekonomické časové řady

Title: Deep neural networks and their application for economic data processing

Author: Bc. Tomáš Witzany

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Doc. RNDr. Iveta Mrázová, CSc., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Analysis of macroeconomic time-series is key for the informed decisions of national policy makers. Economic analysis has a rich history, however when considering modeling non-linear dependencies there are many unresolved issues in this field. One of the possible tools for time-series analysis are machine learning methods. Of these methods, neural networks are one of the commonly used methods to model non-linear dependencies. This work studies different types of deep neural networks and their applicability for different analysis tasks, including GDP prediction and country classification. The studied models include multilayered neural networks, LSTM networks, convolutional networks and Kohonen maps. Historical data of the macroeconomic development across over 190 different countries over the past fifty years is presented and analysed. This data is then used to train various models using the mentioned machine learning methods. To run the experiments we used the services of the computer center MetaCentrum.

Keywords: neural networks, lstm, cnn, macroeconomic time series analysis

Contents

Chapter 1

Introduction

Analysis and prediction of macroeconomic time-series is key for national policy-makers. However, economic forecasting is not a simple task because of the lack of an accurate theoretical model of the economy. For the analysis of non-linear dynamical systems, such as macroeconomics, it is often infeasible to obtain an analytical model. In such cases the solution often is to resort to black-box models that ignore the internal mechanisms of the system and simply attempt to reproduce the system behavior. In this work we focus on such models, classic and deep neural networks.

In this work we examined and compared classic neural network models to more complex deep neural networks. The available research in this field mostly focuses on multilayer neural-networks with varying rates of success. In this work we developed multilayer neural network models which we then compared to recurrent and convolutional neural network models. Even though both recurrent and convolutional neural networks are architectures that are well performing on tasks involving time-series, the available research applying these models to economic data is lacking.

The work is based on publicly available macro-economic data from international organisations as the World Bank and the United Nations. The data is first analysed and visualised using self-organising maps. Afterwards examine many different architectures of artificial neural networks and their applicability to a prediction task formulated on the data. One of the first models we trained for comparison is a multilayer perceptron (MLP) network and we experimented with many different parameters of multilayer neural networks.

One of the possible problems that arise when using multilayer neural networks to model time-series is that they have difficulties learning a temporal dependency. MLP networks look at all the historical data at once, but on the other hand recurrent neural networks (RNN) are neural network architectures that handle time-series by looking at the data in order and learning how a sequence changes over time. In this work we focus on Long-Short-Term-Memory neural networks as they are a RNN architecture that had been shown to generally outperform traditional RNNs on time-series analysis tasks.

Convolutional neural networks (CNN) are neural networks that aggregate data in a similar manner to an animals visual cortex. This is in practice employed by training filters on the data, such as Gaussian or averaging filters. We apply this method to economic time series as many of the classic time-series analysis

methods employ in one way or another such filters - moving averages, max/min filters and more. The network will train the filter best applicable and then use this data to create a prediction.

The work is structured into a theoretical and experimental part. The theoretical part describes multilayer neural networks in Chapter 2, recurrent neural networks in Chapter5 and convolutional neural networks in Chapter 4. The theoretical part finally describes self-organising maps in Chapter 3. The experimental chapter 6 shows an exploratory analysis of the data and visualises it using SOMs in Section 6.2. Afterwards we define a prediction task in Section 6.3 and explore the applicability of different artificial neural networks to this task. We first show how MLP networks perform in Section 6.3.2 and determine the best parameters for them. Later we examine the performance of LSTM networks in Section 6.3.3. Lastly we show the performance of convolutional neural networks in Section 6.3.4.

The work includes the data and programs necessary to execute the experiments described. The experiments involving LSTM networks and CNNs require significant computation time to execute and therefore the experiments were executed using the services of the computational center MetaCentrum. The results of these experiments are included as well.

Chapter 2

Artificial neural networks

Artificial neural networks represent a biologically inspired model for distributed computation. The model simulates the biological nervous system with a network of artificial neurons as shown in Figure 2.1. Artificial neurons or nodes in the network correspond to a mathematical function in the sense that a neuron's output is determined by the neuron's transfer function and its inputs. ANNs can be organised into many different connection topologies and can have several different neuron types [1]. In this chapter we will first present an brief overview of ANN properties in general and later focus on one of the most common type of ANNs - the multilayered perceptron network (MLP).

The main advantages of ANNs are their high flexibility and the fact that no assumptions have to be made about the data. MLPs have the ability to learn any limited function - from the universal approximation theorem [2] it follows that a single hidden layer perceptron network can approximate any continuous function on a compact subset of $\mathbb{R}^n$ given enough neurons. Also, generally there is no expert knowledge necessary to train the network, only training data.

ANNs in general can be used for both supervised and unsupervised learning [3] but also for reinforcement learning [4]. Supervised learning is learning by example from a teacher. The neural network is presented the inputs with their desired outputs and a training algorithm is used to adapt the network to exhibit this behavior. Contrary to this during unsupervised learning the model is not presented with this information and its goal is to inferr any patterns in the unlabeled data. Lastly reinforcement learning is a method inspired by biology. The model is rewarded whenever it is performing well and the learning algorithms task is to predict which actions will yield the best reward.

A very simple example of an ANN with one hidden layer can be seen in Figure 2.1. In this figure each circle represents a neuron and the arrows between them represent channels of information flow. The column of neurons on the left - the input layer - are neurons that simply hold a certain value - the input. This input is then sent by the connecting arrows into the next layer - the hidden layer. Each neuron in the hidden layer transforms (weighted sum) these signals to form a neuron potential. This potential is then transformed by a transfer function and the resulting value is sent to the output layer. The output layer acts in a similar way as the hidden layer, but its neurons can have different transfer functions.

ANNs can take many different forms and variants. One of the main variants are multilayered ANNs where neurons are arranged into layers as shown in

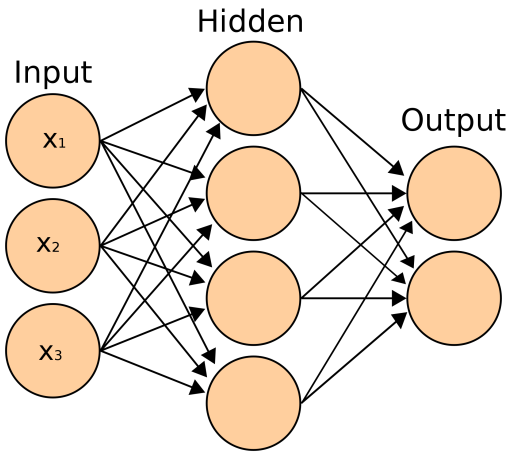


Figure 2.1: **Multilayered artificial neural network**

The figure contains a simple artificial neural network with one hidden layer. The circles denote neurons and arrows connections.

Figure 2.1. Multilayered ANNs are often further divided into shallow and deep neural networks. Shallow ANNs generally only have one hidden layer, while deep neural networks have several. Another important classification of ANNs is into feedforward and recurrent neural networks. Feedforward neural networks do not form cycles in the network topology. The network on Figure 2.1 is a feedforward network. We discuss recurrent neural networks in more detail in Chapter 5.

Each variant is best suited for a slightly different task. ANNs are commonly used to approximate functions, pattern recognition and classification. In practice ANNs are applied in many areas such as marketing, finance, retail, image recognition and more [5].

In the following sections we will focus feedforward multilayer perceptron (MLP) variant of ANNs trained with supervised learning. We first describe how a MLP computes its output in Section 2.1 followed by Section 2.2 on how to train them. In Section 2.3 we describe different transfer functions, their advantages and downfalls. And finally in Section 2.4 we describe the problem of overfitting and its possible solutions.

2.1 Feed-forward pass

In this section we look at the formal way a feed-forward multilayer perceptron (MLP) neural network evaluates its output. Later on we will use this knowledge to describe how such a neural network can be trained. The equations governing how the forward pass works are inspired by how the biological neuron functions in the human brain [6].

A multilayer feedforward neural network consists of two or more layers of neurons such as in Figure 2.1 where the flow of information is from the left to the right. First we define the notation to be used in the following formulas. We consider a neural network divided into L layers of neurons (including the input and output layer), the input dimension is denoted n and the output dimension m. The first layer indexed as the 1-st layer is the input layer. The input simply

exposes an input vector $\vec{x} = (x_1, \ldots, x_n)$. A weight representing a connection between the i-th neuron in the $l-1$-st layer and the j-th neuron in the l-th layer is denoted $w_{i,j}^{(l)}$. The goal of the forward pass is to calculate the output of the final layer - the L-th layer, which we will save into the output vector $\vec{p} = (p_1, \ldots, p_m)$. The number of neurons in a layer is denoted by s_l.

Each neuron computes its potential and activation value from the activation values provided by the connecting neurons. We denote an activation of the i-th neuron in the l-th layer as $a_i^{(l)}$ and the potential of the same neuron as $z_i^{(l)}$. A vector of activations of neurons in a certain layer is $\vec{a}^{(l)}$. The activations of the neurons in the input layer are fixed as $a_i^{(1)} = x_i$. We can now describe how any neuron in the network computes its potential and activation.

In a biological neuron, elictrical impulses are carried by dendrites to the neuron and are added up in its body. If the electric potential accumulated from the dendrites reaches a certain threshold, the neuron fires into its output - the axon. In Equation 2.1 we describe how a neuron i in layer l gathers up potential $z_i^{(l)}$ by means of a weighted sum of the activations of the s_{l-1} connected neurons in the $l-1$-th layer. The threshold necessary for it to fire is commonly called the neuron bias. Sometimes biases are written separately from the sum $z_i^{(l)} = \sum w_{ji}^{(l)} a_j^{(l-1)} - b_i^{(l)}$ where $b_i^{(l)}$ denotes the bias. For this analysis we include the biases into the sum for simplicity, which is done by having an extra neuron in each layer with a fixed activation value of -1.

$$z_i^{(l)} = \sum_{k=1}^{s_{l-1}} w_{k,i}^{(l)} a_k^{(l-1)} \tag{2.1}$$

The next discussed bioligical mechanism of the neuron is its firing. In the case of the artificial neuron we calculate the activation of the i-th neuron in the l-th layer - $a_i^{(l)}$ as shown in Equation 2.2. The activation is calculated as the potential transformed by the transfer function f. The transfer functions commonly used in artificial neural networks are sigmoidal (logistic) function - $f_{log}(s) = \frac{1}{1+e^{-\lambda s}}$ or a step function. Typical transfer functions result in a behavior where the neuron fires whenever the potential is above 0 and is silent otherwise. Transfer functions are discussed in more detail later in Section 2.3.

$$a_i^{(l)} = f(z_i^{(l)}) \tag{2.2}$$

Listing 2.1: Feed forward pass

```
1  FF(net, x⃗)
2  #set the 1st layer as the input
3  a⃗^(1) = x⃗
4  #process the non-input layers in order
5  foreach layer l = 2,...,L:
6          foreach i = 1,...,s_l:
7                  #compute the potential and activation
8                  z_i^(l) = Σ_k w_ki^(l) a_k^(l-1)
9                  a_i^(l) = f(z_i^(l))
10 #activations in the last layer are the network output
11 p⃗ = a⃗^(L)
```

These equations describe an algorithm in Listing 2.1 that is used to calculate the output of the network - the activations of the last layer. The algorithm's input is a structure describing the network - net (including the network's weights) - and the network input x. The algorithm's output is $\vec{p}$. In this algorithm we first set up the input of the network assigning the activation of the first layer. After that the layers are processed in topological order, computing the activation of each neuron. The activations of neurons in the last layer are then output.

2.2 Backpropagation

In the previous section we described how a neural network computes its output. However to make the network compute anything useful we need to assign appropriate weights to the network. For certain simple networks and tasks it is possible to compute the weights directly, though in most cases it would be a nearly impossible task to assign the weights manually. The main strength of neural networks lies in the fact that we can optimize weights of a network to fit to any specific data set using an algorithm called Back Propagation (BP) [7] [8].

BP is a algorithm that trains the weights of a multilayer feedforward network with smooth transfer functions to fit the provided data. The algorithm minimises the network error on the data, with respect to the network's weights and biases. The algorithm works with various different definitions of what the network erorr is, in this analysis we use a mean squared error 2.3. Furthermore there are several different types of backpropagation, such as total gradient descent [9], stochastic gradient descent [10] (which we describe here), momentum based BP [11] or various adaptive BP variants.

2.2.1 Notation and overview

The algorithm takes a data set of input-desired output pairs and a network to train. The algorithm works iteratively where we present the network with an input and correct the weights according the the output produced. In order to do this we calculate the gradient of the error function, especially its weight components. We can then use these weight components to update the weights in order to lower the error of the network on the presented input. This continues iteratively until a stopping condition is reached, for example until the gradient is low enough. In this analysis we focus on describing the stochastic version of the algorithm, where the weights of the network are updated in each iteration.

Suppose the training set $\{(\vec{x}^{(1)},\vec{y}^{(1)}),\ldots,(\vec{x}^{(T)},\vec{y}^{(T)})\}$ of T training input-output pairs (the superscript here denotes the index of the sample, not the layer). For a single training sample $(\vec{x},\vec{y})$ we define the error cost function simply as the squared difference between the actual network output and the desired output in Equation 2.3 where FF is the network output function with the defined parameters as described in Listing 2.1.

$$E(net,\vec{x},\vec{y}) = \frac{1}{2}||FF(net,\vec{x}) - \vec{y}||^2 \tag{2.3}$$

We can use the single sample error cost functions to define the overall error cost function simply as an average error across the whole data set in 2.4. The goal

of the algorithm then is to minimise this overall error. However, since we are using the stochastic gradient descent variant of backpropagation we do not minimise the overall error directly. Nevertheless the idea is that we are still closing in on the minimum of such a overall error cost function.

$$E(net) = \left[\frac{1}{m}\sum_{k=1}^{m} E(net,\vec{x}^{(k)},\vec{y}^{(k)})\right] + \lambda \sum_{l=1}^{layers}\sum_{i=1}^{s_{l-1}}\sum_{j=i}^{s_l}(w_{ij}^{(l)})^2 \tag{2.4}$$

The stochastic gradient descent algorithm instead of calculating the gradient of the overall error 2.4 it calculates only the gradient of the single sample error. This greatly simplifies the process and lowers the computational requirements with often improving on found solutions [12]. The algorithm calculates the gradient in order to update the weights in the neural network with the goal of lowering the error as shown in Equation 2.5.

$$\Delta w_{ij}^{(l)} = -\alpha \frac{\partial}{\partial w_{ij}^{(l)}} E \tag{2.5}$$

2.2.2 Analysis

The goal of this analysis is to derive the partial $\frac{\partial}{\partial w_{ij}^{(l)}}E$ from Equation 2.5. The trick backpropagation uses to achieve this is that we use the error flow from the previous layer, which is $\frac{\partial}{\partial a_j^{(l)}}E$, to derive the partial. This error flow is easily computed for the output layer, and afterwards the algorithm propagates from layer by layer to the input (hence the name backpropagation). Our task then is to calculate how error flow propagates to the lower layer, $\frac{\partial}{\partial a_k^{(l-1)}}E$ based on the error from the higher level, along with deriving the partial $\frac{\partial}{\partial w_{ij}^{(l)}}E$ with the use of the error flow from the higher layer $\frac{\partial}{\partial a_k^{(l)}}E$. We will also use a helper variable, usually called deltas which is $\delta_k^{(l)} := \frac{\partial E}{\partial z_k^{(l)}}$.

When given the error flow from the layer $\frac{\partial E}{\partial a_j^{(l)}}$ we can derive the deltas $\delta_k^{(l)}$ using the chain rule as described in Equation 2.6. The first identity follows from the chain rule and the second from the definition of the activation $a_j^{(l)}$ in Equation 2.2.

$$\delta_j^{(l)} := \frac{\partial E}{\partial z_j^{(l)}} = \frac{\partial E}{\partial a_j^{(l)}}\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \frac{\partial E}{\partial a_j^{(l)}} f'(z_j^{(l)}) \tag{2.6}$$

Following this result we use the deltas to derive the formula for the weight gradient component $\frac{\partial}{\partial w_{ij}^{(l)}}E$ in Equation 2.7 again using the chain rule. In this equation the last identity follows now from the definition of the potential $z_j^{(l)}$ in Equation 2.1.

$$\frac{\partial E}{\partial w_{ij}^{(l)}} = \frac{\partial E}{\partial z_j^{(n)}}\frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)} \tag{2.7}$$

Next given the deltas $\delta_k^{(l)}$ we can derive the error flow to the lower layer $\frac{\partial}{\partial a_k^{(l-1)}}E$ in Equation 2.8.

$$\frac{\partial E}{\partial a_j^{(l-1)}} = \sum_r \frac{\partial E}{\partial z_r^{(l)}} \frac{\partial z_r^{(l)}}{\partial a_j^{(l-1)}} = \sum_r \delta_r^{(l)} w_{j,r}^{(l)} \tag{2.8}$$

To summarize we now have the equations to calculate the weight gradient component $\frac{\partial E}{\partial w_{ij}^{(l)}}$ when we are given the error flow from the previous layer - $\frac{\partial E}{\partial a_j^{(l)}}$. We can also propagate this error to the lower layers from the higher layers. Therefore the last thing to resolve is where to get the error flow to the output layer which we get directly from the definition of our single sample error from Equation 2.4.

$$\frac{\partial E}{\partial a_j^{(n)}} = \frac{\partial}{\partial a_j^{(n)}} \frac{1}{2} ||FF(net,\vec{x}) - \vec{y}||^2 = a_j^{(n)} - \vec{y}_j \tag{2.9}$$

2.2.3 Algorithm

With this knowledge we can construct an algorithm, that computes the single sample partials for a given input-output pair. These partials can be then used to update the weights in order to lower the error rate of the network. The BP algoritm executes the pseudocode in Listing 2.2 for a specific sample in the training set. We have L layers, with the L-th layer being the output layer and the 1-st layer the input layer.

Listing 2.2: BP Algorithm - single sample update

```
1  BPSingle(net,x⃗,y⃗)
2  #perform the feedforward pass
3  FF(net,x⃗)
4  #calculate propagation of the output layer
5  foreach layer l in (L...1):
6          foreach neuron j in (1...s_l):
7                  if l==L: #output layer
8                          δ_j^(l) = (a_j^(l) − y⃗_j) f′(z_j^(n))
9                  else: #hidden layer
10                         δ_j^(l) = (Σ_r δ_r^(l+1) w_jr^(l+1)) f′(z_j^(l))
11                 foreach weight i in (1,...s_{l−1}):
12                         ∂/∂w_ij^(l) E = δ_j^(l) a_i^(l−1)
13                         Δw_ij^(l) = −α ∂/∂w_ij^(l) E
```

Listing 2.3: BP Algorithm

```
1  BP(training set)
2  initialize net with small random weights
3  while stopping condition not met:
4          select (x⃗,y⃗) from training set:
5          BPSingle(net,x⃗,y⃗)
```

The complete BP algorithm is described in listing 2.3. We initialize the weights with small random values, for example according to $Normal(0,\varepsilon^2)$ for a small ε. Then we iteratively lower the error rate on the training set by going through samples and updating the weights according to the gradient partials. The algorithm lined up in 2.3 is purposely left vague on several points. As the stopping condition mentioned on line 3 we want to use some kind of indicator of the model convergence on the data, for example small gradient. The algorithm can measure how much the weights of the network changed over the several past iterations. The algorithm could also measure the network performance on an independent (validation) sample and stop whenever this performance stops improving. On line 4 the algorithm selects a sample from the training set. The sample can be selected completely randomly from the training set, or we can pass the whole training set in order before going back to the first sample.

2.3 Transfer Functions

In the previous sections we mentioned transfer functions as being a key component of an artificial neuron. Transfer functions are the mechanism by which neural networks introduce non-linearity into their underlying model. Furthermore, for the backpropagation algorithm to work its differential (see Equation 2.6) needs to be defined. There are many different types of activation functions that can be used in neural networks. In this section we describe a few important ones, some of which we will later use in our experiments.

2.3.1 Sigmoid

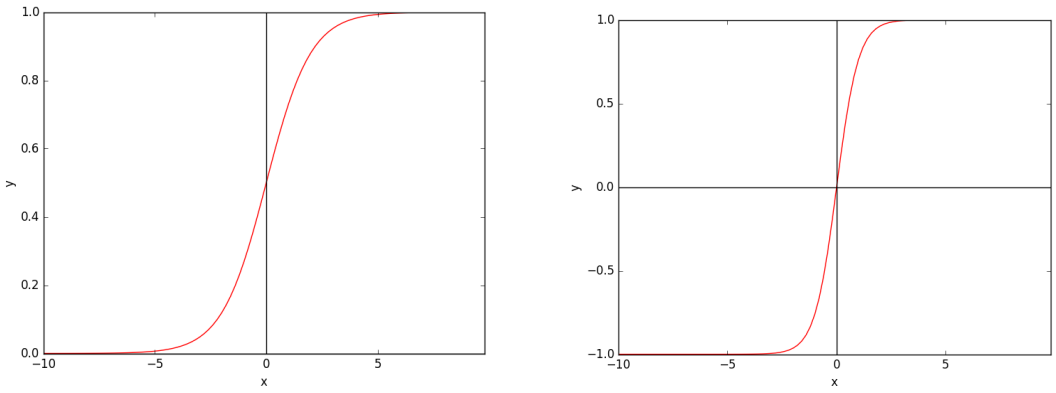
The sigmoid, or logistic, non-linearity has the mathematical form of $\sigma(x) = \frac{1}{1+e^{-x}}$ and is charted in Figure 2.2. The sigmoid function has historically been one of the more used activation functions, since it can be interpreted as a firing of a biological neuron. A biological neuron is either not firing at all or it is fully saturated and firing at its maximum level.

In practice sigmoid activation functions have recently fallen out of favor because of one of their major drawbacks. An undesirable property of the sigmoid activation function is the when the neurons activation saturates in either of the tails of 1 or 0, its gradient in these regions is nearly zero [13]. This mean that nearly no information will flow through this neuron back into the network. This also comes in hand with random initializations of the neuron, if the weights are randomly initialised as too high, the neuron will immediately saturate and the network will barely learn.

A variant of the sigmoid is the so called tanh function with a mathematical form of $tanh(x) = 2\sigma(2x) - 1$, charted in Figure 2.2. The tanh activation function is usually prefered over the sigmoid as networks using tanh functions often converge faster than networks using standard sigmoid function.

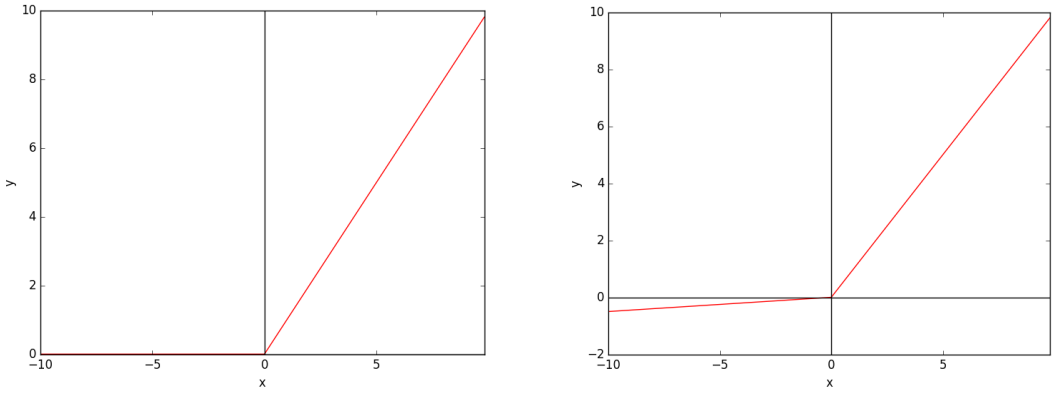
2.3.2 Rectified Linear Unit

The Rectified Linear Unit (ReLU) is an activation function that does not saturate at the high signal tail, unlike the sigmoid, first introduced in [14]. Its



(a) sigmoid (logistic) function (b) tanh function

Figure 2.2: **The sigmoid activation function variants.**



(a) ReLU function (b) LeReLU function

Figure 2.3: **The ReLU activation function variants.**

mathematical form is very simple - $f(x) = max(0,x)$ a linear function thresholded at zero - charted in Figure 2.3. It has been found to accelerate the gradient descent convergence in comparison to the classic sigmoid/tanh activation functions [15] and it is easily computed and implemented.

However in a similar manner to the sigmoid activation functions there are cases where the ReLU neuron unit can underperform. It can adapt in such a way that the neurons never activate and therefore stop propagating error in the network (dying neurons). In order to allow some small amount of error to flow through the unit even though it is not activated the Leaky ReLU unit has been introduced in [16]. With the mathematical form of $f(x) = max(x, \alpha x)$ where α is a small constant - leak coefficient. So far the Leaky ReLU has not had large success over a simple ReLU [17].

2.4 Overfitting

Overfitting is a problem in statistics and machine learning that appears when fitting a model to training data. The model is called to overfit, whenever the model describes the random noise of the training data more-so that the underlying

relationship. Generally speaking overfitting often occurs whenever the ratio of a model complexity compared to the training set size gets too high. This can be also explained in that when the model has enough 'capacity' to store random noise, it will overfit.

In order to prevent overfitting, a multitude of techniques can be employed. In the following sections we describe some of these methods, including cross-validation, two different types of regularization and sensitivity analysis.

2.4.1 Cross-validation

Cross-validation is a model evaluation method for assessing how a model generalizes to an independent data set [18]. It is usually used during supervised training. During supervised training a model is given a training set that the training algorithm uses to fit the model parameters to the training data. The main purpose of cross-validation is to define a dataset to test the model against in the training phase (a validation set).

The simplest kind of cross-validation is called the holdout method. Before training, the data is separated into two sets - a training and testing set. The training algorithm fits the model to the training set only, and we hold out the testing set. After the training algorithm finishes the model is evaluated on the testing set.

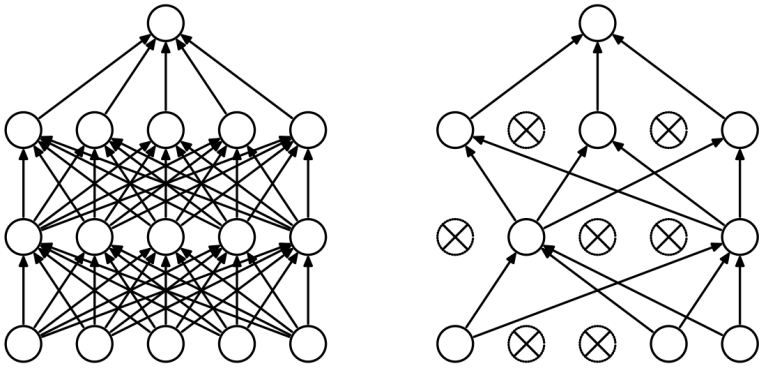
A simple alternative expanding the holdout method is k-fold cross-validation. The data is now separated into k subsets instead of only two. The training algorithm is then executed k times, each time with one of these subsets as the training set. After each run the validation set is constructed by a union of the $k-1$ subsets that were not used during training. To evaluate the total error across all folds an average of the evaluations in each run is calculated. The variance of this error decreases with increasing k, while each sample in the dataset was used during only one training algorithm run, which gives us a good estimation of the model strength.

2.4.2 Noise regularization

To prevent a model from overfitting one of the possible solutions is to add noise to the training data. This in turn makes it hard for the model to perfectly train random noise of the original training data. Even if the model overfitted, it would overfit on the random noise instead on the noise of the training set.

$$x_{noised} = x\mathcal{N}(1,\alpha) \tag{2.10}$$

Noise can be inserted into the data for example as an added Gaussian Gaussian noise according to Equation 2.10. Noise can be introduced to the training set at multiple points during training, in each epoch during training or before training once on the whole dataset. Adding noise each epoch during training makes it very difficult for the network to overfit even on the random noise.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.4: **Depiction of how Dropout affects a network during training.**

2.4.3 Dropout regularization

A technique employed in order to prevent overfitting of artificial neural networks is to randomly turn on or off neurons during the training process called Dropout [19]. In each training epoch individual neurons are either 'dropped out' - turned off with a probability of $1 - p$ or left on with a probability of p. Along with the turned off neurons the connecting edges are turned off aswell and they are not adapted during that epoch. An illustration of this process is shown in Figure 2.4. After the epoch is done the dropped out neurons are turned on again and the process iterates, by once again turning off some neurons randomly.

2.4.4 Sensitivity analysis

Determining an appropriate architecture for a neural network can be a very difficult task, as if we use a model that is too complex the training might lead to overfitting. On the other hand by using a simple model we risk not being able to capture some of the relationships present in the data at all. Sensitivity analysis is a way to extract information about the significance of neurons from a trained multilayer neural network. We can use sensitivity analysis to determine which feature of the input data are relevant and which aren't to prune an overfitted neural network. It is also possible to find hidden neurons that are almost irrelevant to the performance of the neural network.

If we examine the Taylor expansion of the neural network function F around its parameters θ and omit the expansion past the first derivative we get an equation (2.11) quantifying the change in the output of the neural network due to small perturbations in its parameters.

$$F(\theta + \Delta\theta) - F(\theta) = \Delta\theta F'(\theta) \tag{2.11}$$

Equation (2.11) shows us that the change in output of the neural netwok can be calculated by the derivative $F'(\theta)$ with respect to the network hyperparameter θ. We can now derive equations to calculate the sensitivity of the network output due to change in input values and hidden neuron activations. To demonstrate assume a three-layer neural network with one input, one hidden and one output

layer. The sigmoidal activation function is used in the hidden layer and the linear activation is used in the output layer.

For each input pattern $\vec{x}$, its input feature x_i and output neuron o_k we can calculate the sensitivity coefficients using (2.12) where w_{kj}^{out} are the weights from the hidden neuron j to the output neuron k, w_{ji}^{in} are weights from the input neuron i to the hidden neuron j and y_j is the activation of the j-th neuron due to input $\vec{x}$.

$$S_{o_k,x_i} = \frac{\partial o_k}{\partial x_i} = \frac{\partial}{\partial x_i} \sum_{j=1} w_{kj}^o y_j = \sum_j w_{kj}^o y_j (1 - y_j) w_{ji}^i \tag{2.12}$$

In a similar fashion we can compute the sensitivity coefficients for hidden unit activations according to equation (2.13).

$$S_{o_k,y_j} = \frac{\partial o_k}{\partial y_j} = o_k(1 - o_k) w_{kj} \tag{2.13}$$

Using (2.12) we can calculate the overall sensitivity coefficient of the output with regard to a certain input feature i. First we calculate the sensitivity coefficients of all output features with regard to the input feature specified for each pattern according to (2.12). We can then calculate the sensitivity coefficient of the output neuron k with regard to change in the input neuron i as an absolute average over all patterns (2.14).

$$S_{o_k,i} = \frac{\sum_{x \in X} |S_{o_k,x_i}|}{|X|} \tag{2.14}$$

The total sensitivity coefficient of the output with regard to the feature could be then computed in several different ways. As the maximum (2.15), a sum (2.16) or a weighted average (2.17) over the output features. In Equation (2.17) v_m are weights that denote the relative importance of the specific outputs and can be selected by an expert.

$$S_i = \max_m \{S_{o_m,i}\} \tag{2.15}$$

$$S_i = \sum_{i=1}^{m} S_{o_m,i} \tag{2.16}$$

$$S_i = \frac{\sum_{i=1}^{m} v_m S_{o_m,i}}{\sum_{i=1}^{m} v_m} \tag{2.17}$$

If the sensitivity coefficient of an input feature is very low relatively to the other features, that feature seems to contribute little to the ouput value. Large sensitivity coefficients in turn indicate significant features. We can use the computed coefficients to prune the irrelevant input features.

Chapter 3

Self-organizing maps

A self-organizing map (SOM) alternatively called Kohonen map [20] is a type of neural network that uses unsupervised learning to represent data of a high-dimension typically by means of a 2-dimensional map. The process of reducing the dimensionality of vectors, is essentially a data compression technique known as vector quantisation. In addition, this technique creates a map that stores information in such a way that any topological relationships within the training set are maintained. Another important aspect of SOMs is that it is an unsupervised learning technique. The mentioned properties make SOMs especially useful for visualising high-dimensionality data. The architecture is inspired by the biological neuron model and uses neurons to perform the dimension compression. Similarily to most artificial neural networks, SOMs work in two modes, training and classification (or mapping).

These networks have been used in various classification tasks in security[21], finance [22], text mining [23] and more. Further they can be used to visualise high dimensionality data such as voting patterns [24] or the structure of welfare in the world [25].

3.1 Network architecture

The self-organizing map consists of a set of neurons arranged into a grid, rectangular and hexagonal being the most common. A neuron is a vector of weights $\vec{w}$ of the same dimension as the input data. Say we have real number input data of dimension n. Then a neuron is a vector $\vec{w} = \{w_1, \ldots, w_n\}$. Each neuron in the topological map also has a typically pre-assigned position on the topological grid. An example of a SOM is presented in Figure 3.1.

A SOM consists of neurons in a grid as shown in Figure 3.1. Instead of directly indexing neurons by their position, we index neurons independently. A neuron j is placed on an position in the map, defined by $coordinate(j)$. Usually it is also necessary to define a metric of distance between these coordinates (e.g. Euclidean). Each neuron is connected to the input by a weight vector. The dimensionality of the input is n. We denote a weight from an input feature i to the neuron j in the map as w_{ij}.

In a SOM map a data sample will be mapped on the neuron that is most similar to it. Aditionally, since the topological relationships are maintained within the training set, samples mapped close to one another in the map are more similar

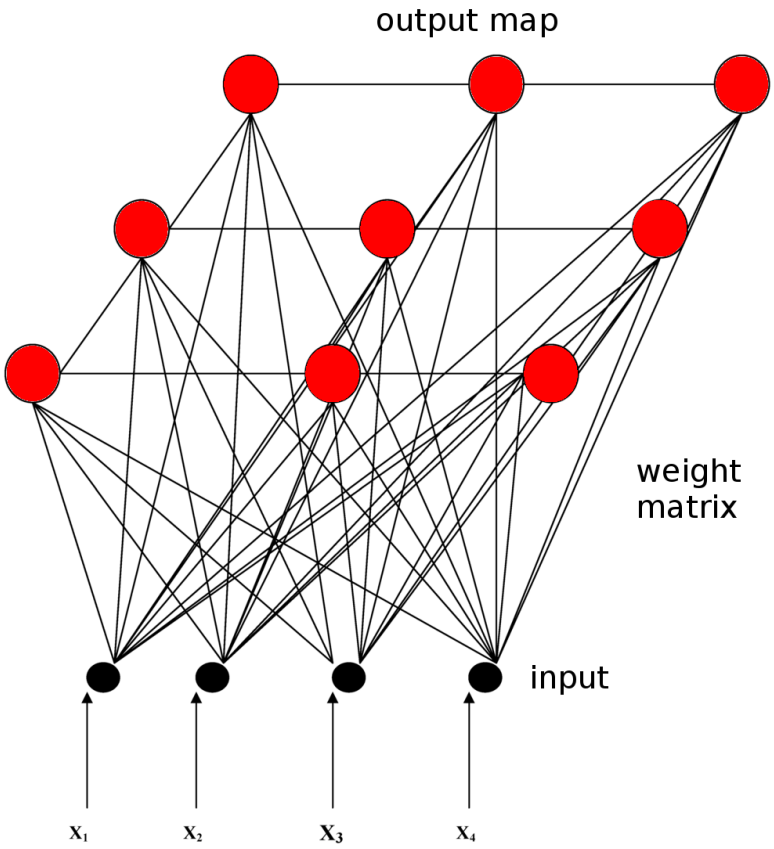


Figure 3.1: **An example SOM with a 3x3 neuron map and an input of dimension 4.**

than samples mapped further away from each other. This means that the map can be interpreted as a similarity graph of the input data and even used for clustering. The mapping of an input consists of finding the neuron most similar to the input as shown in Equation 3.1. The neuron c closest to the input is then called the winner. The equation uses a metric to determine the winner, any valid metric is applicable (e.g. Euclidean).

$$\min_c ||\vec{x} - \vec{w}^{(j)}|| \tag{3.1}$$

3.2 Training

To train a self-organizing map, we first initialize its weights with small random number and then we iteratively present it with samples and update the weights of its neurons. In each iteration t we present a sample $x^{(t)}$ and update the weights of the most active neuron in the map and its neighborhood. The most active neuron, the winner, is determined by a metric (e.g. Euclidian) to find the neuron whose weights are the closest to the presented sample as explained in equation (3.1). We denote the winner with the index c.

Afterwards the winner and its neighbors in the map are adapted towards the presented sample as described in equation (3.2).

$$\Delta w_{i,j} = \alpha(t) h_{c,j}(t)[x_i^{(t)} - w_{i,j}] \tag{3.2}$$

This equation describes how to update any arbitrary weight of the network, not just the winners weights. Instead we define a neighborhood function - $h_{c,j}(t)$ - which is a function of distance between neuron c and neuron j in the map. This parameter controls the how the winners neighbors are adapted. The adaptation speed is controlled by the learning rate $\alpha(t)$, which should converge to 0 with increasing iterations.

The parameter $h_{c,j}(t)$ controls the area in which the winners neighbors are also adapted by a rate that decays with increasing distance between the neighbor neuron and the winner neuron in the topology map. The so-called neighborhood function thus often mimics a normal distribution around the winner, but it can also be a linear function of the distance from the winner or similar. A very simple example neighborhood function is shown up in Equation 3.3 which makes the SOM adapt the winner and the neurons directly next to it in the map.

$$h_{c,j} = \begin{cases} ||coordinates(c) - coordinates(j)|| > 1 \Rightarrow 0 \\ ||coordinates(c) - coordinates(j)|| = 1 \Rightarrow 0.5 \\ ||coordinates(c) - coordinates(j)|| = 0 \Rightarrow 1 \end{cases} \tag{3.3}$$

The pseudocode of the algorithm is presented below in listing 3.1.

Listing 3.1: SOM Algorithm

```
1 Map=initmap(nxn)  # initialize the weights randomly
2 while stopping condition not reached:
3         x^(t)=random_sample # draw a sample from the data set
4         c=winner(Map,x^(t)) # find the winner neuron
5         foreach neuron i in Map:
```

The neuron weights are initalized with random values and iteratively adapted, which drives a competition between different regions in the input space for neurons in the map. This yields more dense regions of the input space to be mapped to more neurons in the map rather than sparse regions, which is why the distribution of the input space is reflected in the map. The neighborhood function also ensures preservation of the topology of the input space, such that neurons close on the topological map represent similar regions of the input space.

Chapter 4

Convolutional neural networks

Convolutional neural networks (CNN) popularised by LeCun in [26] are a special type of neural network that is based on the organisation of the neurons in an animals visual cortex [27]. They were originaly designed to process and classify two dimensional digital pictures with minimal pre-processing and are mainly used in image recognition and classification. The convolutional neural network model is based on a previous work by that describes a simpler architecture called the neocognitron [28].

The neocognitron is a multilayer neural network designed for visual recognition and classification. The neocognitron is based on two types of cells found in the visual cortex called simple and complex cells. The simple (S) and complex (C) cells are cascaded in a way where the S-cells extract local features from the input are and the C-cells look at a surrounding area of S-cells to find these features, thus tolerating some small deformations. This network architecture was successfully used to classify handwritten digits [28] and also used to classify temporal patterns in [29].

The neocognitron architecture for recognising handwritten digits was further extended by LeCun in [26] and [30]. These works introduce convolutional neural networks and a gradient descent based algorithm to train them. An example of a convolutional neural network is presented in Figure 4.1.

CNNs work in a similar manner to the neocognitron model. As with the neocognitron the process can be separated into two steps that are then repeated multiple times. The first step applies convolutions to the input to extract features from images into multiple feature maps. The second - pooling - step downsamples these feature maps, to decrease the size and introduce some resistance to deformations in the data. After the last layer a classic multilayer neural network is then used to process the output of the last pooling (subsampling) layer.

The process of convolution and pooling (subsampling) can be very successful at extracting and detecting very complicated features. This architecture was succesfully demonstrated in the field of image classification - high performance on the ImageNet database [31] in [15], and various image synthesis tasks such as image super-resolution [32] face synthesis [33] and even creative tasks such as artistic style transfer [34].

In the following text we will mainly focus on describing the CNN architecture specifically tailored for use with images, or 2D data. Nevertheless the principles described can be applied to one dimensional data aswell. This is important since

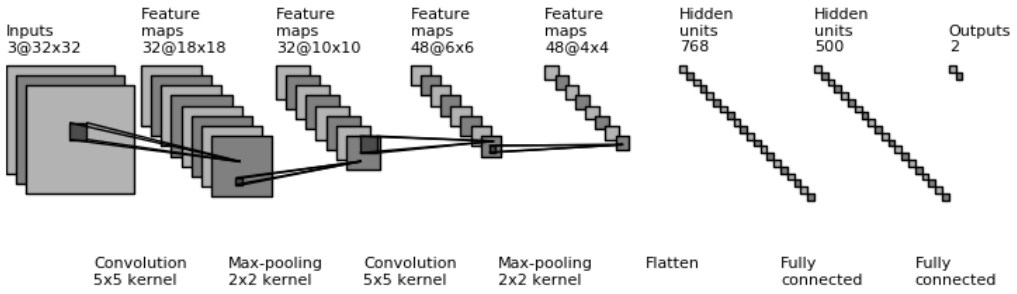


Figure 4.1: **Convolutional neural network example**

This figure shows a convolutional neural network with 2 convolutional and 2 pooling layers followed by a 2-layer dense neural network. Under each layer you can see the main parameters of the layer. The label above the layer describes the input of the layers.

the data we will be using later in the experiments will be a multidimensional signal, rather than 2D data in the sense images are. Convolutional networks have been used in the past for modeling 1D data such as sentences in natural language processing [35], [36], waveform modeling [37] and in more general signal processing [38].

4.1 Classic CNN architecture

In this section we describe the CNN architecture in more detail and how it processes input - the forward pass.

As mentioned earlier the unique functionality of CNNs consists in 2 special layers: a convolutional layer and a pooling layer. A CNN consists of pairs of these layers stacked on top of each other, followed by a classic densely connected multilayer neural network. In the following sections we describe how the forward pass works for both the convolutional layer and the pooling layer.

First however, before we explain how these layers work, we define how the input data of the network looks like. The network input is a 2-dimensional picture, for simplicity greyscale, which can be represented as a matrix of numbers ranging from a minimum to a maximum color value. An example of a small 4x4 picture is shown in Figure 4.2. In this matrix the values fit in one byte in the range 0-255, however we can also use floating point numbers from 0 to 1.

4.1.1 Convolutional layer

A convolutional layer is a layer that holds multiple convolutional filters (kernels) which are used to extract features from the input data. A convolutional filter is a matrix representing a transformation that is applied to overlapping sliding windows of the input image. The inner workings of a convolution is shown in Figure 4.3. The convolution transforms an area of $n \times n$ (in the figure 3x3). This transformation is applied on the whole picture to create the feature map. The

150	130	90	115
23	15	85	58
68	48	78	84
183	215	235	215

Figure 4.2: **A representation of a 4x4 greyscale picture, with pixel values ranging from 0 to 255**

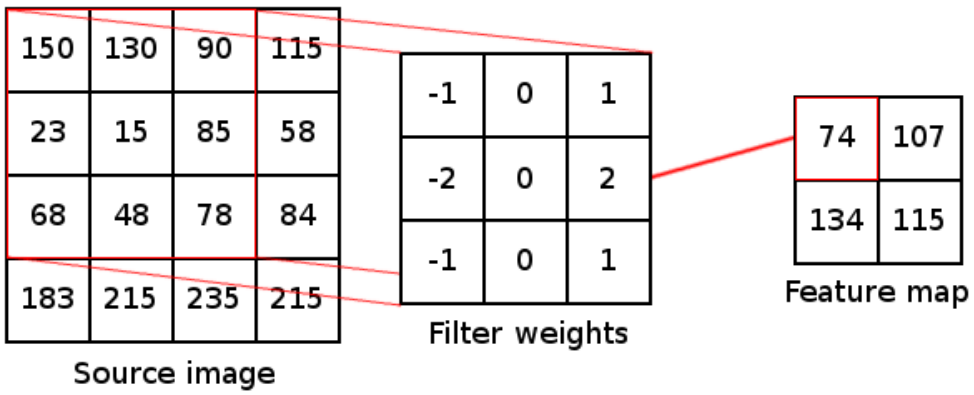


Figure 4.3: **A demonstration of a convolutional layer.**

The filter weight matrix elementwise multiplies the input window and is summed up to calculate one pixel value in the feature map. This is then repeated for all applicable input windows.

dimension of the feature map can be slightly smaller as seen in the figure because of padding.

Formally a convolutional filter is a neuron with $n \times n$ weights. The feature map is created by passing all the $n \times n$ windows of the input image one by one to the convolutional neuron and calculating its output. To formally define this process we first define a window function that selects a matrix with the pixel values surrounding a certain point in Equation 4.1. The function takes the window center $[x,y]$ and the offset of if from the edge k, to get a square filter of a width n. The offset is used to center the filter - in the case of odd sized filters we can use $k = (n-1)/2$ and in the case of even sized filters $k = n/2$.

$$I(x,y,k,n) = \begin{pmatrix} IN_{x-k,y-k} & \cdots & IN_{x,y-k} & \cdots & IN_{x-k+n,y-k+n} \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ IN_{x-k,y} & \cdots & IN_{x,y} & \cdots & IN_{x-k+n,y} \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ IN_{x-k,y-k+n} & \cdots & IN_{x,y-k+n} & \cdots & IN_{x-k+n,y-k+n} \end{pmatrix} \tag{4.1}$$

A single transformation can be then written as the dot (Frobenius) product of the input window and its weights as described in Equation 4.2. Lastly this value is passed to an activation function, various applicable activation functions are discussed in Section 2.3. To create the feature map this process is applied to each applicable coordinate. You can see that the window function is not defined for coordinates near the edges of the input picture ($x < k$, $y < k$, ...) and for these coordinates the convolution cannot be applied (unless we extend the window function). From this we get get the feature map for a given convolutional kernel W in 4.3. Which is in fact very similar to the perceptron equation from Section 2.1.

$$z(x, y, W) = \sum_{i,j} I(x,y,k,n)_{ij} W_{ij} \tag{4.2}$$

$$F_{x,y} = f(z(x,y,W)) = f(\sum_{i,j} I(x,y,k)_{ij} W_{ij}) \tag{4.3}$$

The laid out process describes how one feature map is computed using a single filter. In practice multiple filters are used in each layer as shown in Figure 4.1 creating multiple feature maps. In general all filters in one layer use the same filter width.

4.1.2 Pooling layer

The next step in the convolutional layer is the pooling step. The convolutional step alone is fairly impractical as it greatly increases the amount of data the deeper layers would have to work with. The primary utility of the pooling layer therefore is to reduce the spatial dimensions of the data.

Much like the convolutional layer described higher, the pooling layer takes a rectangular window and transforms it into one single value. In this case however the windows do not overlap, therefore pooling greatly reduces the dimensionality of the data. Furthermore, contrary to the convolutional layer the transformation applied here is fixed - no weights are being trained. There are few operations that can be used, including computing the maximum or the average across the input window. An example of a max-pooling layer is presented in Figure 4.4.

The pooling layer can formally described using some of the functions used to describe the convolutional layer. We use the window selection function as defined in Equation 4.1 to select the area to downsample. A downsampling operation is then applied to the resulting matrix, such as selecting the maximum value or calculating the average, as shown in Equation 4.4. Sometimes a transfer function can be used after the pooling layer aswell, here denoted by f, however usually a linear transfer function is used.

$$S_{x,y} = f(downsample(I(nx, ny, k))) \tag{4.4}$$

4.1.3 Summary

The CNN architecture is formed by pairs of convolutional layers and pooling layers stacked on top of each other multiple times followed by a fully connected multilayer neural network as described in Figure 4.1. The convolutional layers create feature maps by applying filters to windows as described in Equation 4.3.

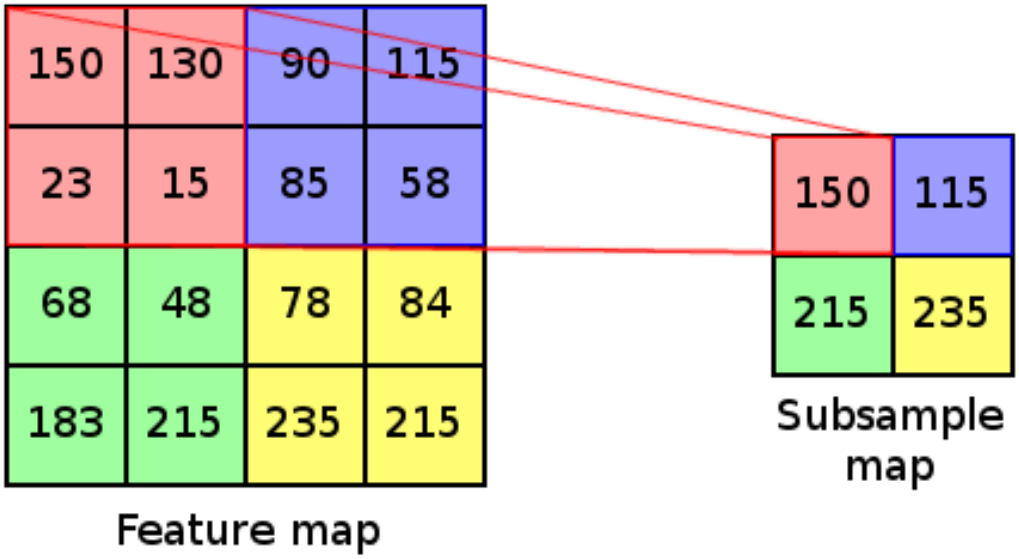


Figure 4.4: **A demonstration of a max-pooling layer.**

The pooling layer divides the feature map into non-overlapping regions and calculates a downsampling operation on each region, in this case the maximum. The source windows in the feature map correspond to the calculated output.

The pooling layers reduce the amount of data by downsampling it as shown in Equation 4.4. The output of the last pooling layer is then flattened and passed as input to a classic multilayer neural network which we described in Chapter 2.

4.2 Backpropagation

Convolutional neural networks are trained using the back propagation algorithm. In the previous section we described a CNN as a series of pairs of convolutional layers and pooling layers followed by a fully connected MLP neural network. The MLP network can be trained using the backpropagation algorith as we described in Section 2.2. The other layers of the CNN are trained using the same basic principles using the error flow passed from the MLP network.

4.2.1 Convolutional layer

So far we explained how a convolutional network acts on a fairly high level. For a simple approach to deriving a learning algorithm we can transform a convolutional layer into a multilayer neural network and then use the classic back propagation algorithm. We perform this transformation by creating a set of neurons for each filter with each neuron corresponding to a pixel. We then prune the connections to the input picture so that each neuron (representing a pixel in the feature map) is connected to its respective receptive field. Lastly we force the neurons that belong to one filter to share weights. An illustration of this process is lined up in Figure 4.5.

To derive the backpropagation we now define the forward pass process formally. For simplicity we consider a convolutional layer with only one filter, we

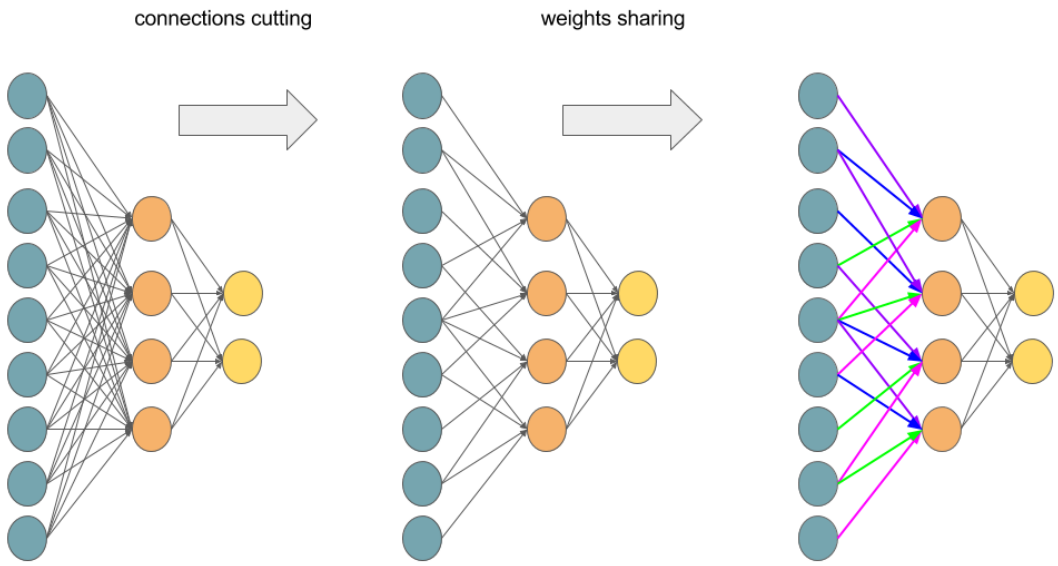


Figure 4.5: **The MLP-CNN abstraction.**

A CNN is a MLP with pruned connections and weight sharing. The first step demonstrates connection cutting, after which each neuron is only connected to its receptive field. The second step shows connection sharing.

label this layer as l-th. We divide neurons into two types, one the input neurons and the other type the filter neurons, while each neuron corresponds to a pixel. We denote the weight vector representing the filter $w_{i,j}$ with i,j describing the position in the filter matrix. The potential of a filter neuron at coordinates $[x,y]$ then is

$$z^l_{x,y} = \sum_{i,j} w_{i,j}.a^{l-1}_{i+x,j+y} \tag{4.5}$$

For simplicity we dont define the ranges of i and j, even though we could range the indexes from $-k$ to $n-k$. In the equation 4.5 we use $a^{l-1}_{i+x,j+y}$ which denotes values of pixel in a picture (or a feature map). The input layer is a picture or a pooling map and these values are their pixel values. To define the activation of neurons in the convolutional layer at coordinates $[x,y]$ we have

$$a^l_{x,y} = f(z^l_{x,y}) \tag{4.6}$$

where f is an activation function (for example sigmoid).

To derive the backpropagation algorithm for the convolutional layer lets have some error function E (for example MSE as used in Section 2.2), and let us have given the error flow to the convolutional layer from the higher layer. Assuming we have the error flow $\frac{\partial E}{\partial a^l_{x,y}}$ we need to compute the error flow to the previous layer - $\frac{\partial E}{\partial a^{l-1}_{x,y}}$. Also we need to derive the gradient component for each weight of the matrix w - $\frac{\partial E}{\partial w_{i,j}}$.

First let us look at the gradient component for each weight and simplify by applying the chain rule. Since the weight $w_{i,j}$ is used in every single potential

calculation $z_{x,y}$ we get the equation:

$$\frac{\partial E}{\partial w_{i,j}} = \sum_{x,y} \frac{\partial E}{\partial z^l_{x,y}} \frac{\partial z^l_{x,y}}{\partial w_{i,j}} = \sum_{x,y} \frac{\partial E}{\partial z^l_{x,y}} a^{l-1}_{x+i,y+j} \tag{4.7}$$

We have to sum over all the neurons in the filter layer since they all share the weight $w_{i,j}$. The identity $\frac{\partial z^l_{x,y}}{\partial w_{i,j}} = a^{l-1}_{x+i,y+j}$ follows simply from the Equation 4.5.

To to compute this sum, we need to calculate the deltas $\delta^l_{x,y} = \frac{\partial E}{\partial z^l_{x,y}}$ using the error flow we have from the previous layer $\frac{\partial E}{\partial a^l_{x,y}}$. The result in Equation 4.8 follows from the chain rule and Equation 4.6.

$$\frac{\partial E}{\partial z^l_{x,y}} = \frac{\partial E}{\partial a^l_{x,y}} \frac{\partial a^l_{x,y}}{\partial z^l_{x,y}} = \frac{\partial E}{\partial a^l_{x,y}} f'(z^l_{x,y}) \tag{4.8}$$

And we can substitute this result into Equation 4.7to get the weights gradient component shown in Equation 4.9

$$\frac{\partial E}{\partial w_{i,j}} = \sum_{x,y} \frac{\partial E}{\partial a^l_{x,y}} f'(z^l_{x,y}) a^{l-1}_{x+i,y+j} \tag{4.9}$$

What we have left is to compute the error flow to the previous layer, which we can derive using the chain rule:

$$\frac{\partial E}{\partial a^{l-1}_{x,y}} = \sum_{i,j} \frac{\partial E}{\partial z^l_{x-i,y-j}} \frac{\partial z^l_{x-i,y-j}}{\partial a^{l-1}_{x,y}} \tag{4.10}$$

In this equation we perform a sort of backward convolution. As instead of looking which input pixels contribute toward a convolution, we look at which convolutions use a certain pixel. When we take a look at Equation 4.5 we can see that the identity $\frac{\partial z^l_{x-i,y-j}}{\partial a^{l-1}_{x,y}} = w_{i,j}$ holds. Therefore we get the error flow to the previous layer as:

$$\frac{\partial E}{\partial a^{l-1}_{x,y}} = \sum_{i,j} \frac{\partial E}{\partial z^l_{x-i,y-j}} w_{i,j} \tag{4.11}$$

Where $\frac{\partial E}{\partial z^l_{x-i,y-j}}$ is as derived in Equation 4.8. This result is fairly intuitive as the error flow is a weighted sum of the error flows from the previous layer, which is very similar to how the classic backpropagation algorithm acts.

In conclusion we have 3 main results. The gradient component for the neuron potentials (deltas) $\frac{\partial E}{\partial z^l_{x,y}}$ which is derived in Equation 4.8. The gradient component for the filter weights $\frac{\partial E}{\partial w_{i,j}}$, which we will use to update them, derived in Equation 4.9 which uses the error flow from the previous layer. Finally we have the error flow to the previous layer $\frac{\partial E}{\partial a^{l-1}_{x,y}}$ which is shown in Equation 4.11 and which enables more than one convolutional layer architectures to learn.

4.2.2 Pooling layer

The pooling layers do not actually do any learning themselves as their parameters (if any) are fixed. They simply reduce the size of the feature map by a

factor. However we still need to have error flow through them, and that is what we derive in this section.

A pooling operation takes a block of $n \times n$ pixels from the feature map and reduces them to a single value. Each pooling operation is associated with a certain block, described by its coordinates. Lets define the output of the pooling operator on coordinates $[x,y]$ as $a^l_{x,y}$ (where l denotes the layer) and we have given the error flowing from the following layer $\frac{\partial E}{\partial a^l_{x,y}}$. We need to propagate this error to the corresponding neurons in the previous layer - $\frac{\partial E}{\partial a^{l-1}_{x,y}}$.

In the case of a average-pooling operator we have $a^l_{x,y} = \frac{1}{n^2} \sum_{i,j} a^{l-1}_{nx+i,ny+j}$ and then we have the error flow:

$$\frac{\partial E}{\partial a^{l-1}_{x,y}} = \frac{\partial E}{\partial a^l_{x,y}} \frac{\partial a^l_{x,y}}{\partial a^{l-1}_{x,y}} = \frac{1}{n^2} \frac{\partial E}{\partial a^l_{x,y}} \tag{4.12}$$

In the case of a max-pooling operator $a^l_{x,y} = \max_{i,j} a^{l-1}_{nx+i,ny+j}$ and then the error flows fully through the maximum selected value, and its 0 for the other ones.

Chapter 5

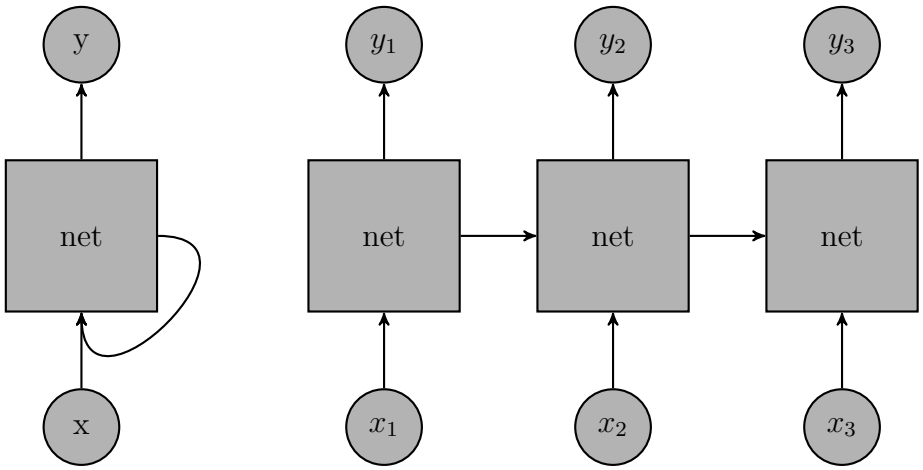
Recurrent neural networks

Recurrent neural networks (RNNs) are a type of artificial neural networks where the connections between the neurons in the network form directed (feed-back) cycles as seen in Figure 5.1. As with MLPs there are many architectures of RNNs such as Hopfield networks [39], Boltzmann machines [40], Echo state networks [41] and Long-short term memory networks [42]. An example RNN can be seen in figure 5.1.

The feedback cycles on neurons makes RNNs able to memorize relevant data over time, acting as a memory cell, which in principle can make them more powerful than standard feed-forward networks. An MLP can only map from input to output vectors, whereas an RNN can map from the entire history of previous inputs. To gain similar information in a MLP we would have to provide it the whole history we provide an RNN, which in turn increases the number of hyperparameters to train, and might introduce noise. An equivalent result to the universal aproximation theory for MLPs is that a RNN can map a sequence to sequence with enough hidden units [43].

Despite of the theoretical strength of the model, the early variants of RNNs had very limited success. The main issue of the classic variants of RNNs is a rapid decay of the error propagated through time, also called the vanishing gradient problem [44] [45]. The vanishing gradient problem is that the gradient gets very low even after a few time-steps of an RNN. An example of the vanishing gradient problem is depicted in Figure 5.2. This causes the RNNs to only ever remember information for a few time-steps. In this chapter we will focus on Long-short term memory networks introduced in [42] that attempt to overcome the vanishing gradient problem of traditional RNNs by introducing a constant error flow into the network. This model has been also further expanded in [46].

LSTMs have had great success in recent years in many different applications. They have been used in the field of speech recognition [47], text recognition [48], natural language processing for phoneme classification [49]. LSTMs have proven themselves to be very effective when compared to other adaptive approaches that keep no internal state (hidden markov models, support vector machines and feedforward networks) [50] [51].



(a) Recursive neural network with the feedback loop

(b) Time-series unfolded RNN

Figure 5.1: **A recursive neural network.**

On the left there is a depiction of a RNN with the recursive connection. When used for an input over 3 time-steps, we can depict the process as a network with connections between time-steps that pass information from the older time-steps to the newer ones.

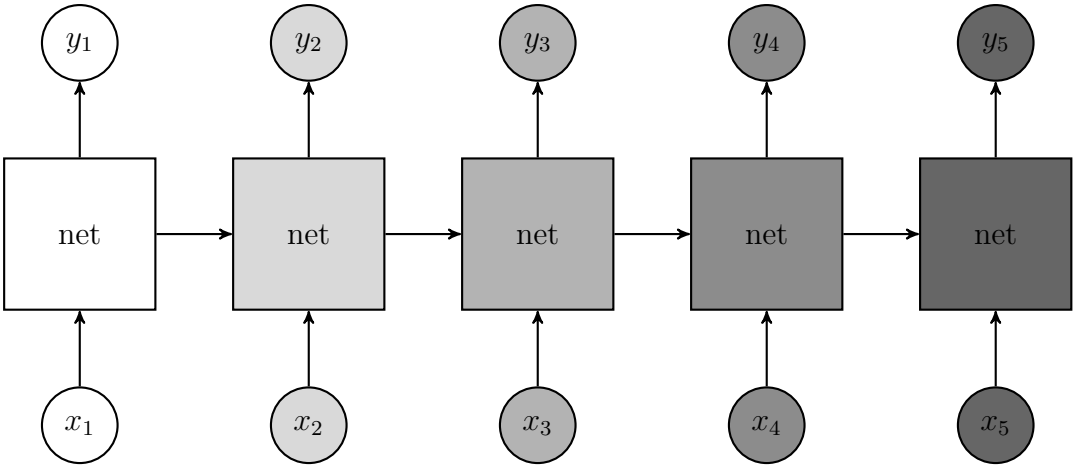


Figure 5.2: **Vanishing gradient in a recursive neural network.**

The shading of the nodes depict how much of the information from the first time step is available. The information about the first time step decays over time.

5.1 Long-short term memory network architecture

Long-short term memory networks (LSTMs) are a subclass of recursive neural networks capable of learning long term dependencies in the data. The model was first introduced in [42]. They were explicitly designed to learn long term dependencies as classic RNNs were ineffective at that task.

The main issue solved by LSTM is the rapid decay of error propagation over time. The core principle that solves decaying information flow in LSTM is a memory cell, which is a recurrently connected linear unit. The linear recurrence of the memory cell is what causes the error flow to remain constant.

This memory cell in an abstract way acts similarily as an electronic memory cell does. The LSTM network can read, write or erase data from it. These memory cells are often arranged into memory cell blocks, where all cells in one block share some of their weights. The blocks (or just single cells) are then arranged into layers which of there can be multiple in one network. TODO obrazek

The processes of reading, writing and erasing the data from the memory cell are controlled by structures called gates; cell, input, output and more recently a forget gate which was proposed in [46] as seen in a depiction of a single memory cell in Figure 5.3. First the cell gate calculates a candidate input value which then is used with conjunction with the input gate to generate a new state of the memory cell. The forget gate controls the memory cell state decay, to what amount will the old memory cell state be retained. Finally the output gate uses both the cell memory and the network input to determine when to output infromation for use in the network.

For example we have a time series of two inputs, x_1, x_2. The task of the LSTM is to count the number of times the input x_1 is high and output this value whenever x_2 is high. The LSTM trains its input gate so that it fires whenever x_1 is high, which causes the input to the cell to be open and in turn increments the value in the memory cell. The LSTM also trains the output gate to fire whenever x_2 is high which in turn makes the network output the value of the memory cell. We could also expand this example by adding another input that tells the LSTM to erase the memory cell value. An expanded example that sums real numbers on a separate input was presented in [42].

5.2 Single cell forward pass

In this section I explain how a single memory cell computes its state and its output.

The input of a single cell is the output of the previous layer (or the network input), we denote this as $\vec{x}$, along with the recurrent connections, which we will denote $\vec{h}$. A single memory cell is composed of four gates, the cell, input, forget and output gate and its memory state. Each gate can be represented by a artificial neuron, where the gate is connected to the network and recurrent inputs $\vec{x}$ and $\vec{h}$ with a weighted connection. The weight vectors for different gates will be denoted $\vec{w}_{cell}$, $\vec{w}_{input}$, $\vec{w}_{forget}$ and $\vec{w}_{output}$. The different transfer functions will be discussed

along with the gate definitions. Lastly we define the cell state in time t as $s(t)$ along with the output of the cell at time t as $a_c(t)$. The cell state will be denoted by $s(t)$.

The goal of the forward pass is to update the cell state - calculate $s(t+1)$ and to get the cell output $a_c(t)$. This computation can be divided into five steps, each controlled by one of the gates:

1. Compute a candidate value to store in the cell a_{cell} - controlled by the cell gate.
2. Compute how much of the candidate value will be stored a_{input} - controlled by the input gate.
3. Compute how much of the original cell state will be retained a_{forget} - controlled by the forget gate.
4. Update the cell state using the gate outputs - $s(t+1)$.
5. Compute the cell output value $a_c(t)$ - controlled by the output gate.

The first step is for the memory cell to determine a candidate for what could be stored in the cell - a_{cell}. The output of the gate is calculated as a artificial neuron as shown in Equation 5.3. The transfer function used is often a tanh scaled to the range [−2,2] in Equation 5.2.

$$\begin{aligned} z_{cell}(t) &= \vec{w}_{cell}(t) \cdot [\vec{x}(t),\vec{h}(t), 1] \\ a_{cell}(t) &= g(z_{cell}(t)) \end{aligned} \tag{5.1}$$

$$g(x) = \frac{4}{1+e^{-x}} - 2 \tag{5.2}$$

The second step is to determine how much a_{input} of the candidate value a_{cell} will be actually stored in the state $s(t+1)$. This is controlled by the input gate, which again is an artificial neuron. The input gate output is shown in Equation 5.3. The transfer function used is most commonly a simple sigmoid in Equation 5.4. We will then use the neuron output - a_{input} to scale the candidate value from the cell gate in Equation 5.6.

$$\begin{aligned} z_{input}(t) &= \vec{w}_{input}(t) \cdot [\vec{x}(t),\vec{h}(t), 1] \\ a_{input}(t) &= g(z_{input}(t)) \end{aligned} \tag{5.3}$$

$$f(x) = \frac{1}{1+e^{-x}} \tag{5.4}$$

The third step in the computation of the cell output $a_c(t)$ is to decide how much of the cell state $s(t)$ is retained between time steps - a_{forget}. The forget gate takes a look at the network input and the recurrent input and outputs a number between 0 and 1 which will determine how much information the cell state retains. The output of the forget gate can be written again as a artificial neuron as shown in Equation 5.5. The transfer function used is most commonly a sigmoid in Equation 5.4.

$$\begin{aligned} z_{forget}(t) &= \vec{w}_{forget}(t) \cdot [\vec{x}(t),\vec{h}(t), 1] \\ a_{forget}(t) &= g(z_{forget}(t)) \end{aligned} \tag{5.5}$$

We can now show how the cell state is updated during the forward pass using the the cell, input and forget gates. The network calculates its cell, input and forget gate activations - $a_{cell}, a_{input}, a_{forget}$ as defined in Equations 5.1, 5.3 and 5.5. These values are then used to calculate the cell state using Equation 5.6. The candidate value is scaled by the input gate and added to the cell state that is scaled by the forget gate.

$$s(t+1) = a_{input}(\vec{x}(t+1),\vec{h}(t+1))a_{cell}(\vec{x}(t+1),\vec{h}(t+1))+a_{forget}(\vec{x}(t+1),\vec{h}(t+1))s(t) \tag{5.6}$$

The last step left to compute is the memory cell output $a_c(t)$, which is determined by the output gate. The output gate is again an artificial neuron as shown in Equation 5.7.

$$\begin{aligned} z_{output}(t) &= \vec{w}_{output}(t) \cdot [\vec{x}(t),\vec{h}(t), 1] \\ a_{output}(t) &= g(z_{output}(t)) \end{aligned} \tag{5.7}$$

We then use the output gate activation a_{output} to compute the cell output $a_c(t)$. We first squash the cell state by a transfer function and then scale it by the output gate value as shown in Equation 5.8. The transfer function used is most commonly a tanh function in Equation 5.9. The index c denotes the index of the cell in question.

$$a_c(t) = a_{output}(\vec{x},\vec{h})h(s(t)) \tag{5.8}$$

$$h(x) = \frac{2}{1+e^{-x}} - 1 \tag{5.9}$$

These equations describe how a single memory cell is updated in the forward pass. In the LSTM there can be multiple memory cells and often memory cells are arranged into memory cell blocks. A memory cell block is a set of memory cells that share input, output and forget gate weights. Note the blocks do not share cell gates, as they are meant to store vectors of information based on the same triggers.

5.3 LSTM Backpropagation

The original LSTM training algorithm uses an approximate error gradient using back-propagation through time (BPTT) [52] trunctated after one time-step. Since then there have been developed other ways to train LSTMs, such as using an exact error gradient with an untrunctated BPTT [53]. There are also a few non-gradient-based methods [54], [55].

In this section we summarise the trunctated BPTT version of the training algorithm. The trunctated formula approximates the partial derivatives by cutting off the error flow once it leaves memory cells or gate units.

We assume one layer of LSTM memory cell blocks and one layer of output neurons densely connected to the LSTM cells. The neurons in the output layer will be denoted by the index o. Memory cells are arranged into memory cell blocks, each indexed by j, the size of a block is denoted S_j. Memory cell blocks share input, forget and output gate weights, therefore these gates are indexed directly - e.g. out_j. The cell gate and the cell state are indexed by the memory

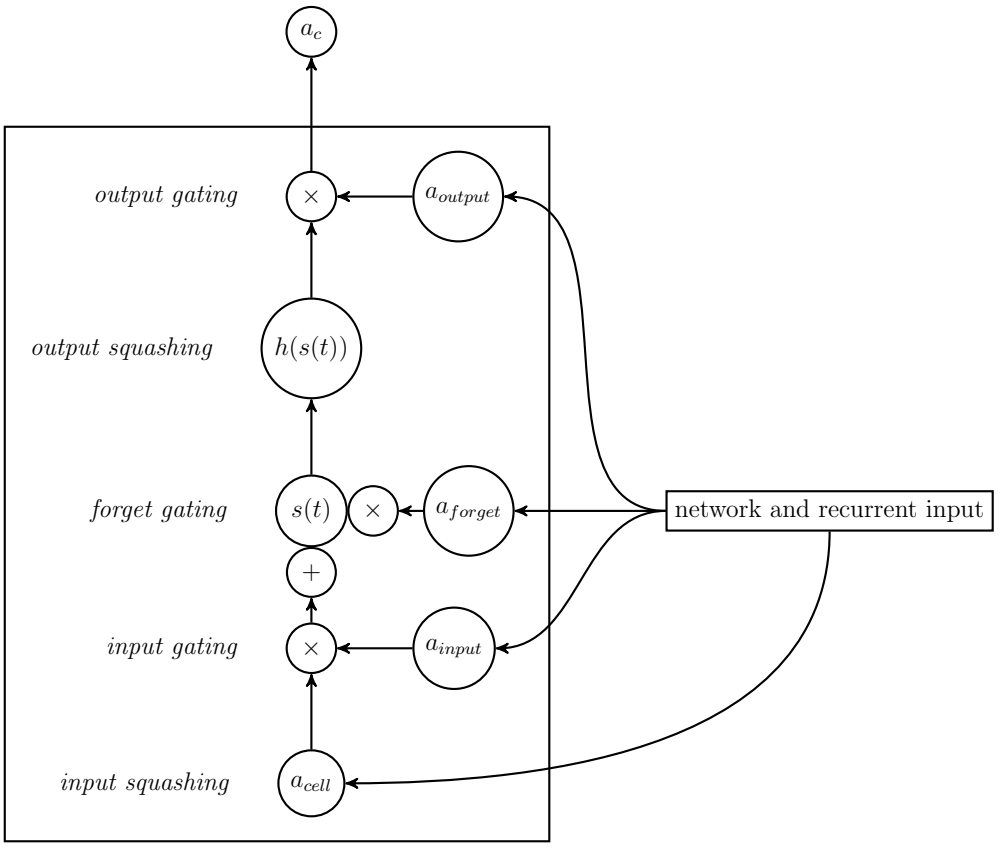


Figure 5.3: **A computational graph of a LSTM memory cell.**

The LSTM creates a new memory state by scaling it (a_{forget}) and adding a new state from $a_{input}.a_{cell}$. The cell state is then squashed and passed through a scaling gate a_{output} as the memory cell output.

cell index c_j^v, where v is the index of the cell within the cell block j. For example the cell state of the memory cell v in the cell block j is denoted by $s_{c_j^v}$. Each output neuron has an activation a_o and a potential z_o as defined in the MLP forward pass in Section 2.1. We use this notation in a similar fashion for gates. We do not use a index for the layer, but each result is dependent on time, which is put in brackets after the property of the neuron, such as $a_o(t)$. The weight $w_{l,m}$ denotes the connection outgoing from the unit l to the unit m.

First we define the squared error over all output units.

$$E(t) = \sum_o e_o(t)^2 \tag{5.10}$$

$$e_o(t) = \frac{\partial E(t)}{\partial a_o(t)} = \vec{d}_o(t) - a_o(t) \tag{5.11}$$

where a_o is the output on the o-th output node and $\vec{d}_o(t)$ the o-th component of the desired output. If this layer was not the final layer but for example was followed by another layer of perceptrons we could inject the error flow as necessary in e_o. In a similar fashion as in the MLP backpropagation we minimize the error using gradient descent by iteratively adding the changes $\Delta w_{l,m}$ to every applicable weight.

First we compute the contribution to a general $w_{l,m}$'s gradient based update with learning rate α as shown in Equations 5.12, 5.13, 5.14, 5.15. These equations follow the chain rule and BPTT. Our goal is to compute the gradient component for weights connected to the output units and the output gates.

$$\Delta w_{l,m}(t) = -\alpha \frac{\partial E(t)}{\partial w_{l,m}} \stackrel{1}{=} -\alpha \sum_o \frac{\partial E(t)}{\partial a_o(t)} \frac{\partial a_o(t)}{\partial w_{l,m}} \tag{5.12}$$

$$\stackrel{2}{=} \alpha \sum_o \sum_i e_o(t) \frac{\partial a_o(t)}{\partial a_i(t)} \frac{\partial a_i(t)}{\partial z_i(t)} \frac{\partial z_i(t)}{\partial w_{l,m}} \tag{5.13}$$

$$\stackrel{3}{=} \alpha \sum_o \sum_i e_o(t) \frac{\partial a_o(t)}{\partial a_i(t)} \frac{\partial a_i(t)}{\partial z_i(t)} \left(\delta_{i,m} a_l(t-1) + \frac{\partial z_i(t)}{\partial a_l(t-1)} \right) \tag{5.14}$$

$$\stackrel{4}{=} \alpha \left(\sum_o e_o(t) \frac{\partial a_o(t)}{\partial a_m(t)} \right) \frac{\partial a_m(t)}{\partial z_m(t)} a_l(t-1) \tag{5.15}$$

$$\stackrel{4}{=} \alpha \delta_m(t) a_l(t-1) \tag{5.16}$$

$$\delta_m(t) := \left(\sum_o e_o(t) \frac{\partial a_o(t)}{\partial a_m(t)} \right) \frac{\partial a_m(t)}{\partial z_m(t)} \tag{5.17}$$

The first identity in equation 5.12 follows simply from the chain rule on all the output units o. We then substitute $e_o(t)$ and the second identity in 5.13 follows from a chain rule on a gate or cell unit i (we will later subset these to simplify further). The third identity in equation 5.14 follows from the backpropagation through time principle ($\delta_{i,l}$ is the Kronecker delta 5.18). In the fourth identity 5.15 we trunctate the error when it leaves a memory block by setting the derivative $\frac{\partial z_i(t)}{\partial a_m(t-1)} = 0$ and simplify the result to account for the Kronecker delta.

$$\delta_{i,j} = \begin{cases} 1, & \text{if } i = j. \\ 0, & \text{otherwise.} \end{cases} \tag{5.18}$$

In Equation 5.17 we isolated the delta component of the gradient. For weights coming to output units, we set $m = o'$. The equation in then reduces to:

$$\delta_{o'} = e_{o'}(t) f'_{o'}(z_{o'}(t)) \tag{5.19}$$

In a similar fashion we can simplify for weights incoming to output gates. We set $m = output_j$ and then simplify to 5.20 by differentiating Equations 5.8 and 5.7 from the forward pass.

$$\delta_{output_j} = f'_{output_j}(z_{output_j}(t)) \left(\sum_{v=1}^{S_j} h(s_{c_j^v} \sum_o w_{c_j^v o} \delta_o(t) \right) \tag{5.20}$$

The delta for the output gate sums over all the cells in the memory block as every one contributes to the weight change on the output gate.

From Equations 5.16, 5.17, 5.20 and 5.19 we get the weight update formulas for the output neurons and the output gates.

To calculate the gradient component of weights connected to the other gates, we first define the error on the cell state and then propagate this error to the gates, since each gate contributes to the cell state. We define an internal cell state error $e_{s_{c_j^v}}(t)$ as shown Equation 5.21 which we can compute as 5.22 from the differentiation of 5.20.

$$e_{s_{c_j^v}}(t) := -\frac{\partial E(t)}{\partial s_{c_j^v}(t)} \tag{5.21}$$

$$= f'_{output_j}(z_{output_j}(t)) h'(s_{c_j^v}) \sum_o w_{c_j^v o} \delta_o(t) \tag{5.22}$$

We can then use this error to compute the gradient contribution according to:

$$\Delta w_{l,m}(t) = -\alpha \frac{\partial E}{\partial w_{l,m}} = -\alpha \frac{\partial E(t)}{\partial s_{c_j^v}(t)} \frac{\partial s_{c_j^v}(t)}{\partial w_{l,m}} = \alpha e_{s_{c_j^v}}(t) \frac{\partial s_{c_j^v}}{\partial w_{l,m}} \tag{5.23}$$

In the above equation we already computed the term $e_{s_{c_j^v}}(t)$. To obtain the partial $\frac{\partial s_{c_j^v}(t)}{\partial w_{l,m}}$ we differentiate Equation 5.6 from the forward pass to get:

$$\begin{aligned} \frac{\partial s_{c_j^v}(t)}{\partial w_{l,m}} = & \frac{\partial s_{c_j^v}(t-1)}{\partial w_{l,m}} a_{forget_j}(t) + \frac{\partial a_{forget_j}(t)}{\partial w_{l,m}} s(t-1) \\ & + \frac{\partial a_{input_j}(t)}{\partial w_{l,m}} a_{cell_{c_j^v}}(t) + \frac{\partial a_{cell_{c_j^v}}(t)}{\partial w_{l,m}} a_{input_j}(t) \end{aligned} \tag{5.24}$$

Forturnately these partials are zero unless correct weight connections are set. For those we can differentiate forward pass equations 5.1 5.3 and 5.5 to obtain:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{l,cell_{c_j^v}}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{l,cell_j}} a_{forget_j}(t) + g'(z_{cell_{c_j^v}}(t)) a_{input_j}(t) a_m(t-1) \tag{5.25}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{l,input_j}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{l,input_j}} a_{forget_j}(t) + g(z_{cell_{c_j^v}}(t)) f'(z_{input_j}(t)) a_m(t-1) \tag{5.26}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{l,forget_j}} = \frac{\partial s_{c_j^v}(t-1)}{\partial w_{l,forget_j}} a_{forget_j}(t) + s_{c_j^v}(t-1) f'(z_{forget_j}(t)) a_m(t-1) \tag{5.27}$$

We can now take these partials and substitute them as necessary into equation 5.23 to obtain the weight updates for the input, cell and forget gates. To update weights of the input and forget gate, we have to sum over the contributions of all the cells in the cell block as shown in equation 5.28.

$$\forall m \in \{forget_j, input_j\} : \Delta w_{l,m}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) \frac{\partial s_{c_j^v}}{\partial w_{l,m}} \tag{5.28}$$

The equation for the cell gate is simply

$$\forall m \in \{cell_j\} : \Delta w_{l,m}(t) = \alpha e_{s_{c_j^v}}(t) \frac{\partial s_{c_j^v}}{\partial w_{l,m}} \tag{5.29}$$

To summarize the backpropagation process, first we calculate deltas for the output neurons 5.19 and the output gate 5.20. We use these deltas to substitute in the classic weight update equation 5.16 to update the incoming and outgoing output gate weights. To update the other gate weights, we have to first propagate the error from the output through the output gates to a cell state error as defined in 5.22. Each cell has to hold variables during training that save the cell state error component of weights incoming to the input, cell and forget gates, as defined in equations 5.25, 5.26 and 5.27. These equations define a recursive update that is performed each step. The initial values can be set at 0. These variables are then used to update the incoming weights of input, forget and cell gates according to equations 5.28 and 5.29.

Chapter 6

Experimental results

Monetary policy-makers tend to consider forecasts of economic activity for their decision making. In general any monetary policy applied to a national economy may change the economy in the long run. In order to make informed decisions policy-makers often require predictions and other analysis of the economy. The indicators of GPD and GDP growth rate being the most prominent are usually forecasted by means of macroeconomic aggregates such as investment and consumption indexes. In this chapter, our goal will be to analyse multidimensional macroeconomic time-series with these goals in mind.

In the following chapter we will first describe the data in Section 6.1 and visualise the macroeconomic development of different countries in Section 6.2. We will then explore the task of GDP prediction and analyse the applicability of different types of deep neural networks for prediction in Section 6.3.

6.1 Data

The data used in the experiments had been collected by the UN National Accounts Main Aggregates Database [56] supplemented with datapoints from the World Bank Open Database [57]. The database created contains yearly national aggregates for 198 countries in total for the time period from 1970 to 2015. In the following section we will describe the data and perform some exploratory analysis.

The dataset is collected from 2 sources, the UN National Accounts Main Aggregates Database and the World Bank Open Database. Most of the economic indicators are available from the UN database and we augmented this database with a few socioeconomic indicators from the WB. The indicators from the WB are marked in the list below: Internet coverage, Life expectancy and Population growth. During the training we used 16 different macroeconomic indicators in total which are listed below. The list contains indicators representing the share of a specific branch of business, these branches are formally defined by the UN and are called ISIC divisions [58], for the scope of this work the short description will suffice.

- GDP per capita - Gross Domestic Product per citizen. UN Indicator
- Absolute GDP - Gross Domestic Product absolute value. UN Indicator

- Agriculture share of GDP - Value added in agriculture, forestry and fishing. ISIC divisions A-B as a share of the country GDP. UN Indicator
- Mining share of GDP - Value added in mining. ISIC divisions C-E as a share of the country GDP. UN Indicator
- Construction share of GDP - Value added in construction ISIC division F as a share of the country GDP. UN Indicator
- Wholesale share of GDP - Value added in wholesale, retail, restaurants and hotels. ISIC divisions G-H as a share of the country GDP. UN Indicator
- Transport share of GDP - Value added in transportation. ISIC divisions I as a share of the country GDP. UN Indicator
- Other share of GDP - Value added in other activities. ISIC divisions J-P as a share of the country GDP. UN Indicator
- General government final consumption expenditure - Government expenditure on goods and services that are used for the direct satisfaction of individuals or the community as a share of GDP. UN Indicator
- Household consumption expenditure - Expenditure incurred by resident households on individual consumption of goods and services as a share of GDP. Household final consumption expenditure consists of the expenditure, including imputed expenditure, incurred by resident households on individual consumption goods and services, including those sold at prices that are not economically significant. UN Indicator
- Gross capital formation - Gross fixed capital formation is measured by the total value of a producer's acquisitions, less disposals, of fixed assets during the accounting period plus certain additions to the value of non-produced assets (such as subsoil assets or major improvements in the quantity, quality or productivity of land) realised by the productive activity of institutional units. UN Indicator
- Exports of goods and services share of GDP - Represents the value of goods and services provided to the rest of the world. UN Indicator
- Imports of goods and services share of GDP - Represents the value of goods and services recieved from the rest of the world. UN Indicator
- Internet coverage - Internet users per 100 people. WB Indicator
- Life expectancy - The number of years a infant is expected to live with mortality patterns current at its birth. WB Indicator
- Population growth - Yearly population growth rate as a percentage. WB Indicator

Some basic statistical parameters of the indicators are described in Table 6.1. Moreover the distrubutions of every indicator are charted in Appendix A. The

indicator	mean	std	min	max
Agriculture (ISIC A-B) - % of GDP	16.48	14.85	0.03	80.51
Mining (ISIC C-E) - % of GDP	23.33	13.64	0.05	93.78
Construction (ISIC F) - % of GDP	6.19	3.03	0.14	28.06
Wholesale (ISIC G-H) - % of GDP	14.99	6.50	0.57	51.96
Transport (ISIC I) - % of GDP	7.9	3.71	-5.23	29.09
Other (ISIC J-P) - % of GDP	30.18	12.18	1.40	78.82
General gov. final cons. expend. - % of GDP	17.86	10.27	1.11	201.02
Household cons. expend. - % of GDP	63.85	18.66	3.85	179.22
Gross capital formation - % of GDP	24.40	10.18	-13.41	113.31
GDP Per Capita - US dollars	7078.44	13.40e3	33.88	157.09e3
Absolute GDP - US dollars	172.29e9	783.85e9	2.58e6	17.34e12
Exports of goods and services - % of GDP	35.53	29.63	-2.46	295.75
Imports of goods and services - % of GDP	42.79	30.12	1.60	297.94
Internet users per 100 people	8.67	19.37	0.00	98.16
Life expectancy at birth - years	64.11	11.55	19.50	83.33
Population growth - % of population	1.74	1.60	-10.96	17.62

Table 6.1: **Statistical properties of indicators**

distributions of the indicators are fairly normal, and that suggests the indicators do not have to be split further.

There are a few interesting outliers that can be noticed in the data around the extremes. For example the minimum life expectancy being only 19.5 years old comes from the datapoints of Cambodia around the period of the Cambodian Civil War. Another worrying value is a negative exports share of GDP, which presented itself in the Communist Czechoslovakia. The datapoints in the Eastern Bloc are in general not very trustworthy and this datapoint is probably not actual exports, but maybe an aggregate of imports minus exports. Another interesting extreme is the transport share of GDP minimum value being negative. This value appears in the datapoints of Marshall Islands and its most likely incorrect data, even though it could be explained by very heavy investments into transportation with no returns, possibly the government spent a lot of money on infrastructure. Since the goal of this work was not to create a clean database of macroeconomic indicators, we decided to leave these inaccuracies in the data and use the data as was provided.

6.1.1 Data preprocessing

Missing values

The data from UN has farily few missing datapoints. The main missing datapoints come from newly formed countries after the fall of the Soviet Union. Fortunately generally speaking both the former country and the newly formed country was present in the data. Therefore the missing data in the countries newly formed could have been filled in from the country it was previously part of. For example the datapoints from Czechoslovakia were used to fill in the missing data in Czech republic before 1990. Even though this might introduce some inaccuracies into the data, we felt this replacement to be necessary since in total

it was required for 27 countries.

After this former country replacement we used interpolation and k-NN to replace more missing values in a similar manner as is described in [59]. There was in total there 318465 datapoints in the database from which there was 8193 (2.5% of total datapoints) missing values replaced with k-NN or interpolated. If only a few values (4) were missing in an otherwise complete timeseries (of any indicator) interpolation was used to fill in the blank values. In total 666 (8% of missing datapoints) values were replaced using interpolation. If there was more datapoints missing in a time series, we employed the k-NN algorithm to find a country most similar to the other (non-missing indicators) and used that similar country to fill in the missing values. Using k-NN we replaced 7527 values in total (92% of missing datapoints). Countries that were completely missing an indicator were completely removed from the dataset.

Normalisation

The data has been min-max normalised according to Equation 6.1 in order to reduce weights in the trained networks. The min-max weights are noted in the table 6.1. This causes the data to be mapped to the interval [−1,1].

$$x* = \frac{x - min}{max - min} \tag{6.1}$$

6.2 Visualising time-series

As a part of the exploratory analysis of the data we first employ self-organising maps to visualise the data. Self-organizing maps (SOM) are used to map multidimensional data onto a two-dimensional map. One of the possible applications of SOM for multidimensional time series is to visualise the development of the time series over a period of time. In this experiment we take time series data from the dataset described in Section 6.1 to get a visual representation of how different countries economics evolved over time. The results give an insight into how specific countries develop over time in an easy-to-understand and visual way. This method however does not use the time series during its training, each time-step is considered a different, and independent sample.

The training algorithm used was an Euclidean distance 30x30 SOM and the map has been trained using an 20000-iteration algorithm that drew a random sample each time to adjust the map. For implementation details review Apendix

6.2.1 Results

The training resulted into a map that generally places well developed countries to the upper right corner, less developed countries to the left middle and top. We can see the distribution of each feature in Figure 6.1. In the heatmaps lighter colors correspond to higher values for the respective feature. We can see that there is a large overlap between import and export. Life expectancy is also mostly a complement to population growth with life expectancy. Life expectancy forms a gradient from the top left corner to the lower right corner, which corresponds to fast developing countries with a generally lower life standard. Agriculture peaks

in the upper left corner while industry in the lower middle and it overlaps highly with exports, which follows common sense. Suprisingly the highest GDP does not overlap with high import/export areas but it does overlap with very high life expectancy.

These figures help interpret the movement of each country on the map over time. Let us take a look at Bangladesh in Figure 6.2. In this figure the lighter colors of the neurons indicate a lower density area where neurons are further away to each other, while dark shades indicate high density areas where neurons are fairly close to one another. According to the World Bank [57] Bangladesh has been growing at an impressive rate. This fact that Bangladesh is a growing country is consistent with the displayed time series on the map as upwards movements correspond with an increased life expectancy and GDP.

The World Bank classifies countries each year by their GNI per capita and Bangladesh aspires to become classified by the World bank as a middle income country within the next ten years. We can examine this aspiration by examining the agreggated time series for the lower-middle and middle income classes. In Figures 6.3 and 6.4 we can see that Bangladesh is much more similar to the lower middle income class than the middle income class, but it is fairly close to the lower middle income time series. Another good example is France, for which it is immediately clear that it is a highly developed country as seen in Figure 6.5 and is one of the highest GDP/life expectancy countries in our data by 2015.

However this analysis can be easily mislead by the other variables as the income class depends only on one of the variables observed. For example Malaysia is considered a upper-middle income economy, but is placed in the lower left corner on the map as seen in Figure 6.6. This is most likely caused by Malaysia being a leading exporter in several electronic products and natural resources.

In conclusion it is important to realise that the position on the map represents all the variables that were used to construct it. This then means that in some cases one variable becomes dominant in representing the country. For example with France its GDP highly above average and therefore we cannot for example determine the Import/Export situation of France just from the map.

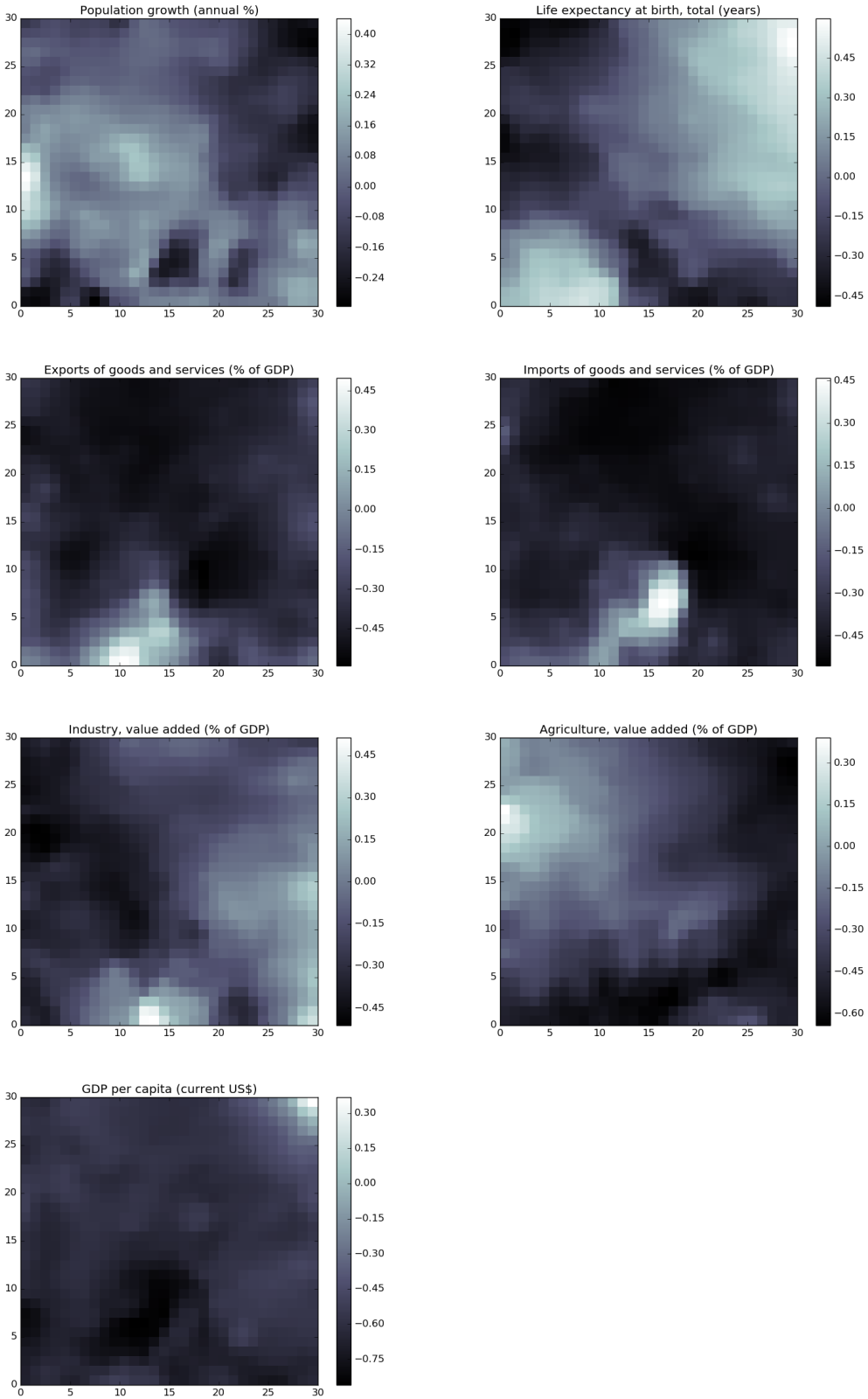


Figure 6.1: **Indicator heatmaps**

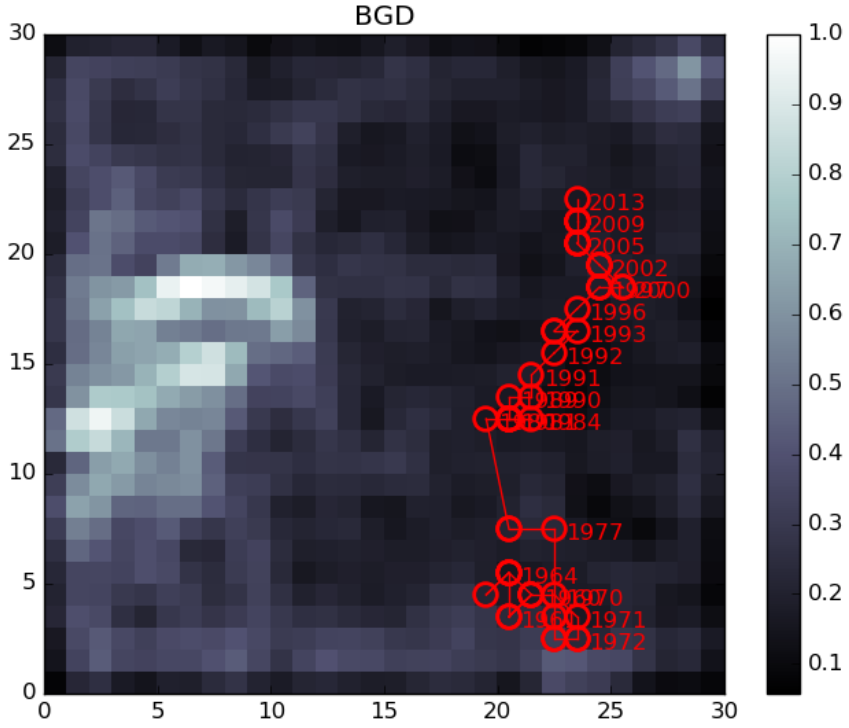


Figure 6.2: **Bangladesh time series**

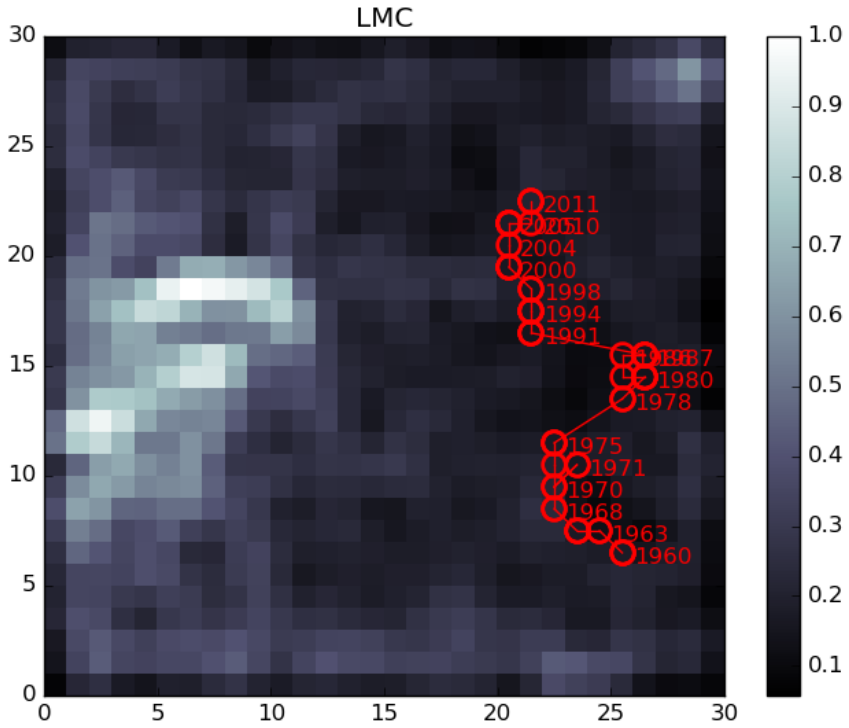


Figure 6.3: **Lower-middle income class time series**

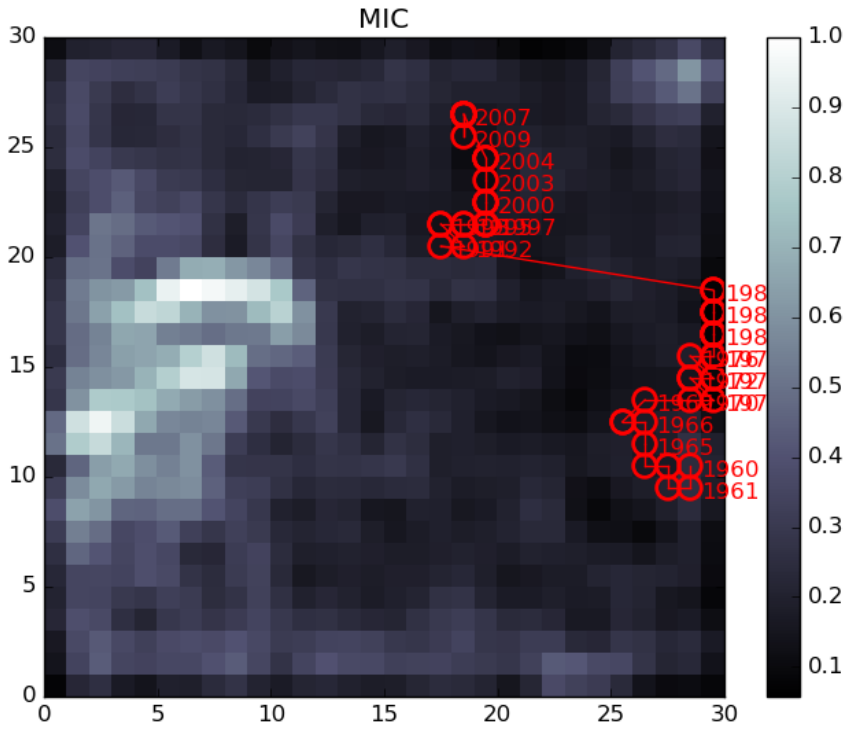


Figure 6.4: **Middle income class time series**

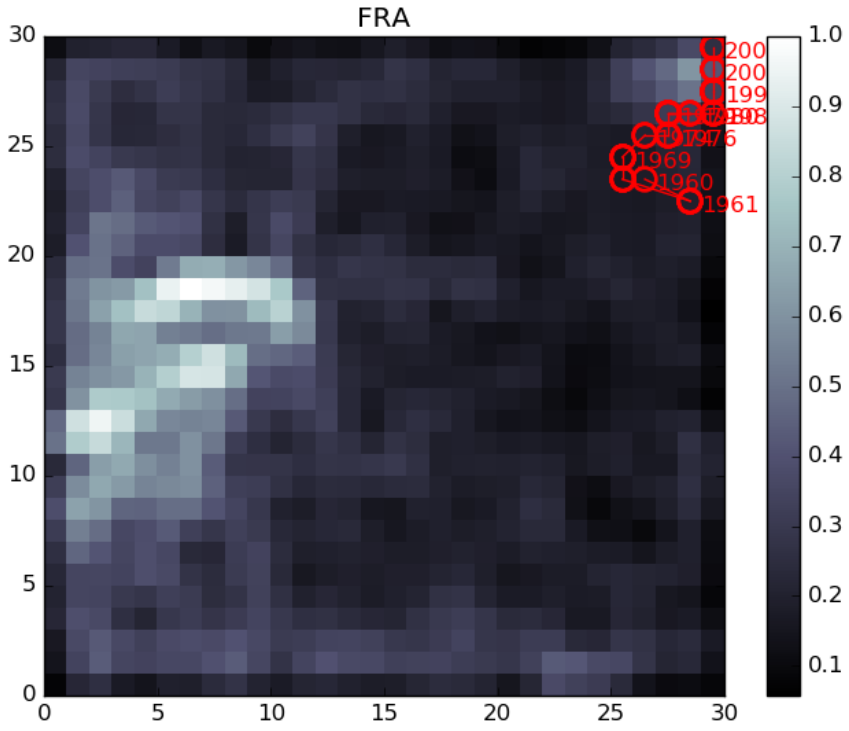


Figure 6.5: **France time series**

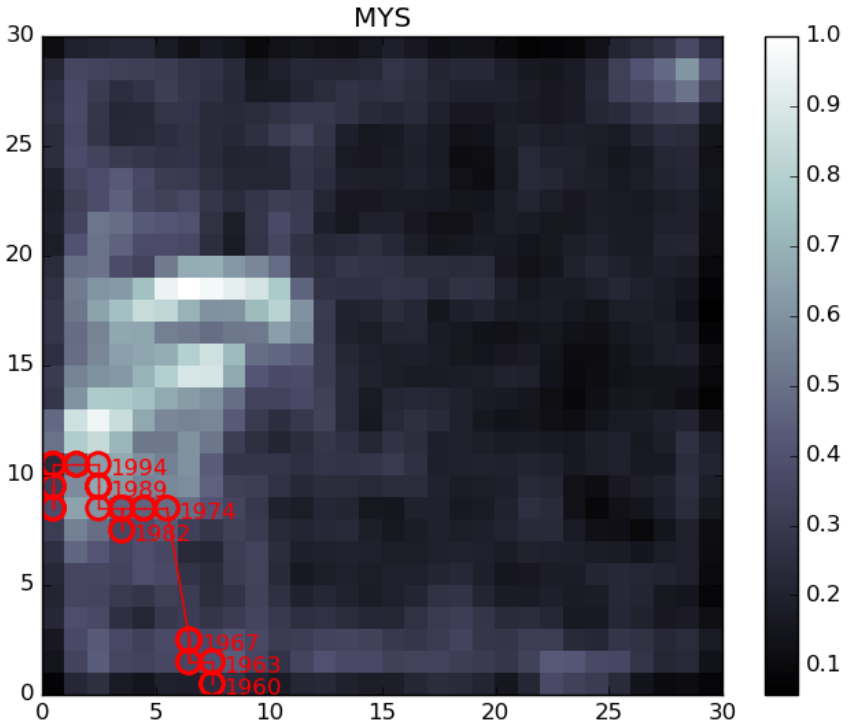


Figure 6.6: **Malaysia time series**

6.3 GDP Prognosis

One of the common goals in time-series analysis is to predict one of the variables into the future. Our data describes the macroeconomic development of countries, and arguably the most decisive indicator for determine the overall development of a country are the GDP and GDP per capita indicators. In this section we define such a prediction task and explore how different deep neural networks perform.

6.3.1 Experiment setup

First in order to give us an idea about what would be a reasonable time-lag into the past we have done a simple autocorrelation analysis. Autocorrelation is the correlation of a signal with itself at different time lags. Given a time series with n observations $\{X_1, \ldots, X_n\}$, its estimated mean μ and variance σ^2, an estimate of the autocorrelation index $R(k)$ of a time lag k can be obtained as:

$$R(k) = \frac{1}{(n-k)\sigma^2} \sum_{t=1}^{n-k} (X_t - \mu)(X_{t+k} - \mu) \tag{6.2}$$

When performed on the autocorrelation of GDP, the resulting autocorrelation, see Figure 6.7, suggests that the importance of past data drops significantly after about 6 years.

The input data can be represented as a matrix $X^c_{i,j}$ for each country. The superscript c denotes the country, i index denoting the indicator and j denoting

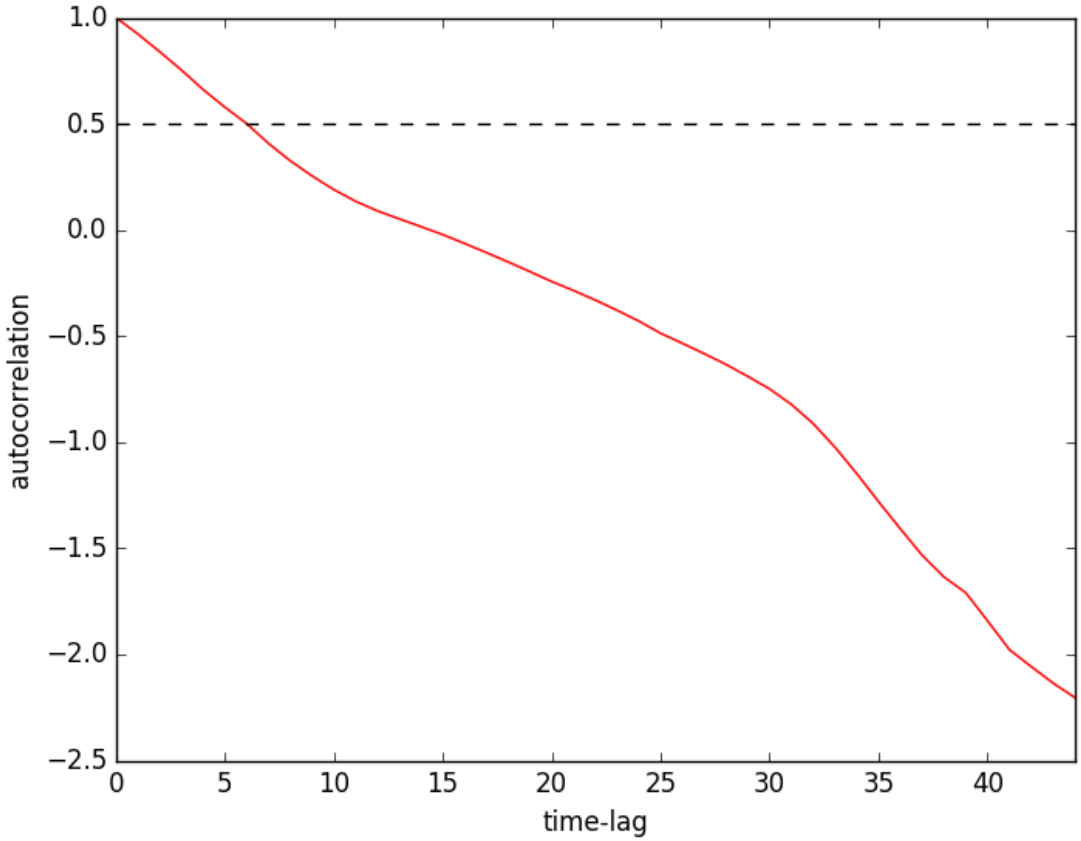


Figure 6.7: **Autocorrelation plot of GDP**

The plot denotes the correlation coefficient between different time-lags of GDP. The value at the x-coordinate k denotes the $R(k)$ coeffient as defined in Equation 6.2 averaged over the GDP time-series of all countries in the dataset. It represents the correlation between GDP values at time t and time $t + k$.

the year of the datapoint. The number of indicators is denoted by I. To create the set of samples for the prediction task, each country matrix has been split by columns into into 6-year sliding windows for each sample. This means for each country a set of samples has been created according to Equation 6.3 with the window size d set to 6.

$$S^c(t,d) \in \mathbb{R}^{I \times d} : S^c(t,d)_{i,j} = X^c_{i,t+j-d} \tag{6.3}$$

To formally define the prediction task the only thing left is to select the target variable. Our goal here is to predicting GDP, which is present in our data in two indicators - GDP Per Capita (GDPPC) and Absolute GDP (AGDP). We have selected AGDP as the target variable, keeping the GDPPC as an indicator.

Formally the prediction task then is to find a function $f : \mathbb{R}^{I \times d} \to \mathbb{R}^2$. The input of the function is a matrix of $I = 16$ vectors, each vector a time series of an indicator. The target of the function are two numbers: the first the AGDP and the second a differential variable. We denote the AGDP indicator index as a and then get the definition of the target as:

$$f(S^c(t,d)) = (X^c_{a,t+d+1}, X^c_{a,t+d+1} - X^c_{a,t+d}) \tag{6.4}$$

For example for a matrix of indicators from the year 1970 to 1975, the first target variable will be the AGDP in the year 1976. As the second target variable we included a difference indicator, in our example the second target variable would be the difference between the AGDP of 1976 and 1975. Including the second differential variable follows the learning with hints principle which can improve the prediction strength.

In the following sections we will explore how this prediction task can be solved using different data mining approaches and compare their effectiveness. In each architecture we explore how different model parameters affect its performance. The model parameters include:

- **Number of hidden layers.** For data that is linearly separable there is no need for any hidden layers. However this is seldom the case for more complex data. For the majority of problems one hidden layer is considered sufficient. Furthermore, additional layers can lead to overfitting and much longer training times.

- **Number of neurons in hidden layers.** There is no clear rule as to how to select the number of neurons [60]. Generally an amount around the number of input or output neurons is considered a good amount. In this case the number of input neurons is 112 (16 indicators times 7 - width of window).

- **Activation function.** Each hidden neuron has an activation function as described in Chapter 2. The most common activation functions are discussed in 2.3.

- **Regularization method.** The most common problem when training neural networks is the problem of overfitting. There are many different ways to prevent overfitting. In these experiments we consider two different types. One introduces noise into the data during each training epoch and the other is Dropout. These techniques are discussed in more detail in Section 2.4

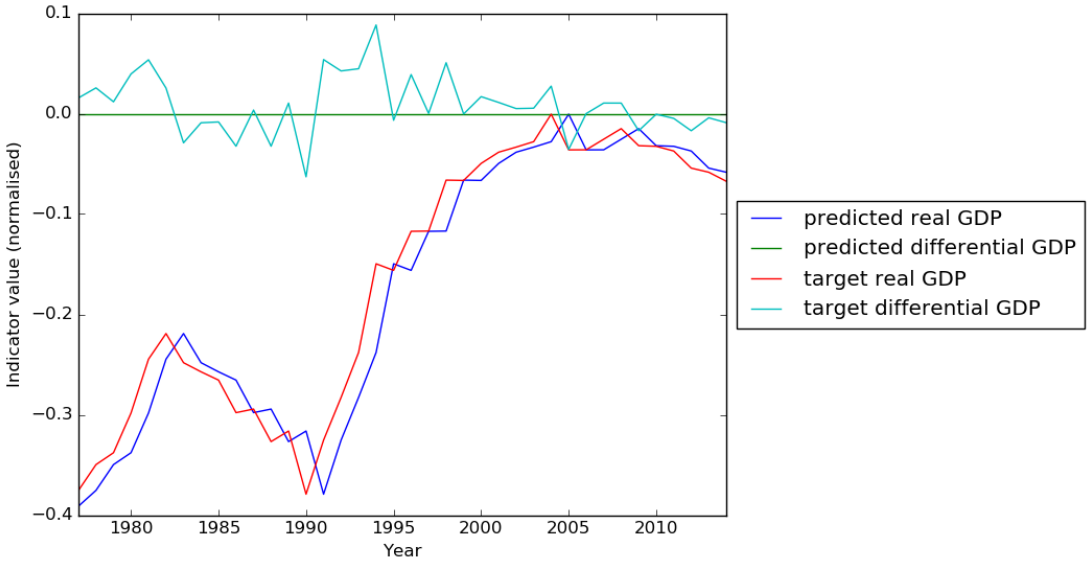


Figure 6.8: **Example of how the naive model predicts the GDP of Czech Republic**

The exact values considered for each parameter will be discussed separately for each architecture tested. In each experiment, the data will be randomly separated into five disjunct sets by countries. Afterwards the training/testing set split will follow the crossvalidation scheme, where one set is left out being the testing set. The resulting performances will be presented on the disjunct training, validation and testing set.

The naive model

As a benchmark model for the prediction task we used a naive model. The naive model simply predicts a zero-difference in GDP. The model is defined in Equation 6.5. The model achieves a MSE of 4,17E-3 on our dataset. An example of the prediction of the naive model is presented in Figure 6.8.

$$f(S^c(t,d)) := (X^c_{a,t+d}, 0) \tag{6.5}$$

The naive model is often used as a benchmark to compare different methods for GDP prediction. To gauge how different models perform compared to the naive model we present a few examples of related works to the task of GDP prediction.

The first example presents a comparison of different classic models used for GPD prediction of Japan in [61]. In this work a naive model is used to compare classic time series modeling techniques - Vector Autoregression and the VECM model. The RMSE achieved by these models is shown in Table 6.2. In this work the classic models achieved a RMSE about three-times better than that of the naive model. Another related work predicts Canadian GDP in [62]. The work compares a few classic time-series models, regression models and neural network models. The results presented show that neural network models can achieve a MSE about half the MSE of the naive model when used for GDP prediction. The exact values are presented in Table 6.3.

model	RMSE	relative
naive	1,784	100%
VAR	0,651	36,5%
VECM	0,645	36,2%

Table 6.2: **Classic time-series models compared to the naive model**

model	MSE	relative
naive	5,26	100%
AR	5,53	105,1%
regressor	3,28	62,4%
neural net	2,51	47,7%

Table 6.3: **Regression and neural network models compared to the naive model**

6.3.2 MLP networks for GDP prediction

The first neural network-based approach we will consider for the prediction task at hand is using multilayer perceptron networks (MLP) which have been described in Chapter 2. In the experiment we have explored many different network architectures to determine which model is best suited to the task at hand. The number of input and output neurons are fixed by the prediction task. The variable parameters explored are the following:

- **Number of hidden layers.** In these experiments we consider networks with one, two and three hidden layers.
- **Number of neurons in hidden layers.** In our tests we will consider networks with 100, 200 and 300 neurons in each hidden layer.
- **Activation function.** In our experiments we tested the classic sigmoid activation function and a leaky ReLU function.
- **Regularization method.** In these experiments we consider two different types. One introduces noise into the data during each training epoch and the other is Dropout. These techniques are discussed in more detail in Section 2.4

With the model parameters described above we have executed experiments while measuring their performance. Each model has been tested using 50 random initialisations in order to get a statistically significant performance average.

In the following sections we examine how different model parameters affect the model prediction strenght. The prediction task is as described in 6.3. The loss function used during training is mean squared error. The optimizer used is a momentum based back-propagation algorithm. For more details on the implementation see Appendix B.

First, we present the raw experiment results and afterwards we analyse them, demonstrate the best performing ones and formulate how we should select the model architecture and asses this architectures applicability to this task.

Results

First we present the results for models that are using the tanh activation function. The results are listed in Table 6.4. For a visual illustration of the results examine the boxplot in Figure 6.9. Next we present the results for models that are using the Leaky ReLU activation function. The results are listed in Table 6.5. Again for a better illustration of the results examine the boxplot in Figure 6.10.

model id	epochs	tr. loss	std	val. loss	std	test loss	std	best	cpu
50U-1L-D	84	3,94E-3	2,36E-4	3,56E-3	4,50E-4	3,70E-3	9,52E-4	1,85E-3	0,1
50U-1L-N	85,5	2,98E-3	2,16E-4	4,23E-3	1,26E-3	5,99E-3	2,71E-3	3,09E-3	0,1
50U-2L-D	93,5	4,61E-3	3,44E-4	3,77E-3	8,34E-4	3,94E-3	1,29E-3	1,80E-3	0,1
50U-2L-N	82	2,20E-3	2,95E-4	4,36E-3	1,26E-3	5,44E-3	1,63E-3	2,80E-3	0,1
50U-3L-D	97	5,18E-3	3,84E-4	3,92E-3	7,44E-4	4,02E-3	1,24E-3	1,89E-3	0,1
50U-3L-N	79,5	1,79E-3	2,19E-4	3,89E-3	5,93E-4	4,94E-3	1,30E-3	2,43E-3	0,1
100U-1L-D	86,5	3,82E-3	2,74E-4	3,58E-3	6,33E-4	3,56E-3	9,85E-4	1,98E-3	0,1
100U-1L-N	87	3,71E-3	2,48E-4	5,20E-3	2,09E-3	6,47E-3	3,03E-3	3,22E-3	0,1
100U-2L-D	91	4,51E-3	3,58E-4	4,01E-3	9,04E-4	4,23E-3	1,47E-3	1,73E-3	0,1
100U-2L-N	77,5	2,06E-3	1,73E-4	4,06E-3	1,22E-3	4,86E-3	1,30E-3	2,53E-3	0,1
100U-3L-D	91,5	5,08E-3	3,21E-4	3,92E-3	8,32E-4	4,14E-3	1,33E-3	2,08E-3	0,1
100U-3L-N	81	1,76E-3	2,68E-4	4,11E-3	7,89E-4	5,09E-3	1,26E-3	3,35E-3	0,1
200U-1L-D	83,5	4,01E-3	2,31E-4	3,97E-3	8,15E-4	4,12E-3	1,04E-3	2,15E-3	0,1
200U-1L-N	103,5	5,04E-3	4,56E-4	7,43E-3	3,49E-3	8,27E-3	4,04E-3	3,11E-3	0,1
200U-2L-D	93	4,76E-3	3,28E-4	4,53E-3	1,78E-3	4,51E-3	1,72E-3	2,25E-3	0,1
200U-2L-N	79	2,17E-3	1,97E-4	4,06E-3	1,09E-3	4,81E-3	1,37E-3	2,56E-3	0,1
200U-3L-D	90	5,59E-3	3,86E-4	5,24E-3	1,58E-3	5,15E-3	1,70E-3	2,53E-3	0,1
200U-3L-N	80,5	1,82E-3	2,96E-4	4,06E-3	6,50E-4	5,14E-3	1,24E-3	3,20E-3	0,1
300U-1L-D	84	4,16E-3	2,59E-4	4,19E-3	1,16E-3	4,33E-3	1,42E-3	2,16E-3	0,1
300U-1L-N	100	5,78E-3	5,44E-4	7,33E-3	4,60E-3	8,16E-3	4,59E-3	2,58E-3	0,1
300U-2L-D	82	5,25E-3	2,98E-4	4,82E-3	1,52E-3	5,05E-3	1,89E-3	2,42E-3	0,1
300U-2L-N	84,5	2,22E-3	2,52E-4	4,04E-3	6,19E-4	4,70E-3	1,04E-3	3,10E-3	0,1
300U-3L-D	89	6,39E-3	3,63E-4	6,43E-3	2,29E-3	6,73E-3	2,43E-3	2,97E-3	0,1
300U-3L-N	84,5	1,84E-3	3,13E-4	4,36E-3	1,47E-3	5,29E-3	1,66E-3	2,91E-3	0,1

Table 6.4: **AGDP Centered Sigmoid performance**

The table contains the model identifier which is comprised of the number of units in each hidden layer, the number of layers and the regularization type - D for dropout and N for noise. In the following columns the table contains the average amount of epochs needed before the stopping condition has been hit, the average training loss, the average validation loss and the average test loss along with their standard deviations. The last two columns contain the best test loss and the average cpu time it took to train the model in hours. The two best average validation, test losses and the two overall best test losses have been underlined.

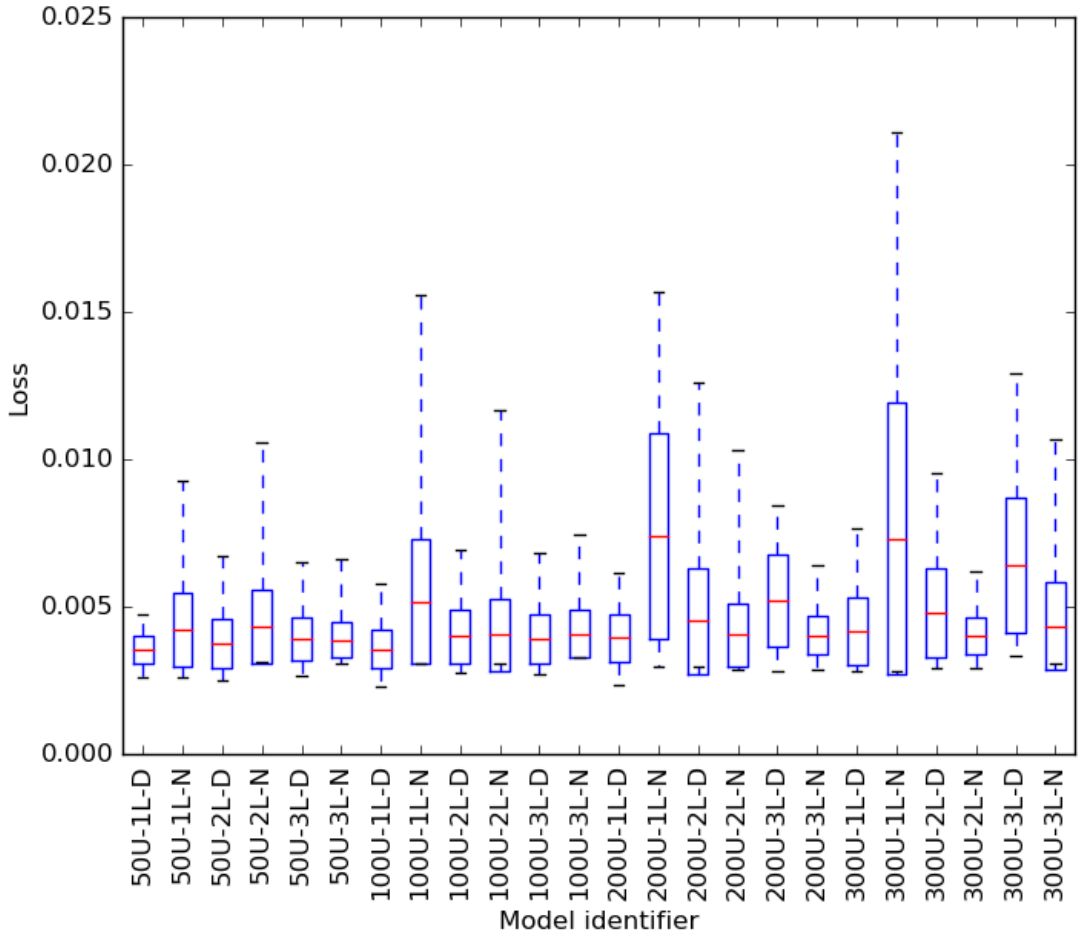


Figure 6.9: **Boxplot of the AGDP prediction model performance with the Tanh function**

The plot contains the average validation loss denoted by the red line, the size of the box represents the standard deviation and the whiskers in the box plot are the minimum and maximum validation loss.

model id	epochs	tr. loss	std	val. loss	std	test loss	std	best	cpu
50U-1L-D	98,5	3,80E-3	2,62E-4	3,37E-3	5,54E-4	3,45E-3	7,42E-4	1,99E-3	0,1
50U-1L-N	87,5	2,70E-3	1,54E-4	3,53E-3	5,72E-4	3,60E-3	6,89E-4	2,32E-3	0,1
50U-2L-D	107	4,17E-3	3,69E-4	3,41E-3	6,97E-4	3,42E-3	9,47E-4	1,83E-3	0,1
50U-2L-N	70,5	1,81E-3	1,94E-4	3,86E-3	4,55E-4	4,55E-3	9,34E-4	2,60E-3	0,1
50U-3L-D	113	4,51E-3	3,24E-4	3,58E-3	5,28E-4	3,56E-3	7,57E-4	2,07E-3	0,1
50U-3L-N	66,5	1,56E-3	2,13E-4	3,93E-3	5,43E-4	4,72E-3	1,03E-3	2,99E-3	0,1
100U-1L-D	96	3,57E-3	2,30E-4	3,66E-3	6,81E-4	3,60E-3	9,97E-4	1,82E-3	0,1
100U-1L-N	89,5	2,72E-3	2,35E-4	3,40E-3	7,39E-4	3,51E-3	8,19E-4	2,15E-3	0,1
100U-2L-D	102	3,95E-3	2,69E-4	3,38E-3	6,02E-4	3,44E-3	9,06E-4	1,74E-3	0,1
100U-2L-N	69	1,68E-3	2,14E-4	3,96E-3	4,90E-4	4,52E-3	1,04E-3	2,88E-3	0,1
100U-3L-D	108,5	4,35E-3	2,83E-4	3,74E-3	7,55E-4	3,77E-3	9,29E-4	1,89E-3	0,1
100U-3L-N	68,5	1,40E-3	2,02E-4	4,01E-3	6,94E-4	4,81E-3	1,10E-3	2,54E-3	0,1
200U-1L-D	89	3,73E-3	1,92E-4	3,72E-3	1,13E-3	3,82E-3	1,17E-3	2,37E-3	0,1
200U-1L-N	89,5	2,76E-3	2,09E-4	3,44E-3	6,96E-4	3,50E-3	9,06E-4	1,99E-3	0,1
200U-2L-D	90	4,10E-3	2,54E-4	3,75E-3	8,28E-4	3,78E-3	8,87E-4	2,50E-3	0,1
200U-2L-N	71	1,80E-3	2,32E-4	4,14E-3	1,20E-3	4,76E-3	1,49E-3	2,92E-3	0,1
200U-3L-D	96	4,61E-3	2,11E-4	4,08E-3	1,27E-3	4,12E-3	1,22E-3	2,55E-3	0,1
200U-3L-N	73,5	1,47E-3	2,58E-4	4,04E-3	6,68E-4	4,60E-3	1,20E-3	2,70E-3	0,1
300U-1L-D	93	3,78E-3	2,68E-4	3,94E-3	8,73E-4	4,00E-3	1,21E-3	1,64E-3	0,1
300U-1L-N	89	2,88E-3	2,05E-4	3,63E-3	1,04E-3	3,84E-3	1,25E-3	2,29E-3	0,1
300U-2L-D	89	4,31E-3	2,51E-4	3,95E-3	9,41E-4	3,99E-3	1,05E-3	1,86E-3	0,1
300U-2L-N	70	1,96E-3	1,95E-4	4,14E-3	8,56E-4	4,68E-3	1,17E-3	2,74E-3	0,1
300U-3L-D	81	5,07E-3	2,67E-4	4,56E-3	1,09E-3	4,66E-3	1,24E-3	2,68E-3	0,1
300U-3L-N	72	1,62E-3	2,39E-4	4,18E-3	1,04E-3	4,77E-3	1,57E-3	1,79E-3	0,1

Table 6.5: **AGDP LeReLU performance**

The table contains the model identifier which is comprised of the number of units in each hidden layer, the number of layers and the regularization type - D for dropout and N for noise. In the following columns the table contains the average amount of epochs needed before the stopping condition has been hit, the average training loss, the average validation loss and the average test loss along with their standard deviations. The last two columns contain the best test loss and the average cpu time it took to train the model in hours. The two best average validation, test losses and the two overall best test losses have been underlined.

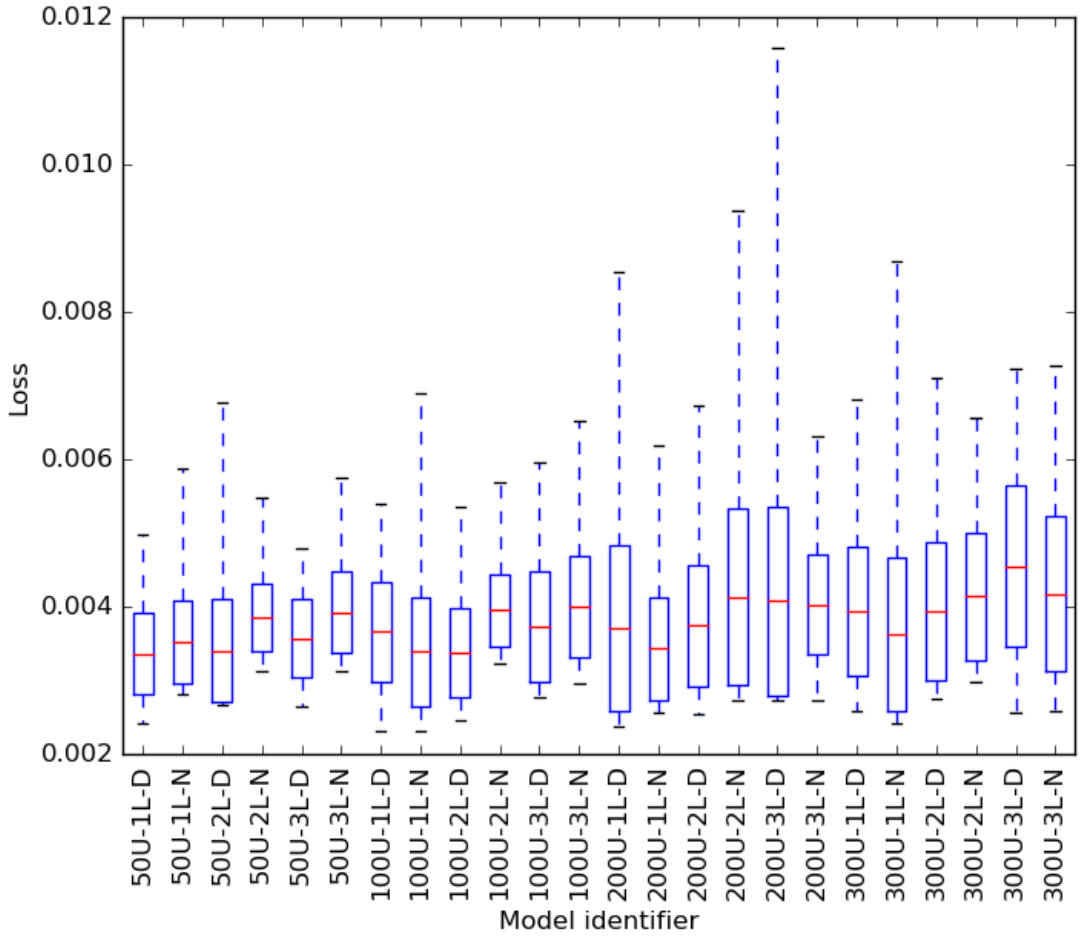


Figure 6.10: **Boxplot of the AGDP prediction model performance with the LeReLU function**

The plot contains the average validation loss denoted by the red line, the size of the box represents the standard deviation and the whiskers in the box plot are the minimum and maximum validation loss.

Analysis and model demonstration

To analyse the results, we take a look at how the models tend to perform depending on each separate parameter tested. For a list of average validation losses over a certain parameter, review Table 6.6.

Overall the performance between different model architectures does not differ that much. First we discuss the model architecture - the layer count and unit count - and its influence on performance. The best (underlined) models in the performance Tables 6.5 and 6.4 are those that in all cases have either one or two layers. The aggregates in Table 6.6 confirm that that 2 layer models perform slightly better than others with statistical significance. When it comes to the number of neurons in each hidden layer the best performing models are in general either 50 unit or 100 unit models, with 50 neuron models having a slight edge according to the t-test. This goes against the rule of thumb we mentioned in the experiment setup, but since the 100 unit networks are farily close (even the p-value is not that low) it gave us a decent starting point estimate.

From the performance Tables 6.5 and 6.4 we can see that a big majority of the best performing models are models using Dropout. This is further confirmed by the fact that the dropout models have an average validation loss of 4,04E-3 compared to the average noise validation loss of 4,31E-3. Finally when comparing the Leaky ReLU models to the tanh models in the performance Tables 6.5 and 6.4 the best performances of the LeReLU models are significantly better than those of the tanh models, and this holds for the averages aswell - with LeReLU achieving a 3,81E-3 average and tanh a significantly higher 4,54E-3 average validation loss.

Next we take a look at the training process. In general the training process was very similar to that of model `100U-2L-D-AGDP-lerelu`, which is shown in Figure 6.11. The validation and test loss seem to be still decreasing even after 70 epochs of training. However as mentioned in the section 6.1, it is very likely that the data might be strongly correlated. Even though we selected whole countries to be in the test set separating the database as well as we could into two independent sets, the test and validation loss stay strongly correlated during training.

We now take a few prediction models and examine how they predict GDP of a few countries to see if they predict in any interesting ways. We compare the best performing Dropout model with the parameters - 100U-2L-D-lerelu - and the best performing noisy model with parameters - 300U-3L-N-lerelu. The Dropout model achieved a MSE of $1{,}74E-3$ on the test set while the Noise model achieved a $1.79E-3$ MSE on the test set.

In Figure 6.12 we first take a look at how the two models perform on the samples inside the training set. In the case of better performing dropout model, the model copies the naive model fairly closely. Notice especially that the model does not even attempt to train the differential GDP. The 300U-3L-N-lerelu on the other hand seems to have trained the differential GDP at least a little bit. This effect could be explained by the fact that the 100U-2L-D-lerelu has a lower capacity and cannot learn the differential effectively. Another reason could be that the model simply got stuck in a local minimum of the gradient, since the naive model is already a fairly good model.

The next examples are presented in Figures 6.13 and 6.14 as the predictions of out-sample countries - Nepal and Cuba. In both cases the predictions follow the observations made in the previous paragraph. The model trained can be mostly

grouping	avg val. loss	std	runs	best test loss	P-value
1-layer models	4,26E-3	1,30E-3	800	1,64E-3	$< 0{,}0001$
2-layer models	4,02E-3	9,56E-4	800	1,73E-3	-
3-layer models	4,25E-3	9,71E-4	800	1,79E-3	$< 0{,}0001$
50-unit models	3,79E-3	7,07E-4	600	1,80E-3	-
100-unit models	3,92E-3	8,69E-4	600	1,73E-3	0.0046
200-unit models	4,37E-3	1,27E-3	600	1,99E-3	$< 0{,}0001$
300-unit models	4,63E-3	1,46E-3	600	1,64E-3	$< 0{,}0001$
dropout models	4,04E-3	9,78E-4	1200	1,64E-3	-
noise models	4,31E-3	1,17E-3	1200	1,79E-3	$< 0{,}0001$
tanh models	4,54E-3	1,36E-3	1200	1,73E-3	-
lerelu models	3,81E-3	7,89E-4	1200	1,64E-3	$< 0{,}0001$

Table 6.6: **MLP model performance aggregates**

The table contains the average validation loss over all models with a certain parameter fixed, and the best test loss on the same subset of models. The first column is the fixed parameter, with the other columns being the aggregates. The last column lists pairwise t-test P-values with the other pair being the best performing model aggregate within the specific parameter (denoted by horizontal lines).

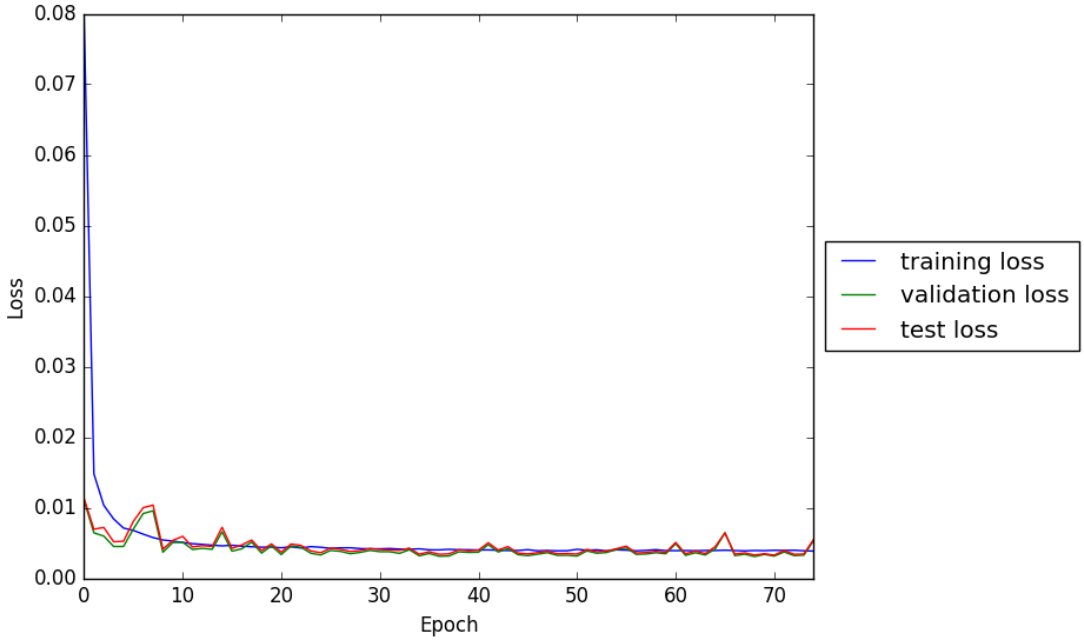


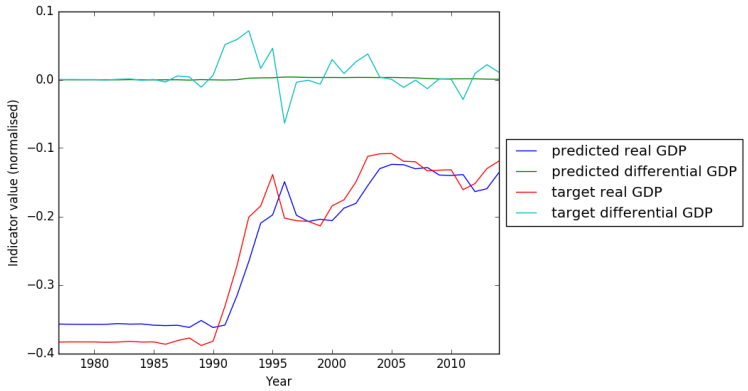
Figure 6.11: **A chart of the loss values over the training process of a MLP network**

summarized as the naive model. In the case of prediction of Nepals GDP using the 100U-2L-D-lerelu model, the prediction looks better than that of the naive model. Either the model has trained on sample countries similar to Nepal, or the model learnt some underlying dependency.

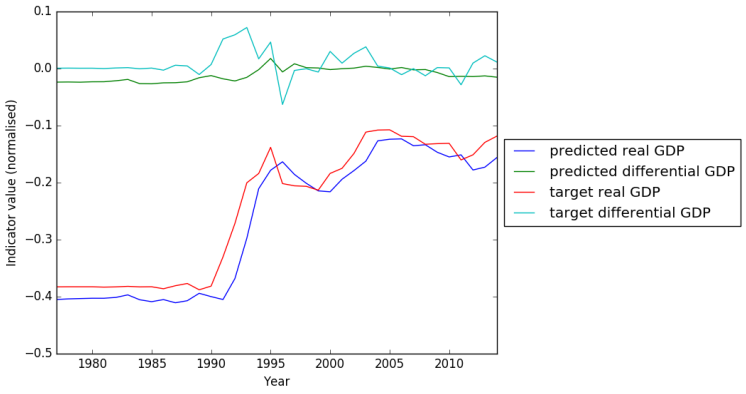
In many countries, the years following the fall of the Soviet Union in 1991 caused large turbulences in their data. As mentioned in Section 6.1 during the exploratory analysis, especially countries of the eastern bloc can contain noisy or incorrect data around these years. We examined how the models perform during these years to see how reliable are the models during these years. You can see histograms of the error rates based on the year of the prediction in Figure 6.15. The histograms show that the models underperform during the period from 1990-1995.

We can see that the dropout model mostly follows the naive model defined in Section 6.3.1, with a possible few improvements. The noisy model does correctly predict a large swing in GDP around the year 1991. As mentioned in Section 6.1 during the exploratory analysis, especially countries of the eastern bloc can contain noisy or incorrect data around these years. We can further examine this trend by looking at the histogram of loss rates by year in Figure 6.15.The histograms show that the models underperform during these years. That means that the network has not overtrained itself to detect these changes even though it correctly predicts it in the case of Lithuania.

To summarise the analysis, the best performing models use 50 neurons in 2 layers with the Leaky ReLU function. For regularization a combination of cross-validation and Dropout saw best performance. Overall the MLP models achieve a better performance than that of the naive model 4,17E-3 MSE versus the best losses of 1,64E-3 which is 39,3% of the naive model loss. This result is consistent with the performance of neural networks reported in related works discussed in Section 6.3.1. We further examined the models and in most cases it seemed like the models act very similarily to the naive model.

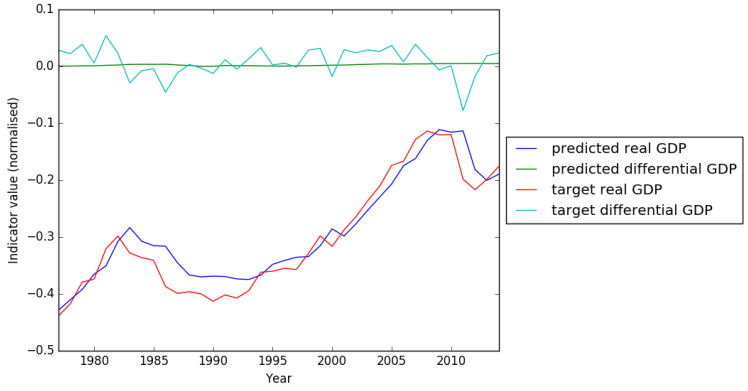


(a) 100U-2L-D-lerelu

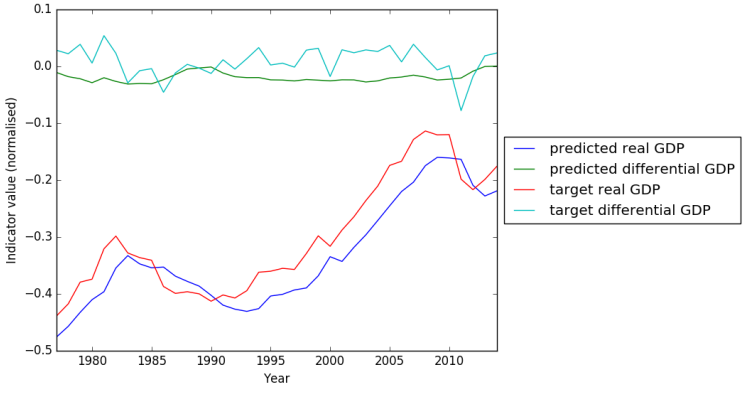


(b) 300U-3L-N-lerelu

Figure 6.12: **The GDP prediction of in-sample Poland**

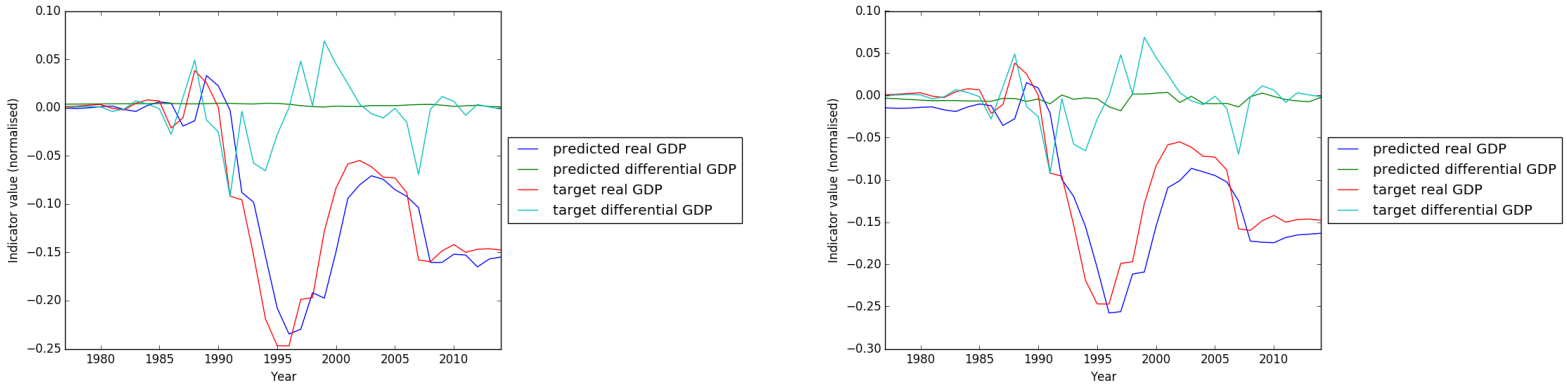


(a) 100U-2L-D-lerelu



(b) 200U-3L-N-tanh

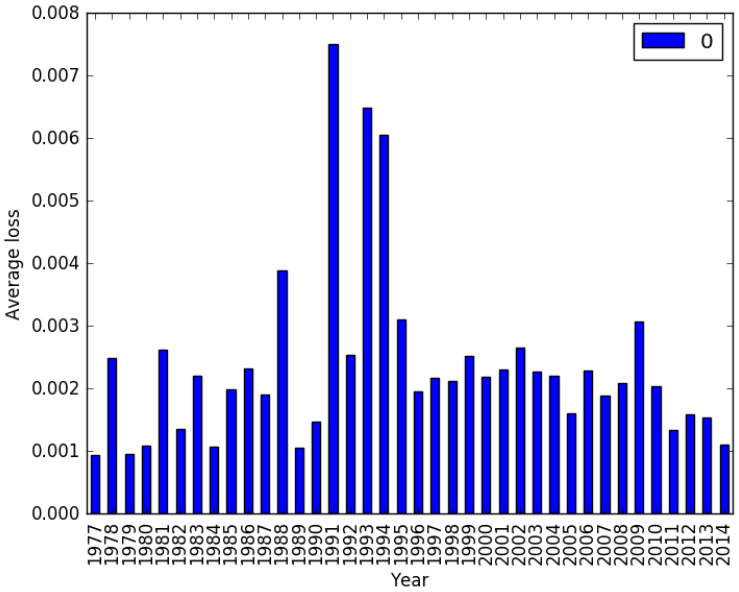
Figure 6.13: **The GDP prediction of out-sample Nepal**



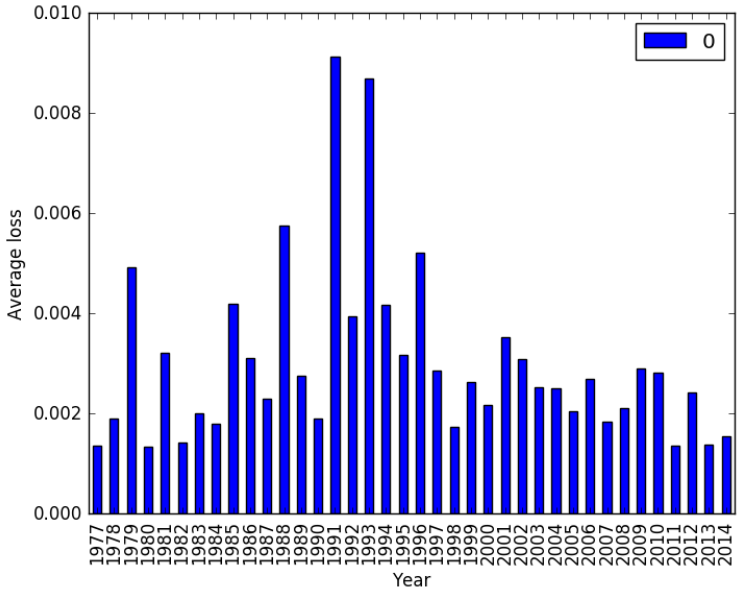
(a) 100U-2L-D-lerelu

(b) 300U-3L-N-lerelu

Figure 6.14: **The GDP prediction of out-sample Cuba**



(a) 100U-2L-D-lerelu



(b) 300U-3L-N-lerelu

Figure 6.15: **The error rates over the whole dataset of two different models.**

6.3.3 LSTM networks for GDP prediction

The second neural network architecture we tested for the prediction task defined in Section 6.3 was recurrent neural networks using the Long Short Term Memory architecture described in Chapter 5.1. Similar to the MLP experiment we have considered the following model parameters:

- **Number of hidden layers.** The network is setup in layers of LSTM cell blocks in a similar manner to multilayer perceptron networks, except in this case, each unit is an LSTM cell. We tested networks with one, two and three layers. The output layer was always a layer of neurons densely connected to the last LSTM layer.
- **Number of neurons in hidden layers.** In our tests we will consider networks with 100 and 200 LSTM cells in each hidden layer. Since the training time is greatly increased compared to MLP and CNN networks, networks with 300 hidden units were not tested. Furthermore networks with 200 hidden units were not tested with three layers.
- **Activation function.** The activation functions of LSTM cells used were a sigmoid for the inner gate nodes and tanh for the state gates.
- **Regularization method.** Both Dropout and noising in each epoch were tested for regularization, see Section 2.4.

Experiment results

In this section we examine the results for models predicting the Absolute GDP using LSTM networks. The results are listed in Table 6.7 and again for a better illustration of the results examine the boxplots in Figure 6.16.

model id	epochs	tr. loss	std	val. loss	std	test loss	std	best	cpu
100U-D-1L	63	2,57E-3	2,56E-4	3,33E-3	3,73E-4	3,53E-3	8,35E-4	1,96E-3	1,4
100U-D-2L	85	2,34E-3	5,89E-4	3,09E-3	9,83E-4	3,50E-3	1,42E-3	1,19E-4	2,9
100U-D-3L	112,5	2,06E-3	8,52E-4	2,58E-3	1,47E-3	2,92E-3	2,02E-3	1,08E-4	3,9
100U-N-1L	95	2,19E-3	7,63E-4	2,71E-3	1,24E-3	2,81E-3	1,41E-3	1,81E-4	1,2
100U-N-2L	69	2,56E-3	2,72E-4	3,46E-3	3,79E-4	3,93E-3	1,01E-3	2,01E-3	3,1
100U-N-3L	74,5	2,62E-3	2,04E-4	3,35E-3	3,83E-4	3,94E-3	9,81E-4	2,44E-3	5,2
200U-D-1L	53,5	2,15E-3	2,81E-4	3,46E-3	4,23E-4	3,81E-3	9,17E-4	2,22E-3	2,4
200U-D-2L	62	2,18E-3	3,33E-4	3,54E-3	4,62E-4	4,43E-3	1,20E-3	2,46E-3	6
200U-N-1L	50,5	2,21E-3	1,99E-4	3,41E-3	3,99E-4	3,72E-3	9,45E-4	2,26E-3	2,3
200U-N-2L	60,3	2,25E-3	3,37E-4	3,47E-3	3,86E-4	4,09E-3	1,28E-3	2,34E-3	5,7

Table 6.7: **AGDP LSTM performance**

The table contains the model identifier which is comprised of the number of units in each hidden layer, the regularization type - D for dropout and N for noise and the number of layers. In the following columns the table contains the average amount of epochs needed before the stopping condition has been hit, the average training loss, the average validation loss and the average test loss along with their standard deviations. The last two columns contain the best test loss and the average cpu time it took to train the model in hours. The two best average validation, test losses and the two overall best test losses have been underlined.

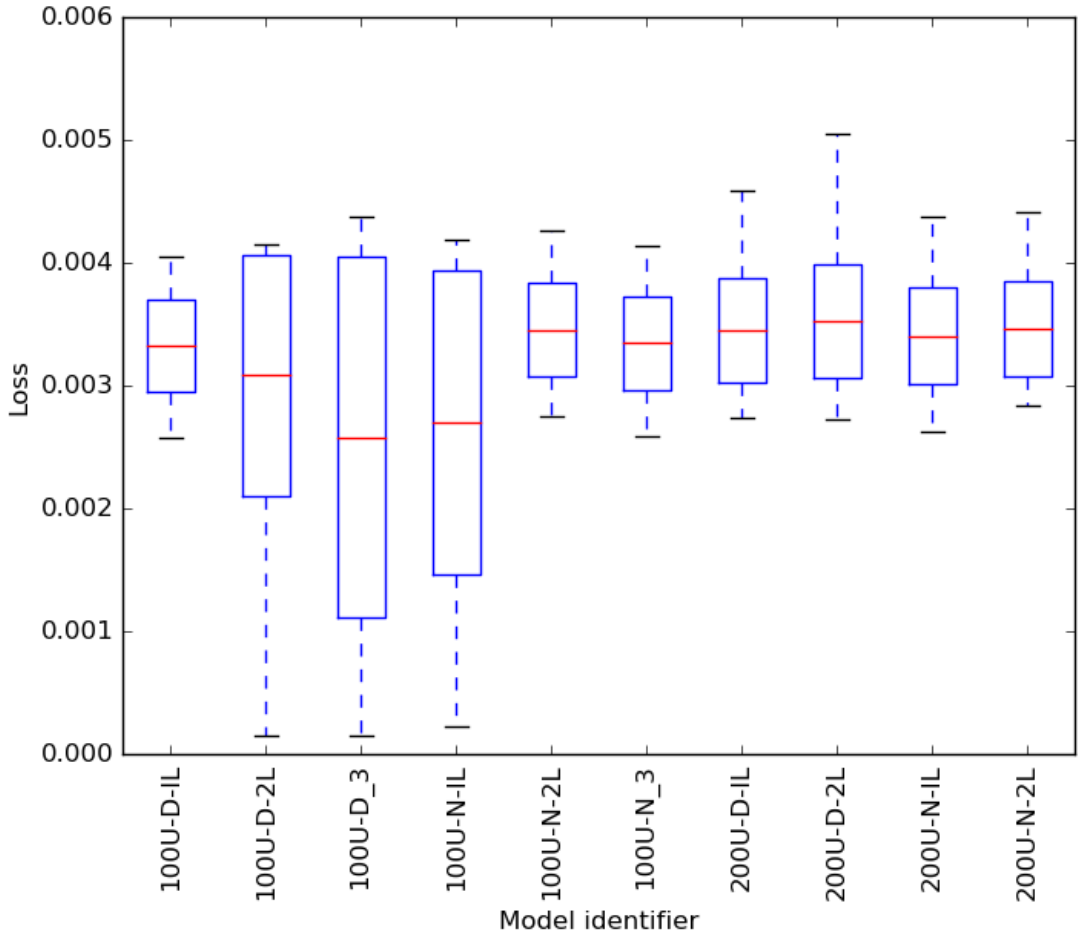


Figure 6.16: **Boxplot of the AGDP prediction model performance with the LSTM architectures**

The plot contains the average validation loss denoted by the red line, the size of the box represents the standard deviation and the whiskers in the box plot are the minimum and maximum validation loss.

grouping	avg val. loss	std	runs	best test loss	P-value
1-layer models	3,23E-3	6,09E-4	200	1,81E-4	0.0039
2-layer models	3,39E-3	5,52E-4	200	1,19E-4	< 0.0001
3-layer models	2,97E-3	9,26E-4	100	1,08E-4	-
100-unit models	3,09E-3	8,04E-4	300	1,08E-4	-
200-unit models	3,47E-3	4,18E-4	200	2,22E-3	< 0.0001
dropout models	3,20E-3	7,42E-4	250	1,08E-4	-
noise models	3,28E-3	5,57E-4	250	1,81E-4	0.1734

Table 6.8: **LSTM model performance aggregates**

The table contains the average validation loss over all models with a certain parameter fixed, and the best test loss on the same subset of models. The first column is the fixed parameter, with the other columns being the aggregates. The last column lists pairwise t-test P-values with the other pair being the best performing model aggregate within the specific parameter (denoted by horizontal lines).

Analysis and model demonstration

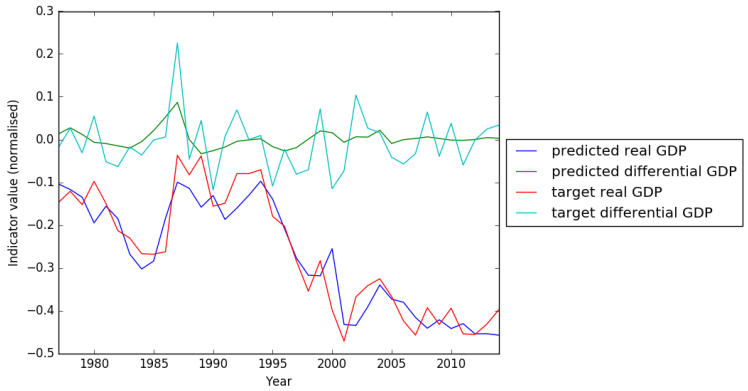
First we analyse how different parameters affect the performance in Table 6.8. The main difference the results show us is that 200 neuron models perform significantly worse than 100 neuron models. The difference between 3-layer and 1-layer models is not that significant, in some cases 1-layer models perform well, and in other cases the 3-layer models. Suprisingly the 2-layer models in the middle have been shown to underperform. Finally the regularization type used does not seem to be too significant, with dropout having a slight edge.

In this section we take the best LSTM prediction models for absolute GDP and examine examples of a few hand picked cases. Comparatively the LSTM networks performed very well, especially on the high end initialisations. A closer examination of how one of the `100U-3L-D` networks performs in Figure 6.17 shows us a model that is in general very similar to the naive model. In the case of predicting Congo the prediction seems to be following the target very well. Nevertheless compared to the naive model the plots show that our models react on the differential GDP target, at least during turbulent periods.

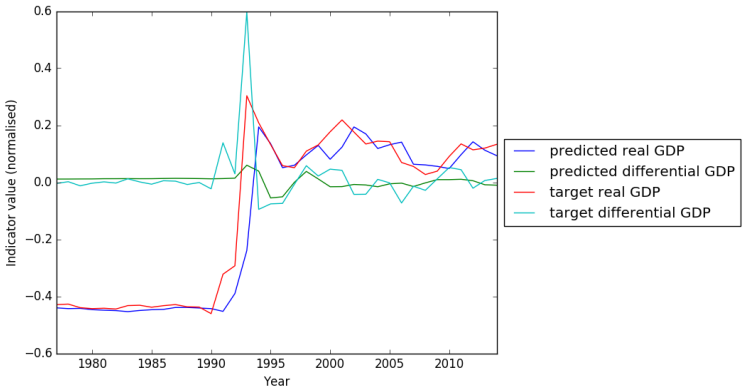
To further examine the workings of these models, we can compare the performance of the differential prediction to the absolute prediction. To do this we define a new variable as $diffGPD(t) = GDP(t-1) + differentialprediction(t)$. This variable shows us how close to the GDP would we be if we used only the differential target variable for prediction. We can see an example of how this variable predicts the GDP of Spain and Estonia in Figure 6.18. In a lot of cases the predicted real GDP variable can get very far from the target, as it is not directly tied to the previous time step. A possible improvement to better predict GDP could be to use the differential compound prediction along with the real GDP prediction (e.g. average), even though we would lose the flexibility of moving away from the naive model.

Finally we take a look at the training process of the model `100U-2L-D`, which is shown in Figure 6.19. The plot shows the validation and test loss stop improving shortly after the training start. Since the LSTM models were one of the most computationally taxing ones, it could be recommended to implement an earlier

stopping condition.

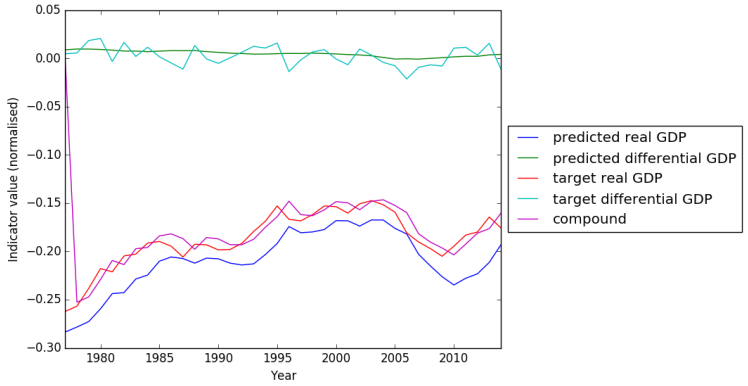


(a) Republic of Congo

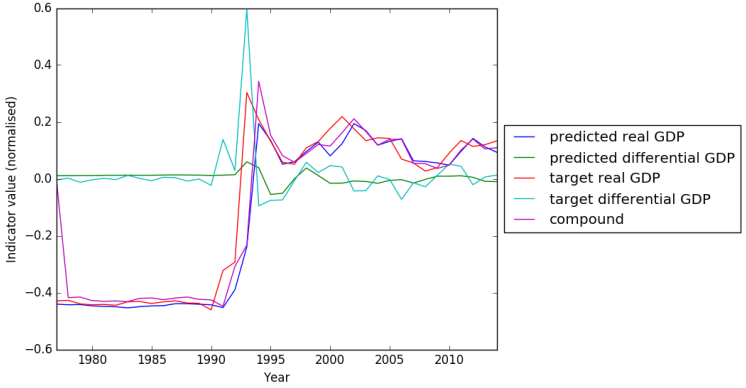


(b) Estonia

Figure 6.17: **The GDP prediction of a LSTM network**



(a) Spain



(b) Estonia

Figure 6.18: **A comparison of the absolute and differential target variable of a LSTM network**

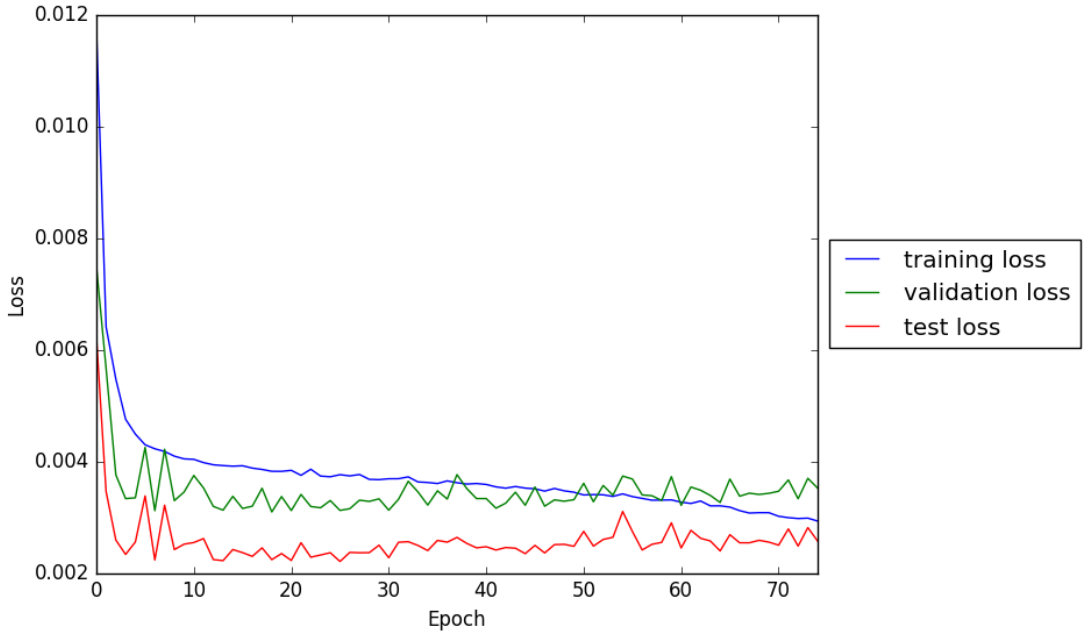


Figure 6.19: **A chart of the loss values over the training process of a LSTM network**

6.3.4 CNN networks for GDP prediction

The last neural network-based approach considered for the prediction task was ti use convolutional neural networks (CNN) which have been described in Chapter 4. In line with the previous experiments we have explored a few different network architectures to determine how different parameters affect the task at hand.

The CNNs used in our task are constructed using a stack of convolutional layers and a multilayer perceptron network on top. Even though the different indicators are strongly correlated, we have decided to train filters for each indicator separately. An illustration of such a CNN architecture is presented in Figure 6.20, where two convolutional layers were connected to each indicator. The variable parameters explored are then the following:

- **Number of hidden layers in the MLP.** In these experiments we consider networks with one or two layer MLPs connected to the output of the convolutional layers.
- **Number of neurons in MLP layers.** In our tests we will consider networks with 100 and 200 neurons in each MLP layer.
- **Activation functions.** The activations used in both the MLP layers and the convolutional layers were tanh.
- **Regularization method.** In these experiments we consider only the noise noise regularization, see Section 2.4.
- **Filter width.** Each convolutional layer trains multiple filters that convolve the input as defined in Chapter 4. The filter widths considered were 2, 4 and 6. In the case of the 6 wide filter the window width was expanded by 3 to allow some data to flow through the network.

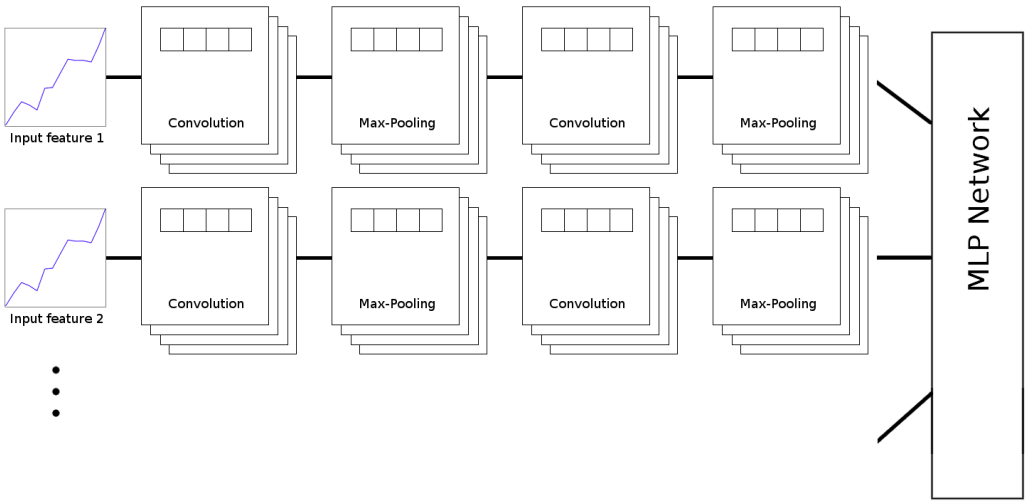


Figure 6.20: **A chart of the CNN architecture.**

The CNN trains filters for each indicator separately. The results of the small CNNs for each input feature are concatenated and passed into a MLP.

- **Filter count.** Each convolutional layer trains multiple filters that convolve the input, the number of filters considered in each convolutional layer was 2 and 4.

With the model parameters described above we have executed experiments while measuring their performance. Each model has been tested using 50 random initialisations in order to get a statistically significant performance average.

In the following sections we examine how different model parameters affect the model prediction strenght. The prediction task is as described in 6.3. The loss function used during training is mean squared error. The optimizer used is a momentum based back-propagation algorithm. For more details on the implementation see Appendix B.

First, we present the raw experiment results and afterwards we analyse them, demonstrate the best performing ones and formulate how we should select the model architecture and asses this architectures applicability to this task.

Model performance

The raw performance of the models using the CNN architectures described earlier are presented in Tigure 6.9. For a more visual examination of the results we present Figure 6.21 a boxplot of the validation losses of the different model parameter runs.

model id	epochs	tr. loss	std	val. loss	std	test loss	std	best	cpu
100U-1L-[[2-4]-[2-2]]	92,5	2,97E-3	2,40E-4	3,34E-3	4,06E-4	3,43E-3	8,02E-4	1,96E-3	0,2
100U-1L-[[4-4]-[4-2]]	83	2,80E-3	2,69E-4	3,36E-3	4,24E-4	3,40E-3	9,00E-4	1,90E-3	0,4
100U-1L-[[6-4]-[4-2]]	71	2,80E-3	1,91E-4	3,27E-3	3,30E-4	3,49E-3	6,61E-4	1,91E-3	0,2
100U-2L-[[2-4]-[2-2]]	83,5	2,72E-3	2,35E-4	3,65E-3	5,43E-4	3,78E-3	8,36E-4	2,24E-3	0,2
100U-2L-[[4-4]-[4-2]]	68	2,55E-3	2,48E-4	3,63E-3	4,70E-4	3,84E-3	1,06E-3	1,78E-3	0,2
100U-2L-[[6-4]-[4-2]]	68	2,46E-3	2,19E-4	3,70E-3	6,22E-4	3,88E-3	8,54E-4	2,31E-3	0,2
200U-1L-[[2-4]-[2-2]]	89,5	2,95E-3	2,63E-4	3,42E-3	4,92E-4	3,49E-3	8,77E-4	1,60E-3	0,2
200U-1L-[[4-4]-[4-2]]	80,5	2,82E-3	2,46E-4	3,40E-3	4,62E-4	3,45E-3	7,25E-4	2,36E-3	0,2
200U-1L-[[6-4]-[4-2]]	76	2,74E-3	2,15E-4	3,45E-3	4,72E-4	3,54E-3	7,18E-4	2,03E-3	0,2
200U-2L-[[2-4]-[2-2]]	82,5	2,74E-3	2,81E-4	3,66E-3	5,09E-4	3,94E-3	1,02E-3	2,28E-3	0,2
200U-2L-[[4-4]-[4-2]]	73,5	2,50E-3	2,03E-4	3,77E-3	5,98E-4	3,96E-3	9,17E-4	2,01E-3	0,2
200U-2L-[[6-4]-[4-2]]	66,5	2,48E-3	2,39E-4	3,74E-3	5,06E-4	4,01E-3	8,99E-4	2,31E-3	0,2

Table 6.9: **CNN performance**

The table contains the model identifier. Each model is comprised of two CNN layers followed with a MLP. The identifier contains the number of units in the MLP layers and the number of layers. The pairs of numbers in the brackets describe the convolutional layers. Each convolutional layer is described by the number of filters trained for each indicator and the window width . In the following columns the table contains the average amount of epochs needed before the stopping condition has been hit, the average training loss, the average validation loss and the average test loss along with their standard deviations. The last two columns contain the best test loss and the average cpu time it took to train the model in hours. The two best average validation and two best test losses have been underlined.

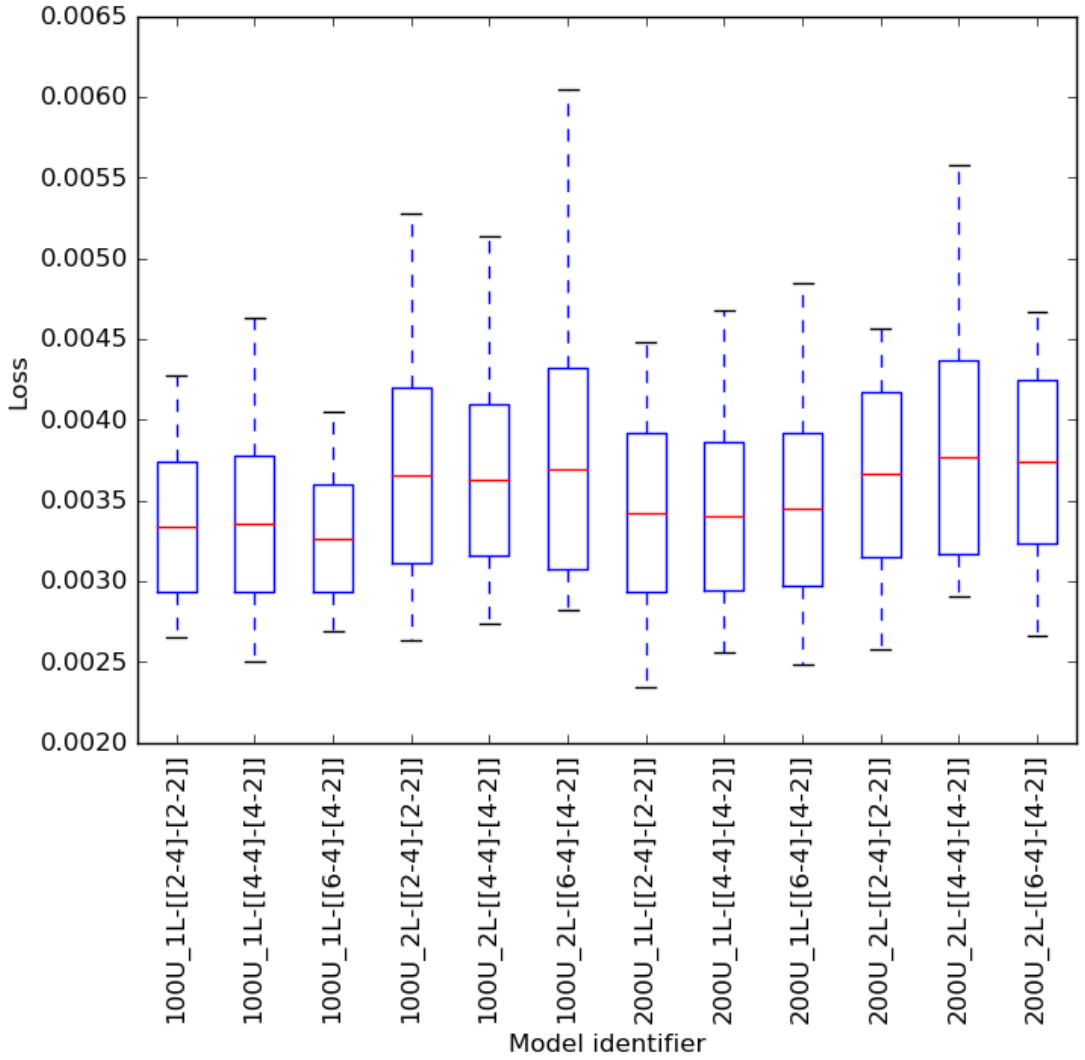


Figure 6.21: **Boxplot CNN prediction models performance.**

The plot contains the average validation loss denoted by the red line, the size of the box represents the standard deviation and the whiskers in the box plot are the minimum and maximum validation loss.

grouping	avg val. loss	std	runs	best test loss	P-value
1-layer models	3,37E-3	4,31E-4	300	1,60E-3	-
2-layer models	3,69E-3	5,42E-4	300	1,78E-3	$< 0{,}0001$
100-unit models	3,49E-3	4,66E-4	300	1,78E-3	-
200-unit models	3,57E-3	5,07E-4	300	1,60E-3	0,04
2-filter models	3,52E-3	4,88E-4	200	1,60E-3	-
4-filter models	3,54E-3	4,89E-4	200	1,78E-3	0,68
6-filter models	3,54E-3	4,83E-4	200	1,91E-3	0,68

Table 6.10: **CNN model performance aggregates**

The table contains the average validation loss over all models with a certain parameter fixed, and the best test loss on the same subset of models. The first column is the fixed parameter, with the other columns being the aggregates. The last column lists pairwise t-test P-values with the other pair being the best performing model aggregate within the specific parameter (denoted by horizontal lines).

Analysis and model demonstration

To analyse the results, we take a look at how the models tend to perform depending on each separate parameter tested. For a list of average validation losses over a certain parameter, review Table 6.6.

The table shows us the aggregate validation losses over the tested parameters. The results show with statistical significance that in the case of CNN-architectures, 1-layer MLP performs better than a 2-layer model. The results also show that 100 unit MLPs perform better. Unfortunately it seems that the performance of the network does not depend on the number of filters in the convolutional layers. The absolute best test losses in fact show that filters might be hurting the networks performance.

Next we take a look at the training process. In general the training process was very similar to that of model `100U-1L-[[2-4]-[2-2]]`, which is shown in Figure 6.22. The training chart shows that the validation loss started stagnating already around the 20-th epoch and hasnt improved much in the last 50 epochs. The test loss and the validation loss seem to still be fairly closely correlated, however a lot less than during MLP training in Figure 6.11.

We now take the best performing model architecture and examine the trained convolutions in closer detail. Examples of how the model `200U-1L-[[2-4]-[2-2]]` transforms two specific indicators using the trained convolutions is presented in Figures 6.23 and 6.24. When training CNNs, the convolutions trained in the first layer are usually simpler transformations, while the following layers combine the transformations into more complicated features.

In the case of our models, the convolutions trained in the first layer could be usually classified into one of: smoothed out original, multiplied/transposed original and prediction of difference. For example in the case of transforming Wholesale share of GDP in Figure 6.23 convolution 1 is probably a convolution detecting larger change in the original convolution. Convolution 2 in the same Figure looks like the original time series smoothed out and possibly scaled down. The first layer convolutions on the Gross capital formation time-series in Figure 6.24 show a similar story. Convolution 0 seems simply like a scaled copy of the

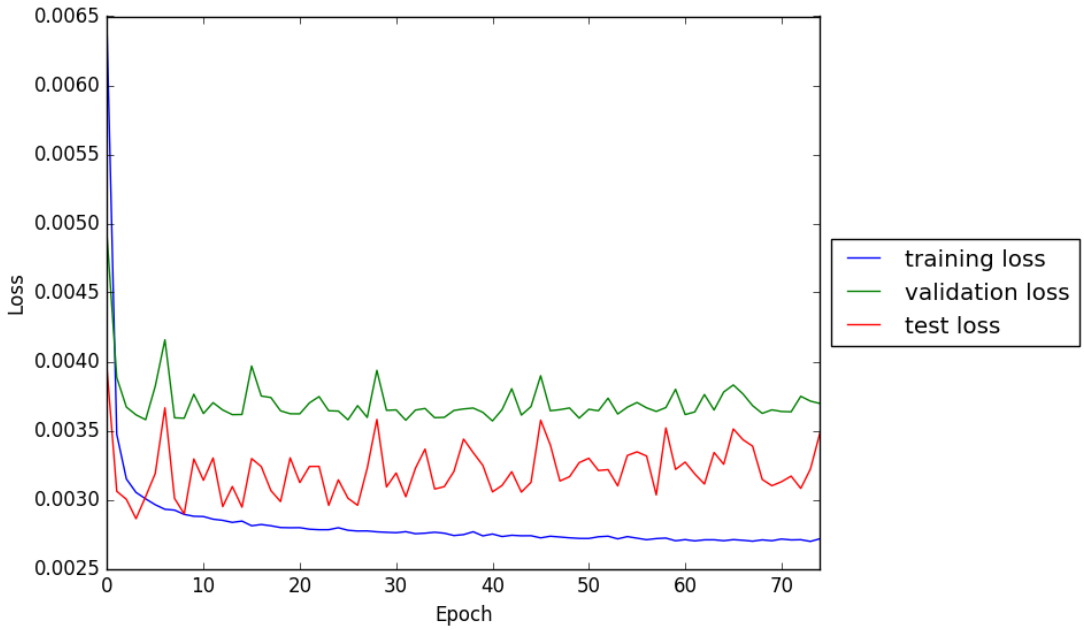
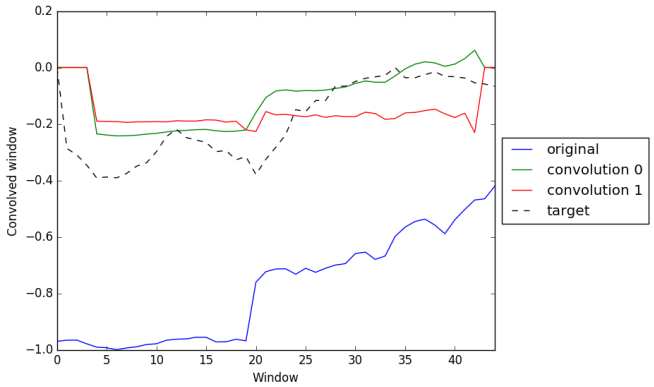


Figure 6.22: **A chart of the loss values over the training process of a CNN network**

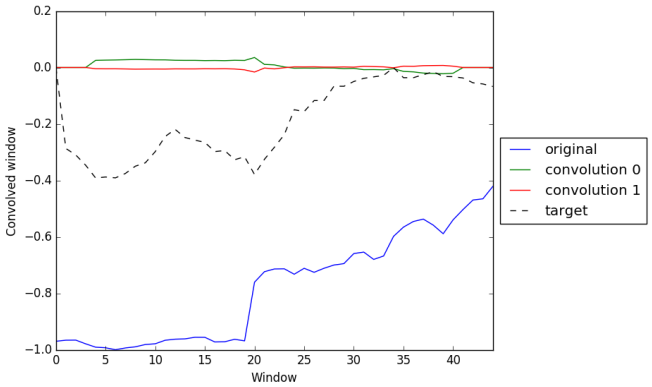
original, while convolution 1 detects difference.

The convolutions trained during image classification are often compound features of the convolutions in the lower layers. In our case the convolutions in the second layer seemed to flatten out the whole time series of both the Wholesale time series and the Gross capital formation time series and only rarely send any of the data forward. The only indicator that the convolutions in the second layer seemed to respond to in greater effect was the GDP indicator as shown in Figure 6.25. These results suggest that the CNNs trained recognise very little valuable information in time-series other than GDP. Even then the model attains a better performance than that of the naive model.

As a final point we examined how the loss rate depends on the year of the window error in Figure 6.26. The error has a similar trend to that of the MLP networks, in that between 1990-1995 the model performs at its worst.

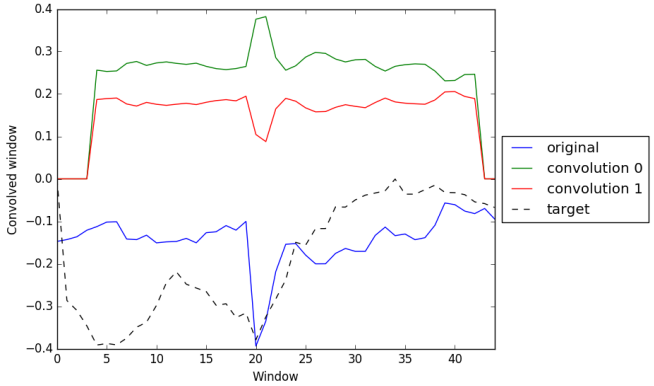


(a) Convolutions in first layer

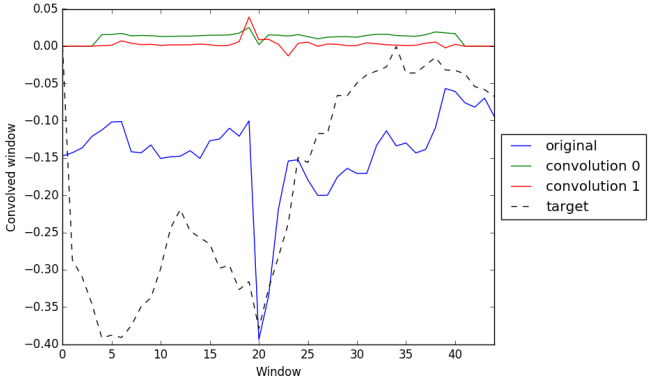


(b) Convolutions in second layer

Figure 6.23: **How a CNN transforms the Wholesale share of GDP of Czech Republic**



(a) Convolutions in first layer



(b) Convolutions in second layer

Figure 6.24: **How a CNN transforms the Gross capital formation share of GDP of Czech Republic**

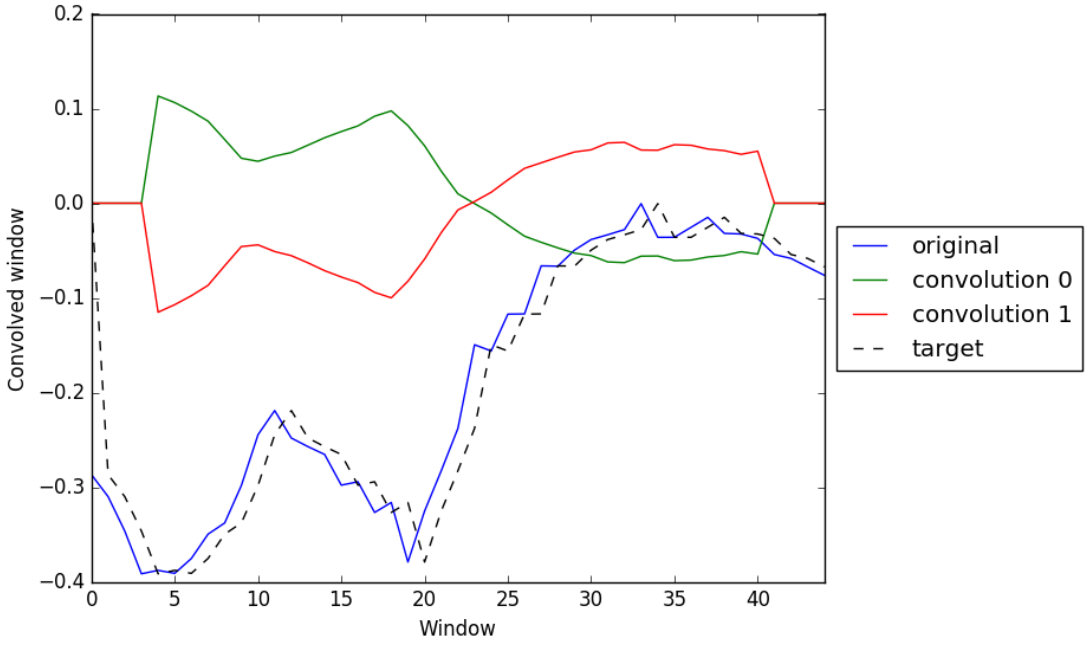


Figure 6.25: **How a CNN in the second layer transforms the GDP of Czech Republic.**

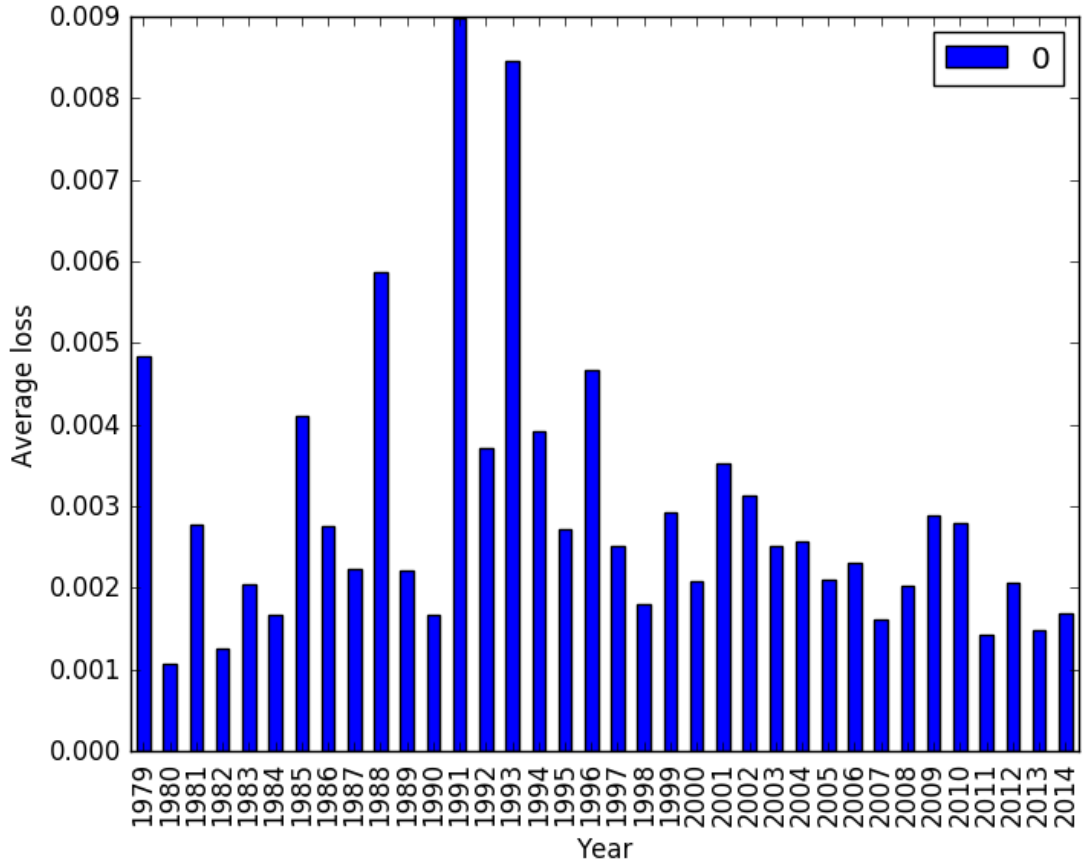


Figure 6.26: **The error rates over the whole dataset of two different models.**

6.3.5 Summary

In the previous experiments we analysed different parameters of MLP networks, CNNs and LSTMs and the models applicability for the task of predicting GDP using a multidimensional macroeconomic time-series.

In Table 6.11 we provide a overall quantitative comparison of the best performing models of each architecture. From this table we can asses that all deep neural network based approches achieved on average at least the performance of the naive model 6.3.1. MLP networks have proven themselves to be a decent baseline model for this task, their average performance slightly better than that of a naive model. CNNs follow this rule, achieving the performance of MLPs. It seems that convolutions do not offer too much when it comes to predictive power, nevertheless they provide an interesting alternative approach. The best performing architecture tested were LSTM networks, with their average performance about 2 times better than that of the naive model.

The table contains the test losses of the best models. If these best runs were on test sets that are independent from the training and validation set, these results would be very nice. It is important to note, that the database is strongly correlated and therefore it is likely that at least in a few runs, the test loss was not independent at all from the training and validation set. Nevertheless the best test losses are included for completion.

architecture	model id	tr. loss μ	val. loss μ	val loss % of naive	test loss μ	best test loss	best % of naive	cpu
NAIVE	NAIVE	4,17E-3	4,17E-3	100%	4,17E-3	4,17E-3	100%	-
MLP	50U-1L-D	3,80E-3	3,37E-3	80,8%	3,45E-3	1,99E-3	47,7%	0,1
MLP	100U-2L-D	3,95E-3	3,38E-3	80,8%	3,44E-3	1,74E-3	41,7%	0,1
CNN	100U-1L-[[2-4]-[2-2]]	2,97E-3	3,34E-3	80,8%	3,43E-3	1,96E-3	47,0%	0,2
CNN	100U-1L-[[6-4]-[4-2]]	2,80E-3	3,27E-3	78,4%	3,49E-3	1,91E-3	45,8%	0,2
LSTM	100U-D-3L	2,06E-3	2,58E-3	61,8%	2,92E-3	2,02E-3	48,4%	3,9
LSTM	100U-N-1L	2,19E-3	2,71E-3	64,9%	2,81E-3	1,41E-3	33,8%	1,2

Table 6.11: **Comparison of deep neural network performance for GDP prediction**

The table contains a comparison of the neural network architectures that were tested to perform well on GDP prediction.

Chapter 7

Conclusion

The aim of this thesis was to compare the viability of several deep neural network architectures for macro-economic data analysis. To this end, the introduced architectures were trained on a prediction task and tested with multiple different parameters. The data used to train the models was real-world publicly available economic data from the United Nations and the World bank.

The problem of time-series forecasting is one of the most publicly visible activities of professional economists. Economic time-series forecasting is present in many aspects of business, such as planning, state and local budgeting, management, financial engineering and national policy. The available research in this field mostly focuses on multilayer neural networks with varying rates of success. In this work we focused on experimenting with multidimensional time-series and whether deep neural networks can improve on the performance of MLP networks. The the network architectures examined included feedforward multilayer neural networks, recurrent neural networks and convolutional networks.

The data used to run the experiments was publicly available data from the World Bank and United Nations including a history macroeconomic indicators of over 200 countries since 1970. We analysed the data and examined it using self organising maps. Afterwards we defined a training task to predict the GDP of a country from its macro-economic history and analysed several available parameters of multilayer neural networks and tested their performance during this prediction. Further we analyzed available parameters of long-short term memory networks and finally the examined how convolutional neural networks predict GDP.

We compared the performance of these networks to a conservative model that is often used in literature. The performance achieved falls in line with the performances achieved in literature. The MLP networks achieved a slightly better performance than the conservative model, while the LSTM networks further improve on this performance. The CNN networks do not improve on the performance of MLPs.

For each architecture type we analysed the influence of different model parameters on performance and formulated statistically grounded hypothesis about them. For MLP and LSTM network architectures, the results have shown that significant non-linearities are present in the experiment data. MLP and LSTM networks with multiple layers performed better than networks with one layer. The results also confirm the general opinion that Dropout regularization per-

forms well. In our experiments we compared Dropout regularization with a more traditional noise regularization, which it outperformed in both the MLP and LSTM architecture experiments. The CNN experiments show that they are not very attractive for economic prediction in general, the best performing CNNs had the least number of filters. This was also confirmed by analysing the convolutional filter function in the networks, where the networks used only a handful of filters in any significant manner.

The experiments were ran using Python and the Keras neural networks library with the Theano backend. Since the computational requirements of the experiments were fairly high we used the services of the computation center MetaCentrum. In total we used over 400 CPU days of computation power. The complete full run of experiments would take about 70 CPU days, however some iteration was necessary, hence the higher absolute total. The experience of working with deep neural networks and their training on a computational cluster was a very beneficial personal experience.

In summary, we have managed to evaluate different neural network architectures and their applicability to time-series prediction. We have shown in this thesis that LSTMs improve on MLPs and that the filters trained by CNNs do not expose any new information to the prediction task. Moreover, we analysed the influence of different model parameters on performance and shown that the data analysed contains significant non-linearities to warrant a multiple layer architecture. Further architecture parameters analysed included neuron count, regularization, transfer functions and filter count. These are what I consider the major contributions of this work.

Bibliography

[1] Włodzisław Duch and Norbert Jankowski. Survey of neural transfer functions. *Neural Computing Surveys*, 2(1):163–212, 1999.

[2] Balazs Csanad Csaji. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24:48, 2001.

[3] Terence D Sanger. Optimal unsupervised learning in a single-layer linear feedforward neural network. *Neural networks*, 2(6):459–473, 1989.

[4] Kenneth O Stanley. Efficient reinforcement learning through evolving neural network topologies. In *In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002*. Citeseer, 2002.

[5] Kate A Smith and Jatinder ND Gupta. Neural networks in business: techniques and applications for the operations researcher. *Computers & Operations Research*, 27(11):1023–1044, 2000.

[6] Mark F Bear, Barry W Connors, and Michael A Paradiso. *Neuroscience*, volume 2. Lippincott Williams & Wilkins, 2007.

[7] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[8] Yves Chauvin and David E Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

[9] James L McClelland, David E Rumelhart, PDP Research Group, et al. Parallel distributed processing, vol. 1 and 2, 1986.

[10] Halbert White. Learning in artificial neural networks: A statistical perspective. *Neural computation*, 1(4):425–464, 1989.

[11] Fernando M Silva and Luis B Almeida. Acceleration techniques for the backpropagation algorithm. In *Neural Networks*, pages 110–119. Springer, 1990.

[12] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[13] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

[14] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814, 2010.

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

[16] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

[17] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.

[18] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145, 1995.

[19] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[20] Teuvo Kohonen. Self-organising maps. *Springer Science & Business Media*, 30, 2001.

[21] Jonathan Owens and Andrew Hunter. Application of the self-organising map to trajectory classification. In *Visual Surveillance, 2000. Proceedings. Third IEEE International Workshop on*, pages 77–83. IEEE, 2000.

[22] Guido Deboeck and Teuvo Kohonen. *Visual explorations in finance: with self-organizing maps*. Springer Science & Business Media, 2013.

[23] Rasika Amarasiri, Damminda Alahakoon, Kate Smith, and Malin Premaratne. Hdgsomr: a high dimensional growing self-organizing map using randomness for efficient web and text mining. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 215–221. IEEE Computer Society, 2005.

[24] Leo Bispo. Us congress som. `https://github.com/leobispo/som`, 2012.

[25] Samuel Kaski and Teuvo Kohonen. Exploratory data analysis by the self-organizing map: Structures of welfare and poverty in the world. In *Neural networks in financial engineering. Proceedings of the third international conference on neural networks in the capital markets*. Citeseer, 1996.

[26] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[27] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[28] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[29] Les E Atlas, Toshiteru Homma, and Robert J Marks II. An artificial neural network for spatio-temporal bipolar patterns: Application to phoneme classification. In *Proc. Neural Information Processing Systems (NIPS)*, page 31, 1988.

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[31] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[32] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *European Conference on Computer Vision*, pages 184–199. Springer, 2014.

[33] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[34] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2414–2423, 2016.

[35] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[36] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[37] Yedid Hoshen, Ron J Weiss, and Kevin W Wilson. Speech acoustic modeling from raw multichannel waveforms. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4624–4628. IEEE, 2015.

[38] Patrice Y Simard, Léon Bottou, Patrick Haffner, and Yann LeCun. Boxlets: a fast convolution algorithm for signal processing and neural networks. *Advances in Neural Information Processing Systems*, pages 571–577, 1999.

[39] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[40] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[41] Herbert Jaeger. The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148:34, 2001.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[43] Barbara Hammer. On the approximation capability of recurrent neural networks. *Neurocomputing*, 31(1):107–123, 2000.

[44] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013.

[45] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[46] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[47] Alex Graves, Douglas Eck, Nicole Beringer, and Juergen Schmidhuber. Biologically plausible speech recognition with lstm neural nets. In *International Workshop on Biologically Inspired Approaches to Advanced Information Technology*, pages 127–136. Springer, 2004.

[48] Thomas M Breuel, Adnan Ul-Hasan, Mayce Ali Al-Azawi, and Faisal Shafait. High-performance ocr for printed english and fraktur using lstm networks. In *2013 12th International Conference on Document Analysis and Recognition*, pages 683–687. IEEE, 2013.

[49] Daniel Soutner and Luděk Müller. Application of lstm neural networks in language modelling. In *International Conference on Text, Speech and Dialogue*, pages 105–112. Springer, 2013.

[50] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. Hybrid speech recognition with deep bidirectional lstm. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pages 273–278. IEEE, 2013.

[51] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

[52] Ronald J Williams and David Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. *Back-propagation: Theory, architectures and applications*, pages 433–486, 1995.

[53] Alex Graves, Nicole Beringer, and Juergen Schmidhuber. Rapid retraining on speech data with lstm recurrent networks. Technical report, Technical Report IDSIA-05-05, IDSIA, www. idsia. ch/techrep. html, 2005.

[54] Daan Wierstra, Faustino J Gomez, and Jürgen Schmidhuber. Modeling systems with internal state using evolino. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1795–1802. ACM, 2005.

[55] Jürgen Schmidhuber, Daan Wierstra, Matteo Gagliolo, and Faustino Gomez. Training recurrent networks by evolino. *Neural computation*, 19(3):757–779, 2007.

[56] United Nations national accounts main aggregates database. `http://unstats.un.org/unsd/snaama/introduction.asp`. Accessed: 2016-02-01.

[57] World Bank open data. `http://data.worldbank.org/`. Accessed: 2016-02-01.

[58] United Nations. International standard industrial classification of all economic activities (isic) rev. 4. *United Nations Statistical Papers*, (4), 2008.

[59] Olga Troyanskaya, Michael Cantor, Gavin Sherlock, Pat Brown, Trevor Hastie, Robert Tibshirani, David Botstein, and Russ B Altman. Missing value estimation methods for dna microarrays. *Bioinformatics*, 17(6):520–525, 2001.

[60] Guang-Bin Huang. Learning capability and storage capacity of two-hidden-layer feedforward networks. *IEEE Transactions on Neural Networks*, 14(2):274–281, 2003.

[61] Masahiro Ashiya. Forecast accuracy of the japanese government: Its year-ahead gdp forecast is too optimistic. *Japan and the World Economy*, 19(1):68–85, 2007.

[62] Greg Tkacz. Neural network forecasting of canadian gdp growth. *International Journal of Forecasting*, 17(1):57–69, 2001.

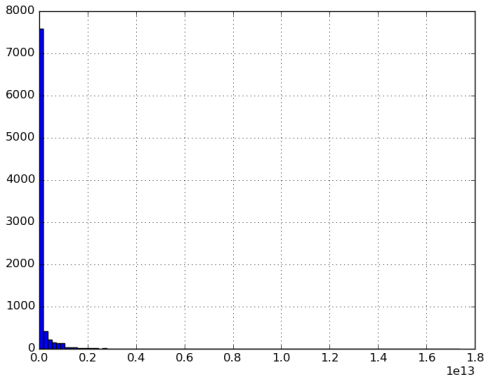
List of Figures

List of Tables

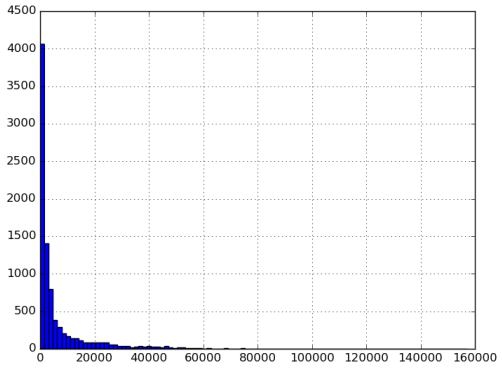
Appendices

Appendix A

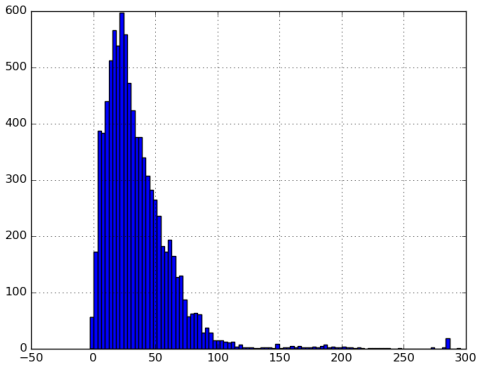
Indicator Distributions



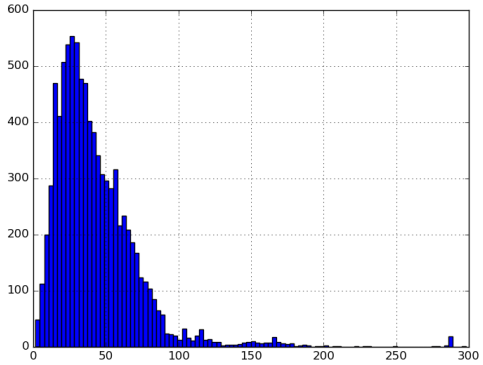
(a) Absolute Gross Domestic Product (GDP)



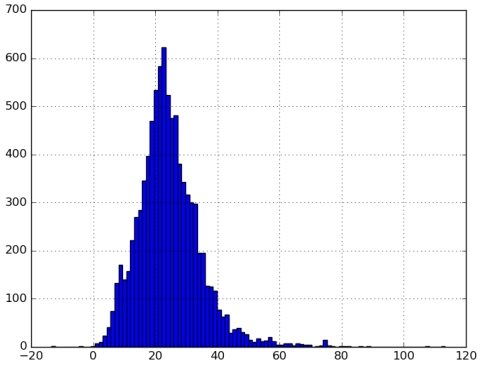
(b) GDP Per Capita



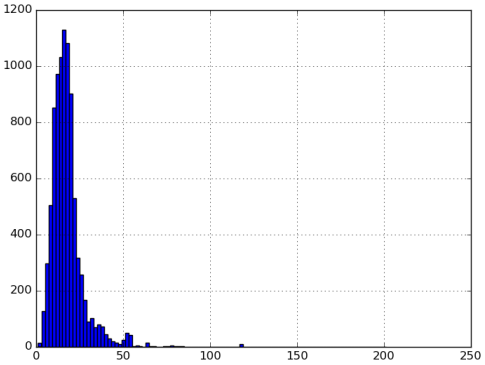
(c) Exports of goods and services



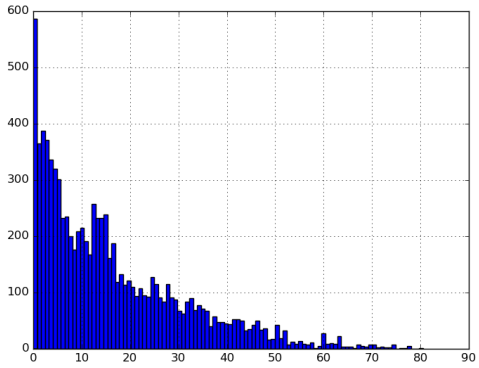
(a) Imports of goods and services



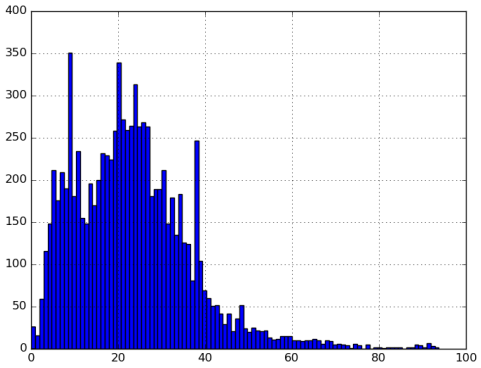
(b) Gross capital formation



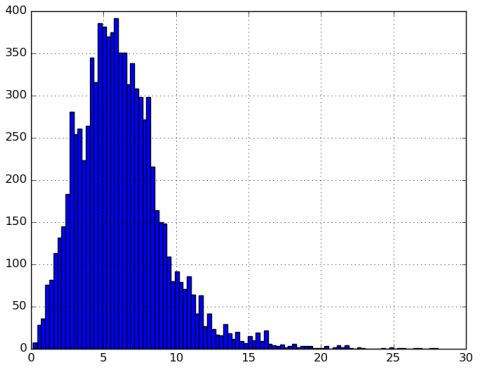
(c) General government final consumption expenditure



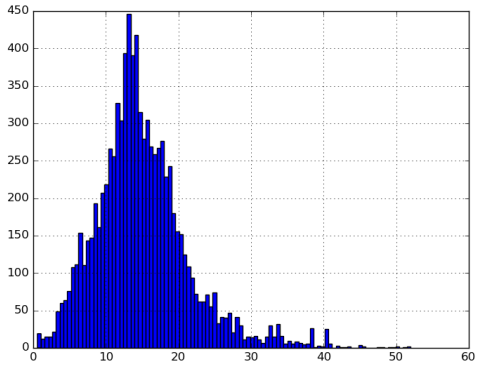
(a) Agriculture, hunting, forestry, fishing (ISIC A-B)



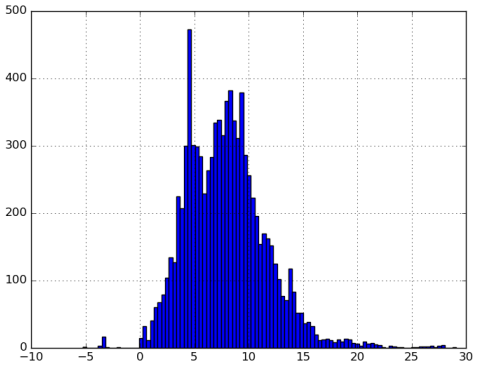
(b) Mining (ISIC C-E)



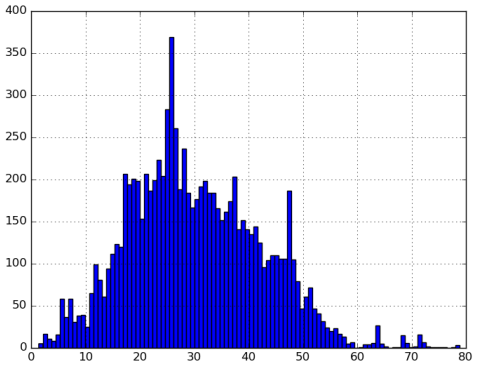
(c) Construction (ISIC F)



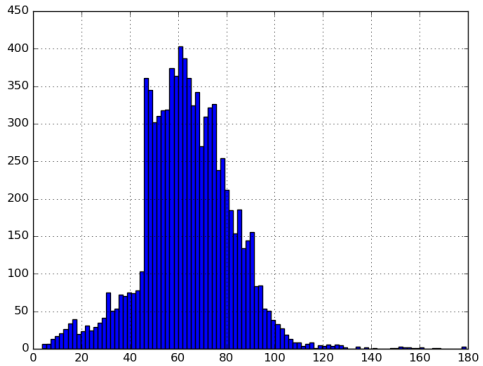
(a) Wholesale (ISIC G-H)



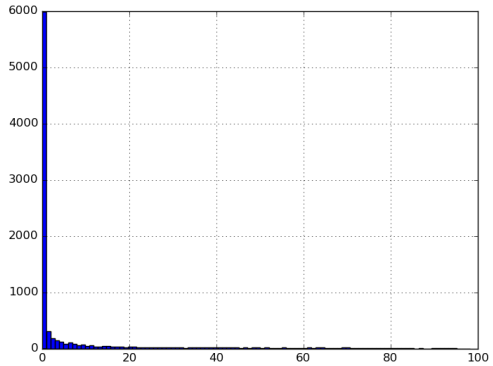
(b) Transport (ISIC I)



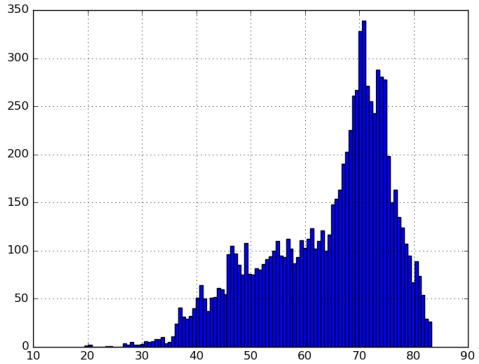
(c) Other Activities (ISIC J-P)



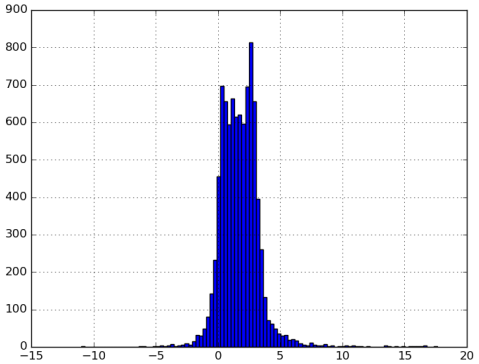
(a) Household consumption expenditure



(b) Internet users (per 100 people)



(a) Life expectancy at birth, total (years)



(b) Population growth (annual %)

Appendix B

Implementation and documentation

In this appendix we describe the contents of the appended DVD and how to run the experiments using our code.

B.1 Project overview

The following diagram describes the project directory structure and the locations of the notable files and scripts in the attachment:

```
/
├── thesis.pdf/
├── results/
│   ├── dense_configs/
│   ├── lstm_configs/
│   └── cnn_configs/
└── src/
    ├── mlp_experiment.py
    ├── lstm_experiment.py
    ├── cnn_experiment.py
    ├── som_experiment.py
    ├── configs/
    ├── database.db
    ├── data_src
    └── ...
```

The data is separated into two folders, the `results` folder contains the experiment results - the loss history files, model weights and other statistics. The `src` folder contains the scripts that were used to run the experiments and other utility scripts, such as scripts that can be used to load weights of a trained model and use the model for prediction etc.

B.2 Requirements and installation

The project scripts run on Python3.4, the models are trained using the Keras with the Theano backend libraries. The plots are made using the matplotlib and

Pandas python libraries.

The list of debian packages required to run the project is placed in `src/apt-requirements.txt`. The list of `pip` requirements is placed in `src/requirements.txt`. An installation script made for Ubuntu based systems installing the necessary requirements is placed in `src/install.sh`. Except that the script installs the requirements, it edits the Keras configuration file so that Keras uses the Theano backend.

The scripts create files with results (weights, logs) in their local directory. Before running scripts copy the contents of the DVD to a local drive and run the scripts from the `src` subdirectory to avoid unforseen interactions.

B.3 Browsing Results

There are no installation requirements to browse the results of experiments. The results are structures in 3 stages:

1. the architecture directory (dense, lstm, cnn)
2. parameter identifier directory (50U-1L-D-AGDP-lerelu)
3. the run directory (1-10)

The results are segmented into directories based on their model architecture type (cnn, dense, lstm). Each experiment directory contains directories for each parameter permutation tested. For the exact meaning of the parameter identifier, review Chapter 6. Each parameter permutation contains 10 folders for different runs of the experiment. Each run then contains weights for 5 models, each for one of the model from the cross-validation folds. An outline of the folder structure for one run of 5 folds is:

```
dense_configs/
├── 50U-1L-D-AGDP-lerelu.json/
│   └── 1-10/
│       ├── fold0-model
│       ├── fold0-model.csv
│       ├── ...
│       ├── model.json
│       ├── run_stats
│       └── toss.npz
└── ...
```

Each experiment folder contains 5 sets of weight files each named `fold[0-4]-model` and a history of this training run in `fold[0-4]-model.csv`. The run folder then contains a json file containing the exact model definition as exported from Keras in `model.json`. The experiments were ran on the MetaCentrum computation center and the `run_stats` file contains an information about the execution, including CPU time used. Finally the run folder contains a file that contains the k-fold random toss value for each country in the file `toss.npz`.

The history of each training k-fold run in is saved in a csv named `fold[0-4]-model.csv`. The csv contains three columns `epoch`, `loss`, `val_loss` and `test_loss`. Each row of the csv file represents one epoch during the training and the

row values are the training, validation and test losses recorded during training. The final row shows the final performance of this run of the experiment.

B.4 Running experiments

The experiments were ran using scripts in the `src` folder:

1. `mlp_experiment.py` - Script used to train and test MLP networks
2. `lstm_experiment.py` - Script used to train and test LSTM networks
3. `cnn_experiment.py` - Script used to train and test CNN networks

The scripts expose commands as switches to perform various different tasks relevant to the model in question. Each script has a way to train a model and create some plots based on the results. The CNN script can plot the convolutions in each layer. The MLP script can plot a error histogram. This functionality will be discussed in another section.

To train a specific architecture, select one of the parameter configurations from the `src/configs/*_configs`. Each configuration is saved in a json and contains parameter values for the NN architecture. Certain architectures use different configs. An example of a config is shown in Listing B.1.

Listing B.1: JSON config

```
{
  "hidden_units":50,
  "noise_epoch":false,
  "num_layers":1,
  "use_dropout":true,
  "max_epoch_count":250,
  "target_indicator":"Absolute_Gross Domestic Product (GDP)",
  "activation_function":"lerelu",
  "lerelu_alpha":0.3
}
```

The config contains the number of hidden units in each layer (50), whether or not to noise the data in each epoch, the number of layers, whether or not to use dropout, the maximum epochs to train, the target indicator, the activation function used (lerelu or tanh) and an optional lerelu parameter. The folders under `src/configs` contain all the network configurations used in this work.

To run an experiment you run the relevant python script with the -train switch and provide it the path to the configuration file. An example of how a MLP network would be trained

Listing B.2: Training MLP networks

```
python3 mlp_experiment.py -train ./configs/dense_configs
    /test_config.json
```

After an experiment is finished running, it will have created weight files, loss history files and a `toss.npy` file as described higher.

B.5 Evaluating and plotting results

After an experiment is finished running, we can use the created weight files to evaluate the model and plot the predictions made by the model. To do this we use the same scripts as used to run the experiments:

1. `mlp_experiment.py`
2. `lstm_experiment.py`
3. `cnn_experiment.py`

B.5.1 Main analysis

The functionality available to all three scripts is to evaluate the models MSE and to plot the prediction. The functionality is invoked using either the `-plot` or the `-eval` switch. Both commands require a config file to be provided from the `src/configs/` folder and the weight file to be used during the evaluation and plotting.

Plotting

To plot we need to use the `-plot` switch. The plot command further requires a config file to be provided from the `src/configs/` folder and the weight file to be used during the evaluation and plotting. An optional parameter is a country code of which country to plot. If no country is provided, all countries are plotted. The list of country codes available is in `src/data_src/country_codes.csv`. The syntax of this command is in Listing B.6. The plots of the countries are outputted in a subfolder `mlp_plots` (or lstm/cnn).

Listing B.3: Syntax of -plot command

```
1 mlp_experiment.py -plot config -weights foldmodel [-
    country ccode]
```

An example of plotting the GDP prediction of Czech Republic using a MLP network trained in the experiments is shown below

Listing B.4: Plotting GDP prediction

```
1 python3 mlp_experiment.py -plot ./configs/dense_configs
    /50U-1L-D-AGDP-lerelu.json -weights ../results/
    dense_configs/50U-1L-D-AGDP-lerelu.json/1/fold0-model
    -country CZE
```

Evaluating

To evaluate a model we need to use the `-eval` switch. The eval command further requires a config file to be provided from the `src/configs/` folder and the weight file to be used during the evaluation and plotting. An optional parameter is the cross-validation `toss.npy` file to output MSE on all folds. If no toss is provided, overall MSE is output. The syntax of this command is in Listing B.5.

Listing B.5: Syntax of -eval command

```
1 mlp_experiment.py -eval config -weights foldmodel [-toss
      tossfile]
```

An example of how to evaluate the GDP prediction of a MLP network trained in the experiments is shown below

Listing B.6: Evaluating GDP prediction

```
1 python3 mlp_experiment.py -eval ./configs/dense_configs
    /50U-1L-D-AGDP-lerelu.json -weights ../results/
    dense_configs/50U-1L-D-AGDP-lerelu.json/1/fold0-model
    -toss ../results/dense_configs/50U-1L-D-AGDP-lerelu.
    json/1/toss.npy
```

B.5.2 Futher analysis

The different scripts expose further functionality specific for different network architectures. For example the `cnn_experiment.py` script can plot how the convolutions in a network transform a specific country.

Convolutions

After a CNN is finished training, we might want to look at what the convolutions trained do to an input. To do this we can use the command invoked by the switch `-convolutions` in the `cnn_experiment.py` script. The syntax of the command is shown the listing below

Listing B.7: Syntax of -convolutions command

```
1 mlp_experiment.py -convolutions config -weights
    foldmodel -country ccode
```

The command plots how the convolutions in the first and second layer of the network transform the indicators of the specified country. The convolutions are saved in the folder `cnn_plots/`. An example of how to plot convolutions is in the listing below

Listing B.8: Plotting convolution transformations

```
1 python3 cnn_experiment.py -convolutions ./configs/
    dense_configs/50U-1L-D-AGDP-lerelu.json -weights ../
    results/dense_configs/50U-1L-D-AGDP-lerelu.json/1/
    fold0-model -country CZE
```

Error histogram

The MLP and CNN script files expose a functionality to plot the error rate of a network dependent on the target year. To invoke this functionality use the switch `-eval_hist`.The syntax of the command is shown the listing below

Listing B.9: Syntax of -convolutions command

```
1 mlp_experiment.py -eval_hist config -weights foldmodel
```

The command plots a histogram of the loss rate over time into `mlp_plots/` (cnn respectively). An example of how to draw histograms is in the listing below:

Listing B.10: Plotting convolution transformations

```
1 python3 mlp_experiment.py -eval ./configs/dense_configs
    /50U-1L-D-AGDP-lerelu.json -weights ../results/
    dense_configs/50U-1L-D-AGDP-lerelu.json/1/fold0-model
```

Compound plot

The LSTM script exposes a functionality to plot the compound variable as defined in section 6.3.3. To invoke this functionality use the switch `-compound_plot`.The syntax of the command is the same as the syntax of the plot command.

Listing B.11: Syntax of -compound_plot command

```
1 lstm_experiment.py -compound_plot config -weights
    foldmodel [-country ccode]
```

An example of plotting the GDP prediction of Czech Republic using a MLP network trained in the experiments is shown below

Listing B.12: Compound plotting GDP prediction

```
1 python3 lstm_experiment.py -compound_plot ./configs/
    dense_configs/50U-1L-D-AGDP-lerelu.json -weights ../
    results/dense_configs/50U-1L-D-AGDP-lerelu.json/1/
    fold0-model -country CZE
```

B.6 Data preparation and implementation details

The data is exposed to the scripts via a sqlite interface impemented in `shared.py`. The sqlite database is saved in the file `database.db` and was created from csv files downloaded from the sources mentioned in the thesis. The script `prepare_data.py` parses the csv files, replaces missing values, normalizes it and exports the data to the sqlite database.

The `shared.py` is the script file containing most of the implementation guts. The classes implemented are:

1. **Database** - an object holding the sqlite connection handle
2. **CountryIndicatorTable** - an object exposing the table containing the data used in the experiments via various getter methods
3. **ArgumentParse** - a utility class that loads a json and replaces default values with the values in the json
4. **ArgumentParse** - a utility class parsing command line parameters
5. **ExperimentData** - a wrapper class for preparing data for the NN experiments

6. **Model** - a superclass of a model, creating an interface for creating, saving and training models
7. **MLPModel** - subclass of Model implementing the MLP specifics
8. **LSTMModel** - subclass of Model implementing the LSTM specifics
9. **CNNModel** - subclass of Model implementing the CNN specifics
10. **SOM** - a class implementing a SOM algorithm (can be used separately from shared.py)

The models are created using the Keras Sequential API. An example of how the MLP network was implemented is in the following listing

Listing B.13: Keras Sequential API example

```
model = Sequential()

model.add(Flatten(input_shape=input_shape[1:]))
for i in range(self.config['num_layers']):
        if i==0:
                print(input_shape[1:])
                model.add(Dense(self.config['
                   hidden_units']))
        else:
                model.add(Dense(self.config['
                   hidden_units']))

        if self.config['activation_function']=='tanh':
                model.add(Activation('tanh'))
        else:
                model.add(LeakyReLU(alpha=self.config['
                   lerelu_alpha']))

        if self.config['use_dropout']:
                model.add(Dropout(0.5))

model.add(Dense(target_shape[1],input_shape=input_shape
    [1:]))
model.add(Activation('linear'))

model.compile(loss='mean_squared_error',
              optimizer='rmsprop',
              metrics=['accuracy','mean_absolute_error
                 '])
```

The LSTM and CNN models were implemented using a similar approach. An important thing to note is that the SOM class has been implemented without the Keras library and is therefore standalone.