University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

Master's thesis

Time Series Forecasting using Deep Neural Networks

Plzeň 2019 Bc. Lada Zadranská

Místo této strany bude zadání práce.

Declaration

I hereby declare that this master's thesis is completely my own work and that I used only the cited sources.

Plzeň, 27th June 2019

Bc. Lada Zadranská

Abstract

Recurrent Neural Networks are models designed to operate over sequential data, used for classification and regression tasks. Types of Recurrent Neural Networks are described in this thesis and the algorithms are used in the implementation of a baseline model for time series forecasting. Grid Search or Bayesian Optimisation are strategies that assist in finding the best combination of hyperparameters – variables, which have a great impact on the process of learning.
The purpose of the thesis is to find whether Side-Channel Injection could improve the accuracy of predictions using RNNs. Mean and slopes and intercepts of a fitted line are fed as new variables with input data to the networks and the functionality of this method is verified using the implementation of the two models. Another RNN structure is introduced as the Augmented RNN model as a compromise between simple RNNs and RNN Autoencoders. Several experiments were conducted for the aforementioned models and the Augmented RNN model was selected as the model with the best performance using Akaike's Information Criterion, Hypothesis Testing and visual result comparison. Another contribution of this thesis is a detailed overview of the process of implementing RNN Autoencoders and the techniques used for hyperparameters optimisation.

Keywords: Machine Learning, Recurrent Neural Networks, Long-Short Term Memory, Gated Recurrent Unit, Seq2Seq, Time Series Forecasting, Side-Channel Injection, TensorFlow.

Abstrakt

Rekurentní neuronové sítě jsou modely pracující s posloupnostmi dat používané pro klasifikační i regresní úlohy. Typy rekurentních neuronových sítí jsou definovány v této práci společně se svými algoritmy, které jsou použity při implementaci výchozího modelu. Grid Search či Bayesovská optimalizace jsou metody pomáhající nalézt optimální hodnoty hyperparametrů – proměnných, které ovlivňují rychlost a přesnost učení. Tyto metody byly použity k nastavení neuronové sítě.
Cílem této práce bylo zjistit, zda může metoda injekce postranních kanálů zlepšit přesnost predikcí rekurentních neuronových sítí. Průměr a směrnice a úsek přímky dané lineární regresí jsou použity jako nové proměnné vstupních dat a funkčnost této metody byla ověřena implementací těchto dvou modelů. Další struktura rekurentní neuronové sítě je definována jakožto Rozšířený RNN model, který je kompromisem mezi klasickou rekurentní neuronovou sítí a autoenkodéry využívajícími rekurentní neuronové sítě. Pro všechny modely bylo provedeno několik experimentů a Rozšířený RNN model byl za pomoci Akaikova informačního kritéria, testování hypotéz a analýzy výsledků na základě pozorování grafů vybrán jako nejlepší model. Dalším přínosem této práce je detailní přehled postupů při implementaci autoenkodérů využívajících rekurentní sítě a popis technik použitých při optimalizaci hyperparametrů.

Klíčová slova: Strojové učení, rekurentní neuronové sítě, Long-Short Term Memory, Gated Recurrent Unit, Seq2Seq, predikce časových řad, injekce postranních kanálů, TensorFlow.

Acknowledgements

I would first like to thank my thesis advisor Ing. Kamil Ekštein, Ph.D. for his guidance and patience. My thanks also go to Ing. Jakub Sido and Ing. Ondřej Pražák for their help with implementation. Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Contents

1 Introduction

Over the past couple of years, machine learning has become very popular and its range of applications is still increasing. This thesis focuses on univariate time series forecasting using neural networks. The thesis is structured in two logical segments – the theoretical part and the practical part. Each segment consists of multiple sections.

The first section introduces Recurrent Neural Networks with a brief history, the mathematical background of their models and possible structures and applications. There is also an overview of machine learning algorithms to differentiate between the supervised learning algorithm, that is used in this thesis, and others. The problems that may occur when training these types of networks are summarised at the end of this section, together with several solutions of how to avoid them. Some of them will be explained in further detail and applied in the programming part of this thesis.

Long Short-Term Memory networks and Gated Recurrent Units are described in the second section. Similarly the whole algorithm is examined of how the predictions are computed. Next, Autoencoders are recapitulated and the advantages of LSTM Autoencoders using a machine translation task are explained.

The difference between parameters and hyperparameters is clarified in the following section, where a list of hyperparameters used in the practical part is provided. Each of these is analysed with their impacts on a model and the learning process, and possible methods of setting the hyperparameters are proposed.

Time series and possible metrics for model selection are described in the following section. Simple linear regression is performed to extend the model by adding information about the trend of the time series.

The practical part introduces the datasets used for forecasting and methods for reshaping the data in order to create a supervised learning problem.

Side-channel injection as a method of how to provide the data with a new information derived directly from the dataset is described in the following section. To create a new model using side-channel injection is the main target of this thesis.

A brief overview of programming language and the libraries used is provided afterwards.

Thereafter, a basic LSTM Autoencoder model is defined, together with four

extensions. Each model is run with the three presented datasets.
At the end of the practical part of this thesis, the results of the models are compared with each other using Hypothesis testing, Akaike's Information Criterion and an analysis w.r.t. visualisations of the predictions.

2 Recurrent Neural Networks

In this chapter, Recurrent Neural Networks are discussed, together with their importance in sequential data processing via artificial neural networks. Readers are introduced to the basics of the whole topic, as well as to the mathematical background.

Recurrent Neural Networks (RNNs) are Artificial Neural Network (ANN) models proposed in the 1980s (Rumelhart et al., 1986; Elman, 1990; Werbos, 1988) that allow us to operate over sequences of vectors. Due to their chain-like structure, RNNs are mostly used as generative models (see Section 3.1). They are applied in handwritten text generation, machine translation, speech recognition, video classification, image captioning and other tasks. The crucial difference between Feedforward Neural Networks (FNNs) and RNNs is that RNNs have a memory in which they store information computed from previous inputs, i.e. the last output is influenced not only by the previous input but by all the inputs that were fed in the network. Figure 2.1 shows a few examples of how the structure of the network may be designed, depending on whether the input or output (or both) is a sequence. Red rectangles represent input vectors, blue rectangles represent output vectors and green rectangles represent (hidden) RNN blocks. There is a dataflow not only from the input layer through the hidden layer to the output layer, but the green arrows represent the flow between the RNN block and its successor. In Figure 2.2 there is no flow between neurons in the same layer, unlike in Figure 2.3. Feeding the input data into FNN and RNN differs. Considering a sentence as input data in FNN, in RNN the sentence would be split into words (or characters, depending on the task) and one word would be fed at a time.

Figure 2.1: RNN structures based on [38]

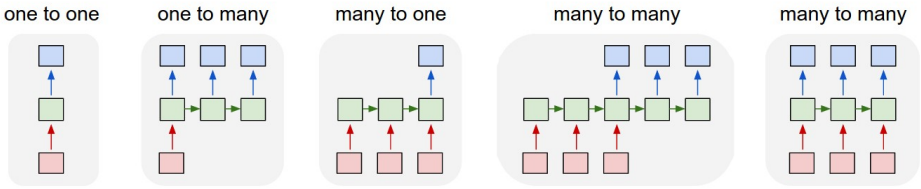


Figure 2.2: FNN based on [63]

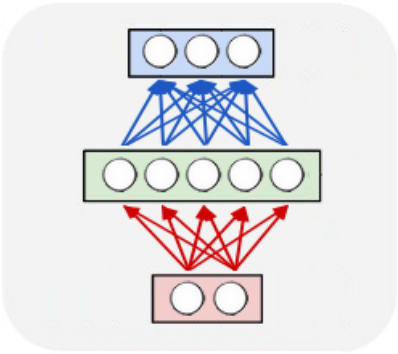
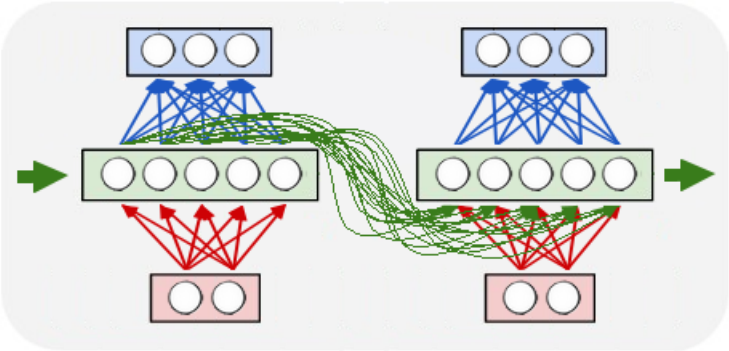


Figure 2.3: Unrolled RNN for 2 time steps



2.1 Machine learning algorithms

Prior to a deeper explanation of RNNs, machine-learning algorithms and the problems they solve need to be specified. According to their purpose, machine-learning algorithms can be divided into three categories:

- supervised learning,
- unsupervised learning,
- reinforcement learning.

In order to make a system to learn and to be able to evaluate the model accuracy, the input dataset is usually divided into training data and test data. The training data are used to fit the parameters of the selected model, whereas the test dataset is meant to verify how well the model performs on data that were not used during training. To get the best results (possible), the test data should be from the same distribution as the training data. Sometimes a validation dataset is also used, constructed from the training data. This set cannot be used during the learning phase, it is intended to assist the updating of hyperparameters correctly. Section 4 [29] provides an explanation of what a hyperparameter is.
The training dataset of supervised learning algorithms is labelled. Its data consist of an input data and desired output. During training, the network learns patterns to output these target values.
Unsupervised learning algorithms have only input data – instead of creating a mapping from input variables to target variables, the model finds relationships between the inputs and groups them together. This approach is usually used when data are not able to be manually labelled, or when it is desired to detect patterns that are not obvious at first sight.
During reinforcement learning, the model finds all the possible states and picks one to maximise its performance. The system's behaviour is regulated by a system of rewards that is determined in advance.
Some of the sources [24] also mention semi-supervised learning algorithms. The cost of labelling the data can be high, thus we can manually label some of the unlabeled data, train the model with supervised learning methods and then feed the remaining data as test data. Predicted labels for these data are obtained and the model is retrained [24].
In this thesis, the supervised learning algorithm is used, so its algorithm will be further addressed. It solves classification or regression problems. In classification tasks, the targets can be divided into discrete classes and the input variables are assigned to these classes, whereas in regression problems the targets are continuous output variables [27]. According to the problem and task, the objective function, which computes the total error between the output and target values, is chosen.

2.2 Mathematical background

The flow between two layers in FNN models represents a mathematical operation using weight matrices and biases. The process of learning consists of four steps.
The first step is a feedforward propagation. During feedforward propagation, the model feeds the input data into the network, computes the values of neurons in each hidden layer (in the direction of the arrows) and then the value of output. The next step can differ with respect to the type of learning algorithm. However, as mentioned earlier, the supervised learning algorithm will be used.
The second step is the comparison of the real output with the predicted one, passing these variables as arguments to an objective function (explained in Section 4.4).
As a third step, the backpropagation is performed starting from the output layer back to the input layer, computing the gradients of the objective function with respect to the weights and biases of the network.
During the fourth step, the values of these parameters are updated with respect to the computed gradients in order to minimise the total error.
The difference between FNN and RNN is that the input going to the hidden layer in RNNs consists not only of the input vector from the input layer but also of the hidden state. A hidden state is a unit which sees to its own history. In his work, Elman [23] describes a new network based on Jordan's network published in 1986 [37]. Both networks use recurrent connections to create a memory. While Jordan used these connections to feed previous output to the hidden layer (see Equation 2.1), Elman feeds activation of the previous hidden state (in his work named "Context Unit") to the following one (see Equation 2.2). Both these networks are depicted in Figures 2.4 and 2.5.

$$\begin{aligned} h_t &= \sigma_h(W_{xh}x_t + W_{hh}y_{t-1} + b_h) \\ y_t &= \sigma_y(W_{hy}h_t + b_y) \end{aligned} \tag{2.1}$$

$$\begin{aligned} h_t &= \sigma_h(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\ y_t &= \sigma_y(W_{hy}h_t + b_y) \end{aligned} \tag{2.2}$$

Figure 2.4: Jordan's recurrent network [23, 183]

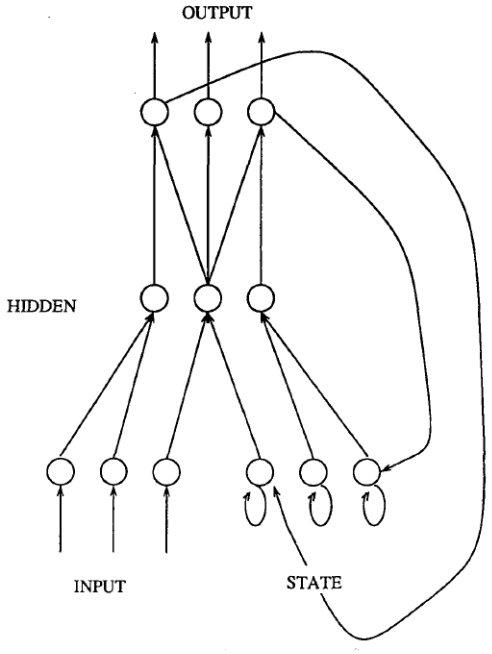
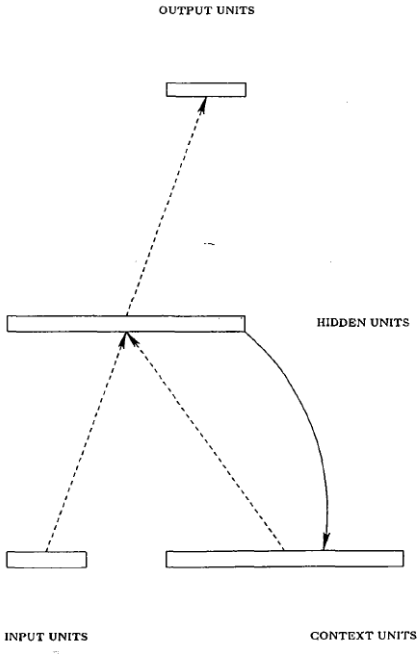


Figure 2.5: Elman's recurrent network [23, 184]



These models are similar simple three-layered networks; however, Elman's network is the currently used one in deep learning frameworks such as TensorFlow (to be explained later in this thesis).
To better understand the difference between FNNs and RNNs and also the

Jordan's and Elman's networks, in Section 2.2.1 the computations are explained step by step.

2.2.1 Feedforward propagation

Assume a simple three-layered FNN, consisting of an input layer, one hidden layer and an output layer as the one shown in Figure 2.2. We have an input x – vector of length N, the hidden layer with vector h of length H and the output vector $\hat{y}$ of length M. To connect each element of a vector x with each element of a vector h (it is a fully-connected, sometimes also called "dense" network) we define weight matrix[1] W_{xh} of size $H \times N$ and also bias b_h to improve the properties of a neuron[2]. Bias b_h is a vector of length H. To be able to learn even non-linear functions, we also define a non-linear activation function σ_h (more about activation functions in Section 4.2).
To compute the values of elements of the output vector, we repeat this technique and define a matrix W_{hy} of size $M \times H$, a bias b_y of length M and an activation function σ_y.
The process of computation is described in Equation 2.3.

$$\begin{aligned} h &= \sigma_h(W_{xh}x + b_h) \\ \hat{y} &= \sigma_y(W_{hy}h + b_y) \end{aligned} \tag{2.3}$$

In Equation 2.2 another variable t representing the individual time steps of a sequence can be seen. If we have an input sequence x of length N, where at single time t ($t = 1, \ldots, N$) we feed only one element of the sequence (scalar) to the network, we denote the element at this time step x_t[3]. The hidden layer h compounds of N vectors of length H called the "hidden states". Each vector is fed into the "RNN cell" which outputs the hidden state at the time $t + 1$. As we need a predecessor to output the first hidden state h_1, we initiate h_0 as a vector of zeros of length H. The matrix W_{hh} is the holder of the recurrent connection going from the hidden state to the next one. Since the hidden layer has length H, but the input and output vectors are now just scalars, the matrices W_{xh} and W_{hy} are vectors of length N and M (although they be still referenced as "matrices"). The matrices do

[1] Instead of matrices we can sum all the multiplications of the weights with corresponding input neurons flowing to the neuron in the hidden layer. For input i going to hidden neuron j and the weight ij the value of the hidden neuron j (before the activation) would be $\sum_{j=1}^{k=H} \sum_{i=1}^{n=N} w_{ij}x_i$, but this operation is actually a matrix multiplication.

[2] it allows moving the threshold of activation function

[3] x_t can be even vector, but here we use just scalar for simplicity.

not change during the time t. In RNNs, the hyperbolic tangent is usually used, applied element-wise as the activation function σ_h. The choice of an activation function σ_y depends on the selected model. See the basic RNN in Equation 2.4.

$$\begin{aligned} h_t &= \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\ \hat{y}_t &= \sigma_y(W_{hy}h_t + b_y) \end{aligned} \tag{2.4}$$

2.2.2 Backpropagation

In order to train a network, total error first needs to be computed. For that an objective function L with parameters W and b representing all the weights and biases of the network is used. The type of the objective function differs by the type of a problem defined in Section 2.1. Since the result of the objective function is an error, we target to minimise it. For such task the Gradient Descent Algorithm is used.
"The backward pass starts by computing $\frac{\partial L}{\partial \hat{y}}$ for each of the output units."[56]. Consequently, the chain rule going all the way to the target weight matrix or bias is applied. Then the parameters subtracting the gradient are updated. Usually the gradient is multiplied with a hyperparameter η called the learning rate. Learning rate controls the convergence of the algorithm – the lower it is, the longer the training usually takes. A too high learning rate can cause the gradient to "overshoot" the minimum, or even diverge [47]. For the FNN defined in Equation 2.3 the gradients and the updates are shown in Equations 2.5 to 2.12[4]. For simplicity, we neglect dimensions and will not write the transpose signs.

$$\begin{aligned} \frac{\partial L}{\partial W_{hy}} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_{hy}} \\ \frac{\partial L}{\partial W_{hy}} &= \frac{\partial L}{\partial \hat{y}} \circ \sigma_y'(W_{hy}h + b_y)h \end{aligned} \tag{2.5}$$

$$W_{hy} := W_{hy} - \eta \cdot \frac{\partial L}{\partial W_{hy}} \tag{2.6}$$

$$\begin{aligned} \frac{\partial L}{\partial b_y} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_y} \\ \frac{\partial L}{\partial b_y} &= \frac{\partial L}{\partial \hat{y}} \circ \sigma_y'(W_{hy}h + b_y) \end{aligned} \tag{2.7}$$

[4]For element-wise multiplication we use the Hadamard product with symbol $\circ$, the matrix multiplication does not have an operator, the $\cdot$ operator symbolises the multiplication of an integer with a matrix or a vector or multiplication of two integers.

$$b_y := b_y - \eta \cdot \frac{\partial L}{\partial b_y} \tag{2.8}$$

$$\begin{aligned} \frac{\partial L}{\partial W_{xh}} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial W_{xh}} \\ \frac{\partial L}{\partial W_{xh}} &= \frac{\partial L}{\partial \hat{y}} \circ \sigma_y'(W_{hy}h + b_y) W_{hy} \circ \sigma_h'(W_{xh}x + b_h)x \end{aligned} \tag{2.9}$$

$$W_{hy} := W_{xh} - \eta \cdot \frac{\partial L}{\partial W_{xh}} \tag{2.10}$$

$$\begin{aligned} \frac{\partial L}{\partial b_h} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial b_h} \\ \frac{\partial L}{\partial b_h} &= \frac{\partial L}{\partial \hat{y}} \circ \sigma_y'(W_{hy}h + b_y) W_{hy} \circ \sigma_h'(W_{xh}x + b_h) \end{aligned} \tag{2.11}$$

$$b_h := b_h - \eta \cdot \frac{\partial L}{\partial b_h} \tag{2.12}$$

With time included in RNNs, the backpropagation algorithm slightly changes and is called the Backpropagation Through Time (BPTT). "BPTT unrolls the recurrent neural network and propagates the error backward over the entire input sequence, one time step at a time. The weights are then updated with the accumulated gradients." [10] The following equations are constructed for the model many-to-many where $N = M$.

$$\begin{aligned} \frac{\partial L}{\partial W_{hy}} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W_{hy}} \\ \frac{\partial L}{\partial W_{hy}} &= \sum_{t=1}^{M} [\frac{\partial L_t}{\partial \hat{y}_t} \circ \sigma_y'(W_{hy}h_t + b_y)h_t] \end{aligned} \tag{2.13}$$

$$W_{hy} := W_{hy} - \eta \cdot \frac{\partial L}{\partial W_{hy}} \tag{2.14}$$

$$\begin{aligned} \frac{\partial L}{\partial b_y} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b_y} \\ \frac{\partial L}{\partial b_y} &= \sum_{t=1}^{M} [\frac{\partial L_t}{\partial \hat{y}_t} \circ \sigma_y'(W_{hy}h_t + b_y)] \end{aligned} \tag{2.15}$$

$$b_y := b_y - \eta \cdot \frac{\partial L}{\partial b_y} \tag{2.16}$$

$$\begin{aligned}\frac{\partial L}{\partial W_{hh}} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_M} \prod_{t=1}^{M} (\frac{\partial h_t}{\partial W_{hh}} \frac{\partial h_t}{\partial h_{t-1}}) \\ \frac{\partial L}{\partial W_{hh}} &= \sum_{k=0}^{M} \frac{\partial L_M}{\partial \hat{y_M}} \circ \sigma_y'(W_{hy} h_M + b_y) W_{hy} \prod_{t=k+1}^{M} [(\mathbb{1} - h_t^2) W_{hh}]\end{aligned} \tag{2.17}$$

$$W_{hh} := W_{hh} - \eta \cdot \frac{\partial L}{\partial W_{hh}} \tag{2.18}$$

$\mathbb{1}$ is the vector of ones of length H.

$$\begin{aligned}\frac{\partial L}{\partial b_h} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_M} \prod_{t=1}^{M} (\frac{\partial h_t}{\partial b_h} \frac{\partial h_t}{\partial h_{t-1}}) \\ \frac{\partial L}{\partial b_h} &= \sum_{k=0}^{M} \frac{\partial L_t}{\partial \hat{y_t}} \circ \sigma_y'(W_{hy} h_M + b_y) W_{hy} \prod_{t=k+1}^{M} [(\mathbb{1} - h_t^2)]\end{aligned} \tag{2.19}$$

$$b_h := b_h - \eta \cdot \frac{\partial L}{\partial b_h} \tag{2.20}$$

$$\begin{aligned}\frac{\partial L}{\partial W_{xh}} &= \frac{\partial L}{\partial \hat{y}} \circ \frac{\partial \hat{y}}{\partial h_M} \circ \prod_{t=1}^{N} (\frac{\partial h_t}{\partial W_{xh}} \frac{\partial h_t}{\partial h_{t-1}}) \\ \frac{\partial L}{\partial W_{xh}} &= \frac{\partial L_t}{\partial \hat{y_t}} \circ \sigma_y'(W_{hy} h_M + b_y) W_{hy} \circ \prod_{t=1}^{N} [W_{hh}^{N-t} (\mathbb{1} - h_t^2)^T x_t]\end{aligned} \tag{2.21}$$

$$W_{xh} := W_{xh} - \eta \cdot \frac{\partial L}{\partial W_{xh}} \tag{2.22}$$

2.3 Long-Term Dependencies

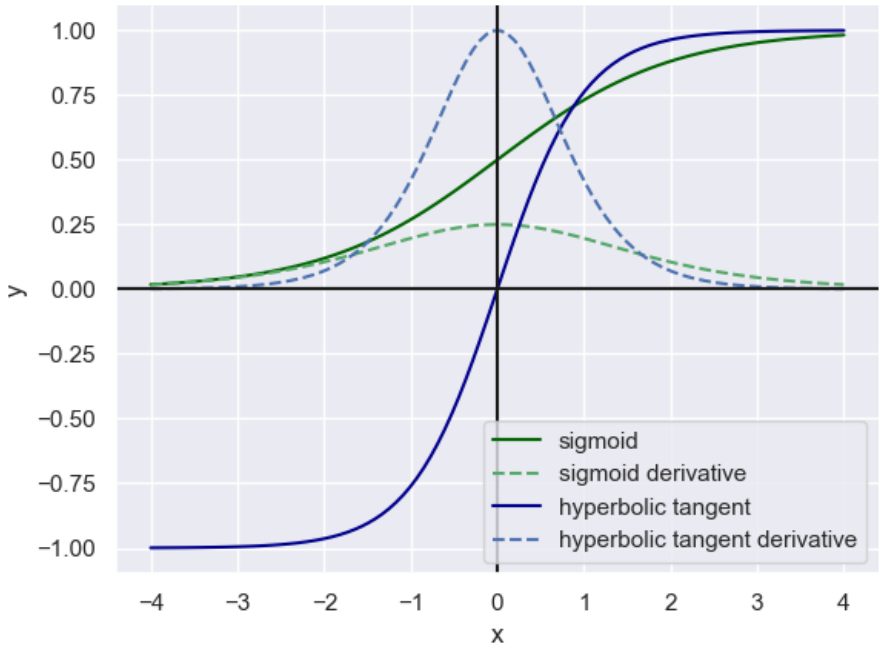
2.3.1 Vanishing gradients

As mentioned earlier, to be able to learn and model more complicated types of data, such as images, audio or video, non-linear activation functions are used. Two of the most used are the sigmoid function and the hyperbolic tangent function. See the functions and their derivatives in Equations 2.23 and 2.24.

$$\begin{aligned}\sigma(x) &= \frac{1}{1 + e^{-x}} \\ \frac{d\sigma(x)}{dx} &= \sigma(x)(1 - \sigma(x))\end{aligned} \tag{2.23}$$

$$\begin{aligned} \tanh(x) &= \frac{sinh(x)}{cosh(x)} = \frac{e^{2x}-1}{e^{2x}+1} \\ \frac{dtanh(x)}{dx} &= 1 - tanh^2(x) \end{aligned} \tag{2.24}$$

Figure 2.6: Sigmoid and hyperbolic tangent functions (graphs.py)



Sigmoid function compresses the input data into interval $(0, 1)$, which makes it useful especially for models with probability as an output. The range of the hyperbolic tangent function is in the interval $(-1, 1)$ and the curve is also so-called S-shaped. Both of the functions and their derivatives are displayed on the graph in Figure 2.6 (on the interval $[-4, 4]$). Notice that as $|x|$ increases, the result of both derivatives gets close to zero. If we have a network with n hidden layers this can cause the gradient to be too small for the network to learn because, by using the chain rule during backpropagation, these activations close to zero are multiplied, so the gradient decreases exponentially. This situation is called the Vanishing Gradient Problem and was first discovered by Sepp Hochreiter in 1991. It may seem that a fine solution for this problem is to change the activation

function or to have a shallower network. With RNNs this is sometimes not possible.

The computation is more complicated with time included, as can be seen by comparing Equation 2.9 with Equation 2.21, because with the partial derivatives $\frac{\partial h_t}{\partial h_{t-1}}$ we get the product of N Jacobian matrices. The distance between the next predicted output and the relevant information can be wide, so to have accurate predictions, we need to hold the information in the hidden states h_t for plenty of time steps, which means N can be a large integer. "This kind of dependence between sequence data is called the long-term dependencies" [3], "because the computation of output at time t depends on input presented at an earlier time $\tau \ll t$" [6].
Since RNNs use the hyperbolic tangent as the activation function, during training these models suffer from vanishing gradients, especially in the earlier layers [50]. Consequently, because there is a problem at the very beginning of the network, the result built up by the whole model is not accurate.

There are a few methods to avoid the Vanishing Gradients Problem, such as:

- initialise weights so that the potential for vanishing gradient is minimised;
- randomly drop most of the weights in W_{hh} (called the Echo State Network);
- Long Short-Term Memory Networks (LSTMs).
- batch normalisation

The most frequently used solution is the LSTM network defined in Section 3. Next, another gradient problem is defined.

2.3.2 Exploding gradients

In the same way that a gradient can shrink to zero, it can explode to infinity. Another proposed solution to the Vanishing Gradient Problem is to change the activation function. Currently, a very popular activation function is ReLU

– Rectified Linear Unit, first introduced by Hahnloser et al. in 2000 [31].

$$\begin{aligned} ReLU(x) &= max(0, x) \\ \frac{dReLU(x)}{dx} &= \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \end{aligned} \tag{2.25}$$

ReLU does solve the vanishing gradient in most cases, unless it has all the values as negative. To address this problem, the function was modified and Leaky ReLU (and other modifications) was introduced. In practice, many developers prefer ordinary ReLU, avoiding the vanishing gradient by setting the learning rate properly. This activation function is very popular also for faster convergence than the sigmoid or hyperbolic tangent function and being computationally efficient. ReLU can also be used as a classification function [1].
In view of the fact that the derivative of ReLU can be 1, with large weight initialisation or identity activation function[5] (which cannot be avoided in some problems), the norm of the gradients can become very large and overflow its number implementation in given programming language. The weights' components result in NaN (Not a Number) values and can no longer be updated [8]. This problem is called the Exploding Gradient Problem introduced by Bengio et al. in 1994, further explained in [51].

This problem may be solved by:

- use of Gradient Clipping (described in Section 4.8);
- use of the Truncated Backpropagation Through Time[6];
- use of weight regularisation (L1 or L2 penalty on the recurrent weights);
- data normalisation;
- use of Long Short-Term Memory networks (LSTMs).

Section 3 explains LSTMs in detail.

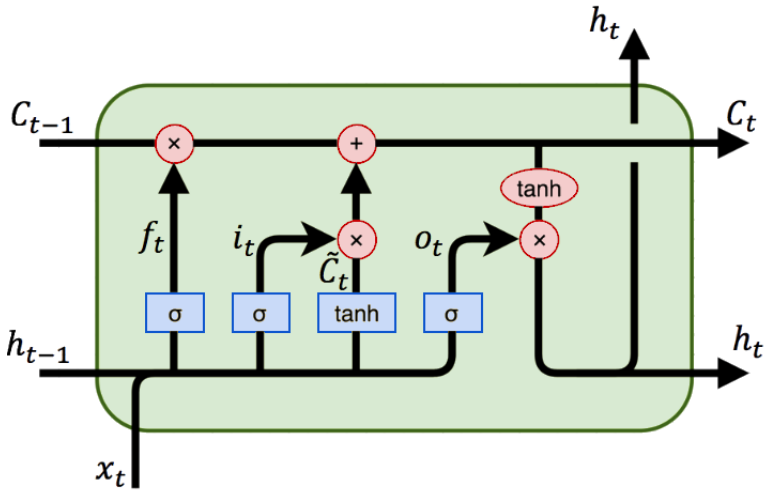
[5] $f(x) = x$

[6] this method modifies BPTT dropping the gradients in some defined first time steps – the forward propagation is the same, but the weights are not updated by these gradients [10]

3 Long Short-Term Memory

Long Short-Term Memory networks (first introduced in 1997 by Hochreiter and Schmidhuber [33]) imitate the way that the human brain processes sequential data. A good example is reading a text – to be able to remember valid information, we forget redundant parts of the text. While RNNs have problems with long-term dependencies, LSTM units solve this issue with additional features.

Figure 3.1: LSTM cell [19]



Although the article written by Christopher Colah [19] contains more of a summary of [33], it sheds light on the whole topic of LSTMs. The notations are very clear and Colah assists the understanding with detailed visualisations, thus a lot of authors cite his blog. This is the reason why the notations here are used from Colah rather than from the original paper.

The basic unit is called the memory cell, where its activation C_t is called the cell state. This variable holds the information of the previous inputs. The input gate unit i_t regulates the flow of the input data into the memory cell, learning what is relevant information for the correct prediction. "Output

gate unit o_t learns to protect other units from perturbation by currently irrelevant memory contents stored in the memory cell. The gates learn to open and close access to constant error flow."[33] The model is described by the system of equations below. At first the input gate decides which values will be updated – it closes (activation close to zero) for the irrelevant ones.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{3.1}$$

The square brackets in Equation 3.1 signify concatenation of the vectors. The activation of an input to the memory cell $\tilde{C}_t$ is then computed – these values are the candidates to be added to memory and propagated further to other time steps.

$$g(x) = \frac{4}{1 + e^{-x}} - 2 \tag{3.2}$$

The activation function g is the centred logistic function with range $[-2, 2]$.

$$\tilde{C}_t = g(W_C[h_{t-1}, x_t] + b_C) \tag{3.3}$$

The candidate values are then fed to the cell state, regulated by the input gate.

$$C_t = C_{t-1} + i_t \circ \tilde{C}_t \tag{3.4}$$

The hidden state[1] is then computed as the filtered version of the cell state, using the output gate as a filter.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{3.5}$$

$$f(x) = \frac{2}{1 + e^{-x}} - 1 \tag{3.6}$$

The activation function f of the cell state is again a centred logistic function, but this time with range $[-1, 1]$.

$$h_t = o_t \circ f(C_t) \tag{3.7}$$

Feeding a continuous input stream, the memory cells may grow and cause saturation of the activation function f in Equation 3.7. This makes the derivative of the activation function vanish, therefore the hidden state h_t outputs only the activation of the output gate. This means a complete loss of the memory and the equation is then equal to simple RNN, see Equation 2.4. Due to this situation, in 1999 the forget gate was presented [26]. The

[1] also known as the cell output

gate learns to reset the cell state by assigning a number between 0 and 1 to each value in it.

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{3.8}$$

Thus the Equation 3.4 is adjusted to 3.9.

$$C_t = f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \tag{3.9}$$

Instead of functions f and g – the centred logistic functions – most of the up-to-date sources use the hyperbolic tangent function. This is the case even in TensorFlow, the software library used for implementation in this thesis, hence the model used further is summarised in Equation 3.10, visualised in Figure 3.1.

$$\begin{aligned} f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \\ o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ \tilde{C}_t &= tanh(W_C[h_{t-1}, x_t] + b_C) \\ C_t &= f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \\ h_t &= o_t \circ tanh(C_t) \end{aligned} \tag{3.10}$$

The backpropagation of LSTM is described in Appendix.

3.1 LSTM Autoencoders

In Section 2, machine translation was mentioned as one of the applications of RNNs and will be used as a good example in this section. When using a sentence in one language as an input trying to produce a sentence in another language as accurately as possible, a context is needed. Usage of the LSTM model defined in Equation 3.10 with the structure many-to-many shown in Figure 2.1 would not be possible for this task. In Figure 3.2 the model does not consider the previously predicted word when trying to find the new one.

The problem of the second many-to-many model (Figure 3.3) is that this model outputs a sentence of the same length or a shorter sentence and does not take longer outputs into account.

Figure 3.2: Many-to-many [19]

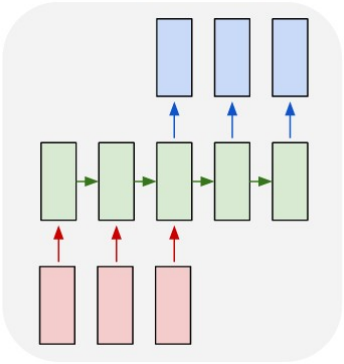
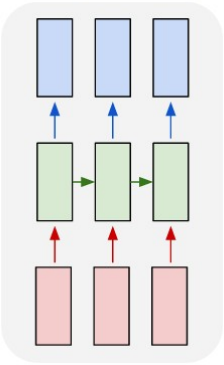


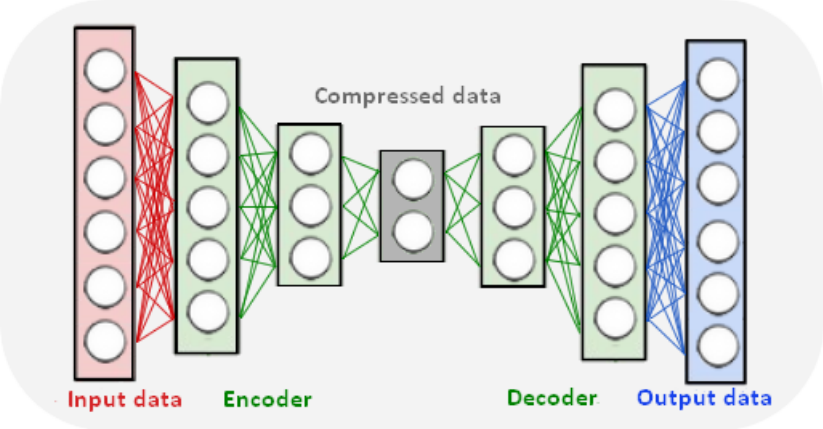
Figure 3.3: Many-to-many [19]



To deal with these issues, LSTMs were included in autoencoders.

Autoencoder is an ANN, the aim of which is to make a representation of the input vector (encoding) and then a reconstruction of the input vector from this representation. AEs are considered as unsupervised learning algorithms (because they do not use labelled data), but they use supervised learning methods – the input is used as the target. Hence the lengths of the input and output vectors are the same. Between input and output layers are one or more hidden layers with fewer neurons to reduce the size of the input – these layers are called the encoder, followed by the representation vector (the "bottleneck"). The flow continues to other hidden layers, called the decoder, and to the output layer. The structure is shown in Figure 3.4.

Figure 3.4: Autoencoder



See simple autoencoder model in Equation 3.11 with only one hidden layer. The weight matrix W' does not have to rely on the original matrix W, but can be the transpose of it. The activation functions and biases also do not need to be the same. The process of learning is similar as in Section 2.2.2, see [65].

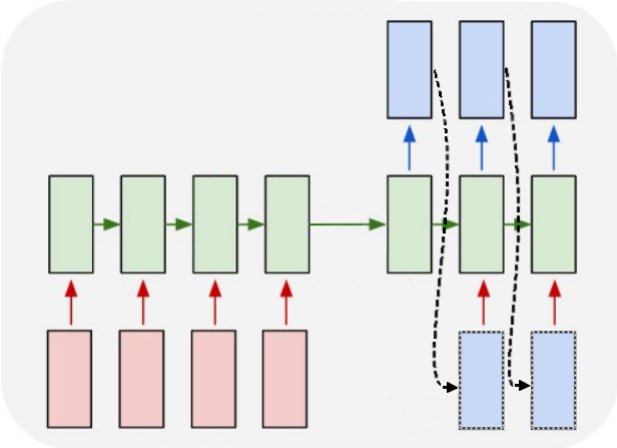
$$\begin{aligned} h &= \sigma_h(Wx + b) \\ x' &= \sigma'_h(W'h + b') \end{aligned} \tag{3.11}$$

Historically, autoencoders were used as a pre-training of ANNs [57]. Currently, they are typically used for dimension reduction, feature variation, watermark removal or image denoising. The amount of variations increases with the inclusion of different types of ANNs, such as RNNs.

Autoencoders are designed to work with inputs of fixed length, but in machine translation, the lengths of the sentences can vary. Use of LSTMs in autoencoders allows variable lengths not only of the input but also of the output, see Figure 3.5, having RNN as an encoder and another RNN as a decoder [17].

This is provided by adding the information of the length of the input and output sentence in the model. The model still expects a vector of definite length (the length of the longest sentence), thus a padding is added at the

Figure 3.5: LSTM Autoencoder



end for shorter sentences. In machine translation, special characters are used for the network to know when the sentence begins or ends. The characters[2] are usually:

- <GO> – the symbol at the beginning of the decoder, sometimes also used <START> or <NULL> (in the example in Figure 3.6);
- <PAD> – the token used at the end of a sentence to fulfil the network's expectation;
- <UNK> – character which replaces unknown words, i.e. rare words that are not in the dictionary;
- <EOS> – the end-of-sentence symbol[3] [68].

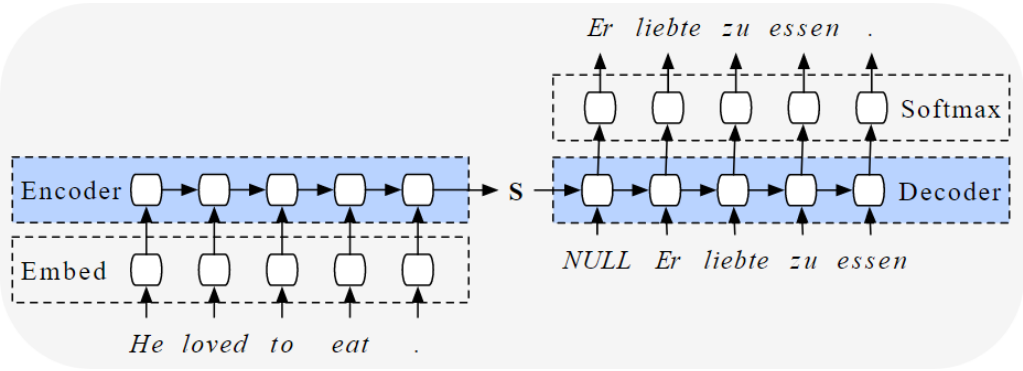
See Figure 3.6 with an example of an English-to-German translator, where the symbol S illustrates how the cell state and the hidden state from the encoder are fed to the decoder.

Since the model has a sequence as input and output, it is also called Seq2Seq (sequence-to-sequence) model. For words to be fed into the network, they need to be converted to numbers. The preprocessing is performed at the

[2]interchangeable with tokens or symbols

[3]some developers use the full stop symbol, depending on the data

Figure 3.6: Neural machine translation [45]



beginning of the process (like data cleansing, lowercasing, etc.), after which vocabularies of both the languages are created. The simplest conversion of words to integers is One-Hot Encoding. Here the vector representing one word has the same length as the number of words and punctuation marks in the vocabulary (plus the special characters defined before), where all positions are filled with zeros, except the index of the word in the vocabulary, where the value is one. A disadvantage of this encoding is that there is no information about the context or frequency. To avoid this problem, other methods are used, e.g. Word Embeddings (for further details see [40]). The crucial part of this conversion is that the task becomes a classification problem and cannot output other values than the ones defined in the vocabulary. The problem in this thesis, on the other hand, is a regression problem.

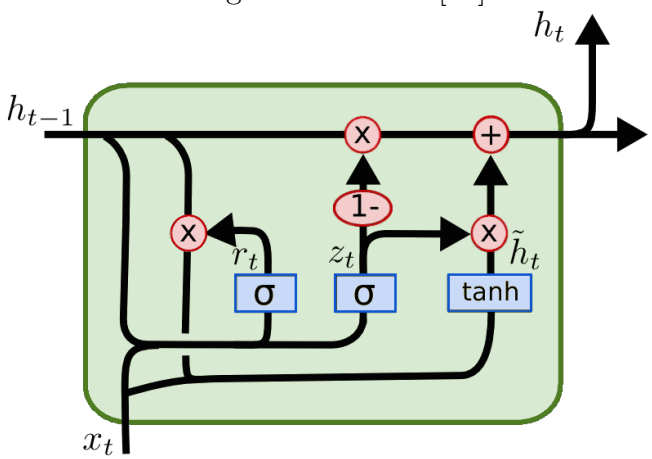
3.2 Gated Recurrent Unit

Despite the fact that in the machine-learning community the name LSTM Autoencoder is frequently used, Seq2Seq models do not only apply to LSTM units. There are other variants of LSTM units. To date, the most popular one is the Gated Recurrent Unit (GRU) by Cho, et al. [17]. The unit is inspired by the LSTM unit, merging the forget and input gate into an update gate z_t. This unit controls how much information from the previous hidden state is passed into the next hidden state, allowing long-term dependencies to be captured without the need to have the cell state. This means that the unit is simpler to compute, because there are fewer parameters. With regard to LSTMs, in GRU there are candidate values $\tilde{h}_t$ which update the hidden state. To control henceforth redundant information, there is a second gate, the reset gate r_t which decides how much of the previous hidden state

should be forgotten. Values close to zero force the candidate values to ignore the previous hidden state h_{t-1}, learning short-term dependencies. The GRU model is defined in Equation 3.12, where the notations are again taken from Colah[4] [19].

$$\begin{aligned} r_t &= \sigma(W_r[h_{t-1}, x_t] + b_r) \\ z_t &= \sigma(W_z[h_{t-1}, x_t] + b_z) \\ \tilde{h}_t &= tanh(W_h[r_t \circ h_{t-1}, x_t] + b_h) \\ h_t &= z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t \end{aligned} \tag{3.12}$$

Figure 3.7: GRU [19]



[4]Biases were added in the equation and z was interchanged with $(1 - z)$ to match the original paper [17]. The activation functions are the hyperbolic tangent and sigmoid as for the LSTM cell and also the default setting of TensorFlow function applied during implementation.

4 Hyperparameters

Hyperparameters are mentioned at the end of Section 3. While parameters are the variables required to estimate using an optimisation algorithm and are updated during training, as mentioned in Sections 2.2 and 3 (weights and biases), hyperparameters are not directly estimated from the data. They are external to the model and set before the learning phase starts, although some of them can be tuned during training. The settings of hyperparameters have a great impact on if and how the model learns and what the predictions are. In this section, the hyperparameters that were used in the model defined in Section 9, are introduced:

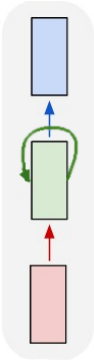
- number of hidden layers and number of hidden neurons,
- activation function,
- learning rate,
- objective function,
- optimiser,
- batch size,
- train ratio,
- gradient clipping,
- number of epochs.

4.1 Hidden layers and neurons

The first hyperparameter taken into consideration is usually a hidden layer, or more precisely the number of hidden layers and the number of neurons in each hidden layer. This hyperparameter depends on the model and type of ANN used, e.g. for image classification more layers can mean more recognised features, but too many layers can cause the Vanishing/Exploding Gradient Problem defined in Section 2.3. The number of hidden layers and hidden

neurons in the layers depends on the programmer and the dimension of the input data. For RNNs there is an inconsistency in understanding what a hidden layer is. Some sources claim that the model has the same amount of hidden layers as the number of time steps. This idea makes sense w.r.t. optimisation, because a large number of time steps (hidden layers in this case) means a long computational time. On the other hand, taking into consideration that the hidden layers usually do not share parameters (see not unrolled RNN in Figure 4.1), in this thesis it is assumed that the RNN model defined in Equation 2.4 has only one hidden layer.

Figure 4.1: Not unrolled RNN based on [19]



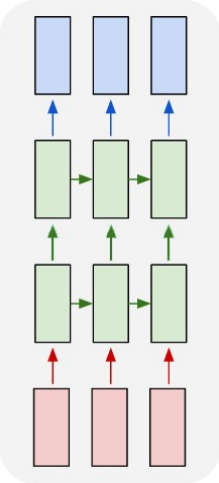
In order to have more hidden layers, another hidden layer can be stacked on the first one, e.g. in Equation 4.2. This RNN is called the Stacked RNN or Multi-cell RNN and its possible structure is shown in Figure 4.2. Stacked LSTMs work on the same principle.

$$\begin{aligned} h_t &= \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\ j_t &= \tanh(W_{hj}h_t + W_{jj}j_{t-1} + b_j) \\ \hat{y}_t &= \sigma_y(W_{jy}j_t + b_y) \end{aligned} \tag{4.1}$$

4.2 Activation function

Activation functions were already mentioned in Sections 2.2 and 2.3. These functions are considered to be hyperparameters, because they are specified

Figure 4.2: Stacked RNN - 2 hidden layers



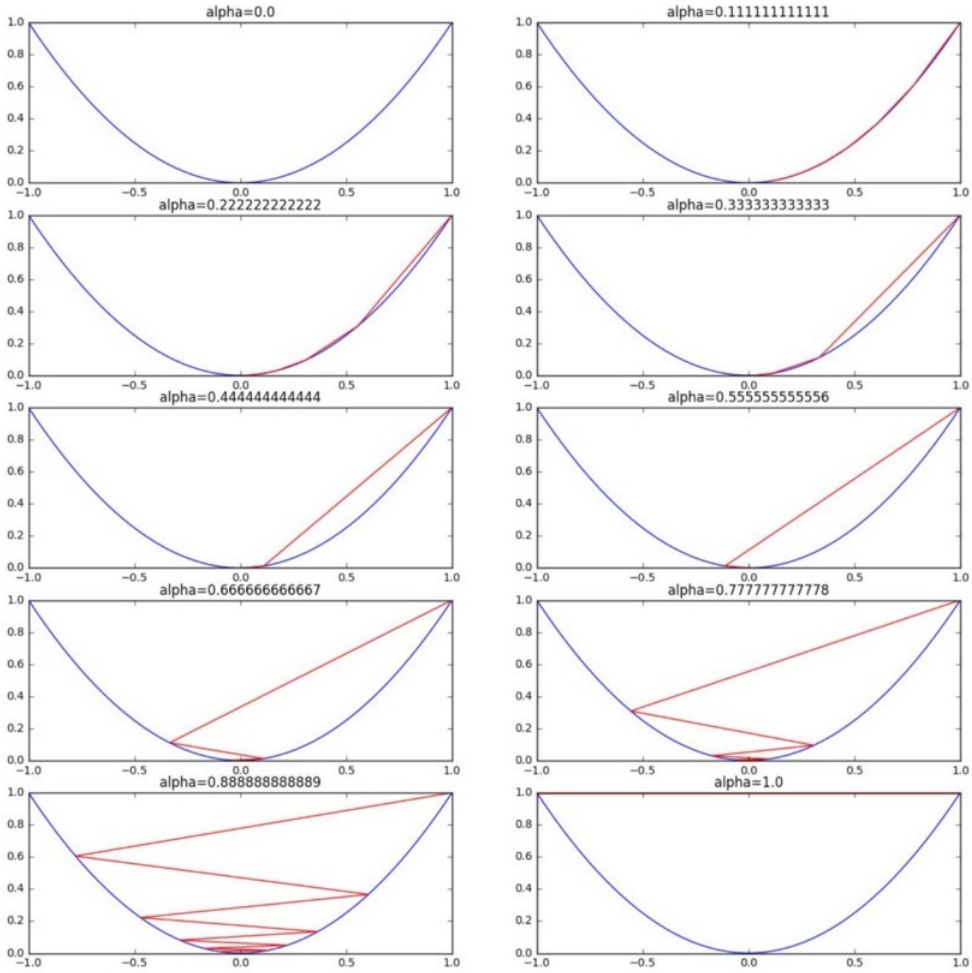
beforehand by the programmer. The most popular activation functions are the hyperbolic tangent function (Equation 2.24), the sigmoid function (Equation 2.23) and the rectified linear units function (Equation 2.25). For classification problems, the softmax function is also a very popular activation function (see [29]). This function is usually used between the last hidden layer and the output layer, because it squashes the results into the interval $(0, 1)$, where the sum of these values is one, so they can be interpreted as probabilities. However, in this thesis, a regression problem is solved and a prediction of the next real value is needed, not the probability of the input being classified in a class. There will be no activation function (also known as identity activation function) in the output layer, see Equation 4.2.

$$\begin{aligned} f(x) &= x \\ \frac{f(x)}{dx} &= 1 \end{aligned} \tag{4.2}$$

4.3 Learning rate

Another hyperparameter is also the learning rate, already defined in Section 2.2.2. Usually the value is in the interval $(0, 1)$. A too high learning rate can cause overshooting of the minimum and divergence, whereas a too low learning rate may not guarantee the minimum being reached after the given iterations. The behaviour of learning, having simple quadratic function ($y = x^2$) with different learning rates starting from point $[1, 1]$, is shown in Figure 4.3, where the learning rate is denoted by *alpha*.

Figure 4.3: Visualisation with different learning rate [5]



4.4 Objective function

Objective function is a function we want to maximise or minimise[1] and it serves as the evaluation of how well the model fits the data. If we have a labelled data, it is required to minimise the difference between the model's prediction and the target value. The parameters of the objective function are all the learnable parameters of the model. Having the objective function L, where θ is the set of all the weights and biases in the model, the aim is to

[1]When maximising the objective function, it is called a reward, when minimising, it is usually called a loss, cost or error function.

find these weights and biases, so that the condition of Equation 4.3 is met [32].

$$arg\min_{\theta}(L(\theta)) \tag{4.3}$$

The result of an objective function is a scalar value, which in minimising optimisation problems is called a loss or an error and the target is to reduce it. To find if the model learns, it is best to display the loss every iteration, and overall this value should decrease.

When dealing with objective functions in a supervised learning algorithm, they can be divided into two groups, regression losses and classification losses, and should be chosen with respect to the problem as well as to the data. We face a regression problem, having time series as an input. The measures of estimators' quality for these types of problems are SSE (Sum of Squared Errors) 4.4, MSE (Mean Squarred Error) 4.5 and MAD[2] (Mean Absolute Deviation) 4.6.

$$SSE = \sum_{t=1}^{N}(y_t - \hat{y}_t)^2 \tag{4.4}$$

$$MSE = \frac{1}{N}\sum_{t=1}^{N}(y_t - \hat{y}_t)^2 \tag{4.5}$$

$$MAD = \frac{1}{N}\sum_{t=1}^{N}|y_t - \hat{y}_t| \tag{4.6}$$

Because we want to know how far on average our predicted values are, we prefer MSE and MAD. MSE penalises the outliers more than MAD because of the square, so the objective function in our model will be MSE.

4.5 Optimiser

To correctly update the parameters, an optimisation algorithm is used. The Gradient Descent algorithm was mentioned in Section 2.2.2. Gradient Descent[3] and its optimisations are the most common method of numerically optimising neural networks. "Gradient descent is a way to minimise objective function $L(\theta)$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta L(\theta)$ w.r.t. to the parameters. The

[2]also known as MAE (Mean Absolute Error)

[3]also known as Standard Gradient Descent, Batch Gradient Descent or Vanilla Gradient Descent

learning rate η determines the size of the steps we take to reach a (local) minimum." [55] Gradient Descent (GD) takes the whole training dataset, computes the sum of accumulated errors and then updates the parameters, see Equation 4.7 where $L(\theta) = \sum_{i=1}^{K} L_i(\theta)$ and K is the number of training examples. Stochastic Gradient Descent (SGD), on the other hand takes training examples randomly and updates the parameters after each iteration, see Equation 4.8, where i is the i-th example. This causes noisier error, but also faster convergence, especially, when the training dataset is huge or has very similar examples.

$$\Delta\theta = \theta - \eta \cdot \nabla_\theta L(\theta) \tag{4.7}$$

$$\Delta\theta = \theta - \eta \cdot \nabla_\theta L_i(\theta) \tag{4.8}$$

Mini-batch Gradient Descent is a compromise of GD and SGD, because it updates parameters after feeding a subset of the training data, the size of which (in Equation 4.9 denoted by B) is given in advance. To prevent cycles, these examples are chosen randomly. The convergence is usually smoother than for SGD, because the gradient used for the update is a sum of the gradients computed for each example in the mini batch.

$$\Delta\theta = \theta - \eta \cdot \nabla_\theta L_{i:i+B}(\theta) \tag{4.9}$$

Nevertheless, both SGD and Mini-batch Gradient Descent are sensitive to the learning rate. A bad setting can cause fluctuations around minimum or even divergence. Because their gradients are only approximations of the true gradient, they do not guarantee that in each iteration the error will decrease. To improve some of the features, optimisers have been presented which modify the GD algorithm.

Adam (Adaptive Moment Estimation) is an optimiser that enhances its predecessors. Like AdaGrad, it changes the learning rate for each parameter, performing larger updates for infrequently occurring features and smaller updates for more frequent ones. Just like AdaDelta it prevents a decay of the learning rate which could cause the model to stop learning. Moreover, Adam stores momentum changes for each parameter separately. We calculate the first moment (the mean) m_t and the second moment (the uncentred variance) v_t (initialised as vectors of zeros) of the gradients g_t and use these values to update the parameters. There are new variables – exponential decay rates for the moment estimates β_1 and $\beta_2 \in [0, 1)$. One iteration is shown in Equation

4.10.

$$\begin{aligned} g_t &= \nabla_\theta L_t(\theta_{t-1}) \\ m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\ v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\ \hat{m}_t &= \frac{m_t}{(1 - \beta_1^t)} \\ \hat{v}_t &= \frac{v_t}{(1 - \beta_2^t)} \\ \theta_t &= \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned} \tag{4.10}$$

g_t^2 represents element-wise square $g_t \circ g_t$ and all operations on vectors are element-wise. The variable ϵ is to avoid dividing by zero and it is taken over from AdaGrad. Unlike GD, Adam is very good for sparse data; it converges very fast and is overall a good choice for deep networks [55].

4.6 Batch size

Although in Section 4.5 the term "batch" was used for the whole dataset and for its subset, the term "mini-batch" was used, in the deep learning community the terms "batch" and "batch size" are commonly used to mean "mini-batch". Batch size is the number of samples fed to the model during one iteration, in Section 4.5 denoted as B[4]. Using the Mini-batch Gradient Descent algorithm, or other algorithms using this approach, in one iteration, the parameters are updated only once. Batch size can radically affect how quickly the model converges as well as the results. Large batch size can cause the model's ability to generalise being lost. To have the fastest learning, the samples should be shuffled, i.e. randomly chosen. The more dissimilar the examples in one batch are, the faster the model learns [41]. The suggested batch size is 32, 64 or 128, but it largely depends on the number of examples in the training data.

4.7 Train ratio

Another hyperparameter that can be defined is the ratio of the training data to the whole dataset. In some cases, the data are split beforehand, but with

[4]if $B = 1$ we have SGD, if $B = K$ the algorithm is GD

a time series, usually a whole dataset is obtained. In this situation it depends on the target of building the model and it is defined by the researcher without using any hyperparameter optimization. Most deep learning papers suggest 70% of training and 30% of test data, sometimes 80% to 20% or also 90% to 10%. However, less than 70% is rarely used for training data. In case of having validation data, the training data ratio is decreased by 10% of the whole dataset.

4.8 Gradient Clipping

In Section 2.3.2 Gradient Clipping is mentioned as a solution of exploding gradients. Despite being claimed in [34] that, that LSTM solves problems with exploding gradients, in practice it is not always true. To provide the numerical stability of training deep neural network models, a method called Gradient Clipping was presented.
This method rescales the norm of a gradient when it exceeds a threshold. This gives us a new hyperparameter – the threshold ξ. The simple algorithm is shown in Equation 4.11, where $\|g\|$ is the Euclidean norm[5] of the gradient [50].

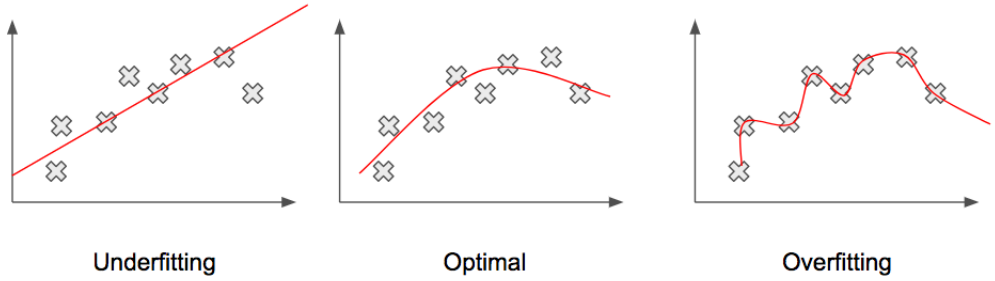
$$\begin{aligned} g &= \nabla_\theta L(\theta) \\ g &= \begin{cases} g & \|g\| < \xi \\ \frac{\xi}{\|g\|} g & \|g\| \geq \xi \end{cases} \end{aligned} \tag{4.11}$$

4.9 Number of epochs

In one epoch, the whole training dataset is passed forward and backpropagated through the model only once, whereas in one iteration, B observations (one mini-batch) are fed to the network. Defining how many epochs are the correct amount so that the model fits the data well, but is not overfitted, can be challenging. The model is underfitted when it does not capture the relationship between the independent (x) and dependent (y) variables, with the possible reason being that it is too simple. The opposite is overfitting, when the model is too fitted on the training data that it is unable to generalise to the test data, which leads to making poor predictions, see Figure 4.4. This may be due to a too long training.

[5]also known as L2 norm

Figure 4.4: Underfitting and overfitting [22]



One solution could be performing several trials with the same model and hyperparameters, changing only the number of epochs, selecting the amount that results in the best performance. A disadvantage of this approach is the long time taken to try all the models and computational inefficiency.
An alternative is a method called Early Stopping. In this method, the number of epochs is set to a large number and the parameters stored for every step. Thereafter, the errors of the training and the validation dataset for each iteration are compared, which are supposed to decrease. When the error of the validation dataset increases, this is the sign that the training should stop. There is also a patience parameter which signifies a number of iterations to observe the worsening validation set error before ending the training, because the error can fluctuate. After stopping the training, the stored parameters are used before increasing the validation error to obtain better performance [29].

4.10 Hyperparameter optimisation

Before the first data feed, the hyperparameters need to be set. Selection of the proper configuration to result in quick convergence of the algorithm, can be challenging. There are four main methods used for hyperparameter tuning [30].

4.10.1 Grad Student Descent

Grad Student Descent also known as the Trial and Error method provides manual adjustments of the hyperparameters. The model is launched several times with different hyperparameters. The researcher explores how these changes affect the results. For this method it is crucial to visualise multiple different quantities changing over each epoch, such as the error, accuracy (for classification problems), or distribution of the parameters. The method itself makes a significant contribution to the understanding of the relationships between parameters and hyperparameters, but can be very time consuming and with different models, the dependencies can be slightly different.

4.10.2 Grid Search

Another method is Grid Search which combines manual setting with an automation. This is a naive approach, where the user defines a grid of dimensions corresponding to the searched hyperparameters. For each dimension, a range of possible values is manually set. The algorithm "tries" all the possible configurations to minimise the objective function (or maximise the accuracy). The configuration with the best performance is then used. In order to reduce the number of configurations, fewer hidden layers or some rules of thumb can be used. This method has a high computational cost and can be also time consuming.

4.10.3 Random Search

Random Search varies from Grid Search in the setting. A user has to define the range of possible values, but the algorithm chooses the combination of hyperparameters randomly, resulting in wider exploration of the configuration space. This means that more possible values are tried in fewer iterations, whereas in one iteration of the algorithm, one configuration of hyperparameters is passed. A rule of thumb is to prefer Random Search over Grid Search when the configuration space has more than 4 dimensions.

4.10.4 Bayesian Optimisation

Neither Grid Search nor Random Search take into consideration the computation history – the picks of the configurations are independent of each other. Despite the long time required, when applying the Grand Student Descent method, the researcher slowly modifies the values of hyperparameters so that in each epoch the error decreases. The aim of Bayesian Optimisation is to simulate the researcher's job by the use of machine learning.

The method constructs a Gaussian Process[6] describing the function wanted to be minimised. The basic assumption of the process is that similar inputs have similar outputs. The algorithm learns the scales of each dimension for measuring similarity, according to the different hyperparameter values and the results w.r.t. these hyperparameters. Apart from the outputs, Gaussian Processes also predict the expected value of the result and the expected variance – (prior) probability distribution.

With all available data (mini-batch) and initialisation of the hyperparameters, the algorithm finds the posterior probability distribution over possible functions. In the next step, the algorithm computes the acquisition function using the current posterior distribution, that determines the next picked combination of hyperparameters in order to maximise the expected improvement. If the result is better (lower error), the algorithm will save this setting and next it looks for hyperparameters that maximise the expected improvement w.r.t. this configuration. Otherwise it continues with the last kept setting. Finally, the algorithm updates the prior distribution function w.r.t. to the current best configuration and new observed data and repeats the whole process until the number of specified iteration is exceeded. For further maths-related details of the algorithm, refer to [61].

[6]"Gaussian distribution is a probability distribution over possible functions." [39]

5 Time series forecasting

Time series are chronologically ordered observations x_t recorded at a specific time t. If the set of time steps is T where $t \in T$ is discrete, this is called a discrete time series whereas if the observations are recorded continuously over some time interval, the time series is continuous [7]. The target of time series analysis is usually the construction of a model and fitting it to the observations in order to study dependencies in data. The aim is to understand the mechanism of how the observations are generated, find patterns and predict further development of observed variables [18].
Time series can be split into several components that represent the underlying pattern type – trend, seasonality, cycles and remaining component.

- T_t: trend – the increasing or decreasing value,
- S_t: seasonality – the repeating short-term cycle with known frequency,
- C_t: cycles are also repeating, the frequency is not precise, usually longer than two years,
- R_t: remaining part captures everything else.

If we assume additive decomposition, by adding these components together, the original time series is obtained.

$$y_t = T_t + S_t + C_t + R_t \tag{5.1}$$

If the variation around the trend or magnitude of the seasonal fluctuations does not differ from the level (expected value) of the time series, additive decomposition is suitable. Otherwise multiplicative decomposition is more appropriate.

$$y_t = T_t \cdot S_t \cdot C_t \cdot R_t \tag{5.2}$$

In many decomposition methods, the cycle is combined with the trend.
This chapter summarises some classic time series evaluation and forecasting methods that were used to extend the model to be presented in Section 9 and for comparison. Most of the definitions were taken from [36].

5.1 Linear regression

Regression is a set of statistical methods used for analysis of the relationship among variables – dependent y and independent X. The aim is to explain y using a combination of parameters and X. If X is univariate (to be denoted as x) and a linear combination of parameters is used, this is a simple linear regression model. With independent variable x and dependent variable y, linear regression is a fitted line:

$$y = \beta_0 + \beta_1 x, \tag{5.3}$$

where β_0 is the intercept and β_1 the slope of the line. We do not assume, that the variable y depends only on the variable x, so the random variable ϵ is added, to capture all remaining influences. The noise variable is modelled to be a Gaussian noise, such that $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$. Having n observed pairs $(x_1, y_1), \ldots (x_n, y_n)$, the regression line has an equation of the form:

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i \qquad i = 1, \ldots, n. \tag{5.4}$$

If there are m independent variables, it is a multiple linear regression, see Equation 5.5, and the equation can be rewritten to the matrix form, see Equation 5.6.

$$y_i = \beta_0 + \beta_1 x_{i,1} + \cdots + \beta_m x_{i,m} + \epsilon_i \tag{5.5}$$

$$y = X\beta + \epsilon \tag{5.6}$$

The unknown parameters β can be estimated with an ordinary least squares (OLS) method, if the assumptions of the linear regression model in 5.1 are satisfied [42].

$$y = X\beta + \epsilon,$$

where

1. $E(\epsilon_i) = 0 \qquad \forall i = 1, 2, \ldots, n$ (the mean of the random variable is zero),
2. $D(\epsilon_i) = \sigma^2 \qquad \forall i = 1, 2, \ldots, n$ (there is no heteroscedasticity[1]),

[1] the variance of errors is constant

3. $Cov(\epsilon_i, \epsilon_j) = 0 \quad \forall i \neq j$ where $i, j = 1, 2, \ldots, n$ (there is no autocorrelation),
4. $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$ (the random variables have normal distribution),
5. X is deterministic and has full rank,
6. the linear regression model is linear in parameters.

(5.7)

Under these conditions when minimising an objective function $O(\beta)$ using OLS (Equation 5.8), interpretation of the model is possible and the results are statistically conclusive. For the Side-Channel Injection, we will not explore all of these conditions.

$$\begin{aligned} O(\beta) &= \|\boldsymbol{y} - \boldsymbol{X}\beta\|^2 \\ b &= arg\min_{\beta} O(\beta) \\ b &= (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \end{aligned} \tag{5.8}$$

b is the estimate of parameters β [36].

5.2 Naïve method

One of the simplest forecasting approaches is the naïve method, which sets all the forecasts to the value of the last observation.

$$\hat{y}_{t+h} = y_t \tag{5.9}$$

5.3 Seasonal naïve method

For highly seasonal data, the naïve method is slightly modified to fit the seasonal period.

$$\hat{y}_{t+h} = y_{t+h-m(k+1)} \tag{5.10}$$

m represents the number of seasons in a year, and $k = \lfloor \frac{h-1}{m} \rfloor$ is an integer that controls that for the forecast the last set of seasonal components from the observed points is used.

5.4 Forecast evaluation

To evaluate the (final) forecast, RMSE (Root Mean Square Error) 5.11 is used, because it is a scale-dependent measure, which means the error is in the same units as the predictions.

$$RMSE = \sqrt{\frac{1}{N}\sum_{t=1}^{N}(y_t - \hat{y}_t)^2} \tag{5.11}$$

5.5 Akaike's Information Criterion

Neither MSE nor RMSE take into consideration the number of parameters of a model. For this reason the best of the implemented models will be selected also w.r.t. Akaike's Information Criterion (AIC) [2]. AIC is an objective method of deciding among several statistical models. This criterion penalises not only models with worse predictions (underfitting), but also models with a large number of parameters (overfitting). The goodness of parameters is calculated by the expected log likelihood. Equation 5.12 will be used for comparing neural network models, where SSE is the Sum of Squared Errors ($SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$), which means MSE can be used as an argument of the natural logarithm. k is the number of estimated parameters (weights) in the model and n is the number of observations.

$$AIC = n \cdot ln(\frac{SSE}{n}) + 2k \tag{5.12}$$

If the ratio $\frac{n}{k}$ is less than 40, the adjusted model in Equation 5.13 is used [4], [49].

$$AIC = n \cdot ln(\frac{SSE}{n}) + 2k + \frac{2k(k+1)}{n-k-1} \tag{5.13}$$

5.6 Hypothesis Testing

It is possible to use Hypothesis Testing to test whether one model performs significantly better than the other. The selection of a statistical hypothesis test needs to be done w.r.t. the distribution of errors and other characteristics of observed data. For this reason, the Hypothesis Testing is further explained in Section 10.

6 Data

6.1 Data shape

AEs are considered as unsupervised learning algorithms using supervised learning methods (mentioned in Section 3.1). Having a time series such as daily temperature or stock prices, the data does not have any label. We use a "lag method" to create the target data. We define w observations that are used to predict our q predictions. We create a "window" of observations, where their amount is the width of the window. The window then slides to the future to create the targets, where the number of time steps q in decoder can vary from number of time steps in encoder w. In Section 3.1 Figure 3.2 shows 3 time steps as input followed by 3 time steps as output that are shifted by 3 time steps. This principle is modified to have the possibility of having different number of time steps of the decoder q from time steps of the encoder w with a condition $w \geq q$.

$$y_t = x_{t+w} \tag{6.1}$$

As mentioned in Section 4.6, the data samples should be shuffled. Applying this method to the time series data, would cause a loss of the temporal order. Instead of shuffling the whole dataset, and then performing the train-test-split, the data is first split into validation, training and test datasets and an observation is randomly chosen from the training set. Then this sample is taken as the first element in the sliding window and the lag method is performed as defined above. This will provide a random character of the training and prevent cycles, because the chosen observation will not have the same position in the window in every iteration [13].

6.2 Preprocessing

Unlike statistical techniques of forecasting, neural networks do not assume any underlying pattern of the data. Nevertheless, preprocessing of the data can (and should) be done to output the best forecast possible.
One method for supporting stable convergence, is scaling the data. Unscaled

inputs may result in slow convergence and unscaled targets in exploding gradients. A feature that has an extensively higher range than the others will influence the result more, neglecting the units. To have all the features of input in a comparable range, normalisation of ratings is used, namely Min-Max scaling, see Equation 6.2.

$$x_{scaled} = (b - a) \cdot \frac{x - min(x)}{max(x) - min(x)} + a \tag{6.2}$$

The function scales the range in $[a, b]$, where the scale is mostly used in the range $[-1, 1]$ or $[0, 1]$. Min-Max scaling gets the values closer to sample mean, which can suppress the effect of outliers. To properly scale the input data, firstly a train-test-split is performed, after which the training data are scaled and this scaling is used on the test data (and validation data). To reverse the data back to their original scale, Equation 6.3 is used.

$$x = \frac{x_{scaled} - a}{b - a} \cdot (max(x) - min(x)) + min(x) \tag{6.3}$$

Another common transformation of input data is standardisation, whereby the distribution of observation is scaled to have zero mean μ and standard deviation σ equal to 1.

$$x_{scaled} = \frac{x - \mu}{\sigma} \tag{6.4}$$

The equation is also referred to z-score. Here the standard deviation and mean are computed only from the training data, then applied to the test data. When splitting the data, it is obvious that only sample data are used, thus it is necessary to use sample mean $\bar{x}$ and sample standard deviation s, see Equation 6.5.

$$x_{scaled} = \frac{x - \bar{x}}{s} \tag{6.5}$$

When the data are reversed back to original scale, the scaled data are multiplied by the sample standard deviation and added to the sample mean.
A good rule of thumb is to use both of these methods, unless there are important outliers in the data, the impact of which should not be lost. The standardization is applied first and the normalization after, so that the training dataset is within defined scale.

6.3 Datasets

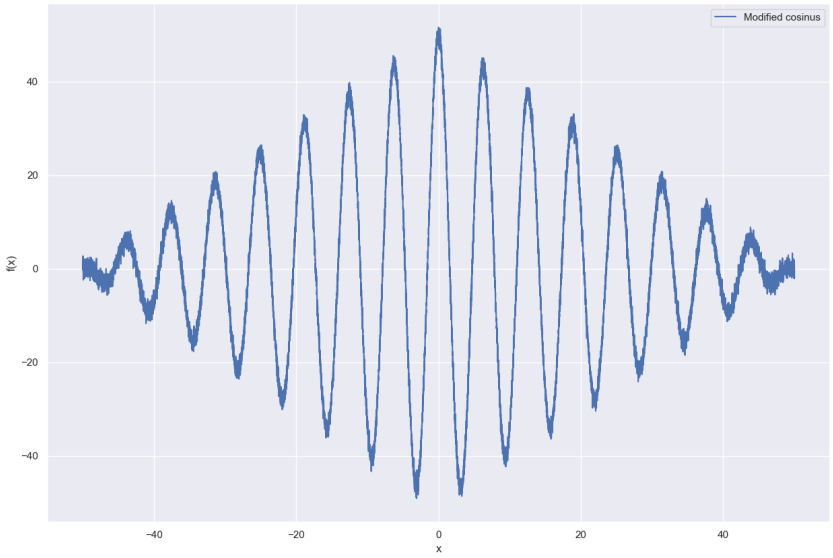
6.3.1 Generated data

First dataset is generated by the script *data_generator.py*. 15 000 samples are equally distributed over an interval $[-50, 50]$ and assigned to function $f(x)$ from Equation 6.6.

$$f(x) = (a - |x|) \cdot cos(x) + \epsilon \qquad x \in [-a, a] \tag{6.6}$$

Noise $\epsilon \sim \mathcal{N}(0, 1)$ is added to the modificated cosine function. The dataset is used as a basic validation that the model works and the aim is to use more than a common $sin(x)$ to show that the model does not merely copy previous data.

Figure 6.1: Modified cosine (*preprocessing.py*)

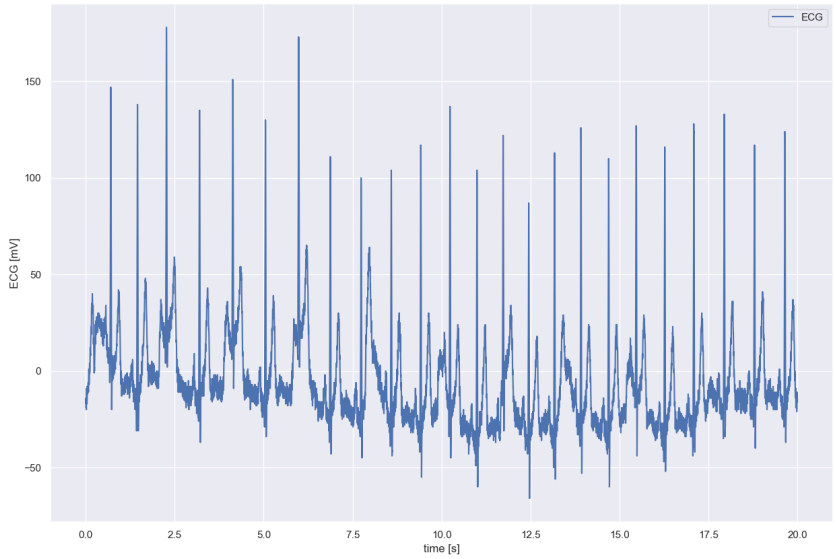


Since the data is generated, there are no missing values or outliers.

6.3.2 ECG data

The second dataset is a raw ECG signal recorded for 20 seconds, digitised at 500 Hz with 12-bit resolution over a nominal ± 10 mV range, obtained from a 25-year-old man [43] retrieved from PhysioBank[1] [28]. The record contains 10 000 samples and, before using it, the quotation marks were manually removed out of the dataset and the column *ECG I* was renamed as *ECG*. This dataset was chosen because the data are real and it has a obvious seasonal component. There are no missing values or outliers.

Figure 6.2: ECG (*preprocessing.py*)



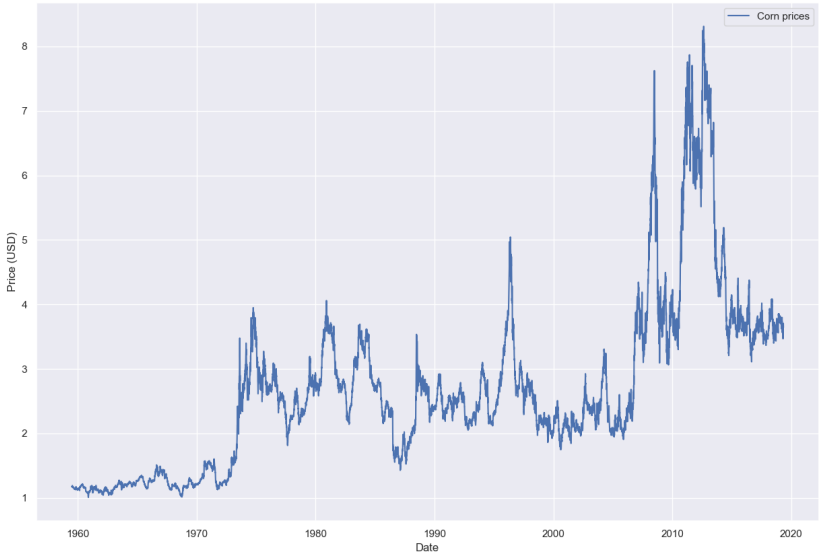
6.3.3 Corn prices

The third dataset collects daily (workdays) corn prices over the years 1959 to 2019 retrieved from [20]. It consists of the date and the current price of corn in U.S. Dollars per bushel[2]. The additional information apart from the data (15 rows) was deleted, leaving 15 110 dates and corresponding prices.

[1]Database: ECG-ID Database, Record: Person_01/rec_1, Signals: ECG I

[2]a measure of capacity equal to 8 gallons (equivalent to 36.4 litres), used for corn, fruit, liquids, etc. [21]

Figure 6.3: Corn prices (*preprocessing.py*)



7 Side-channel injection

One of the greatest advantages of neural networks over classic methods is that these models can process multivariate data (and also multivariate time series) irrespective of the type of network. The goal is to speed up the training or to have the forecasts more accurate. Some of the classic methods have been modified to treat multivariate data as well – e.g. ARIMAX [15], where the letter X represents Exogenous. Another exogenous (determined outside the model) variable is added to improve forecasts. A good example are daily restaurant sales as the variable to be forecast, and a vector of zeros and ones defining holidays. The model needs values (true or good estimates) of the exogenous variable also for the predicted time steps.

In stock price forecasting, it is common practice to add other stock tickers as an input in order to take into consideration other unexplored factors [25]. For some datasets, there are no another useful time series available. For this reason, other inputs computed directly from observed data are used in this thesis.

The datasets described in Section 6 all consist of two columns - the index and the observe variable. Although the index could be fed as another input (two variables per time step), for some data it does not hold any information. The modified cosine dataset has index values ranging from -50 to 50, which means that there is no information about the repeating periods (which the cosine function has). For this reason, the index values will not be used as an input vector, because the aim is to present more general technique.

The first injected side channel is the mean. The mean is computed for a defined number of time steps m (new hyperparameter) of the last data before the target data and copied for the whole part as a new variable. As an example, considering input data $[10, 11, 12, 13, 14, 15, 16]$ with 7 time steps in encoder, the target dataset is $[17, 18]$ using 2 time steps in decoder. Having $m = 10$, the mean is computed from the observations $[7, 8, 9, 10, 11, 12, 13, 14, 15, 16]$ and a value $mean([7, 8, 9, 10, 11, 12, 13, 14, 15, 16]) = 11.5$ is copied 7 times and

concatenated with the defined observations. When there are not enough observations to have m data points (at the beginning of data), the mean is computed of the maximum data possible. It is assumed that this method could improve learning.

The second injected side channel is a slope and intercept of linear regression. With regard to the mean, there is a new hyperparameter which defines number of observations used to fit a line. Linear regression for these observations is performed and the slope and intercept is then fed as two new variables with the observed input data. It is assumed that these new variables could provide information on the increasing or decreasing trend and assist the network to learn it.

Both of the methods prolong the process of training. First, the hyperparameter m needs to be found and second, the dimension of the input is larger, which results in larger size of the weight matrices.

8 Software toolkit

This section is dedicated to an overview of programming language, libraries and the tools used overall for the implementation.

8.1 Python

Python is an easy-to-use programming language, the standard library of which offers a wide range of facilities. It is also interoperable with a vast selection of other libraries, modules and even entire application development frameworks, which makes it an excellent tool for use in multitude fields of study, specifically Data Science and Machine Learning. Its additional library NumPy supports multi-dimensional arrays and adds high-level mathematical functions useful for data science [66], [48].

8.2 TensorFlow

TensorFlow developed by the Google Brain team is an open-source framework especially used for machine learning applications. It is available for different operating systems and mobile computing platforms. Implemented functions and modules are extended by a module *tensorflow.contrib* which is not included in the main repository. However users can contribute with own code or use functions of this module. TensorFlow can also run on multiple CPUs and GPUs which fastens the training. The platform also provides checkpoints to save and restore TensorFlow (TF) models built with Estimators and a utility called TensorBoard for visualising computation graphs, parameters distributions and other features. Since its release in 2015, TF added the support in 2017 of another open-source neural network interface – Keras, using it as a high-level API[1] for building deep learning models. In this work, Low Level APIs are used that provide more transparency and the necessary flexibility.

[1] Application Programming Interface

8.2.1 TensorFlow API

TensorFlow API provides two major features – a dataflow graph which stores all the computations with abstract representation of the data and a session where the actual data are fed and the operations defined in the graph are exectuted. One big advantage is that the parts of the graph are executed separately and one can control in a session what part of the graph is required to be run. The term "tensor" represents a multi-dimensional array of objects (generally numbers) with specified properties. Tensors in TensorFlow represent the data that flow in the computational graph. There are three major tensor types:

1. Constant – a tensor with a value that does not change,
2. Variable – a tensor that can change its value,
3. Placeholder – an empty tensor without initial value, to which the data (training examples) are fed in a session.

Variables are used as network parameters which are required to be updated in order to minimise the error. Before running a session they need to be initialised with operation *global_variables_initializer()*.

Through placeholders, different data can be fed in every iteration (mini-batches). These types of tensors have different properties, but the basics are name, data type, shape or value. When the shape is defined, it is not possible to change it. However, in situations where the shape is required to be changed (for example size of the mini-batch), the value *None* can be used to have an unspecified (dynamic) dimension in placeholders. To feed the data into the placeholders, they are passed inside the *feed_dict* argument to a method *run* of class *Session*. Another parameter of this method is *fetches*, where the tensors are specified in the computational graph required to be evaluated.

8.2.2 TensorBoard

TensorBoard is mentioned in Section 8.2. This is a visualisation software that comes packaged with TensorFlow to see the connections in the graph

suitable for debugging and optimising the program. In the bookmark Graphs, the computation graphs can be visualised, using a *name_scope* class that creates namespaces to differentiate between operations to make the graph understandable. It is possible to zoom in to what the inputs and outputs of each operation are, or even the properties of the tensors. To see the histograms of parameters, the operation *summary.histogram* needs to be added, visible in the bookmark Histograms. In the bookmark Distributions, the distributions of the parameters can be seen. With *summary.scalar* any metric that changes over time (iteration) can be added, e.g. the loss, to be seem in bookmark Scalars, if the model is learning. There are another bookmarks, but these are not suitable for this model [64].

8.3 Matplotlib

Matplotlib is an open-source Python library used for visualisations of data. It provides a wide range of different plots [44].

8.4 Seaborn

Seaborn is a library based on Matplotlib, used for more attractive visualisations of statistical graphs [60].

8.5 Scikit-learn

Scikit-learn is an open-source software library that provides tools for data analysis. There are classes for preprocessing and overall machine learning problems, such as classification, regression, clustering or dimensionality reduction. It is built on NumPy and Matplotlib [58].

8.6 Pandas

Pandas is another open-source library for Python used for data analysis. It is good for data importing. Its class DataFrame is an excellent method for representing tabular data, assisting in data preprocessing, modification or slicing [53].

8.7 StatsModels

"StatsModels is a Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration."[62] The module does not support tensors from TensorFlow.

8.8 SciPy

SciPy is another Python-based open-source library for science and mathematics [59].

9 Implementation

9.1 Main program

The main script is defined in *main.py*. Firstly, all the libraries are imported and the frequently used ones are given aliases.

```
import numpy as np
import tensorflow as tf
import pandas as pd
```

It is also necessary to import other scripts with the defined functions or classes that the main script will call. In the script *hyperparameters.py* the hyperparameters summarised in Section 4 are defined. The user changes these hyperparameters and in that way modifies the model. In the script *models.py*, different models are defined. The script *data_processing.py* is used for the computations specified in functions that are not in the main script for clarity, and visualisations. In the script *optimize.py*, the optimiser described in Section 4.5 is defined.

```
import hyperparameters as hp
from models import *
from data_processing import *
from optimize import *
```

The data below are imported, divided into validation, training and test dataset (1:8:1) and scaled, using the z-score and Min-Max scaling, so that the training dataset is within the interval $[-1, 1]$.

```
data, column = import_data(hp.options['input_data'])
data[data.columns[0]] = range(0, len(data))
valid, train, test = train_test_valid_split(
    data, hp.options['train_ratio'])


valid_scaled, train_scaled, test_scaled = valid, train, test

deviation, mean, valid_scaled['scaled'], train_scaled['scaled
    '], test_scaled['scaled'], train_scaled_mean = zscore(
    valid_scaled[column], train_scaled[column], test_scaled[
        column])
```

```
valid_scaled['scaled'], train_scaled['scaled'], test_scaled['
    scaled'], scaler = normalization(
     valid_scaled, train_scaled, test_scaled, 'scaled')

original = column
column = 'scaled'
columns = []
columns.append(column)
```

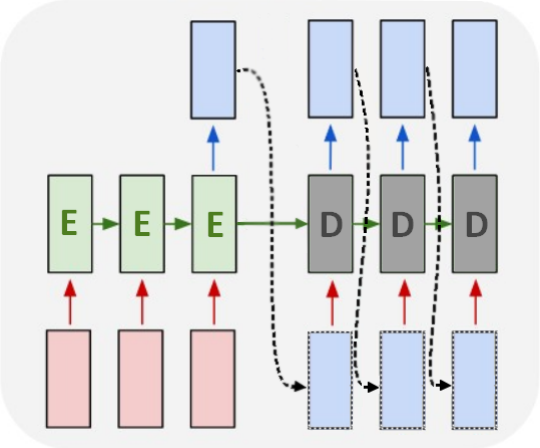
In the model, the network takes the hyperparameters from *hyperparameters.py*, whether to use a side-channel or not, the number of time steps in the encoder and decoder, batch size, number of stacked hidden layers, number of hidden neurons, learning rate, type of the architecture are defined as well as the type of RNN unit. This information is used to build the model.

```
with tf.name_scope('input'):
        encoder_inputs = tf.placeholder(tf.float32, shape=(
                None, options['steps_enc'], num_inputs), name
                    ='encoder_inputs')
        encoder_inputs_ta = [tf.squeeze(t, [1]) for t in tf.
            split(
                encoder_inputs, options['steps_enc'], 1)]

        current_batch_size = tf.shape(encoder_inputs_ta)[1]

        decoder_targets = tf.placeholder(
                tf.float32, shape=(None, options['steps_dec'
                    ], num_outputs), name='decoder_targets')
        decoder_targets_ta = [tf.squeeze(t, [1]) for t in tf.
            split(
                decoder_targets, options['steps_dec'], 1)]

model = init_model(options['model_type'], current_batch_size,
                           encoder_inputs_ta,
                              decoder_targets_ta, num_units,
                              options['cell_type'])
decoder_outputs = model.prediction
targets = model.target

with tf.name_scope('loss'):
        loss = tf.reduce_mean(
                tf.square(decoder_outputs - targets), name='
                    loss')

training = Optimize(loss, gradient_clipping=False)
```

The model called from the function *init_model* is defined as RNN Autoencoder

(in Tables referenced as "RNNAutoencoder") and the structure is shown in Figure 9.1. The encoder's and decoder's parameters are separated from each other, which means there are twice as many parameters as in single RNN (e.g. in Equation 3.10).

Figure 9.1: RNN Autoencoder as a baseline model



In TensorFlow Session, the data are fed into placeholders and the network is trained. Early stopping is applied to stop the training, where the stopping condition is based on the validation dataset (defined at the beginning of the time series) when the error (of validation data) has not decreased for the set number of time steps, the training is stopped and parameters are restored from iteration, with the lowest error of the validation dataset and its predictions. To see the training progress, the errors of all three datasets and their predictions are sent to TensorBoard using *summary.scalar*. The predictions are visualised every 100th iteration, to see the progress of the predictions, not only of the error.

9.2 Find a baseline

Some of the hyperparameters were set w.r.t. experiments conducted during programming and tuning.

- Activation functions: set to the default functions in TensorFlow, which means they correspond to the models in Equations 3.10 and 3.12;

- learning rate: 0.001;
- optimizer: Adam;
- objective function: MSE;
- train ratio: 80% – 10% validation data, 80% training data, 10% test data;
- gradient clipping: None;
- early stopping: 2 000 iterations without decreasing of MSE of validation dataset and its predictions stop the training.

To find a baseline model, firstly the number of time steps for both encoder w and decoder q needs to be defined. Time steps are usually set by the researcher according to the desired number of predicted days, but here Bayesian Optimisation and the script *hopt_bo_w.py* are also used to assist with the decision. The possible interval of encoder time steps was set to $[5, 100]$ and decoder time steps to $[1, 100]$, and $w \geq q$. The training loop was set to 100 or 300 iterations, both with 50 models, using MSE of the validation dataset as the function to be minimised. Batch size was set to 250 and number of stacked hidden layers to 1 with 350 hidden neurons and 2 with 200 hidden neurons (two different models).
Both the best combination of time steps for validation and the training dataset are shown in Tables 9.1 and 9.2.

Table 9.1: Time steps exploration: Bayesian Optimization - training

TRAINING		**ECG**	**COS**	**CORN**
100 iter	**2x200**	[88,1]	[5,1]	[5,1]
	1x350	[100,1]	[5,1]	[5,1]
300 iter	**2x200**	[6,1]	[99,2]	[6,1]
	1x350	[5,1]	[99,1]	[5,1]

As a result model with 100 time steps in encoder and 20 time steps in decoder is used.

Table 9.2: Time steps exploration: Bayesian Optimization - validation

VALIDATION		**ECG**	**COS**	**CORN**
100 iter	**2x200**	[100,17]	[100,1]	[100,17]
	1x350	[100,11]	[58,1]	[6,1]
300 iter	**2x200**	[54,48]	[96,19]	[6,1]
	1x350	[41,36]	[100,18]	[5,1]

The most suitable number of hidden layers and hidden neurons together with mini-batch size and type of cell (GRU or LSTM) is found using Bayesian Optimization. Possible configurations were set to:

- number of stacked layers: $[1, 5]$,
- number of hidden neurons in one layer: $[10, 400]$,
- size of mini-batch: $[5, 400]$,
- cell types: $[GRU, LSTM]$.

The limitation is due to a memory error caused by a tensor with too many elements. For 100 or 300 iterations the Bayesian Optimization was performed with two different random initialisations:

1. 3 stacked hidden layers, 150 hidden neurons in each layer, mini-batch size:20, LSTM cell;
2. 1 hidden layer, 10 hidden neurons, mini-batch size: 100, GRU.

50 models for each initialisation were executed for 300 iterations. Both best combination of hyperparameters in order [number of layers, number of hidden neurons, mini-batch size, type of RNN cell] for validation and training dataset is shown in Tables 9.3 and 9.4.

Table 9.3: Hyperparameters exploration: Bayesian Optimization - training

TRAINING		**ECG**	**COS**	**CORN**
[100,20]	**1. init**	[1,385,384,GRU]	[1,95,381,GRU]	[1,399,392,GRU]
	2. init	[1,396,363,LSTM]	[1,392,385,LSTM]	[1,393,391,GRU]

Table 9.4: Hyperparameters exploration: Bayesian Optimization - validation

VALIDATION		**ECG**	**COS**	**CORN**
[100,20]	**1. init**	[2,344,383,LSTM]	[3,234,138,GRU]	[1,399,392,GRU]
	2. init	[1,396,363,LSTM]	[3,255,98,GRU]	[1,393,391,GRU]

Using more iterations the number of hidden neurons and batch size converges to the limits. For this reason the hyperparameters are set to [1, 400, 400, GRU][1]. When a baseline is set, 10 experiments for each dataset are conducted to obtain the errors (MSE) of the targets and predictions. For the ECG dataset, the experiments were conducted only with LSTM as the RNN unit.

9.3 Models with side channels

The number of observations m of which the new variables defined in Section 7 are computed is a new hyperparameter, but it is likely to be different for each dataset, highly depending on fluctuating trend. Grid Search method was performed to find a suitable m and the process was repeated three times, each time with different seed. For each m, the model was trained for 300 iterations. The final m equals to a minimum of the averages over the three trials of the validation error (the errors of targets and prediction of the validation dataset). The values for both model using linear regression and model using means as a side channel are recorded in Table 9.5.

Table 9.5: Hyperparameters for side channels

	ECG	**COS**	**CORN**
LR	400	120	280
means	40	340	320

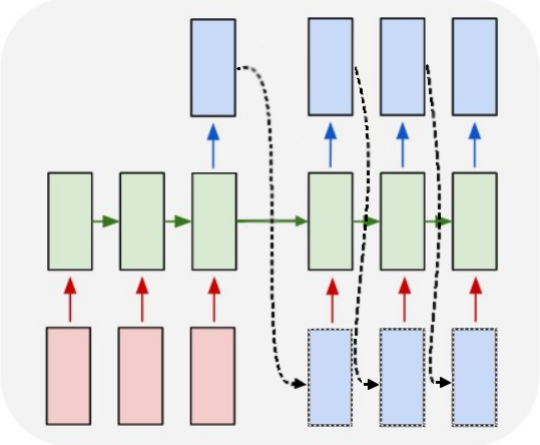
[1]For the ECG dataset several experiments were conducted also with LSTM unit.

For the baseline model, where the data are extended by the new variables (the algorithm is defined in Section 7), the experiments were conducted 10 times for each dataset. As for the baseline model, no seed was used.

9.4 Augmented model

To lower the number of parameters, new model called "Augmented RNN" is introduced. This model feeds the outputs of the encoder as an input to the decoder the same way as RNN Autoencoders, but the parameters are shared across the whole model. This means that the number of parameters is the same as in basic RNN, but the model can output predictions derived w.r.t. own previous predictions, not only fed observations. The structure of the model is shown in Figure 9.2.

Figure 9.2: New model RNNAugmented



Unlike for machine-learning translation tasks, it is possible to omit decoder, because the data of observations and the targets are from the same distribution. The hyperparameters of this model were set to the same values as for the baseline model. 10 experiments were conducted again.

10 Results

The averages of RMSE of the targets and re-scaled predictions of the test dataset are given in Table 10.1. Since only 10 experiments were conducted and means are affected by outliers, medians are included in the table.

Table 10.1: Average test RMSE for different models

Models/Data	ECG		COS		CORN	
	mean	median	mean	median	mean	median
RNNAutoencoder	7.6172	8.1292	1.0399	1.0400	0.1900	0.1893
RNNAugmented	8.1357	7.6583	1.0790	1.0817	0.2060	0.2042
LR	16.0131	16.0660	1.0779	1.0611	0.2159	0.1991
means	12.8580	13.6399	1.0468	1.0467	0.2244	0.1933
AE_LSTM	9.4129	8.7434	–	–	–	–

Sample means, sample standard deviations and numbers of samples in the test data for each dataset are written Table 10.2.

Table 10.2: Characteristics of the datasets

	ECG	COS	CORN
sample mean	-6.3695	-0.4403	3.7854
sample standard deviation	21.0198	4.1754	0.3651
number of samples	820	1320	1320

10.1 Model selection using Hypothesis Testing

Looking at the results in Table 10.1, the baseline model seems to have the best performance for all of the datasets. To support this claim,

statistical significance test need to be applied, to find out, whether the error difference of the baseline model and others is significant. However, selecting a statistical hypothesis test for comparing machine learning algorithms can be challenging and the authors argue, which test is the best to be applied.

For each algorithm and dataset, we have only 10 computed errors. Several sources recommend the McNemar's Statistical Hypothesis Test, because it is suitable for small sample size of data, but the test is suitable for classifiers only. Another recommended test is the Paired t-test (Student's or Welch's), but one of the assumptions is, that the data are normally distributed. A non-parametric version of paired t-test is the Wilcoxon Signed-Rank Test. However, the test assumes that the errors are in pairs, but since seeds were not used for the experiments, the order of the errors in each sample could affect the results. Instead, Kolmogorov-Smirnov Test is applied. It is a non-parametric test, which quantifies the distance between the empirical distribution functions of two samples and by that decides, whether the two samples are from the same distribution. If the null hypothesis is rejected (p-value is less than the chosen significance level), the datasets are not drawn from the same distribution [11], [52], [12], [54].

For normally distributed data, the Welch's t-test[1] is applied. To decide, whether the data are Gaussian, Shapiro–Wilk Test is used, because it is appropriate for small datasets.

P-values of the tests were calculated using SciPy library with the script *results.py* and MSE of the test dataset for the models. The results are given in Table 10.3. For ECG dataset, the null hypothesis cannot be rejected for Augmented RNN and the RNN Autoencoder with LSTM units, whereas for both models with side channels the null hypothesis is strongly rejected. This means that the results of these two models are similar to the baseline model (especially the Augmented RNN model) and with the significance level 0.05 and 0.1 we cannot tell that the baseline model is more significant and better. For the modified cosine dataset the null hypothesis is rejected for the Augmented RNN model and model using means as a side channel – both for Kolmogorov-Smirnov test and Welch's t-test. For the model using linear regression, the null hypothesis cannot be rejected, but it does not differ much

[1]Welch's t-test, unlike the Student's t-test, does not assume equal variance.

from the significance level 0.1 (the p-value is low). For dataset with corn prices, Kolmogorov-Smirnov test does not reject the null hypothesis for the model using means as the side channel. Both Augmented RNN and model with linear regression are strongly rejected as to have errors from the same distribution as the baseline model. The results are quite different for different datasets. To have more significant results, it is required to conduct more experiments. However, this hypothesis testing has shown, that for selecting the best model, it is not sufficient to choose to model w.r.t. results of only one dataset.

Table 10.3: Computed p-values

Models/Data	**ECG**	**COS**		**CORN**
	K-S	**Welch's**	**K-S**	**K-S**
RNNAugmented	0.6751	1.148e-07	1.888e-05	1.888e-05
LR	1.888e-05	-	0.1108	0.0069
means	0.0012	0.0443	0.0310	0.3129
AE_LSTM	0.1108	-	-	-

10.2 Model selection using AIC

AIC was defined in Section 5.5. First, it is required to compute the number of parameters in each model. The weight matrix in the output layer has size $M \cdot H$ and the bias is a vector of length M. LSTM cell has 3 gates and 1 candidate value, each with a weight matrix of size $(N+H) \cdot H$, where N is the dimension of vector of observations in one time step and H is the number of hidden neurons. Each of these units has also a bias – a vector of length H. Hence simple RNN with LSTM units has $4 \cdot [(N+H) \cdot H + H] + M \cdot H + M$ parameters. RNN Autoencoder has twice as many parameters, because the weights in RNN cells of an encoder and a decoder are separated. Similarly, GRU has $3 \cdot [(N+H) \cdot H + H]$ parameters. The number of parameters for each model is:

- GRU Autoencoder – $k = 2 \cdot [3 \cdot [(1+400) \cdot 400 + 400] + 400 + 1] = 965602$,
- RNNAgumented – $k = 3 \cdot [(1+400) \cdot 400 + 400] + 400 + 1 = 482801$,

- LR – $k = 2 \cdot [3 \cdot [(3 + 400) \cdot 400 + 400] + 400 + 1] = 970402$,
- means – $k = 2 \cdot [3 \cdot [(2 + 400) \cdot 400 + 400] + 400 + 1] = 968002$,
- LSTM Autoencoder – $k = 2 \cdot [4 \cdot [(1+400) \cdot 400+400]+400+1] = 1287202$.

Due to large numbers of parameters, the values of AIC are large numbers as well. AIC value was computed for each experiment (using the script *aic.py*, model and dataset, and the averages are given in Table 10.4.

Table 10.4: AIC of different models and datasets

Models/Data	**ECG**	**COS**	**CORN**
RNNAutoencoder	2.275e+9	1.414e+9	1.414e+9
RNNAugmented	5.690e+8	3.537e+8	3.537e+8
LR	2.298e+9	1.428e+9	1.428e+9
means	2.286e+9	1.421e+9	1.421e+9
AE_LSTM	4.042e+9	–	–

The number of parameters strongly penalises the AIC and therefore the model with the lowest number of parameters has the lowest AIC. Since the corn dataset has MSE lower than 1, it is the only dataset where the natural logarithm returns negative number, but it is significantly lower than the other addends. As a result the criterion considers mostly the number of parameters and not how well the model predicts.

10.3 Visual result comparison

To see how well the model fits the data, it is a common practice to visualise the predictions together with the target data. For all three datasets the experiment where its RMSE is the median[2]. At the first sight, both methods with side channels fit the data well for the modified cosine dataset (see Figures 10.1 and 10.2). However, for the ECG dataset, the models do not predict as well as the models with only one variable (see Figures 10.5, 10.6, 10.7 10.3 and10.4).

Figure 10.1: Cos – LR

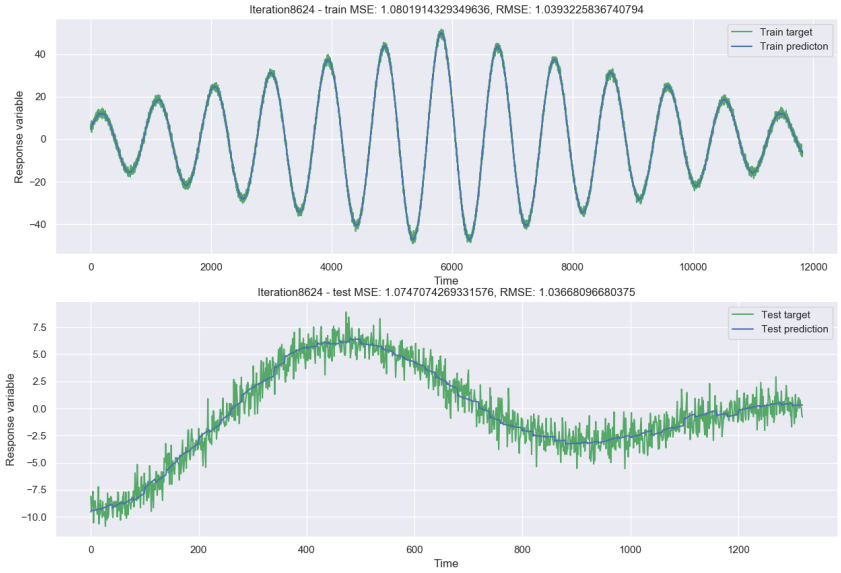
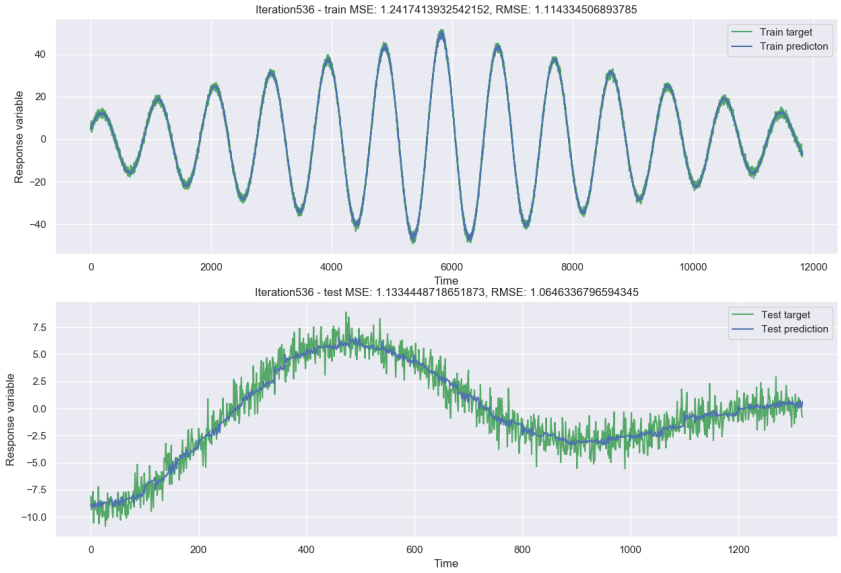
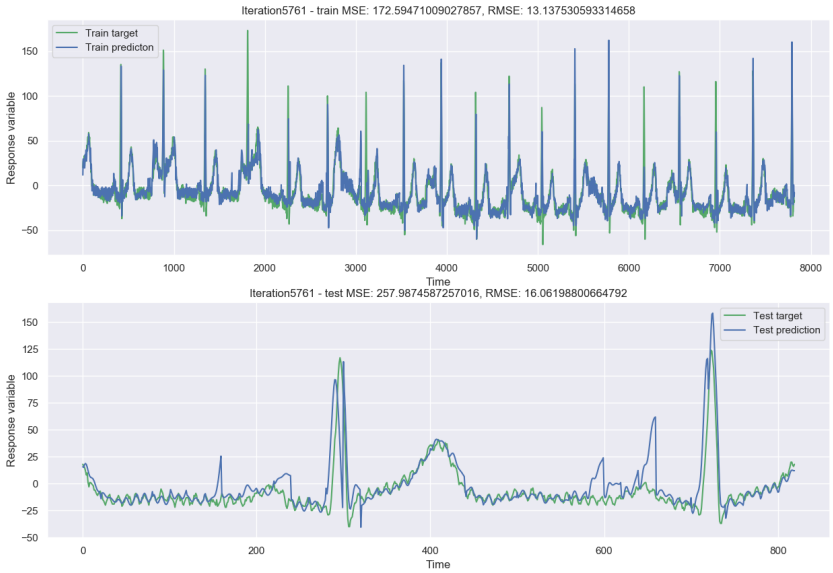


Figure 10.2: Cos – mean



[2]since the number of experiments is even, it is taken the fifth value in ascending order

Figure 10.3: ECG – LR



To decide, which model to choose from the three models using univariate data (GRU Autoencoder, LSTM Autoencoder and Augmented RNN), it is possible to visualise the individual parts of the test data. The Figures A.1 to A.15 are shown in Appendix. As a result we decided to go with the new model Augmented RNN, because the results are very similar to the baseline model, it is faster to train the model (i.e. not as time-consuming

Figure 10.4: ECG – mean

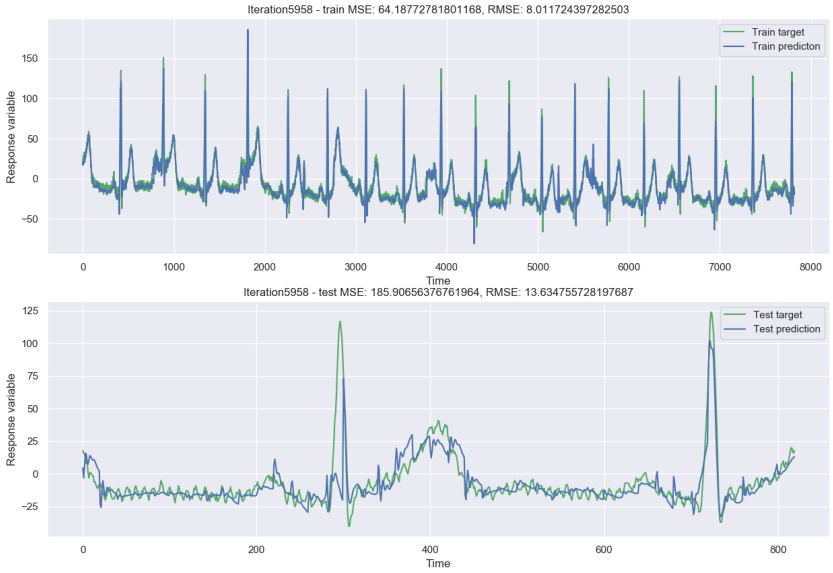
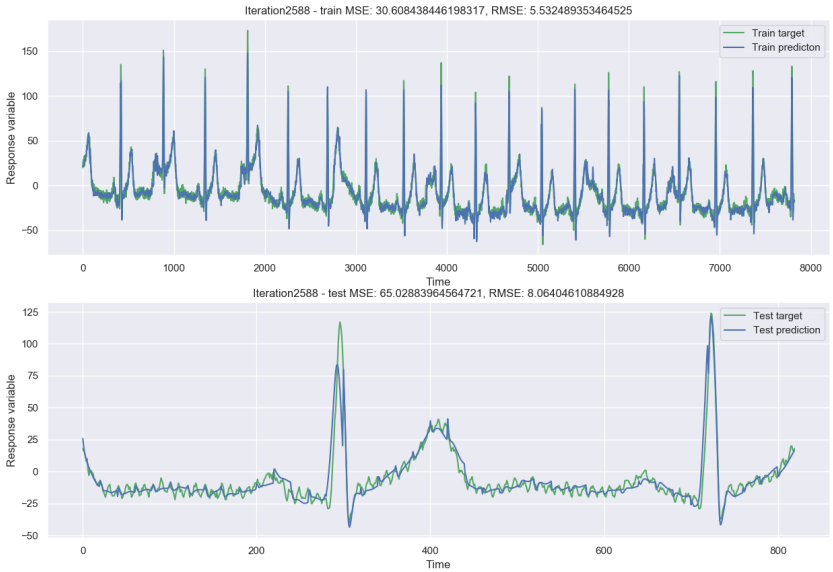


Figure 10.5: ECG – baseline



as the RNN Autoencoder) and it is likely to assume, that by optimising the hyperparameters w.r.t. the augmented model, the results could be improved.

Finally, the results of the third dataset are to be analysed. The MSE results

Figure 10.6: ECG – LSTM Autoencoder

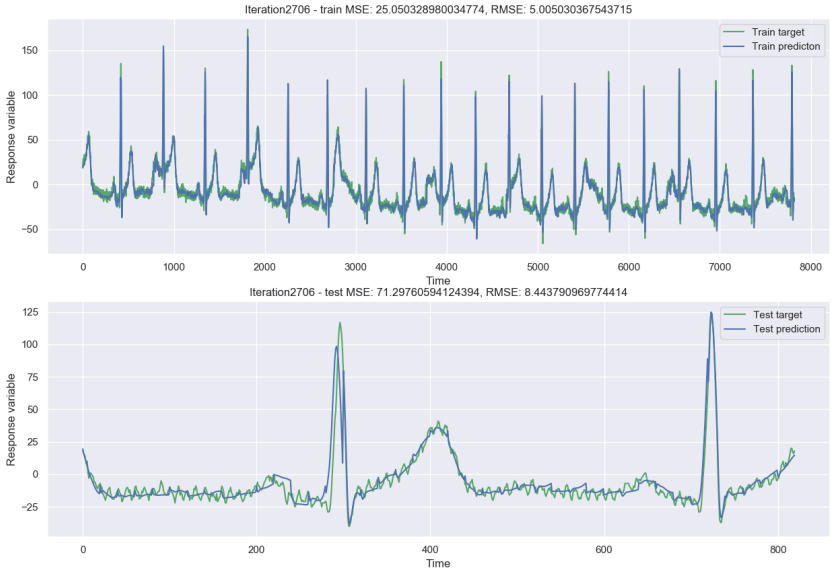
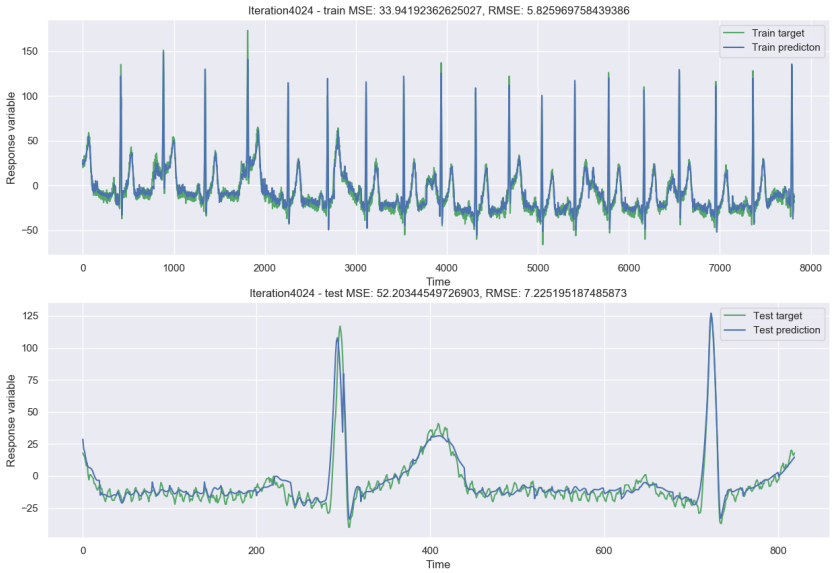


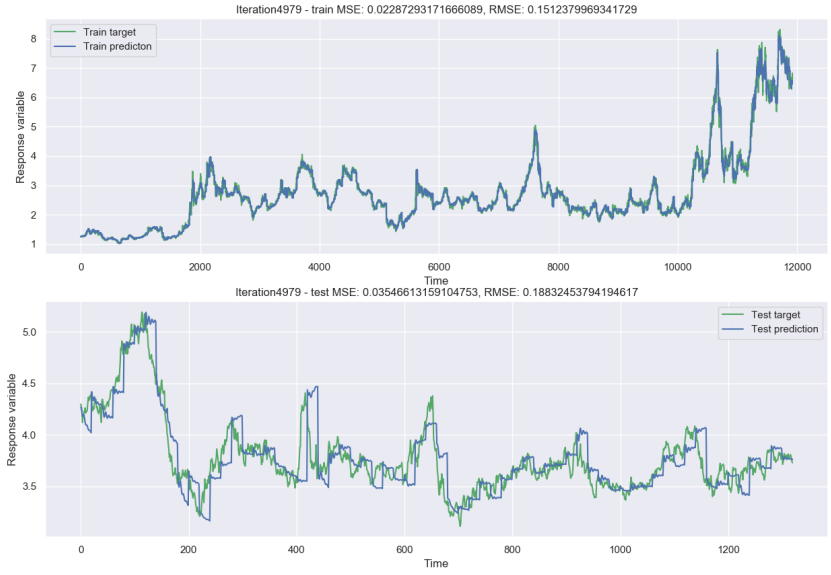
Figure 10.7: ECG – RNNAugmented



of the corn prices data were surprising, because we have assumed, that this model is going to be the hardest to train. From the graph shown in Figure 10.8 it seems, that the network has learnt the levels of each prediction quite

well and lowering the number of time steps in the encoder and the decoder (as suggested by the Bayesian Optimisation).

Figure 10.8: Corn – baseline



The Figure 10.9 shows how well does the model perform using the 5 time steps in the encoder and 1 time step in the decoder. The test targets are suspiciously well fitted for a dataset with no clear seasonality (visible at first sight). In Figure 10.10 it is clearly visible, that the predictions are shifted by 1 time step to the future. The network simply reproduces the last given observation for each fed mini-batch. However, Naïve method "works remarkably well for many economic and financial time series" [36] which means, that the network did not fail to fit the data and 1 time step in the decoder is an appropriate value.

Figure 10.9: Corn w5_q1

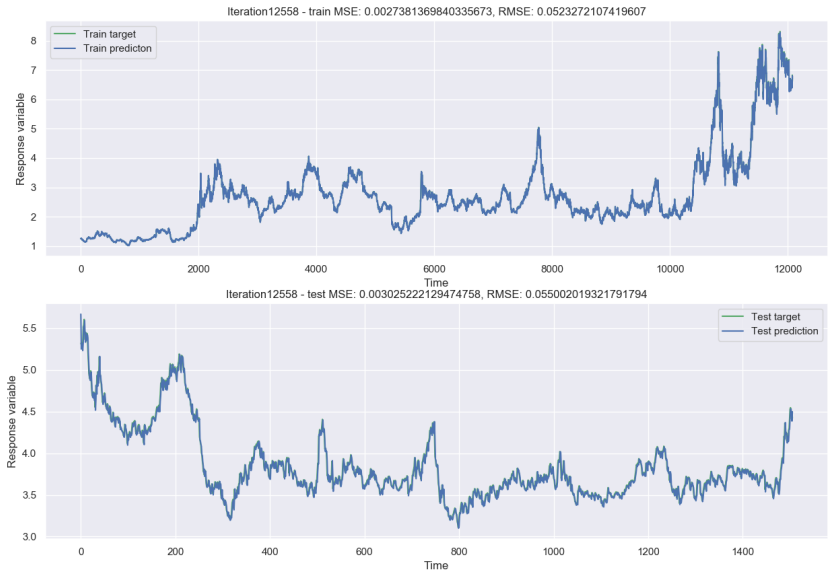
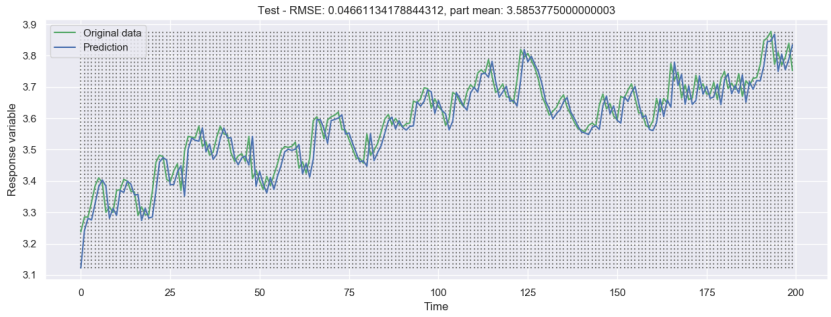


Figure 10.10: Corn w5_q1



11 Conclusion

In this thesis, methods for time series forecasting, using machine learning techniques (RNNs, LSTMs, GRUs, RNN Autoencoders) were described, along with possible data structures and respective algorithms in detail. Given this theory a baseline model was implemented. After, the datasets, presented in Section 6, were modified in order to create a supervised learning problem, and the baseline model's hyperparameters, discussed in theoretical part, were set using Bayesian Optimisation.

In following Section 7 new models, which use additional information derived directly from the dataset, were introduced. One of the models uses the mean and the second one slopes and intercepts of linear regression performed over several previous observations. These values extend the input data as new variables and the number of previous observations is defined using Grid Search.

A new structure of RNN was introduced during implementation as a compromise between simple RNNs and RNN Autoencoders. This augmented version of RNN decreases the number of hyperparameters, while having similar structure as RNN Autoencoders. For every one of these four models several experiments were conducted. Hypothesis testing and Akaike's Information Criterion were used as methods for suitable model selection.

The result of performed experiments is, that the models with side-channel injection defined in this thesis do not demonstrate any significant advantage over the baseline model and the augmented model. The baseline model has on average the best performance, but the Kolmogorov-Smirnov test did not reject the null hypothesis for all of the models and datasets, which means that the errors of different models may be drawn from the same distribution. However, the Augmented RNN model is selected as the best model, because the results are similar to the baseline model and the structure outperforms the baseline in AIC. Future work can be focused on performing massive testing of the new augmented structure by various sets of data.

Bibliography

[1] AGARAP, Abien Fred M. Deep Learning using Rectified Linear Units (ReLU) [online]. 7 February 2019. Retrieved from: https://arxiv.org/pdf/1803.08375.pdf

[2] AKAIKE, Hirotogu. Information Theory and an Extension of the Maximum Likelihood Principle. Selected Papers of Hirotugu Akaike. New York, NY: Springer New York, 1998, 1998, 199-213. Springer Series in Statistics. DOI: 10.1007/978-1-4612-1694-0_15. ISBN 978-1-4612-7248-9. Retrieved from: http://link.springer.com/10.1007/978-1-4612-1694-0_15

[3] ALESE, Eniola. The curious case of the vanishing & exploding gradient [online]. In: Learn.Love.AI - Medium. 6 June 2017 [cit. 2019-05-08]. Retrieved from: https://medium.com/learn-love-ai/the-curious-case-of-the-vanishing-exploding-gradient-bf58ec6822eb

[4] BAL, Cagatay, Serdar DEMIR and Cagdas ALADAG. A Comparison of Different Model Selection Criteria for Forecasting EURO/USD Exchange Rates by Feed Forward Neural Network. International Journal of Computing, Communication and Instrumentation Engineering [online]. 2016, 3(2) [cit. 2019-06-20]. DOI: 10.15242/IJCCIE.U0616010. ISSN 23491477. Retrieved from: http://iieng.org/images/proceedings_pdf/U0616010.pdf

[5] BALAMULARI, Murugesan. Importance of learning rate in machine learning. In: Medium [online]. 15 September 2017 [cit. 2019-06-12]. Retrieved from: https://medium.com/@balamuralim.1993/importance-of-learning-rate-in-machine-learning-920a323fcbfb

[6] BENGIO, Y., P. SIMARD and P. FRASCONI. Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks [online]. 5(2), 157-166 [cit. 2019-05-11]. DOI: 10.1109/72.279181. ISSN 10459227. Retrieved from: http://ieeexplore.ieee.org/document/279181/

[7] BROCKWELL, Peter J. and Richard A. DAVIS. Introduction to time series and forecasting. 2nd ed. New York: Springer, c2002. ISBN 03-879-5351-5.

[8] BROWNLEE, Jason. A Gentle Introduction to Exploding Gradients in Neural Networks. In: Machine Learning Mastery [online]. 18 December 2017 [cit. 2019-05-11]. Retrieved from: https://machinelearningmastery.com/exploding-gradients-in-neural-networks/

[9] BROWNLEE, Jason. How to Improve Neural Network Stability and Modeling Performance With Data Scaling. In: Machine Learning Mastery [online]. 4 February 2019 [cit. 2019-05-11]. Retrieved from: https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/

[10] BROWNLEE, Jason. How to Prepare Sequence Prediction for Truncated Backpropagation Through Time in Keras. In: Machine Learning Mastery [online]. 28 June 2017 [cit. 2019-05-11]. Retrieved from: https://machinelearningmastery.com/truncated-backpropagation-through-time-in-keras/

[11] BROWNLEE, Jason. How to Use Statistical Significance Tests to Interpret Machine Learning Results. In: Machine Learning Mastery [online]. 3 May 2017 [cit. 2019-06-25]. Retrieved from: https://machinelearningmastery.com/use-statistical-significance-tests-interpret-machine-learning-results/

[12] BROWNLEE, Jason. Statistical Significance Tests for Comparing Machine Learning Algorithms. In: Machine Learning Mastery [online]. 20 June 2018 [cit. 2019-06-25]. Retrieved from: https://machinelearningmastery.com/statistical-significance-tests-for-comparing-machine-learning-algorithms/

[13] BROWNLEE, Jason. Time Series Forecasting as Supervised Learning. In: Machine Learning Mastery [online]. 5 December 2016 [cit. 2019-05-11]. Retrieved from: https://machinelearningmastery.com/time-series-forecasting-supervised-learning/

[14] BROWNLEE, Jason. What is the Difference Between a Parameter and a Hyperparameter?. In: Machine Learning Mastery [online]. 26 July 2017 [cit. 2019-06-06]. Retrieved from: https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/

[15] ĎURKA, Peter and Silvia PASTOREKOVÁ. ARIMA vs. ARIMAX – which approach is better to analyze and forecast macroeconomic time series?. Mathematical methods in Economics : proceedings of 30th international conference [online]. 11-13 September 2012 [cit. 2019-06-16]. Retrieved from: http://mme2012.opf.slu.cz/proceedings/pdf/024_Durka.pdf

[16] CHEN, Gang. A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation [online]. 14 January 2018 [cit. 2019-06-15]. Retrieved from: https://arxiv.org/pdf/1610.02583.pdf

[17] CHO, Kyunghyun, Bart VAN MERRIENBOER, Caglar GULCEHRE, Dzmitry BAHDANAU, Fethi BOUGARES, Holger SCHWENK and Yoshua

BENGIO. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) [online]. Stroudsburg, PA, USA: Association for Computational Linguistics, 2014, 2014, s. 1724-1734 [cit. 2019-06-20]. DOI: 10.3115/v1/D14-1179. Retrieved from: http://aclweb.org/anthology/D14-1179

[18] CIPRA, Tomáš. Analýza časových řad s aplikacemi v ekonomii: celostátní vysokoškolská učebnice pro stud. matem.-fyz. fakult studijních oborů 11 Fyzikálně matematické vědy. Praha: Státní nakladatelství technické literatury, 1986.

[19] COLAH, Christopher. Understanding LSTM Networks [online]. In: colah's blog. 27 August 2015 [cit. 2019-05-11]. Retrieved from: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[20] Corn-prices-historical-chart-data.csv. In: MacroTrends: Corn Prices - 45 Year Historical Chart [online]. [cit. 2019-06-17]. Retrieved from: https://www.macrotrends.net/2532/corn-prices-historical-chart-data

[21] Definition of bushel in English by Lexico Dictionaries: bushel [online]. [cit. 2019-06-24]. Retrieved from: https://www.lexico.com/en/definition/bushel

[22] DESHPANDE, Mohit. A Guide to Improving Deep Learning's Performance [online]. In: Zenva. 10 February 2017 [cit. 2019-06-12]. Retrieved from: https://pythonmachinelearning.pro/a-guide-to-improving-deep-learnings-performance/

[23] ELMAN, J. Finding structure in time. Cognitive Science [online]. 1990, 14(2), 179-211 [cit. 2019-03-13]. DOI: 10.1016/0364-0213(90)90002-E. ISSN 03640213. Retrieved from: http://doi.wiley.com/10.1016/0364-0213(90)90002-E

[24] FUMO, David. Types of Machine Learning You Should Know [online]. In: Towards Data Science. 15 June 2017 [cit. 2019-05-08]. Retrieved from: https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861

[25] Galovič, Marek. Time Series Forecasting with RNNs [online]. In: Towards Data Science. 2 November 2018 [cit. 2019-06-16]. Retrieved from: https://towardsdatascience.com/time-series-forecasting-with-rnns-ff22683bbbb0

[26] GERS, Felix A., Jürgen SCHMIDHUBER and Fred CUMMINS. Learning to Forget: Continual Prediction with LSTM. Neural Computation [online]. IEE, 2000, 1999, 12(10), 2451-2471 [cit. 2019-05-13]. DOI:

10.1162/089976600300015015. ISBN 0852967217. ISSN 0899-7667. Retrieved from: http://www.mitpressjournals.org/doi/10.1162/089976600300015015

[27] GITAU, Catherin. Classification in Supervised Machine Learning: All you need to know! [online]. In: Learn.Love.AI - Medium. 13 April 2018 [cit. 2019-06-04]. Retrieved from: https://medium.com/@categitau/in-one-of-my-previous-posts-i-introduced-machine-learning-and-talked-about-the-two-most-common-c1ac6e18df16

[28] Goldberger AL, Amaral LAN, Glass L, Hausdorff JM, Ivanov PCh, Mark RG, Mietus JE, Moody GB, Peng C-K, Stanley HE. PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals (2003). Circulation. 101(23):e215-e220.

[29] GOODFELLOW, Ian, Yoshua BENGIO and Aaron COURVILLE. Deep learning. Cambridge, Massachusetts: The MIT Press, [2016]. ISBN 978-026-2035-613.

[30] GOZZOLI, Allessio. Practical guide to hyperparameters search for deep learning models [online]. 5 September 2018 [cit. 2019-06-23]. Retrieved from: https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/

[31] HAHNLOSER, Richard H. R., Rahul SARPESHKAR, Misha A. MAHOWALD, Rodney J. DOUGLAS and H. Sebastian SEUNG. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. Nature [online]. 2000, 405(6789), 947-951 [cit. 2019-05-11]. DOI: 10.1038/35016072. ISSN 0028-0836. Retrieved from: http://www.nature.com/articles/35016072

[32] HASTIE, Trevor, Robert TIBSHIRANI and J. H. FRIEDMAN. The elements of statistical learning: data mining, inference, and prediction. Corrected ed. New York: Springer, 2003. ISBN 03-879-5284-5.

[33] HOCHREITER, Sepp and Jürgen SCHMIDHUBER. Long Short-Term Memory. Neural Computation [online]. 1997, 9(8), 1735-1780 [cit. 2019-05-11]. DOI: 10.1162/neco.1997.9.8.1735. ISSN 0899-7667. Retrieved from: http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735

[34] HOCHREITER, Sepp and Jürgen SCHMIDHUBER. LSTM Can Solve Hard Long Time Lag Problems [online]. 1997 [cit. 2019-06-12]. Retrieved from: https://papers.nips.cc/paper/1215-lstm-can-solve-hard-long-time-lag-problems.pdf

[35] HOLT, Charles C. Forecasting seasonals and trends by exponentially weighted moving averages. International Journal of Forecasting [online]. 2004,

20(1), 5-10 [cit. 2019-06-22]. DOI: 10.1016/j.ijforecast.2003.09.015. ISSN 01692070. Retrieved from: https://linkinghub.elsevier.com/retrieve/pii/S0169207003001134

[36] HYNDMAN, Rob and George ATHANASOPOULOS. Forecasting: principles and practice. 2nd edition. OTexts, 2018. ISBN 978-0987507105.

[37] JORDAN, Michael I. Serial Order: A Parallel Distributed Processing Approach. Neural-Network Models of Cognition - Biobehavioral Foundations [online]. Elsevier, 1997, 1997, s. 471-495 [cit. 2019-06-20]. Advances in Psychology. DOI: 10.1016/S0166-4115(97)80111-2. ISBN 9780444819314. Retrieved from: https://linkinghub.elsevier.com/retrieve/pii/S0166411597801112

[38] KARPATHY, Andrej. The Unreasonable Effectiveness of Recurrent Neural Networks [online]. May 21, 2015 [cit. 2019-02-28]. Retrieved from: https://karpathy.github.io/2015/05/21/rnn-effectiveness/

[39] KNAGG, Oscar. An intuitive guide to Gaussian processes [online]. 16 January 2019 [cit. 2019-06-23]. Retrieved from: https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/

[40] LE, James. The 7 NLP Techniques That Will Change How You Communicate in the Future: (Part I) [online]. In: Heartbeat. 6 June 2018 [cit. 2019-06-06]. Retrieved from: https://heartbeat.fritz.ai/the-7-nlp-techniques-that-will-change-how-you-communicate-in-the-future-part-i-f0114b2f0497

[41] LECUN, Yann, Leon BOTTOU, B. Genevieve ORR and Klaus-Robert MÜLLER. Efficient BackProp [online]. 1998 [cit. 2019-06-06]. Retrieved from: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

[42] LITSCHMANNOVÁ, Martina. JEDNODUCHÁ LINEÁRNÍ REGRESE [online]. [cit. 2019-06-16]. Retrieved from: https://homel.vsb.cz/ lit40/STA1/Cviceni/PDF/14cRegrese.PDF

[43] Lugovaya T.S. Biometric human identification based on electrocardiogram. [Master's thesis] Faculty of Computing Technologies and Informatics, Electrotechnical University "LETI", Saint-Petersburg, Russian Federation; June 2005.

[44] Matplotlib: Python plotting [online]. [cit. 2019-06-16]. Retrieved from: https://matplotlib.org/

[45] MERITY, Stephen. Peeking into the neural network architecture used for Google's Neural Machine Translation [online]. In: Smerity.com. 2017 [cit.

2019-06-04]. Retrieved from:
https://smerity.com/articles/2016/google_nmt_arch.html

[46] MITCHELL, Tom M. Machine Learning. New York: McGraw-Hill, c1997. ISBN 0070428077.

[47] NIELSEN, Michael. Neural Networks and Deep Learning [online]. Determination Press, 2015 [cit. 2019-03-16]. Retrieved from: http://neuralnetworksanddeeplearning.com/index.html

[48] NumPy [online]. [cit. 2019-06-16]. Retrieved from: https://www.numpy.org/

[49] PANCHAL, Gaurang, Amit GANATRA, Y.P. KOSTA and Devyani PANCHAL. Searching Most Efficient Neural Network Architecture Using Akaike's Information Criterion (AIC). International Journal of Computer Applications [online]. 2010, 1(5), 54-57 [cit. 2019-06-20]. DOI: 10.5120/126-242. ISSN 09758887. Retrieved from: http://www.ijcaonline.org/journal/number5/pxc387242.pdf

[50] PASCANU, Razvan, Tomas MIKOLOV and Yoshua BENGIO. On the difficulty of training recurrent neural networks. Proceedings of the 30th International Conference on Machine Learning [online]. Atlanta, Georgia, USA: PMLR, 2012, 28(3), 1310–1318 [cit. 2019-05-09]. Retrieved from: http://proceedings.mlr.press/v28/pascanu13.pdf

[51] PASCANU, Razvan, Tomas MIKOLOV and Yoshua BENGIO. Understanding the exploding gradient problem [online]. 21 November 2012 [cit. 2019-05-11]. Retrieved from: https://pdfs.semanticscholar.org/728d/814b92a9d2c6118159bb7d9a4b3dc5eeaaeb.pdf

[52] PIZARRO, Joaqín, Elisa GUERRERO and Pedro L. GALINDO. A Statistical Model Selection Strategy Applied to Neural Networks [online]. Bruges (Belgium): ESANN'2000 proceedings - European Symposium on Artificial Neural Networks, 2000 [cit. 2019-06-26]. Dostupné z: https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2000-46.pdf

[53] Python Data Analysis Library – pandas [online]. [cit. 2019-06-16]. Retrieved from: https://pandas.pydata.org/

[54] RONAGHAN, Stacey. Statistical Tests for Comparing Machine Learning and Baseline Performance [online]. In: Towards Data Science. 14 March 2019 [cit. 2019-06-26]. Retrieved from: https://towardsdatascience.com/statistical-tests-for-comparing-machine-learning-and-baseline-performance-4dfc9402e46f

[55] RUDER, Sebastian. An overview of gradient descent optimization algorithms [online]. 15 June 2017 [cit. 2019-06-07]. Retrieved from: https://arxiv.org/pdf/1609.04747.pdf

[56] RUMELHART, David E., Geoffrey E. HINTON and Ronald J. WILLIAMS. Learning representations by back-propagating errors. Nature [online]. 1986, 323(6088), 533-536 [cit. 2019-03-16]. DOI: 10.1038/323533a0. ISSN 0028-0836. Retrieved from: http://www.nature.com/articles/323533a0

[57] SCHMIDHUBER, Jürgen. Deep learning in neural networks: An overview. Neural Networks [online]. 2015, 61, 85-117 [cit. 2019-06-04]. DOI: 10.1016/j.neunet.2014.09.003. ISSN 08936080. Retrieved from: https://linkinghub.elsevier.com/retrieve/pii/S0893608014002135

[58] Sci-kit learn: machine learning in Python [online]. [cit. 2019-06-16]. Retrieved from: https://scikit-learn.org/stable/

[59] SciPy.org [online]. [cit. 2019-06-24]. Retrieved from: https://www.scipy.org/

[60] Seaborn: statistical data visualization [online]. [cit. 2019-06-23]. Retrieved from: https://seaborn.pydata.org/

[61] SNOEK, Jasper, Hugo LAROCHELLE and Ryan ADAMS. Practical Bayesian Optimization of Machine Learning Algorithms [online]. 2012 [cit. 2019-06-23]. Retrieved from: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf

[62] StatsModels: Statistics in Python [online]. [cit. 2019-06-23]. Retrieved from: https://www.statsmodels.org/stable/index.html

[63] SUTSKEVER, Ilya. Training Recurrent Neural Networks [online]. 2013 [cit. 2019-02-28]. Retrieved from: https://www.cs.utoronto.ca/∼ilya/pubs/ilya_sutskever_phd_thesis.pdf. PhD thesis. University of Toronto.

[64] TensorFlow [online]. [cit. 2019-06-13]. Retrieved from: https://www.tensorflow.org/

[65] VIEIRA, Armando. Predicting online user behaviour using deep learning algorithms [online]. 27 May 2016 [cit. 2019-06-04]. Retrieved from: https://arxiv.org/pdf/1511.06247.pdf

[66] Welcome to Python.org [online]. [cit. 2019-06-16]. Retrieved from: https://www.python.org/

[67] WERBOS, Paul J. Generalization of backpropagation with application to a recurrent gas market model. Neural Networks [online]. 1988, 1(4), 339-356 [cit. 2019-05-11]. DOI: 10.1016/0893-6080(88)90007-X. ISSN 08936080. Retrieved from: https://linkinghub.elsevier.com/retrieve/pii/089360808890007X

[68] XIE, Paul. Practical Guide of RNN in Tensorflow and Keras [online]. In: Paul's Blog. 2017 [cit. 2019-06-04]. Dostupné z: https://paulx-cn.github.io/blog/4th_Blog/

A Backpropagation of LSTM

Before proceeding to backpropagation, Equation 3.10 is firstly modified, so that there is no concatenation. We split the weight matrices into two and add the flow to the output layer, see Equation A.1.

$$\begin{aligned} f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\ i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\ \tilde{C}_t &= tanh(W_C x_t + U_C h_{t-1} + b_C) \\ C_t &= f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \\ h_t &= o_t \circ tanh(C_t) \\ \hat{y}_t &= W_{hy} h_t + b_y \end{aligned} \tag{A.1}$$

If the matrix W_f formerly had size $H \times (H + dim_x)$, where dim_x is a length of the input vector for each example, now W_f has size $H \times dim_x$ and new matrix U_f has a size $H \times H$. We define a matrix θ containing all the matrices and biases (A.2).

$$\theta = \begin{bmatrix} W_f & U_f & b_f \\ W_i & U_i & b_i \\ W_o & U_o & b_o \\ W_C & U_C & b_C \end{bmatrix} \tag{A.2}$$

We denote the net input of the gates and candidates to the activation functions with a hat and a vector of these net inputs as z_t. Having a vector $I_t = [x_t, h_{t-1}, 1]$, we can rewrite the gate and input to the memory cell computations as in Equation A.3.

$$z_t = \begin{bmatrix} \hat{f}_t \\ \hat{i}_t \\ \hat{o}_t \\ \hat{\tilde{C}}_t \end{bmatrix} = \theta I_t = \begin{bmatrix} W_f & U_f & b_f \\ W_i & U_i & b_i \\ W_o & U_o & b_o \\ W_C & U_C & b_C \end{bmatrix} \begin{bmatrix} x_t \\ h_{t-1} \\ 1 \end{bmatrix} \tag{A.3}$$

Having L as our objective function, we first find gradients w.r.t. the weight matrix of our output layer, which is the same as in Equations 2.13, 2.14, 2.15 and 2.16 (considering we again have the model many-to-many where $N = M$). Then we denote the partial derivative of the hidden states as δh_t.

$$\delta h_t = \frac{\partial L}{\partial h_t} = \frac{\partial L}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} = \frac{\partial L}{\partial \hat{y}_t} \circ W_{hy} \tag{A.4}$$

In Equation A.4 only the gradient for the last step is considered,, using the same notation as in Section 2.2.1, h_M. At previous time steps the hidden state is influenced not only by the output layer, but also by the gates (going backward), as will be further explained in Equation A.15. Now the gradients w.r.t. the cell state and output gate can be defined.

$$\delta o_t = \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial o_t} = \delta h_t \circ tanh(C_t) \tag{A.5}$$

$$\delta C_t = \frac{\partial L}{\partial h_t}\frac{\partial h_t}{\partial C_t} = \delta h_t \circ o_t \circ (\mathbb{1} - tanh^2(C_t)) \tag{A.6}$$

Next we compute the gradients w.r.t. the input and forget gate, the candidates and the cell state from the previous step.

$$\delta i_t = \frac{\partial L}{\partial C_t}\frac{\partial C_t}{\partial i_t} = \delta C_t \circ \tilde{C}_t \tag{A.7}$$

$$\delta f_t = \frac{\partial L}{\partial C_t}\frac{\partial C_t}{\partial f_t} = \delta C_t \circ C_{t-1} \tag{A.8}$$

$$\delta \tilde{C}_t = \frac{\partial L}{\partial C_t}\frac{\partial C_t}{\partial \tilde{C}_t} = \delta C_t \circ i_t \tag{A.9}$$

$$\delta C_{t-1} += \frac{\partial L}{\partial C_t}\frac{\partial C_t}{\partial C_{t-1}} += \delta C_t \circ f_t \tag{A.10}$$

The operation $+=$ denotes that this gradient is added to the previous gradient from time step $(t+1)$. To get to the net inputs of the gates and candidates, we need to derive through the activation functions (the derivatives were shown in Equations 2.24 and 2.23.

$$\begin{aligned}
\delta \hat{f}_t &= \frac{\partial L}{\partial f_t}\frac{\partial f_t}{\partial \hat{f}_t} = \delta f_t \circ f_t(\mathbb{1} - f_t) \\
\delta \hat{i}_t &= \frac{\partial L}{\partial i_t}\frac{\partial i_t}{\partial \hat{i}_t} = \delta i_t \circ i_t(\mathbb{1} - i_t) \\
\delta \hat{o}_t &= \frac{\partial L}{\partial o_t}\frac{\partial o_t}{\partial \hat{o}_t} = \delta o_t \circ o_t(\mathbb{1} - o_t) \\
\delta \hat{\tilde{C}}_t &= \frac{\partial L}{\partial \tilde{C}_t}\frac{\partial \tilde{C}_t}{\partial\, \hat{\tilde{C}}_t} = \delta \tilde{C}_t \circ (\mathbb{1} - tanh^2(\tilde{C}_t)) \\
\delta \hat{z}_t &= \begin{bmatrix} \delta \hat{f}_t \\ \delta \hat{i}_t \\ \delta \hat{o}_t \\ \delta \hat{\tilde{C}}_t \end{bmatrix}
\end{aligned} \tag{A.11}$$

Equation A.11 gives us $\delta\hat{z}_t$. In Equation A.12 there is the gradient w.r.t. the parameters corresponding to the forget gate.

$$\begin{aligned}\frac{\partial L}{\partial W_f} &= \sum_{t=1}^{M} \delta\hat{f}_t x_t \\ \frac{\partial L}{\partial U_f} &= \sum_{t=1}^{M} \delta\hat{f}_t h_{t-1} \\ \frac{\partial L}{\partial b_f} &= \sum_{t=1}^{M} \delta\hat{f}_t\end{aligned} \tag{A.12}$$

Since these parameters are shared across the whole model, they need to be summed up over the time $t = 1, \ldots, M$. The computations for the rest of the parameters are very similar, thus we can use the notation given in Equation A.3.

$$\delta\theta = \frac{\partial L}{\partial \theta} = \sum_{t=1}^{M} \delta\hat{z}_t I_t = \sum_{t=1}^{M} \begin{bmatrix} \delta\hat{f}_t \\ \delta\hat{i}_t \\ \delta\hat{o}_t \\ \delta\hat{\tilde{C}}_t \end{bmatrix} \begin{bmatrix} x_t \\ h_{t-1} \\ 1 \end{bmatrix} \tag{A.13}$$

For brevity, to update the parameters, we use a general formula A.14 [16].

$$\theta := \theta - \eta \cdot \delta\theta \tag{A.14}$$

In Section 4.5, the notation $\delta\theta$ will be replaced by notation $\nabla_\theta L(\theta)$ to remind us, that the updates are computed w.r.t. gradient of the objective function L.

To derive the hidden state, we have two sources – the gates (see δh_{t-1}) and the objective function in Equation A.4.

$$\begin{aligned}\delta h_{t-1} &= f_t \circ (\mathbb{1} - f_t) U_f \delta o_f + i_t \circ (\mathbb{1} - i_t) U_i \delta i_t \\ &\quad + o_t \circ (\mathbb{1} - o_t) U_o \delta o_t + (\mathbb{1} - \tilde{C}_t^2) U_C \delta\tilde{C}_t \\ \delta h_{t-1} &= \delta\hat{f}_t U_f + \delta\hat{i}_t U_i + \delta\hat{o}_t U_o + \delta\hat{\tilde{C}}_t U_C \\ \frac{\partial L}{\partial h_{t-1}} &= \delta h_{t-1} + \frac{\partial L_{t-1}}{\partial h_{t-1}}\end{aligned} \tag{A.15}$$

A similar principle applies to the cell state as well, as already implied in Equation A.10. In this case, the sources are not only the objective function,

but also the next cell state, see Equation A.16.

$$\begin{aligned}
\delta C_{t-1} &= \frac{\partial L_{t-1}}{\partial C_{t-1}} = \frac{\partial L_{t-1}}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial C_{t-1}} \\
&= \frac{\partial L_{t-1}}{\partial \hat{y}_{t-1}} \circ W_{hy} \circ o_{t-1} \circ (\mathbb{1} - tanh^2(C_{t-1})) \\
\frac{\partial L_t}{\partial C_{t-1}} &= \frac{\partial L_t}{\partial h_t} \frac{\partial h_t}{\partial C_{t-1}} = \frac{\partial L_t}{\partial \hat{y}_t} \circ W_{hy} \circ o_t \circ (\mathbb{1} - tanh^2(C_t)) \circ f_t \\
\frac{\partial L}{\partial C_{t-1}} &= \delta C_{t-1} + f_t \circ \delta C_t
\end{aligned} \tag{A.16}$$

A Visualisations

Figure A.1: ECG – RNNAugmented: 1. part

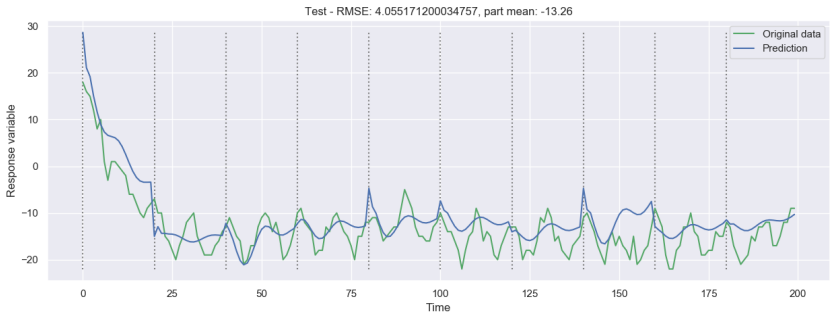


Figure A.2: ECG – RNNAugmented: 2. part

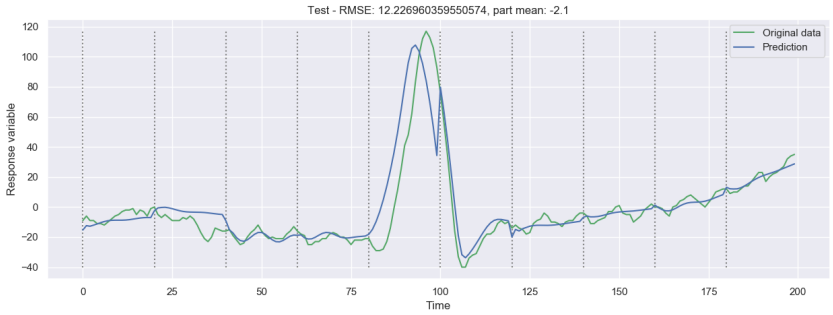


Figure A.3: ECG – RNNAugmented: 3. part

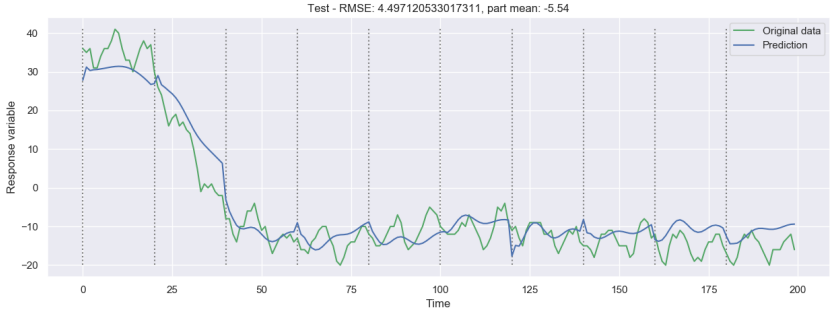


Figure A.4: ECG – RNNAugmented: 4. part

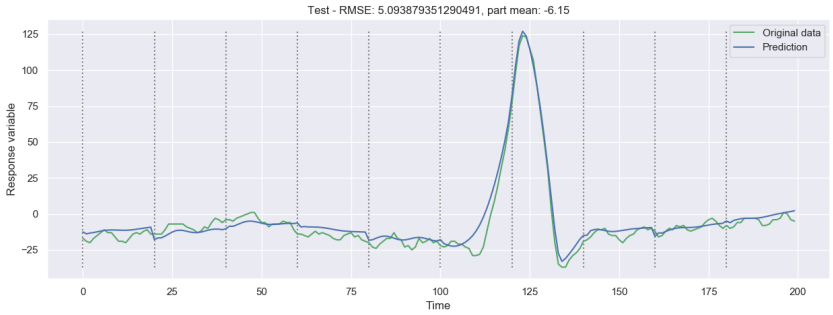


Figure A.5: ECG – RNNAugmented: 5. part

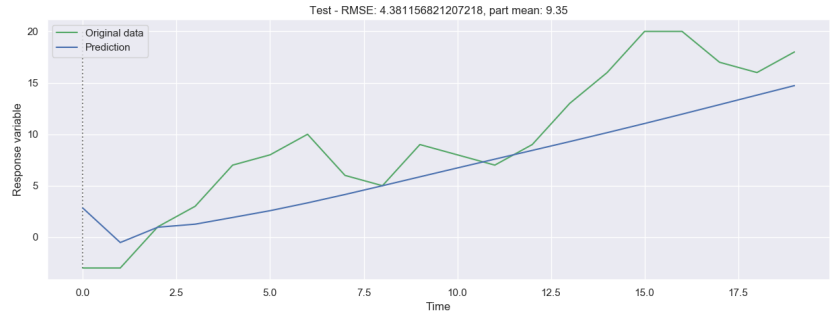


Figure A.6: ECG – baseline: 1. part

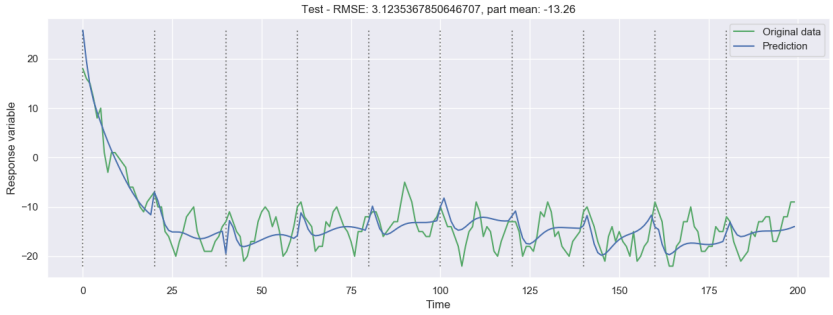


Figure A.7: ECG – baseline: 2. part

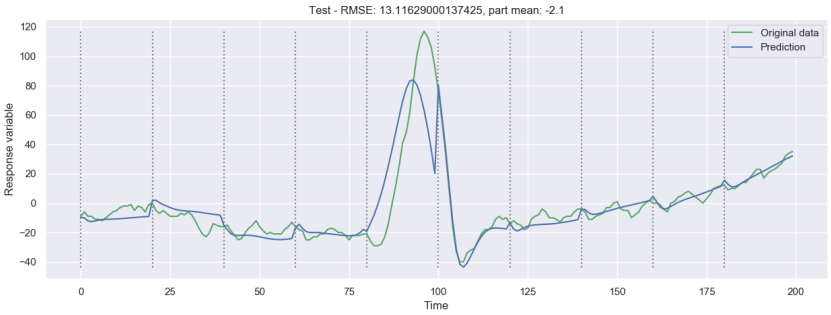


Figure A.8: ECG – baseline: 3. part

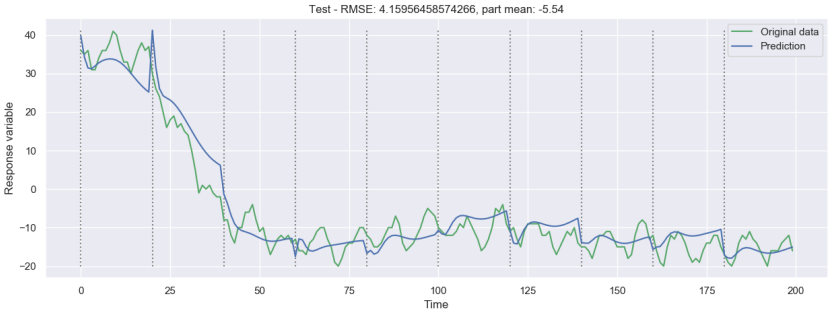


Figure A.9: ECG – baseline: 4. part

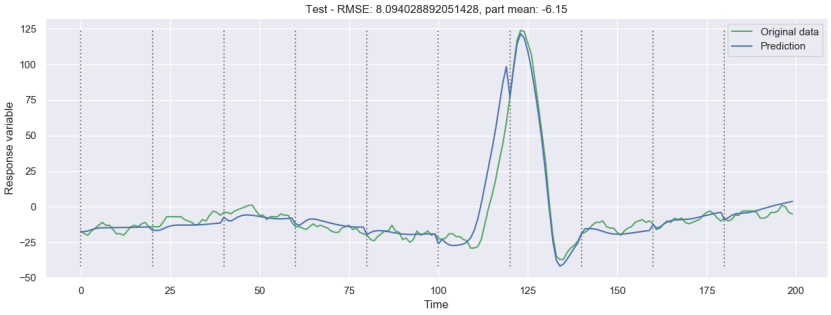


Figure A.10: ECG – baseline: 5. part

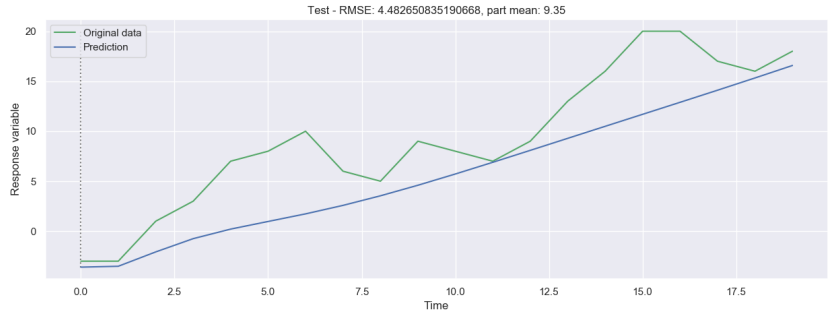


Figure A.11: ECG – LSTM Autoencoder: 1. part

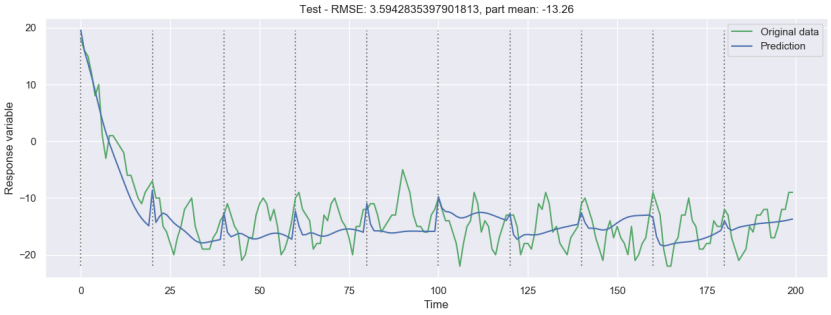


Figure A.12: ECG – LSTM Autoencoder: 2. part

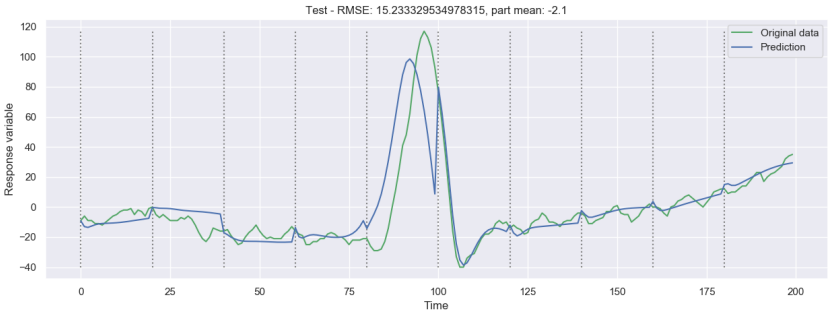


Figure A.13: ECG – LSTM Autoencoder: 3. part

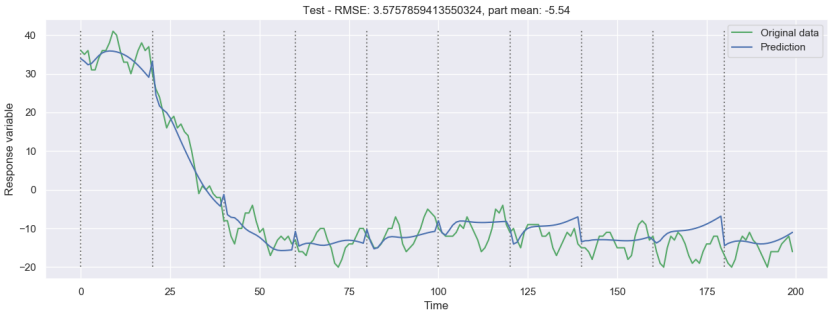


Figure A.14: ECG – LSTM Autoencoder: 4. part

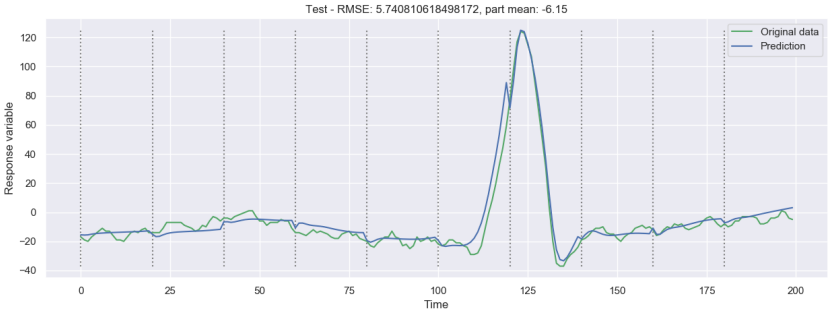
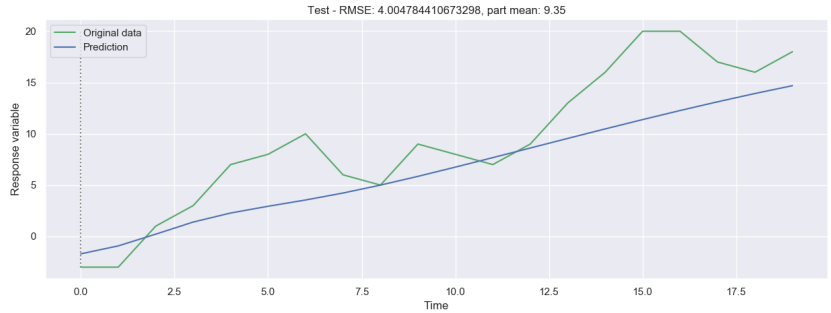


Figure A.15: ECG – LSTM Autoencoder: 5. part



List of Abbreviations

AE Autoencoder

AIC Akaike Information Criterion

ARIMAX Auto Regressive Integrated Moving Average with Exogeneous Input

ANN Artificial Neural Network

API Application Programming Interface

BPTT Backpropagation Through Time

CPU Central Processing Unit

FNN Feedforward Neural Network

GD Gradient Descent

GPU Graphics Processing Unit

GRU Gated Recurrent Unit

MAD Mean Absolute Deviation

MAE Mean Absolute Error

MSE Mean Squared Error

OLS Ordinary Least Squares

RMSE Root Mean Squared Error

RNN Recurrent Neural Network

SSE Sum of Squared Errors

SGD Stochastic Gradient Descent

TB TensorBoard

TF TensorFlow

List of Figures