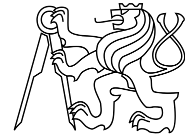
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF MASTER'S THESIS

Title:	Time Series Classification with Artificial Neural Networks
Student:	Bc. Jakub Waller
Supervisor:	Ing. Tomáš Borovi ka
Study Programme:	Informatics
Study Branch:	Knowledge Engineering
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2017/18

Instructions

Artificial Neural Networks are often used for time series modeling and classification. In the last decade, advanced architectures such as deep neural networks or neural networks with memory became popular, because computational power is more widely available. Convolutional neural networks or long-short term memory networks are examples successfully applied on time series modeling and classification.

1) Review and theoretically describe different state-of-the-art ANN architectures suitable for univariate as well as multivariate time series classification.
2) Propose an experiment design in order to compare their ability to learn, effectivity of the training process and classification performance.
3) Compare different architectures on a set of benchmark datasets.

References

Will be provided by the supervisor.

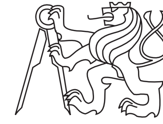
doc. Ing. Jan Janoušek, Ph.D. Head of Department	prof. Ing. Pavel Tvrdík, CSc. Dean

Prague February 15, 2017

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science



Master's thesis

Time Series Classification with Artificial Neural Networks

Bc. Jakub Waller

Supervisor: Ing. Tomáš Borovička

9th May 2017

Acknowledgements

I would like to thank my supervisor Ing. Tomáš Borovička for his valuable advice and inspiring guidance. I am also very grateful to the Data Science Laboratory at CTU FIT and Datamole for their support. Finally, I wish to thank all precious people close to me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 9th May 2017 . . . .

Czech Technical University in Prague

Faculty of Information Technology

© 2017 Jakub Waller. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Waller, Jakub. *Time Series Classification with Artificial Neural Networks.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Mnoho různých architektur umělých neuronových sítí bylo navrženo, kupříkladu konvoluční neuronové sítě a long short-term memory neuronové sítě. Cílem této práce je aplikovat tyto sítě na klasifikaci časových řad. Po teoretickém popisu těchto architektur je navržena metoda pro jejich experimentální porovnání, a ta je následně implementována v Pythonu. Tato metoda zahrnuje automatickou optimalizaci hyperparametrů neuronových sítí. Popsané architektury jsou poté důkladně porovnány na třech benchmarkových datasetech. Toto porovnání ukazuje, že long short-term memory neuronové sítě dosahují na dvou ze tří datasetů lepších výsledků než konvoluční neuronové sítě.

Klíčová slova konvoluční neuronové sítě, rekurentní neuronové sítě, long short-term memory neuronové sítě, hluboké učení, optimalizace hyperparametrů, grid search, random search, klasifikace časových řad

Abstract

Various architectures of artificial neural networks have been developed such as convolutional neural networks and long short-term memory neural networks. The aim of this thesis is to apply these networks on the classification of time series. After a theoretical description of the architectures, an experimental procedure for their comparison is proposed and implemented in Python. The procedure includes automatic optimisation of neural network's hyperparameters. Based on this procedure, the architectures are thoroughly compared on three benchmark data sets. The comparison shows that long short-term memory neural networks achieve better results than convolutional neural networks on two of these data sets.

Keywords convolutional neural networks, recurrent neural networks, long short-term memory neural networks, deep learning, hyperparameters optimisation, grid search, random search, time series classification

Contents

Citation of this thesis viii

Introduction **1**

1 Artificial Neural Networks **3**

1.1 Feedforward Neural Networks 3

1.1.1 Structure 3

1.1.2 Neurone 4

1.1.3 Learning 5

1.2 Convolutional Neural Networks 9

1.2.1 Structure 9

1.3 Recurrent Neural Networks 11

1.3.1 Structure 12

1.3.2 Learning 13

1.3.3 Vanishing Gradient Problem 14

1.4 Long Short-Term Memory Neural Networks 14

1.4.1 Hidden Unit 15

1.4.2 Peephole Connections 16

2 Implementation **19**

2.1 Technical Details 19

2.2 Implementation Details 20

3 Experiments **23**

3.1 Experimental Design 23

3.1.1 Process of the Experiment 23

3.1.2 Hyperparameters 26

3.2 Benchmark Data Sets 30

3.2.1 Time Series 30

3.2.2 Classification 30

3.2.3 Data Sets 31

3.3 Comparison of Architectures 33
3.3.1 Gun-Point 33
3.3.2 Strawberry 40
3.3.3 Japanese Vowels 45

Conclusion **55**

Bibliography **57**

A Acronyms **61**

B Contents of enclosed CD **63**

List of Figures

1.1 Structure of a simple 3-layer neural network. 4
1.2 Artificial neurone. 5
1.3 A simple network with two input nodes, one output neurone, and a node representing the error function E. 7
1.4 Scheme of a convolutional neural network with $B = 2, C = 2$, and $D = 1$. 10
1.5 The processing of two regions by one filter of size three, with zero-padding equal to zero and stride equal to one. 11
1.6 Recurrent neural network unfolded into three time-steps with a sequential output. 12
1.7 Recurrent neural network unfolded into three time-steps with a non-sequential output. 13
1.8 The sigmoid function and its derivative. 14
1.9 LSTM's hidden unit. 15
1.10 Adding peephole connections to LSTM's hidden unit. 17

3.1 A graphical schema of the experimental procedure. 25
3.2 An example of the *Gun-Draw* class from the Gun-Point data set. . 31
3.3 An example of the *Point* class from the Gun-Point data set. 31
3.4 An example of the *Strawberry* class from the Strawberry data set. 32
3.5 The Japanese Vowels data set. 33
3.6 The classification performance in dependency on the number of neurones and the number of layers for CNN on the Gun-Point data set. 35
3.7 The classification performance in dependency on the number of neurones and the number of layers for LSTM on the Gun-Point data set. 35
3.8 The classification performance in dependency on the number of neurones and the number of layers for RNN on the Gun-Point data set. 36

3.9 Time of the training (same scale) in dependency on the number of neurones and the number of layers for CNN on the Gun-Point data set. 37
3.10 Time of the training (different scale) in dependency on the number of neurones and the number of layers for CNN on the Gun-Point data set. 37
3.11 Time of the training in dependency on the number of neurones and the number of layers for LSTM on the Gun-Point data set. 37
3.12 Time of the training in dependency on the number of neurones and the number of layers for RNN on the Gun-Point data set. 37
3.13 Comparison of the effectivity of the training process on the Gun-Point data set. 38
3.14 Comparison of the classification performance for the testing subset of the Gun-Point data set. 39
3.15 Comparison of F-score for the Gun-Draw class of the Gun-Point data set. 39
3.16 Comparison of F-score for the Point class of the Gun-Point data set. 39
3.17 The classification performance in dependency on the number of neurones and the number of layers for CNN on the Strawberry data set. 41
3.18 The classification performance in dependency on the number of neurones and the number of layers for LSTM on the Strawberry data set. 42
3.19 The classification performance in dependency on the number of neurones and the number of layers for RNN on the Strawberry data set. 43
3.20 Time of the training (same scale) in dependency on the number of neurones and the number of layers for CNN on the Strawberry data set. 43
3.21 Time of the training (different scale) in dependency on the number of neurones and the number of layers for CNN on the Strawberry data set. 43
3.22 Time of the training in dependency on the number of neurones and the number of layers for LSTM on the Strawberry data set. 44
3.23 Time of the training in dependency on the number of neurones and the number of layers for RNN on the Strawberry data set. 44
3.24 Comparison of the effectivity of the training process on the Strawberry data set. 44
3.25 Comparison of the classification performance for the testing subset of the Strawberry data set. 45
3.26 Comparison of F-score for the Strawberry class of the Strawberry data set. 46
3.27 Comparison of F-score for the Non-Strawberry class of the Strawberry data set. 46

3.28 The classification performance in dependency on the number of neurones and the number of layers for CNN on the Japanese Vowels data set. 47
3.29 The classification performance in dependency on the number of neurones and the number of layers for LSTM on the Japanese Vowels data set. 47
3.30 The classification performance in dependency on the number of neurones and the number of layers for RNN on the Japanese Vowels data set. 49
3.31 Time of the training (same scale) in dependency on the number of neurones and the number of layers for CNN on the Japanese Vowels data set. 49
3.32 Time of the training (different scale) in dependency on the number of neurones and the number of layers for CNN on the Japanese Vowels data set. 49
3.33 Time of the training in dependency on the number of neurones and the number of layers for LSTM on the Japanese Vowels data set. . 50
3.34 Time of the training in dependency on the number of neurones and the number of layers for RNN on the Japanese Vowels data set. . . 50
3.35 Comparison of the effectivity of the training process on the Japanese Vowels data set. 50
3.36 Comparison of the classification performance for the testing subset of the Japanese Vowels data set. 51
3.37 Comparison of F-score for the Speaker 2 class of the Japanese Vowels data set. 52
3.38 Comparison of F-score for the Speaker 9 class of the Japanese Vowels data set. 52

List of Tables

3.1 Structures of convolutional neural networks, defined using hyperparameters B, C, and D. . . . 24
3.2 Optimised hyperparameters for the Gun-Point data set. . . . 34
3.3 Comparison of the effectivity of the training process on the Gun-Point data set. . . . 38
3.4 Comparison of the classification performance for the testing subset of the Gun-Point data set. . . . 38
3.5 Comparison of precision separately for classes of the Gun-Point data set. . . . 40
3.6 Comparison of recall separately for classes of the Gun-Point data set. . . . 40
3.7 Comparison of F-score separately for Classes of the Gun-Point data set. . . . 40
3.8 Optimised Hyperparameters for the Strawberry data set. . . . 41
3.9 Comparison of the effectivity of the training process on the Strawberry data set. . . . 45
3.10 Comparison of the classification performance for the testing subset of the Strawberry data set. . . . 45
3.11 Comparison of precision separately for classes of the Strawberry data set. . . . 46
3.12 Comparison of recall separately for classes of the Strawberry data set. . . . 46
3.13 Comparison of F-score separately for Classes of the Strawberry data set. . . . 48
3.14 Optimised Hyperparameters for the Japanese Vowels data set. . . . 48
3.15 Comparison of the effectivity of the training process on the Japanese Vowels data set. . . . 51
3.16 Comparison of the classification performance for the testing subset of the Japanese Vowels data set. . . . 51

3.17 Comparison of precision separately for classes of the Japanese Vowels data set. 52
3.18 Comparison of recall separately for classes of the Japanese Vowels data set. 52
3.19 Comparison of F-score separately for Classes of the Japanese Vowels data set. 53

Introduction

Artificial neural networks, a group of machine learning algorithms based on the principles of the human brain, are nowadays used in a wide variety of domains. State-of-the-art architectures have been improving the progress in computer vision, speech recognition, natural language processing (e.g. machine translation), recommender systems, and games (a recent example is Go [1]). Furthermore, neural networks[1] are successfully used for finding new drug compounds [2], improving energy efficiency in datacentres [3], image denoising and inpainting [4], super-resolution [5], text generating (including handwriting) [6] as well as for developing new encryption methods [7].

Many of the domains, where neural networks are used, contain data sets comprised of time series, sets of sequentially collected observations. The time series can be divided into several disjoint classes. One of the endeavours of machine learning is the classification of time series into correct classes. To accomplish this task with neural networks, it is necessary to select a proper neural network architecture. It is further needed to optimise hyperparameters of the selected neural network, which are higher-level properties considerably influencing the classification performance of the network.

Accomplishing the classification of time series with neural networks is the goal of this thesis. After theoretically introducing artificial neural networks, the work proposes and implements an experimental procedure for comparing different architectures including automatic optimisation of neural network's hyperparameters. Based on this procedure, two state-of-the-art architectures and one baseline architecture are compared on three benchmark data sets.

The thesis is divided into three chapters with several sections and subsec-

[1] For simplification, here and further in the thesis, by neural networks it is meant artificial neural networks.

tions. Chapter 1 theoretically describes four architectures of artificial neural networks: feedforward neural networks, convolutional neural networks, recurrent neural networks, and long short-term memory neural networks. The description includes the structures of the networks, their building blocks—artificial neurones, and learning algorithms. Chapter 2 provides the transition from the theoretical part to the experimental part, describing implementation details, including architecture, programming language, libraries, and specific settings of the implemented algorithms. Chapter 3 proposes an experimental design for optimising neural network's hyperparameters and comparing different architectures of neural networks. It introduces three benchmark data sets and compares three of the architectures (recurrent, convolutional, and long short-term memory neural networks) on these data sets.

Chapter **1**

Artificial Neural Networks

The first chapter of this thesis, divided into four sections, focuses on artificial neural networks. Their principles are explained on feedforward neural networks in Section 1.1. These networks can be extended into a state-of-the-art architecture—convolutional neural networks, described in Section 1.2. The next section defines neural networks with memory called recurrent neural networks. Lastly, Section 1.4 outlines an improvement of recurrent neural networks: long short-term memory neural networks.

1.1 Feedforward Neural Networks

A feedforward neural network is a directed computational graph whose nodes, arranged into vertical layers, represent simple functions. Edges of the graph connect outputs of nodes in the i-th layer to inputs of nodes in the $(i+1)$-th layer. The network, therefore, represents a chain of functions and can be called the network function [8]. The following parts describe the structure of neural networks and the backpropagation algorithm used for their learning.

1.1.1 Structure

Each neural network consists of at least three layers: one input layer, one or more hidden layers, and one output layer. An example of a traditional neural network with three layers is shown in Figure 1.1. The following paragraph looks at the three types of layers (input, hidden, and output) in more detail.

The purpose of the input layer is to distribute all components of an input vector into all nodes of the first hidden layer. The number of nodes in the input layer equals to the length of the input vector. The hidden layers together with the output layer build the core of neural networks. Through the learning process (also called the training process), described later in Section 1.1.3,

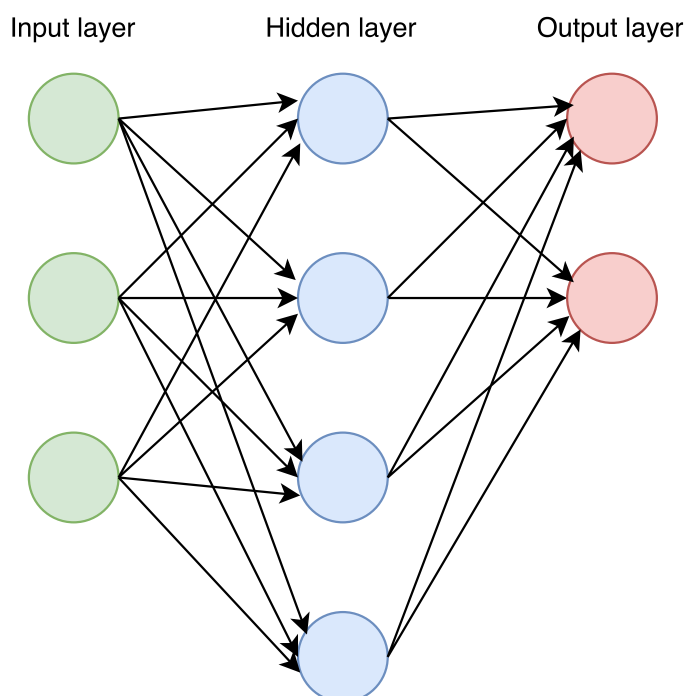


Figure 1.1: Structure of a simple 3-layer neural network.

they learn to solve a certain problem.[2] The number of hidden layers, as well as the number of nodes in these layers, are one of the several hyperparameters which have to be set and which usually depend on a particular problem. With increasing number of hidden nodes, artificial neural networks can tackle more complex data, but they are more difficult to train and more prone to overfitting. In this case, a neural network appropriately learns the data it is trained on but cannot generalise for previously unseen data, called the testing set. The output layer, connected to the last hidden layer, produces an output vector for each input vector. The number of nodes in the output layer equals to the length of the output vector.

1.1.2 Neurone

The nodes in the hidden layers and the output layer are artificial neurones. A scheme of such a neurone[3] is depicted in Figure 1.2. The green nodes $x_0, x_1, \ldots, x_n$ are neurone's inputs. For neurones in the first hidden layer, the inputs represent an input vector. For neurones in other hidden layers or in the output layer, the inputs represent outputs of neurones in the previous

[2] There are attempts to solve more problems at once by using only one neural network. For further information see for example article Progressive Neural Networks [9].

[3] For simplification, "neurone" means "artificial neurone".

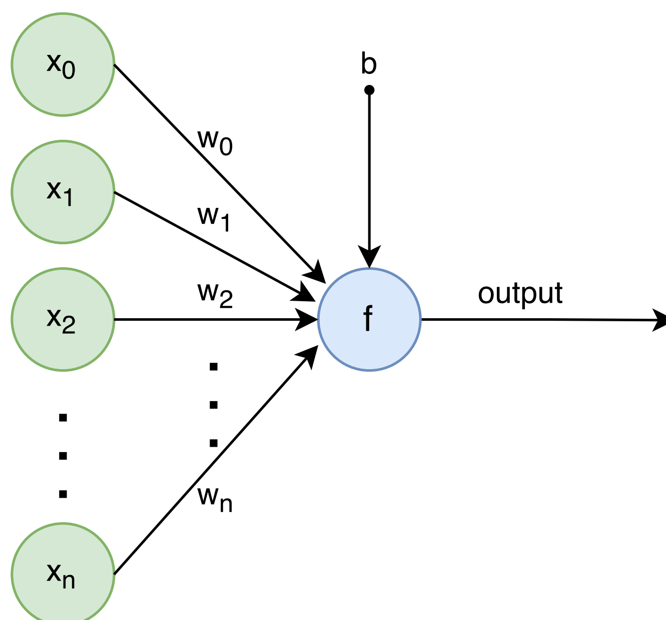


Figure 1.2: Artificial neurone.

layer. Each input x_i has its weight w_i. The weights and bias b (called also threshold; bias $=$ $-$threshold) are parameters of the neural network. These parameters are learnt during the learning process, described in the following section. The weighted inputs are summed, the bias is added and the result is put into the activation function $f(\sum_{i=0}^{n} w_i \cdot x_i + b)$, represented by the blue node. The common choice for the activation function is the sigmoid function $S(x)$ or the hyperbolic tangent $\tanh(x)$, which transform the input space into intervals $(0, 1)$ or $(-1, 1)$, respectively:

$$S(x) = \frac{1}{1 + e^{-x}}$$
$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

1.1.3 Learning

The weights and biases, discussed in the previous section, must be appropriately adjusted to train the neural network. Let $\{(\mathbf{x_1}, \mathbf{y_1}), (\mathbf{x_2}, \mathbf{y_2}), \ldots, (\mathbf{x_m}, \mathbf{y_m})\}$ be a training set where $\mathbf{x_i}$ is an input vector and $\mathbf{y_i}$ is an output vector. Let g be a given function such that

$$g(\mathbf{x_i}) = \mathbf{y_i} \tag{1.1}$$

for all $i \in \langle 1, m \rangle$. The learning problem consists of optimising the network weights and biases in a way that the network function λ approximates the function g. That is, the learning problem consists of minimising the error function[4], defined as

$$E = \frac{1}{2} \sum_{i=1}^{m} ||\mathbf{o_i} - \mathbf{y_i}||^2, \tag{1.2}$$

where $\mathbf{o_i}$ are the output vectors of the network, i.e. $\lambda(\mathbf{x_i}) = \mathbf{o_i}$ for all $i \in \langle 1, m \rangle$. When introducing new unknown input vectors to the network, it is expected to interpolate and to produce output vectors approximating outputs of the function g.

Backpropagation, used as the learning algorithm to minimise the error function, is an iterative process repeating four steps:

1. forward propagation,
2. calculating the error function,
3. backward propagation,
4. updating the weights[5].

There are three types of the backpropagation algorithm, depending on the number of input vectors used in one iteration. The on-line gradient descent uses only one randomly selected input vector. The off-line (also called full batch) gradient descent uses all inputs from the training set. And the mini-batch gradient descent uses a randomly selected subset of the training data.

The following paragraphs first outline the iteration process of the on-line gradient descent and later generalise it for the off-line and mini-batch gradient descents. During the first phase of the backpropagation algorithm, an input vector is propagated from the input layer through all hidden layers to the output layer. In the second phase, the error function (1.2) is calculated. Subsequently, the error is propagated backwards through the network to analytically obtain the gradient of the error function with respect to all weights

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_r} \right). \tag{1.3}$$

[4]Called also the loss function, cost function or objective function. This particular error function is the mean squared error.

[5]To simplify the notation used in this section, the biases are represented by extra weights connected to a node with output value 1.

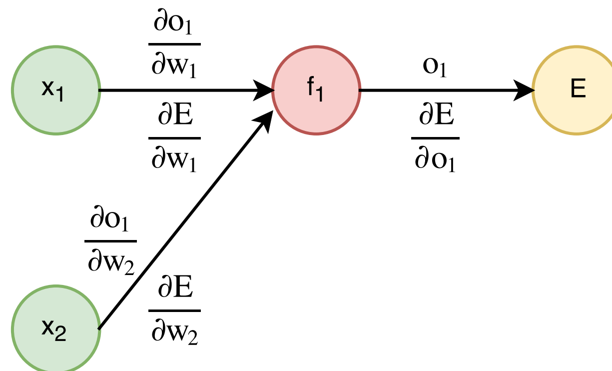


Figure 1.3: A simple network with two input nodes, one output neurone, and a node representing the error function E.

The simple network in Figure 1.3 illustrates the calculating of the partial derivatives.[6] This network has two inputs x_1, x_2, and one output neurone f_1. The node E represents the error function. During the forward propagation, the following is calculated for each neurone in all hidden and output layers of a neural network:

1. the output of the neurone,
2. partial derivatives of the output with respect to weights of all inputs connected to the neurone.

For the neurone f_1 in Figure 1.3 the result (pictured above the edges) is

1. output o_1,
2. $\dfrac{\partial o_1}{\partial w_1}$ for the input x_1, and $\dfrac{\partial o_1}{\partial w_2}$ for the input x_2.

The error function E and the partial derivative of the error function with respect to the output of the output neurone $\dfrac{\partial E}{\partial o_1}$ is calculated in the second phase. During the backward propagation (pictured below the edges in Figure 1.3), the partial derivatives of the error function with respect to the neurone's weights are computed using the chain rule:

[6] For a detailed explanation see, for example, Rojas's Neural Networks [8].

1. $\dfrac{\partial E}{\partial w_1} = \dfrac{\partial E}{\partial o_1} \cdot \dfrac{\partial o_1}{\partial w_1}$,
2. $\dfrac{\partial E}{\partial w_2} = \dfrac{\partial E}{\partial o_1} \cdot \dfrac{\partial o_1}{\partial w_2}$.

In the case of a network with more layers, the chain rule is applied iteratively until the partial derivatives of neurones in all layers are calculated, resulting in the gradient in Formula (1.3).

The partial gradients can be as well calculated numerically, using the limit definition of a derivative

$$\frac{\partial E}{\partial w_i} = \lim_{h \to 0} \frac{E(w_i + h) - E(w_i - h)}{2h},$$

where $E(w_i + h)$ is the error function for the same network in which only one weight w_i is changed. This numerical gradient is calculated for a few weights and the result is compared to the analytical gradient. If there is no substantial difference, the gradient is used for updating the weights using the increment

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i} \text{ for all } i \in \langle 1, r \rangle, \tag{1.4}$$

where γ is the learning rate influencing the speed of the learning process. A small learning rate means small increments, and usually slower learning. On the other hand, a large learning rate makes the network improve faster, but there is a higher probability of overlooking the optimum. A too large[7] learning rate can result in divergence instead of convergence of the error function.

The learning process of the on-line gradient descent, described above, is applicable to the off-line and mini-batch gradient descents after changes to the weights increments which look as follows,

$$\Delta w_i = \Delta_1 w_i + \Delta_2 w_i + \cdots + \Delta_s w_i.$$

$\Delta_j w_i$ is the increment calculated for the input vector $\mathbf{x_j}$. The number of increments s equals the size of the training set m for the off-line gradient descent, i.e. every input vector is used. For the mini-batch gradient descent, the number of increments $1 < s < m$ represents the size of the batch.

The backpropagation algorithm is repeated as long as the error (the training loss) decreases. These repetitions are counted in epochs. During one epoch, every input vector is fed into the neural network once. The number of epochs largely influences the performance of the network as well as the

[7] The definition of "too large" depends on a particular data set.

duration of the learning process. Therefore, it is important to interrupt the training of the network when the training loss has stopped decreasing. A common solution is to implement the early stopping algorithm. This algorithm interrupts the learning process if the training loss has not improved for a specified number of epochs. To prevent overfitting, the training data are split into a training set and a validation set. The backpropagation algorithm uses only the training set to adjust the network weights. At the end of each epoch, the validation loss is calculated using the validation set. The early stopping algorithm then compares the validation loss instead of the training loss.

The previous three sections described the structure of feedforward neural networks, their building blocks—artificial neurones, and the backpropagation algorithm used for training the networks. Various extensions and improvements of this architecture have been developed, such as convolutional neural networks, outlined in the following sections.

1.2 Convolutional Neural Networks

A common machine learning problem using neural networks can be separated into two parts. First, a hand-designed feature extractor selects the most relevant and important features from the data set. Second, a feedforward neural network is employed for a machine learning task using these features. Convolutional neural networks, described in this section, have a specifically designed architecture enabling an automatic extraction of the features.[8]

1.2.1 Structure

The structure of convolutional neural networks consists of three main building blocks. Convolutional layers (CONV), pooling layers (POOL), and fully-connected layers (FC). The composition of these layers commonly follows this pattern

$$INPUT \rightarrow [CONV * B \rightarrow POOL?] * C \rightarrow FC * D \rightarrow FC,$$

where $INPUT$ is the input layer, the last FC is the output layer, and the rest of the layers are the hidden layers. Symbol “$*$” indicates repetition, symbol “?” indicates an optional addition of a layer preceding this symbol, and symbol “$\rightarrow$” represents edges between layers. $B, C, D \geq 0$ are adjustable hyperparameters. Figure 1.4 shows a scheme of a convolutional network, where $B = 2, C = 2$, and $D = 1$. The input layer is pictured in green colour, the

[8]Moreover, convolutional neural networks give better results for inputs with a local structure (such as time series or images). [10] Object recognition in images is probably the machine learning field where the success of convolutional neural networks is best known. For further information, see, for instance, the article ImageNet Classification with Deep Convolutional Neural Networks [11].

hidden layers are blue (four convolutional layers, two pooling layers and one fully-connected layer), and the output layer is depicted in red colour. The following passages identify and explain in detail the three types of the hidden layers.

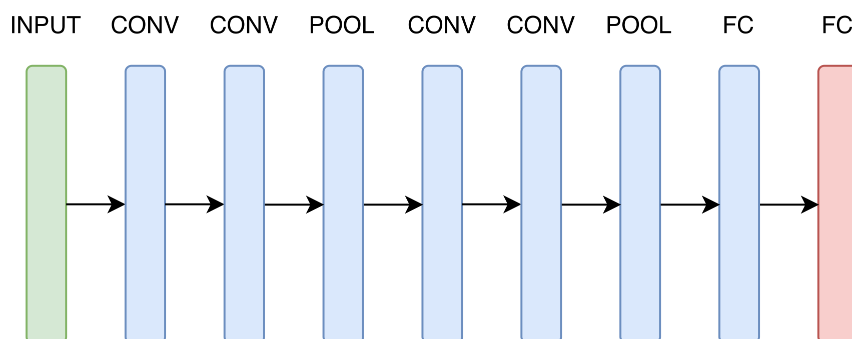


Figure 1.4: Scheme of a convolutional neural network with $B = 2, C = 2$, and $D = 1$.

Convolutional Layer

A convolutional layer consists of several filters (also called local receptive fields) which can extract elementary features from an input vector. The number of these filters is a hyperparameter of the network. Each filter is a neurone, as described in Section 1.1.2, connected to regions of the input vector. One such filter (labeled f) is depicted in Figure 1.5. The input vector (green nodes) of size four is divided into two overlapping regions. The first region (on the left side of the Figure 1.5) contains input nodes x_1, x_2, and x_3 and produces the output o_1. Input nodes x_2, x_3, and x_4, producing the output o_2, belong to the second region (on the right side of the Figure 1.5). For both regions, the filter's weights w_1, w_2, and w_3 and bias b stay the same.

There are three hyperparameters specific for the convolutional layer: the filter size, the stride, and zero-padding. The filter size defines the size of the regions. The number of different nodes between two consecutive regions is determined by the stride. Lastly, zero-padding specifies the number of zero nodes (nodes with value zero) added to the edges of the input vector. Without these zero nodes, the edges of the input vector are processed fewer times than the rest of the input. In Figure 1.5, $filter\ size = 3, stride = 1, zero\text{-}padding = 0$.

Pooling Layer

Outputs of the filters form so-called feature maps. A pooling layer, usually placed after a convolutional layer, reduces the dimension of the feature maps

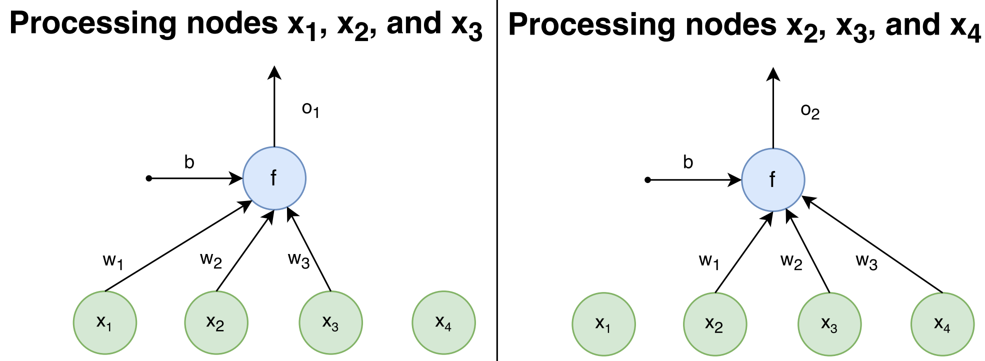


Figure 1.5: The processing of two regions by one filter of size three, with zero-padding equal to zero and stride equal to one.

to make the network less prone to overfitting and to reduce the computational complexity. Pooling depends on two hyperparameters: the size of the pooling windows, and the stride, which is the downsampling factor. Two types of pooling are commonly used: average pooling and max pooling. The average pooling computes the average of its inputs and applies a non-linear function to this average. An output of the max pooling is the maximum of its inputs. Scherer et al. [12] empirically show that the max pooling significantly outperforms the average pooling.

Fully-connected Layer

The extracted and down-sampled features are forwarded into one or more fully-connected layers. Neurones in such a layer are connected to all outputs of the neurones in the previous layer, as in the feedforward neural networks. By using these fully-connected layers, the neural network assigns an output vector to every input vector (such as a class in a classification task).

1.3 Recurrent Neural Networks

Feedforward neural networks make an assumption that all inputs are independent of each other. This is not an invalid assumption for most of the data sets, but seemingly not for all of them. Time series classification is an example of a machine learning problem using sequential inputs which are dependent on each other. Recurrent neural networks, also called neural networks with memory, however, are suitable for such problems due to their specific structure. The following part analyses the structure and the learning algorithm: backpropagation through time.

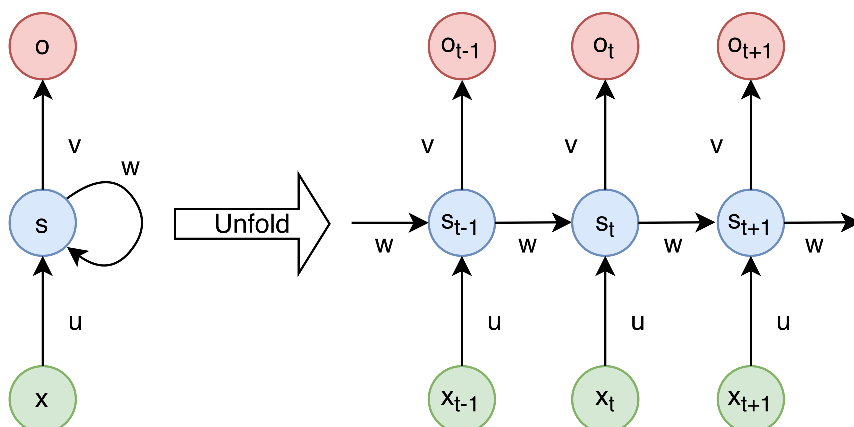


Figure 1.6: Recurrent neural network unfolded into three time-steps with a sequential output.

1.3.1 Structure

The structure of a recurrent neural network stays the same as described in Section 1.1.1. The only difference is that the network is evaluated more times in a row for all sequential components of an input (such as time points in a time series). Figure 1.6 demonstrates this process using a simple network with only one hidden neurone and a one-dimensional output. The left part of the Figure 1.6 shows the structure of the recurrent neural network. The node x is an input, the node s is the hidden state, and the node o is the network's output. The information about the hidden state s is forwarded in a loop back to the hidden state. Similarly to feedforward neural networks, weights u, v and w, and bias b are parameters adjusted during the learning process. An advantage of recurrent neural networks is that all weights remain the same for the full sequential input. This significantly reduces the number of parameters as opposed to feedforward networks where weights change with every input.

The right part of the Figure 1.6 shows three time-steps of evaluating the network. In the time-step t, the hidden state s_t is calculated using the input x_t and the previous hidden state s_{t-1} as

$$s_t = f(u \cdot x_t + w \cdot s_{t-1} + b),$$

where f is the activation function discussed in Section 1.1.2. The output of the hidden state s_t is forwarded into the input of the next hidden state s_{t+1}. This enables the network to process previous information.

Figure 1.6 depicts a network where both input and output are sequential. Examples of usage include text generating [13] and machine translation [14,

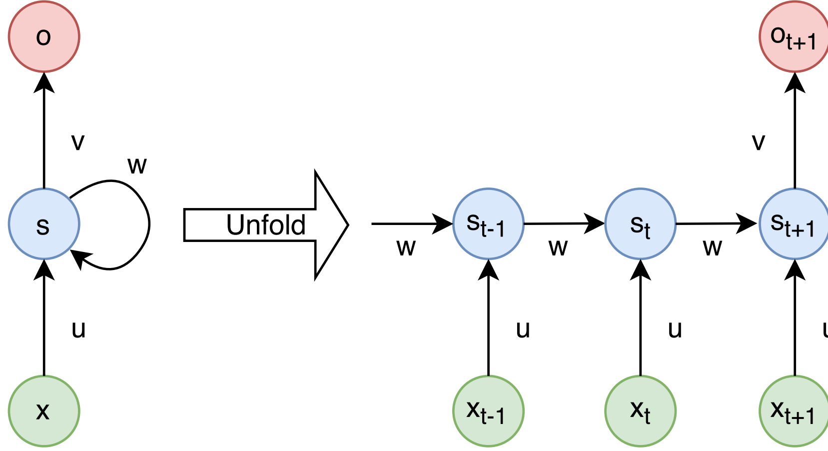


Figure 1.7: Recurrent neural network unfolded into three time-steps with a non-sequential output.

15]. However, there are machine learning problems where a non-sequential output is needed, such as time series classification. This introduces a small change into evaluating recurrent neural networks, pictured in Figure 1.7. The hidden state s (in the left part of the Figure 1.7) produces the output o only for the last input (time-point) x. This process is depicted in the right part of the Figure 1.7, where hidden states s_{t-1}, and s_t do not produce any outputs; the only output o_{t+1} is given by the hidden state s_{t+1}.

1.3.2 Learning

This section introduces the learning algorithm of recurrent neural networks, called backpropagation through time. This algorithm is very similar to back-propagation described in Section 1.1.3. The error function is defined as

$$E = \sum_{t=0}^{l} E_t, \tag{1.5}$$

where l is the length of the output sequence and

$$E_t = \frac{1}{2}||o_t - y_t||^2.$$

$(y_1, y_2, \ldots, y_l)$ is the true output and $(o_1, o_2, \ldots, o_l)$ is the output produced by the network. For simplification, the error function of the on-line gradient descent is used, which contains only one output vector. The aim is to calculate the gradient of the error function with respect to all weights (such as in Equation (1.3)), where

$$\{u, v, w\} \in \{w_i\}.$$

Partial derivatives are then calculated as

$$\frac{\partial E}{\partial w_i} = \sum_{t=0}^{l} \frac{\partial E_t}{\partial w_i}.$$

Considering a part of the unfolded network from Figure 1.6 for time-steps $\{0, 1, \ldots, t\}$, the expression $\frac{\partial E_t}{\partial w_i}$ is calculated as in the standard backpropagation for feedforward networks.

1.3.3 Vanishing Gradient Problem

Backpropagation through time, discussed above, has a limitation called the vanishing gradient problem. Figure 1.8 pictures the sigmoid function and its derivative. It can be seen that the limits of the derivative approach zero at both ends. Multiplication of such small derivatives in the chain rule causes gradients at later time-steps to vanish.[9] This implies an inability of recurrent neural networks to learn long-term dependencies in sequential inputs. One solution to this problem is using rectified linear units (ReLUs) [11], defined as $f(x) = \max(0, x)$, instead of the sigmoid or hyperbolic tangent activation functions. Another approach is offered by LSTMs, further described in the following section.

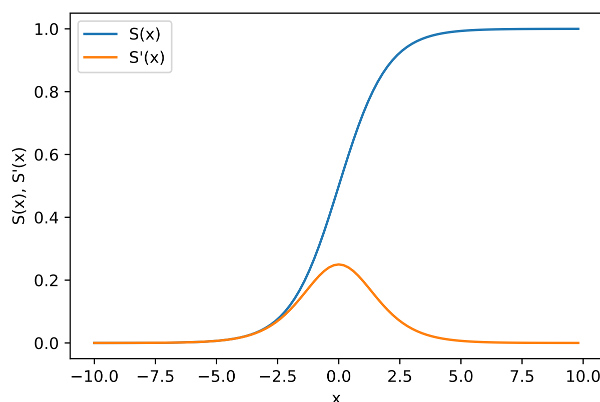


Figure 1.8: The sigmoid function and its derivative.

1.4 Long Short-Term Memory Neural Networks

LSTMs were originally introduced by Hochreiter and Schmidhuber [17], and further improved by Gers, Schmidhuber, and Cummins [18]. Recurrent neural

[9] For a detailed description see for example article [16].

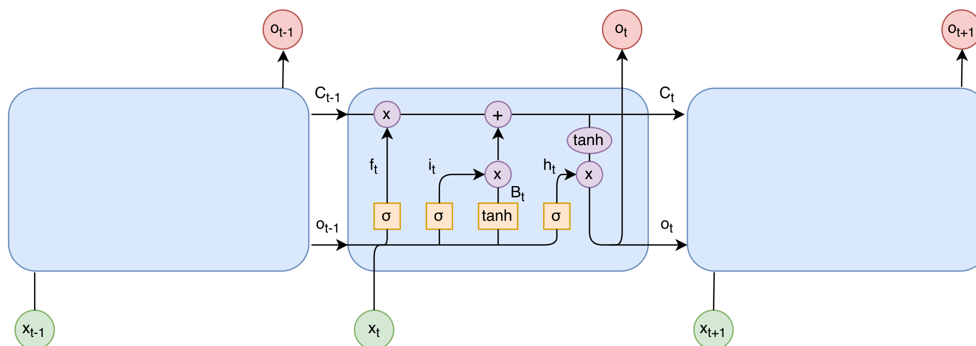


Figure 1.9: LSTM's hidden unit.

networks with LSTMs share the same structure as well as the learning algorithm (backpropagation through time); the difference is in the hidden units, explained in the following passage.

1.4.1 Hidden Unit

A hidden unit of LSTMs is depicted in Figure 1.9. It consists of four parts: a cell state C_t, a forget gate f_t, an input gate i_t and an output gate h_t. The cell state (represented by a number in this case and by a vector of numbers in the case of an input vector with more dimensions) functions as a memory of the unit containing all important information available in the sequential input vector. This information is selected and filtered by the three gates, described in detail in the following paragraphs. Each of the gates has its set of weights and biases. Weights u_f and w_f and bias b_f belong to the forget gate, weights u_i, w_i, u_b, and w_b and biases b_i and b_b are used by the input gate, and the output gate contains weights u_h and w_h and bias b_h. Compared to the recurrent neural networks, the higher amount of weights and biases increases the computational complexity of the learning process of LSTMs. However, it enables the LSTMs to learn long-term dependencies in the data.

The forget gate is responsible for deleting previously saved information from the unit's cell state and is defined as

$$f_t = \sigma(u_f \cdot x_t + w_f \cdot o_{t-1} + b_f).$$

This gate uses the input x_t and the output o_{t-1} of the previous hidden unit (on the left of the Figure 1.9). The sigma function outputs a number between zero and one. If the output is zero, the information in the cell state will be fully deleted. In the case of an output equal to one, the information will be kept in the cell state.

Subsequently, the input gate, defined below, stores new information to the cell state:

$$i_t = \sigma(u_i \cdot x_t + w_i \cdot o_{t-1} + b_i),$$
$$B_t = \tanh(u_b \cdot x_t + w_b \cdot o_{t-1} + b_b).$$

The sigma function in the input gate i_t selects what parts of the cell state will be updated (in the case of a vector), and the hyperbolic tangent in B_t specifies what values will be saved to these selected parts. The cell state is then updated as follows, using the forget gate f_t and the multiplication of the input gate i_t and B_t:

$$C_t = i_t \cdot B_t + f_t \cdot C_{t-1}.$$

The last part of the hidden unit is the output gate h_t, defined below, which selects what parts of the cell state will be forwarded to the next hidden unit (the arrow o_t between the second and third blue node in Figure 1.9)

$$h_t = \sigma(u_h \cdot x_t + w_h \cdot o_{t-1} + b_h).$$

After this selection, the output gate h_t is multiplied with the cell state to obtain the actual information forwarded to the next hidden unit

$$o_t = \sigma(h_t \cdot \tanh(C_t)).$$

In the case of a sequential output, the same information is used as the output of the unit (the red node o_t in Figure 1.9).

Similarly to recurrent neural networks, all weights and biases are shared for the full sequential input and learnt during the learning process. After they are properly adjusted, the LSTM can learn the information contained in the input vector and use this information for producing an appropriate output (i.e. an output which minimises the error function (1.5)).

1.4.2 Peephole Connections

Gers and Schmidhuber [19] suggested a modification of this architecture by adding so-called peephole connections. Figure 1.10 pictures these connections (red lines). This enables all three gates to use the information contained in the cell state and to improve the performance of the network. The definitions of the gates change as follows:

$$f_t = \sigma(u_f \cdot x_t + w_f \cdot o_{t-1} + p_f \cdot C_t + b_f),$$
$$i_t = \sigma(u_i \cdot x_t + w_i \cdot o_{t-1} + p_i \cdot C_t + b_i),$$
$$h_t = \sigma(u_h \cdot x_t + w_h \cdot o_{t-1} + p_h \cdot C_t + b_h),$$

where p_f, p_i and p_h are weights assigned to the peephole connections. According to [20], this architecture together with using full gradient training [21] is the most commonly used architecture in literature.

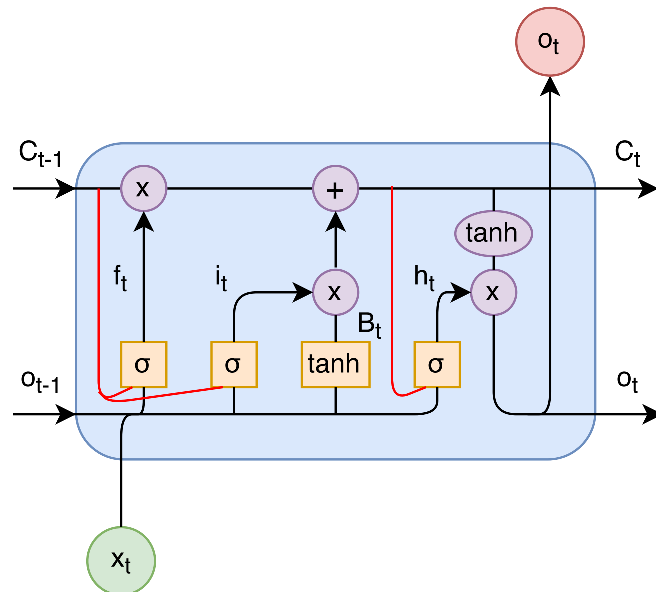


Figure 1.10: Adding peephole connections to LSTM's hidden unit.

Chapter 2

Implementation

The principles and four different architectures of artificial neural networks were theoretically introduced in the previous chapter. Three of the architectures (convolutional, recurrent and long short-term memory neural networks) were implemented in order to compare them in the last part of this thesis. This chapter focuses on the implementation and is divided into two sections. Section 2.1 describes technical details of the implementation, including architecture, programming language, and libraries. Section 2.2 introduces the implemented program and specifies various settings of algorithms.

2.1 Technical Details

The implementation part of the thesis is written in Python [22], a high-level programming language widely used in machine learning. Its main advantage is a large variety of existing libraries, further described in the next section. The code was written and executed in The Jupyter Notebook [23], a convenient web-based environment supporting many programming languages such as Scala, Ruby, R, JavaScript, and Python. The Jupyter Notebook was running in Docker [24], a software container, using the keras-full image [25].

As mentioned above, Python includes many useful libraries. The following list shortly describes six of them that were used in the implementation:

- Keras [26] is a high-level neural networks library running on top of either Theano or TensorFlow. Modularity and minimalism make it a very efficient tool for creating and experimenting with neural networks.
- TensorFlow [27] is an open source library for machine learning developed by Google. It features implicit scalability running on both CPUs and GPUs.

- SciPy [28] is a library for scientific computing. Among others, it implements a broad range of scientific functions.
- Numpy [29] is a Python extension introducing mainly N-dimensional array objects. It also includes an extensive library of mathematical functions to operate on these objects.
- Matplotlib [30] is a 2D plotting library. It offers various types of graphs such as plots, histograms, bar charts, and scatterplots.
- scikit-learn [31] is a machine learning library built on Numpy, SciPy, and Matplotlib. It implements many supervised and unsupervised algorithms, preprocessing methods, and model selection techniques.

2.2 Implementation Details

The previous section focused on the technical detailes of the implementation; this part describes various features of the experiment process and specific settings of the algorithms. The program offers three ways of optimising network hyperparameters: a manual search, a grid search, and a random search. The input of the manual search is a neural network model with defined hyperparameters. This model is trained on a training set and evaluated on a testing set. The program iteratively plots the training loss and the validation loss after every epoch. This process is repeated three times and the results are averaged in order to lower the effects of randomness.

The grid search and the random search automatically optimise all hyperparameters using a cross-validation on a training set. The grid search takes sets of values of various hyperparameters as inputs. These sets are further described in Chapter 3. Inputs of the random search are uniform distributions of the hyperparameters values. The output of both functions is the selected best model and its hyperparameters. This model is then evaluated on a testing set using the manual search.

The following list describes specific settings of the algorithms:

- Early stopping is implemented for all methods, i.e. if the validation loss has not improved for n epochs, the training process stops. $n = 5$ during the grid search and the random search and $n = 10$ during the training process on the full training data set using the manual search. The minimum change of the loss counting as an improvement is $delta = 0.001$. The maximum number of epochs is limited to 400.

- The activation function f used in the convolutional layers and the fully-connected layers (discussed in Section 1.2) as well as in the hidden layers of recurrent neural networks (described in Section 1.3) is the ReLU.

- The pooling layer is always added to the architecture of the convolutional neural networks.

- Zero-padding iteratively adds zeros to the front and the back of an input vector as long as the formula

$$(l - s + pf + pb) \mod t$$

  is not equal to zero. l represents the length of an input vector, pf and pb is the amount of zeros padded to the front and to the back of the vector, respectively. s stands for the size of the filter, and t is the filter stride.

- Output layers use the softmax function as the activation function defined as follows:

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

  for $j = 1, \ldots, K$, where $\mathbf{z}$ is a vector produced by the last hidden layer. The length K of the vector represents the number of classes. The outputs of the softmax function for all components of the vector add up to one and can be interpreted as a categorical distribution.

- Categorical cross entropy is implemented as the loss function, using this categorical distribution.

Chapter **3**

Experiments

The second chapter outlined the detailes of the implemented program. This program is used in the third chapter, which focuses on an experimental comparison of two state-of-the-art architectures of neural networks and one baseline architecture: convolutional neural network, long short-term memory neural network, and recurrent neural network, respectively. The goal is to compare them in terms of their ability to learn, the effectivity of the training process and the classification performance. The first section describes these criteria and proposes an experimental design for their comparison. The experimental design includes three benchmark data sets, introduced in the second section. The last section compares the three network architectures on these data sets.

3.1 Experimental Design

This section is divided into two subsections. The first subsection focuses on experiments for comparing different architectures of neural networks. It lists and describes evaluation criteria and metrics, and proposes an experimental procedure. An important step in this procedure is to properly set neural network hyperparameters, in order to successfully train it on a given data set. A design of experiments used for finding the best hyperparameters for any given data set is presented in the second subsection.

3.1.1 Process of the Experiment

The first subsection focused on the process of the experiment contains three parts: evaluation criteria, evaluation metrics, and the experimental procedure.

Evaluation Criteria

Different architectures of neural networks are compared in terms of three criteria:

- The **ability to learn** examines the structure of the networks, and how changes in the structure influence the classification performance.
- The **effectivity of the training process** focuses on the technical parameter, i.e. the time spent on training the network.
- The **classification performance** belongs to one of the most important metrics in a classification task. It measures how well the network approximates the function g (1.1), that is whether the network can learn the patterns in the data.

Evaluation Metrics

Each criterion has a different set of evaluation metrics, outlined in the following list. These metrics are obtained during the experimental procedure, described in the last part of this subsection. Subsequently, the metrics are compared separately for each of the benchmark data sets.

Table 3.1: Structures of convolutional neural networks, defined using hyperparameters B, C, and D.

Number of layers	B	C	D
3	1	1	1
4	2	1	1
5	1	2	1
6	1	2	2
7	2	2	1
8	2	2	2

- The **ability to learn** is analysed by plotting the dependency of the classification performance on the number of neurones and the number of hidden layers. For LSTMs and recurrent neural networks, the latter number means the number of layers containing LSTM and recurrent neurones, respectively. The performance is calculated for one to five hidden layers. Table 3.1 shows structures of convolutional neural networks, defined using hyperparameters B, C, and D, as described in Section 1.2. The structures have six different depths (three to eight hidden layers). Set $N = \{5x | 1 \leq x \leq 50\}$ defines possible numbers of neurones for all architectures, as well as numbers of filters for convolutional networks.
- The **effectivity of the training process** is compared by measuring the time of the training process.
- The **classification performance** is evaluated by comparing precision, recall, and F-score obtained on testing data sets. Furthermore, these

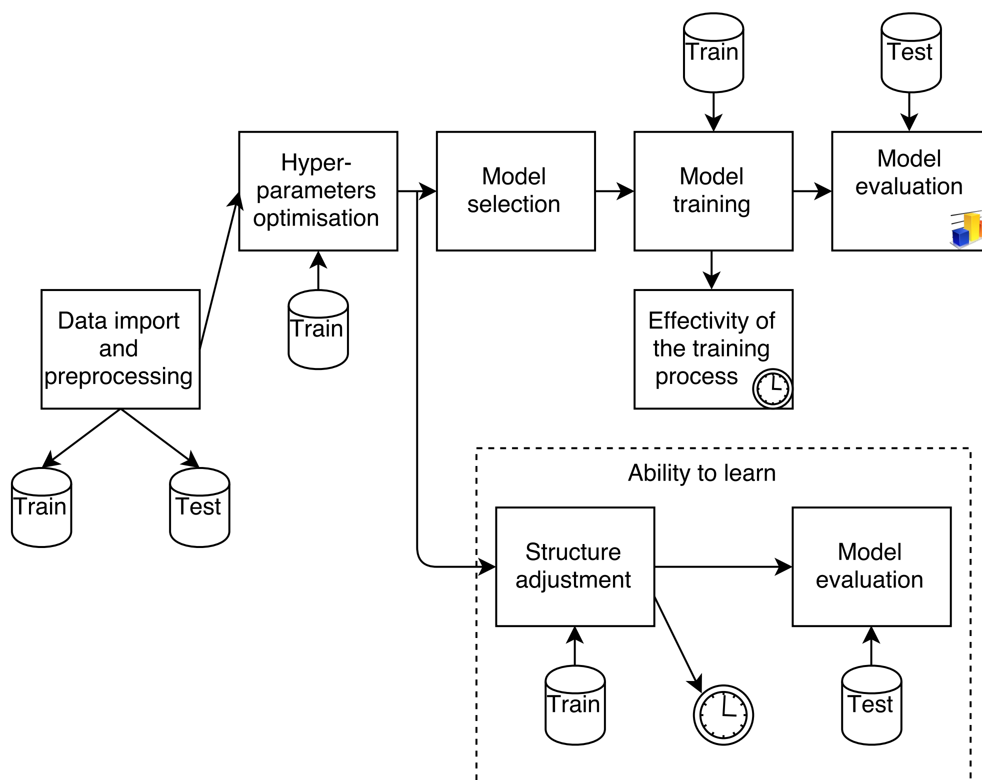


Figure 3.1: A graphical schema of the experimental procedure.

measures are compared separately for each class to determine whether there is a difference in the classification performance for various classes.

Experimental Procedure

The graphical schema in Figure 3.1 shows the experimental procedure. The data, divided into training and testing sets, are imported and transformed into a proper format. Both sets are randomly shuffled.[10] The hyperparameters of all network architectures are optimised using the training set. This process is further described later in the following section. The model with the best classification performance is selected for each of the architectures.

The evaluation metrics are calculated by applying this model to the full data set. Stratified 70% of the training set is used in the learning process, and 30% is held out as the validation set for early stopping. The effectivity of the training process is measured. The model is subsequently applied to

[10]The pseudorandom number generator seed is fixed for a better reproducibility of the results.

the testing set, and the classification performance is evaluated. This process is repeated three times, and the evaluation metrics are averaged accordingly. The training set is always shuffled between iterations.

The classification performance in dependency on the structure is calculated. The number of neurones and the number of layers are selected from the sets described in the second part of this subsection. Other hyperparameters are fixed according to the selected model. An exhaustive grid search is executed, resulting in $5 \cdot 50 = 250$ models for recurrent neural networks and $6 \cdot 50 = 300$ models for convolutional networks. These models are trained on the training data and tested on the testing data. This process is repeated three times, and the results are averaged.

3.1.2 Hyperparameters

Hyperparameters are variables defining neural network's higher-level properties such as its structure. They are set before the training process, i.e. before optimising the network parameters.

Recurrent Neural Networks

The following list shortly describes hyperparameters of recurrent neural networks.

- The **learning rate** influences the speed of the learning process by controlling how much the weights change during each update. For more information see Equation (1.4).
- The **learning rate decay** sets the speed at which the learning rate decays. A high learning rate has the advantage of a fast learning process and a broad exploration of the space of the loss function values. On the other hand, a small learning rate decreases the loss steadily and enables exploring parts of the search space which would be unreachable with a high learning rate. The learning rate decay combines the advantages of both. It sets a high learning rate at the beginning of the learning process and continually decreases it by little steps. However, Bengio [32] recommends keeping the learning rate constant. To reduce the complexity of the experimental design, this recommendation is followed.
- The **optimiser** is another important hyperparameter. Sections 1.1.3 and 1.3.2 describe the stochastic gradient descent (SGD) as the backpropagation algorithm, but more optimisers have been developed, such as RMSprop, Adagrad, Adadelta, Adam, Adamax, and Nadam.

- The **number of epochs**, further described in Section 1.1.3, influences the performance of the network and the length of the training process. Early stopping interrupts the training process if the validation loss has not improved for a defined number of epochs. Therefore, this hyperparameter is not optimised.

- The **size of the batch** is another hyperparameter. As described in Section 1.1.3, the mini-batch gradient descent (and other optimisers) use only a subset of the training set for updating the weights. A small batch means more updates and a broader exploration of the weights space. A bigger batch enables more precise gradient descent, however only locally, because the descent keeps changing during iterations.

- The **number of hidden layers** specifies the depth of the network. Deeper networks are usually able to better generalise from the data because of their ability to learn features at more levels of abstraction. On the other hand, larger networks are more prone to overfitting.

- The **number of neurones** in each hidden layer defines the width of the network and together with the number of layers forms its structure. Bengio [32] states that a larger number of neurones usually works better because the network can generalise well and the negative effects of overfitting are reduced by regularisation (see Dropout below).[11] Bengio also argues that using the same amount of neurones in each layer generally works better or the same as using a decreasing or increasing size.

- **Weights initialisation** is an important step at the beginning of the learning algorithm. These eight initialisers are tested: uniform, lecun uniform, normal, zero, glorot normal, glorot uniform, he normal, and he uniform.

- **Dropout** [33] is a regularisation technique used to avoid overfitting. During training, randomly selected units along with their connections are dropped out from the network. This makes the network less sensitive to specific settings of weights, thus prevents it from co-adapting too much to the training data. The tunable hyperparameter is the dropout rate, i.e. the amount of dropped out neurones.

- **Batch normalisation** [34] can be placed after each layer that includes activation functions. As the network parameters change during training, the distribution of the activations changes as well. This phenomenon is called *internal covariate shift* and it slows down the training process. Batch normalisation reduces the shift by normalising network activations during every batch update. The hyperparameter controls whether it is added into the network's architecture.

[11] The disadvantage is more computations.

These hyperparameters are not independent of each other and cannot be adjusted separately. Instead, the grid search is used, which is capable of optimising more hyperparameters simultaneously. However, they cannot be optimised all at once because of the exponential computational complexity of the grid search. Therefore, they are divided into five smaller subsets:

1. {number of neurones, number of layers, dropout rate},
2. {optimisers, learning rate},
3. {learning rate, batch normalisation, dropout rate},
4. {weights initialisation},
5. {batch size, learning rate}.

The values of the hyperparameters are limited as well:

- **learning rate** $\in$ {0.00001, 0.0001, 0.001, 0.01, 0.1},
- **optimisers** $\in$ {SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam},
- **batch size** $\in$ {1, 2, 3, 4, 8, 16, 32},
- **number of layers** $\in$ {1, 2, 3, 4, 5},
- **number of neurones** $\in$ {50, 100, 150, 200, 250},
- **weights initialisation** $\in$ {uniform, lecun uniform, normal, zero, glorot normal, glorot uniform, he normal, he uniform},
- **dropout rate** $\in$ {0.0, 0.2, 0.4, 0.6, 0.8},
- **batch normalisation** $\in$ {true, false}.

The experimental procedure for choosing the hyperparameters given a data set has five iterations. During each iteration, the grid search returns the best values for a subset of the hyperparameters, which are fixed before the next iteration. Evaluation of the values of the hyperparameters is based on cross-validation, where the training set is split into four stratified folds. Three models are learnt using two of these folds as training data, one fold as a validation set used by the early stopping algorithm, and the remaining fold as a testing set.

The random search is another optimisation technique showing good results as well [32]. During each iteration, every hyperparameter is randomly selected from the uniform distribution of its values. Thus, the random search can effectively search in the hyperparameter space. This technique is tested alongside the grid search, and the better model is selected.

Convolutional Neural Networks

The experimental procedure, introduced above, can be used for both recurrent neural networks and convolutional neural networks, but the set of hyperparameters differs. The number of layers is extended to three hyperparameters, **B**, **C** and **D**, as described in Section 1.2. That section also introduces six more hyperparameters specific for convolutional networks: the **number of filters**, the **size** of each **filter**, the **stride**, the **zero-padding**, the **pooling size**, and the **pooling stride**. To reduce the time complexity of the optimisation algorithm, the **pooling size** and the **pooling stride** are fixed to the value 2. The ordered list of subsets of the remaining hyperparameters looks as follows:

1. {number of neurones, number of filters, B, C, D},
2. {number of neurones, D, dropout rate},
3. {filter size, filter stride},
4. {optimisers, learning rate},
5. {learning rate, batch normalisation, dropout rate},
6. {weights initialisation},
7. {batch size, learning rate}.

The values are limited to these subsets:

- **learning rate** $\in$ {0.00001, 0.0001, 0.001, 0.01, 0.1},
- **optimisers** $\in$ {SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, Nadam},
- **batch size** $\in$ {1, 2, 3, 4, 8, 16, 32},
- **number of neurones** $\in$ {50, 100, 150, 250},
- **number of filters** $\in$ {50, 100, 150, 250},
- **filter size** $\in$ {2, 3, 4, $0.05n, 0.1n, 0.2n$},

- **filter stride** $\in \{1, 2, 3\}$,
- $B \in \{1, 2\}$,
- $C \in \{1, 2\}$,
- $D \in \{1, 2\}$,
- **weights initialisation** $\in$ {uniform, lecun uniform, normal, zero, glorot normal, glorot uniform, he normal, he uniform},
- **dropout rate** $\in \{0.0, 0.2, 0.4, 0.6, 0.8\}$,
- **batch normalisation** $\in$ {true, false}.

Three filter sizes are defined as ratios of the length n of a time series, as recommended by [35].

3.2 Benchmark Data Sets

The architectures are compared on three time series classification data sets. This section theoretically introduces time series and classification and describes single data sets.

3.2.1 Time Series

A time series is a set of sequentially collected observations. In the case of equally spaced time points, it is a set $\{y_t\}$, where $t \in \mathbb{Z}$ is the time at which an observation was taken. If the observations are not equally spaced in time, the notation is $\{y_{t_i}\}$, where $i \in \mathbb{N}$. Time series are usually represented by plots. Two examples are in Figures 3.2 and 3.3.

A multivariate time series is a set of time series with the same timestamps. Following the notation from above, for equally spaced time points, it is a set $\{\{y_{1t}\}, \{y_{2t}\}, \ldots, \{y_{qt}\}\}$, where q is the number of univariate time series (also called features), and $t \in \mathbb{Z}$ is the time. In the case of unequally spaced time points, it is a set $\{\{y_{1t_i}\}, \{y_{2t_i}\}, \ldots, \{y_{qt_i}\}\}$, where $i \in \mathbb{N}$.

3.2.2 Classification

Supervised learning is a machine learning task, where training examples are pairs consisting of an input and a corresponding output. Classification is a subset of supervised learning, where outputs represent classes (categories). The classification problem consists of sorting inputs into correct classes. More formally, a training set has a form of $\{(\mathbf{x_1}, y_1), (\mathbf{x_2}, y_2), \ldots, (\mathbf{x_m}, y_m)\}$, where $\mathbf{x_i}$ are input vectors, and y_i are corresponding classes. Time series classification is a classification problem, where inputs $\mathbf{x_i}$ represent time series.

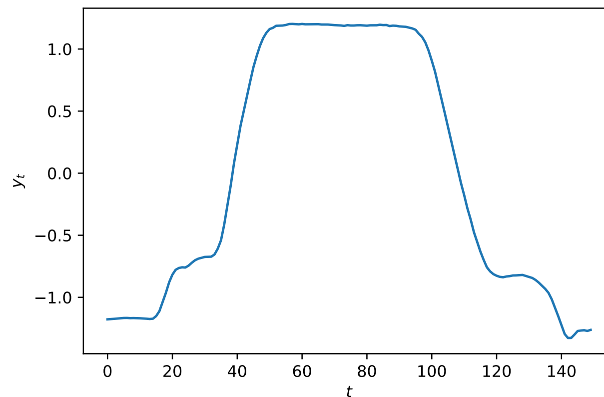


Figure 3.2: An example of the *Gun-Draw* class from the Gun-Point data set.

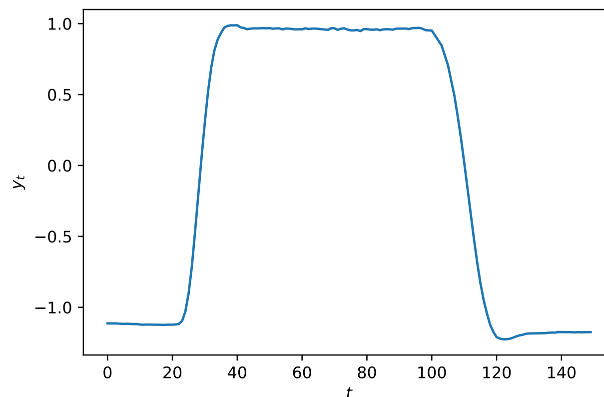


Figure 3.3: An example of the *Point* class from the Gun-Point data set.

3.2.3 Data Sets

Time series classification data sets used in the experimental part of this thesis come from the UCR Time Series Classification Archive [36], and the UCI Machine Learning Repository [37].

Gun-Point

The first set is the Gun-Point data set, originally published by Ratanamahatana and Keogh [38]. It contains 200 univariate time series, 50 of them belong to the training set and 150 to the testing set. The time series were obtained by tracking the motion of the right hand of one male and one female actor. The data set is divided into two classes: *Gun-Draw* and *Point*.

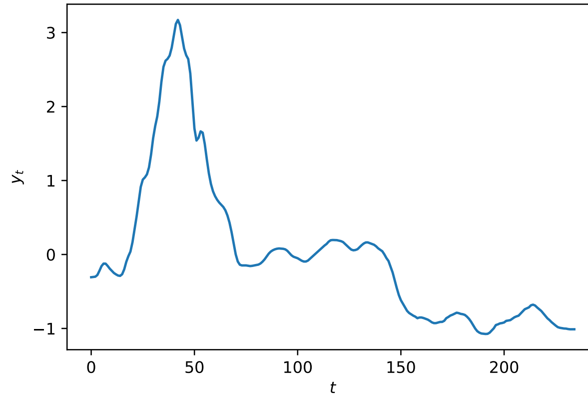


Figure 3.4: An example of the *Strawberry* class from the Strawberry data set.

Both classes begin with the actors standing and having both hands at their sides. In the *Gun-Draw* class, they reach for a gun stored in a holster, which is mounted at their hips. They draw the gun at an imaginary opponent for approximately one second, return it to the holster and put their hands back to their sides. In the *Point* class, they only point their index fingers at the opponent and put their hands back to their sides. The time series are composed of the x-coordinates of the centroids of their hands tracked during these procedures. Examples of time series for the *Gun-Draw* and *Point* classes are shown in Figures 3.2 and 3.3, respectively. It can be seen that there are two lumps in the first figure, as the actor had to reach for the gun. No such lumps are in the *Point* class figure.

Strawberry

The Strawberry data set, published in [39], comes from the field of food analysis. It contains 983 univariate time series, divided into the training (370) and testing (613) set. Two classes were obtained using Fourier transform infrared spectroscopy of strawberry purées for a *Strawberry* class and non-strawberry (such as raspberry, apple, blackberry or adultered strawberry) purées for a *Non-Strawberry* class. An example of the *Strawberry* class is in Figure 3.4.

Japanese Vowels

The Japanese Vowels data set, originally collected by Kudo et al. [40], is a part of the UCI Machine Learning Repository [37]. It is split into the training set (270 time series) and the testing set (370 time series). All time series are multivariate and contain 12 features. Each time series represents a sound of

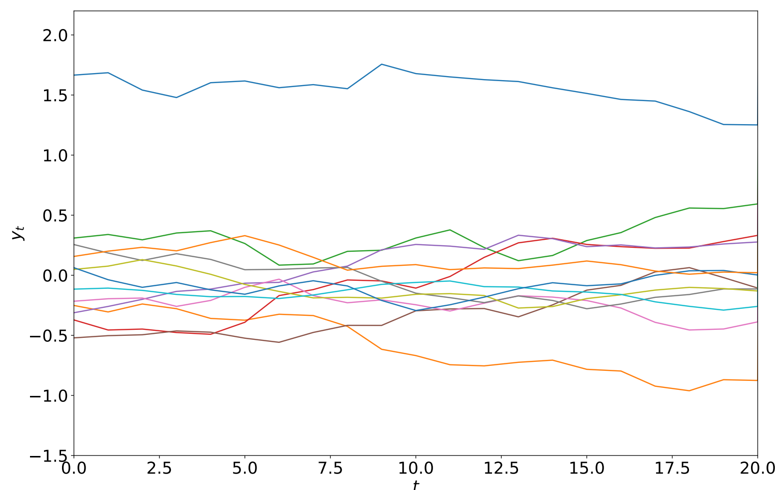


Figure 3.5: The Japanese Vowels data set.

two Japanese vowels /ae/ uttered successively by one of nine male speakers, forming a set of nine classes. The features are 12 LPC cepstrum coefficients obtained from the 12-degree linear prediction analysis. An example of the time series is in Figure 3.5.

3.3 Comparison of Architectures

The three data sets, introduced above, are used for the comparison of three architectures of neural networks. Every set is compared in a separate subsection. Each architecture is represented by one best model with a set of hyperparameters found during the optimisation process. These models are compared in terms of the ability to learn, the effectivity of the training process, and the classification performance.

3.3.1 Gun-Point

The first subsection focuses on the smallest of the data sets, the Gun-Point data set. The following paragraphs list the optimised hyperparameters and compare the ability to learn, the effectivity of the training process, and the classification performance.

Table 3.2: Optimised hyperparameters for the Gun-Point data set.

	RNN	LSTM	CNN
Learning rate	0.0001	0.000684	0.003244
Optimiser	Adamax	Adamax	RMSprop
Batch size	1	27	17
Number of layers	3	4	
Number of neurones	100	100	182
Weights initialisation	glorot normal	he uniform	glorot uniform
Dropout rate	0.0	0.439786	0.24793
Batch normalisation	False	False	False
Number of filters			128
Filter size			3
Filter stride			2
B			1
C			1
D			2

Hyperparameters

Table 3.2 shows the optimised hyperparameters for three architectures: recurrent neural network (RNN), long short-term memory neural network (LSTM), and convolutional neural network (CNN). The random search found better models than the grid search for two architectures (LSTM and CNN). For the RNN, the model selected by the grid search had better classification performance than the model selected by the random search.

Ability to Learn

Figure 3.6 shows the classification performance (F-score) in dependency on the number of neurones (x-axis) and the number of layers (different colour shades) for CNNs. All results were calculated for up to 250 neurones, but because of no further information in the data, the plots are limited to 150 neurones. Networks with seven or eight layers are generally worse on this data set than networks with fewer layers. The classification performance seems to be correlated with the number of neurones for smaller architectures (five to approximately 30 neurones). For architectures with more than 30 neurones, the performance fluctuates but does not change on average. The classification performance for the CNN architecture on the Gun-Point data set cannot be concluded independent on the structure of the network.

The classification performance in dependency on the number of neurones and the number of layers for LSTMs can be seen in Figure 3.7. The classification performance increases with the number of neurones for networks with

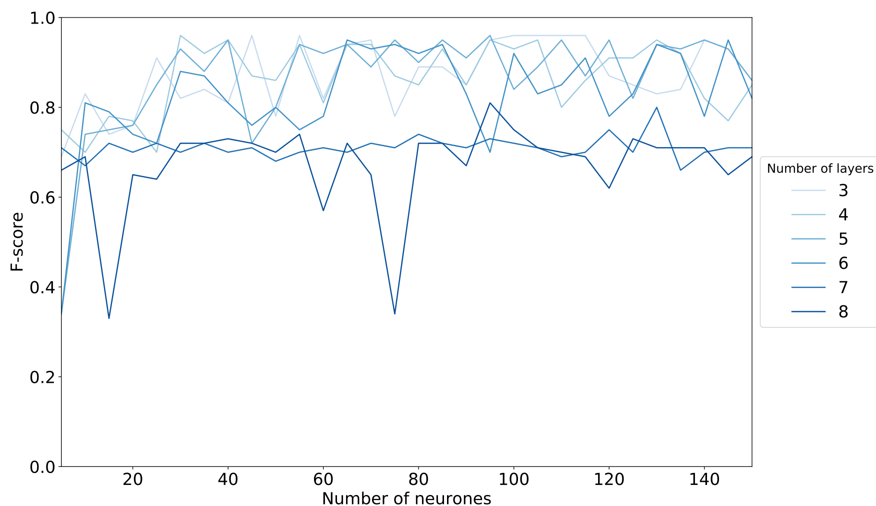


Figure 3.6: The classification performance in dependency on the number of neurones and the number of layers for CNN on the Gun-Point data set.

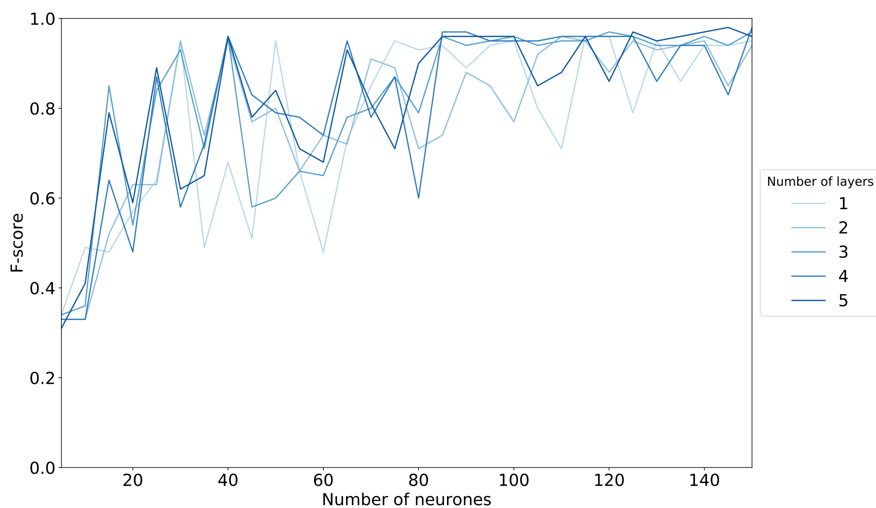


Figure 3.7: The classification performance in dependency on the number of neurones and the number of layers for LSTM on the Gun-Point data set.

5 to approximately 100 neurones. For networks with more than 100 neurones, the performance stays almost the same with a few fluctuations. These fluctuations are highest for networks with fewer layers and lowest for deeper networks.

The classification performance in dependency on the RNN structure, shown in Figure 3.8, shows big fluctuations. It is positively correlated with the num-

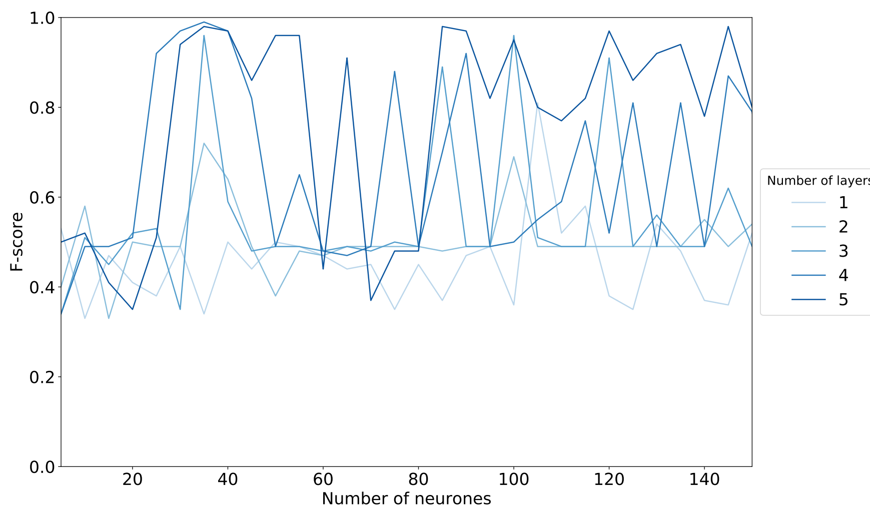


Figure 3.8: The classification performance in dependency on the number of neurones and the number of layers for RNN on the Gun-Point data set.

ber of layers; the more layers a network has, the better its classification performance is. No such simple correlation can be found between the performance and the number of neurones. As the performance for many networks is higher than for the network selected by the hyperparameters optimisation process (depicted in Figure 3.14), this process does not work optimally. This behaviour is not unexpected because of the computational and time limitations, discussed in Section 3.1.2. Nevertheless, the average performance for RNNs is certainly smaller than for CNNs and LSTMs. Thus, the Figure 3.14 contains relevant information about the three architectures.

In conclusion, the classification performance on the Gun-Point data set is the least dependent on the LSTM structure. Therefore, this architecture can be considered the most suitable for learning these data.

Figures 3.9, 3.11, and 3.12 compare the time of the training for different architectures. As the scales of the y-axes are the same, it can be easily concluded that the training of CNNs is the fastest, and the training of RNNs is the slowest. The training time for LSTMs and RNNs is positively correlated with the number of layers, but not with the number of neurones. The training time for CNNs (shown in detail in Figure 3.10) seems positively correlated both with the number of layers and the number of neurones.

Effectivity of the Training Process

Table 3.3 and Graph 3.13 compare the effectivity of the training process. The second column shows the time spent during the training process. The training

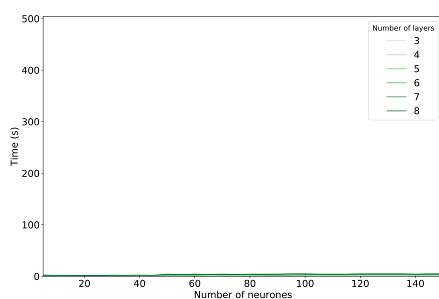


Figure 3.9: Time of the training (same scale) in dependency on the number of neurones and the number of layers for CNN on the Gun-Point data set.

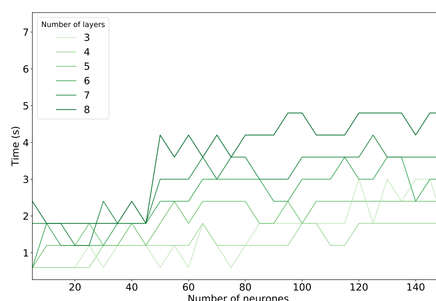


Figure 3.10: Time of the training (different scale) in dependency on the number of neurones and the number of layers for CNN on the Gun-Point data set.

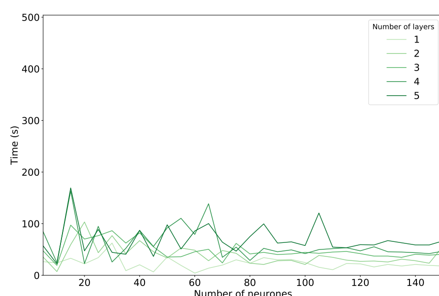


Figure 3.11: Time of the training in dependency on the number of neurones and the number of layers for LSTM on the Gun-Point data set.

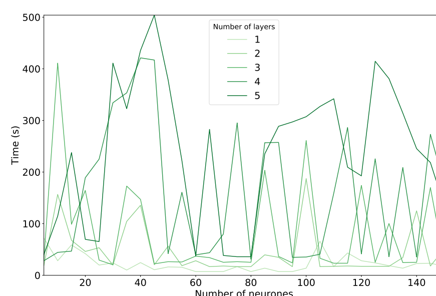


Figure 3.12: Time of the training in dependency on the number of neurones and the number of layers for RNN on the Gun-Point data set.

of the CNN is significantly faster than the training of the LSTM and RNN.

Classification Performance

Probably the most important criteria for comparing architectures of neural networks on a classification problem is the classification performance, shown in Table 3.4 and Graph 3.14. Each row/colour represents one architecture. It can be concluded that the LSTM has better classification performance on the testing data, achieving an *F-score* of 0.95. Also, the state-of-the-art architectures are significantly better than the baseline RNN.

Tables 3.5, 3.6 and 3.7 show precision, recall and F-score, respectively,

Table 3.3: Comparison of the effectivity of the training process on the Gun-Point data set.

	Time (s)
CNN	3.6
LSTM	37.8
RNN	47.4

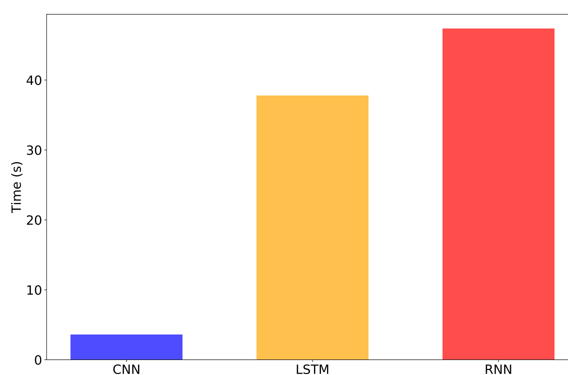


Figure 3.13: Comparison of the effectivity of the training process on the Gun-Point data set.

Table 3.4: Comparison of the classification performance for the testing subset of the Gun-Point data set.

	Precision	Recall	F-score
CNN	0.92	0.91	0.91
LSTM	0.96	0.95	0.95
RNN	0.5	0.5	0.5

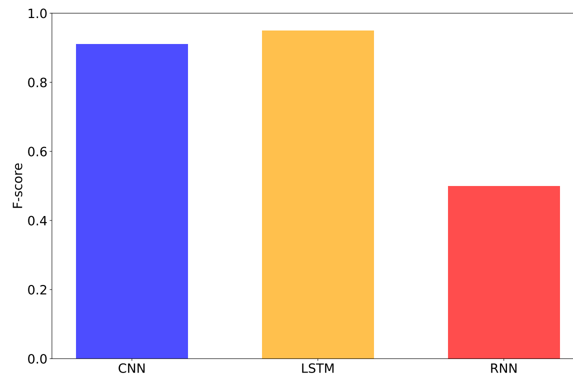


Figure 3.14: Comparison of the classification performance for the testing subset of the Gun-Point data set.

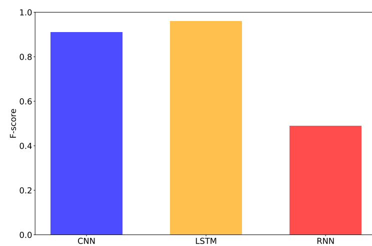


Figure 3.15: Comparison of F-score for the Gun-Draw class of the Gun-Point data set.

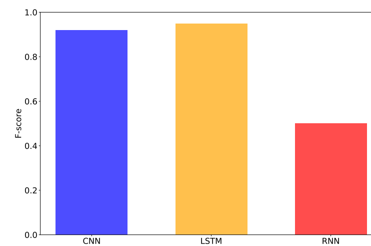


Figure 3.16: Comparison of F-score for the Point class of the Gun-Point data set.

separately for two classes of the Gun-Point data set. Graphs 3.15 and 3.16 then focus on F-score comparison for the Gun-Draw and Point class, respectively. The RNN gives solid baseline results, recognising 48% of time series in the *Gun-Draw* class, and 51% in the *Point* class.

The LSTM recognises all time series in the *Gun-Draw* class, and 91% in the *Point* class. The CNN can recognise only 88% of time series in the *Gun-Draw* class, but 95% of them in the *Point* class. These numbers correlate with precision, where the CNN has better results than the LSTM for the *Gun-Draw* class (0.94 versus 0.92), but worse results for the *Point* class (0.89 versus 1.0).

Considering all three metrics (the ability to learn, the effectivity of the training process, and the classification performance), the LSTM can be con-

Table 3.5: Comparison of precision separately for classes of the Gun-Point data set.

	CNN	LSTM	RNN
Gun-Draw	0.94	0.92	0.5
Point	0.89	1.0	0.49

Table 3.6: Comparison of recall separately for classes of the Gun-Point data set.

	CNN	LSTM	RNN
Gun-Draw	0.88	1.0	0.48
Point	0.95	0.91	0.51

Table 3.7: Comparison of F-score separately for Classes of the Gun-Point data set.

	CNN	LSTM	RNN
Gun-Draw	0.91	0.96	0.49
Point	0.92	0.95	0.5

cluded as the most suitable for the Gun-Point data set, though it needs the most time to learn the data.

3.3.2 Strawberry

The second subsection focuses on the largest of the data sets, the Strawberry data set. Similarly to the previous subsection, the ability to learn, the effectivity of the training process, and the classification performance are evaluated for the three architectures.

Hyperparameters

The optimised hyperparameters for the Strawberry data set for three architectures (RNN, LSTM, and CNN) are in Table 3.8. The best RNN model was optimised by the grid search, and the CNN and the LSTM were selected by the random search.

Ability to Learn

Figure 3.17 shows the classification performance (F-score) in dependency on the number of neurones (x-axis) and the number of layers (different colour shades) for the CNN. All results were again calculated for up to 250 neurones, but because of no further information in the data, the plots are limited to 175

Table 3.8: Optimised Hyperparameters for the Strawberry data set.

	RNN	LSTM	CNN
Learning rate	0.001	0.00027	0.001335
Optimiser	Adagrad	Adamax	Nadam
Batch size	4	26	15
Number of layers	1	1	
Number of neurones	150	203	118
Weights initialisation	glorot normal	he uniform	he normal
Dropout rate	0.4	0.381902	0.153957
Batch normalisation	True	False	False
Number of filters			212
Filter size			3
Filter stride			1
B			1
C			2
D			1

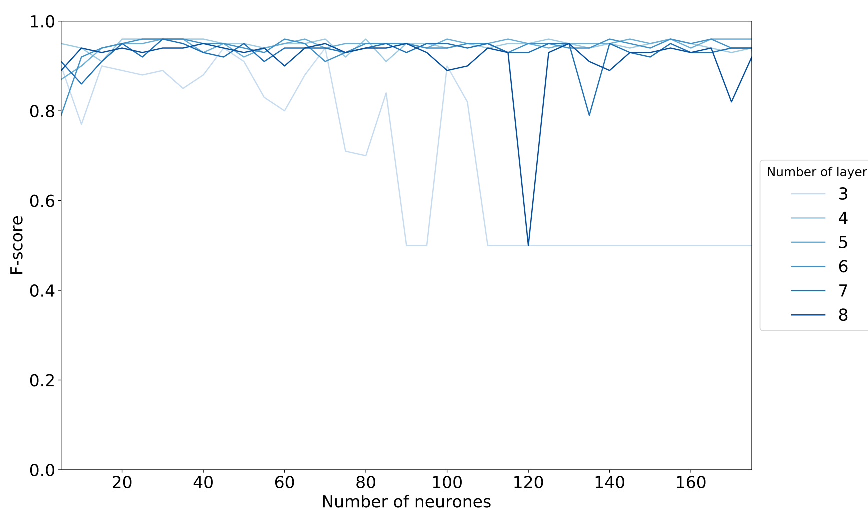


Figure 3.17: The classification performance in dependency on the number of neurones and the number of layers for CNN on the Strawberry data set.

neurones. The classification performance of architectures with three layers is negatively correlated with the number of neurones for up to 90 neurones—the more neurones each layer has, the lower the performance of the network is. Architectures with more than three layers give mostly high and stable results for any number of neurones.

The classification performance in dependency on the number of neurones

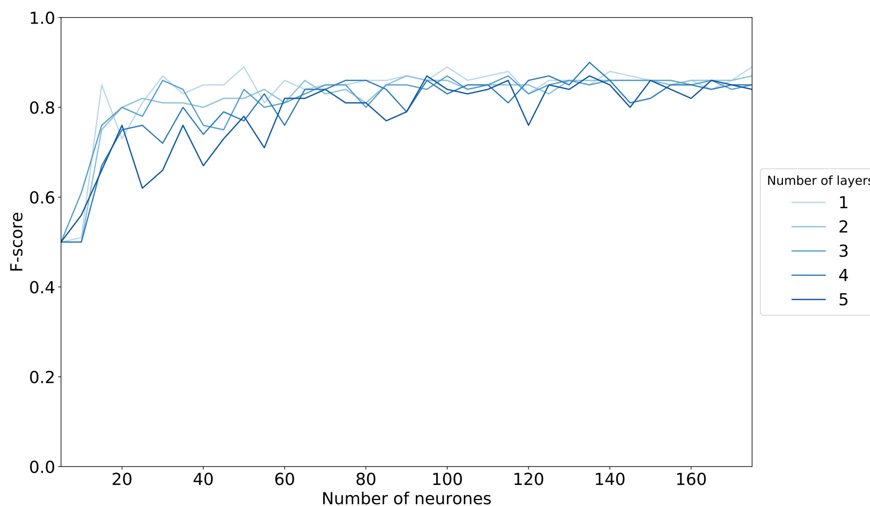


Figure 3.18: The classification performance in dependency on the number of neurones and the number of layers for LSTM on the Strawberry data set.

and the number of layers for the LSTM can be seen in Figure 3.18. There seems to be a slight negative correlation between the number of layers and the classification performance. The performance increases for the first 60 neurones and then stays the same with small fluctuations.

The classification performance in dependency on the RNN structure is shown in Figure 3.19. Architectures with only one layer give the most unstable results, but the mean of the classification performance is the highest. On the other hand, the performance of architectures with four layers fluctuates the least, but its mean is the lowest. No simple correlation between the classification performance and the number of neurones can be deduced.

Time in dependency on structure is in Figures 3.20, 3.22 and 3.23. LSTMs and CNNs (in detail shown in Graph 3.21) with more layers need a longer time to be trained than networks with fewer layers. For CNNs, there is also a correlation between the time and the number of neurones. When comparing the three networks, the CNNs are again significantly faster than the two recurrent architectures.

Effectivity of the Training Process

Table 3.9 and Graph 3.24 show comparison of the effectivity of the training process. Training of the CNN selected by the hyperparameters optimisation process is around eight times faster than the training of the recurrent networks.

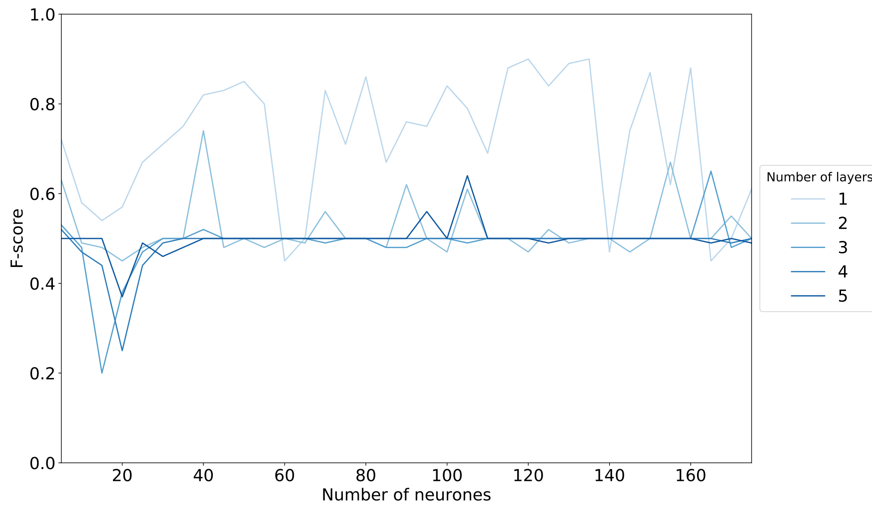


Figure 3.19: The classification performance in dependency on the number of neurones and the number of layers for RNN on the Strawberry data set.

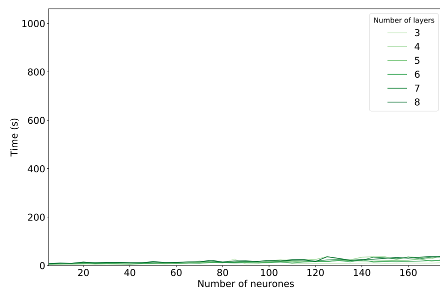


Figure 3.20: Time of the training (same scale) in dependency on the number of neurones and the number of layers for CNN on the Strawberry data set.

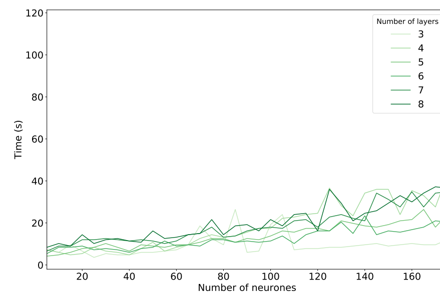


Figure 3.21: Time of the training (different scale) in dependency on the number of neurones and the number of layers for CNN on the Strawberry data set.

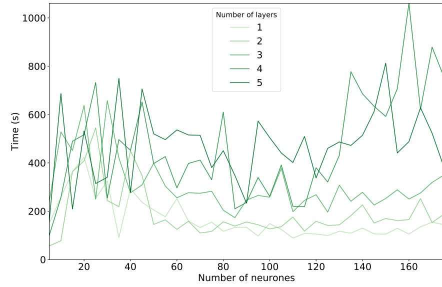


Figure 3.22: Time of the training in dependency on the number of neurones and the number of layers for LSTM on the Strawberry data set.

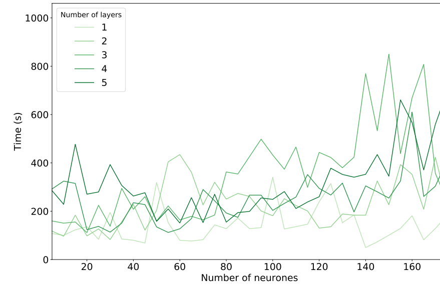


Figure 3.23: Time of the training in dependency on the number of neurones and the number of layers for RNN on the Strawberry data set.

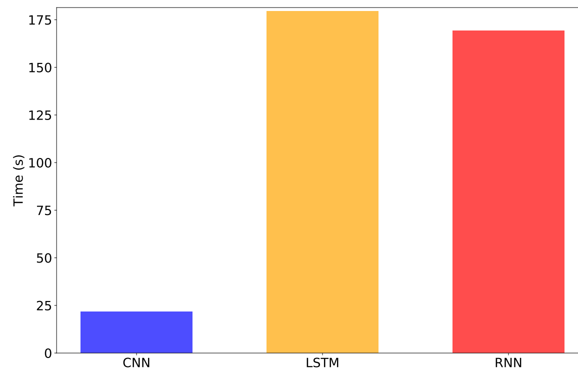


Figure 3.24: Comparison of the effectivity of the training process on the Strawberry data set.

Classification Performance

Comparison of the classification performance for the Strawberry data set is in Table 3.10 and Figure 3.25. Interestingly, the RNN and the LSTM have the same results, *F-score* $= 0.87$. The CNN, with *F-score* $= 0.96$, has the best classification performance for this data set.

Comparing the measures separately for the two classes of the data set (Tables 3.11, 3.12, and 3.13, and Figures 3.26 and 3.27), the CNN classifies both classes equally well, but the LSTM and the RNN are more successful in classifying the Non-Strawberry class (*F-score* $= 0.89$ versus *F-score* $= 0.83$ for the Strawberry class). The CNN can be concluded a more suitable architecture

Table 3.9: Comparison of the effectivity of the training process on the Strawberry data set.

	Time (s)
CNN	21.6
LSTM	179.4
RNN	169.2

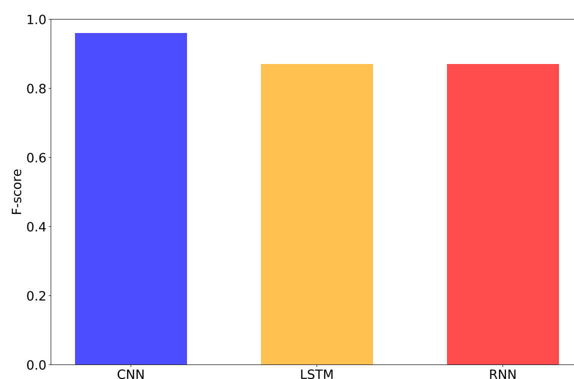


Figure 3.25: Comparison of the classification performance for the testing subset of the Strawberry data set.

Table 3.10: Comparison of the classification performance for the testing subset of the Strawberry data set.

	Precision	Recall	F-score
CNN	0.96	0.96	0.96
LSTM	0.88	0.87	0.87
RNN	0.88	0.87	0.87

for this data set.

3.3.3 Japanese Vowels

The last subsection focuses on the comparison of the architectures on the Japanese Vowels data set.

Hyperparameters

The optimised hyperparameters for the Japanese Vowels data set are shown in Table 3.14. The grid search found better models for the RNN and the CNN

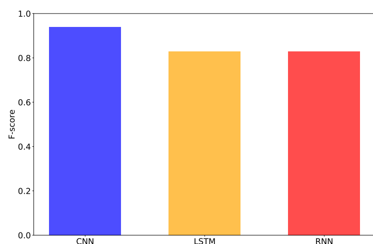


Figure 3.26: Comparison of F-score for the Strawberry class of the Strawberry data set.

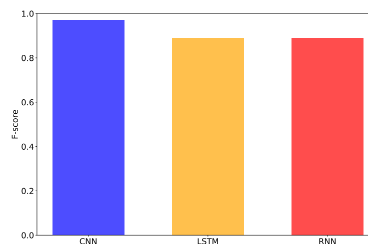


Figure 3.27: Comparison of F-score for the Non-Strawberry class of the Strawberry data set.

Table 3.11: Comparison of precision separately for classes of the Strawberry data set.

	CNN	LSTM	RNN
Strawberry	0.93	0.77	0.78
Non-Strawberry	0.97	0.94	0.93

Table 3.12: Comparison of recall separately for classes of the Strawberry data set.

	CNN	LSTM	RNN
Strawberry	0.95	0.91	0.89
Non-Strawberry	0.96	0.85	0.86

architectures; the LSTM model was selected by the random search.

Ability to Learn

The classification performance in dependency on the number of neurones and the number of layers for CNNs (Figure 3.28) and LSTMs (Figure 3.29) look similar, containing a correlation between the F-score and the number of neurones for the first approximately 50 neurones. For more than 50 neurones, the classification performance is almost constantly high, only slightly fluctuating (with $F - score$ close 1.0 for LSTMs, and around 0.9 for CNNs). Except for LSTMs with one hidden layer, which have worse results than networks with more layers, the performance is not dependent on the number of layers.

The classification performance for RNNs with more than 40 neurones fluctuates a lot with changes to the structure (Figure 3.30). Networks with fewer

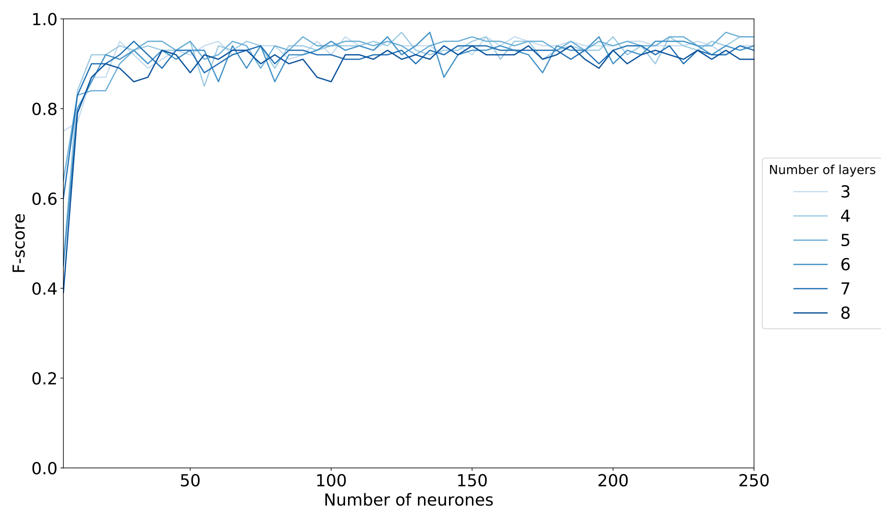


Figure 3.28: The classification performance in dependency on the number of neurones and the number of layers for CNN on the Japanese Vowels data set.

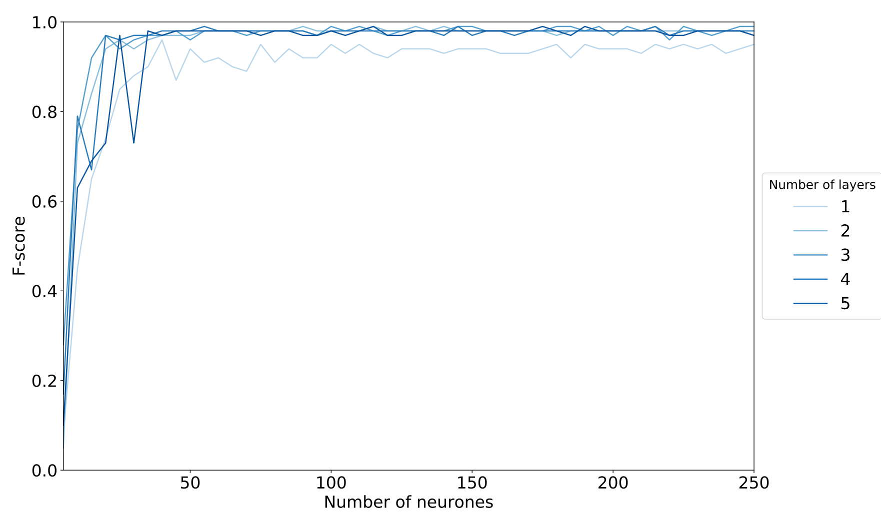


Figure 3.29: The classification performance in dependency on the number of neurones and the number of layers for LSTM on the Japanese Vowels data set.

Table 3.13: Comparison of F-score separately for Classes of the Strawberry data set.

	CNN	LSTM	RNN
Strawberry	0.94	0.83	0.83
Non-Strawberry	0.97	0.89	0.89

Table 3.14: Optimised Hyperparameters for the Japanese Vowels data set.

	RNN	LSTM	CNN
Learning rate	0.0001	0.000179	0.001
Optimiser	RMSprop	Adamax	RMSprop
Batch size	32	7	4
Number of layers	2	2	
Number of neurones	250	227	50
Weights initialisation	lecun uniform	uniform	uniform
Dropout rate	0.2	0.349926	0.0
Batch normalisation	True	True	True
Number of filters			250
Filter size			5
Filter stride			1
B			1
C			2
D			1

layers seem to give more unstable results than deeper networks. The results of the RNN architecture are more dependent and on average worse (0.29) than the results of CNNs and LSTMs (0.92 and 0.94, respectively). The optimisation process works well on this data set and selects a model with high *F-score* (0.83, listed in Table 3.16).

CNNs (Graph 3.31) and RNNs (Graph 3.34) need similar times for training (on average 20.69 and 19.45 seconds for CNNs and RNNs, respectively). Similarly to the previous two data sets, training times of CNNs (Graph 3.32) are positively correlated with their structure. Interestingly, training times of LSTMs (Figure 3.33) seem negatively correlated with the number of neurones. This is probably caused by the early stopping algorithm. More complex networks can better learn the data and need fewer epochs to be trained.

Effectivity of the Training Process

Comparison of the effectivity of the training process for networks selected by the hyperparameters optimisation process is shown in Table 3.15 and

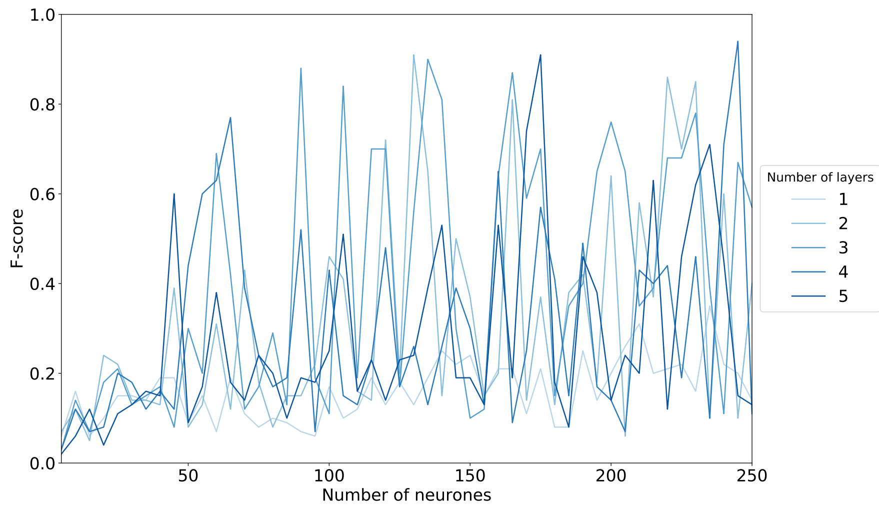


Figure 3.30: The classification performance in dependency on the number of neurones and the number of layers for RNN on the Japanese Vowels data set.

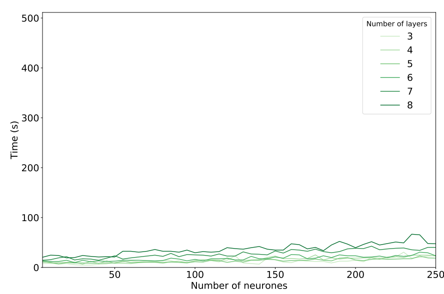


Figure 3.31: Time of the training (same scale) in dependency on the number of neurones and the number of layers for CNN on the Japanese Vowels data set.

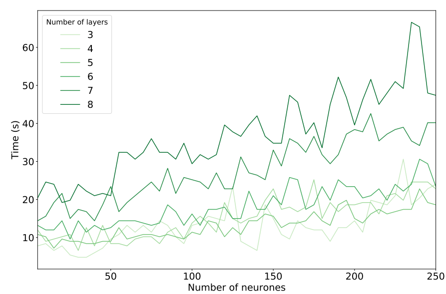


Figure 3.32: Time of the training (different scale) in dependency on the number of neurones and the number of layers for CNN on the Japanese Vowels data set.

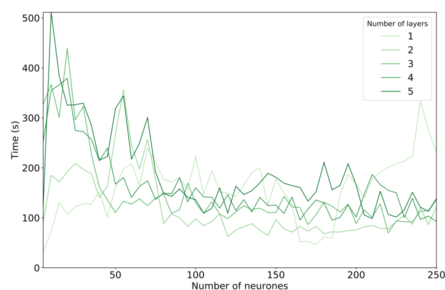


Figure 3.33: Time of the training in dependency on the number of neurones and the number of layers for LSTM on the Japanese Vowels data set.

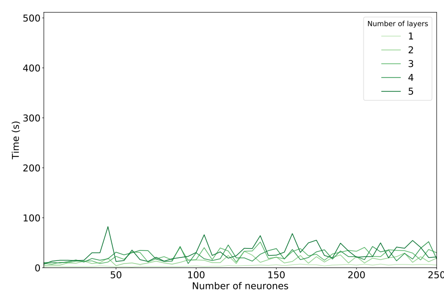


Figure 3.34: Time of the training in dependency on the number of neurones and the number of layers for RNN on the Japanese Vowels data set.

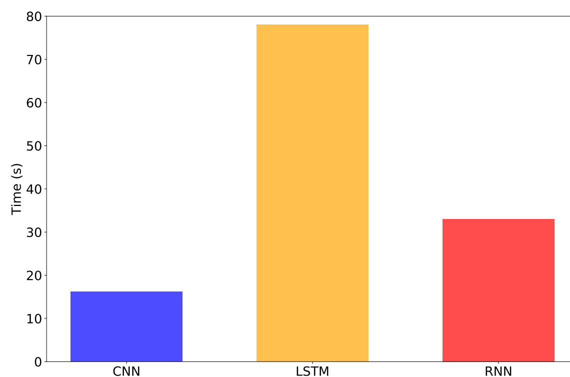


Figure 3.35: Comparison of the effectivity of the training process on the Japanese Vowels data set.

Graph 3.35. Training of the CNN is again significantly faster than the training of the LSTM. The RNN needs less time to be trained than the LSTM, though still two times more than the CNN.

Classification Performance

The classification performance for the Japanese Vowels data set is compared in Table 3.16 and Figure 3.36. The LSTM has the best results with *F-score* $= 0.98$, followed by the CNN with *F-score* $= 0.96$. The RNN gives solid baseline results for this data set, *F-score* $= 0.83$.

Table 3.15: Comparison of the effectivity of the training process on the Japanese Vowels data set.

	Time (s)
CNN	16.2
LSTM	78.0
RNN	33.0

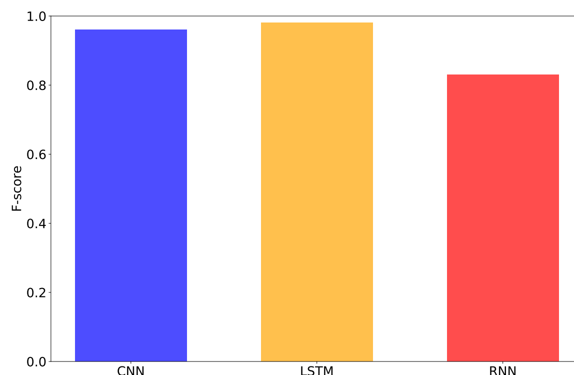


Figure 3.36: Comparison of the classification performance for the testing subset of the Japanese Vowels data set.

Table 3.16: Comparison of the classification performance for the testing subset of the Japanese Vowels data set.

	Precision	Recall	F-score
CNN	0.96	0.96	0.96
LSTM	0.98	0.98	0.98
RNN	0.86	0.82	0.83

The Japanese Vowels data set contains nine classes, as described in Section 3.2.3. For six of the classes the CNN and the LSTM give similarly good results, but there are three exceptions where the F-score of the CNN is lower than the F-score of the LSTM (with a *delta* of at least 0.05)—Speakers 2, 4, and 9. F-scores of Speakers 2 and 9 are compared in Figures 3.37 and 3.38. Tables 3.17, 3.18, and 3.19 show detailed comparison of precision, recall and F-score, respectively, for all nine classes.

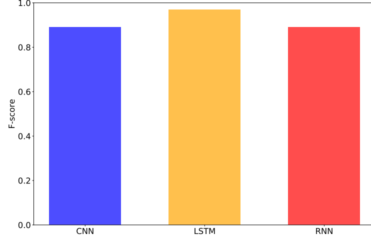


Figure 3.37: Comparison of F-score for the Speaker 2 class of the Japanese Vowels data set.

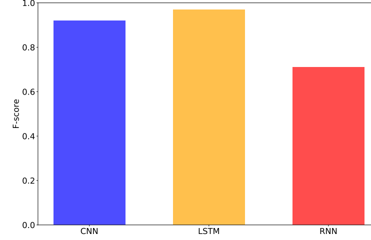


Figure 3.38: Comparison of F-score for the Speaker 9 class of the Japanese Vowels data set.

Table 3.17: Comparison of precision separately for classes of the Japanese Vowels data set.

	CNN	LSTM	RNN
Speaker 1	0.93	1.0	0.95
Speaker 2	0.91	0.97	0.85
Speaker 3	0.96	0.99	0.97
Speaker 4	0.98	0.98	0.96
Speaker 5	0.99	0.98	0.49
Speaker 6	0.99	0.99	1.0
Speaker 7	0.96	0.98	0.88
Speaker 8	0.97	0.97	0.74
Speaker 9	0.94	0.95	0.71

Table 3.18: Comparison of recall separately for classes of the Japanese Vowels data set.

	CNN	LSTM	RNN
Speaker 1	0.99	0.98	0.88
Speaker 2	0.87	0.96	0.94
Speaker 3	0.98	0.97	0.65
Speaker 4	0.89	0.98	0.85
Speaker 5	0.99	0.98	0.92
Speaker 6	1.0	1.0	0.99
Speaker 7	1.0	1.0	0.88
Speaker 8	0.98	0.97	0.85
Speaker 9	0.91	0.99	0.71

Table 3.19: Comparison of F-score separately for Classes of the Japanese Vowels data set.

	CNN	LSTM	RNN
Speaker 1	0.96	0.99	0.92
Speaker 2	0.89	0.97	0.89
Speaker 3	0.97	0.98	0.78
Speaker 4	0.93	0.98	0.9
Speaker 5	0.99	0.98	0.64
Speaker 6	0.99	0.99	0.99
Speaker 7	0.98	0.99	0.88
Speaker 8	0.97	0.97	0.79
Speaker 9	0.92	0.97	0.71

Conclusion

This thesis focused on artificial neural networks, machine learning algorithms nowadays used in a wide variety of domains. It theoretically described four different architectures: feedforward, convolutional (CNN), recurrent (RNN), and long short-term memory (LSTM) neural networks. These networks were explained in terms of their structure, their building blocks—artificial neurones, and two learning algorithms: backpropagation and backpropagation through time. Three of the architectures (CNN, RNN, and LSTM) were implemented in Keras, a Python neural network library running on top of TensorFlow.

One of common machine learning problems where neural networks can be used is time series classification. To accomplish this task, it is necessary to select a proper architecture and to optimise hyperparameters of the network. Thus, an experimental procedure for comparing different architectures in terms of their ability to learn, the effectivity of the training process, and the classification performance was proposed and implemented in the third chapter of this thesis. The process also includes automatic optimisation of neural network's hyperparameters using scikit-learn grid and random search functions.

Based on the experimental procedure, the three implemented architectures suitable for time series classification were compared on three benchmark data sets: the Gun-Point data set, the Strawberry data set, and the Japanese Vowels data set. On the Gun-Point data set, which is the smallest of the three, the LSTM had the highest classification performance with *F-score* $= 0.95$ compare to the CNN (0.91) and the RNN (0.50). The performance of LSTMs was also the least dependent on the structure of the network. On the other hand, the training of the LSTM took substantially more time than the training of the CNN.

On the Strawberry data set, containing the largest training set consisting

of 370 univariate time series, the CNN gave the best classification performance with *F-score* = 0.96, compare to *F-scores* of the RNN and the LSTM, which were both equal to 0.87. Furthermore, the training of the CNN was around eight times faster than the training of the recurrent networks.

The classification performance for the Japanese Vowels data set was highest for the LSTM (*F-score* = 0.98), followed by the CNN (*F-score* = 0.96) and the RNN (*F-score* = 0.83). However, the average training time for different structures of CNNs and RNNs was approximately eight times lower than the average training time of LSTMs. For almost all tested structures, the CNNs and the LSTMs gave high classification performance.

Comparing the classification performance in dependency on the structure for all data sets, the following can be concluded. First, the variance of the performance of CNNs and LSTMs seems correlated with the complexity of the training data set, i.e. with the size of the set and the number of features. On the least complex set, the Gun-Point data set, the results showed the highest variance (0.140). On the other hand, on the Japanese Vowels data set, which is the most complex set including 270 time series with 12 features, the variance of the results was the lowest (0.097). This indicates that the LSTMs and CNNs can better learn more complex data sets which is probably caused by more information available in these sets.

Second, the results show that LSTMs and CNNs have better learning abilities than RNNs. The average performance of the RNNs on all data sets was significantly lower than the performance of the state-of-the-art architectures. The largest difference was on the Japanese Vowels data set (0.29 versus 0.94 and 0.92 for LSTMs and CNNs, respectively), also indicating that the performance of RNNs decreases with the increasing complexity of the data set.

In conclusion, the LSTM architecture was the most suitable for two of the data sets, and the CNN architecture gave the best results for one data set. This does not imply that LSTMs are in general better than CNNs. Machine learning problems are data-dependent: for each data set, it is necessary to optimise hyperparameters of every architecture, compare the selected models in terms of the ability to learn, the effectivity of the training process, and the classification performance, and use the best model on such data set.

Bibliography

[1] Silver, D.; Huang, A.; et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, volume 529, 2016: pp. 484–503. Available from: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[2] Gómez-Bombarelli, R.; Duvenaud, D. K.; et al. Automatic chemical design using a data-driven continuous representation of molecules. *CoRR*, volume abs/1610.02415, 2016. Available from: http://arxiv.org/abs/1610.02415

[3] Gao, J.; Jamidar, R. Machine learning applications for data center optimization. *Google White Paper*, 2014.

[4] Xie, J.; Xu, L.; et al. Image Denoising and Inpainting with Deep Neural Networks. In *Advances in Neural Information Processing Systems 25*, edited by F. Pereira; C. J. C. Burges; L. Bottou; K. Q. Weinberger, Curran Associates, Inc., 2012, pp. 341–349. Available from: http://papers.nips.cc/paper/4686-image-denoising-and-inpainting-with-deep-neural-networks.pdf

[5] Dong, C.; Loy, C. C.; et al. Learning a Deep Convolutional Network for Image Super-Resolution. Cham: Springer International Publishing, 2014, ISBN 978-3-319-10593-2, pp. 184–199, doi:10.1007/978-3-319-10593-2_13. Available from: http://dx.doi.org/10.1007/978-3-319-10593-2_13

[6] Graves, A. Generating Sequences With Recurrent Neural Networks. *CoRR*, volume abs/1308.0850, 2013. Available from: http://arxiv.org/abs/1308.0850

[7] Abadi, M.; Andersen, D. G. Learning to Protect Communications with Adversarial Neural Cryptography. *CoRR*, volume abs/1610.06918, 2016. Available from: http://arxiv.org/abs/1610.06918

[8] Rojas, R. Neural networks: a systematic introduction. Springer Science & Business Media, 2013.

[9] Rusu, A. A.; Rabinowitz, N. C.; et al. Progressive Neural Networks. *CoRR*, volume abs/1606.04671, 2016. Available from: `http://arxiv.org/abs/1606.04671`

[10] LeCun, Y.; Bengio, Y.; et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, volume 3361, no. 10, 1995: p. 1995.

[11] Krizhevsky, A.; Sutskever, I.; et al. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[12] Scherer, D.; Müller, A.; et al. Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networks*, Springer, 2010, pp. 92–101.

[13] Sutskever, I.; Martens, J.; et al. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 1017–1024.

[14] Cho, K.; Van Merriënboer, B.; et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[15] Sutskever, I.; Vinyals, O.; et al. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[16] Pascanu, R.; Mikolov, T.; et al. On the difficulty of training recurrent neural networks. *ICML (3)*, volume 28, 2013: pp. 1310–1318.

[17] Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural computation*, volume 9, no. 8, 1997: pp. 1735–1780.

[18] Gers, F. A.; Schmidhuber, J.; et al. Learning to forget: Continual prediction with LSTM. *Neural computation*, volume 12, no. 10, 2000: pp. 2451–2471.

[19] Gers, F. A.; Schmidhuber, J. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, IEEE, 2000, pp. 189–194.

[20] Greff, K.; Srivastava, R. K.; et al. LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 2016.

[21] Graves, A.; Schmidhuber, J. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, volume 18, no. 5, 2005: pp. 602–610.

[22] Python Software Foundation. Python Language Reference, version 2.7. 2001–, [Online; accessed on 9th May 2017]. Available from: `http://www.python.org/`

[23] Kluyver, T.; Ragan-Kelley, B.; et al. Jupyter Notebooks—a publishing format for reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, 2016: p. 87.

[24] Merkel, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, volume 2014, no. 239, Mar. 2014, ISSN 1075-3583. Available from: `http://dl.acm.org/citation.cfm?id=2600239.2600241`

[25] gw000 2.0.2. gw000/keras-full. 2016, [Online; accessed on 9th May 2017]. Available from: `https://hub.docker.com/r/gw000/keras-full/`

[26] Chollet, F. Keras 2.0.2. `https://github.com/fchollet/keras`, 2015, [Online; accessed on 9th May 2017].

[27] Abadi, M.; Agarwal, A.; et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[28] Jones, E.; Oliphant, T.; et al. SciPy: Open source scientific tools for Python. 2001–, [Online; accessed on 9th May 2017]. Available from: `http://www.scipy.org/`

[29] van der Walt, S.; Colbert, S. C.; et al. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, volume 13, no. 2, 2011: pp. 22–30, doi:10.1109/MCSE.2011.37, `http://aip.scitation.org/doi/pdf/10.1109/MCSE.2011.37`. Available from: `http://aip.scitation.org/doi/abs/10.1109/MCSE.2011.37`

[30] Hunter, J. D. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, volume 9, no. 3, 2007: pp. 90–95, doi:10.1109/MCSE.2007.55.

[31] Pedregosa, F.; Varoquaux, G.; et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, volume 12, no. Oct, 2011: pp. 2825–2830.

[32] Bengio, Y. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, Springer, 2012, pp. 437–478.

[33] Srivastava, N.; Hinton, G.; et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, volume 15, 2014: pp. 1929–1958. Available from: `http://jmlr.org/papers/v15/srivastava14a.html`

[34] Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[35] Cui, Z.; Chen, W.; et al. Multi-scale convolutional neural networks for time series classification. *arXiv preprint arXiv:1603.06995*, 2016.

[36] Chen, Y.; Keogh, E.; et al. The UCR Time Series Classification Archive. July 2015, `www.cs.ucr.edu/~eamonn/time_series_data/`.

[37] Lichman, M. UCI Machine Learning Repository. 2013. Available from: `http://archive.ics.uci.edu/ml`

[38] Ratanamahatana, C. A.; Keogh, E. Everything you know about dynamic time warping is wrong. In *Third Workshop on Mining Temporal and Sequential Data*, Citeseer, 2004.

[39] Holland, J.; Kemsley, E.; et al. Use of Fourier transform infrared spectroscopy and partial least squares regression for the detection of adulteration of strawberry purees. *Journal of the Science of Food and Agriculture*, volume 76, no. 2, 1998: pp. 263–269.

[40] Kudo, M.; Toyama, J.; et al. Multidimensional curve classification using passing-through regions. *Pattern Recognition Letters*, volume 20, no. 11, 1999: pp. 1103–1111.

Appendix A

Acronyms

ANN Artificial Neural Network

CNN Convolutional Neural Network

CPU Central Processing Unit

GPU Graphics Processing Unit

LSTM Long Short-Term Memory Neural Network

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

SGD Stochastic Gradient Descent

Appendix B

Contents of enclosed CD

```
 readme.txt ....................... the file with CD contents description
 src ....................................the directory of source codes
   impl ....................................implementation sources
   thesis ..............the directory of LaTeX source codes of the thesis
 text .......................................... the thesis text directory
   DP_Waller_Jakub_2017.pdf ...........the thesis text in PDF format
```