FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reproduzierbarkeit von Deep Learning Datenanalysen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Business Informatics

eingereicht von

Bojan Čavić, BSc
Matrikelnummer 01251071

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber

Wien, 4. Oktober 2018

Bojan Čavić	Andreas Rauber

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reproducibility of Deep Learning Data Analyses

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Bojan Čavić, BSc
Registration Number 01251071

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber

Vienna, 4th October, 2018

Bojan Čavić	Andreas Rauber

Erklärung zur Verfassung der Arbeit

Bojan Čavić, BSc
Liebenstraße 33/2/15 1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. Oktober 2018

Bojan Čavić

Danksagung

Ich bedanke mich bei Ao.univ.Prof. Dr. Andreas Rauber für die Anregung zum Thema und darüber hinaus für die intensive Betreuung und Hilfe.
Ein großes Dankeschön gebührt allen meinen Freunden für ihren Glauben an mich und ihr Verständnis während der Erstellung dieser Arbeit.
Schließlich möchte ich mich bei meinen Eltern, Brüdern und meiner ganzen Familie bedanken, die mich während der Erstellung dieser Diplomarbeit aber auch während der ganzen Studienzeit unterstützt haben. Bedanken möchte ich mich auch, dass sie immer an mich geglaubt und nie an mir gezweifelt haben. Diesen Erfolg widme ich ihnen.

Acknowledgements

I would like to thank Ao.univ.Prof. Dr. Andreas Rauber for the suggestion on the subject of this thesis and for the intensive assistance and help.
A big thank you to all my friends for their belief in me and their understanding while creating this work.
Finally, I would like to thank my parents, brothers, and my family, who supported me during the creation of this thesis, but also throughout the entire study period. I would also like to thank you for always believing in me and never doubting me. I dedicate this success to you.

Kurzfassung

Deep Learning floriert in den letzten Jahren und wird dank der derzeit verfügbaren Rechenleistung von Computersystemen immer mehr eingesetzt. Große IT-Unternehmen wie Google oder Facebook nutzen Deep Learning Algorithmen in ihrem Tagesgeschäft. Daher ist die Reproduzierbarkeit von Forschungsarbeiten, die Deep Learning Algorithmen beinhalten, ein entscheidender Faktor. Diese Masterarbeit konzentriert sich auf die Analyse des Einflusses verschiedener Betriebssysteme sowie verschiedener Deep Learning Frameworks. Zu diesem Zweck wird das gleiche Deep Learning Modell in drei sehr populären Frameworks (TensorFlow, Theano und Deeplearning4J) erstellt und ausgeführt. Darüber hinaus werden verschiedene Versionen dieser Frameworks berücksichtigt, da einige von ihnen möglicherweise wichtige Methoden auf andere Weise implementieren. Danach wird das Modell auf sieben Betriebssystemversionen ausgeführt. Zusätzlich werden verschiedene Versionen der verwendeten Ausführungsplattform (Python und Java) berücksichtigt. Die erhaltenen Modellergebnisse werden analysieren und testen, ob die Ergebnisse durch den Ausführungskontext beeinflusst werden.

Abstract

Deep Learning is thriving in recent years and finding increasing deployment thanks to the currently available processing power of computer systems. Big IT companies like Google or Facebook use Deep Learning algorithms in their daily business. Therefore, the reproducibility of research based upon Deep Learning algorithms is a crucial factor. This master thesis will focus on analyzing the influence of different operating systems as well as different Deep Learning frameworks. For this purpose, the same Deep Learning model is constructed and executed in three very popular frameworks (TensorFlow, Theano and Deeplearning4J). Further, different versions of these frameworks are considered as maybe some of them may implement crucial methods in a different way. Afterwards, the model is executed on seven operating system versions. Additionally, different versions of the used execution platform (Python and Java) will be considered. Finally, this thesis focuses on analyzing all obtained model results and testing if the results are significantly different when changing the execution context.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Computer science is a very young discipline, but indispensable for our society. The involvement of information technology in almost every area of life has been triggered by the so-called digital revolution and has led to the information age in which we now live [Sch18]. The scientific research in computer science can be regarded as the center of innovation for new digital technologies. It is common practice that some research is built upon already existing research which try to further develop the underlying idea of the topic. The effectiveness and performance of scientific research can on one hand be validated by theoretic properties of algorithms or methods, or on the other hand by experimentation. A major challenge scientists face today concerns the reproducibility of computer science experiments, as most computational experiments are specified only informally in papers, where experimental results are briefly described [FFR16].

Reproducibility concern of scientific research has been steadily rising in recent years as numerous results of experiments could not be replicated [GFI16, Aar15] . It is important to note that terms such as reproducibility, replicability or reliability have led to confusion as they are not standardized. According to a U.S. National Science Foundation (NSF) subcommittee on replicability in science, reproducibility is the ability of a researcher to duplicate the results of a prior study by using the same materials as the original researcher [GFI16]. But reproducible results are not just beneficial to other researchers, they are also beneficial to the initiator of the research, as by making an experiment reproducible the researcher is forced to document execution pathways and makes them analyzable. Another advantage is that reproducible experiments can help to get familiar with the problem and tools that were used [FFR16].

As, Deep Learning is thriving in the last years and is employed more and more, the reproducibility of research studies considering Deep Learning algorithms is a crucial factor. The reproducibility of Deep Learning algorithms can be influenced by many factors, not only by the Deep Learning architecture with its hyperparameters. Therefore, the whole execution stack needs to be considered.

The execution stack considers the following parameters:

- underlying hardware
- operating system
- execution platform
- Deep Learning framework
- Deep Learning model
- dataset

Thereby, every parameter can have an influence on the result of a Deep Learning algorithm, as every operating system or framework may implement the same functionality in a different way, e.g. different implementation of core OS libraries. Knowing this, this master thesis will put the reproducibility factor in the center of the investigation in regard to the execution stack. Therefore, this study will explain what happens with results of research studies based on Deep Learning algorithms when they are reproduced on different operating systems, execution platforms and using different Deep Learning frameworks.

1.2 Problem statement

Beside the fact that Deep Learning models are computationally intensive, an interesting thing about these models is that they, as opposed to standard data analysis models, contain many test data points and probably a small change of the model could lead to a different output that is statistically significant. Research studies that consider the reproducibility of Deep Learning algorithms mostly concentrate on a concrete definition of the underlying neural network implementation [DRF18]. Even if an exact definition of the neural network architecture is important, the complete execution stack should be considered. Different parameters can have an influence on Deep Learning model results, including the operating system version, execution platform version or the Deep Learning framework version.

Therefore, the goal of this thesis is to develop and implement Deep Learning experiments and analyze effects when reproducing them concentrating on the whole execution stack. All investigated experiment changes are structured according to a model called PRIMAD [FFR16]. The PRIMAD model has the aim of categorizing various types of reproducibility, where PRIMAD stands for – (P)latform, (R)esearch Objective, (I)mplementation, (M)ethod, (A)ctor and (D)ata. This thesis will be built upon on the PRIMAD model where aspects such as the Platform, Operating System and the Implementation are changed while others remain constant. Constants that are kept fixed throughout the experiments are the Deep Learning model with all of its parameters except the random number initialization, the used dataset and the hardware.

The first problem that is covered in this master thesis analyzes the influence of different Deep Learning frameworks while reproducing the same model. Implementing a Deep Learning model in different environments requires changing the parameters, dependencies and libraries used in each environment respectively. Also, every framework uses a different syntax which has to be adapted from framework to framework. Follow up experiments analyze the influence of different framework versions as well as a different execution platform. For some of the frameworks and execution platforms, changing the version requires an adoption of code. The Deep Learning model is adapted to fit the respective version.

The second problem that is covered in this master thesis analyzes the influence of the operating system on the results. In this master thesis, the model is tested on Windows, Linux and Mac OS in different versions. Each operating system is run on a virtual machine, except for the original system (Mac OS High Sierra). It is possible that a change of the experiment configuration can lead to significantly different results, which in fact, should not occur as this would require future studies to expose the complete setup used for the performed experiment. Key aspect of this investigation will analyze if a different configuration will lead to statistically significant results.

The main challenges faced in this master thesis are:

1. The implementation of the same Deep Learning model in different Deep Learning frameworks as well as for different framework versions.
2. Preparation of the input dataset to be usable in every Deep Learning framework.
3. Set up of required environments for all operating systems.
4. Analysing the results for all possible setup combinations.

1.3 Research questions

The main purpose of this master thesis is to investigate if certain parameters of the execution stack have a statistical significant influence on Deep Learning model results while reproducing them. It is important to note that the random seed is a crucial parameter for the reproducibility of those kind of algorithms, as without its specification the Deep Learning model, or in general Machine Learning algorithms, would always deliver slightly different results. This study focuses on following research questions:

1. Is it possible to build the same Deep Learning architecture in different frameworks?
2. Is the performance difference of the same Deep Learning model statistically significant when using:
   a) different frameworks?
   b) different framework versions?
   c) different execution platform versions?
   d) different operating systems?
   e) different operating system version?
3. Does the random seed have an impact on Deep Learning model results?

Another goal of this study is to encourage the Deep Learning community to concern reproducibility of experiments as crucial factor. When publishing certain results, it should be possible for other researchers to reproduce those and facilitate the verification of published results and increase trustwortheness. In fact, to be able to arrange this, an exact documentation of the whole experiment is needed, starting from model specific characteristics, such as the Deep Learning architecture, and ending with global information, such as the operating system and its version.
To which level of detail these aspects need to be documented (especially regarding the description of the Deep Learning architecture and the data preparation) is, however, not a focus of this study. Insights into this aspect of reproducibility (i.e. the 'actor' dimension of the PRIMAD model) is increasingly studied in dedicated reproducibility tracks of conferences. Furthermore, the potential impact of different hardware (such as Intel vs. AMD, training the model on GPUs) was not evaluated.

1.4 Methodological approach

The methodological approach used for this thesis is experimentally oriented and comprises following parts:

In the first step, the experiments are defined. Each experiment differs from the others by changing one parameter from the execution stack. This results in a tree like structure where one branch defines one experiment and the whole tree comprises all possible combinations of parameter settings. The input dataset is constant across all experiments. The different Deep Learning frameworks and their different versions are other parameters of the execution stack. The execution stack parameters that are changed for this thesis are:

1. Deep Learning framework as well as different versions of these frameworks,
2. the execution platform, i.e. different Java Virtual Machines, Java Development Kit versions and Python stacks,
3. the operating system and different versions of these operating systems.

Frameworks that were explored for this thesis are:

i) TensorFlow[1],

ii) Theano[2],

iii) Deeplearning4J[3].

Also, the operating systems with their different versions and the execution platform are parameters of the execution stack. Different operating systems are installed on the same hardware while using Mac OS High Sierra as base operating system. For this master thesis seven different operating systems are explored and installed on virtual machines:

[1] https://www.tensorflow.org
[2] http://deeplearning.net/software/theano/
[3] https://deeplearning4j.org

- Mac
  - Mac OS High Sierra
- Linux
  - Linux Fedora 25
  - Linux Mint 18.3
  - Linux Ubuntu 16.04
- Windows
  - Windows 7
  - Windows 8
  - Windows 10

As platforms progress during the years certain code implementation progress as well. As already mentioned, different execution platforms are considered in this thesis. For Python-based Deep Learning frameworks version 2.7.14 and 3.5.4 are considered. And for the only framework using Java (i.e. Deeplearning4J) the JDK versions, 7 and 8 are considered. Besides the standard Oracle JVM[4] that is used almost everywhere, the ZULU JVM[5] is also considered in this master thesis. The execution stack is represented in Figure 1.1. All possible combinations are shown in this representation. A total of **519** (19 configurations that are run once on seven operating systems + 19 equivalence classes that are run 20 times + 6 DL4J runs with more epochs = 133 + 380 + 6 = 519) experiments are conducted to analyze the influence of various parameters to Deep Learning results.

The output of every experiment is a log file. The file contains model metrics, i.e. accuracy, precision, recall and F1-score and general system information such as used framework, operating system and the used random seed. In the third step, the results of the experiments are statistically analyzed. For this analysis, Python and the Python package SciPy[6] is used. All network implementations, dataset, trained models and raw results are available on Zenodo[7] (DOI: 10.5281/zenodo.1414542).

[4] https://docs.oracle.com/javase/6/docs/technotes/guides/vm/index.html?intcmp=3170
[5] https://www.azul.com/downloads/zulu/
[6] https://www.scipy.org
[7] https://zenodo.org

1.5 Structure of the work

In Chapter 1, the motivation for this master thesis is stated and the goal defined. Also, the problem that should be solved is described in this Chapter. Further, the methodological approach to solve the problem is outlined. In Chapter 2, studies that as main focus have the same type of problem are analyzed and a brief history of reproducibility in general is given. Chapter 3 focuses on the execution of every experimental scheme in detail. In this Chapter the used concepts, data models and experimental methods that are used for the experiments are described. First, a detailed explanation of the used Deep Learning model is provided as well as the dataset that is used. Second, the operating systems, the execution platforms and the Deep Learning frameworks are stated. In the last part of this section, the analysis methods that are used are discussed. Chapter 4 comprises the statistical analysis of results obtained from all conducted experiments. Furthermore, the results of this statistical analysis are explained. Finally, Chapter 5 summarizes and provides a critical reflection. A part of this section discusses possible future work that can be taken to further investigate reproducibility in Deep Learning.

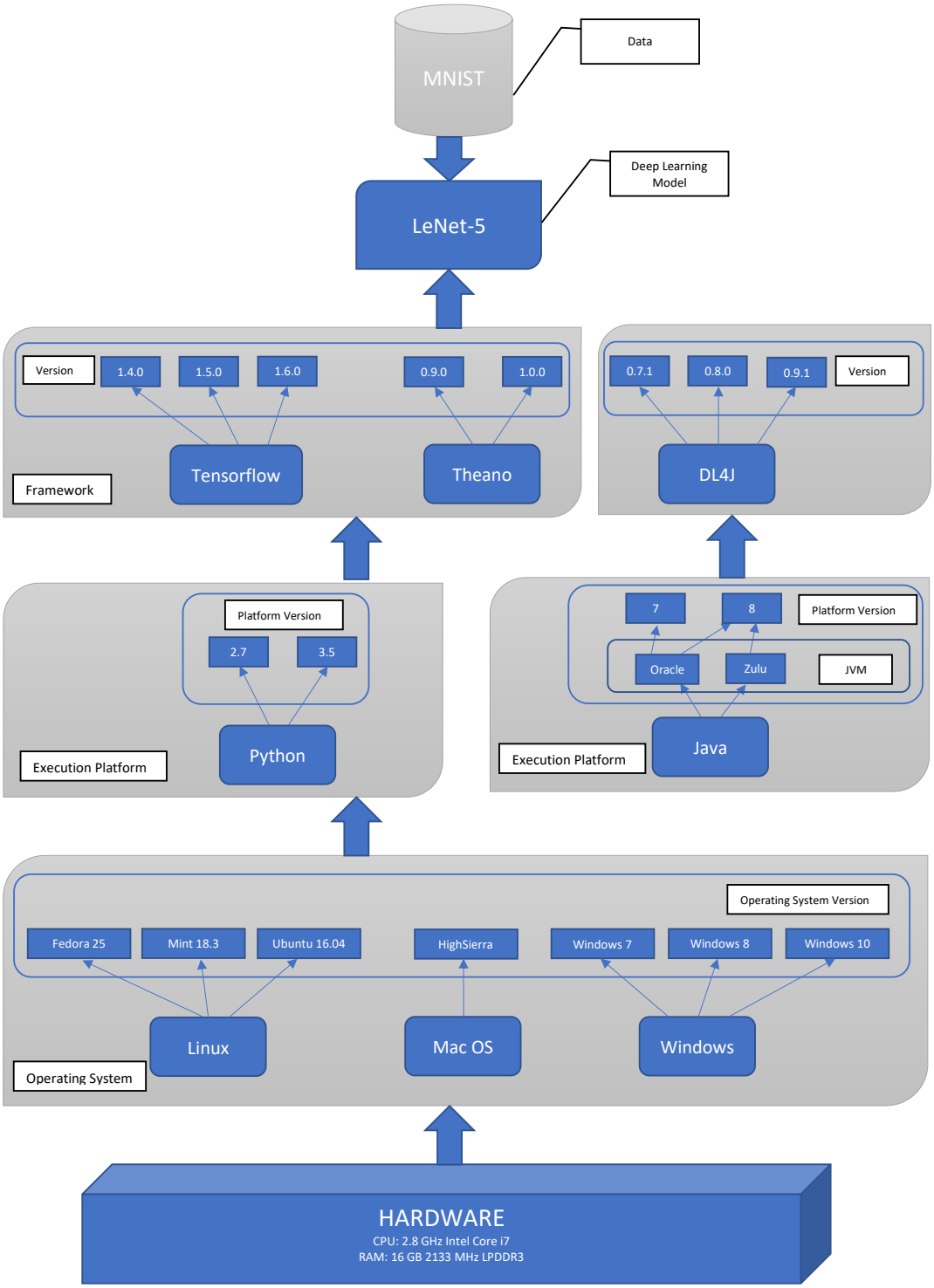

Figure 1.1: Execution stack that underlies the experiments.

CHAPTER 2

State of the art

2.1 Literature studies

Reproducibility of research does not only concern the discipline of computational science, but science in general. The first scientist that rose awareness about reproducibility in science was Robert Boyle in the 1660s. To make his research reproducible he tried to build it up on three pillars in his work. First, he gave detailed explanation of used equipment and materials. Second, many of his experiments were conducted in public and later on the names and qualifications of witnessing scientists were published along with the results. Third, besides the used methods, he described his experiments' circumstances and settings, failures and much more [Sha84].
Reproducibility of computational research does not only concern one computational science discipline, researchers across a range of computational science disciplines have been calling for reproducibility. As in that way, a minimum standard for assessing the value of scientific claims can be obtained [LGGS07, Rob05]. Reproducible research is essential as it is sometimes the only way to validate results. Researchers from various fields, such as geoscience, neuroscience, bioinformatics, applied mathematics, psychology, and computer science are calling for data and code to be made available to facilitate the reproduction of published computational results [Sto12].

In [FFR16] the author claims that the standard of reproducibility calls for the data and the computer code that were used to analyze the data to be made available to others. The article suggests a reproducibility standard, to fill the gap in the scientific evidence-generating process between full replication of a study and no replication. A number of possibilities between these two end points is possible, see Figure 2.1.

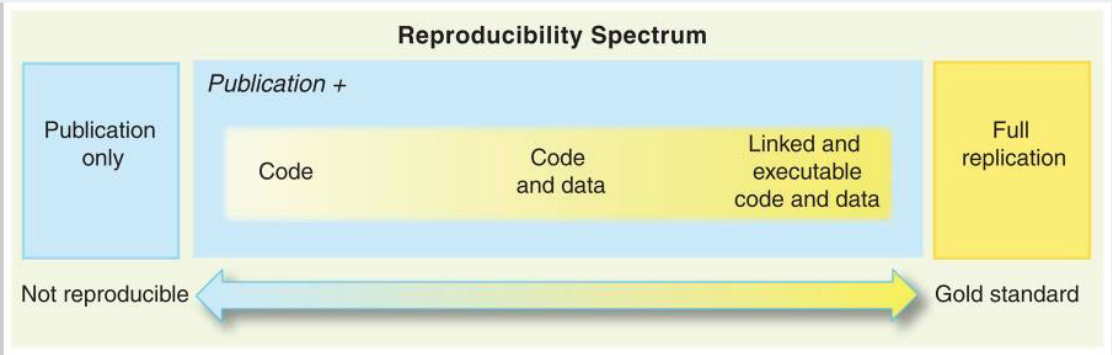

Figure 2.1: Reproducibility standard [Pen11]

The basis of this standard is a detailed log file which in theory, every computational experiment should maintain. In case, these computer codes are available a greater level of detail regarding the analysis can be guaranteed as opposed to experimental descriptions using natural language [Pen11]. The article states, that the biggest barrier to reproducible research is the lack of integrated culture, but to develop such a culture it will require time and sustained effort from the scientific community. Peng says, that journals can play an important role. For example, the Biostatistics journal, has already implemented a reproducibility policy, where authors can submit their code or data to the journal for posting and requesting a 'reproducibility review', in which an associate editor for reproducibility runs the submitted code on the data and then verifies if the code produces the published results.

Another paper published in 2015 [Boe15] says that, reproducibility seems more straight forward than replicating physical experiments, hence the complex and rapidly changing nature of computer environments makes such work a serious challenge. The paper deals with the fact that code developed for one research project cannot be successfully executed or extended by subsequent researchers. Beside cultural challenges, there are four technical challenges that can contribute to impede reproducibility. The first challenge deals with installing or building software to run the research code. The second challenge concerns missing documentation that is needed to install or run code associated with research. Another challenge are updates, which can change the results generated by the code. The last challenge regards the entry barriers of learning new tools and approaches, such as workflow systems or virtual machines. The solution to all these challenges given in paper is DOCKER[1], an open source framework that builds on many long- familiar technologies from operating systems research [Boe15].

[1] https://www.docker.com

In [NGP09] reproducible descriptions of neuronal network models are addressed. The description of neuronal network models is the main focus of this study. An unstandardized model description practice not only hinders the critical evaluation of network models but also their re-use. In this study 14 research papers proposing neuronal network models where analyzed, where widely varying approaches of model descriptions where observed as well as a great variation of graphical representations. The authors believe that a good model description practice together with other initiatives on data-, model-, and software sharing may lead to a more promising exchange of ideas among scientist. With this goal in mind the study proposed guidelines for the organization of publications, a checklist for model descriptions, templates for tables and guidelines for diagrams of networks.

[DRF18] also focuses on reproducibility of neural network models. In this study it is shown that an uncertainty about only two components of the neural network model and no precise description of the training process, it is not possible to reproduce the original experimental results [DRF18]. The study emphasizes that the three most important elements in artificial neural network models are:

- the structure of the nodes,
- the topology of the network and,
- the learning algorithm [Roj96].

Hence, reproducible research should convey these three aspects in such detail, that the model architecture can be implemented identically to the original model. Therefore, an explicit description of the neural network structure, a detailed description of the training process and access to the data. The study attempts to reproduce the Gated Self-Matching Network [WYW+17] and uses the SQuAD dataset. The re-implemented neural network could not even achieve performance close to the original results. Even if the original research provides insights into the architecture of the model it is missing information to implement and train the model [DRF18].

For this thesis, the report" Reproducibility of Data-Oriented Experiments in e-Science" [FFR16] is one of the most important fundamentals. The report documents the program and the outcomes of Dagstuhl Seminar held in Dagstuhl, Germany, from the 24th to the 29th of January of 2016, where experts from various sub-fields of computer science took part to create a joint understanding of the problems of reproducibility of experiments. It discusses the reproducibility of experiments in e-Science and points out that experimental results depend on the input data, settings for input parameters, and potentially on characteristics of the computational environment where the experiments were designed and run. One result of the seminar was the PRIMAD model with the aim of categorizing various types of reproducibility.

PRIMAD stands for – (P)latform, (R)esearch Objective, (I)mplementation, (M)ethod, (A)ctor and (D)ata. For example, to test the portability, stability, or platform-independence of the experiment (R), (M), and (I) can be fixed and the platform (P) can be changed to (P'). By changing some of these variables, different kind of knowledge can be gained, see Figure 2.2.

Label	Data		Platform / Stack	Implementation	Method	Research Objective	Actor	Gain
	Parameters	Raw Data						
Repeat	-	-	-	-	-	-		Determinism
Param. Sweep	x	-	-	-	-	-		Robustness / Sensitivity
Generalize	(x)	x	-	-	-	-		Applicability across different settings
Port	-	-	x	-	-	-		Portability across platforms, flexibility
Re-code	-	-	(x)	x	-	-		Correctness of implementation, flexibility, adoption, efficiency
Validate	(x)	(x)	(x)	(x)	x	-		Correctness of hypothesis, validation via different approach
Re-use	-	-	-	-	-	x		Apply code in different settings, Re-purpose
Independent *x* *(orthogonal)*							x	Sufficiency of information, independent verification

Figure 2.2: PRIMAD Model: Reproducibility of Data-Oriented Experiments in e-Science [FFR16]

In the following various aspects of an experiment that can be changed are discussed.

Platform: By changing the platform, the independency or the portability of an experiment can be tested. In that way, a wider adoption or higher stability may be gained.

Research Objective: Changing the hypothesis may be regarded as re-using or re-purposing an earlier experiment. This, may allow science to progress faster.

Implementation: The correctness of the previous implementation can be verified by changing the actual implementation. Higher efficiency, higher adoption, or a broader set of execution platform are some advantages that may be gained by changing the implementation. But, an implementation change may incur a platform change as well.

Method: To be able to validate the correctness of a hypothesis a different method can be used. By changing the method, one should keep in mind that also an implementation change will incur, and probably also a platform change.

Actor: Changing the actor is orthogonal to all other changes, allowing independent verification of characteristics and also whether the provided information is sufficient to achieve independent verification.

Data: Data can be sub-divided into raw input data and parameters. By changing input data already made statements can be verified across a larger part of the input space. A change of parameters allows the determination of robustness of sensitivity of an experiment.

2.2 Comparison and summary of existing approaches

As discussed in the previous section, there are many studies that try to address the problem of reproducibility. Many of them focus on creating an overall standardized understanding of what reproducible research is and how to achieve it. As discussed in [DRF18] a study concerning Deep Neural Network has to include a comprehensive documentation of all necessary information needed to reproduce the original results and even by missing only one or two details could lead to totally different results.

For this master thesis the PRIMAD model is used to describe which part of the experimental execution stack is changed while the others, including the research objective, the data and the implementation remain the same. This master thesis will raise interest in reproducible research especially in the Deep Learning community.

CHAPTER 3

Methodology

In this Chapter, the methodology and concepts used in this master thesis are described in detail. Beginning with describing the experimental procedure in Section 3.1. To understand what is happening in this study an overview of Deep Learning is given in Section 3.2, but with main focus on the concepts and theories that are essential for the Deep Learning model that is used for this study.In Section 3.3, the Deep Learning model underlying this study is discussed in detail. Section 3.4 describes the dataset that is used for training and testing the model. Section 3.5 describes the execution platforms and frameworks that are used as well as their versions and possibly important changes from different versions. In Section 3.6, different operating systems are discussed and described. Finalizing all with a detailed description of the statistical methods used for analysis in Section 3.7.

3.1 Experiment design

In order to be able to analyze the influence of different factors, such as the operating system or Deep Learning framework, on different Deep Learning results an exact experimental scheme was designed. The process that is followed during all experiments can be seen in Figure 3.1. First, the same Deep Learning architecture is built in all three frameworks including their respective versions, with same hyperparameters and random seed. Further, framework specific adaptions are considered. Each experiment or framework uses the same dataset as input. The dataset it split into a training set with 60.000 examples and a test set with 10.000 examples. The first step regarding the input data is to convert it into a format that is usable by all three frameworks. The original dataset is using the IDX file format, which is a simple format for vectors and multidimensional matrices of various numerical types[1].

[1] http://yann.lecun.com/exdb/mnist/

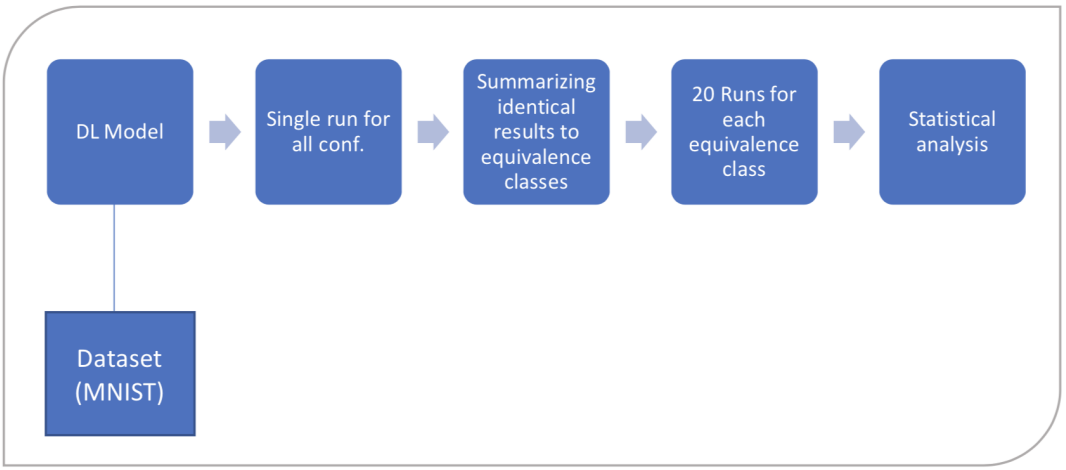

Figure 3.1: Experimental process

In order to be able to use this dataset in all frameworks it needs to be converted programmatically into the CSV file format (the used code is listed in Appendix A.1). Second, for a specific operating system the model is run once with all possible configurations in accordance to Figure 1.1. Third, if configurations delivered the exact same results they are summarized to equivalence classes. Afterwards, a representative from each equivalence class is chosen and run 20 times where only the random seed is changed to understand the statistical variance caused by merely changing the input data representation order in the training process. Finally, the results from all representatives are statistically analyzed to infer if they originate from the same distribution.

3.2 Used concepts

In this Section, the general concepts that are used in this master thesis are discussed. The concepts in general are not crucial for the end result of the thesis but are necessary to establish the same experimental setup and understanding. Main focus of this Section is the description of the term 'Deep Learning' in general to be further able to understand the Deep Learning model which builds the basis for this master thesis.

3.2.1 Deep Learning

Even if the term Deep Learning seems to be relatively new, the underlying concepts and techniques existed for decades. The basis of Deep Learning are artificial neural networks. Artificial neural networks are inspired by biological neural networks that form animal brains.

Deep learning models are generally composed of multiple processing layers with the goal to learn representations of data, where each layer transforms the representation at one level into a representation at a higher, slightly more abstract level. These transformations are achieved by composing simple non-linear modules. Hence, higher layers amplify aspects of the input data that are important and suppress irrelevant aspects. It is important to notice, that these layers of features are not designed by humans but are learned from data using learning procedure [Sch15]. Mostly, a fixed-sized input is mapped to a fixed-sized output, where units that are not in the input or output layer are called hidden units. According to [Sch15], hidden layers can be seen as distorting the input in a non-linear way so that categories become linearly separable by the output layer. An important algorithm that Deep Learning models are using is the Backpropagation algorithm. In general, this algorithm tells the model how to change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer [Sch15].

Of, course there exist different types of neural networks and basically, they can be divided into two groups, single-layer networks and multi-layer networks. As the name already says, single-layer networks consist of only one layer and is the simplest kind of a neural network. Probably, the most famous and even the first model that could actually learn the categories defining weights was the perceptron [Ros58]. This kind of neural network has a threshold activation functions and were applied to classification problems, in which the inputs were usually binary images of characters [CBBHB95]. On the other side, multi-layer networks consist of multiple layers of units which are interconnected. This means, that each neuron in one layer is directly connected to neurons of subsequent layers. Nowadays, mostly multi-layer networks are used as more promising architectures can be build which in most cases lead to better results. Besides classifying neural networks regarding their number of layers, they can also be divided into feed-forward- and recurrent neural networks.

Feed-forward neural networks obtained this name, because information flows through all connections without feedback connections or cycles in which outputs of the model are fed back into itself [GBC16]. Graphically such a network can be represented by a directed acyclic graph, shown in Figure 3.2. On the other side, when feedback connections are included then the neural network is called a recurrent neural network and can be represented by a directed graph, as shown in Figure 3.3.

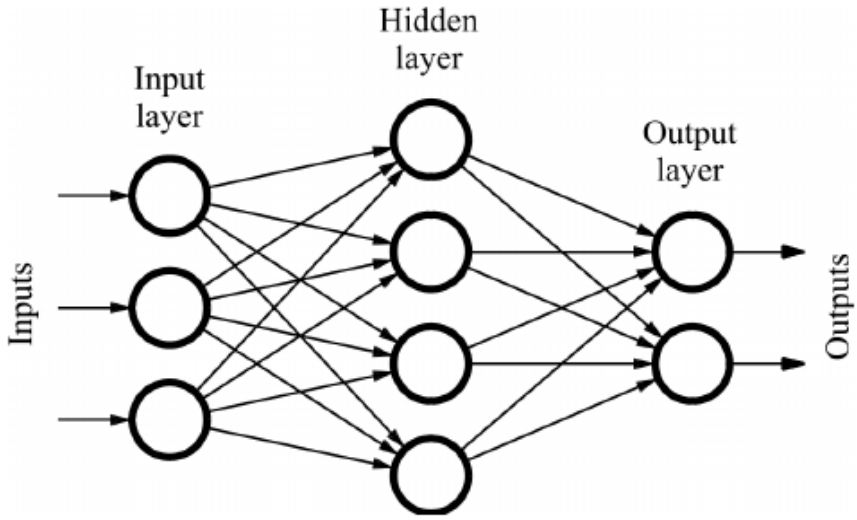

Figure 3.2: Example feed-forward neural network [QD11]

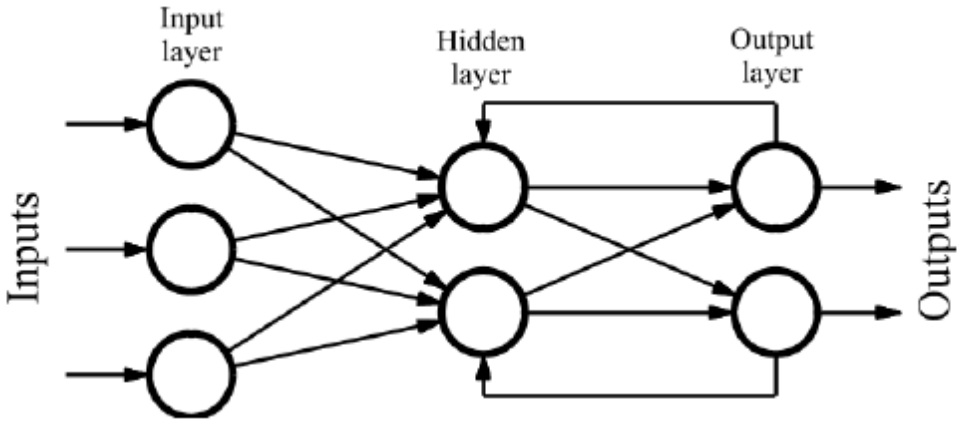

Figure 3.3: Example recurrent neural network [QD09]

Convolutional Neural Networks

The Deep Learning model underlying this master thesis is a so-called convolutional neural network. Convolutional neural networks (CNN) belong also to the class of feed-forward neural networks and are used for processing data that has a known grid-like topology [GBC16]. Convolutional neural networks have been successfully applied for image classification tasks. Convolutional neural networks combine three architectural ideas, respectively local connections, shared weights and sub-sampling or pooling [LBBH98].

Convolutional neural networks are typically structured in stages, where the first few stages are composed of convolutional layers and pooling layers. As [Sch15] describe, units in a convolutional layer are organized in feature maps, within each unit is connected to feature maps of the previous layer through a set of weights. The weighted sum is then passed through a non-linear function. It is very important that all units in a feature map share the same weights. The idea behind pooling-, or sub-sampling-layers is to merge semantically similar features into one [Sch15]. With local connections elementary visual features such as edges or corners can be extracted. These features are then combined by the subsequent layers to be able to detect higher-order features [LBBH98].
A typical convolutional neural network for recognizing handwritten characters is the popular LeNet model created by Yann Lecun, shown in Figure 3.4 [LBBH98].

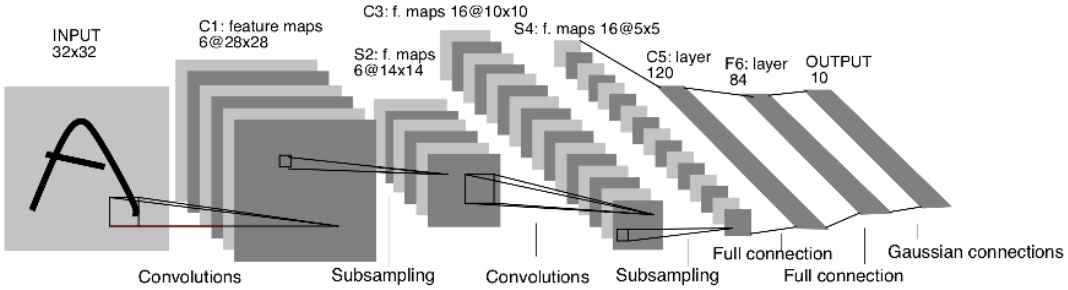

Figure 3.4: LeNet-5 architecture [LBBH98]

3.3 Methods and Models

The convolutional neural network model architecture that is used in this master thesis is based on the popular LeNet-5 architecture in regard to the number of layers, the types of layers and the kernal sizes of each layer.. This architecture is a very popular convolutional neural network mostly used for image processing. LeNet-5 comprises 7 layers, without the input layer, with two convolutional layers, two sub-sampling layers and three fully connected layers, including the output layer. The original LeNet-5 model uses images with 32 x 32 pixels and the model used in this thesis uses images with 28 x 28 pixels as the official MNIST dataset contains images with 28 x 28 pixels.

The architectural design of the model is shown in Figure 3.5.

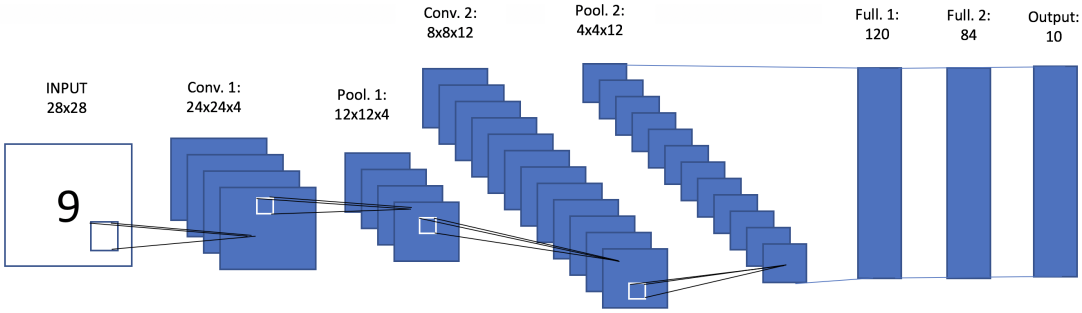

Figure 3.5: Convolutional neural network archtecture of model that is used in this thesis

3.3.1 Computations within a layer

The output of each layer is calculated as a dot product between the input vector and the weight vector, to which a bias is added. The weighted sum, denoted a_i for unit $_i$, is then passed through a sigmoid squashing function f to produce the state of unit i, denoted by x_i.

$$x_i = f(a_i) \tag{3.1}$$

The original LeNet-5 model uses as squashing function a scaled hyperbolic tangent:

$$f(a) = Atanh(Sa) \tag{3.2}$$

The downside of this squashing function is that it is mostly more time consuming with gradient descent [KSH12]. As the goal of this master thesis is not to propose a model that achieves a small error rate, the non-saturating nonlinearity f(x) = max(0, x) is used. This non-linearity is called Rectified Linear Units, or ReLUs. Deep convolutional neural nets with this activation function train several times faster than equivalents with tanh units and is thus widely used in current Deep Learning settings [KSH12].

The output layer uses a softmax non-linearity to predict the probability distribution over classes given in the dataset [KGB14]. Besides the activation function, the weights are also important. They should be initialized in a way that promote learning. Wrong weight initialization can probably make gradients too large or too small, which can make it difficult to update the weights in the long run. Small weights could lead to small activations, whereas large weights may lead to the opposite. The model specified for this thesis uses the Xavier weight initialization, where the weights are normal distributed and following standard deviation is used:

$$\sigma(W) = \sqrt{2/(x_{\text{in}} + x_{\text{out}})} \tag{3.3}$$

Where x_{in} is the number of units in the previous layer and x_{out} is the number of units in the following layer. The purpose of this weight initialization is to maintain same distribution of activations, meaning that they are not too small or too large [GB10].

The optimization of the weights is done according to the Stochastic Gradient Decent, or SDG, algorithm. SDG is a simplification of the Gradient Descent algorithms which goal it is to minimize the empirical risk $E_n(f_w)$, that measures the training set performance, using gradient descent (GD). Each iteration updates the weights w on the basis of the gradient of $E_n(f_w)$ with the following formula:

$$w_{t+1} = w_t - \gamma \frac{1}{n} \sum_{i=1}^{n} \nabla_w Q(z_i, w_t), \tag{3.4}$$

where γ is an adequately chosen gain and $Q(z_i,w_t)$ is the loss function, $l(\hat{y},y)$, that measures the cost of predicting $\hat{y}$ when the actual answer is y. When the initial estimate w_0 is close to the optimum and when the gain γ is sufficiently small, this algorithm achieves linear convergence [Bot10]. SDG does not compute the gradient of $E_n(f_w)$ exactly, but tries to estimate this gradient for each iteration on the basis of a single randomly picked data point z_t, according to this formula:

$$w_{t+1} = w_t - \gamma_t \nabla_w Q(z_i, w_t). \tag{3.5}$$

The stochastic process depends on the examples that are randomly picked at each iteration [Bot10].

3.3.2 Explaination of each layer

Figure 3.5 shows the principal archtitecture of the convolutional neural network that is used in this master thesis. The first layer is a convolutional layer with four feature maps, where each feature map is of size 24x24, where a feature map is defined as the output of the previous layer [GBC16]. Each unit in each feature map is connected to a 5x5 neighborhood in the input. Which means that every unit in this layer receives inputs from a set of units located in a neighborhood of the previous layer. In that way neurons can extract elementary feature such as corners or edges. These features are combined by subsequent layers in order to detect higher-order features [LBBH98]

The second layer is a sub-sampling layer with four feature maps of size 12x12. Each unit in each feature map is connected to a 2x2 neighborhood in the corresponding feature map of the first layer. Note, that the sub-sampling layer divides the size of the first layer in half.

The third layer is again a convolutional layer, but now with 12 feature maps of size 8x8 and where each unit is connected to a 5x5 neighborhood of the previous layer.

The fourth layer is a sub-sampling layer with 12 feature maps of size 4x4. Each unit is connected to a 2x2 neighborhood of the third layer. Similar to the second layer, this layer also divides the size of the previous layer in half.

The last three layers are fully connected layers. Fully connected means that every neuron in one layer is connected to every neuron in the previous layer. The first fully connected layer gets as input 4*4*12 = 192 units and outputs 120. The next fully connected layer reduces then the number from 120 to 84. The final layer minimizes the number of units to ten, the number of different classes of the dataset.

3.3.3 Hyperparameters

As every Deep Learning model needs some specific parameters to be specifies for training purposes, the model used in this master thesis specifies the following parameters:

- *Random Seed*, important for reproducibility,
- *Epochs* that the model is trained,
- *Batch Size*, the number of samples that are going to be propagated through the network,
- *Learning Rate* with which the model is trained.

3.4 MNIST Data Set

MNIST[2] is a freely available dataset of handwritten digits (digits from 0-9) and has become a standard for testing Machine- and Deep Learning algorithms [Den12].

The MNIST, Modified NIST set, dataset was constructed from the National Institute of Standards and Technology's, or NIST's, Special Database 3 (SD-3) and Special Database 1 (SD-1). These databases contain binary images of handwritten digits, where NIST originally used SD-3 as training set and SD-1 as test set. SD-1 contains 58.527 digit images written by 500 different writers from high-school students. For the MNIST dataset, SD-1 was split, where characters written by the first 250 writers went into the MNIST training set and the characters written by the remaining 250 writers went into the MNSIT test set. Further, the MNIST training set was filled with enough examples from SD-3 to create a training set of 60.000 examples. The MNIST test set contains 10.000 examples, where 5.000 are from SD-1 and the other 5.000 from SD-3 [LBBH98]. All black and white images are size normalized and centered in a fixed size image at the center of the image with 28 x 28 pixels each[Den12].

[2] http://yann.lecun.com/exdb/mnist/

Figure 3.6: MNIST examples [LBBH98]

3.5 Environments

In this Section details about the environmental setup that form the foundation for the experiments are provided. As the aim of this master thesis is to examine the reproducibility of Deep Learning results considering different environments, it is essential that for every operating system the same execution platform as well as the same frameworks are installed. Of course, the same versions are as well important. Every experiment is done using the same Hardware, with a 2,8 GHz Intel Core i7 processor and 16 GB 2133 MHz LPDDR3 RAM. As base operating system MacOS High Sierra is used. On top of this the other operating systems are set up in Virtual Machines using Oracle's VirtualBox[3].

3.5.1 Execution Platform

In this Section the execution platforms that are needed by the Deep Learning frameworks are discussed. Two different execution platforms are used within this master thesis, respectively Python and Java. For every execution platform, two different versions are installed. As already mentioned, three different Deep Learning frameworks are tested where two are using Python and the third one Java as execution platform.

[3]https://www.virtualbox.org

Packages	Version
numpy	1.14.1
pandas	0.22.0
scikit-learn	0.19.1
scipy	1.0.0

Table 3.1: Python 2.7.14 and Python 3.5.4 packages

Python 2.7.14	Python 3.5.4	Description
print	print()	print function syntax changed
xrange()	range()	range function changed

Table 3.2: Comparison of Python 2.7.14 and Python 3.5.4

Python

Python is regarded as general-purpose, high-level programming language with the overall goal of code readability. A major advantage is that it allows programmers to express concepts in fewer lines of code than in other programming languages [vR18]. In this master thesis, the two frameworks that are built upon Python are implemented for Python 2.7.14 as well as for Python 3.5.4. Further, the Deep Learning model is trained with specific parameters mentioned in the Section 3.3.3. Important evaluation metrics, such as accuracy, recall, precision and F1-score, are calculated with the Python package 'scikit-learn'[4]. In order, to be able to install different python version on one operating system without getting a mess, the package management system Anaconda 3[5] is used. With the help of Anaconda, the environmental setup can be done in a clear and structural way, without having the fear of unexpected side-effects as all packages are encapsulated in a specific environment. Important packages that are used for all experiments using Python as execution platform are listed in Table 3.1.

Different Version of Python As already mentioned, the Python frameworks are tested with two different Python versions, respectively Python 2.7.14 and Python 3.5.4. As one goal of this master thesis is to analyze if different execution platform versions have influence on the Deep Learning model, it is clear that the same code needs to be adapted to fit the different Python versions. The adaptions that are needed to migrate code from Python 2.7.14 to Python 3.5.4 are listed in Table 3.2.

[4]http://scikit-learn.org/stable/
[5]https://www.anaconda.com/download/

Java

Java is one of the most used programming languages worldwide and is considered as a general-purpose, concurrent, class-based and object-oriented language. Additionally, its design principle is to be simple enough that many programmers can achieve fluency in the language [GJSB00]. The third Deep Learning framework that is used in this master thesis is using Java as its execution platform. The Deep Learning model is trained with the same parameters as the Python model. Fortunately, the framework possesses built in functions that calculate evaluation metrics, so that no further package is needed. Similar to the Python experiments, two different Java versions are tested, respectively Java 7 (JDK 1.7.0_80) and Java 8 (JDK 1.8.0_171). Regarding these two versions no adaptions are needed to migrate the code from one version to another. Besides the standard Java Virtual Machine from Oracle[6], another JVM is tested in this master thesis. Zulu 8[7] is an open source Java Virtual Machine that is built upon OpenJDK 1.8.0_172. Zulu leverages all the latest advances in OpenJDK and open source community support. An advantage of Zulu is that no coding changes are required as it is compliant with the Java SE standards.

3.5.2 Deep Learning Frameworks

In this Section the Deep Learning frameworks that are used in this master thesis are discussed and explained in detail. Three Deep Learning frameworks are used in this thesis, where two are using in Python and the other one Java.

TensorFlow

TensorFlow is currently one of the most popular Deep Learning frameworks and has been developed by Google's Machine Intelligence research organization. TensorFlow is an open source library for numerical computation using data flow graphs to represent computation shared state and operations that mutate that state. Therefore, it maps the nodes of a dataflow graph across many machines in a cluster and within a machine across multiple computational devices, including CPUs and GPUs. A strength of TensorFlow is its flexible architecture, that in case enables the computation on one or more CPUs or GPUs as well as on heterogeneous environments, such as desktops PCs, servers or mobile devices [ABC+16]. Besides the ability to develop and execute Deep Learning models, TensorFlow can be used in general as well for Machine Learning tasks. Another advantage of TensorFlow is the active community as well as a detailed and comprehensive documentation.

[6] https://docs.oracle.com/javase/6/docs/technotes/guides/vm/index.html?intcmp=3170
[7] https://www.azul.com/downloads/zulu/

Different Version of TensorFlow: For this master thesis three different Versions of TensorFlow are used. The three Versions are:

- TensorFlow 1.4.0,
- TensorFlow 1.5.0,
- TensorFlow 1.6.0.

As for the different Python versions, some minor changes need to be done to the model to be able to migrate it to the different versions of TensorFlow. The changes regarding this specific model are listed in Table 3.3.

TensorFlow 1.4.0	**TensorFlow 1.5.0**	**TensorFlow 1.6.0**	**Description**
tf.nn.softmax _cross_entropy _with_logits	tf.nn.softmax _cross_entropy _with_logits_v2	tf.nn.softmax _cross_entropy _with_logits_v2	function that defines the loss function was changed and renamed

Table 3.3: Different versions of TensorFlow

Theano

The second Python framework is very popular and used by a huge community. Theano aims to improve execution time and development time of Deep Learning algorithms but also of Machine Learning applications [BBL+11]. Theano is a library that enables the definition, optimization and evaluation of mathematical expressions that involve multi-dimensional arrays and has been developed by MILA lab from the University of Montreal. With Theano it is possible to attain speeds rivaling hand-crafted C implementations for problems that involve large amounts of data.

Different Version of Theano: Two different Theano versions are considered in this thesis, Theano 0.9.0 and Theano 1.0.0. To transform the model from one version to the other only a minor change need to be done. This change is listed in Table 3.4.

Theano 0.9.0	**Theano 1.0.0**	**Description**
T.nnet.conv.conv2d()	T.nnet.conv2d()	change of the input parameter names from *image_shape* to *input_shape*

Table 3.4: Different versions of Theano

Deeplearning4J

The Deep Learning framework that uses the Java programming language is Deeplearning4J. The reason for the choice of this framework is that it uses a totally different technology stack than the other two frameworks and that probably every operating system implements that technology stack in a different way. Deeplearning4J is the first distributed Deep Learning library written for Java and Scala. It is designed to be used in business environments on distributed GPUs and CPUs, as it is integrated with Hadoop and Spark. To bridge the gap between Python and JVM, DL4J can import neural net models from major frameworks via Keras, TensorFlow, Theano and Caffe [Tea18].

Different Versions of Deeplearning4J: Three different Deeplearning4J Versions are used in this master thesis. The three versions are:

- DL4J 0.7.1,
- DL4J 0.8.0,
- DL4J 0.9.1.

Some minor changes need to be done in order to migrate the model to the different versions of Deeplearning4J. The changes are listed in Table 3.5.

DL4J 0.7.1	**DL4J 0.8.0**	**DL4J 0.9.1**	**Description**
activation("relu")	activation (Activation.RELU)	activation (Activation.RELU)	Activation function changed from String to Enum
CSVRecordReader (numLinesToSkip, (String) delimiter)	CSVRecordReader (numLinesToSkip, (String) delimiter)	CSVRecordReader (numLinesToSkip, (Char) delimiter)	CSV delimiter changed from String to Char

Table 3.5: Different versions of Deeplearning4J

3.6 Operating Systems

In this Section the operating systems that are used for the experiments are discussed. The reason for testing multiple operating systems is that every operating system may implement things, such as the random number generation, in a quite different way which may lead to different Deep Learning results. In total seven operating systems are set up and tested in this master thesis. The following operating systems are considered:

i) Mac
   a) Mac OS High Sierra 10.13.5

ii) Linux
   a) Linux Fedora 25
   b) Linux Mint 18.3
   c) Linux Ubuntu 16.04

iii) WIndows
   a) Windows 7
   b) Windows 8
   c) Windows 10

Only official ISO files are used to install the operating systems. To be able to install these different operating systems on a single machine, Oracle VirtualBox[8] is used, except for Mac OS High Sierra which serves as the base operating system. In order to be able to conduct the experiments the following steps are considered for every operating system:

[8] https://www.virtualbox.org

For the Python frameworks:

- download and install the package manager system Anaconda 3,
- create different Anaconda environments,
- install all needed frameworks and required packages in the Anaconda environments,
- download and install JetBrains PyCharm[9],
- import the code as well as the dataset to PyCharm.

For the Java framework:

- download and install Oracles Java 7[10] and Java8[11],
- download and install Zulu JVM,
- download and install JetBrains IntelliJ[12],
- import the code as well as the dataset to IntelliJ.

After every operating system is set up, the experiments are conducted, and the results are written to a JSON file, where besides model specific metrics also global parameters, such as the operating system and the framework, are recorded. The detailed structure of this file is discussed in section 3.7.1. Afterwards, the results are statistically analyzed. The analytical methods used in this thesis are discussed in the next sub-section.

[9] https://www.jetbrains.com/pycharm/
[10] http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html
[11] https://www.java.com/de/download/faq/java8.xml
[12] https://www.jetbrains.com/idea/

3.7 Analysis Methods

In this Section the metrics that every model execution delivers as well as the statistical analysis method that is used to compare the different model results are described.

3.7.1 General

In order to be able to compare model results the following metrics are captured for every model run:

- *Accuracy*, the number of correct predictions from the model divided by the total number of predictions.
- *Precision*, the ratio of correctly predicted positive observations to the total number of positive predicted observations.
- *Recall*, the ratio of correctly predicted positives to all observations in the actual class.
- *F1-Score*, the weighted average of precision and recall.

As there were many experiments executed during this master thesis, a log file is created for every model run. The log file has the same structure regardless of the operating system, frameworks or any other parameter that is changed during the experiments. In Listing 3.1 the log file for TensorFlow 1.4.0 on Max OS High Sierra and Python version 2.7.14 is shown.

```
{
  "Normal Model Run": {
  "Recall": 0.9714587183336663,
  "F1-Score": 0.9712320621858522,
  "Precision": 0.9716708526959671,
  "Accuracy": 0.9717
},
  "System Configuration": {
  "Framework": "TensorFlow 1.4.0",
  "Seed": 10,
  "OS": "Mac OS X",
  "PythonVersion": "2.7.14"
}
```

Listing 3.1: Example log file

3.7.2 Two-Sample Kolmogorov-Smirnov Test

The main part of this master thesis deals with the question if the different results that every experiment delivers are from the same distribution. In order to show if the differences between distributions of the experiments are statistically significant, the Two-Sample Kolmogorov-Smirnov Test is used. In this Section this statistical test is described in detail.

The Two-Sample Kolmogorov-Smirnov Test is mainly used to test if two data samples come from the same distribution [Jr.51]. This test is a variation of the One-Sample Kolmogorov-Smirnov Test and belongs to the supremum class of empirical distribution function (EDF) statistics. EDF statistics in general are based on measuring the discrepancy between the empirical and hypothesized distributions. The supremum class of EDF statistics is based on the largest vertical difference between hypothesized and empirical distribution [RW11]. The Kolmogorov-Smirnov statistic is defined as,

$$D = \sup_x |F_{\mathrm{n}}(x) - F(x)| \tag{3.6}$$

The test statistic is computed as the maximum of D^+ and D^-. D^+ is the largest vertical distance between the EDF and the distribution function in the case the EDF is greater than the distribution function. D^- is the largest vertical distance when the EDF is less than the distribution function [Ins12].

In the case of the Two-Sample Kolmogorov-Smirnov Test, two empirical distribution functions are compared instead of comparing an empirical distribution function to a hypothesized distribution function [RW11]. The equation for the computation of the test statistic is described in [You77] and shown in equation 3.7.

$$D = \sup_x |F_{\mathrm{n1}}(x) - F_{\mathrm{n2}}(x)| \tag{3.7}$$

$F_{n1}(x)$ and $F_{n2}(x)$ are the empirical distribution functions of the two samples and these two empirical distribution functions are computed at each point in both samples. The empirical distribution function is defined as a set of independent ordered observation $x_1, \ldots, x_n$ with a common distribution function and where $F_{n*}(x)$ is a step function that takes a step of height $\frac{1}{n}$ at each observation [Ins12].

The hypotheses underlying the Two-Sample Kolmogorov-Smirnov Test are:

- H_0: The two samples come from the same distribution.
- H_a: The two samples do not come from the same distribution.

If the test statistic D exceeds the 1-α quantile as given by the table of quantiles for this test, then the null hypothesis (H_0) is rejected at the level of significance α and the alternative hypothesis (H_a) is accepted [RW11]. For this master thesis the statistical analysis of the experiments was done in Python. The Two-Sample Kolmogorov-Smirnov Test statistics were computed with the Python package Scipy[13].

3.8 Summary

In this Chapter the experimental design were discussed. Further, the used concepts were part of this Chapter, where a short introduction to Deep Learning was given, especially Convolutional Neural Networks that underly the model that is used in this master thesis. Additionally, the MNIST dataset was presented. An important part of this Chapter was the description of the Deep Learning model, considering its architecture and computations within each layer. Other parts of this Chapter regarded the technical setup, considering used execution platforms, Deep Learning frameworks and operating systems. The Chapter were finalized by explaining the methods that are used to analyze the results that are received from every model run.

[13] https://www.scipy.org

CHAPTER 4

Results

In this Chapter the results from the experiments are presented and organized in Sections where each Section corresponds to the research question that the respective experiments tries to answer. Section 4.1 deals with the question if is even possible to build the same Deep Learning model in the three frameworks that are used for this master thesis. The experiments that are described in Sections 4.2-4.4 follow a top-down approach in accordance to the execution stack diagram, shown in Figure 1.1. Section 4.2 tries to answer the question if the framework version has an effect on the model results as well as if the framework per se has an effect on the model results. Section 4.3 deals with the question if the execution platform has an influence on the results. Section 4.4 take the operating system version and the operating system in general into account. The experiments described in Sections 4.2-4.4 are conducted as preparation for the experiments conducted in Section 4.5, where same results are grouped into equivalence classes and then this equivalence classes are statistically analyzed with the help of the Two-Sample Kolmogorov-Smirnov Test.

4.1 Deep Learning Model Implementation

In order to be able to analyze results of different Deep Learning models, it must in the first place even be possible to build the same Deep Learning model in different Deep Learning frameworks. This Section describes the implementation of the Deep Learning model, specified in Section 3.3.2, for the three Deep Learning frameworks that are described in Section 3.5.2. For the implementation the operating system is negligibly as the frameworks that are used in this master thesis are selected in accordance to the used operating systems as well as the execution platforms. In general, the implementation of the same Deep Learning model in the three frameworks is possible without problems, but some parameters, such as the weight initialization, need more detailed analysis.

Precisely, for Theano and Tensorflow the distribution of the random initialization needed to be specified for the weights (Listing 4.1 and Listing 4.2), whereas DL4J uses standardly normal distributed random initialization.

```
#Weight initialization according to Xavier Glorot and Yoshua Bengio
fan_in = np.prod(filter_shape[1:])
fan_out = (filter_shape[0] * np.prod(filter_shape[2:]) //
        np.prod(poolsize))
W_bound = np.sqrt(2. / (fan_in + fan_out))
self.W = theano.shared(
  np.asarray(
    rng.normal(size=filter_shape, scale=W_bound),
    dtype=theano.config.floatX
  ),
borrow=True
)
```

Listing 4.1: Implementation of the Xavier Weight initialization in Theano 0.9.0

```
#Weight initialization according to Xavier Glorot and Yoshua Bengio
initializer = tf.contrib.layers.xavier_initializer_conv2d(uniform=False,
  seed=seed)
conv1_W = tf.get_variable("W1", shape=(5, 5, 1, 4), initializer=
  initializer)
```

Listing 4.2: Implementation of the Xavier Weight initialization in Tensorflow 1.4.0

In general, the implementation of the model in the two Python frameworks need more research and lead to more lines of code than the implementation of the model in the Java framework. The implementation of the first convolutional layer in DL4J is shown in Listing 4.3. In contrast to the Python frameworks, the Java frameworks defines the Weights for the whole neural net and not additionally for every layer.

```
  // Layer 1: Convolutional Layer: Input = 28x28x1. Output = 24x24x4
.layer(0, new ConvolutionLayer.Builder(5, 5)
    .nIn(1)
    .stride(1, 1)
    .padding(new int[]{0,0})
    .nOut(4)
    //RELU Activation function
    .activation("relu")
    .build())
```

Listing 4.3: Implementation of the first convolutional layer in DL4J 0.7.1

In Listing 4.4 the implementation of the first convolutional layer in Tensorflow 1.4.0 is shown. In Listing 4.5 the same implementation is shown for Theano 0.9.0. Note, that the implementation in Theano needs more lines of code compared to the other two frameworks.

```
#Layer 1: Convolutional Layer: Input = 28x28x1. Output = 24x24x4
conv1_W = tf.get_variable("W1", shape=(5, 5, 1, 4), initializer=
  initializer)
```

```
conv1_b = tf.Variable(tf.zeros(4))
conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') +
   conv1_b
#RELU Activation function
conv1 = tf.nn.relu(conv1)
```

Listing 4.4: Implementation of the first convolutional layer in Tensorflow 1.4.0

```
#The bias is a 1D tensor -- one bias per output feature map
b_values = np.zeros((filter_shape[0],), dtype=theano.config.floatX)
self.b = theano.shared(value=b_values, borrow=True)

#Definition of the Convolutional Layer
conv_out = conv2d(
  input=input,
  filters=self.W,
  filter_shape=filter_shape,
  image_shape=image_shape,
  subsample=(1, 1),
  border_mode='valid'
)
#RELU Activation function
conv_out = T.nnet.relu(conv_out + self.b.dimshuffle('x', 0, 'x', 'x'))
```

Listing 4.5: Implementation of the first convolutional layer in Theano 0.9.0

The implementation of the sub-sampling layer looks similar in all three frameworks and is shown in Listings 4.6 - 4.8. As can be seen, the same parameters are needed to define this layer in all three frameworks.

```
 //Layer 2: Pooling Layer: Input = 24x24x4. Output = 12x12x4
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
.kernelSize(2,2)
.stride(2,2)
.padding(new int[]{0,0})
.build())
```

Listing 4.6: Implementation of the sub-sampling layer in DL4J 0.7.1

```
#Layer 2: Pooling Layer: Input = 24x24x4. Output = 12x12x4
conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
  padding='VALID')
```

Listing 4.7: Implementation of the first convolutional layer in Tensorflow 1.4.0

```
#Definition of Pooling Layer
pooled_out = pool.pool_2d(
  input=conv_out,
  ws=poolsize,
  ignore_border=True,
  stride=(2,2),
  pad=(0,0)
)
```

Listing 4.8: Implementation of the first convolutional layer in Theano 0.7.1

The implementation of the model in all three frameworks as well as the code that is used to train and evaluate the model is shown in Appendix A.

4.1.1 Conclusion

The implementation of the neural network in the used frameworks is possible. In order to implement the neural network, the used methods need to be analyzed in detail to be able to configure the model appropriately. In our case, the Xavier weight initialization needed detailed analysis as the weight initialization method of Deeplearning4J standardly uses normal distributed weights. On the other hand, the method used in Tensorflow standardly uses uniform distributed weights, which can be changed to normal distribution by defining the parameter appropriately. For Theano the Xavier weight initialization has to be done by calculating them in accordance to equation 3.3. The implementation in DL4J was the least complicated as all needed methods are predefined. The implementation in DL4J resulted in 55 lines of code. The implementation in Tensorflow resulted in 54 lines of code. The implementation in Theano resulted in 187 lines of code. Theano needed more lines of code as every layer type that was used (convolutional layer, sub-sampling layer, fully connected layer and output layer) were implemented as separate class to facilitate the concrete model definition and to produce clean code. The biggest difference to the other frameworks is the definition of the weights. As already said, to initialize the weights no predefined method can be used. So, the calculations needed to be implemented on our own. All the other elements are pretty much the same, just different names are used to call the methods.

4.2 Framework

The first experiments that are conducted focused on one specific framework and its different versions that are considered in this master thesis. The different versions of the frameworks are tested by running the Deep Learning model with the exact same hyperparameters and dataset. Important to note is that the same random seed is used for every framework version. By testing the framework versions (the 'Implementation' dimension of the PRIMAD model is changed in this case), the highest stage of the execution stack is tested, as shown in Figure 4.1. The configuration for TensorFlow is shown in Table 4.1. All the other execution stack parameters are fixed, only the framework version is changed!

By regarding the different TensorFlow framework versions results in Table 4.2, it shows that the results of the version 1.6.0 differ slightly in comparison to version 1.4.0 and version 1.5.0. Table 4.3. shows the configuration for the next framework, Theano.

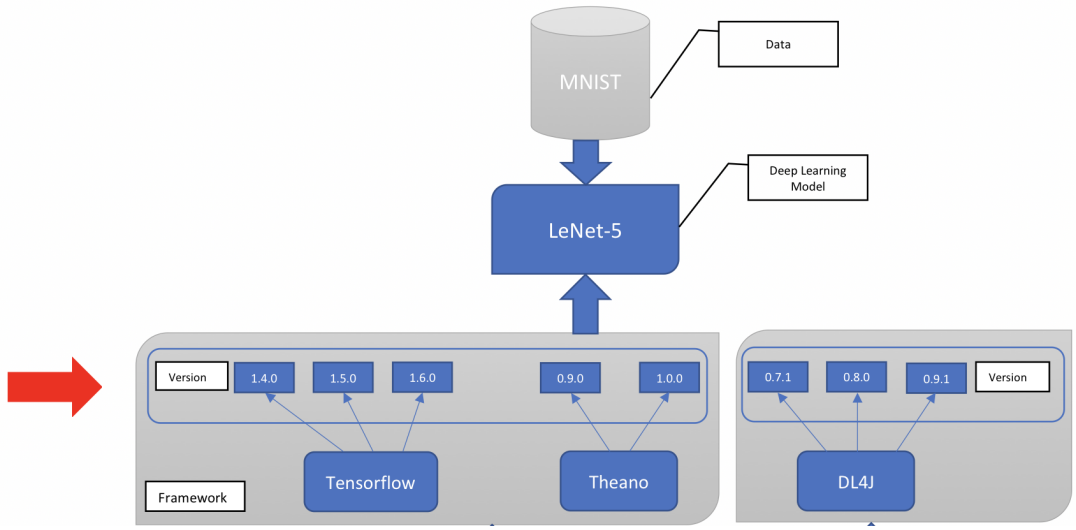

Figure 4.1: Execution stack stage that is tested, the framework versions

PRIMAD	
Research goal	classifying MNIST data
Method	LeNet-based Deep Learning model
Implementation	***script in TensorFlow 1.4.0, 1.5.0 and 1.6.0***
Platform	Python 2.7.14, Mac OS High Sierra
Input data	MNIST
Parameter	Seed = 10, Epochs = 10, Batch size = 100, Learning rate = 0.01
Actor	Bojan Čavić

Table 4.1: Example configuration for framework version testing, in this case for TensorFlow on Mac OS High Sierra

Version	**1.4.0**	**1.5.0**	**1.6.0**
Accuracy	0,97170	0,97170	0,97210
Precision	0,97167	0,97167	0,97206
Recall	0,97146	0,97146	0,97188
F1-Score	0,97123	0,97123	0,97166

Table 4.2: Results for different TensorFlow versions on Mac OS High Sierra

Version	**TF 1.4.0**	**TF 1.5.0**	**TF 1.6.0**	**TH 0.9.0**	**TH 1.0.0**
Accuracy	0,97170	0,97170	0,97210	0,97640	0,97700
Precision	0,97167	0,97167	0,97206	0,97618	0,97681
Recall	0,97146	0,97146	0,97188	0,97634	0,97696
F1-Score	0,97123	0,97123	0,97166	0,97617	0,97681

Table 4.4: Results for different Tensorflow and Theano versions on Mac OS High Sierra

PRIMAD	
Research goal	classifying MNIST data
Method	LeNet-based Deep Learning model
Implementation	***script in Theano 0.9.0 and 1.0.0***
Platform	Python 2.7.14, Mac OS High Sierra
Input data	MNIST
Parameter	Seed = 10, Epochs = 10, Batch size = 100, Learning rate = 0.01
Actor	Bojan Čavić

Table 4.3: Example configuration for framework version testing, in this case for Theano on Mac OS High Sierra

The two Theano versions show also slightly different results, as can be seen in Table 4.4, even if everything else is configured in the same way. This count only for Mac OS High Sierra. The results on other operating systems yield the same results for different Theano versions.

The last framework that that is tested, is DeepLearning4J. The configuration of the parameters is shown in Table 4.5.

PRIMAD	
Research goal	classifying MNIST data
Method	LeNet-based Deep Learning model
Implementation	***java program in DL4J 0.7.1, 0.8.0 and 0.9.1***
Platform	JDK 8, Mac OS High Sierra
Input data	MNIST
Parameter	Seed = 10, Epochs = 10, Batch size = 100, Learning rate = 0.01
Actor	Bojan Čavić

Table 4.5: Example configuration for framework version testing, in this case for DeepLearning4J

This framework is quite interesting as it uses a completely different execution platform which may be implemented in every operating system in another way. Even if these experiments is run on the same operating system, every framework version shows different results, as shown in Table 4.6.

By regarding all the framework versions, it shows that only TensorFlow version 1.4.0 and 1.5.0 deliver the same results, the other versions, independent of the framework, compute different results.

Version	**TF 1.4.0**	**TF 1.5.0**	**TF 1.6.0**	**TH 0.9.0**	**TH 1.0.0**
Accuracy	0,97170	0,97170	0,97210	0,97640	0,97700
Precision	0,97167	0,97167	0,97206	0,97618	0,97681
Recall	0,97146	0,97146	0,97188	0,97634	0,97696
F1-Score	0,97123	0,97123	0,97166	0,97617	0,97681
Version	**DL4J 0.7.1**	**DL4J 0.8.0**	**DL4J 0.9.1**		
Accuracy	0,9711	0,9705	0,9695		
Precision	0,9710	0,9706	0,9696		
Recall	0,9709	0,9703	0,9693		
F1-Score	0,9710	0,9705	0,9692		

Table 4.6: Results for different Tensorflow, Theano and DL4J versions on Mac OS High Sierra

4.3 Execution Platform

Experiments at a lower stage of the execution stack diagram concentrate on the execution platform. Hereby, two different versions of Python and Java are considered. The selection of the execution platform versions considered the compatibility of the different frameworks, as not all frameworks support the same execution platform. Considering this obstacle, the Python-based frameworks used Python version 2.7.14 and 3.5.4. On the other hand, the Java framework used JDK version 1.7.0_80 and 1.8.0_171. Additionally, besides the standard Java Virtual Machine by Oracle another JVM is tested, as already mentioned in Section 3.5.1. The execution stack configuration for this experimental scheme is shown in Table 4.7 and Figure 4.2 shows the stage of the execution stack that is tested. The execution platform corresponds to the platform in the PRIMAD model.

	PRIMAD
Research goal	classifying MNIST data
Method	LeNet-based Deep Learning model
Implementation	script in TensorFlow 1.4.0
Platform	***Python 2.7.14 and Python 3.5.4, Mac OS High Sierra***
Input data	MNIST
Parameter	Seed = 10, Epochs = 10, Batch size = 100, Learning rate = 0.01
Actor	Bojan Čavić

Table 4.7: Example configuration for testing different execution platform versions, here Python

The results for this configuration are shown in Table 4.8. The experiments report that the execution platform does not have any influence on the model results on Mac OS High Sierra for Tensorflow and Theano. The execution stack configuration for the analysis of the different Java Virtual Machines is shown in Table 4.9.

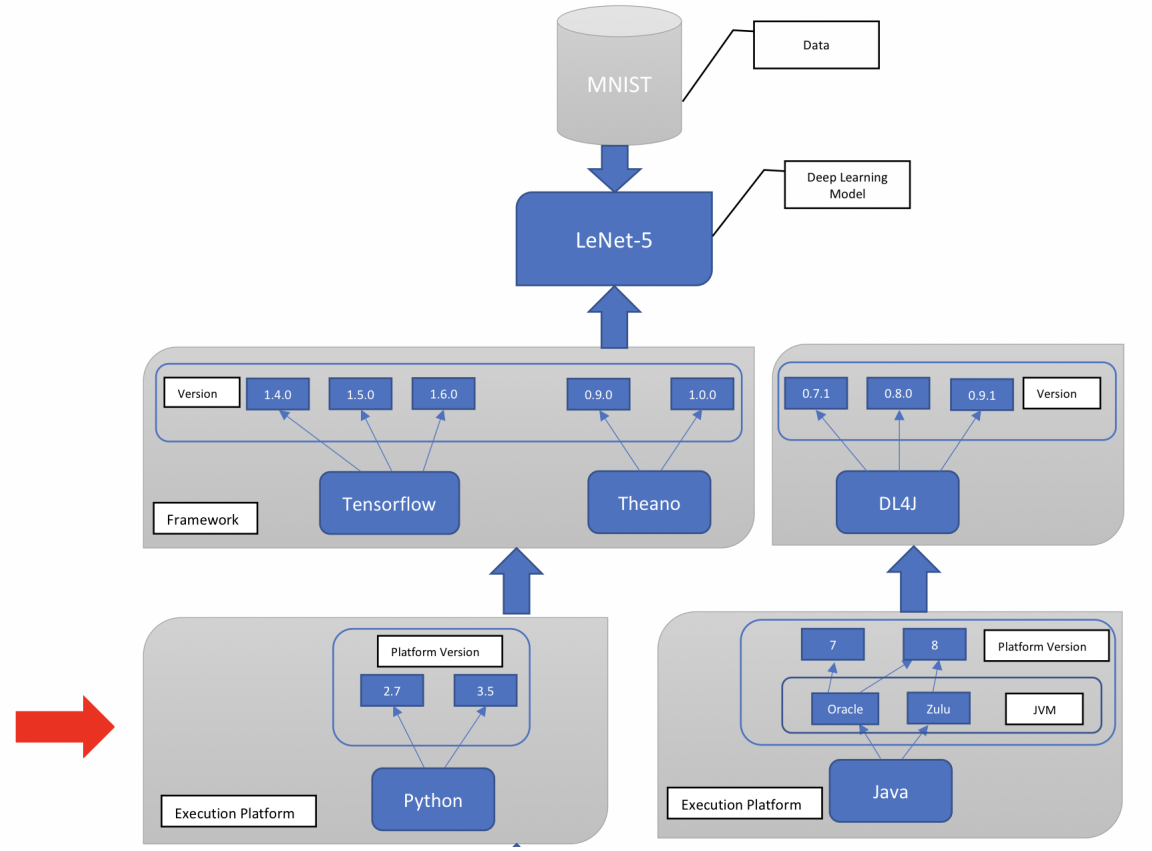

Figure 4.2: Execution stack stage that is tested, the execution platform

Execution Platform	Python 2.7.14	Python 3.5.4
Accuracy	0,97170	0,97170
Precision	0,97167	0,97167
Recall	0,97146	0,97146
F1-Score	0,97123	0,97123

Table 4.8: Tensorflow 1.4.0 results for different execution platform versions on Mac OS High Sierra

The investigation for the Java-based framework shows that neither the JDK version nor the JVM have an influence on the Deep Learning model results, see Table 4.10.

A reason that the different JVM's deliver the same results is probably that Zulu is built on OpenJDK where one development contributor is Oracle. All experiments considering the execution platform lead to the fact that the execution platform does not have any influence on the results. No matter which Java VM was being used, the results were always identical to the differences observed for the different DL4J versions shown in Table 4.6. We can thus consider - at least for Mac OS High Sierra - the different VM

PRIMAD	
Research goal	classifying MNIST data
Method	LeNet-based Deep Learning model
Implementation	java program in DL4J 0.8.0
Platform	***JDK 8 (Oracle JVM) and JDK 8 (Zulu JVM), Mac OS High Sierra***
Input data	MNIST
Parameter	Seed = 10, Epochs = 10, Batch size = 100, Learning rate = 0.01
Actor	Bojan Čavić

Table 4.9: Example configuration for testing different JVM's

Execution Platform	**Java 8 (Oracle)**	**Java 8 (Oracle)**	**Java 8 (Zulu)**
Accuracy	0,9705	0,9705	0,9705
Precision	0,9706	0,9706	0,9706
Recall	0,9703	0,9703	0,9703
F1-Score	0,9705	0,9705	0,9705

Table 4.10: DL4J 0.8.0 results for different jvm's on Mac OS High Sierra

types to form one equivalence class.

PRIMAD	
Research goal	classifying MNIST data
Method	LeNet-based Deep Learning model
Implementation	script in TensorFlow 1.4.0
Platform	Python 3.5.4
	Windows 8 and Windows 10
Input data	MNIST
Parameter	Seed = 10, Epochs = 10, Batch size = 100, Learning rate = 0.01
Actor	Bojan Čavić

Table 4.11: Example configuration for testing different operating system version

4.4 Operating System

Another parameter that is tested is the operating system. The assumption for the experiments is that the operating system does not have influence on the results. An example configuration for experiments considering different operating system is shown in Table 4.11 (the 'Platform' dimension of the PRIMAD model is changed in this case). Figure 4.3 shows the corresponding stage in the execution stack diagram. As already mentioned, seven operating systems are analyzed (Mac OS High Sierra, Linux Fedora 25, Linux Mint 18.3, Linux Ubuntu 16.04, Windows 7, Windows 8 and Windows 10). As every operating system implements some functionalities in another way, different results are most likely to happen. The execution of the experiments leads to the results described in Table 4.14.

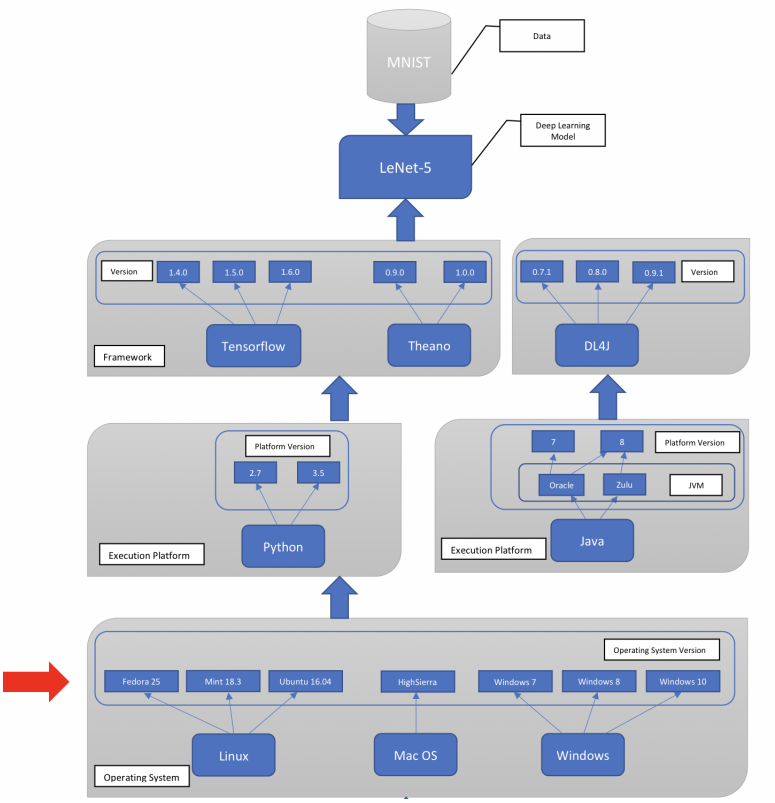

Figure 4.3: Execution stack stage that is tested, the operating system versions

Operating System	Execution Platform	Framework	Accuracy	Precision	Recall	F1-Score
Mac OS High Sierra	Python 2.7.14	TF 1.4.0	0,97170	0,97167	0,97146	0,97123
		TF 1.5.0	0,97170	0,97167	0,97146	0,97123
		TF 1.6.0	0,97210	0,97206	0,97188	0,97166
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97700	0,97681	0,97696	0,97681
	Python 3.5.4	TF 1.4.0	0,97170	0,97167	0,97146	0,97123
		TF 1.5.0	0,97170	0,97167	0,97146	0,97123
		TF 1.6.0	0,97210	0,97206	0,97188	0,97166
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97700	0,97681	0,97696	0,97681
	Java 7 (Oracle)	DL4J 0.7.1	0,9711	0,9710	0,9709	0,9710
		DL4J 0.8.0	0,9705	0,9706	0,9703	0,9705
		DL4J 0.9.1	0,9695	0,9696	0,9693	0,9692
	Java 8 (Oracle)	DL4J 0.7.1	0,9711	0,9710	0,9709	0,9710
		DL4J 0.8.0	0,9705	0,9706	0,9703	0,9705
		DL4J 0.9.1	0,9695	0,9696	0,9693	0,9692
	Java 8 (Zulu)	DL4J 0.7.1	0,9711	0,9710	0,9709	0,9710
		DL4J 0.8.0	0,9705	0,9706	0,9703	0,9705
		DL4J 0.9.1	0,9695	0,9696	0,9693	0,9692
Linux Fedora 25	Python 2.7.14	TF 1.4.0	0,97140	0,97142	0,97116	0,97093
		TF 1.5.0	0,97140	0,97142	0,97116	0,97093
		TF 1.6.0	0,97170	0,97166	0,97145	0,97122
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Python 3.5.4	TF 1.4.0	0,97140	0,97142	0,97116	0,97093
		TF 1.5.0	0,97140	0,97142	0,97116	0,97093
		TF 1.6.0	0,97170	0,97166	0,97145	0,97122
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Java 7 (Oracle)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689
	Java 8 (Oracle)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689
	Java 8 (Zulu)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689
Linux Mint 18.3	Python 2.7.14	TF 1.4.0	0,97140	0,97142	0,97116	0,97093
		TF 1.5.0	0,97140	0,97142	0,97116	0,97093
		TF 1.6.0	0,97170	0,97166	0,97145	0,97122
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Python 3.5.4	TF 1.4.0	0,97140	0,97142	0,97116	0,97093
		TF 1.5.0	0,97140	0,97142	0,97116	0,97093
		TF 1.6.0	0,97170	0,97166	0,97145	0,97122
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Java 7 (Oracle)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689
	Java 8 (Oracle)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689
	Java 8 (Zulu)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689

Linux Ubuntu 16.04	Python 2.7.14	TF 1.4.0	0,97140	0,97142	0,97116	0,97093
		TF 1.5.0	0,97140	0,97142	0,97116	0,97093
		TF 1.6.0	0,97170	0,97166	0,97145	0,97122
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Python 3.5.4	TF 1.4.0	0,97140	0,97142	0,97116	0,97093
		TF 1.5.0	0,97140	0,97142	0,97116	0,97093
		TF 1.6.0	0,97170	0,97166	0,97145	0,97122
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Java 7 (Oracle)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689
	Java 8 (Oracle)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689
	Java 8 (Zulu)	DL4J 0.7.1	0,9701	0,9701	0,9699	0,9700
		DL4J 0.8.0	0,9706	0,9707	0,9704	0,9706
		DL4J 0.9.1	0,9692	0,9694	0,9690	0,9689
Windows 7	Python 2.7.14	TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Python 3.5.4	TF 1.4.0	0,97210	0,97204	0,97187	0,97163
		TF 1.5.0	0,97210	0,97204	0,97187	0,97163
		TF 1.6.0	0,97200	0,97198	0,97176	0,97153
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Java 7 (Oracle)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697
	Java 8 (Oracle)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697
	Java 8 (Zulu)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697
Windows 8	Python 2.7.14	TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Python 3.5.4	TF 1.4.0	0,97210	0,97204	0,97187	0,97163
		TF 1.5.0	0,97210	0,97204	0,97187	0,97163
		TF 1.6.0	0,97200	0,97198	0,97176	0,97153
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Java 7 (Oracle)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697
	Java 8 (Oracle)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697
	Java 8 (Zulu)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697
Windows 10	Python 2.7.14	TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Python 3.5.4	TF 1.4.0	0,97210	0,97204	0,97187	0,97163
		TF 1.5.0	0,97210	0,97204	0,97187	0,97163
		TF 1.6.0	0,97200	0,97198	0,97176	0,97153
		TH 0.9.0	0,97640	0,97618	0,97634	0,97617
		TH 1.0.0	0,97640	0,97618	0,97634	0,97617
	Java 7 (Oracle)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697
	Java 8 (Oracle)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697
	Java 8 (Zulu)	DL4J 0.7.1	0,9705	0,9704	0,9703	0,9704
		DL4J 0.8.0	0,9694	0,9695	0,9692	0,9694
		DL4J 0.9.1	0,9700	0,9701	0,9698	0,9697

Table 4.14: Results for all operating systems including different versions of the execution platform and Deep Learning framework

Experiments that were conducted in this Section show that the operating system version does not have influence on the results, but the operating system itself does have an influence as can be observed from Table 4.14.

4.5 Equivalence Classes

Considering the results from Section 4.2 - 4.4, it shows that different Deep Learning frameworks never produce identical results, but different framework versions sometimes do. Tensorflow 1.4.0 and 1.5.0 produce the same results across all operating systems. Tensorflow 1.6.0 produces different results in comparison to Tensorflow 1.4.0 and 1.5.0. The two different Theano versions that are considered in this master thesis produce always identical results, except on Mac OS High Sierra. Different Deeplearnig4J versions always produce different results. Further, different operating system versions always produce the same results, so specific versions have no impact, but different operating system types produce never identical results. This observations lead to the possibility to group identical results in so-called equivalence classes. Further, for every equivalence class a representative configuration is chosen for further investigation. In the Tables 4.15 - 4.17 all equivalence classes are displayed. The colored execution stack configuration is the representative that is used for further investigation. In Section 4.4 is explained that different operating system versions do not have influence on the Deep Learning results, this enable us to group the operating system versions to the overall operating system.

Equivalence class	**Execution Stack**
MacOS-M1	TensorFlow 1.4.0, Python 2.7
	TensorFlow 1.5.0, Python 2.7
	TensorFlow 1.4.0, Python 3.5
	TensorFlow 1.5.0, Python 3.5
MacOS-M2	TensorFlow 1.6.0, Python 2.7
	TensorFlow 1.6.0, Python 3.5
MacOS-M3	Theano 0.9.0, Python 2.7
	Theano 0.9.0, Python 3.5
MacOS-M4	Theano 1.0.0, Python 2.7
	Theano 1.0.0, Python 3.5
MacOS-M5	DL4J 0.7.1, Java 7
	DL4J 0.7.1, Java 8
MacOS-M6	DL4J 0.8.0, Java 7
	DL4J 0.8.0, Java 8
MacOS-M7	DL4J 0.9.1, Java 7
	DL4J 0.9.1, Java 8

Table 4.15: Equivalene classes: Mac OS High Sierra

Equivalence class	Execution Stack
Linux-L1	TensorFlow 1.4.0, Python 2.7
	TensorFlow 1.5.0, Python 2.7
	TensorFlow 1.4.0, Python 3.5
	TensorFlow 1.5.0, Python 3.5
Linux-L2	TensorFlow 1.6.0, Python 2.7
	TensorFlow 1.6.0, Python 3.5
Linux-L3	Theano 0.9.0, Python 2.7
	Theano 0.9.0, Python 3.5
	Theano 1.0.0, Python 2.7
	Theano 1.0.0, Python 3.5
Linux-L4	DL4J 0.7.1, Java 7
	DL4J 0.7.1, Java 8
Linux-L5	DL4J 0.8.0, Java 7
	DL4J 0.8.0, Java 8
Linux-L6	DL4J 0.9.1, Java 7
	DL4J 0.9.1, Java 8

Table 4.16: Equivalene classes: Linux Fedora 25 & Linux Mint 18.3 & Linux Ubuntu 16.04

Equivalence class	Execution Stack
Windows-W1	TensorFlow 1.4.0, Python 3.5
	TensorFlow 1.5.0, Python 3.5
Windows-W2	TensorFlow 1.6.0, Python 3.5
Windows-W3	Theano 0.9.0, Python 2.7
	Theano 0.9.0, Python 3.5
	Theano 1.0.0, Python 2.7
	Theano 1.0.0, Python 3.5
Windows-W4	DL4J 0.7.1, Java 7
	DL4J 0.7.1, Java 8
Windows-W5	DL4J 0.8.0, Java 7
	DL4J 0.8.0, Java 8
Windows-W6	DL4J 0.9.1, Java 7
	DL4J 0.9.1, Java 8

Table 4.17: Equivalene classes: Windows 7 & Windows 8 & Windows 10

4.6 Statistical analysis

In order to be able to statistically analyze the influence of the certain configurations, every equivalence class representative is run 20 times with a different random seed, but for every equivalence class the same 20 random seeds are used! In this way a distribution of 20 different results is produced which then can be further analyzed and tested. By specifying the experimental scheme in that way, where only the random seed is changed, it resulted in minimal variations of the other parameters where only the random seed could probably have an influence on the model output.

4.6.1 Evaluation Criterion

The statistical analysis is conducted with the Two-sample Kolmogorov-Smirnov Test. Hence, every equivalence class is tested in comparison to the others for the same operating system. Further, equivalence classes from different operating systems are tested against each other. For Mac OS High Sierra seven equivalence classes are tested, which resulted in 21 Two-sample Kolmogorov-Smirnov test executions. For Windows and Linux six equivalence classes are investigated, with 15 Two sample Kolmogorov-Smirnov test executions per operating system. Every model execution delivers four different values as result, accuracy, precision, recall and the F1-Score. For the Two-sample Kolmogorov-Smirnov Test, the 20 accuracy values of each equivalence class are used. The null hypothesis (H_0) underlying this test suggest that the distributions gathered from the experiments (C_1 and C_2) are from the same distribution.

- H_0: $C_1 = C_2$
- H_a: $C_1 \neq C_2$

In case to facilitate the conclusion the d-value is used. with a significance level alpha, of 0.05. D-values above the critical value (d_crit = 0,43) are regarded as significant, which results in rejecting the null Hypothesis (H_0) and accepting the alternative hypothesis. On the other hand, when the d-value is smaller than d_crit the null hypothesis is not rejected. Cruical for the acceptance or rejection of the hypothesis is the critical value (d_crit), which is calculated (for significance level aplha of 0.05) as follows:

$$d\_crit = 1,36 - \sqrt{\frac{1}{20} + \frac{1}{20}} = 0,43 \tag{4.1}$$

4.6.2 Mac OS High Sierra Analysis

In this Section the statistical analysis of the seven equivalence classes that are received as result from Mac OS High Sierra are described.

Equivalence class	M1	M2	M3	M4	M5	M6	M7
M1		0.10	0.25	0.30	0.20	0.25	0.30
M2	0.10		0.25	0.30	0.20	0.30	0.35
M3	0.25	0.25		0.15	0.20	0.15	0.25
M4	0.30	0.30	0.15		0.20	0.15	0.25
M5	0.20	0.20	0.20	0.20		0.15	0.35
M6	0.25	0.30	0.15	0.15	0.15		0.25
M7	0.30	0.35	0.25	0.25	0.35	0.25	

Table 4.18: d-values of the Two-Sample Kolmogorov-Smirnov Test executed for Mac OS High Sierra

Equivalence class	M1	M2	M3	M4	M5	M6	M7
M1		0.99999	0.49734	0.27527	0.77095	0.49734	0.27527
M2	0.99999		0.49734	0.27527	0.77095	0.27527	0.13495
M3	0.49734	0.49734		0.96548	0.77095	0.96548	0.49734
M4	0.27527	0.27527	0.96548		0.77095	0.96548	0.49734
M5	0.77095	0.77095	0.77095	0.77095		0.96548	0.13495
M6	0.49734	0.27527	0.96548	0.96548	0.96548		0.49734
M7	0.27527	0.13495	0.49734	0.49734	0.13495	0.49734	

Table 4.19: p-values of the Two-Sample Kolmogorov-Smirnov Test executed for Mac OS High Sierra

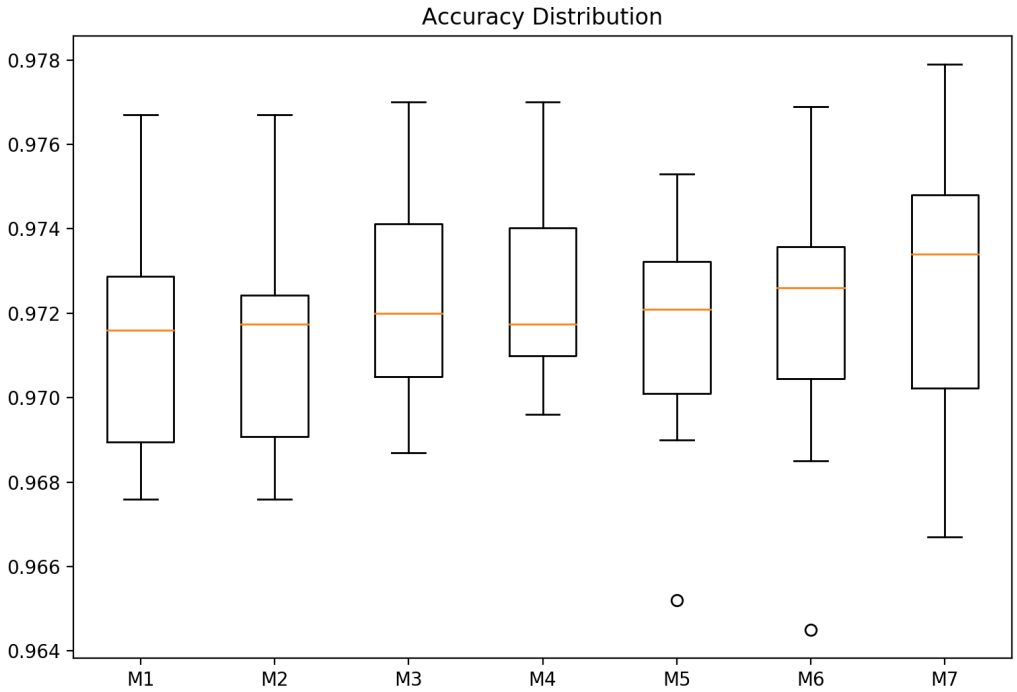

Figure 4.4: Boxplot for equivalence classes obtained for Mac OS Hiegh Sierra

All equivalence classes are statistically tested against each other, e.g. equivalence class 1 is tested with the Two-sample Kolmogorov-Smirnov test against equivalence class 2-6. Table 4.18 contains the d-values from all 2-Sample KS-Tests conducted for the seven equivalence classes on Mac OS High Sierra, whereas Table 4.19 contains the p-values. Figure 4.4 shows boxplots for every equivalence class. By regarding the result of the 2-Sample KS Test of equivalence class 1 and 2, it shows that the 20 accuracy results from these equivalence classes originate from the same distribution as the d-value is smaller than the critical value, d_crit ($0.10 < 0.43$).

With a p-value of 0.99999 and a d value of 0.10 the null hypothesis is retained with high confidence. These two equivalence classes are, in fact, two different versions of the same Deep Learning framework (Tensorflow 1.4.0 and Tensorflow 1.6.0). By investigating equivalence classes 3 and 4, which as well represent the two versions of Theano, the p-value is also quite high, 0.96548. The d-value amount 0.15. Finally, the conclusion is the same as for the first test, H_0 is retained.

The two Python frameworks with its versions deliver results that are not identically but originate from the same distribution. The last three equivalence classes represent the three versions of the Java framework. The Two-sample KS test for class 5 and 6 delivers a p-value of 0.96548 and a d-value of 0.15. Obviously, the results originate from the same distribution. Interestingly, the p-value resulting from testing equivalence class 5 and 7 is drastically smaller than the previous one, namely 0.13495. The d-value adds up to 0.35, which is also smaller than the critical value, d_crit. Even if the p-value is smaller than the others, the null hypothesis is retained as it is not smaller than the significance level alpha (0.05). But it shows that there is a discrepancy of results for these two versions of the framework. The d-value for the test of equivalence classes 6 and 7 is 0.25 which leads to a p-value of 0.49734.

From table 4.19 results from statistically testing the different frameworks can also be derived. All d-values as well as p-values considering these tests are higher than the significance level alpha and smaller than the critical value, d_crit, but mostly smaller than the tests of the different framework versions.

One of the smallest p-values is resulting from testing equivalence class 2 and 7. Equivalence class 2 is represented by Tensorflow 1.6.0, whereas equivalence class 7 is represented by DL4J 0.9.1. The obtained p-value is 0.13495 and the test statistic 0.35. This result shows the comparison of TensorFlow 1.6.0 and Deeplearning4J 0.9.1. Equivalence class 2 and 4 represents the comparison of TensorFlow 1.6.0 and Theano 1.0.0, resulting in a p-value of 0.27527. Further, this p-value yield the retainment of the null hypothesis. By comparing all seven equivalence classes that are obtained for Mac OS High Sierra, it shows that there are no significant differences.

4.6.3 Linux Analysis

In this Section the statistical analysis of the six equivalence classes that are received as result from the three Linux-based operating systems are described. Contrary to Mac OS High Sierra, the Linux based operating systems classifying six equivalence classes as both Theano versions deliver the same results. All p-values are shown in Table 4.21 and the corresponding d-values are shown in Table 4.20. Figure 4.5 shows the distribution of each equivalence class in the form of boxplots.

Equivalence class	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
L1		0.15	0.20	0.20	0.35	0.35
L2	0.15		0.25	0.20	0.35	0.40
L3	0.20	0.25		0.20	0.25	0.25
L4	0.20	0.20	0.20		0.40	**0.45**
L5	0.35	0.35	0.25	0.40		0.15
L6	0.35	0.40	0.25	**0.45**	0.15	

Table 4.20: d-values of the Two-Sample Kolmogorov-Smirnov Test executed for Linux-based systems

Equivalence class	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**
L1		0.96548	0.77095	0.77095	0.13495	0.13495
L2	0.96548		0.49734	0.77095	0.13495	0.05914
L3	0.77095	0.49734		0.77095	0.49734	0.49734
L4	0.77095	0.77095	0.77095		0.05914	**0.02321**
L5	0.13495	0.13495	0.49734	0.05914		0.96548
L6	0.13495	0.05914	0.49734	**0.02321**	0.96548	

Table 4.21: p-values of the Two-Sample Kolmogorov-Smirnov Test executed for Linux-based systems

The first two equivalence classes represent different TensorFlow versions. Class one represents the first two TensorFlow versions, 1.4.0 and 1.5.0, and equivalence class two TensorFlow 1.6.0. The Two-sample Kolmogorov-Smirnov test for these two equivalence classes yield a p-value of 0.96548 and a d-value of 0.15. As the d-value is not greater than d_crit (=0.43) the null hypothesis is retained.

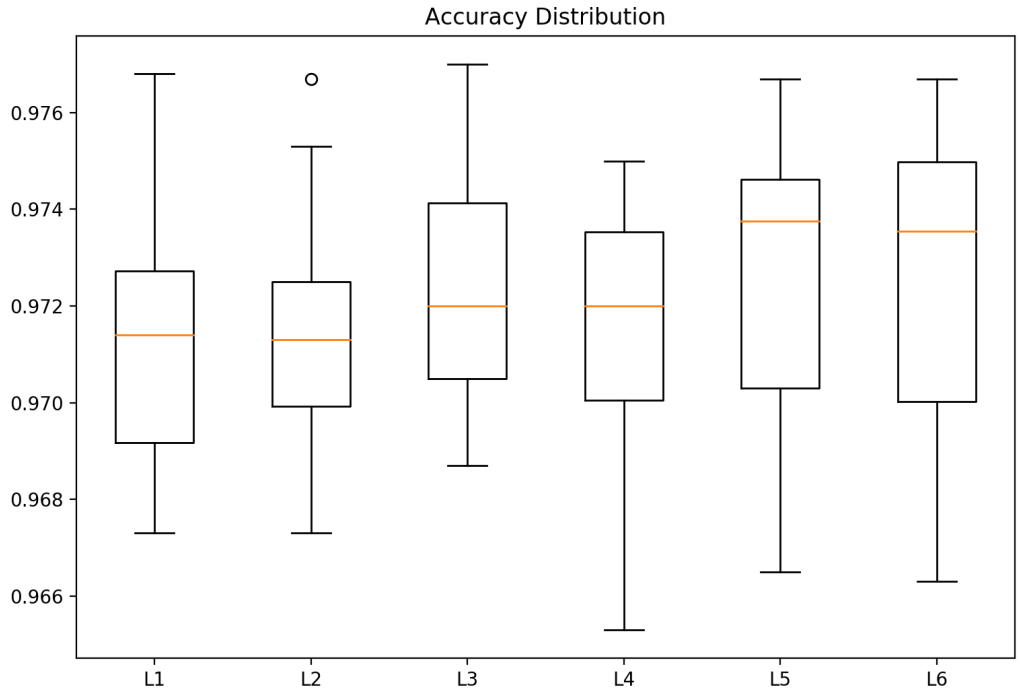

Figure 4.5: Boxplot for equivalence classes obtained for Linux-based systems

This concludes that the TensorFlow framework version within this configuration do not have a significant influence. As already mentioned, the two Theano versions behave identically on Linux Fedora 25, Linux Mint 18.03 and Linux Ubuntu 16.04. More interesting are the results of the last three equivalence classes which represent the three versions of the Java-based Deep Learning framework. The p-value for comparing equivalence class 4 and 5 is 0.05914 and the d-value is 0.40. This d-value is slightly smaller than the d_crit (= 0.43), which leads to retaining the null hypothesis but confidently concluding that the results are from the same distribution is risky. When testing equivalence classes 2 and 6 a p-value of 0.05914 and a d-value of 0.40 is obtained. Further, a p-value of 0.13495 is yielded for equivalence classes 2 and 5 as well as a d-value of 0.35.

Even more interesting results were achieved for equivalence class 4 and 6. The 2-Sample KS-Test yield a p-value of 0.02321 and a d-value of 0.45. By considering a significance level alpha of 0.05 and the d_crit of 0.43, the null hypothesis H_0 is discarded in this case and the alternative hypothesis H_a is accepted, concluding that the results from these equivalence classes do not originate from the same distribution! Considering the associated Figure 4.5, the inhomogeneous distribution is apparent. Even more interesting is the fact that, the statistical test returns a d-value of 0.15 for the equivalence classes 5 and 6, whereas when comparing these two equivalence classes with equivalence class 4 the d-value increases and in one case is even greater than d_crit.

4.6.4 Windows Analysis

In this Section the statistical analysis of the six equivalence classes that are received as result from the three Windows based operating systems are described. Here also six equivalence classes are analyzed as here as well both Theano versions deliver the same results. The p-values of all 2-Sample KS-Tests are summarized in Table 4.23 and the d-values in Table 4.22. Figure 4.6 shows the distribution of each equivalence class in the form of boxplots.

Equivalence class	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**
W1		0.20	0.25	0.20	0.35	0.35
W2	0.20		0.20	0.20	0.35	0.35
W3	0.25	0.20		0.25	0.25	0.25
W4	0.20	0.20	0.25		0.35	0.30
W5	0.35	0.35	0.25	0.35		0.15
W6	0.35	0.35	0.25	0.30	0.15	

Table 4.22: d-values of the Two-Sample Kolmogorov-Smirnov Test executed for Windows-based systems

Equivalence class	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**
W1		0.77095	0.49734	0.77095	0.13495	0.13495
W2	0.77095		0.77095	0.77095	0.13495	0.13495
W3	0.49734	0.77095		0.49734	0.49734	0.49734
W4	0.77095	0.77095	0.49734		0.13495	0.27527
W5	0.13495	0.13495	0.49734	0.13495		0.96548
W6	0.13495	0.13495	0.49734	0.27527	0.96548	

Table 4.23: p-values of the Two-Sample Kolmogorov-Smirnov Test executed for Windows-based systems

The analysis for the first two equivalence classes yield a p-value of 0.77095 and d-value of 0.20. Even if the p-value is not as high as for the tests on the other operating systems, there is no doubt that the results originate from the same distribution.

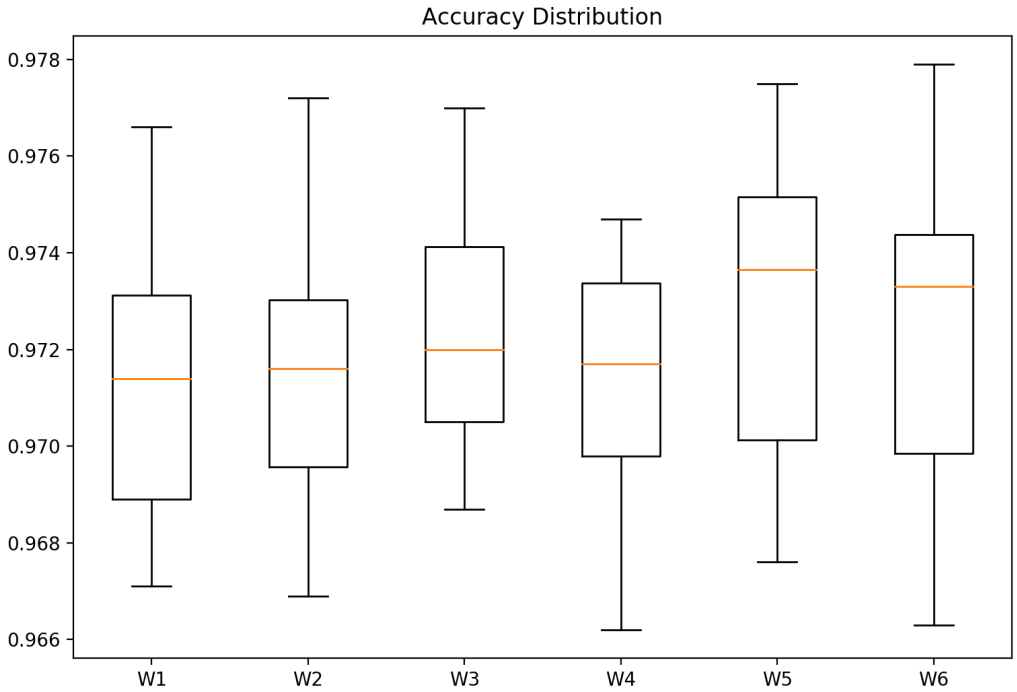

Figure 4.6: Boxplot for equivalence classes obtained for Windows-based systems

Similar to the analysis of the Linux-based operating systems, equivalence classes 5 and 6 (DL4J 0.8.0 and DL4J 0.9.1) show the highest p-values. The analysis of equivalence classes 4 and 5 deliver a p-value of 0.13495 and a d-value of 0.35. Further, a p-value of 0.27527 is retrieved for testing equivalence class 4 with equivalence class 6. But, not only the analysis of these two among themselves result in lower score, also comparing the other equivalence classes with these two results in lower scores in comparison to the other classes. When comparing the first two equivalence classes with the last two, the p-value is always the same, 0.13495.

The statistical analysis with the help of the Two-Sample Kolmogorov-Smirnov test showed some interesting insights, where the different framework versions show a higher cohesion of results than across different frameworks. This impression account across all investigated operating systems. Further, the statistical analysis of the Java framework lead to the conclusion that this framework seems to be more unstable than the Python frameworks. The instability is observed on all operating systems, where the Deeplearning4J versions 0.8.0 and 0.9.1 show lower results than observed for other frameworks. Even when testing these two versions with version 0.7.1 it shows lower scores. As a result, further investigations are conducted for this framework versions which are described in Section 4.6.6.

4.6.5 Cross-operating system equivalence class testing

In order to investigate if the operating system type has an influence on the results when reproducing the Deep Learning experiment, all 19 equivalence classes are tested against each other. Table 4.24 summarizes the results of this statistical analysis.

The Two-Sample Kolmogorov-Smirnov Test for equivalence class 1 on Mac OS and Linux yields a d-value of 0.10. The same results are obtained when testing equivalence class 1 from Mac OS and equivalence class 1 from Windows. This means that the first equivalence class on all operating systems deliver nearly the same results and the operating system do not have a significant influence. As already mentioned, equivalence class 3 deliver exactly the same results for all operating systems. When comparing equivalence class 4 from Mac OS,(represented by Theano 1.0.0) with equivalence class 3 (represented by Theano 0.9.0) from other operating systems only small fluctuations of results occur. The d-values is 0.15 in both cases and the null hypothesis is retained.

In some cases the d-value is 0.40 and therefore near the critical value d_crit (0,43). Even when a d-value of 0.40 is not significant, a confident conclusion that these results originate from the same distribution is risky. This d-value is obtained when testing following equivalence classes against each other:

- M2 and L6
- M2 and W6
- M5 and L5
- M5 and L6
- M5 and W5
- L1 and W6
- L2 and L6
- L4 and W5

So, by testing each equivalence class against each other no significant results are obtained, except when testing L4 against L6 (DL4J 0.7.1 and DL4J 0.9.1 on Linux-based operating systems). This analysis concludes that the results from the equivalence classes originate from the same distribution and showing that the random seed does not have an impact.

Eq. class	M1	M2	M3	M4	M5	M6	M7	L1	L2	L3	L4	L5	L6	W1	W2	W3	W4	W5	W6
M1		0.10	0.25	0.30	0.20	0.25	0.30	0.10	0.15	0.25	0.15	0.35	0.35	0.10	0.15	0.25	0.15	0.30	0.35
M2	0.10		0.25	0.30	0.20	0.30	0.35	0.15	0.20	0.25	0.20	0.35	0.40	0.10	0.15	0.25	0.15	0.35	0.40
M3	0.25	0.25		0.15	0.20	0.15	0.25	0.20	0.25	0	0.20	0.25	0.25	0.25	0.20	0	0.25	0.25	0.25
M4	0.30	0.30	0.15		0.20	0.15	0.25	0.30	0.25	0.15	0.25	0.25	0.25	0.35	0.25	0.15	0.30	0.25	0.30
M5	0.20	0.20	0.20	0.20		0.15	0.35	0.20	0.25	0.20	0.10	0.40	0.40	0.25	0.15	0.20	0.15	0.40	0.30
M6	0.25	0.30	0.15	0.15	0.15		0.25	0.20	0.30	0.15	0.20	0.30	0.25	0.25	0.20	0.15	0.20	0.25	0.25
M7	0.30	0.35	0.25	0.25	0.35	0.25		0.35	0.35	0.25	0.35	0.15	0.10	0.35	0.35	0.25	0.30	0.10	0.15
L1	0.10	0.15	0.20	0.30	0.20	0.20	0.35		0.15	0.20	0.20	0.35	0.35	0.10	0.15	0.20	0.20	0.35	0.40
L2	0.15	0.20	0.25	0.25	0.25	0.30	0.35	0.15		0.25	0.20	0.35	0.40	0.15	0.20	0.25	0.20	0.40	0.35
L3	0.25	0.25	0	0.15	0.20	0.15	0.25	0.20	0.25		0.20	0.25	0.25	0.25	0.20	0	0.25	0.25	0.25
L4	0.15	0.20	0.20	0.25	0.10	0.20	0.35	0.20	0.20	0.20		0.40	**0.45**	0.20	0.20	0.20	0.15	0.40	0.30
L5	0.35	0.35	0.25	0.25	0.40	0.30	0.15	0.35	0.35	0.25	0.40		0.15	0.35	0.35	0.25	0.35	0.15	0.15
L6	0.35	0.40	0.25	0.25	0.40	0.25	0.10	0.35	0.40	0.25	**0.45**	0.15		0.30	0.30	0.25	0.35	0.10	0.15
W1	0.10	0.10	0.25	0.35	0.25	0.25	0.35	0.10	0.15	0.25	0.20	0.35	0.30		0.20	0.25	0.20	0.35	0.35
W2	0.15	0.15	0.20	0.25	0.15	0.20	0.35	0.15	0.20	0.20	0.20	0.35	0.30	0.20		0.20	0.20	0.35	0.35
W3	0.25	0.25	0	0.15	0.20	0.15	0.25	0.20	0.25	0	0.20	0.25	0.25	0.25	0.20		0.25	0.25	0.25
W4	0.15	0.15	0.25	0.30	0.15	0.20	0.30	0.20	0.20	0.25	0.15	0.35	0.35	0.20	0.20	0.25		0.35	0.30
W5	0.30	0.35	0.25	0.25	0.40	0.25	0.10	0.35	0.40	0.25	0.40	0.15	0.10	0.35	0.35	0.25	0.35		0.15
W6	0.35	0.40	0.25	0.30	0.30	0.25	0.15	0.40	0.35	0.25	0.30	0.15	0.15	0.35	0.35	0.25	0.30	0.15	

Table 4.24: d-values of the Two-Sample Kolmogorov-Smirnov Test for comparing equivalence classes across operating systems

4.6.6 Variance Analysis

This Section concentrates on further investigating some Deep Learning frameworks, mainly the Java framework highlighted an unstable behavior on all operating systems. Therefore, the variance of the equivalence classes are considered, because it measures how far a set of numbers are spread out from their average value. The variances for all equivalence classes is shown in Table 4.25.

MacOS		**Linux**		**Windows**	
Eq. class	**Variance**	**Eq. class**	**Variance**	**Eq. class**	**Variance**
M1	$6.8068452E10^{-6}$	L1	$7.878472E10^{-6}$	W1	$7.82315E10^{-6}$
M2	$6.6792363E10^{-6}$	L2	$6.0394577E10^{-6}$	W2	$7.3227266E10^{-6}$
M3	$6.0044913E10^{-6}$	L3	$6.0044913E10^{-6}$	W3	$6.0044913E10^{-6}$
M4	$4.921163E10^{-6}$				
M5	$5.3235867E10^{-6}$	L4	$5.7844845E10^{-6}$	W4	$5.290602E10^{-6}$
M6	$7.767228E10^{-6}$	L5	$8.51474E10^{-6}$	W5	$8.266827E10^{-6}$
M7	$8.854833E10^{-6}$	L6	$9.105658E10^{-6}$	W6	$9.10261E10^{-6}$

Table 4.25: Variances calculated for accuracy for every equivalence class per operating system type

The variances for equivalence classes across all operating systems show a similar result, where version 0.8.0 and 0.9.1 of Deeplearning4J shows the highest variances, which confirm the assumption that these versions are more unstable than the others in this master thesis. Further investigation consider the three Deep Learning Java framework versions run on Linux-based systems, as on this operating systems in one case a significant results was obtained. For further investigation the training epochs are considered as important factor, as the model may need more training time to deliver more stable results.
The hypothesis for this analysis is that more training epochs lead to more stable results and a smaller variance and in a not significant result when conducting the Two-Sample Kolmogorov-Smirnov test, meaning that the obtained results originate from the same distribution. The Deep Learning model implemented in these three mentioned frameworks is therefore trained with 10 epochs, 20 epochs and 30 epochs and afterwards the variance is calculated.

Framework	DL4J 0.7.1		
Epochs	10	20	30
Variance	5.7844845E10^{-6}	3.6443123E10^{-6}	2.7409424E10^{-6}
Framework	DL4J 0.8.0		
Epochs	10	20	30
Variance	8.51474E10^{-6}	5.2851583E10^{-6}	2.80472E10^{-6}
Framework	DL4J 0.9.1		
Epochs	10	20	30
Variance	9.105658E10^{-6}	5.8445016E10^{-6}	3.2947432E10^{-6}

Table 4.26: Variances calculated for most unstable frameworks, where the model was trained with different number of epochs

As shown in Table 4.26, the variances decrease when the model is trained longer.

	10 Epochs			**30 Epochs**		
Equivalence class	**L4**	**L5**	**L6**	**L4**	**L5**	**L6**
L4		0.40	**0.45**		0.20	0.25
L5	0.40		0.15	0.20		0.20
L6	**0.45**	0.15		0.25	0.20	

Table 4.27: d-values of the Two-Sample Kolmogorov-Smirnov Test executed for Linux-based systems, considering the Java-based Framework and its different versions when training the model with 30 Epochs

	10 Epochs			**30 Epochs**		
Equivalence class	**L4**	**L5**	**L6**	**L4**	**L5**	**L6**
L4		0.05914	**0.02321**		0.77095	0.49734
L5	0.05914		0.96548	0.77095		0.77095
L6	**0.0231**	0.96548		0.49734	0.77095	

Table 4.28: p-values of the Two-Sample Kolmogorov-Smirnov Test executed for Linux-based systems, considering the Java-based Framework and its different versions when training the model with 30 Epochs

Table 4.27 shows the d-values for the model implemented in the different versions of DL4J and trained for 30 epochs. Table 4.28 shows the corresponding p-values. As we already observed, when testing equivalence class 4 and 6 when trained for 10 epochs the d-value (0.45) is significant. Now, when the model is trained 30 epochs, the d-value decreases to 0.25. Concluding that more training epochs lead to a more stable behavior of the mentioned frameworks.

4.7 Summary

In this Chapter the results of the experiments were discussed. One research question was if it is even possible to construct the same Deep Learning model in different Deep Learning frameworks. In this master thesis it is showen that this is possible but a clear specification of the model should be available as a small uncertainty about configuration could lead to a different model which may lead to different results.
Further, the performance of the Deep Learning model were analyzed when changing a specific execution stack parameter while the other remained the same. Fortunately, all results were not significantly different except for testing equivalence class 4 and 6 on Linux. This significant result resulted from the fact that the model were not trained for too long and the variance were respectively big. Further investigation were conducted considering these equivalence classes where it is shown that the variance is lower when training the model for longer epochs. Additionally, the Two-Sample Kolmogorov-Smirnov test showed no significantly different result. Finally, the equivalence classes were analyzed across different operating systems revealing no significant differences.

CHAPTER 5

Conclusion and future work

Reproducibility of scientific research is very important for scientists as it enables the researchers to conduct the same experiments and proof that the original results are trustworthy. Nevertheless, enabling reproducibility is easier said than done. The reason for this is that a lot of information is needed, beginning from the data that was used in the original study, access to used program code, configuration data and more. Of course, the researchers that publish the original study need to provide this information. The goal of this master thesis was to investigate if certain execution stack parameters have an impact on the reproducibility of a specific Deep Learning model. The execution stack parameters that were investigated are:

- Deep Learning framework and different framework versions
- Execution platform and different execution platform version
- Operating system types and different operating system versions

The basic Deep Learning model architecture that was used in this thesis is the famous LeNet-5 [LBBH98] which is mainly used for image classification. The model was reproduced in three Deep Learning frameworks (TensorFlow, Theano and Deeplearning4J). Furthermore, different versions of these frameworks were considered. Additionally, different execution platforms as well as different operating systems were considered. Operating system that were used in this master thesis are:

i) Mac
   a) Mac OS High Sierra 10.13.5

ii) Linux
   a) Linux Fedora 25
   b) Linux Mint 18.3
   c) Linux Ubuntu 16.04

iii) WIndows
   a) Windows 7
   b) Windows 8
   c) Windows 10

A part of the thesis concentrated on testing if those factors have influence on the obtained metrics when running the model. By analyzing different framework versions, it turned out that not all versions deliver the same result, but the differences are in the most cases not significant, except for one configuration. On Linux the comparison of Deeplearning4J 0.7.1 and Deeplearning4J 0.9.1 yield a p-value that is below the significance level alpha. Also, comparing Deeplearning4J 0.7.1 with Deeplearning4J 0.8.0 and TensorFlow 1.6.0 with Deeplearning4J 0.9.1 lead to almost significant differences. Yet, a more detailed analysis of these framework versions where the model is trained with more epochs showed that a well trained model result in none siginificant d-values, meaning the obtained result originate from the same distribution. Further, different frameworks achieve slightly different results, which are not questionable in fact. Identical results are achieved when using different execution platform versions, regardless if it is Python or Java. Even a different Java Virtual Machine lead to same results. Additionally, different operating system versions do not affect Deep Learning model results. Finally, the operating system has an influence on Deep Learning model results, but the results are not statistically significant as they originate from the same distribution.

Fortunately, except for one configuration, no significant result were obtained by the statistical analysis. All necessary information to facilitate reproducibility should be done from the very beginning of the study. This allows others to validate certain results, which is really important in scientific research. In fact, an exact description of the Deep Learning model is needed as small uncertainties of some model configurations may lead to different results [DRF18].

In order to make reproducibility of scientific research easier the PRIMAD model was used, as it enables the researcher to document the configuration to reproduce the conducted experiments. Due to the fact that Machine Learning and especially Deep Learning as part of it is currently widely used in scientific research, close attention should be placed on reproducibility of the research as a lot of factors can influence results.

Therefore, the PRIMAD model can be a useful tool enabling detailed specification of experiment configurations, including all necessary data for reproducing the study and increasing trustworthiness. As data science and big data are coming more and more, Deep Learning may be considered as important tool. And even if the idea behind Deep Learning is not new, the current opportunity to execute those computationally intensive calculations on high-end hardware and at low costs will demand for reproducible research studies.

Due to the fact that this thesis focused on investigating on certain execution stack parameters, without focusing on the underlying hardware, opens the door for future research. The Deep Learning model was run on CPUs. Deep Learning models are normally run on GPUs which speeds up the model execution. Future research regarding the hardware on which Deep Learning models are executed would be very interesting. Further, all operating system were run in virtual machines, except for Mac OS High Sierra which served as the base operating system. Possible VM effects were not investigated in this master thesis and open the door for further investigations. The Deep Learning model that was used in this thesis is not the biggest and most complex. It would be interesting if more complex Deep Learning models, which use more complicated calculations, would deliver the same result as the model used in this thesis.

APPENDIX A

Implementation

A.1 Converting IDX to CSV

```
PATH = "Dataset/"
def convert(imgf, labelf, outf, startPoint, endPoint):
  f = open(imgf, "rb")
  o = open(outf, "w")
  l = open(labelf, "rb")
  f.read(16)
  l.read(8)
  images = []

  for i in range(startPoint, endPoint):
    image = [ord(l.read(1))]
    for j in range(28*28):
      image.append(ord(f.read(1)))
      images.append(image)
  for image in images:
    o.write(",".join(str(pix) for pix in image)+"\n")
  f.close()
  o.close()
  l.close()

  #converting train images + labels
  convert(PATH + "train-images-idx3-ubyte", PATH + "train-labels-idx1-ubyte",
  PATH + "mnist_train.csv", 0, 60000)
  #converting tet images + labels
  convert(PATH + "t10k-images-idx3-ubyte", PATH + "t10k-labels-idx1-ubyte",
  PATH + "mnist_test.csv", 0, 10000)
```

Listing A.1: Code for converting original MNIST data into CSV file format.[1]

[1] https://pjreddie.com/projects/mnist-in-csv/

A.2 Data preperation for Python frameworks

```
1
2  #preparing images
3  def prepareImages(path, fw="tensorflow"):
4
5    ### preparing images ###
6    # fetching training data
7    data = pd.read_csv(path, header=None)
8
9    # seperate images from data
10   images = (data.iloc[:, 1:].values).astype('float32') # all pixel
     values of images
11
12   # reshaping train data
13   if fw == "tensorflow":
14     images = images.reshape(images.shape[0], 28, 28, 1)
15   if fw == "theano":
16     images = images.reshape(images.shape[0], 1, 28, 28)
17
18   # scaling data
19   images = images / 255
20
21   return images
22
23 #preparing lables of corresponding images
24 def prepareLables(path):
25   # fetching training data
26   trainData = pd.read_csv(path, header=None)
27
28   #seperate lables from data
29   lables = trainData.iloc[:, 0].values.astype('int32') # get lables
30
31   ### preparing lables ###
32   # One-hot encoding lables
33   def dense_to_one_hot(labels_dense, num_classes):
34     num_labels = labels_dense.shape[0]
35     index_offset = np.arange(num_labels) * num_classes
36     labels_one_hot = np.zeros((num_labels, num_classes))
37     labels_one_hot.flat[index_offset + labels_dense.ravel()] = 1
38     return labels_one_hot
39
40   # get number of classes
41   num_classes = np.unique(lables).shape[0]
42
43   lables = lables.astype(np.uint32)
44
45   return lables
```

Listing A.2: Methods used to prepare the dataset for further use. The first method prepares the images while the second method prepares the labels.

A.3 Data preperation for Java framework

```
1
2  public static DataSetIterator prepareData(String path, int batchSize)
     throws IOException, InterruptedException {
3
4    //read csv file
5    final int numLinesToSkip = 0;
6    final char delimiter = ',';
7    final int labelIndex = 0;
8    final int numClasses = 10;
9
10   //csv record readers
11   RecordReader reader = new CSVRecordReader(numLinesToSkip, delimiter);
12
13   //initialize recordreader
14   reader.initialize(new FileSplit(new ClassPathResource(path).getFile()))
     ;
15
16   //create data iterator for training and testing
17   DataSetIterator data = new RecordReaderDataSetIterator(reader,
     batchSize, labelIndex, numClasses);
18
19   //normalize data
20   //scale data from 0-255 to 0-1
21   DataNormalization scaler = new ImagePreProcessingScaler(0,1);
22   scaler.fit(data);
23   data.setPreProcessor(scaler);
24
25   return data;
26   }
```

Listing A.3: Methods used to prepare the dataset for further use.The method prepares the required format and returns a DataSetIterator object.

A.4 TensorFlow

```
#LeNet model configuration
def LeNet(x, one_hot_lables, learningRate, seed):

#Weight initialization according to Xavier Glorot and Yoshua Bengio
initializer = tf.contrib.layers.xavier_initializer_conv2d(uniform=False, seed=seed)

#Layer 1: Convolutional Layer: Input = 28x28x1. Output = 24x24x4
conv1_W = tf.get_variable("W1", shape=(5, 5, 1, 4), initializer=initializer)
conv1_b = tf.Variable(tf.zeros(4))
conv1 = tf.nn.conv2d(x, conv1_W, strides=[1, 1, 1, 1], padding='VALID') + conv1_b

#RELU Activation function
conv1 = tf.nn.relu(conv1)

#Layer 2: Pooling Layer: Input = 24x24x4. Output = 12x12x4
conv1 = tf.nn.max_pool(conv1, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

#Layer 3: Convolutional Layer: Output = 8x8x12
conv2_W = tf.get_variable("W2", shape=(5, 5, 4, 12), initializer=initializer)
conv2_b = tf.Variable(tf.zeros(12))
conv2 = tf.nn.conv2d(conv1, conv2_W, strides=[1, 1, 1, 1], padding='VALID') + conv2_b

#RELU Activation function
conv2 = tf.nn.relu(conv2)

#Layer 4: Pooling Layer: Input = 8x8x12. Output = 4x4x12
conv2 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='VALID')

#Flatten Layer Input = 4x4x12. Output = 192
fc0 = flatten(conv2)

#Layer 5: Fully Connected Layer: Input = 192. Output = 120
fc1_W = tf.get_variable("W3", shape=(192, 120), initializer=initializer)
fc1_b = tf.Variable(tf.zeros(120))
fc1 = tf.matmul(fc0, fc1_W) + fc1_b

#RELU Activation function
fc1 = tf.nn.relu(fc1)

#Layer 6: Fully Connected Layer: Input = 120. Output = 84
fc2_W = tf.get_variable("W4", shape=(120, 84), initializer=initializer)
fc2_b = tf.Variable(tf.zeros(84))
fc2 = tf.matmul(fc1, fc2_W) + fc2_b

```

```
#RELU Activation function
fc2 = tf.nn.relu(fc2)

#Layer 7: Fully Connected Layer: Input = 84. Output = 10
fc3_W = tf.get_variable("W5", shape=(84, 10), initializer=initializer)
fc3_b = tf.Variable(tf.zeros(10))
logits = tf.matmul(fc2, fc3_W) + fc3_b

#Define Loss Function - Cross Entropy
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(labels=
    one_hot_lables, logits=logits)
loss_operation = tf.reduce_mean(cross_entropy)
#Define Weight Optimization Algorithm - SDG
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learningRate)
training_operation = optimizer.minimize(loss_operation)

return logits, training_operation
```

Listing A.4: Model implementation in TensorFlow 1.4.0

```

 #function to execute normal model (train: 60k; test: 10k)
 def normalModel(trainImages, trainLables, testImages, testLables, nEpochs
   , batchSize, seed, learningRate):

   # set random seed for reproducibility
   tf.set_random_seed(seed)
   np.random.seed(seed)

   # confusion matrix filename
   confusionMatrixFileName = "Output/CM/
   tensorflow_NormalModelConfusionmatrix_for_seed_{}.txt".format(seed)

   # placeholder variables for batches of input images and batches of
   output labels
   x = tf.placeholder(tf.float32, (None, 28, 28, 1))
   y = tf.placeholder(tf.int32, (None))
   one_hot_y = tf.one_hot(y, 10)

   # create model and training operation
   logits, training_operation = model.LeNet(x, one_hot_y, learningRate,
   seed)

   #dictionary object for results
   results = {}

   #number of examples in train set
   num_examples = len(trainImages)

   with tf.Session() as sess:
     sess.run(tf.global_variables_initializer())
     print "******************** Train model ********************"
     for i in xrange(nEpochs):
       start_time = time.time()
```

```
31          for offset in xrange(0, num_examples, batchSize):
32            end = offset + batchSize
33            batch_x, batch_y = trainImages[offset:end], trainLables[offset:
    end]
34            sess.run(training_operation, feed_dict={x: batch_x, y: batch_y})
35          print "Completed epoch {} in {:.3f} seconds".format((i + 1), (time.
    time() - start_time))
36
37        #Get the predictions
38        prediction = tf.argmax(logits, 1)
39        y_pred = sess.run(prediction, feed_dict={x: testImages, y: testLables
    })
40        #true values
41        y_true = testLables
42
43      #Calculate metrics
44      accuracy = accuracy_score(y_true, y_pred)
45      precision = precision_score(y_true, y_pred, average='macro')
46      recall = recall_score(y_true, y_pred, average='macro')
47      f1 = f1_score(y_true, y_pred, average='macro')
48
49      print "******************** Evaluate model ********************"
50      print "Accuracy: {:.5f}".format(accuracy)
51      print "Precision: {:.5f}".format(precision)
52      print "Recall: {:.5f}".format(recall)
53      print "F1-Score: {:.5f}".format(f1)
54      print "****************Finished********************"
55
56      #put metrics into dictionary
57      results["Accuracy"] = accuracy
58      results["Precision"] = precision
59      results["Recall"] = recall
60      results["F1-Score"] = f1
61
62      # reset graph
63      tf.reset_default_graph()
64
65      return results
```

Listing A.5: Method to train and test model (Tensorflow)

A.5 Theano

```
def createTheanoModel(x, batchSize, seed):

#Define input dimensions
layer1_input = x.reshape((batchSize, 1, 28, 28))

# Layer 1: Convolutional Layer: Input = 28x28x1. Output = 24x24x4 +
# Layer 2: Pooling Layer: Input = 24x24x4. Output = 12x12x4
layer1 = LeNetConvPoolLayer(
seed,
input=layer1_input,
image_shape=(batchSize, 1, 28, 28),
filter_shape=(4, 1, 5, 5)
)

# Layer 3: Convolutional Layer: Output = 8x8x12 +
# Layer 4: Pooling Layer: Input = 8x8x12. Output = 4x4x12
layer2 = LeNetConvPoolLayer(
seed,
input=layer1.output,
image_shape=(batchSize, 4, 12, 12),
filter_shape=(12, 4, 5, 5)
)

# Flatten Layer Input = 4x4x12. Output = 192
layer3_input = layer2.output.flatten(2)

# Layer 5: Fully Connected Layer: Input = 192. Output = 120
layer3 = HiddenLayer(
seed,
input=layer3_input,
n_in=12 * 4 * 4,
n_out=120,
)

# Layer 6: Fully Connected Layer: Input = 120. Output = 84
layer4 = HiddenLayer(
seed,
input=layer3.output,
n_in=120,
n_out=84,
)

# Layer 7: Fully Connected Layer: Input = 84. Output = 10
logits = ModelOutputSoftmax(input=layer4.output, n_in=84, n_out=10, rng=
    seed)

#Parameter (weights + bias) -> needed for gradient computation
params = logits.params + layer4.params + layer3.params + layer2.params +
    layer1.params

```

```
49 return logits, params
```

Listing A.6: Model implementation in Theano 0.9.0

```
1 #Definition of Convolutional Layer + Pooling Layer
2 class LeNetConvPoolLayer(object):
3
4 def __init__(self, rng, input, filter_shape, image_shape, poolsize=(2, 2)):
5
6 assert image_shape[1] == filter_shape[1]
7 self.input = input
8
9 #Weight initialization according to Xavier Glorot and Yoshua Bengio
10 fan_in = np.prod(filter_shape[1:])
11 fan_out = (filter_shape[0] * np.prod(filter_shape[2:]) //
12 np.prod(poolsize))
13 W_bound = np.sqrt(2. / (fan_in + fan_out))
14 self.W = theano.shared(
15 np.asarray(
16 rng.normal(size=filter_shape, scale=W_bound),
17 dtype=theano.config.floatX
18 ),
19 borrow=True
20 )
21
22 #The bias is a 1D tensor -- one bias per output feature map
23 b_values = np.zeros((filter_shape[0],), dtype=theano.config.floatX)
24 self.b = theano.shared(value=b_values, borrow=True)
25
26 #Definition of the Convolutional Layer
27 conv_out = conv2d(
28 input=input,
29 filters=self.W,
30 filter_shape=filter_shape,
31 image_shape=image_shape,
32 subsample=(1, 1),
33 border_mode='valid'
34 )
35
36 #RELU Activation function
37 conv_out = T.nnet.relu(conv_out + self.b.dimshuffle('x', 0, 'x', 'x'))
38
39 #Definition of Pooling Layer
40 pooled_out = pool.pool_2d(
41 input=conv_out,
42 ws=poolsize,
43 ignore_border=True,
44 stride=(2,2),
45 pad=(0,0)
46 )
47
48 #Output result of Layers
49 self.output = pooled_out
50
```

```
#Store parameters of this layer
self.params = [self.W, self.b]

#Keep track of model input
self.input = input
```

Listing A.7: Convolutional and Pooling layer implementation in Theano 0.9.0

```
#Define Fully Connected Layer
class HiddenLayer(object):
def __init__(self, rng, input, n_in, n_out):

self.input = input

#Weight initialization according to Xavier Glorot and Yoshua Bengio
W_values = np.asarray(
rng.normal(size=(n_in, n_out), scale=np.sqrt(2. / (n_in + n_out))
),
dtype=theano.config.floatX
)
W = theano.shared(value=W_values, name='W', borrow=True)

#The bias is a 1D tensor -- one bias per output feature map
b_values = np.zeros((n_out,), dtype=theano.config.floatX)
b = theano.shared(value=b_values, name='b', borrow=True)

self.W = W
self.b = b

#Calculate output of Layer
lin_output = T.dot(input, self.W) + self.b
# Output result of Layer with RELU activation function
self.output = (T.nnet.relu(lin_output))
#Parameters of the model
self.params = [self.W, self.b]
```

Listing A.8: Fully connected layer implementation in Theano 0.9.0

```
#Definition of Output Layer, Error calculation + Loss Function
class ModelOutputSoftmax(object):

def __init__(self, input, n_in, n_out, rng):

#Weight initialization according to Xavier Glorot and Yoshua Bengio
W_values = np.asarray(
rng.normal(size=(n_in, n_out), scale=np.sqrt(2. / (n_in + n_out))
),
dtype=theano.config.floatX
)
W = theano.shared(value=W_values, name='W', borrow=True)
self.W = W
#The bias is a 1D tensor -- one bias per output feature map
self.b = theano.shared(
value=np.zeros(
```

```
(n_out,),
dtype=theano.config.floatX
),
name='b',
borrow=True
)
self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
#Symbolic description of how to compute prediction as class whose
#Probability is maximal
self.y_pred = T.argmax(self.p_y_given_x, axis=1)

#Parameters of the model
self.params = [self.W, self.b]

#Keep track of model input
self.input = input

#Method to calculate the errors made by model
def errors(self, y):
# check if y has same dimension of y_pred
if y.ndim != self.y_pred.ndim:
raise TypeError(
'y should have the same shape as self.y_pred',
('y', y.type, 'y_pred', self.y_pred.type)
)
# check if y is of the correct datatype
if y.dtype.startswith('int'):
# the T.neq operator returns a vector of 0s and 1s, where 1
# represents a mistake in prediction
return T.mean(T.neq(self.y_pred, y))
else:
raise NotImplementedError()
#Definition of the Loss Function
def cross_entropy(self, y):
return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
```

Listing A.9: Output layer implementation in Theano 0.9.0

```
  def createDataSet(img, lbl, batchSize):
    #change type of images to theano.float
    img = img.astype(theano.config.floatX)
    #create shared variable for images
    set_x = theano.shared(img)
    #create shared variable for lables
    #train set
    set_y = theano.shared(lbl)
    set_y = set_y.flatten()
    set_y = T.cast(set_y, 'int32')
    #get number of train batches
    n_batches = img.shape[0] / batchSize

    return set_x, set_y, n_batches

```

```
16  def normalModel(trainImages, trainLables, testImages, testLables, nEpochs
      , batchSize, seed, learningRate):
17
18    # set random seed for reproducibility
19    SEED = np.random.RandomState(seed)
20
21    # confusion matrix filename
22    confusionMatrixFileName = "Output/CM/
      theano_NormalModelConfusionmatrix_for_seed_{}.txt".format(seed)
23
24    # placeholder variables for batches of input images and batches of
      output labels
25    x = T.tensor4('x')  # the data is presented as rasterized images
26    y = T.ivector('y')
27    # allocate symbolic variables for the data
28    # index to a [mini]batch
29    index = T.lscalar()
30
31    # build actual model
32    logits, params = model.createTheanoModel(x, batchSize, SEED)
33
34    # define loss function
35    loss = logits.cross_entropy(y)
36
37    # define weight optimization - SDG
38    grads = T.grad(loss, params)
39
40    updates = [
41      (param_i, param_i - learningRate * grad_i)
42      for param_i, grad_i in zip(params, grads)
43    ]
44
45    #dictionary object for results
46    results = {}
47
48    #create datasets
49    train_set_x, train_set_y, n_train_batches = createDataSet(trainImages,
      trainLables, batchSize)
50    test_set_x, test_set_y, n_test_batches = createDataSet(testImages,
      testLables, batchSize)
51
52    #train operation
53    train_model = theano.function(
54      [index],
55      logits.errors(y),
56      updates=updates,
57      givens={
58        x: train_set_x[index * batchSize: (index + 1) * batchSize],
59        y: train_set_y[index * batchSize: (index + 1) * batchSize]
60      })
61
62    #evaluation operation
63    predictions = theano.function(
```

```
64        [index],
65        outputs=logits.y_pred,
66        givens={
67          x: test_set_x[index * batchSize: (index + 1) * batchSize]
68        }
69      )
70
71      print "******************* Train model *******************"
72      for i in xrange(nEpochs):
73        start_time = time.time()
74        error = 0
75        for j in xrange(n_train_batches):
76          error += train_model(j)
77        print "Epoch: {}; Score: {}".format((i + 1), (error / n_train_batches
      ))
78        print "Completed epoch {} in {:.3f} seconds".format((i + 1), (time.
      time() - start_time))
79
80      print "******************* Evaluate model *******************"
81      #concatenate predictions into one variable
82      y_pred = np.concatenate([predictions(p) for p in range(n_test_batches)
      ])
83      #ground truth values
84      y_true = testLables
85
86      #Calculate metrics
87      accuracy = accuracy_score(y_true, y_pred)
88      precision = precision_score(y_true, y_pred, average='macro')
89      recall = recall_score(y_true, y_pred, average='macro')
90      f1 = f1_score(y_true, y_pred, average='macro')
91
92      # print results in dictionary
93      print "Accuracy: {:.5f}".format(accuracy)
94      print "Precision: {:.5f}".format(precision)
95      print "Recall: {:.5f}".format(recall)
96      print "F1-Score: {:.5f}".format(f1)
97      print "****************Finished*******************"
98
99      #put metrics into dictionary
100     results["Accuracy"] = accuracy
101     results["Precision"] = precision
102     results["Recall"] = recall
103     results["F1-Score"] = f1
104
105     return results
```

Listing A.10: Method used to train and test the actual model. Further, the results are printed to the console. Note, method ćreateDataSetpŕepares the dataset for further use.

A.6 Deeplearning4J

```
public MultiLayerConfiguration createModel(){

MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
//Random seed for reproducibility
.seed(this.seed)
//Weight initialization according to Xavier Glorot and Yoshua Bengio
.weightInit(WeightInit.XAVIER)
//Use stochastic gradient descent as an optimization algorithm
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.iterations(1)
//Specify the learning rate
.learningRate(this.learningRate)
.list()
// Layer 1: Convolutional Layer: Input = 28x28x1. Output = 24x24x4
.layer(0, new ConvolutionLayer.Builder(5, 5)
.nIn(1)
.stride(1, 1)
.padding(new int[]{0,0})
.nOut(4)
//RELU Activation function
.activation(Activation.RELU)
.build())
//Layer 2: Pooling Layer: Input = 24x24x4. Output = 12x12x4
.layer(1,new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
.kernelSize(2,2)
.stride(2,2)
.padding(new int[]{0,0})
.build())
//Layer 3: Convolutional Layer: Output = 8x8x12
.layer(2, new ConvolutionLayer.Builder(5, 5)
//Note that nIn need not be specified in later layers
.stride(1, 1)
.padding(new int[]{0,0})
.nOut(12)
//RELU Activation function
.activation(Activation.RELU)
.build())
//Layer 4: Pooling Layer: Input = 8x8x12. Output = 4x4x12
.layer(3,new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
.kernelSize(2,2)
.stride(2,2)
.padding(new int[]{0,0})
.build())
//Layer 5: Fully Connected Layer: Input = 192. Output = 120
.layer(4,new DenseLayer.Builder().activation(Activation.RELU)
.nOut(120).build())
//Layer 6: Fully Connected Layer: Input = 120. Output = 84
.layer(5, new DenseLayer.Builder().activation(Activation.RELU)
.nOut(84).build())
//Layer 7: Fully Connected Layer: Input = 84. Output = 10
//With Cross Entropy as Loss Function
```

```
.layer(6, new OutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
.nOut(10)
.activation(Activation.SOFTMAX)
.build())
.setInputType(InputType.convolutionalFlat(28,28,1)) //See note below
.backprop(true) //use backpropagation to adjust weights
.pretrain(false).build();

return conf;

}
```

Listing A.11: Model implementation in DL4J 0.9.1

```

public static HashMap<String, String> normalModel(String trainPath, String
    testPath, int nEpochs, int batchSize, int seed, double learningRate,
    String filename) throws IOException, InterruptedException {
    //save results in HashMap object
    HashMap<String, String> results = new HashMap<String, String>();

    //dataset creation
    final short trainSetIndex = 0;
    final short testSetIndex = 1;
    DataSetIterator trainData = DataPreperation.prepareData(trainPath,
    batchSize);
    DataSetIterator testData = DataPreperation.prepareData(testPath,
    batchSize);

    //build and configure model
    Lenet lenet = new Lenet(batchSize, nEpochs, seed, learningRate);
    MultiLayerConfiguration conf = lenet.createModel();
    MultiLayerNetwork model = new MultiLayerNetwork(conf);
    model.init();
    model.setListeners(new ScoreIterationListener(100));

    log.info("Train model ...");
    for (int i = 0; i < nEpochs; i++){
      Stopwatch stopwatch = Stopwatch.createStarted();
      model.fit(trainData);
      log.info("Completed epoch {} in {} seconds", (i+1), stopwatch.elapsed
    (TimeUnit.SECONDS));
      trainData.reset();
    }

    log.info("Evaluate model ...");
    Evaluation eval = model.evaluate(testData);
    log.info(eval.stats());
    model = null;

    //put fold metrics into HashMap
    results.put("Accuracy", Double.toString(eval.accuracy()));
    results.put("Precision", Double.toString(eval.precision()));
    results.put("Recall", Double.toString(eval.recall()));
```

```
36      results.put("F1-Score", Double.toString(eval.f1()));
37      log.info("****************Example finished********************");
38
39      return results;
40      }
```

Listing A.12: The method is used to train and test the model. Further, the results are printed to the console.

List of Figures

List of Tables

Bibliography

[Aar15] Joanna E; Anderson Christopher J; Attridge Peter R; Attwood Angela; Axt Jordan; Babel Molly; Bahnik Stepan; Baranski Erica; Barnett-Cowan Michael; Bartmess Elizabeth; Beer Jennifer; Bell Raoul; Bentley Heather; Beyan Leah; Binion Grace; Borsboom Denny; Bosch Annick; Bosco Frank A.; Bowman Sara D.; Brandt Mark J; Braswell Erin; Brohmer Hilmar; Della Penna Nicolas Aarts, Alexander A; Anderson. Estimating the reproducibility of psychological science. *Science*, 349(6251), 2015.

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[BBL+11] James Bergstra, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, Ian Goodfellow, Arnaud Bergeron, Yoshua Bengio, and Pack Kaelbling. Theano: Deep learning on gpus with python. In *Big Learn workshop, NIPS'11*, 2011.

[Boe15] Carl Boettiger. An introduction to docker for reproducible research. *SIGOPS Operating Systems Review*, 49(1):71–79, January 2015.

[Bot10] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *International Conference on Computational Statistics*, pages 177–186, 2010.

[CBBHB95] J. Chris Bishop, C.M. Bishop, G. Hinton, and P.N.C.C.M. Bishop. *Neural Networks for Pattern Recognition*. Advanced Texts in Econometrics. Clarendon Press, 1995.

[Den12] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141 – 142, 2012.

[DRF18] Alexander Dür, Andreas Rauber, and Peter Filzmoser. Reproducing a neural question answering architecture applied to the squad benchmark dataset: Challenges and lessons learned. In Gabriella Pasi, Benjamin Piwowarski, Leif Azzopardi, and Allan Hanbury, editors, *Advances in Information Retrieval*, pages 102–113, Cham, 2018. Springer International Publishing.

[FFR16] Juliana Freire, Norbert Fuhr, and Andreas Rauber. Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041). *Dagstuhl Reports*, 6(1):108–159, 2016.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Artificial Intelligence and Statistics*, volume 9, pages 249 –256, 2010.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[GFI16] Steven N. Goodman, Daniele Fanelli, and John P. A. Ioannidis. What does research reproducibility mean? *Science Translational Medicine*, 8(341):341ps12, 2016.

[GJSB00] J. Gosling, B. Joy, G. Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Java series. Addison-Wesley, 2000.

[Ins12] SAS Institute. *Base SAS 9.3 Procedures Guide: Statistical Procedures, Second Edition*. SAS Institute, 2012.

[Jr.51] Frank J. Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[KGB14] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *52nd Annual Meeting of the Association for Computational Linguistics*, 2014.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323, 1998.

[LGGS07] Christine Laine, Steven N. Goodman, Michael E. Griswold, and Harold C. Sox. Reproducible research: Moving toward research the public can really trust. *Annals of Internal Medicine*, 146(6):450–453, 2007.

[NGP09] Eilen Nordlie, Marc-Oliver Gewaltig, and Hans Ekkehard Plesser. Towards reproducible descriptions of neuronal network models. *PLOS Computational Biology*, 5(8):1–18, 08 2009.

[Pen11] Roger D. Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

[QD09] Ramon Quiza and J Davim. Computational modeling of machining systems. In *Intelligent machining: Modeling and optimization of the machining processes and systems*, pages 173–213, 01 2009.

[QD11] Ramon Quiza and J Davim. Computational methods and optimization. In *Machining of Hard Materials*, pages 177–208, 01 2011.

[Rob05] Gentleman Robert. Reproducible Research: A Bioinformatics Case Study. *Statistical Applications in Genetics and Molecular Biology*, 4(1):1–25, January 2005.

[Roj96] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, Berlin, Heidelberg, 1996.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[RW11] Nornadiah Mohd Razali and Yap Bee Wah. Power comparisons of shapiro-wilk , kolmogorov-smirnov , lilliefors and anderson-darling tests. *Journal of Statistical Modeling and Analytics*, page 21–33, 2011.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[Sch18] Steven E. Schoenherr. The digital revolution, July 2018. `https://web.archive.org/web/20081007132355/http://history.sandiego.edu/gen/recording/digital.html`.

[Sha84] Steven Shapin. Pump and circumstance: Robert Boyle's literary technology. *Social Studies of Science*, 14(4):481–520, 1984.

[Sto12] Victoria Stodden. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science & Engineering*, 14(4):13–17, 2012.

[Tea18] Eclipse Deeplearning4j Development Team. Deeplearning4j: Open-source distributed deep learning for the jvm, apache software foundation license 2.0, July 2018. `http://deeplearning4j.org`.

[vR18] Guido van Rossum. Python (programming language), July 2018. http://colenak.ptkpt.net/_lain.php?_lain=3721.

[WYW+17] Wenhui Wang, Nan Yang, Furu Wei, Baobao Chang, and Ming Zhou. Gated self-matching networks for reading comprehension and question answering. In *the Association for Computational Linguistics annual meeting (ACL)*, 2017.

[You77] IT Young. Proof without prejudice: use of the kolmogorov-smirnov test for the analysis of histograms from flow systems and other sources. *Journal of Histochemistry & Cytochemistry*, 25(7):935–941, 1977.