University of Miskolc
Institute of Information Science
Department of Information Technology

Single Element Feedforward Neural Network Inversion Methods in Python

Bachelor's Thesis

Author:
Lilla Juhász
FWIY50

Supervisor:
Bence Bogdándy

Miskolc, 2019.

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Tanszék **Szám:**

Szakdolgozat Feladat

Juhász Lilla (FWIY50) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Mesterséges Intelligencia, Adatbányászat, Python

A szakdolgozat címe: Single Element Feedforward Neural Network Inversion Methods in Python

A feladat részletezése:
A mesterséges intelligencia, gépi tanulás és adatbányászat mind nemrég lett újra felkapott témakörök egyike az informatika világában. Az egyik legfontosabb gépi tanulási modell a neurális hálók. A mesterséges neurális hálók, és a mesterséges neuronok elméleti háttere már a 20. század közepe óta létezik, azonban csak a közelmúltban váltak valós alternatívákká számítási költségük miatt. A gépi tanulás programozására és implementálására sok nyelv és könyvtár nyújt segítséget, azonban a legelterjedtebb ilyen technológia az akadémiai közösségben a python programozási nyelv és hozzá tartozó tudományos könyvtárak. A neurális háló invertálás egy nem triviális, összetett matematikai háttérrel rendelkező probléma, melynek gyakorlati megvalósítása és vizsgálata hiányzik a legtöbb ma elérhető technológiai megoldásból.

A hallgató feladata:

- Python 3.X, Anaconda Data Science platform megismerése, használatának elsajátítása.
- Pandas, Numpy, Scikit-Learn, Matplotlib és egyéb tudományos Python könyvtárak használata.
- Neurális hálózatok ismeretének kibővítése, mind matematikai, mind gyakorlati megvalósításban
- Neurális hálózat invertálási módszerek megismerése, feldolgozása és elemzése cikkekből, és egyéb tudományos forrásokból.
- Neurális hálózat egyelemes inverziójának megvalósítása, kiértékelése.

Témavezető(k): Bogdándy Bence, tanszéki mérnök
Konzulens(ek):
A feladat kiadásának ideje: 2018. szeptember 30.

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott .....; Neptun-kód: .....
a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős .....
szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy .....
című szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ...év ..... .hó ..... .nap

.....
Hallgató

1.

A szakdolgozat feladat módosítása

szükséges (módosítás külön lapon)

nem szükséges

....
dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás): konzulens (dátum, aláírás):

....
....
....

3. A szakdolgozat beadható:

....
dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt
..... program protokollt (listát, felhasználói leírást)
..... elektronikus adathordozót (részletezve)
.....
..... egyéb mellékletet (részletezve)
.....

tartalmaz.

....
dátum témavezető(k)

5.

A szakdolgozat bírálatra

bocsátható

nem bocsátható

A bíráló neve: .....

....
dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....
a bíráló javaslata: .....
a szakdolgozat végleges eredménye: .....

Miskolc, ...
a Záróvizsga Bizottság Elnöke

Contents

Chapter 1

Introduction

The interest in the application fields of machine learning has a rising tendency nowadays. By using artificial intelligence, intelligent systems can be established which are able to solve real-world problems. As the computing capacity increases, there are enough resources to train more complex machine learning models and intelligent systems, such as deep artificial neural networks. Artificial neural networks are machine learning tools that allow the creation of intelligent systems. Python and its scientific third-party libraries are appropriate platforms that can create intelligent systems to facilitate machine learning tasks. However there is a subfield in connection with artificial neural networks, that has not received much attention since the rise of machine learning. This subfield is the problem of neural network inversion. The goal of the thesis is to solve this inversion problem.

This work belongs in the realm of machine learning application research. Three main tasks are being tackled in this paper: data mining for extracting information from the data set, building and testing multi-layer perceptron models, and the single element inversion of the feedforward neural network.

Since preprocessing the data is a key factor to neural network performance, inversion can be implemented only after a sufficient neural network model has been established. Data mining is a tool for information extraction, processing, representation and summarization. It is designed to extract information from a dataset and transform it into a comprehensible structure for further use. The product of data mining is the preprocessed training set, whereat machine learning methods can be trained.

Machine learning is a field of artificial intelligence that gives computers the capability to learn from past data to help predict current or future states of a system. Learning relies on patterns and inferences, instead of explicit instructions. Machine learning utilizes statistical methods to process predefined datasets and to predict future output values for given data inputs. The used training methods originates from the application field of machine learning, more precisely from regression.

Regression is a supervised learning task that adapts a function to a dataset to predict the values of a desired target variable. Since regression can fit a function to the training data in order to find the best fitting parameters, and the artificial neural networks can behave as a universal function approximator if there are infinite

size of neurons available, neural networks can be used effectively to solve regression problems. Since there is no capacity for infinite number of neurons, the goal is to find that topology where the neural network is able to learn the regression function. Hence it is a long process to train a feedforward properly and find the best fitting regression function, but a necessary step to perform the inversion of the neural network.

The inverse function means the reverse of another function in mathematics. Neural network inversion means those procedures, that can approximate one or more points from the input set with respect of the examined output. Since feedforward neural networks aim on to capture system mapping from the given training data, the goal is to find those input values that will result the desired output for the given weights. The problem is that in case of a dataset, the output values are just rarely unique. Generally it can be determined that numerous inputs can generate the same output. Thus the neural networks are directly not invertible, but in case of a sufficient number of data with the assistance of a properly big topology, neural networks are able to estimate the inputs accurately for the examined output. In case of single element inversion methods, they are designed to find one point from the input space as opposed to evolutionary methods which are used to map multiple input points.

This work aims to implement and analyse the inversion of a single element feedforward neural network. The implementation that is being tackled in this paper suggests a solution for the inversion problem with the utilization of the Williams-Linder-Kindermann inversion. In the algorithm, the inversion problem is set up as an unconstrained optimization problem and solved by gradient descent, similarly to backpropagation.

Chapter 2

Related Works

2.1 Data Mining

Data mining [35, 17] is an interdisciplinary subfield of computer science and statistics, which can discover patterns and inferences in large datasets. These patterns contain valuable information about the data, from which different conclusions can be drawn for the dataset. Data mining is a tool for information extraction, processing, representation and summarization.

There are five stages that can be mentioned in connection with data processing:

- Selection: Features with the highest perceived information value need to be selected.
- Preprocessing: The dataset may include errors, missing values or inconsistent data that need to be filtered out.
- Transformation: Some features of the dataset require transformation, such as normalization, in order to be processable by data mining or machine learning models.
- Data mining: Data mining techniques need to be used that can discover the patterns.
- Evaluation: The patterns are known, therefore it can be seen, that not all of the patterns are needed for the prediction.

Data mining has applications in multiple fields, which not only encompasses scientific research but industrial applications such as banking, business, insurance, production engineering and so on. It can help companies to develop more effective strategies in a more optimal way.

2.1.1 The Dataset

Selecting the information that can be useful for the dataset needs a thorough examination. A well-defined dataset is large in scale and carries a lot of information whereof patterns can be predicted. Collecting enough relevant data is a long process, but there are public datasets that can be used for free.

Online News Popularity dataset [14, 1] - which is a prediction of Mashable news popularity - is publicly available at UCI Machine Learning repository. It provides

the popularity of online news, and neural networks can predict the future popularity of the news articles from given information before the release of news articles.

Figure 2.1: The Online News Popularity dataset of Mashable news is served by UCI Machine Learning Repository.

The UCI Machine Learning Repository [5] is a collection of datasets as a service to the machine learning community. These datasets can be used for free to various machine learning tasks.

Mashable is a digital media website that was founded in 2005. Online News Popularity dataset [28] contains information about 40.000 articles published between 2013 and 2015 on Mashable's website.

Online News Popularity dataset consists of 58 predictive features, 2 other attributes of accessory information and 1 goal field, which is the number of shares. The dataset was preprocessed before its publication, where general transformation steps were made. The dataset originally contained nominal features like the day of the publication or the type of the data channel, but they were transformed by one-hot encoding prior to publication. After the encoding, the predictive features are numeric values. Three types of keywords such as worst, average and best were captured by ranking all articles keyword average shares. Additionally, a bunch of natural language processing features were extracted such as closeness to top Latent Dirichlet Allocation (LDA) topics, title subjectivity, the rate of positive and negative words and title sentiment polarity. Sentiment polarity and subjectivity scores were also computed.

Online News Popularity dataset meets all the requirements for a regression problem and its preprocessing and transformation has already made before its publication. However the dataset is not prepared for applying data mining techniques, it needs more cleaning.

2.2 Machine Learning

Machine learning [22, 21, 8] is a field of artificial intelligence that gives computers the capability to learn from past data to help predict current or future states of a system. Learning relies on patterns and inferences, instead of explicit instructions. Machine learning utilizes statistical methods to process predefined datasets and to predict future output values for given data inputs.

Fields of application and popularity have been substantial in the recent years. Machine learning can be used during data processing, from the feature selection phase, to the application of data mining methods, as machine learning models are built from data mining techniques.

There are well-known techniques that can be used after analysing and optimizing the data. Two of the most widely adopted machine learning methods are supervised learning and unsupervised learning.

Supervised Learning

Supervised learning provides labeled training data, which are used during the prediction phase. Supervised learning's algorithm analyses the given dataset and processes the labeled data for mapping new examples. There are several types of supervised learning that can be divided into two main classes that are classification and regression. In **classification**, the samples can only be discrete types, but in **regression**, the desired output consists of one or more continuous variables and real numbers.

Unsupervised Learning

Unsupervised learning is another category of the learning problem, in which the training data consists of a set of input vectors without any corresponding target values. One of the goals in these problems may be to discover groups of similar attributes within the dataset, which is called **clustering**.

2.2.1 Biological Nervous Systems

The neural network [13] is a system with the functionalities of the human brain. It can recognize patterns and process real-world data with respect of these patterns.

Learning processes in artificial neural networks originate from the behavior of human brain and nervous system. In a biological brain, neurons communicate with electrical signals. Each neuron has several branches, called dendrites, which generate these signals that are sent through axons to other neurons. Neurons can communicate by their synapses, when the passed signal fires. Firing mechanism of the neuron contains two states. Namely, an off state in which the neuron does not transmit signals, and an on state of the signal, which helps the signal to pass from a neuron's synapses to another. With this sustained operation, signals are processed through a number of other neurons before reaching their destination.

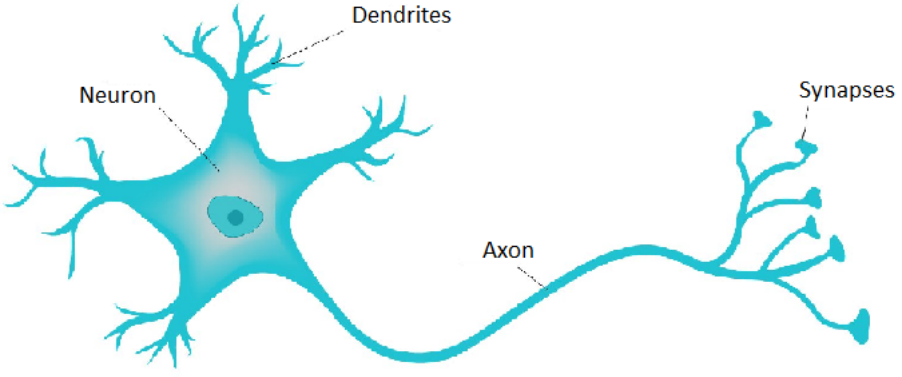


Figure 2.2: The biological neuron's structure

2.2.2 Artificial Neural Networks

The artificial neural network [25, 9] is a model of machine learning. For a simple explanation, it is a machine learning tool, which utilizes different learning functions to gather knowledge and solve various machine learning problems. An artificial neural network uses artificial neurons to model the complex process of the human brain. During the learning process, these neurons receive inputs, then change their internal state, which is called activation, to produce the desired outputs. This activation phase - similarly to the synapses - processes the given inputs from one neuron to another.

The artificial neural network itself is not an algorithm, but rather a framework for many different machine learning algorithms that can process complex data inputs. These learning algorithms learn from the experiences that are obtained from processing many examples. Each process yields an output, which, depending on the other outputs can determine which characteristics of the input are needed to construct the correct output. If there are a sufficient number of processed examples, the neural network can generate potential inputs and see if they produce the correct outputs. As the quantity of training data increases, so does the learning time and complexity of the neural network, however, training data also potentially increases the estimation accuracy.

A general classification task for artificial neural networks is the problem of image recognition. For example, it can be trained to identify whether or not the picture contains a rabbit. The system was manually programmed with showing example images, which was labeled as it contains a rabbit, or not. This process helps the system to identify rabbits on other pictures. The system can accomplish this without any task-specific rules, like any knowledge about how a rabbit looks like. Instead they automatically generate identifying characteristics from the patterns that they processed.

The fields of artificial neural networks' application are quite wide. As already mentioned, image/pattern recognition, but also self driving vehicle trajectory prediction, data mining, email spam filtering, signature verifications, medical diagnosis, cancer research, deep neural networks and several other areas use it nowadays. Some of those examples include Amazon which uses neural networks to power their recommendation engines, Microsoft for their translation services, Facebook for facial recognition and Google across many of its products including Google Translate and Gmail (spam filters). Essentially neural networks can be applied to a broad range of problems and can process many different types of input, including images, videos, files and databases among many others.

2.2.3 Neural Network Inversion

There is a subfield of machine learning that has not received much attention since the rise of deep learning. This subfield is the neural network inversion problem [20].

Neural network inversion procedures seek to find one or more input values that produce a desired output response for a fixed set of synaptic weights. Although the inversion of a neural network propounds various problems, because in case of a

given dataset several inputs can produce the same output. This means, that when an artificial neural network is made to learn a dataset, it will rarely be invertible. Since the artificial neural networks behave as a universal function approximator, they can learn any function with the infinite number of artificial neurons. Thus neural network inversion consists of the procedures which can approximate one or more points from the input set with respect of the examined output.

There are several methods which can perform neural network inversion which can be placed into three broad classes. Exhaustive search should be considered, when the dimensionality of the input and allowable range of each input variable is low. Single element inversion methods are used to find one inversion point per process. Multi element inversion procedures are able to find numerous inversion points simultaneously.

Inversion of the neural network can be useful in many areas which uses machine learning. For example, a single element inversion task is the problem of sonar performance under various environmental conditions [6]. A neural network is trained to generate SIR pixel values as a function of sonar and environmental parameters. Once trained, the inverted neural network can provide input parameters to generate desired SIR performance in a specified target region.

2.3 Python

Figure 2.3: Python and its library Scikit-Learn with the assistance of Anaconda are appropriate platforms to train artificial neural networks

Python [32, 2] is an interpreted, object-oriented, high-level programming language with dynamic semantics, used for general-purpose programming. It was created by Guido van Rossum and released in 1991.

Several goals have been defined for Python [31]:

- an easy and intuitive language;
- open source, so anyone can contribute to its development;
- code that is easily readable;
- suitable for everyday tasks, allowing for short development times.

Due to the realization of these goals, Python is said to be one of the most popular programming languages nowadays. It is easy to learn, has efficient high-level

data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Anaconda is the interpreter of Python, which contains more than 1500 built-in packages for Python. Anaconda provides a distribution for scientific computing, for example executing data mining or machine learning tasks. Anaconda has a package manager system called conda, that helps to facilitate the various scientific tasks.

In the artificial intelligence community, Python is one of the most used language, due to its effectivity. Numerous artificial intelligence fields' researchers develop in Python, with the assistance of the built-in libraries and several other sources found on the internet. The usefulness of Python for data science comes primarily from the large and active third-party packages:

- **SciPy** for containing a wide array of numerical tools such as numerical integration and interpolation;
- **Scikit-Learn** for providing a uniform toolkit for applying common machine learning algorithms to data;
- **NumPy** for providing efficient storage and computing multi-dimensional data arrays;
- **Pandas** for providing a DataFrame object along with a powerful set of methods to manipulate, filter, group, and transform data;
- **Matplotlib** for providing a useful interface for creating publication-quality plots and figures;

and many more tools that aimed scientific computing and other machine learning fields. Also the vast majority of the libraries used for data science have Python interfaces.
Besides the prebuilt libraries, choosing Python for artificial intelligence programming can make the development easier and faster with the specific indenting style and the dynamic typing system, which means less coding and more developing. Python is platform independent, its interpreter and extensive standard library are available in source or binary form for free.

At last one of the core benefits of Python is its flexibility. With options to choose from either scripting and OOP approach, Python is suitable for every purpose. Moreover, it also works as a perfect backend language and linking different data structures together is also suitable in Python.

Chapter 3

Theoretical Background

3.1 Feature Engineering

Data mining is a machine learning tool that uses statistical methods to discover patterns in large datasets. Data mining has various stages to process knowledge and all of the stages run different methods to collect the appropriate data.

Figure 3.1: Stages of data mining

The first and most important part is to select the dataset which will be processed. This means, that firstly the goal needs to be cleared up and then, depending on the problem, the information that seems to be useful for accomplishing the goal needs to be selected. In a dataset that processes real-world data, the selected features are always noisy and often contain faulty and inconsistent values. Hence data selection is a key factor to reach the most appropriate performance of the neural networks.

To decide the usefulness of a feature, the following quality requirements are assured:

- Validity: the degree to which the measures conform to defined business rules or constraints
- Accuracy: the degree of conformity of a measure to a standard or a true value
- Completeness: the degree to which all required measures are known
- Consistency: the degree to which a set of measures are equivalent in across systems
- Uniformity: the degree to which a set data measures are specified using the same units of measure in all systems

There are some other criteria for the qualities of good features. A useful feature vector value appears more than one times in a dataset, which enables the model to learn the relationships between the feature vector values and the associated labels. This means having many examples with the same value helps the model to see the feature in different settings to determine when it is a good predictor. Also, each feature should have a clear and obvious meaning.

3.1.1 Data Mining

Data preprocessing [27] is a data mining technique that involves transforming raw data into an understandable format for computers. The selected dataset often includes errors, missing values or noisy, inconsistent data which have to be filtered during the preprocessing stage.

Data preparation and filtering steps can be time consuming. Data preprocessing includes cleaning, integration and reduction. Sometimes data transformation takes place in the preprocessing too. The product of data preprocessing is the final training set.

The given data in large datasets often contains information that is not clear enough. There are several techniques for repairing these uncertain data.

For resolving missing values, the following steps can be taken:

- Simply remove the feature vector, which is only effective if the vector contains several missing attributes.
- Fill the missing values manually, which can be done with the mean value.

Both techniques can create inaccuracies, but it is important to deal the appropriate technique, because the incorrect feature vector may contain necessary information too.

Noise is a random error or variance in a measured variable. Noisy data needs to be smoothed to get the accurate prediction. Binning methods are local smoothing techniques, because they smooth the value with its neighbour's. Clustering can be used to detect outlier values by organizing them into similar groups. Data can be smoothed by fitting a function like regression to the data.

Data inconsistencies can be solved manually by external references, or with the use of knowledge engineering. Since the meaning of the features are given, it can be obvious to recognize an inconsistent value manually. Knowledge engineering uses technical, scientific and social aspects to deal with a feature value's inconsistency.

Data integration and reduction aims on the same goal to make the dataset clearer. During integration, an algorithm is looking through the dataset and looking for redundancies.

Data reduction is reducing the volume or the dimension of the dataset, without compromising the integrity of the original dataset. As the analysing can be time consuming, these data reduction techniques are really helpful in large datasets:
1. Dimension reduction: drop the irrelevant or weakly relevant values from the dataset
2. Data compression: use various encoding techniques to reduce the size of the dataset

Transformation and Feature Scaling

Some features of the dataset need to be transformed to get an appropriate form for data mining. Data transformation has a strong connection with **feature scaling** [36, 12], which is a statistical method used to standardize the range of the data. Feature scaling uses mathematical procedures to scale the range.

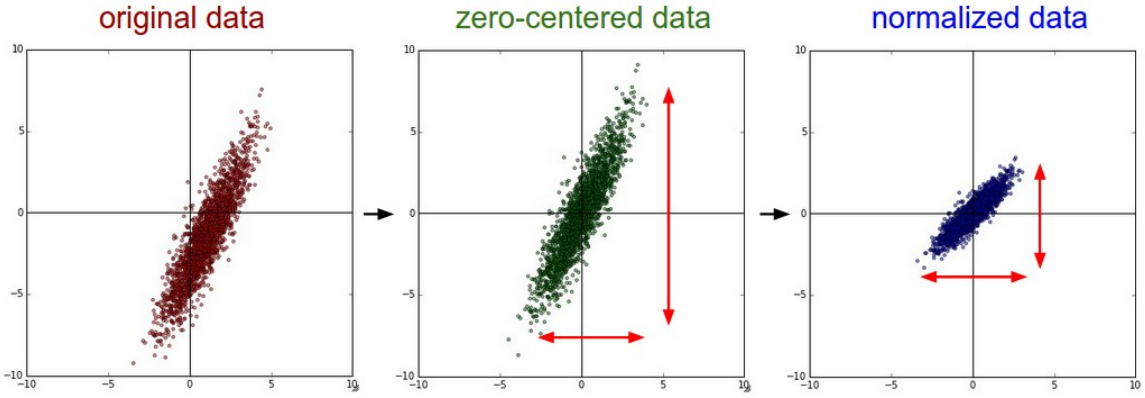


Figure 3.2: Data mining techniques can be used to convert data into understandable form.

Normalization is the process of scaling individual samples to have a unit normal. Min-max scaling is the simplest normalization method that rescales the features to a range of $[0, 1]$ or $[-1, 1]$. The selected range depends on the data. It can be written as the following Equation 3.1, where x is the original, and x' is the normalized value:

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{3.1}$$

Mean normalization in Equation 3.2 is similar to min-max scaling, where $\bar{x}$ is the mean of x:

$$x' = \frac{x - \bar{x}}{max(x) - min(x)} \tag{3.2}$$

Standardization is the process of transforming the values of the given dataset into standard normality. Feature standardization in Equation 3.3 gets the values of each feature to have zero-mean and unit-variance.

$$x' = \frac{x - \bar{x}}{\sigma} \tag{3.3}$$

where x is the original feature vector, $\bar{x}$ is the mean of that feature vector and σ is its standard deviation.

Encoding is the process of converting data into an acceptable form for information processing. It can be used effectively on categorical features, because they use a set of values.

Integer encoding converts the nominal values to numeric values. These numeric values are generally integers in increasing order, and they can cause inconsistencies with the given weights.

One-hot encoding offers a solution for the inconsistency problem. It creates a binary vector for each categorical feature in the dataset where the values appears as follows:
• For those values which apply to the example, set the vector values to 1.
• Set other values to 0.
The length of the binary vector is equal to the number values in the current feature.

Algorithms of Data Mining

The algorithms of data mining originates from a wider field called **data analytics**. Data analytics is the process of examining datasets in order to draw conclusions about the information they contain. It focuses on processing and performing statistical analysis on existing datasets. The dataset is processed by data mining techniques that machine learning models can use.

The product of data mining is the training set. To apply data mining algorithms, the original dataset has to be split into multiple sets. The first one is called **training set**, which is used by the machine learning algorithm to gather knowledge and increase accuracy. The other one is the **testing set**, which is used to provide a dataset to test function estimation accuracy of the learning algorithm on the training dataset.

After the split, the following data mining techniques [26] can be used on the training set to discover patterns:
1. Classification: This method helps to retrieve information by classifying data into different classes. It can be used to draw further conclusions.
2. Clustering: It can be used to identify data which are similar to each other. It is an effective way to understand the similarities and differences between values.
3. Regression: This method analyses the relationship between variables, to determine desired unknown values.
4. Association: It can discover hidden patterns in the dataset by identifying special

events, for example high correlations between values.
5. Outlier detection: It collects outlier values which do not match an expected pattern or expected behavior.
6. Sequential Patterns: This method helps to identify similar patterns for a certain period.
7. Prediction: This method can be used for predicting future values in a dataset with the known of the given data and past events.

3.2 Regression Analysis

Machine learning algorithms are able to use statistical techniques to predict the output values of a given dataset. Regression [16] is a statistical technique, that adapts a function to the dataset to predict the values of a desired target variable.

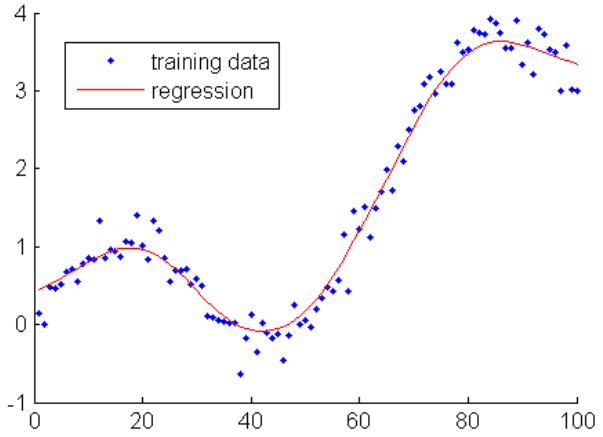


Figure 3.3: Regression is used to find the best fit of the training data by adapting a function to the dataset

Regression analysis is a form of predictive modelling technique which investigates the relationship between one dependent variable, known as target, and a series of other independent variables, called predictors. It also allows us to compare the effects of variables measured on different scales.
Regression models involve the following parameters and variables:

- The unknown parameters, denoted as $\beta \in \mathbb{R}^{\mathbb{N}}$, which may represent a scalar or a vector.
- The independent variables: $X \in \mathbb{R}$.
- The dependent variable: $y \in \mathbb{R}$.

A regression model relates Y to a function of X and β in Equation 3.4.

$$y \approx f(X, \beta) \tag{3.4}$$

Assume that the length of the vector of unknown parameters β is $k \in \mathbb{Z}$. In order to perform a regression analysis, the information about the dependent variable y must be provided:

- If $N \in \mathbb{Z}$ data points of the form (X, y) are observed, where $N < k$, most standard approaches of regression analysis cannot be performed as there are not enough data to recover β.
- If exactly $N = k$ data points are observed, and the function f is linear, the equations $y = f(X, \beta)$ can be solved. This reduces the problem to solving a set of N equations with N unknowns (the elements of β), which has a unique solution as long as the X are linearly independent. If f is non-linear, a solution may not exist, or many solutions may exist.
- The most common situation is where $N > k$ data points are observed. In this case, there is enough information in the data to estimate a unique value for β that best fits the data and the regression model when applied to the data can be viewed as an overdetermined system in β.

Regression in connection with machine learning [7] is a supervised learning task that can predict the output values of a desired target variable. In supervised learning tasks, the input and output values are both known and regression can be used when the data consist of continuous variables and real numbers.

Since regression fits a function on a dataset, and the artificial neural networks can behave as a universal function approximator if there are infinite size of hidden layers available, neural networks can be used effectively to solve regression problems. Since there is no capacity for infinite number of neurons, the goal is to find that topology where the neural network is able to learn the regression function.

3.2.1 Method of Least Squares

In regression analysis, there is a standard approach called the method of least squares which approximates the solution of overdetermined systems and finds a solution for unknown parameters β.

The method of least squares or least squares fitting method [34] is a mathematical procedure for finding the best-fitting curve to a given set of training data by minimizing the error. Error is the difference between the actual value of the dependent variable and the value predicted by the model.

Suppose that we have a series of $N \in \mathbb{Z}^+$ data points $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$, where x_i is an independent variable and y_i is a dependent one. Also suppose that $x_1 < x_2 < \ldots < x_N$, i.e. the points are distinct and are in increasing order depending on x. The fit of a model to a data point is measured by the error E_i in Equation 3.5:

$$E_i = y_i - f(x_i, \beta) \tag{3.5}$$

The method of least squares fitting aims to adjust the best-fitting values to the dataset. By performing least squares fitting algorithm, the required parameters $\alpha_1, \alpha_2, \ldots, \alpha_k$ of the function $f_{\alpha_1,\alpha_2,\ldots,\alpha_k} : [x_1, x_n] \to \mathbb{R}$ have been searched for, to minimize the sum S of squared errors in Equation 3.6:

$$S = \sum_{i=1}^{N} [f_{\alpha_1,\alpha_2,\ldots,\alpha_k}(x_i) - y_i]^2 \tag{3.6}$$

3.3 Perceptron

A perceptron [30] is a single layer feedforward neural network, which has only one input and output layer. The perceptron's units are the artificial neurons, that are conventionally called nodes.

In the input layer, each neuron has a weight connected to it, which represents the influence of that given neuron. The weights are used to decide the importance of the input values that are presented to the perceptron. Also the weights are used during the training of an artificial neural network as changing them in appropriate ways. If the predicted output is the same as the desired output, then the performance is considered satisfactory and no changes to the weights should be made. However, if the output does not match the desired output, then the weights need to be changed to reduce the error.

In order to avoid unnecessary iterations, it is important to adjust the weights in Equation 3.7 properly.

$$\Delta w = \eta * \hat{y} * x \tag{3.7}$$

where Δw stands for the current weight, $\eta \leq 1$ is the learning rate which is the size of the required steps, $\hat{y}$ represents the desired output and x is the input data.

To determine the value of a node, all the inputs would be multiplied by their respective weights, and then summed. This weighted sum stands for **dot product** in this context: $z = w_1x_1 + w_2x_2 + \cdots + w_mx_m = \sum_{j=1}^{m} w_jx_j$

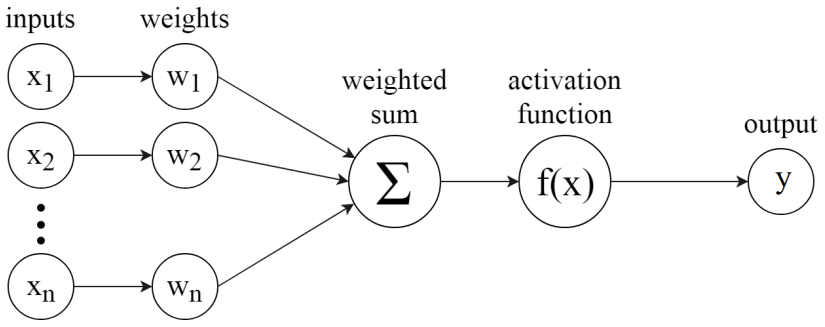


Figure 3.4: The appropriate weights are applied to the inputs and the resulting weighted sum passed to an activation function that produces the output.

The output y in Equation 3.8 is determined by whether the weighted sum $\sum_j(w_jx_j)$ is less than or greater than some threshold value θ, which is the parameter of the neuron. If that value is above a given threshold, it "fires", which means that the neuron gets an activated value.

$$y = \begin{cases} 0, & \text{for } \sum_j(w_jx_j) < \theta \\ 1, & \text{for } \sum_j(w_jx_j) \geq \theta \end{cases} \tag{3.8}$$

3.3.1 Feedforward Neural Networks

The feedforward neural network [15] is a type of neural networks, which aims to create a mapping from a properly trained input dataset to an estimated output. This type of neural network is called feedforward as there are no feedback connections in which outputs of the neuron are connected to itself. A feedforward neural network is able to model complex non-linear functions.

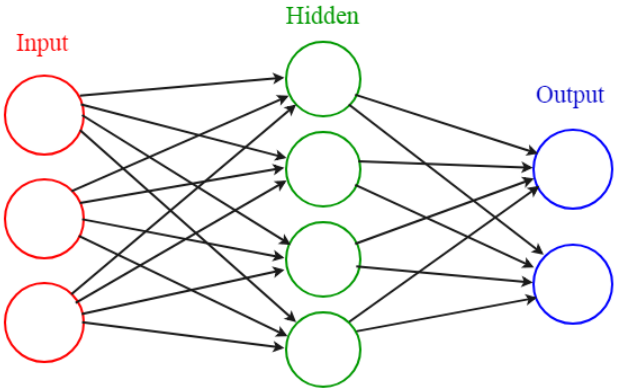


Figure 3.5: A feedforward neural network, that does not contain any feedback connections between the nodes.

A typical feedforward neural network consists of input and output layers. There is an intermediate part between inputs and outputs, called hidden layers. The **input layer** is a set of input neurons, where each neuron represents a feature in our dataset. The **output** of any feedforward network is the sum of the inputs multiplied by the weights. The **hidden layer** contains units which can transform the inputs into a mapping that the output layer can utilize. The relationship between the input and hidden layer is determined by the weights of the network.

The output of a trained feedforward neural network can be characterized by the Equation 3.9

$$y_k = f_k(i, w) \tag{3.9}$$

where y_k means the kth neural network output, (i, w) is a vector of the weights and $f_k(\cdot)$ describes the mapping from the input to the kth output, where $f_k(\cdot)$ also contains the structure of the feedforward perceptron. The neural network can be trained if the input and the output are fixed and the weights are adjusted. When a single scalar output can be found, y_k can be replaced by y and $f_k(\cdot)$ by $f(\cdot)$.

3.3.2 Multi-Layer Perceptron

The multi-layer perceptron is a multi-layered feedforward neural network algorithm that learns a function $f(\cdot) : \mathbb{R}^m \mapsto \mathbb{R}^n$ by training on a dataset, where m is the number of dimensions for the input and n is the number of dimensions for the output. Given a set of features $X = x_1, x_2, \ldots, x_m$ and a target y, it can be a non-linear function approximator for either classification or regression.

The difference between single- and multi-layered neural networks is that the multi-layered perceptron has one or more hidden layers besides the input and output layer. Except for the input nodes, each node is a neuron that uses a linear or non-linear activation function.

The process of making a multi-layer neural network is simple with the use of the perceptrons.
1. Assume that there are $m \in \mathbb{Z}$ features in input X, so $m \in \mathbb{Z}$ weights are needed to perform a dot product.
2. With n hidden neurons in the hidden layer, $n \in \mathbb{Z}$ sets of weights $(w_1, w_2, \ldots, w_n)$ are needed for performing dot products.
3. With one hidden layer, n dot products can be performed to get the hidden output $h : (h_1, h_2, \ldots, h_n)$.
4. Then the hidden output that also has n features can be used as input data to perform dot product with a set of n weights and get the final output y.

In a multi-layer perceptron, each neuron in one layer is connected with a weight to another neuron in the next layer. Each of these neurons stores a value, which is in general a sum of the weighted neurons which comes from previous layers. There is a special unit, called **bias**, that are not influenced by any values in the previous layer, so they do not have any incoming connections. However they have outgoing connections and they can contribute to the output of the artificial neural network. Bias units stores a constant value, which helps the model to fit the best for the given data. Hence the definition of dot product expands like the Equation 3.10:

$$z = \sum_{j=1}^{m} w_j x_j + bias \tag{3.10}$$

Neural networks are designed to learn from datasets by using iterative methods. Estimation value error is calculated from the desired and the predicted values to modify the weights of the connections between neurons. A multi-layer perceptron utilizes a supervised learning technique called **backpropagation** for training.

Backpropagation

The main goal of backpropagation [11] is to update all of the weights in the neural network, in such way that the predicted output to be closer to the target output while minimizing the error of each output neuron and the network.

The algorithm consists of two phases: the forward phase where the activations are propagated from the input to the output layer, and the backward phase, where the error between the actual and the desired value in the output layer is propagated backwards to modify the weights values.

Assume that N is a neural network with e connections, m inputs, and n outputs, where $N, e, m, n \in \mathbb{Z}$. Below, $x_1, x_2, \ldots$ will denote vectors in $\mathbb{R}^m$, $y_1, y_2, \ldots$ vectors in $\mathbb{R}^n$, and $w_0, w_1, w_2, \ldots$ vectors in $\mathbb{R}^e$. These are the inputs, outputs and weights, respectively. The neural network corresponds to a function $y = f_N(w, x)$, which gives a weight w and maps an input x to an output y. The optimization takes

a sequence of training examples $(x_1, y_1), \ldots, (x_p, y_p)$ and produces a sequence of weights $w_0, w_1, \ldots, w_p$ starting from an initial weight w_0 that is usually chosen at random. Then first compute w_i using only (x_i, y_i, w_{i-1}) for $i = 1, \ldots, p$. The output of the algorithm is w_p, giving us a new function $x \mapsto f_N(w_p, x)$. The computation is the same in each step, so only the case $i = 1$ is described. Calculating w_1 from (x_1, y_1, w_0) is done by considering a variable weight w and applying gradient descent to the function $w \mapsto E(f_N(w, x_1), y_1)$ to find a local minimum, starting at $w = w_0$, where E is the error function. This makes w_1 the minimizing weight found by gradient descent.

The function that is used to compute this error is known as **loss function**. For backpropagation, the loss function calculates the difference between the network output and its expected output, after a training example has propagated through the network.

Different loss functions will give different errors for the same prediction, and thus have a considerable effect on the performance of the model. In the training of multi-layer perceptrons, L2 loss function L has been used in Equation 3.11, which is the square of the L2 norm of the difference between actual value y and predicted value $\hat{y}$.

$$L = \sum_{i=1}^{n} (y_i - \hat{y})^2 \tag{3.11}$$

Gradient descent

One generally used method used for backpropagation in the calculation of the weights is gradient descent [10]. It is an optimization method, which aims to minimize a given function to its local minimum by iteratively updating the weights of the model. The input is defined with an initial value and the algorithm calculates the gradient i.e. the partial derivative of the loss curve at this starting point. Gradient descent can be used to solve a system of linear and non-linear equations too, but only the linear one will be explicated in Equation 3.12:

$$\hat{y} = wx + b \tag{3.12}$$

where x is the input, $\hat{y}$ is the predicted output, w defines weights and b stands for bias.

In the operation of gradient descent, cost function $f(w, b)$ in Equation 3.13 utilizes the value of the loss function to define how to update our weights to make the model more accurate.

$$f(w, b) \quad = \quad \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y})^2 \quad = \quad \frac{1}{n} \sum_{i=1}^{n} (y_i - (wx_i + b))^2 \tag{3.13}$$

From the cost function, the partial derivatives of the cost function are calculated with respect to each parameter and results are stored in a gradient in Equation 3.14.

$$f'(w, b) = \begin{bmatrix} \frac{\partial f}{\partial w} \\ \frac{\partial f}{\partial b} \end{bmatrix} = \begin{bmatrix} \frac{1}{n} \sum -2x_i(y_i - \hat{y}) \\ \frac{1}{n} \sum -2(y_i - \hat{y}) \end{bmatrix} \tag{3.14}$$

Thus the process of the gradient descent algorithm is the following:
1. Initialize the weights with an initialize value and calculate the cost function.
2. Calculate the gradient, i.e. change the weights a bit from their original initialized value to the direction in which the cost function is minimized.
3. Adjust the weights with the gradients to reach the optimal values where $f(w, b)$ is minimized.
4. Use the new weights for prediction and to calculate the new cost function.
5. Repeat steps 2, 3 until the adjustments to weights does not significantly reduce $f(w, b)$.

In Figure 3.6 the components of gradient descent are the following:

- Learning rate: size of steps took in any direction
- Gradients: the direction of the steps, i.e. the slope
- Cost function: tells the current height, which is the sum of squared errors

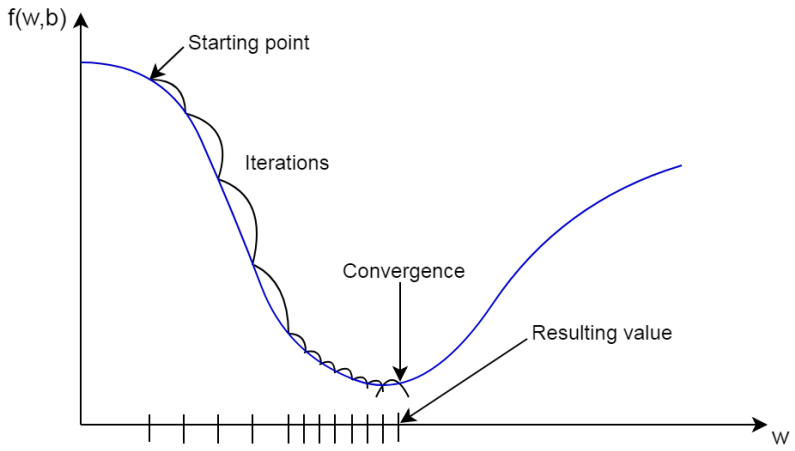


Figure 3.6: The gradient descent is an optimization method used by backpropagation

In backpropagation, the calculation of the gradient passes the network backwards, so the values of the gradient from one layer are reused in the computation of the gradient for the previous layer.

3.4 Training a MLP Model

The task that is being tackled in this thesis is to make the inversion of a single element feedforward neural network. To eventuate this successfully, the appropriate dataset is given and already preprocessed. Hence the main task is now to train a neural network to predict new outputs for the testing set and minimize the loss between the given and the desired outputs from the training set. The method is given by backpropagation, but there are a lot of components of the neural network model that plays a very important role in the effective training to produce accurate results. These components are the activation function, optimization algorithm, the

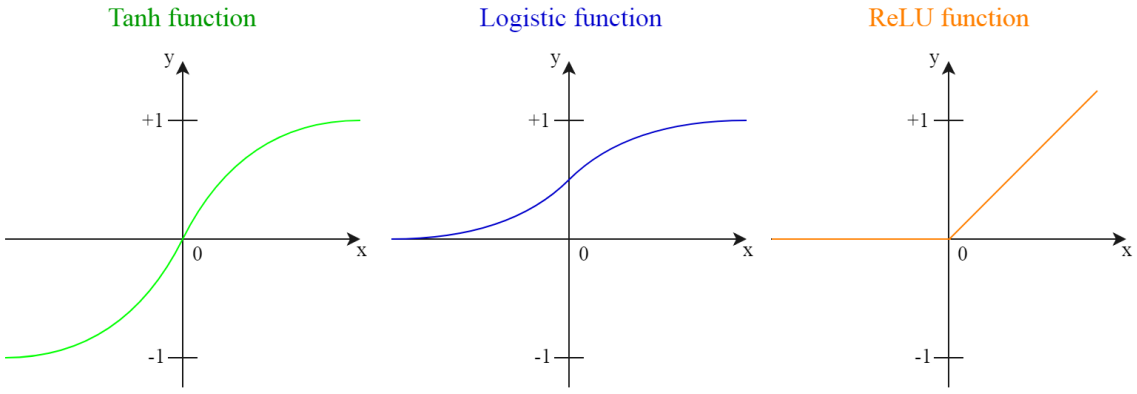
appropriate topology i.e. the size of the layers, the learning rate, and so on. Different tasks require a different set of functions to give the most optimum results. The used methods and functions are described in the following subsections.

3.4.1 Activation Functions

In connection with artificial neural networks, the activation function is a transitional state of the neurons between other layers. It is a mapping of the previous layers and it maps the resulting value into the desired range, which is usually between -1 and 1. The output of the activation function is then used as input for the next layer, until a desired solution is found. There are several activation functions, and each of them utilizes different methods for mapping.

As the multi-layer perceptron approximates non-linear functions, it will be the most accurate by using non-linear activation functions as well [24]. These non-linear activation functions are known for having more than one degrees and have a curve in their graph. As non-linear functions can generate non-linear mappings from inputs to outputs, they can be applied on complex datasets.

Figure 3.7: The graphs of the most popular non-linear activation functions



The two common non-linear activation functions are both sigmoids, as they approximate with a curve instead of a straight line. These are the **hyperbolic tangent** and the **logistic** functions, described by Equation 3.15

$$f(x) = tanh(x) \qquad \text{and} \qquad f(x) = \frac{1}{1 + e^{-x}} \tag{3.15}$$

The hyperbolic tangent ranges from -1 to 1, while the logistic function is similar in shape but ranges from 0 to 1.

The **ReLU** function is another type of non-linear functions, which stands for rectified linear unit. It is called half-rectified, because no negative values are allowed. This operation affects the resulting graph (Figure 3.7), as the negative values are shown as zeros in Equation 3.16.

$$f(x) = max(0, X) \tag{3.16}$$

3.4.2 Optimization Methods

Artificial neural networks have different phases in the process of their operation. The learning process in a neural network uses different training methods with different characteristics and performance. These training methods use optimization algorithms to update weights and biases i.e. the internal parameters of a model to reduce the error. [33, 29]

Stochastic Gradient Descent

Stochastic gradient descent is a stochastic approximation of gradient descent optimization, used effectively in large-scaled datasets. In contrast to gradient descent, the stochastic method can approximate the true gradient of the cost function because it updates the parameters for each training example, one by one.

To demonstrate assume that η is the learning rate, L stands for the loss function and $\nabla_w L = \frac{\partial L}{\partial w}$ is the gradient. Now the updating equation of the weights w in each iteration is:

$$w_{t+1} = w_t - \eta \nabla_w L \tag{3.17}$$

The process of the stochastic gradient descent algorithm is the following:
1. Choose one sample from the dataset (this is what makes it stochastic gradient descent).
2. Calculate all the partial derivatives of loss with respect to weights or biases.
3. Use the update Equation 3.17 to update each weight and bias.
4. Go back to step 1.

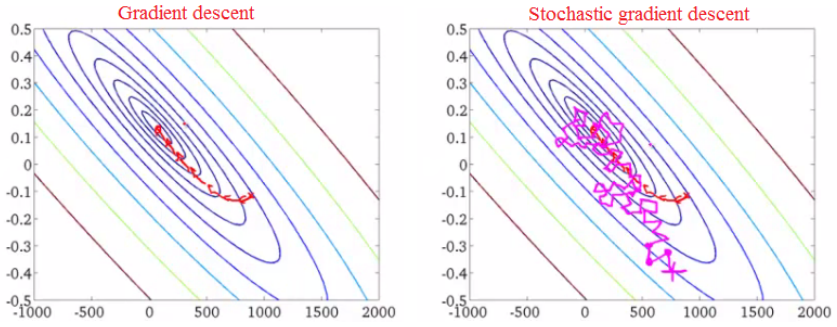


Figure 3.8: The difference between the process of GD and SGD methods

Adaptive Moment Estimation

Another method is adaptive moment estimation, called Adam. This brand-new optimization method is designed to solve deep learning problems. Adam is a transition between adaptive methods and momentum-based methods. In the algorithm, running averages of both the gradients and the second moments of the gradients are used, which means Adam not only stores the exponentially decaying average of past squared gradients v_t, it also keeps the decaying average of past gradients m_t, similarly to momentum-based methods. The algorithm computes the decaying averages

of past m_t and past squared v_t gradients respectively as follows in Equation 3.18:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) L_t \quad \text{and} \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) L_t^2 \tag{3.18}$$

where m_t and v_t are the estimates of the first and second moments, β_1 and β_2 are the decay for gradients and second moments of gradients, and L_t is the loss function. The first and second moment estimates are in Equation 3.19:

$$\hat{m}_t = \frac{m_t}{a - \beta_1^t} \quad \text{and} \quad \hat{v}_t = \frac{v_t}{a - \beta_2^t} \tag{3.19}$$

Then these estimates are used to update the parameters w with a simple scalar ϵ to prevent division by 0 in Equation 3.20:

$$w_{t+1} = w_t - \frac{\mu}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.20}$$

Limited-memory BFGS

Limited-memory BFGS - against the previous ones - is a quasi-Newton method for weight optimization, that uses Hessian matrices to approximate. L-BFGS originates from the Broyden–Fletcher–Goldfarb–Shanno algorithm with limited amount of computer memory. L-BFGS aims on parameter estimation, and the target problem is to minimize $f(x)$ over unconstrained values of the real-vector x where f is a differentiable scalar function. In this method, the Hessian matrix of second derivatives is not computed, just approximated by using gradient evaluation updates.

From an initial guess x_0 and an approximate Hessian matrix B_0, the following steps are repeated as x_k converges to the solution:

1. Obtain a direction p_k by solving $B_k p_k = -\nabla f(x_k)$.
2. Perform a one-dimensional optimization to find an acceptable stepsize α_k in p_k.
3. Set $s_k = \alpha_k p_k$ and update $x_{k+1} = x_k + s_k$.
4. Set $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$.
5. The solution is $B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k}$.

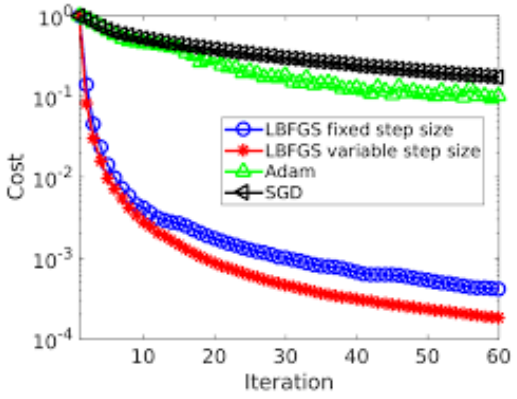


Figure 3.9: The optimization of SGD, Adam and L-BFGS with respect of cost and iteration

3.5 Inversion

The inverse of a function means the reverse of another function in mathematics. For a given $f : X \mapsto Y$, the inverse function of f is that $g : Y \mapsto X$, where $f(x) = y$ and $g(y) = x$. Thus $g = f^{-1}$.

Not all of the functions are invertible. The inverse of a function exists, if that function is **injective**, so there are another function with which they preserve distinctness. This means a one-to-one mapping. However injectivity only provides that every x in X maps to a y in Y. But also it is not a criteria that all of the elements in Y belongs to an x, so here some y may exist which are not a mapping of any x.

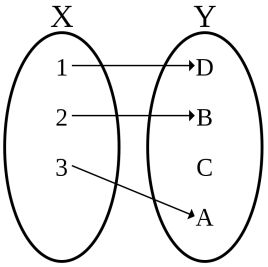


Figure 3.10: Injective function means a one-to-one mapping from X to Y.

In case of **surjection**, for every y in Y there is a mapping from an x in X. Also, x do not has to be unique, so more x can map to the same y.

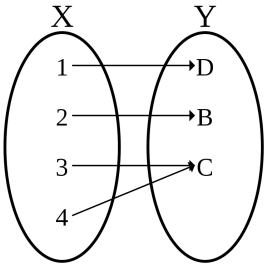


Figure 3.11: In case of surjection, more x can map to the same y.

If a function is **bijective**, it is injective and surjective at the same time, so every x corresponds to one, and only one y, where $X, Y \in \mathbb{Z}^+$ and the number of elements are equal in X and Y.

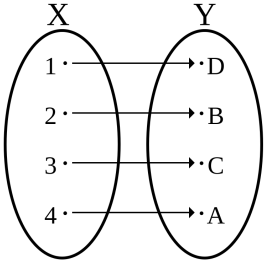


Figure 3.12: If a function is bijective, every x corresponds to one, and only one y.

Bijective functions are mutually unequivocal, so they can be mapped back and forth. When it is fulfilled, it can be said that the given $f(x) = y$ is mappable so that $g(y) = x$. Simply a function f has an inverse function, if for every y in Y that there must be one, and only one x in X so that $f(x) = y$.

Since artificial neural networks can behave as a universal function approximator, the inversion of a neural network is meant to be a process, which consists of clamping the weights and the neural network output while adjusting the input in the neural network until an equality or a best possible fit occurs for one or more values of the input.

Feedforward neural networks aim on to capture system mapping from the given training data. The goal is to find the input values that will result the desired output for the given weights. The problem is that in case of a dataset, bijection is not guaranteed, since the output values are just rarely unique. Generally it can be determined that numerous inputs can generate the same output (seen at surjection in 3.5). So the neural networks are literally not invertible, but in case of a sufficient number of data with the assistance of a properly big topology, neural networks are able to estimate the inputs quite accurately for the examined output.

The process of finding the inversion $f^{-1}(x)$ of a function $f(x)$ is the following:
1. Replace $f(x)$ with y.
2. Replace every x with a y and replace every y with an x.
3. Solve the equation from step 2 for y.
4. Replace y with $f^{-1}(x)$.

3.5.1 Single Element Inversion Methods

In case of single element inversion, one single point can be estimated in every iteration from the input set in respect of the given output. It's accuracy is depending on the initialization and it will result a nearer outcome in further processes. However not the only important task is to find the best topology, but also to find a properly training algorithm, because it notes the previously founded points and reuses them. The process of finding the best estimator is time consuming. As the quantity of parameter combinations increases during the training phase, so does the time that the training takes. However, the number of parameter combinations also potentially increase the inversion accuracy.

To solve the inversion of an unconstrained optimization problem, the inversion method needs to solve the optimization itself in its training phase. After the dataset is properly trained, the inversion problem is the following:

Given some neural network function $f : X \mapsto Y$, for some $y \in Y$, the appropriate $x \in X$ needs to be found, such that $f(x) = y$. Or more generally, if $L : Y \mapsto \mathbb{R}$ is the loss function defined over the network output, an input x have to be found that minimizes $L(f(x))$.

3.5.2 Williams-Linder-Kindermann Inversion

The WLK inversion was named after R. L. Williams, A. Linder and J. Kindermann [20], who firstly introduced the single element search method for inversion of real valued neural network. In this algorithm, the inversion problem is set up as an unconstrained optimization problem and solved by gradient descent, similarly to backpropagation.

The method of WLK inversion involves two main steps:
1. computing the deltas for every value
2. updating the weights manually

During the training, the neural network is trained to learn a mapping from input to output. The proper set can be find by minimizing the loss. Thus the neural network learns a functional relationship between the inputs and the outputs. Now all the weights are fixed.

Assume that the initial input vector i_0 is given. Now the recursive equation of the training phase is the following:

$$i_k^{t+1} = i_k^t - \eta \frac{\partial E}{\partial i_k^t} \tag{3.21}$$

t - the index of the iteration,
i_k^t - the kth component of the i^t vector,
η - the learning rate

Because of the general feedforward topology, the iteration for inversion in Equation 3.21 can be solved by the derivative of Equation 3.22

$$\frac{\partial E}{\partial i_k} = \delta k \quad k \in I \tag{3.22}$$

for every δk in Equation 3.23:

$$\delta j = \begin{cases} \varphi_j'(o_j)(o_j - t_j) :, & j \in O \\ \varphi_j'(o_j) \sum_{m \in H,O} \delta_j w_{jm} :, & j \in I, H \end{cases} \tag{3.23}$$

I, O, H - the set of input, output and hidden neurons,
w_{jm} - the weight value from neuron j to neuron m,
φ_j' - the derivative of the jth neuron squashing function,
o_j - the activation of the jth neuron,
t_j - the desired output of the jth neuron

The derivatives of the neurons need to be solved by backward order from the output to the input.

Chapter 4

Implementation

4.1 Problem Statement

This work aims to implement and analyse the inversion of a single element feedforward neural network. However the implementation of the inversion is just the last step in the process, because the dataset needs data cleaning to make the predictable model.

This work belongs in the realm of machine learning application research. Three main tasks are being tackled in this paper: data mining for extracting information from the dataset, building and testing multi-layer perceptron models, and the single element inversion of the feedforward neural network.

The first task is to choose a dataset which contains predictable data. The selected dataset is called the Online News Popularity Dataset by Mashable news, that was served by UCI's Machine Learning Repository [5].

The dataset is preprocessed for data mining by Pandas. Since being a publicly available dataset, its preprocessing and transformation has already done before its publication, but the dataset needs more cleaning with the assistance of Scikit-Learn. The dataset contains numerous outlying values that should be handled. Furthermore the dataset has a wide range of values, which necessitates a standard scaling. Then the dataset can be split into training and testing sets by Scikit-Learn.

The training phase consists of the application of machine learning techniques [19, 18]. Different multi-layer perceptron models are fitted on the training set with various hyperparameters. These attributes have iterated over different hidden layer sizes, activation functions, optimization algorithms, learning rate sizes and alpha values. Then Scikit-Learn trains the dataset and makes predictions to the testing set. The training is time consuming, since the used dataset is large and its length depends on the amount of attributes of the multi-layer perceptron models. At the end of each process, the testing set's output and the predicted output are compared and the difference between them is calculated. If all of the iterations are over, the best estimator's parameters are shown with the score, and the tested target values and the predicted values are plotted by Matplotlib.

To perform the Williams-Linder-Kindermann inversion, the equations which are mentioned in subsection 3.5.2 have to be implemented. The inversion is defined as a

Python function to be callable in the main program. In the inversion function, the network is initialized with a random input vector. Output is calculated and compared with the given output. Now the error can be calculated and backpropagated to minimize the loss function and the input vector has updated too. This iterative process continues until the error is less than the minimum set value. The return value is the guessed input value.

At the end, the given testing set's values and the guessed input values are written to a .txt file with the accuracy percent of the inversion. A summarization about the accuracy is represented in the .txt file too.

4.1.1 Used Third-Party Libraries

Python can be used effectively with the assistance of its third-party libraries [4], which provide numerous effective and easy-to-use models in scientific research.

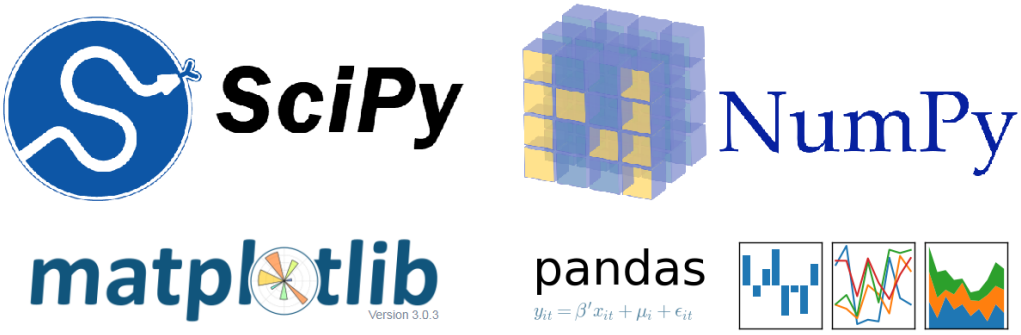


Figure 4.1: The used scientific third-party packages for Python

SciPy

SciPy is a Python-based library of open-source software for mathematics, science, and engineering. It contains modules for optimization, linear algebra, integration, interpolation, special functions and other tasks common in science and engineering. SciPy builds upon a small core of scientific packages, and the functions are built to Python's scientific computing capabilities.

SciPy has various sub-packages included:

- *integrate* is a numerical integration routine.
- *interpolate* is an interpolation tool.
- *io* stands for data input and output.
- *lib* is a Python wrapper to external libraries.
- *linalg* is for linear algebra routines.
- *ndimage* produces various functions for multi-dimensional image processing.
- *optimize* is an optimization algorithm including linear programming.
- *stats* are the statistical functions.

Pandas is an open source library that is providing high-performance, easy-to-use data structures and data analysis tools for Python. The built-in tasks in Pandas originates from classic R tasks. Pandas has a main object called DataFrame, that is for data manipulation by using a set of labeled array data structures. Pandas has tools for reading tabular data and writing data between in-memory data structures and different file formats. The data can be handled integratedly and the datasets are transformable. Some of the examples of this transformation are column insertion and deleting, dataset merging and joining, or the hierarchical axis indexing in high-dimensional data models.

NumPy is a library for Python focused on multi-dimensional arrays and matrices. It provides high-level mathematical functions to operate on these arrays. The built-in methods originates from the Matlab programming language. In NumPy, the n-dimensional array is called *ndarray*, where all the elements of a single array must contain the same type.

The most important attributes of an *ndarray* object are:
- *ndarray.ndim* produces the number of axes of the array
- *ndarray.shape* produces the dimensions of the array
- *ndarray.size* is the total number of elements of the array
- *ndarray.dtype* is an object describing the type of the elements in the array
- *ndarray.itemsize* produces the size in bytes of each element of the array
- *ndarray.data* is a buffer, which contains the actual elements of the array

Matplotlib is an open source Python library which produces publication-quality plots and figures. It ships with several add-on toolkits, including 3d plotting and helpers for axes. There are several common approaches of plotting with Matplotlib. The most popular is *pyplot*, which is a collection of command style functions where each pyplot function makes some change to a figure. Also it is mainly intended for interactive plots and simple cases of programmatic plot generation.

Scikit-Learn

The most important package that is used during the implementation is Scikit-Learn [23, 3]. It is a separately-developed and distributed third-party extension to SciPy, which integrates classic machine learning algorithms into the scientific Python packages.

Scikit-Learn can be used to solve multi-layer neural network learning problems. Among many others, it features various classification, regression and clustering algorithms. Scikit-Learn contains all the various functions which can be used during the training of the neural network. The training lasts from the processing of the dataset, via the iterations, to the inversion itself.

StandardScaler is a subclass of Scikit-Learn's preprocessing class that standardizes features by removing the mean and scaling to unit variance. It has three parameters that are boolean values and the default of all is True:

- *copy* means if the original dataset will be replaced with the scaled one or not
- *with_mean* means if the scaler centers the data before scaling or not
- *with_std* means if the data is scaled to unit variance or not

The standard score of a sample x is calculated as: $z = \frac{(x-u)}{s}$ where u is the mean of the training samples or zero if $with\_mean = False$, and s is the standard deviation of the training samples or one if $with\_std = False$.

StandardScaler has some methods:

- $fit(X[, y])$ computes the mean and the standard deviation to be used for later scaling
- $fit\_transform(X[, y])$ fits to data, then transforms it
- $get\_params([deep])$ gets parameters for this estimator
- $inverse\_transform(X[, copy])$ scales back the data to the original representation
- $partial\_fit(X[, y])$ is the online computation of mean and standard deviation on X for later scaling
- $set\_params(**params)$ sets the parameters of this estimator
- $transform(X[, y, copy])$ performs standardization by centering and scaling

Train_test_split is a subclass of Scikit-Learn's selection class, which splits arrays or matrices into random train and test subsets. The randomization is important in ordered datasets. Allowed inputs are lists, NumPy arrays, SciPy-sparse matrices or Pandas DataFrames. Train_test_split returns with four lists that contains the training and testing sets of features an targets.

The parameters are the following:

- $test\_size$: If $float$, it should be between 0.0 and 1.0 and represents the proportion of the dataset to include in the test split. If int, it represents the absolute number of test samples. If $None$, the value is set to the complement of the train size.

- $train\_size$: If $float$, it should be between 0.0 and 1.0 and represents the proportion of the dataset to include in the train split. If int, it represents the absolute number of train samples. If $None$, the value is automatically set to the complement of the test size.

- $random\_state$: If int, it is the seed used by the random number generator. If $RandomState$ instance, it is the random number generator. If $None$, the random number generator is the $RandomState$ instance used by $np.random$.

- $shuffle$: Whether or not to shuffle the data before splitting.

- $stratify$: It shows if the data is split in a stratified fashion, using this as the class labels.

MLPRegressor is a class of Scikit-Learn, which implements a multi-layer perceptron to solve regression problems. It uses the square error as the loss function. MLPRegressor trains iteratively because the partial derivatives of the loss function are computed in each step to update the parameters of the layers. MLPRegressor also supports multi-output regression, in which a sample can have more than one target outputs.

MLPRegressor has several parameters, but only a few of them will be listed, which were used in the implementation:

- *hidden_layer_sizes* is a tuple where the ith element represents the number of neurons in the ith hidden layer
- *activation* is the activation function for the hidden layer.
  - 'logistic' provides the logistic sigmoid function, returns $f(x) = \frac{1}{1+e^{-x}}$
  - 'tanh' provides the hyperbolic tangent function, returns $f(x) = tanh(x)$
  - 'relu' provides the rectified linear unit function, returns $f(x) = max(0, x)$
- *solver* is the optimization algorithm used in weight optimization
  - 'sgd' refers to stochastic gradient descent
  - 'adam' refers to a transition between adaptive methods and momentum-based methods
  - 'lbfgs' is an optimizer in the family of quasi-Newton methods
- *alpha* is the L2 regularization parameter's value
- *learning_rate_init* is the initial learning rate that is used. It controls the step-size in updating the weights. Only used when the solver is *sgd* or *adam*.
- *learning_rate* means the learning rate schedule for weight updates
  - 'constant' is a constant learning rate given by *learning_rate_init*
  - 'invscaling' gradually decreases the learning rate at each time step using an inverse scaling exponent
  - 'adaptive' keeps the learning rate constant as long as training loss keeps decreasing

MLPRegressor contains built-in methods to perform regression:
- $fit(X, y)$ fits the model to data matrix X and target(s) y
- $get\_params([deep])$ gets parameters for this estimator
- $predict(X)$ predicts an output based on previous training and a given testing dataset
- $score(X, y[, sample\_weight])$ returns with the coefficient of the determinated prediction
- $set\_params(**params)$ sets the parameters of the estimator

GridSearchCV is an exhaustive searching class of Scikit-Learn over specified parameter values for an estimator. It can be used in the tuning of the hyper-parameters, which are those parameters that are not directly learnt within estimators. The parameters of the estimator used to apply the methods of GridSearchCV are optimized by cross-validated grid-search over a parameter grid.

The parameters of GridSearchCV are quite wide, but only a few of them are used during the implementation:

- *estimator* implements the estimator interface.
- *param_grid* is a dictionary with the names of the used parameters as keys and lists of parameter settings to try as values.
- *cv* is an integer that determines the cross-validation splitting strategy.
- *n_jobs* stands for the number of jobs to run in parallel. -1 means that the search uses all of the processors.

From GridSearchCV's attributes, *best_params_* is used, which is a dictionary about the parameter setting that gave the best results on the hold out data.

GridSearchCV contains built-in methods for prediction:
- $fit(X[, y, groups])$ method fits with the adjusted parameter grid on the given dataset.
- $get\_params([deep])$ gets parameters for this estimator.
- $predict(X)$, $predict\_log\_proba(X)$ and $predict\_proba(X)$ make the prediction on the test set
- $score(X[, y])$ returns the score on the given data, if the estimator has been refit.
- $set\_params(**params)$ sets the parameters of this estimator.

4.2 The Implementation

The first step was to import all of the necessary libraries.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
import inversion
```

The *inversion* is not a built-in library of Python, it imports the written function from another .py file.

After, the selected dataset has captured by Pandas. In this case the used dataset is a .csv file, which contains 39.644 instances and 61 attributes. The attributes consist of 58 predictive features, 2 other attributes of accessory information and 1 goal field, which is the number of shares. The dataset can be represented as a matrix, where the columns are the features and the rows are the data. This dataset is preprocessed by Pandas as a DataFrame object. The *url* and *timedelta* columns have been omitted, since they are meta-data and cannot be treated as features.

```
dataset = pd.read_csv('OnlineNewsPopularity.csv')
dataset_copy = dataset.drop(columns=['url','timedelta'])
```

As already mentioned, the dataset's preprocessing and transformation has done before its publication, but the dataset needs more cleaning. Before executing the cleaning phase, the summarization of the dataset needs a review to collect the data that need to be cleaned.

```
dataset_copy.describe()
```

where the first 5 columns results the following from the initial dataset:

	n_tokens_title	n_tokens_content	n_unique_tokens	n_non_stop_words	n_non_stop_unique_tokens
count	39644.000000	39644.000000	39644.000000	39644.000000	39644.000000
mean	10.398749	546.514731	0.548216	0.996469	0.689175
std	2.114037	471.107508	3.520708	5.231231	3.264816
min	2.000000	0.000000	0.000000	0.000000	0.000000
25%	9.000000	246.000000	0.470870	1.000000	0.625739
50%	10.000000	409.000000	0.539226	1.000000	0.690476
75%	12.000000	716.000000	0.608696	1.000000	0.754630
max	23.000000	8474.000000	701.000000	1042.000000	650.000000

As it can be seen in the table, there are outlying values that would cause noise if they are not handled. Also, in the known of the meaning of each columns, some inconsistencies are occured, like the *n_tokens_content* feature contains the number of words in the content, which cannot be zero. These inconsistent values needs a correction.

4.2.1 Optimizing the Dataset

There are a lot of techniques in data mining for optimizing the data into appropriate forms. In the case of outlying and inconsistent values, due to the huge number of data, the chosen technique was to simply ignore those tuples.

```
dataset_copy = dataset_copy[dataset_copy.n_tokens_content != 0]
dataset_copy = dataset_copy[dataset_copy.n_unique_tokens <= 1]
dataset_copy = dataset_copy[dataset_copy.average_token_length != 0]

dataset_copy = dataset_copy[dataset_copy.num_hrefs <= 100]
dataset_copy = dataset_copy[dataset_copy.num_self_hrefs <= 10]
dataset_copy = dataset_copy[dataset_copy.num_imgs <= 10]
dataset_copy = dataset_copy[dataset_copy.num_videos <= 2]
dataset_copy = dataset_copy.reset_index(drop=True)
```

These reductions need some explanation. As mentioned above, *n_tokens_content* is the number of words in the content, so it cannot be zero. *n_unique_tokens* contains the rate of unique words in the content. Because of *n_unique_tokens* is a rate, it need to be between [0,1]. The *average_token_length* contains the average length of the words in the content, which also cannot be zero.

The other optimizations are for handling the outlying values. *num_hrefs* contains the number of links, *num_self_hrefs* is the number of links to other articles published by Mashable, *num_imgs* has the number of images and *num_videos* stands for the number of videos. Furthermore because this dataset is a Pandas DataFrame and *drop* is a function for removing the whole row, the dataset needs to be reindexed.

After the cleaning, the dataset's other part has a big difference between its values, so the whole dataset should be scaled.

```
scaler = StandardScaler()
dataset_copy[:] = scaler.fit_transform(dataset_copy[:])
```

With the usage of StandardScaler's *fit_transform* method, the dataset has been fitted and transformed in one step.

Then the dataset has separated into feature X and target y groups.

```
y = dataset_copy.pop('shares')
X = dataset_copy
```

The column *shares* contains the target values, which are the number of shares.

Learning the parameters of a prediction function and testing the accuracy of the learning on the same data is a methodological mistake: a model that is just repeating the values of the samples would have a perfect score, but it cannot predict anything useful from the data that have not been previously processed. Furthermore a model that completely trains a dataset, also trains its noises, so it will predict worse. This problem is called overfitting. To avoid it, a common practice is when performing a supervised machine learning experiment to hold out a part of the available data as a testing set *X_test*, *y_test*.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.3, random_state=0)
```

Now the training (*X_train*, *X_test*) and testing (*y_train*, *y_test*) sets are made. The *test_size* is a float, which means the testing sets have this proportion from the dataset.

4.2.2 Training the Neural Network

The training consists of the application of machine learning techniques. Different multi-layer perceptron models are fitted on the training sets with a set of hyper-parameters *param_grid*. These parameters are the number of hidden layers, the activation functions, the optimization algorithms, the alpha value, the learning rate's type and the learning rate's initialize value.

In Scikit-Learn, the model is conventionally called estimator. Scikit-Learn's neural network model is the MLPRegressor, which is used as an estimator of GridSearchCV. The process of the training is about fitting the neural network model to the training sets, then testing the accuracy on the testing sets by predicting the output *y_pred*.

```
mlp = MLPRegressor()
param_grid = {
  'hidden_layer_sizes': [(100,100), (100,200,500), (50,150,50),
   (20,80,80,20), (30,90,180,90,30), [200,500,200],
   (100,200,500,200,100), (1000,2000)],
  'activation': ['relu', 'logistic', 'tanh'],
  'solver': ['lbfgs', 'adam'],
  'alpha': [0.03, 0.01, 0.003, 0.001],
  'learning_rate_init': [0.03, 0.01, 0.003, 0.001],
  'learning_rate': ['adaptive'],
}
gs = GridSearchCV(mlp, param_grid, cv=2, n_jobs=-1)
gs.fit(X_train, y_train)
```

This process is time consuming, since the amount of the given parameters are trained on a large-scale dataset. The search was running in parallel on the supercomputer and the training lasted for days. The score was computed by the L2 normal of the loss function, which means the difference between the original and the predicted value.

It can be seen, that stochastic gradient descent is not on the list of the optimization methods. When the dataset was trained, SGD causes NaN or infinite values during the calculation of the gradients. The solution was to remove SGD from the solvers. After the trained multi-layer perceptron is ready, the results can be stored and plotted.

```
regressor = gs.best_estimator_
print(regressor, 'score: ', gs.best_score_, sep='\n ', file=open('
    TrainingResult.txt', 'w'))

y_pred = regressor.predict(X_test)

plt.plot(X_test, y_test, 'o', color='orange')
plt.plot(X_test, y_pred, 'o', color='blue')
plt.savefig("./" + 'plot.pdf')
plt.show()
```

The resulted values called scores return with the coefficient of the determinated prediction. This means, the score is the prediction value of the difference between the original and the predicted outputs. The prediction is the most accurate, if the score approximates to 1. If the score is between the range $[-1, 1]$, the function slowly converge. However if the score is outside of this range, that means there is no convergence.

```
MLPRegressor(activation='logistic', alpha=0.001, batch_size='auto',
  beta_1=0.9, beta_2=0.999, early_stopping=False, epsilon=1e-08,
  hidden_layer_sizes=(100, 200, 500), learning_rate='adaptive',
  learning_rate_init=0.003, max_iter=200, momentum=0.9,
  n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
  random_state=None, shuffle=True, solver='lbfgs', tol=0.0001,
  validation_fraction=0.1, verbose=False, warm_start=False)
score: 0.02001904720978248
```

The results of the training contains a summarization of the best estimator and the score. The best score was 0.02001904720978248, which means the regression function was just rarely approximates the model, as the best fitting hyperparameter combination has not captured yet. The best estimator parameters were the **logistic** activation function and **L-BFGS** as optimization algorithm, with the use of 0.001 alpha value and 0.003 initial learning rate. The training uses 3 hidden layers with $(100, 200, 500)$ neurons in each layers.

As mentioned, the result of the training is plotted, which includes the tested target values and the predicted values.

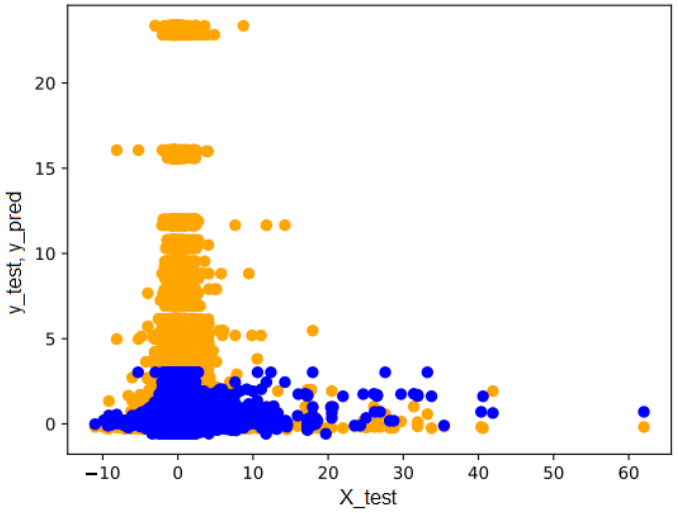


Figure 4.2: The result of the multi-layer perceptron training

The orange points represents the original values from the test set and the blue points are the results of the prediction. The horizontal line means the range of the values located in the original testing set y_test and the predicted one y_pred and the vertical line represents the range of the values in respect of the input testing set X_test.

4.2.3 Inverting the MLP

The neural network inversion was implemented as a python function, called *invert*(). The parameters of the invert function are the best estimator, the expected output, the size of the input vector, the step size, which is the learning rate, the number of iterations, and a boolean which stands for the function to be verbose or not.

```
def invert(regressor,
  expected_output,
  input_size,
  step_size,
  iteration_count = 100,
  verbose = False):

  [...]
```

The process of the inversion is the following: First, the network is initialized with a random input vector, then the sizes of the layer units are defined.

```
def invert(regressor, expected_output, input_size, step_size,
    iteration_count = 100, verbose = False):

  guessedInput = np.random.rand(input_size)
  layer_units = [[input_size] + list(regressor.hidden_layer_sizes) +
   [regressor.n_outputs_]]

    [...]
```

The inversion is an iterative process where the equations in the WLK inversion are computed recursively. The first task is to perform the implementation of the Equation 3.23. The equation separates two cases for computing the value of δ. One of the cases includes the computation for the output layer and the other one is for the input and hidden layers. The outer iteration stands for to approximate the input values more precisely. The internal iterations compute the values of δ in vector $deltas[]$, depending on the layer it stands in. The $deltas[-1]$ means the last value of the vector, that represents the output layer. The $deltas[-i-1]$ provides the recursive computation to the vector's values.

```
def invert(regressor, expected_output, input_size, step_size,
    iteration_count = 100, verbose = False):

  guessedInput = np.random.rand(input_size)
  layer_units = [[input_size] + list(regressor.hidden_layer_sizes) +
   [regressor.n_outputs_]]

  for j in range(0, iteration_count):
    activations = [guessedInput]

    for i in range(regressor.n_layers_ - 1):
      activations.append(np.empty((guessedInput.shape[0],
      layer_units[0][i + 1])))

    regressor._forward_pass(activations)
    y_pred = activations[-1]
    deltas = activations.copy()
    deltas[-1] = _activationFunctionDerivate(activations[-1],
     regressor.activation) * (y_pred - expected_output)

    for i in range(1, len(activations)):
      deltas[-i-1] = _activationFunctionDerivate(activations[-i-1],
       regressor.activation) * (regressor.coefs_[-i] * deltas[-i].T)
       .sum(axis=1)

      [...]
```

As it is mentioned above, a parameter *verbose* of the *invert*() function is a boolean, which stands for to show the detailed information about the computation or not. If $verbose = True$, the following information are shown in connection with $deltas[-i-1]$.

```
    for i in range(1, len(activations)):
      deltas[-i-1] = _activationFunctionDerivate(activations[-i-1],
       regressor.activation) * (regressor.coefs_[-i] * deltas[-i].T)
       .sum(axis=1)

      if verbose:
        print('#', i)
        print(regressor.coefs_[-i])
        print(deltas[-i])
        print(regressor.coefs_[-i] * deltas[-i].T)
        print((regressor.coefs_[-i] * deltas[-i].T).sum(axis=1))
        print(activations[-i-1])
        print(_activationFunctionDerivate(activations[-i-1],
         regressor.activation))
        print(deltas[-i-1])
        print('--------------------')

    [...]
```

After the *deltas*[] are computed, the next step is to implement the Equation 3.21, where the current value of the input vector is approximated. Then the return value is the guessed value of the desired input.

```
def invert(regressor, expected_output, input_size, step_size,
    iteration_count = 100, verbose = False):

  guessedInput = np.random.rand(input_size)
  layer_units = [[input_size] + list(regressor.hidden_layer_sizes) +
   [regressor.n_outputs_]]

  for j in range(0, iteration_count):
    activations = [guessedInput]

    for i in range(regressor.n_layers_ - 1):
      activations.append(np.empty((guessedInput.shape[0],
       layer_units[0][i + 1])))

    regressor._forward_pass(activations)
    y_pred = activations[-1]
    deltas = activations.copy()
    deltas[-1] = _activationFunctionDerivate(activations[-1],
     regressor.activation) * (y_pred - expected_output)

  for i in range(1, len(activations)):
    deltas[-i-1] = _activationFunctionDerivate(activations[-i-1],
     regressor.activation) * (regressor.coefs_[-i] * deltas[-i].T)
     .sum(axis=1)

    if verbose:
      [...]

  guessedInput = guessedInput - step_size * deltas[0]

return guessedInput
```

As it is known, the Equation 3.23 uses the derivatives of the activation functions. A function *_activationFunctionDerivate*() was defined, which computes the derivatives of the used activation.

```
def _activationFunctionDerivate(X, activation):
  if activation == 'tanh':
    return 1.0 / (np.cosh(X)**2)
  if activation == 'logistic':
    log_sigmoid = 1.0 / (1.0 + np.exp(-1 * X))
    return log_sigmoid * (1.0 - log_sigmoid)
  if activation == 'relu':
    return [1.0 if np.any(X > 0.0) else 0.0]
```

The defined inverse function can be called to accomplish the inversion. The neural network inversion is implemented on the testing output set *y_test* to compute those values from the testing input set *X_test*, which eventuate the values in the testing output set. Every iteration results a tuple with the values of the guessed input.
The accuracy percent is computed by getting the division of the difference between the guessed value's prediction and the actual desired output value, and the range of the testing output vector *y_test*.

```
inversionResults = pd.DataFrame(columns=['accuracy_percent'])

for i, value in enumerate(y_test):
  desired_output = [value]
  guessedInput = inversion.invert(regressor, desired_output, pd.
    DataFrame(X_test).columns.size,
  gs.best_params_['learning_rate_init'])
  guessedInput = pd.DataFrame(guessedInput).T

  accuracy = 1 - abs((regressor.predict(guessedInput) - desired_output)
    / (y_test.max() - y_test.min()))
  inversionResults = inversionResults.append({'accuracy_percent':
    accuracy[0]*100}, ignore_index=True)

  print('guessed input vector in X_test: ', np.array(guessedInput),
  'predicted input vector for X_test: ', regressor.predict(guessedInput
    ),
  'desired output value in y_test: ', desired_output,
  'error: ', regressor.predict(guessedInput) - desired_output,
  'accuracy percent: ', accuracy[0]*100, '\n',
    sep='\n ', file=open('InversionResults.txt', 'a'))

print('summarization: ', inversionResults.describe(), sep='\n ', file=
    open('InversionResults.txt', 'a'))
```

The guessed and predicted input vectors, the desired output values, the error and the accuracy percent are stored and written into a .txt file. The summarization of the accuracy is described.

4.3 Results

Neural network inversion has not received much attention since the rise of deep learning. Inversion procedures seek to find those input values, which produces the given output values. The Williams-Linder-Kindermann inversion method is a type of single element search methods, which can solve the inversion problem.

The training was running in parallel on the supercomputer with thousands of hyperparameter combinations of the multi-layer perceptron. After thousands of iterations has processed, the neural network mean test score has not reached acceptable levels. However the WLK inversion has been tested on smaller datasets with trained neural network and achieved good results. The summarization table of the accuracy percents is the following:

	accuracy_percent
count	8727.000000
mean	98.578775
std	2.766924
min	4.258826
25%	98.572265
50%	98.776263
75%	99.105716
max	99.999681

Inversion is firmly depends on the result of the training. The more accurate the training is, the more effective the inversion will be. Finding the best fitting parameters to the dataset, especially to large-scale datasets takes great amount of time. As mentioned the training was running in parallel on the supercomputer with thousands of parameter combinations, but the best fitting combinations are not captured yet. In summary Willams-Linder-Kindermann inversion can generate the features of the input set correctly.

Chapter 5

Summary

The inversion of a neural network has not received much attention since the rise of machine learning, and Python's machine learning libraries have not encompassed any inversion methods. In this paper a solution was given for computing the inversion of a function approximated by neural networks.

Three main tasks were tackled in this paper to perform a solution: data mining for extracting information from the data set, building and testing multi-layer perceptron models, and the inversion of the single element feedforward neural network. The presented tasks were performed in Python with the assistance of its third-party libraries.

The goal is to perform the inversion, for what a sufficient number of neural network models has to be established. The necessary tasks are described in the thesis. Data mining was used for extracting information from the data set. Multi-layer perceptron models were built and tested with numerous hyperparameter combinations to find the best fitting combination which can predict the outputs the best. Then the single element inversion of the feedforward neural network was completed by the Williams-Linder-Kindermann inversion.

The Online News Popularity dataset, which is a publicly available dataset was used during the whole process from data mining to the inversion. Various data mining and cleaning techniques were proceeded on the dataset to make it ready for training. After the optimization, the training was running in parallel on the supercomputer with thousands of hyperparameter combinations of the multi-layer perceptron. After thousands of iterations the neural network mean test score has not reached acceptable levels yet. WLK inversion has been tested on smaller datasets with trained neural network with good results.

For further plans, the best fitting parameter combination has to be found in respect for the processed dataset. To achieve it, the dataset needs more data cleaning steps and to think over to get more features with predictable information. Also, expanding the amount of hidden layers can increase the accuracy of the fitting.

This paper contains applied machine learning research in the field of artificial intelligence with the implementation of neural network inversion. Furthermore as a future plan, this explanation will be sent to Scikit-Learn as a scientific solution of the inversion problem.

Bibliography

[1] Online News Popularity Data Set. `http://archive.ics.uci.edu/ml/datasets/Online+News+Popularity`.

[2] Python. `https://www.python.org/`.

[3] Scikit-Learn. `https://scikit-learn.org/stable/`.

[4] SciPy. `https://www.scipy.org/`.

[5] UCI Machine Learning Repository. `https://archive.ics.uci.edu/ml/index.php`.

[6] Craig A. Jensen, Russell D. Reed, Robert J Marks, Mohamed El-Sharkawi, Jae-byung Jung, Robert T. Miyamoto, Gregory M. Anderson, and J Eggen. Inversion of feedforward neural networks: Algorithms and applications. 02 2001.

[7] M.P. Allen. *Understanding Regression Analysis*. Springer US, 2007.

[8] E. Alpaydin. *Introduction to Machine Learning*. Adaptive Computation and Machine Learning series. MIT Press, 2009.

[9] G.A. Anastassiou. *Intelligent Systems: Approximation by Artificial Neural Networks*. Intelligent Systems Reference Library. Springer Berlin Heidelberg, 2011.

[10] J.A. Anderson. *An Introduction to Neural Networks*. Bradford book. MIT Press, 1995.

[11] Y. Chauvin and D.E. Rumelhart. *Backpropagation: Theory, Architectures, and Applications*. Developments in Connectionist Theory Series. Taylor & Francis, 2013.

[12] G. Dong and H. Liu. *Feature Engineering for Machine Learning and Data Analytics*. Chapman & Hall/CRC Data Mining and Knowledge Discovery Series. CRC Press, 2018.

[13] J. Feldman and R. Rojas. *Neural Networks: A Systematic Introduction*. Springer Berlin Heidelberg, 2013.

[14] Kelwin Fernandes, Pedro Vinagre, and Paulo Cortez. A proactive intelligent decision support system for predicting the popularity of online news. In *EPIA*, 2015.

[15] T.L. Fine. *Feedforward Neural Network Methodology.* Information Science and Statistics. Springer New York, 2006.

[16] F.A. Graybill and H.K. Iyer. *Regression Analysis: Concepts and Applications.* An Alexander Kugushev book. Duxbury Press, 1994.

[17] J. Han, J. Pei, and M. Kamber. *Data Mining: Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011.

[18] K. Jolly. *Machine Learning with scikit-learn Quick Start Guide: Classification, regression, and clustering techniques in Python.* Packt Publishing, 2018.

[19] N. Karayiannis and A.N. Venetsanopoulos. *Artificial Neural Networks: Learning Algorithms, Performance Evaluation, and Applications.* The Springer International Series in Engineering and Computer Science. Springer US, 2013.

[20] J Kindermann and A Linden. Inversion of neural networks by gradient descent. *Parallel Computing*, 14(3):277 – 286, 1990.

[21] R.S. Michalski, J.G. Carbonell, and T.M. Mitchell. *Machine Learning: An Artificial Intelligence Approach.* Number 1. k. Elsevier Science, 2014.

[22] T.M. Mitchell. *Machine Learning.* McGraw-Hill International Editions. McGraw-Hill, 1997.

[23] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jacob VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] G. Pillo and F. Giannessi. *Nonlinear Optimization and Applications.* SpringerLink : Bücher. Springer US, 2013.

[25] K.L. Priddy and P.E. Keller. *Artificial Neural Networks: An Introduction.* SPIE tutorial texts. Society of Photo Optical, 2005.

[26] A.K. Pujari. *Data Mining Techniques.* Universities Press, 2001.

[27] D. Pyle. *Data Preparation for Data Mining.* The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 1999.

[28] He Ren and Quan Yang. Predicting and evaluating the popularity of online news. 2015.

[29] S. Sathasivam. *Optimization Methods in Training Neural Networks.* Universiti Sains Malaysia, 2003.

[30] D.X. Tho. *Perceptron Problem in Neural Network.* GRIN Verlag, 2010.

[31] G. Van Rossum and F.L. Drake. *An Introduction to Python*. Network Theory Limited, 2011.

[32] J. VanderPlas. *Python Data Science Handbook: Essential Tools for Working with Data*. O'Reilly Media, 2016.

[33] T. Veerarajan. *Numerical Methods*. Sigma series. McGraw-Hill Education (India) Pvt Limited, 2007.

[34] J. Wolberg. *Data Analysis Using the Method of Least Squares: Extracting the Most Information from Experiments*. Springer Berlin Heidelberg, 2006.

[35] M.J. Zaki, J.X. Yu, B. Ravindran, and V. Pudi. *Advances in Knowledge Discovery and Data Mining, Part I: 14th Pacific-Asia Conference, PAKDD 2010, Hyderabat, India, June 21-24, 2010, Proceedings*. Advances in knowledge discovery and data mining : 14th Pacific-Asia conference, PAKDD 2010, Hyderabad, India, June 21-24, 2010 : proceedings. Springer, 2010.

[36] A. Zheng and A. Casari. *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. O'Reilly Media, 2018.

Media Instruction Manual

The source code of the implementation can be found on the attached media. As a requirement, Anaconda must be installed since it is the interpreter for running Python files, hence no Python IDE have to be downloaded.

To load the source code, the environment must be set by the assistance of Anaconda's package manager system called conda. Since there are no integrated environment for Python, a virtual environment has to be established. To accomplish it, the following command has to be written in the conda prompt:

```
conda create -n venv python=3.6
```

Then the virtual environment *venv* must be activated.

```
conda activate venv
```

For running the source code, the necessary packages need to be loaded, that are located in the requirements.txt file.

```
pip install -r requirements.txt
```

Finally the virtual environment is ready to load the Python file.

In the folder *python*, there are two .py files. The *inversion.py* contains the implementation of the inversion and the *main.py* includes everything else. Hence the *main.py* has to be loaded by the virtual environment with the following command:

```
python main.py
```