



**BRNO UNIVERSITY OF TECHNOLOGY**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

**NEURAL NETWORK TRAINING PROGRESS  
VISUALIZATION**

VIZUALIZACE PRŮBĚHU TRÉNOVÁNÍ NEURONOVÉ SÍŤE

**BACHELOR'S THESIS**

BAKALÁŘSKÁ PRÁCE

**AUTHOR**

AUTOR PRÁCE

**SUPERVISOR**

VEDOUCÍ PRÁCE

**SILVIE NĚMCOVÁ**

**Ing. KAREL BENEŠ**

BRNO 2021

# Bachelor's Thesis Specification



Student: **Němcová Silvie**  
Programme: Information Technology  
Title: **Neural Network Training Progress Visualization**  
Category: Artificial Intelligence

Assignment:

1. Get acquainted with modern neural networks
2. Get acquainted with recent techniques for inspection of their training
3. Select a set of those and implement them as a toolkit
4. Demonstrate its capabilities on a suitable task
5. Propose and execute a novel inspection experiment

Recommended literature:

- Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein: *Visualizing the Loss Landscape of Neural Nets*. 2018.
- Ian J. Goodfellow, Oriol Vinyals, and Andrew M. Saxe: *Qualitatively characterizing neural network optimization problems*. 2015.

Requirements for the first semester:

- Items 1, 2 and 3.

Detailed formal requirements can be found at <https://www.fit.vut.cz/study/theses/>

Supervisor: **Beneš Karel, Ing.**  
Head of Department: Černocký Jan, doc. Dr. Ing.  
Beginning of work: November 1, 2020  
Submission deadline: May 12, 2021  
Approval date: October 30, 2020

## Abstract

This work studies a neural network model during its training. The aim of this thesis is to visualize the training of the model and to examine the training. To achieve this goal I choose to implement a set of tools in Python language. The implementation successfully reproduces the linear path experiment, the identification of robust and ambient layers and the visualization of the loss surface. In addition the quadratic path experiment is presented in this thesis as novel method for analyzing the neural network training progress visualization.

## Abstrakt

Tato práce se zabývá studiem průběhu trénování modelu neuronové sítě. Cílem je zobrazit a zkoumat trénovací proces modelu neuronové sítě. Pro tyto účely jsem zvolila implementaci v jazyce Python. Implementace úspěšně replikuje vizualizaci průběhu trénování pomocí lineární interpolace, identifikaci robustních a *ambient* vrstev a zobrazení plochy, vytvořené účelovou funkcí okolo natrénovaného modelu. V této práci je navržena a představena metoda zobrazování průběhu trénování pomocí kvadratické interpolace parametrů. Výsledek práce je znázorněn grafy a diskuzí nad dopady změn parametrů modelu na jeho trénování.

## Keywords

Artificial intelligence, neural networks, neural network training, Python, neural network training visualization.

## Klíčová slova

Umělá inteligence, neuronové sítě, trénování neuronové sítě, Python, vizualizace trénování neuronové sítě.

## Reference

NĚMCOVÁ, Silvie. *Neural Network Training Progress Visualization*. Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Karel Beneš

## Rozšířený abstrakt

V minulosti trpělo strojové učení několika problémy. Velmi komplexní a vysoce výpočetně náročné trénování neuronových sítí byl jedním z nich. Se vzrůstajícím výpočetním výkonem dostupných strojů a objevením nových efektivních algoritmů se strojové učení znovu těší velké oblibě. I přesto se nadále v oblasti trénování vyskytují nejasnosti, pochyby o schopnosti trénovacího algoritmu překonat lokální optima nebo sedlové body. Cílem této práce je umožnit zobrazení průběhu trénování neuronové sítě na různých úrovních a pomocí různých metod. Tyto metody poté porovnat a navrhnout vhodnou metodu pro zobrazování průběhu trénování za účelem hlubšího zkoumání trénování neuronových sítí.

Prvním krokem bylo důkladně nastudovat neuronové sítě a jejich trénování. Práce se zaměřuje na jednoduchý model konvoluční neuronové sítě a trénování pomocí *stochastic gradient descent* (SGD) algoritmu. Dále jsou popsány typická problematická místa během trénování.

Pro zkoumání průběhu trénování neuronových sítí je úspěšně zreprodukována metoda lineární interpolace parametrů pro celý model, prezentována Goodfellowem a ostatními v [5]. Na základě této metody je navržena metoda pro zkoumání jednotlivých vrstev či parametrů modelu. Tento návrh byl úspěšně implementován. Experiment může ukázat význam vrstev a jednotlivých parametrů pro daný model a tak umožnit zhodnocení efektivity architektury modelu neuronové sítě.

Nově navrhovaná metoda je metoda kvadratické interpolace na úrovni modelu, vrstev a parametrů. Tato metoda je navržena na základě lineární interpolace s cílem dodat výsledky věrnější realitě a tak umožnit přesnější zhodnocení modelu bez zvýšení nároků na výkon. Tato metoda byla úspěšně implementována s využitím Lagrangeova interpolačního polynomu pro získání koeficientů kvadratické rovnice potřebných pro výpočet interpolovaných hodnot. Výsledky tohoto experimentu korespondují s výsledky lineární interpolace a podle očekávání podávají výsledky podobnější reálných datům.

Další zkoumanou oblastí v průběhu trénování neuronové sítě je zkoumání okolí natrénovaného modelu pomocí 3D vizualizace hodnot účelové funkce v okolí natrénovaného modelu. Zobrazení okolí natrénovaného modelu bylo provedeno pomocí projekce v náhodných směrech podle metody navržené Goodfellowem a ostatními v [5] i pomocí směrů vybraných pomocí analýzy hlavních komponent, metody prezetované Li a ostatními v [9].

Zobrazení prostou projekcí v náhodných směrech nemusí vždy poskytovat vhodné podmínky pro zobrazení cesty, kterou zvolil optimalizační algoritmus. Ve vysoce dimenzionálním prostoru je velká pravděpodobnost, že dva náhodně vybrané směry na sebe budou ortogonální a tak by nemusely efektivně zobrazit cestu [9]. Řešení výše zmíněného problému navrhl Li a ostatní ve svém článku [9]. Navrhuje metodu projekce ve směrech, vybraných pomocí PCA na základě nejvíce vypovídající hodnoty. Tento experiment byl úspěšně reprodukován a jeho výsledky ukazují, že zvolený optimalizační algoritmus bezpečně konverguje.

Navržený a implementovaný nástroj může sloužit výzkumníkům pro zkoumání průběhu trénování neuronové sítě nebo pro identifikaci redundantních vrstev a analýze efektivity architektury neuronových sítí.

# Neural Network Training Progress Visualization

## Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Karel Beneš. I have listed all the literary sources, publications and other sources from which I gathered information.

.....  
Silvie Němcová  
May 9, 2021

## Acknowledgements

My sincere gratitude goes to my supervisor, Ing. Karel Beneš, for his guidance and continuous support when creating this project.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Introduction to Neural Networks</b>	<b>3</b>
2.1	Learning Algorithms in Artificial Intelligence . . . . .	3
2.2	Neural Networks . . . . .	5
2.3	Feed Forward Neural Networks . . . . .	6
2.4	Convolutional Neural Networks . . . . .	7
<b>3</b>	<b>Neural Network Training</b>	<b>10</b>
3.1	Introduction to Neural Network Training . . . . .	10
3.2	Loss Functions . . . . .	11
3.3	Backpropagation . . . . .	11
<b>4</b>	<b>Neural Network Training Progress Visualization</b>	<b>14</b>
4.1	Linear Path Examination . . . . .	14
4.2	Quadratic Interpolation of the Parameters . . . . .	15
4.3	Loss Function Landscape Visualization . . . . .	15
<b>5</b>	<b>Implementation</b>	<b>17</b>
5.1	Design . . . . .	17
5.2	Technology . . . . .	17
<b>6</b>	<b>Examining the Training Progress of Neural Network</b>	<b>19</b>
6.1	Preliminary experiments . . . . .	20
6.2	Linear Path Experiment . . . . .	22
6.3	Quadratic Path Experiment . . . . .	28
6.4	Comparison Between the Quadratic and Linear Path . . . . .	33
6.5	Loss Function Surface Visualization . . . . .	36
<b>7</b>	<b>Conclusion</b>	<b>38</b>
	<b>Bibliography</b>	<b>39</b>

# Chapter 1

## Introduction

Neural networks are computational models inspired by biological neurons and connections between them, generally used for making decisions or recognizing patterns. Nowadays, they have many practical applications such as computation of self-driving cars collision probability estimation or speaker, face or handwriting recognition.

The origin of the artificial neural networks dates back to the 40s of the 20th century when Warren McCulloch and Walter Pitts introduced a computational model of the neural networks based on an algorithm called threshold logic [12]. Donald Hebb took the idea further in his book, *The Organization of Behaviour* [6], proposing that some neural connections could have a bigger impact on the result after each successful use. In 1958, Frank Rosenblatt introduced the Perceptron. The Perceptron builds on top of McCulloch's model [18]. The inability to solve nonlinear problems like logical *exclusive or* is the drawback of the Perceptron. Thus, in the 1960s, the research of neural networks stagnated. In 1969, Minsky and Papert published a book [13], where they introduced two major problems with neural networks. The first of them concerned with the inability of the single layer networks to solve nonlinear problems and the second one is dealing with the lack of computation power to compute the output of the complex neural networks. Since the 1990s, the research of neural networks continues. Nowadays neural networks are a popular tool among computer scientists.

The training of the neural networks is even more computationally demanding task than the output computing is. In this thesis, the training process is visualized and examined. I focus on experimenting with modifying the parameters of a model of the neural network to find which of them are crucial for training and which are not. The experimental results provide useful information for optimizing and analysing the training process of the neural networks. Ideas on how to optimize the training process of neural networks could be later derived from the experiments.

## Chapter 2

# Introduction to Neural Networks

Neural networks build learning algorithms on top of the general artificial intelligence learning algorithms. They implement using the common artificial intelligence learning algorithms, like linear and logistic regression. The combination of these algorithms leads to the ability of the neural network to learn and provide the desired output. The format of the output is defined by the specification of the task to be solved by the neural network. This chapter provides a brief introduction to the use of artificial intelligence algorithms in neural networks.

### 2.1 Learning Algorithms in Artificial Intelligence

A learning algorithm describes a process by which a model is able to learn from the input data. In the *Machine Learning* by Mitchell T. M. (1997) provides a definition: „A computer program is said to learn from experience  $E$  with respect to some class of tasks  $T$  and a performance measure  $P$ , if its performance on tasks in  $T$ , as measured by  $P$ , improves with experience  $E$ “ [14]. The *computer program* represents a computational *model* using artificial intelligence algorithms to solve the tasks. Most tasks for artificial intelligence are formulated as:

- **Classification:** The model is asked to specify to which of the  $k$  categories an input sample from a dataset belongs. Classification is, for example, used in object recognition which enables recognizing faces.
- **Regression:** Regression is finding correlations and estimating relationships between features of the examined data. Solving the regression task shows a trend in the examined dataset. It is used to answer questions about the impact of one feature on another feature.

The performance measure  $P$  is specific to the task  $T$ . The performance of a classification model is often evaluated by the *accuracy*. The accuracy is the proportion of examples for which the model produces the correct output. Equivalent information could be obtained by measuring the *error rate*, the proportion of examples for which the model produces incorrect output. The performance of the model is usually measured on previously unseen data called *test* dataset [4].

*Loss function* is used for training the model. If the output of the neural network deviates too much from the target output, the loss function would give a large number. The smaller



the output of the loss function, the better the model's performance is. The parameters of the model are adjusted on the output of the loss function during the training.

## Unsupervised learning

Unsupervised learning means that the model is learning independently. The training data does not have labels, the model has to find a pattern in the examined dataset itself. Thus, the goal of unsupervised learning algorithms is to find some structure in the dataset. Usually, the algorithm estimates the parameters of a probability distribution or creates clusters of data with similar features [4]. The prediction of the model is then done using the learned structure.

One of the unsupervised learning algorithms is principal component analysis (PCA). PCA is a multivariate technique that analyzes data in which observations are described by intercorrelated quantitative dependent variables. Its goal is to represent them as a set of new orthogonal<sup>1</sup> variables called principal components and display the pattern of similarity of the observations and the variables. PCA depends upon the eigendecomposition of positive semidefinite matrices and upon the singular value decomposition of rectangular matrices. The goals of the PCA are to extract the most important information from the input dataset, simplify and reduce the size<sup>2</sup> of the dataset by keeping only the most important information and analyze the structure of the observations and variables. The principal components are obtained as linear combinations of the original variables. PCA is used in exploratory data analysis, for making predictive models, and for reducing dataset dimensionality [1].

## Supervised learning

Supervised learning uses a dataset with target labels associated with data samples. The model is given a training dataset and for a certain number of *epochs*, the model is trained on this data. The training is done as follows: The model is given an input data sample. The model computes output. The model's output is compared with the expected output. Feedback of the model's performance is given to the model and the model's parameters are then adjusted. One of the most known supervised learning algorithms is *linear regression*. It predicts the value of a scalar  $y$  based on an input vector  $\mathbf{x}$ . The output is an approximation of  $y$ . The output is computed as a result of the linear regression of the input.

$$y' = b + \mathbf{w}^T \mathbf{x}, \quad (2.1)$$

where  $y$  is the output of the linear regression algorithm,  $\mathbf{w}^T$  is a transposed vector of parameters, and  $\mathbf{x}$  is the vector of input data. The vector of parameters determines how much each feature affects the prediction. If a feature's weight has a large absolute value, then changing that feature's value has a large impact on the algorithm's prediction.

To train the model using linear regression, a loss function is used. *Mean squared error* (MSE) is often used as the loss function.

$$\text{MSE} = \frac{1}{m} \sum_{i=0}^m (y - y')_i^2, \quad (2.2)$$

where  $m$  is the number of samples in the dataset,  $y'$  is the output of linear regression, and  $y$  is the target label.

---

<sup>1</sup>Perpendicular, separated features.

<sup>2</sup>Size of the dataset is number of samples that this dataset is containing.

Another algorithm is *logistic regression*, which is a classification algorithm. It is considered a regression in statistics, but in it is used as a classifier machine learning. The logistic regression gives an interpretation of the relative importance of features in a dataset. Logistic regression has as many inputs as the examined data have features. The logistic regression is then described as:

$$z = b + \mathbf{w}^T \mathbf{x}, \quad (2.3)$$

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}, \quad (2.4)$$

where  $b$  is the bias,  $\mathbf{w}$  is the vector of weights,  $\mathbf{x}$  is the vector of input data,  $z$  is the *logit* (weighted sum) of  $\mathbf{w}$  and  $\mathbf{x}$ ,  $\sigma$  is the *logistic (sigmoid)* function, and  $y$  is the output of the logistic regression. Everything except the bias and weights is calculated or given. These two elements are called *parameters*. The point of logistic regression is to *learn* good parameters to achieve good classification. Learning is a process of measuring how inaccurate the classification was and then updating the parameters based on this quantification. The measuring is done using the *loss function*, which is described later in Section 3.1.

## 2.2 Neural Networks

Neural network is a computational model that is inspired by the biological brain. It contains simple units, called *artificial neurons*, and parameters  $\Theta$ , which serve as information storage. The artificial neuron is shown in Figure 2.1. Neurons are interconnected via *weights* in a way that allows signals to travel through the network. Weights are, along with the biases, components of the parameters  $\Theta$ . The computational power of neural networks is derived from the density and complexity of the interconnections. The parameters  $\Theta$  are modified by the experience gained while *training* the model. The goal is to improve the performance of a model of neural network.

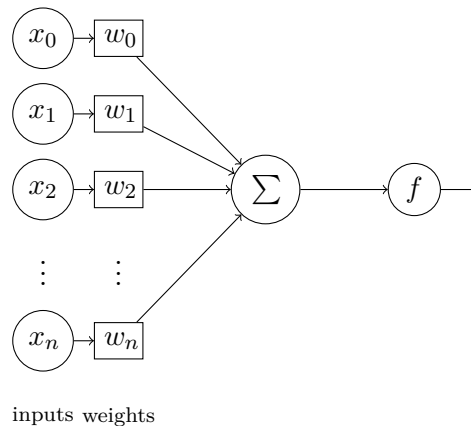


Figure 2.1: Model of an artificial neuron. The  $\mathbf{x}$  is a set of the inputs, the  $\mathbf{w}$  is a set of the weights. The input of the neuron is dot product of these two vectors.

The output of the neuron is the value of its *activation function*:

$$y = f(b + \mathbf{w}^T \mathbf{x}), \quad (2.5)$$

where  $f$  is the activation function,  $\mathbf{w}$  is the vector of weights,  $\mathbf{x}$  is the vector of input data, and  $b$  is bias. Neurons are in the neural network grouped in *layers*, a simple architecture is represented in Figure 2.2.

Neural networks use activation functions to propagate the output of one layer forward to the next layer. Activation functions take a scalar output of a neuron and yields another scalar as an activation of the neuron. The activation functions define the behavior of the model, they introduce nonlinearity into the model [16]. In this thesis, *Rectified Linear Unit (ReLU)* is used as the activation function. The ReLU is widely used thanks to its simplicity and effectiveness [17]. Because the gradient of ReLU is either zero or a constant, it evades the vanishing gradient issue [16]. ReLU is easy to calculate and converges quickly [10]. Commonly used activation functions are briefly introduced in the following table:

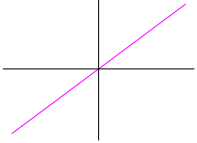
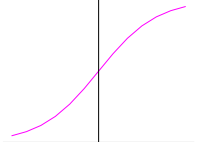
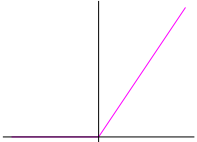
Identity		$f(x) = y$
Sigmoid		$f(x) = \frac{1}{1 + e^{-x}}$
ReLU		$f(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$

Table 2.1: Activation functions

## 2.3 Feed Forward Neural Networks

Feedforward networks are called *feed forward* because the data flows through the network only in one direction from the input layer to the output layer. The structure of a feed forward neural network consists of an input layer, one or more hidden layers, and an output layer. The input layer contains as many neurons as is the dimensionality of the input data or more. The hidden layers can contain various amounts of neurons, which are specified by the model's architecture. The output layer may have as many neurons as is the dimensionality of the desired output. For example, it can be 10 neurons for classifying the handwritten digit. The activation function of each neuron can be different. However, it is common that neurons in one layer use the same activation function.

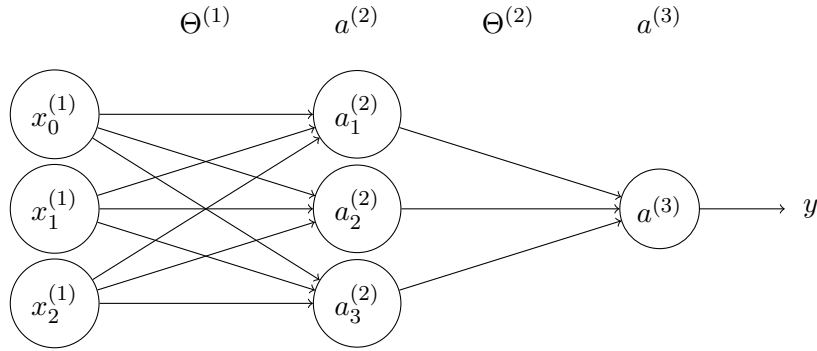


Figure 2.2: Feed forward neural network with one hidden layer. The  $\mathbf{x}^{(1)}$  represents input data,  $\Theta^{(1)}$  represents parameters for layer 1 the input layer.  $a_i^{(2)}$  represents  $i$ -th neuron in first hidden layer. The  $\Theta^{(2)}$  represents parameters for the output layer. The  $a^{(3)}$  represents the output layer and  $y_2$  the output of the network. The arrows indicate direction of data flow inside the network.

The neural networks can also be represented in a matrix notation, for the network in 2.2 it would be:

$$\begin{bmatrix} x_0^{(1)} \\ x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} \begin{bmatrix} \Theta_{00}^{(1)} & \Theta_{10}^{(1)} & \Theta_{20}^{(1)} \\ \Theta_{01}^{(1)} & \Theta_{11}^{(1)} & \Theta_{21}^{(1)} \\ \Theta_{02}^{(1)} & \Theta_{12}^{(1)} & \Theta_{22}^{(1)} \end{bmatrix} + \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \\ b_2^{(1)} \end{bmatrix} \xrightarrow{\text{activation}} \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \begin{bmatrix} \Theta_{00}^{(2)} \\ \Theta_{01}^{(2)} \\ \Theta_{02}^{(2)} \end{bmatrix} + \begin{bmatrix} b_0^{(2)} \\ b_1^{(2)} \\ b_2^{(2)} \end{bmatrix} \xrightarrow{\text{activation}} a^3 \xrightarrow{\text{activation}} y$$

## 2.4 Convolutional Neural Networks

Convolutional neural networks (CNN) are a type of feed forward neural networks. They are classified as feed forward networks because the information flows through the network from the input to the output. The convolutional NNs are inspired by the visual cortex of the biological brain [3].

Convolutional neural networks are usually used to solve difficult pattern recognition tasks on images. From the input image, the CNN outputs a prediction of a probability of how much the data sample belongs to a class, like any other type of NN. The CNNs allow encoding image-specific features into the architecture of the neural network, making the NN more suited for image-focused tasks and reducing the number of parameters of the model. The training of CNN consists usually of supervised learning on a labeled data set [15].

The architecture of convolutional neural networks is usually composed of three types of layers: *convolutional*, *pooling*, and *fully-connected layers* [15]. A simple CNN architecture is shown in Figure 2.3. The functionality of this architecture consists of four basic blocks. The first one is the input layer. This layer holds the pixel values of the input image. The second layer serves as a feature extraction layer. The layers used for feature extraction are the convolutional layers and the pooling layers. The convolutional layers search for features of the input image and the pooling layers reduce the number of the features to reduce the complexity. Finally, the fully connected layers perform the same operation as presented in 2.3. These layers produce output class probabilities [16].

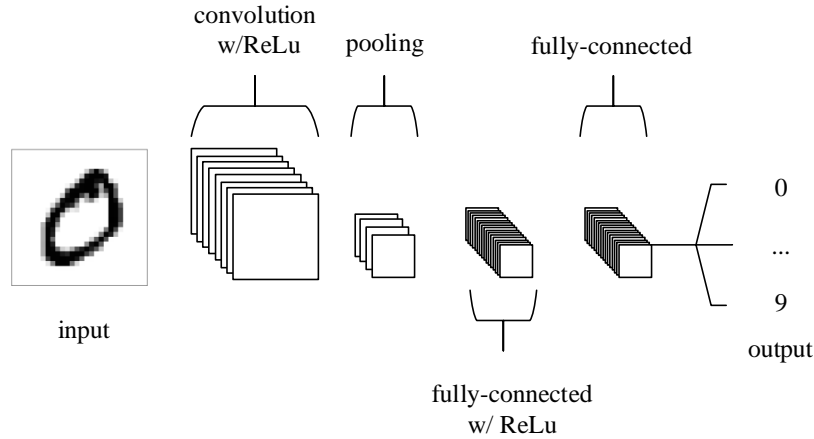


Figure 2.3: A simple convolutional neural network architecture, containing five layers. The architecture depicts an input layer, holding the pixel values of an input image, the feature extraction layers (convolutional) and the classification layers (fully connected) [15].

The input layer is where the model loads and stores raw input data of the image. The input specifies the width, height, and number of channels.

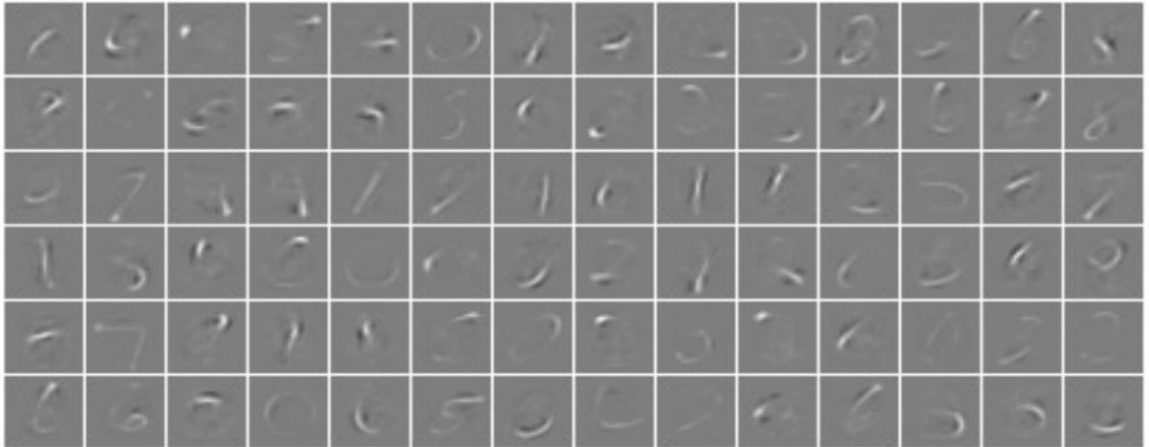


Figure 2.4: Visualization of extracted features obtained after convolution [15].

The convolutional layers are the foundation of how the convolutional neural networks operate. Convolutional layers transform the input data by using a batch of locally connected neurons from the previous layer [16]. The batch of these neurons is called a *convolutional kernel*. The kernels are usually small in spatial dimensionality but spread along the entire depth of the input.

The key concept of the convolutional layers is a *convolution*, represented by Equation 2.6. The convolution is a mathematical operation describing a rule for how to merge two sets of information. It can extract the features from the input. Because of this, convolution is used for feature extraction in convolutional neural networks. For example, a detection of an edge [16]. A visualization of the extracted features is represented by Figure 2.4.

$$f[x, y] * g[x, y] = \sum_i \sum_j f[i, j] \cdot g[x - i, y - j], \quad (2.6)$$

where the  $f[x, y]$  is original data sample,  $g[x, y]$  is *filter*.

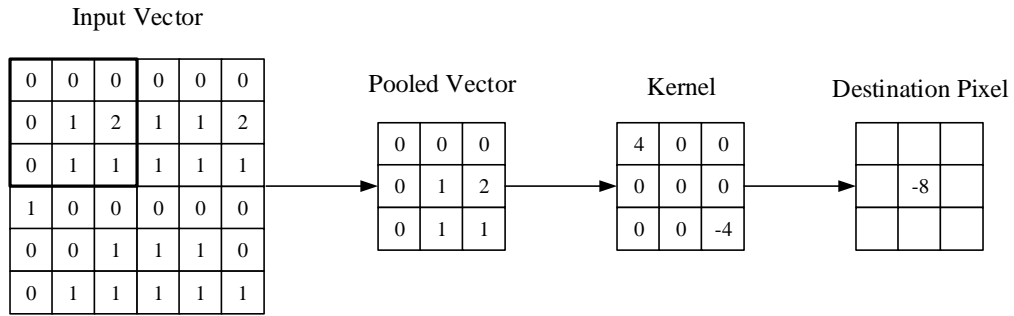


Figure 2.5: Representation of a convolution operation in a convolutional layer [15].

The input to the convolution can be a raw data or a feature map from another convolution. The Figure 2.5 represents one step in the convolutional layer. The filter can be also referred to as the *convolutional kernel* [16]. At each step, the filter is multiplied by the input data values, pooled to the same size as the filter is. This creates a single a value that is mapped to the output of the layer, called an *activation map* or a *feature map*. The values of numbers in the filter are the *weights* of the network that are updated after each optimizer step [16]. The feature maps for each filter are stacked together along the depth dimension to construct the output of the convolutional layer [15].

The pooling layers aim to reduce the dimensionality of the representation. They operate over each feature map in the input and scale its dimensionality. This is usually done using the max function, which chooses the maximum value of the pooled vector. The size of the pooled vector is defined by the kernel size. Usually, the convolutional layer and the pooling layer, which follows the convolutional layer, use the same size of kernel [15].

## Chapter 3

# Neural Network Training

Training neural network is a process of adjusting the parameters  $\Theta$  of the model so that the model can do a more accurate estimation. The Perceptron learning rule is briefly introduced in this section. This simple rule forms a foundation for more powerful algorithms used for training neural network models.

### The Perceptron Learning Rule

The Perceptron is a model of a single layer feed-forward neural network. This model consists of a binary threshold neuron  $y = f(b + \sum_i \mathbf{w}_i \mathbf{x}_i)$ , where  $f()$  is a binary threshold activation function,  $\mathbf{w}$  is a vector of weights associated with the input  $\mathbf{x}$ . The Perceptron is trained as follows [19]:

1. Randomly choose a training case.
2. If the predicted output matches the output label, do nothing.
3. If the Perceptron predicts 0 and it should have predicted 1, add the input vector to the weight vector.
4. If otherwise, subtract the input vector from the weight vector.

This algorithm is limited, it converges only for linearly separable datasets [16]. However, it gives us a glimpse of the way how updating the vector of weights works.

### 3.1 Introduction to Neural Network Training

Neural networks are used for solving complex, nonlinear problems. For this reason, it is necessary to use more advanced algorithms than the Perceptron learning rule. The training algorithm has to be efficient, able to avoid local optima, and convergent for non-convex optimization problems. The goal of the neural network model is to do the most possible accurate prediction. To achieve this, it is necessary to minimize the output of the *loss function*.

Training neural networks is considered as highly computationally demanding [11]. Successful training relies on good minimizing of non-convex functions. Certain choices for neural network hyper-parameters like batch size, learning rate, or optimizer affect the training process. However, the training process consists of iterative updating the parameters of the

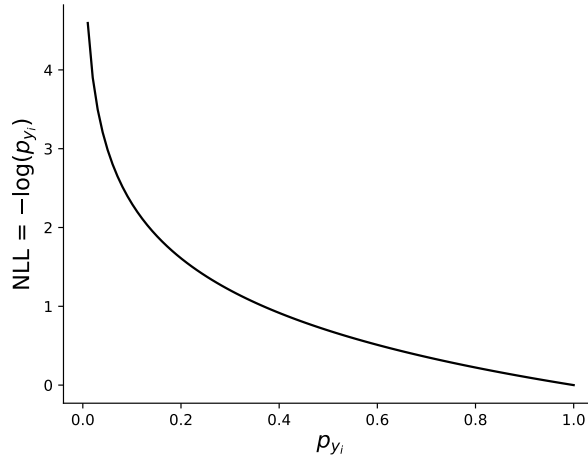


Figure 3.1: Negative Log Likelihood Visualization. The x-axis represents the value of the probability  $P_{y_i}$  and the y-axis the value of the negative log likelihood  $-\log(p_{y_i})$ . The graph clearly shows that as the value of negative log likelihood minimizes, the value of the probability maximizes.

model until the loss function reaches convergence. The way the parameters should update is calculated by an optimization algorithm.

## 3.2 Loss Functions

The loss function measures how much is the output of the model different than the expected output. The output of the loss function is calculated as the average of the aggregated errors over the entire data set. Each type of machine learning task has a different appropriate loss function. However, the choice of the loss function is not constrained since the idea behind all loss functions is the same.

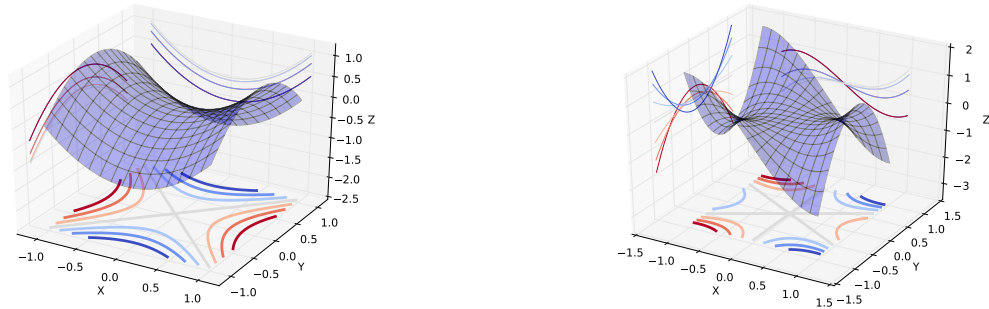
For classifying problems, the logistic loss is used. Generally, they are used when probabilities are of greater interest than hard classifications. Predicting probability means generating numbers between 0 and 1. For this reason, the activation function of the last layer must be the *softmax* activation function. The *negative log likelihood* is used in this thesis. The logarithm is a monotonically increasing function. Thus, minimizing the negative log-likelihood is equivalent to maximizing the probability as shown in Figure 3.1.

## 3.3 Backpropagation

The training process consists of an iterative update of the parameters of the model  $\Theta$  in order to find an optimal set of parameters  $\Theta_f$  for which the loss function has the lowest output value. The parameters are interdependent, so it is not possible to search for the best value of a parameter one by one. Changing one weight in the input layer affects the neuron, it propagates directly, and all neurons in the following layers up to the output.

The optimal set of parameters can be found by randomly guessing the values, but the dimensionality of the parameters makes it computationally very demanding as many various combinations exist [19]. Fortunately, the problem of finding the optimal set of





(a) Classic Saddle

(b) Monkey Saddle

Figure 3.2: Various types of saddle points [2]

parameters to satisfy the loss function can be generalized as a mathematical optimization of a multivariable function [7]. Popular optimization methods used in machine learning are gradient descent methods.

The optimization of the loss function reaches a *critical point* when the derivative of the loss function is close to zero. The critical points can be local minima or saddle points. When the derivative is approaching zero value, the optimization algorithm does not have any information about the current direction of the parameters. This makes the optimization difficult, as the loss functions may have many local minima and saddle points or large flat areas. The problem is even more severe because the loss function is calculated with multi-dimensional input data. For functions with multidimensional input, it is necessary to use the *partial derivatives*. Partial derivative represents how the examined function changes concerning each of the input data separately. The critical points in the multidimensional space are located at all points for which the partial derivation is almost zero [4].

*Gradient* of a function  $f$  is a vector field  $\nabla f$ , whose value at point  $p$  is a vector consisting of the partial derivatives of the function  $f$  at point  $p$ .

*Gradient descent* is an iterative, first-order method to optimize the loss function  $J(\Theta)$  by updating the parameters  $\Theta$ . The way the parameters should update is computed with a gradient of the loss function with respect to the parameters. The calculated change is then applied to the parameters of the model. This process is repeated until reaching the optima. The gradient descent method combined with the parameters update is called *backpropagation*.

Gradient descent converges for a problem that can be solved by linear regression. However, neural networks contain nonlinearity caused by their activation functions. Furthermore, the loss functions of the neural network model are usually non-convex, and they are complex functions with many local optima. Gradient descent might end up in some local minima [16]. Another problem is that the gradient descent is relatively slow.

To overcome these problems, the gradient descent algorithm can be modified. Both of these flaws can be reduced by using *stochastic gradient descent* (SGD). In SGD, the gradient is computed and the parameters are updated after every training sample [16], but the gradient is computed using a reduced set of randomly chosen parameters, which reduces the computational complexity and thus accelerates the optimization. The continual update of parameters after each SGD step can help to avoid the critical points [20].

The backpropagation algorithm combined with SGD can be represented as:

$$w_{new} = w_{old} - \eta \nabla L_k(\Theta), \quad (3.1)$$

where  $w_{new}$  is the updated value of weight,  $w_{old}$  is the former value of weight,  $\eta$  is the learning rate, and  $\nabla L_k(\Theta)$  is the stochastic approximation of the gradient of the loss function for the parameters  $\Theta$ . The progress of the SGD is represented in Figure 6.27. This path was obtained during the experiments, which are described later in Section 6.5. It can be seen that the SGD converges into the minima with confidence.

## Chapter 4

# Neural Network Training Progress Visualization

The training of the neural network involves large non-convex optimization problems. This task was believed to be difficult, with a risk of ending up in local optima or saddle points. In this thesis, the training progress is visualized and qualitatively examined.

### 4.1 Linear Path Examination

The trajectory that the SGD follows is complicated and high-dimensional. This visualization uses a linear interpolation of the parameters in order to obtain a cross-section of the loss function along the line, which shows the behavior of the loss function. The method consists of choosing two sets of parameters  $\Theta_0$  and  $\Theta_1$ , calculating the linear interpolation according to the interpolation coefficient  $\alpha$ , and plotting the values of the loss function when given the interpolated set of parameters  $\Theta$ . This method is represented by the following equation:

$$\Theta = (1.0 - \alpha)\Theta_0 + \alpha\Theta_1, \quad (4.1)$$

This method was introduced by Goodfellow and others in [5]. Results presented in the paper were successfully reproduced and extended by examining the loss function behavior on a lower level.

The examination on the level of the parameters unveils the parameters, which have negligible impact on the performance of the model and shows that in, some cases, the parameter optimization is on the edge with updating. The iterative updating of the parameter oscillates around an optimal point. In this thesis is also observed how far has each parameter traveled from its initial value during the training.

Examination on the level of a layer can unveil the *robust* and *ambient* layers. The robust layers have a big impact on the performance of the model, the ambient layers have only a little impact on the performance. The impact is represented by a change in the performance metrics values when the parameters of the examined layer are reinitialized.

## 4.2 Quadratic Interpolation of the Parameters

The Linear Path Examination provides a computationally easy visualization of the training progress on various levels. However, when the validation loss obtained using the interpolated parameters is compared with the actual values of the validation loss measured during the training, the linear path shows as quite inaccurate. In this thesis, I propose a novel approach to the examination of the neural network training progress, the Quadratic Path. This method interpolates the values of the parameters using a second-degree interpolation polynomial.

The chosen method for obtaining the polynomial coefficients is *Lagrange interpolation polynomial*. The Lagrange interpolation polynomial is calculating the value of the interpolated function at a point  $x$ , using a polynomial of  $n$ -th degree. The Lagrange polynomial, the value of the interpolated function, is calculated as a linear combination:

$$P_n(x) = f_i \ell_i(x), \quad (4.2)$$

where  $f_i$  are known values of the interpolated function and  $\ell_i$  are Lagrange basis polynomials.

The degree of the polynomial depends on the number of known points. It can be calculated as  $d = k - 1$ , where  $d$  represents the degree of the polynomial and  $k$  represents the number of known points. To obtain the quadratic equation, the degree of the polynomial has to be  $d = 2$ , so the number of known points has to be  $k = 3$ . The Lagrange polynomial of degree  $d = 2$  can be represented as follows:

$$P_2(x) = f_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + f_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + f_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \quad (4.3)$$

This equation is used in the Quadratic Interpolation method to obtain the values of parameters at various points of the interpolation coefficient  $\alpha$ .

The parameters of the model are changing in an unknown way. Most likely it is not a linear change and thus, the proposed method should output results more faithful to the reality, since it uses a higher degree interpolation.

## 4.3 Loss Function Landscape Visualization

Loss landscape visualization shows the landscape of the loss function around a trained model.

Using one-dimensional visualizing method can be misleading since the non-convexities in the loss function progress are difficult to visualize in one dimensional projection [9]. The following two-dimensional method tries to overcome the limitation of high dimensionality using bidimensional projection of random directions.

The method of visualization of the loss landscape is very computationally demanding compared to the single-dimensional visualization methods. The single-dimensional methods require calculating the validation loss of the neural network model only in one dimension for a chosen number of steps, but the two-dimensional methods require calculating the grid of the validation loss in the desired number of steps. The computational difficulty of the two-dimensional methods is quadratic compared to the single-dimensional methods. The computational burden can be avoided by using a smaller number of steps at the cost of reduced resolution.

Projection in random directions was proposed by Goodfellow et al. in [5]. This method visualizes the loss landscape in two randomly generated directions. After having the directions generated, the grid of validation loss is calculated.

To capture the variation in trajectories, it is necessary to use nonrandom directions. Hao Li et al. in [9] propose using PCA to measure how much variation was captured. The method chooses the two most explanatory directions. Then it creates a grid of the validation loss values around the trained model. The method visualizes a simplified path that the optimization algorithm takes.

The reason to use the PCA directions is that two random vectors in a high dimensional space will be nearly orthogonal with high probability. This can be problematic when the optimization trajectory lies in a low dimensional space. In this case, a randomly chosen vector will lie orthogonal to the low dimensional space of the optimization path and a projection onto random directions will capture almost no variation. Ineffective visualizations using the random directions projection are shown in Figure 4.1 [9].

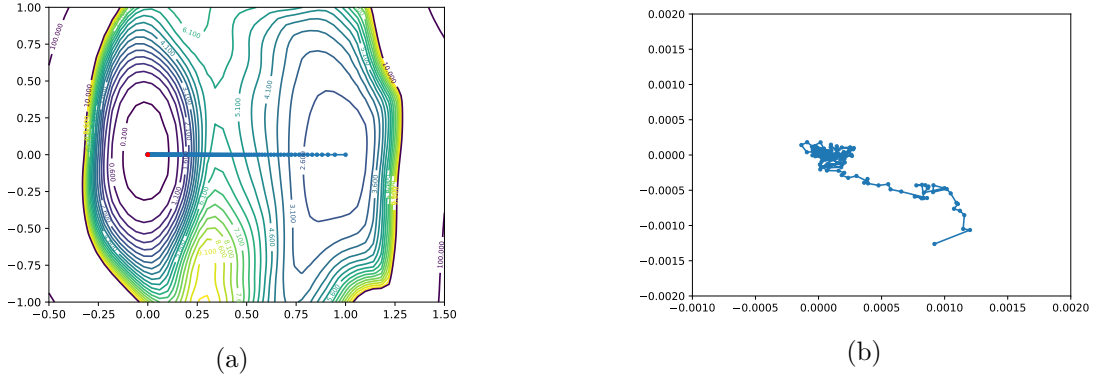


Figure 4.1: Ineffective visualizations of optimizer trajectories [9]. In the Figure 4.1a is visualized a projection of two PCA chosen directions with biggest variance. The Figure 4.1b represents a projection in two random directions. This projection demonstrates only limited variance in comparison with 4.1a.

# Chapter 5

## Implementation

This chapter will present the design of a tool to examine the neural network training progress. It will also, discuss the used technologies and the approach of solution of selected parts of the tool. The tool is available as Python package<sup>1</sup> and the source codes are available on Github<sup>2</sup>.

### 5.1 Design

The goal of the tool is to examine the training progress of a neural network model. It is necessary to provide model-independent examination techniques. The OOP paradigm was chosen to achieve this.

Each type of examination of training progress has a class dedicated to it. One base class for storing the configuration of the experiment exists for both linear and quadratic experiments. A separate class, inheriting from the base class, is created for each single-dimensional experiment. This separation has made the code more readable for use in the future and provides flexibility and independence. The tool can be used on any PyTorch model, when the data loaders for the model are provided. Similarly, surface visualization has its class.

Neural network models typically have a large number of parameters. To be able to examine them, several scripts are provided as an example of the tool. The scripts can be used as base for creating customized scripts, with use of the tool. Each script is designed to execute different experiment. Main script provides an interface for the user. Users can configure the experiment with their own interpolation coefficient, CUDA usage, position of examined parameter and choose of examined layer. The main script provides an option to automatically run all available experiments at once.

### 5.2 Technology

*Python* was chosen as the implementation language for the tool. The decision to use Python as the implementation language was made due to its flexibility. Python is an interpreted high-level programming language, which supports multiple programming paradigms. One of the supported paradigms is object-oriented programming (OOP). The OOP paradigm supports overriding methods of objects. Everything is an object in Python. This property

---

<sup>1</sup><https://pypi.org/project/nervis/>

<sup>2</sup><https://github.com/suzrz/nervis>

allows to perform basic mathematical operations like addition, subtraction, multiplication, and division between different data types. It is very helpful as the tool handles multidimensional structures, called *tensors* and scalars on the other side. In Python, it is possible to use *modules* implemented with C/C++ language which can significantly increase performance.

*PyTorch*<sup>3</sup> is an open-source machine learning library. This library provides optimized work with tensors. Besides the tensor operations implementation, it provides several helper modules. Worth mentioning is the `torch.nn` module which contains a base class for neural network model implementation, providing especially easy work with the parameters of the model. Another great module is `torch.nn.functional` which provides various types of layers, enabling easy composition of a neural network model. The PyTorch enables programming on NVIDIA<sup>4</sup> GPUs through the `torch.cuda` module, which greatly increases the speed of the implemented methods. And finally the `torch.Tensor` module, providing the tensor operations.

Another very useful used modules are *NumPy*<sup>5</sup>, providing various mathematical operations and fast multidimensional arrays handling, *Matplotlib*<sup>6</sup>, a powerful plotting library.

The code for visualizing the loss landscape is inspired by the code provided in a public repository<sup>7</sup> as a supplementary material to the paper authored by Hao Li et al. [9] and *Animating the Optimization Trajectory of Neural Nets* project by Chao Yang<sup>8</sup>.

The tool was implemented with respect to portability. It does not use any system-specific libraries or features. Paths of the files are the only system-specific constraint. This is solved using the `os` module, which detects the operating system during the runtime and behaves according to it. The tool does not require a CUDA GPU to run. It automatically detects whether CUDA is available and if the user has not forbidden the usage of CUDA. If the CUDA cores are not present, the tool will use CPU. Finally, the chosen programming language Python is also a multiplatform technology.

---

<sup>3</sup><https://pytorch.org>

<sup>4</sup><https://developer.nvidia.com/cuda-zone>

<sup>5</sup><https://numpy.org>

<sup>6</sup><https://matplotlib.org>

<sup>7</sup><https://github.com/tomgoldstein/loss-landscape>

<sup>8</sup><https://github.com/logancyang/loss-landscape-anim>

## Chapter 6

# Examining the Training Progress of Neural Network

This thesis focuses on examining the SGD optimization algorithm and the negative log likelihood loss function represented by the following equation:

$$L_i(\Theta) = -\log(p_{y_i}), \quad (6.1)$$

where  $L_i(\Theta)$  is the negative log likelihood,  $i$  represents the  $i$ -th examined data sample, and  $p_{y_i}$  is the likelihood that the examined data sample belongs to one of the classes. The logarithm function is a monotonically increasing function, thus minimizing the negative logarithm likelihood is equivalent to maximizing the probability. The probability  $p_{y_i}$  is computed as follows:

$$p_{y_i} = \frac{\exp(y_i)}{\sum_{n=1}^N \exp(y_n)}, \quad (6.2)$$

where  $y_i$  is output of the model for  $i$ -th data sample and  $N$  is the total number of examined data samples. This is called the *softmax function*. It converts the unnormalized values at the end of the neural network model to normalized probabilities in the interval  $\langle 0; 1 \rangle$ .

The implemented tool, presented in Chapter 5, was successfully used to execute the following experiments and to visualize the results.

### Experimental setup

Experiments were performed on the MNIST dataset<sup>1</sup>. MNIST dataset contains 60000 training samples and 10000 test samples of handwritten digits with labels. The samples are black and white images of original size 28x28 pixels, transformed to 32x32 pixels. The goal of the neural network is to correctly predict what digit is on the input image.

The network architecture I choose to examine contains 5 layers. Model of this network uses the rectified linear unit activation function (*ReLU*). The input layer is a convolutional layer with the size of kernel 3x3. This layer takes as input the 32x32 greyscale image. The second layer of the network is also a convolutional layer with kernel size 3x3. Max pooling is done after each convolutional layer. The following two hidden layers and the output layer are fully connected. The architecture of the examined neural network is based on LeCun's

---

<sup>1</sup><http://yann.lecun.com/exdb/mnist/>



LeNet-5 architecture [8]. LeNet-5 architecture is displayed by Figure 6.1 and the actual architecture is represented in a simplified way by Table 6.1.

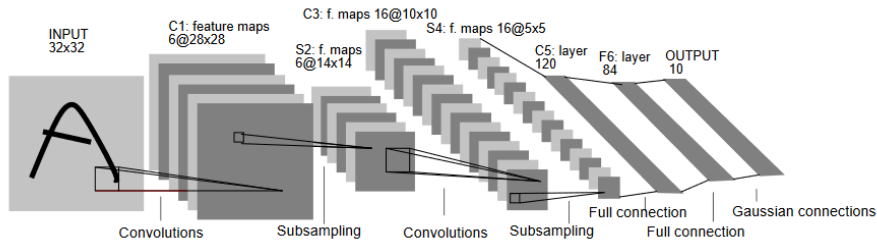


Figure 6.1: Architecture of LeNet-5. Convolutional Neural Network. Each plane is a set of features. [8]

Table 6.1: Architecture of the examined model

Name	Number of Parameters
conv1	60
conv2	880
fc1	69240
fc2	10164
fc3	850

## 6.1 Preliminary experiments

Experimenting with the neural network training process can be time and power-consuming. Because of this fact, I decided to do the preliminary experiments first. They include finding a good number of epochs and sizes of training and test dataset. The goal of this thesis is to examine the training process and not to have the most accurate model. To achieve this goal, it is sufficient to use as minimal datasets and the number of epochs as possible to still have valid results.

### Number of epochs

In this experiment, the model has always the same initial parameters, which are randomly generated before the first run of the experiment. Then the training of the model begins. After executing a certain number of epochs, the performance of the model is measured. Results of them are demonstrated in Figure 6.2.

### Dataset Size Impact

Reducing the size of the datasets could improve the speed of training and validating the model. The goal of this experiment is to find the minimum size of the dataset where results are valid.

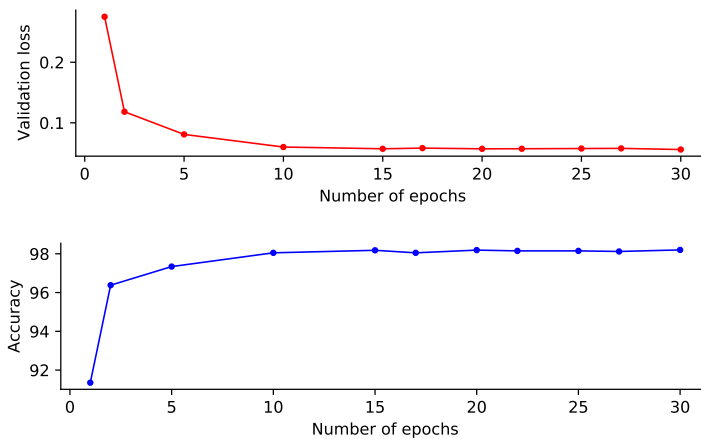


Figure 6.2: Impact of the number of epochs on model’s performance. It is possible to see that 15 epochs are sufficient for the model to perform well on its task. It is also possible to see that with the increasing number of epochs, the performance of the model stops improving rapidly.

For examining the impact of the training dataset on the performance, the model always has the same randomly generated initial parameters. The number of epochs is 15<sup>2</sup>. Samples belonging to the examined subset are randomly chosen from the training dataset. Before training the model with the desired subset size, the model is set to the initial state. Then the model is trained on the training subset and its performance is then evaluated<sup>3</sup>. The final parameters of the model are cleared after evaluating the performance. This process is repeated until selected subsets are examined. The results are shown in Figure 6.3.

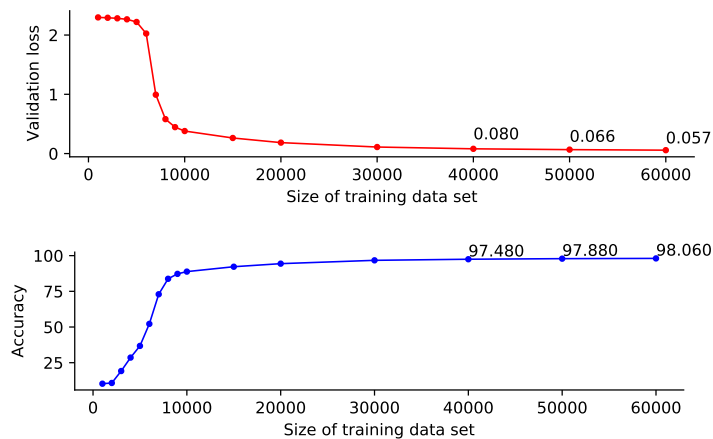


Figure 6.3: Impact of training subset size on the performance of the network. Generally, we can tell that a bigger subset means better results on the performance measuring. Around 10000 training samples, it is possible to see a rapid improvement in the performance of the model.

<sup>2</sup>The number 15 I choose based on the results of the previous experiment.

<sup>3</sup>Performance evaluating is done on the full test dataset.

Second part of experimenting with dataset size is examining the stability of the results of performance measurement with different sizes of the validation subset. This experiment was realized with a model trained on a complete training dataset<sup>4</sup>. The model was not changed during experimenting. The model’s performance was measured 100 times for each examined number of testing samples.

From the results of the preliminary experiments it is possible to see that around 10000 (16.67 %) training samples are enough to see a rapid improvement in the performance of the model. The size of the test data set can be reduced to 8000 (80 %) samples as the Figure 6.4 shows that the results of validation using the reduced dataset are stable enough.

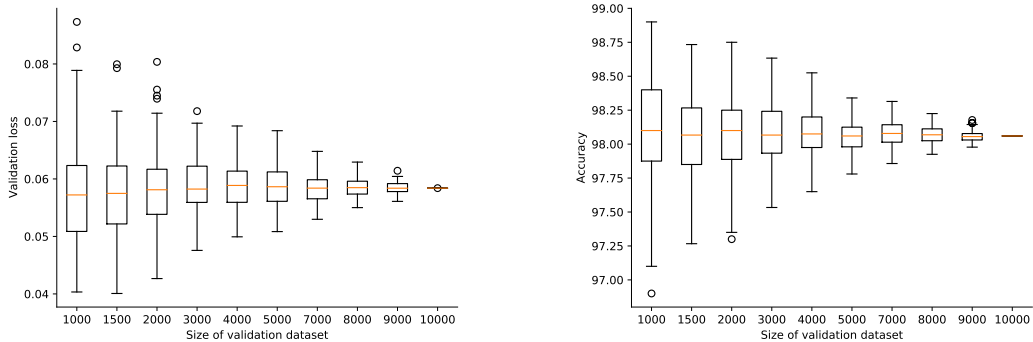


Figure 6.4: Impact of test subset size on stability of network performance measuring. The performance measurement is confident in measured values when testing the model with full validation dataset. Though, for the purpose of this thesis, the measurement is stable enough with 8000 validation samples.

## 6.2 Linear Path Experiment

The method I chose to visualize the training process on the level of layers is the linear path visualization. This method was introduced in [5], but in the paper, the loss surface is examined only on the level of the whole model. In this thesis, the loss function is examined on the level of layers and individual parameters.

The method consists of a linear interpolation between two chosen sets of parameters. The goal is to plot the values of the loss function  $J(\Theta_\alpha)$  along with a series of points for varying values of an interpolation coefficient  $\alpha$ . This method shows a cross-section of the loss function along the calculated line [5]. The parameters are calculated in the following way:

### Linear Path Experiment on the Level of Layers

Individual layers are examined in the first experiment. The following process of the experiment is repeated for each layer individually.

The parameters  $\Theta_i$  are obtained before the training of the model and set as  $\Theta_0$  for the interpolation. The parameters  $\Theta_f$  are obtained after the last training epoch of the model and set as  $\Theta_1$  for the interpolation. Then the parameters  $\Theta_\alpha$  are obtained as a result of

<sup>4</sup>60000 training samples

the interpolation for the value of the interpolation coefficient  $\alpha$ . The parameters of the examined layer are replacing their equivalent in the parameters loaded in the model. The performance of the model is measured after each interpolation step.

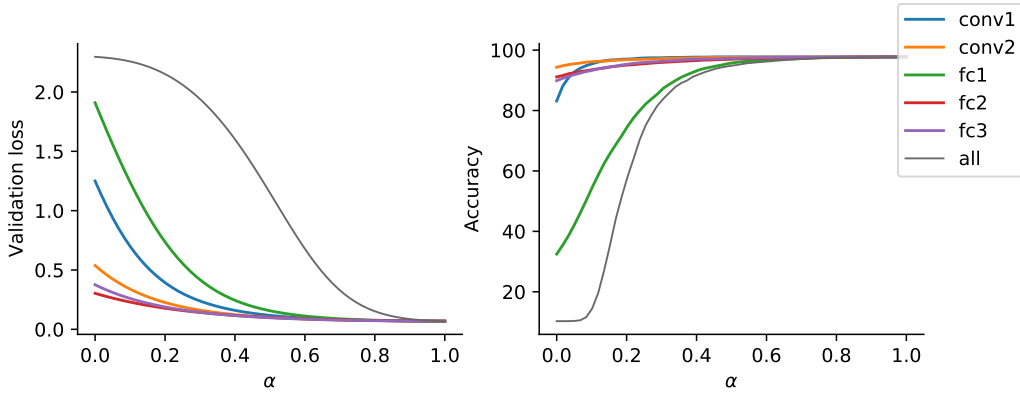


Figure 6.5: **Linear interpolation of parameters in layers.** In this figure is demonstrated a summary of the results of the experiment. It is evident that the first convolutional layer has the biggest impact on the final performance of the model. The second most important layer is the first convolutional layer.

The result of the experiment presented in Figure 6.5 uncovers robust and ambient layers. The biggest impact on the validation loss as well as on the accuracy has the first fully-connected layer **fc1**. This layer can be classified as robust with confidence, it is a critical layer for the model to make accurate predictions. Changing its parameters to the initial state while having others in the trained state caused the model to make more mistakes than correct predictions. The second most important layer is the first convolutional layer **conv1**. The change of the parameters of the **conv1** layer causes a deterioration in accuracy until the interpolation coefficient reaches a value around  $\alpha = 0.1$ , where the accuracy of the model acquires its almost final value. The accuracy in this place even overtakes the accuracies measured with the parameters of other layers in a modified state. The **conv1** layer can also be classified as a robust layer. The other layers also have an impact on the performance of the model, but not as big as **fc1** and **conv1** have. According to the observations, the other layers could be classified as ambient layers.

### Linear Path Experiment on the Level of Parameters

The second experiment examined the individual parameters of the model. Chosen parameters are examined separately and the individual interpolations are not influenced by each other. This experiment is repeated for each chosen parameter as follows.

The parameters  $\Theta_i$  are obtained before the training of the model and set as  $\Theta_0$  for the interpolation, the parameters  $\Theta_f$  are obtained after the training of the model and are set as  $\Theta_1$  for the interpolation. Final parameters  $\Theta_f$  are loaded into the model and thus it is set to its trained state. Then one chosen parameter is interpolated according to (4.1) and the interpolated value replaces the value of this parameter in the parameters loaded in the model. The performance of the model is evaluated after each interpolation step.

In addition, the distance between the initial and final parameters is measured during this experiment. The distance is measured using the Euclidean demonstrated for  $n$ -dimensional vectors by (6.3).

$$d(\Theta_i, \Theta_f) = \sqrt{(\Theta_i^{(1)} - \Theta_f^{(1)})^2 + (\Theta_i^{(2)} - \Theta_f^{(2)})^2 + \dots + (\Theta_i^{(n)} - \Theta_f^{(n)})^2} \quad (6.3)$$

The result of the experiment executed on the convolutional layers is shown in Figure 6.6. Both of the layers have a perceptible impact on the final value of the loss function. The parameters of the second convolutional layer have a similar effect on the loss function as the parameters of the first convolutional layer, despite that the parameters of `conv1` have traveled more distance during the training. The loss function progress with the modified parameter set in both convolutional layers is corresponding to the shape of the negative log-likelihood.

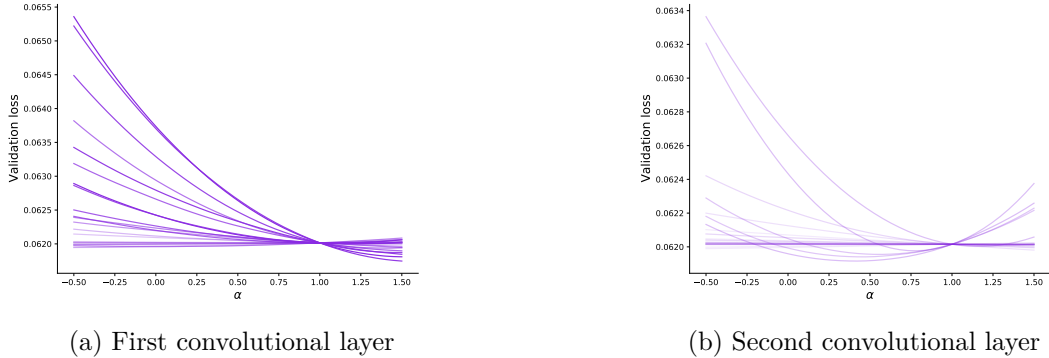
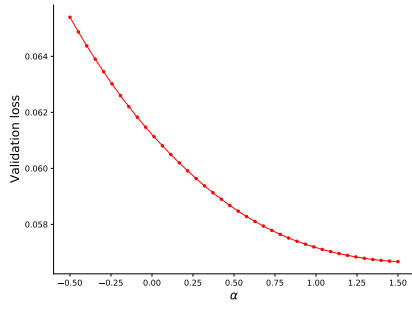


Figure 6.6: **Linear interpolation of individual parameters.** The figure represents the values of the loss function when one parameter is interpolated from first and second convolutional layer. The lines intersect at  $\alpha = 1$ , where the parameters loaded in the model are according to the  $\Theta_f$  parameters. The opacity of the line shows how much distance has the exact parameter traveled. The more opaque, the more distance the parameter has traveled.

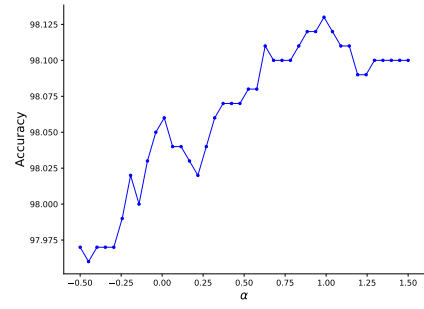
It is also possible to observe how big is the distance that the parameter has traveled from its initial state to its final state. This can indicate how important the exact parameter is. The bigger the distance, the bigger the impact on the final validation loss. However, this can be misleading since the initialization of parameters is random. In this context, the accuracy can also be observed as a similar metric. When the accuracy does not change much, then we know that this exact parameter is not much important.

Figure 6.7 provides a closer look on the chosen parameters. The good behaving progresses of the loss function are presented on the first two rows. Modifying the parameter has affected the performance of the model perceptibly and the interpolation of this parameter from the initial value to the final was accompanied by the well-behaved loss function. The accuracy metric is more unstable when changing the parameters, this could indicate that the parameters are critical for the network to make an accurate prediction. The loss function for the other two parameters is clearly much more unstable. The accuracies of these parameters are not changing, both of these observations could indicate that these two parameters are less significant for the network than the first two.

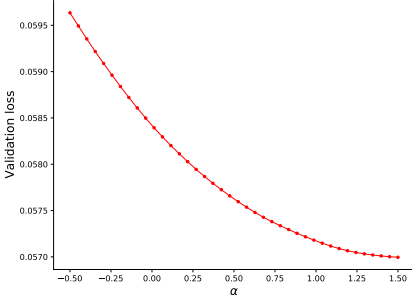
Examination of the parameters in fully connected layers represented by Figure 6.8 shows that individually the parameters of these layers have negligible impact on the final validation loss. The majority of them have small changes between their initial value and their final value. The small change in their values could be explained by the broadness of these layers.



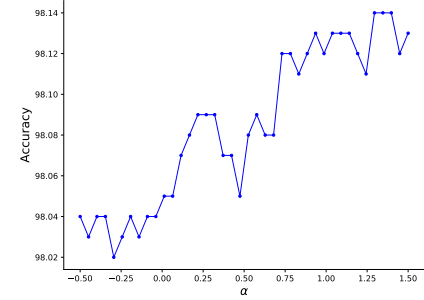
(a)



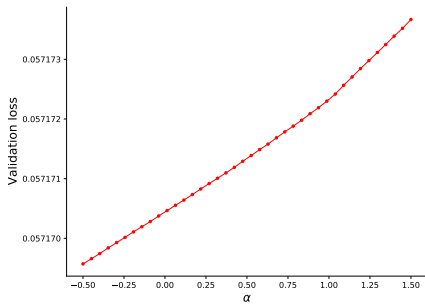
(b)



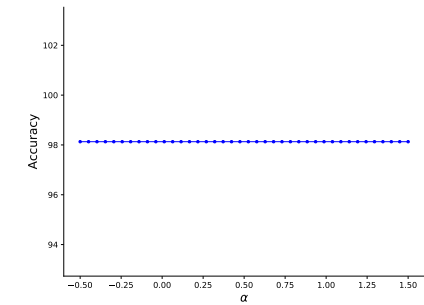
(c)



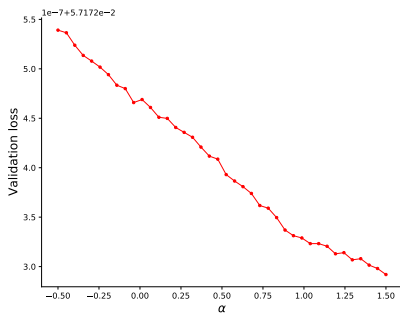
(d)



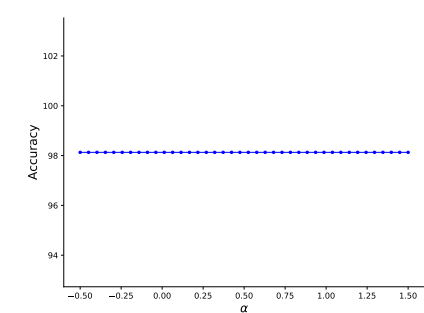
(e)



(f)

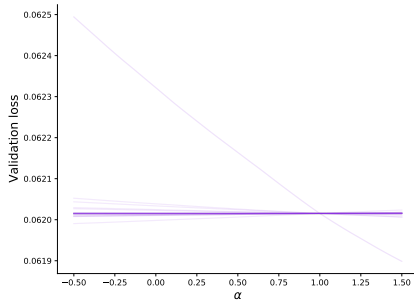


(g)

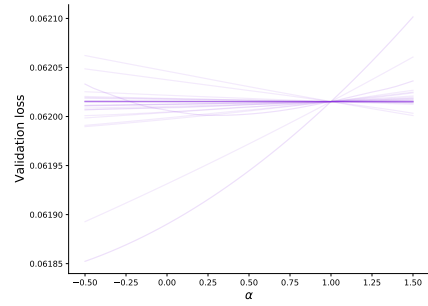


(h)

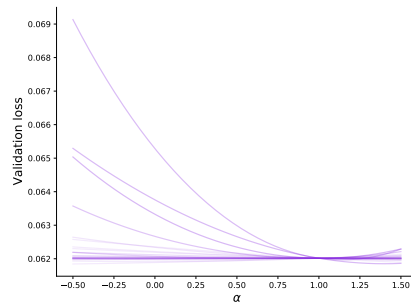
Figure 6.7: Linear Interpolation of the Parameters of the First Convolutional Layer. Validation loss is on the left side, accuracy is on the right



(a) First fully connected layer



(b) Second fully connected layer

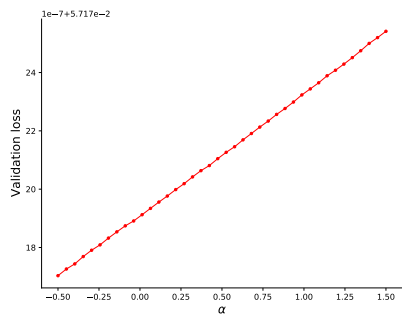


(c) Third fully connected layer

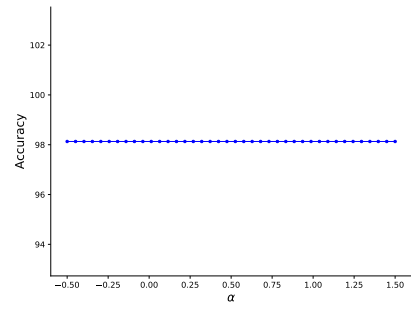
Figure 6.8: Linear Interpolation of Parameters of the Fully Connected Layers. The figure represents a summary of the results of the experiment on fully connected layers. The parameters of the fully connected layers generally did not change much and their impact on the validation loss is negligible.

Those which are fully connected are generally broader than the convolutional layers and as a whole unit, they have an essential effect on the neural network model.

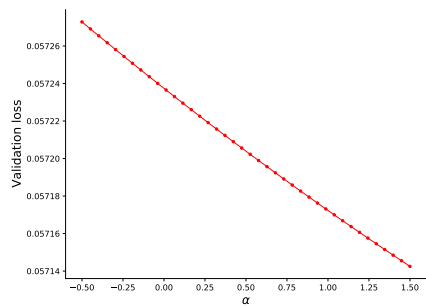
Figure 6.9 provides a closer look on the parameters of the last layer. The lowest value of the loss function is in the initial state for the first chosen parameter, represented by Figure 6.9a. This can happen as the network works as a decentralized computational model. The validation loss can be lower for this parameter, but for the whole set of parameters it can be bigger. The optimization algorithm of the model never reaches this place because of the high validation loss value when having all parameters in the same state. The parameter has relatively little impact on the accuracy of the model. The second chosen parameter, represented in Figures 6.9c and 6.9d, shows that the training progress is a simple line when the neural network model knows the initial and final values of the parameters. However, the change in the validation loss is so small that the impact on the accuracy of the model is immeasurable on this level of precision. The third parameter affects the loss function wildly but the change is small. The behavior of this parameter during the training can be considered as noise or that the optimization algorithm is on the edge with its capabilities for this parameter, it could not decide in which way it should update the parameter. The impact on the accuracy is immeasurable. The last chosen parameter has similar behavior of the loss function as the first chosen parameter, but it affects the validation loss more moderately as the values of the validation loss are smaller and the accuracy of the model is not affected.



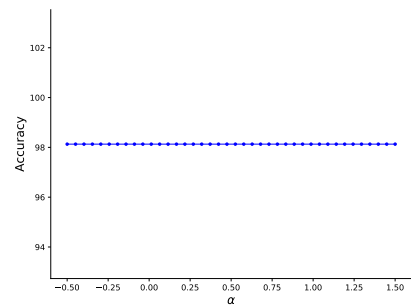
(a)



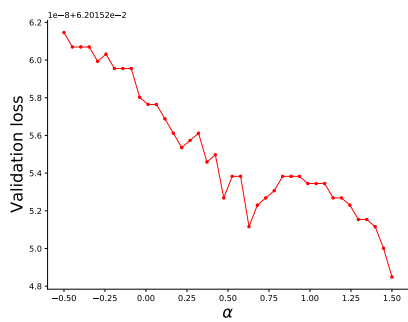
(b)



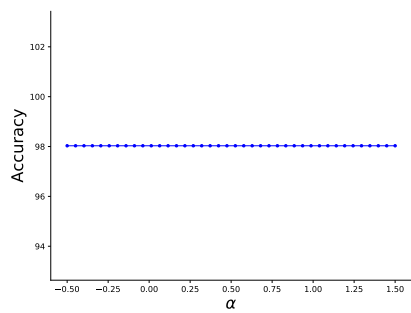
(c)



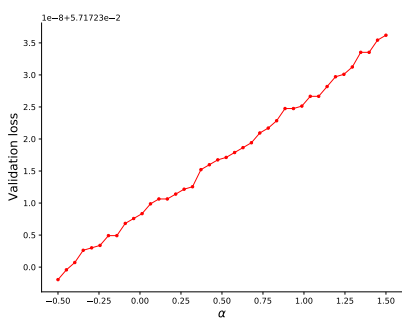
(d)



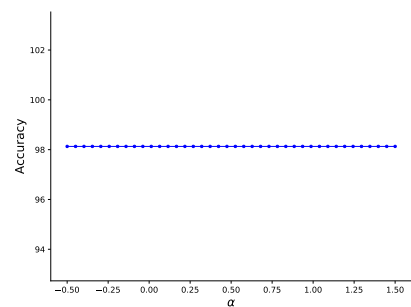
(e)



(f)



(g)



(h)

Figure 6.9: Linear Interpolation of Parameters of the Last Fully Connected Layer.



### 6.3 Quadratic Path Experiment

The proposed method to visualize the training progress more accurately is to interpolate the parameters of the network using the intersection with a curve. The value of the parameters is calculated using the Lagrange interpolation coefficient of degree  $d = 2$ . The calculation of the parameters at a series of points of interpolation coefficient  $\alpha$  is done as it is represented in (4.3). This method should be more accurate than the Linear Path experiment as it uses a higher degree of interpolation. However, it is still very little computationally demanding.

It is necessary to choose three points to perform the quadratic interpolation of the parameters. Following data are used in this case:

Table 6.2: Known Data Coordinates

Point	Parameters	$\alpha$
$x_0$	$\Theta_{initial}$	0
$x_1$	$\Theta_{mid}$	0.5
$x_2$	$\Theta_{final}$	1

These points are used for creating the Lagrange interpolation coefficient of the second degree represented by Equation 4.3. With this interpolation polynomial, it is possible to calculate the values of parameters at each point  $\alpha$ .

#### Quadratic Path Experiment on the Level of Layers

This method examines the neural network training progress on the level of layers. It is repeated for each layer in the same way, based on the following template.

First, the initial parameters  $\Theta_i$  are obtained before the model sees any data. Then, the model is trained, when the model is in half of the training progress, the parameters of the mid point are obtained as  $\Theta_m$ . Finally, the final parameters  $\Theta_f$  of the model are obtained after the model finishes its training progress. The known points for the quadratic interpolation of the layer parameters are following:

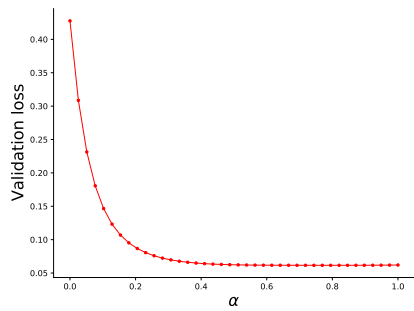
Table 6.3: Known Data for Examining the Layers

x-axis	0	0.5	1
y-axis	$\Theta_i^{layer}$	$\Theta_m^{layer}$	$\Theta_f^{layer}$

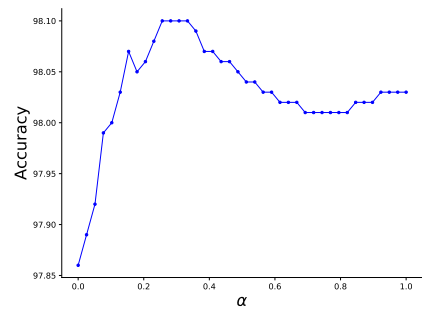
After obtaining the known parameters, the Lagrange polynomial of second degree, presented in 4.3, is constructed with  $\Theta_i^{layer}$  as  $f_0$ ,  $\Theta_m^{layer}$  as  $f_1$  and  $\Theta_f^{layer}$  as  $f_2$ .

The constructed polynomial is then used to calculate the value of parameters for each interpolation coefficient  $\alpha$ . The model has loaded its final parameters, but the examined parameters  $\Theta_f^{layer}$  of the examined layer are replaced with the interpolated values  $\Theta_\alpha^{layer}$  for each interpolation step. The performance of the model is evaluated after each interpolation step. Both the *weights* and *biases* of the layers are interpolated.

The quadratic path on the level of layers supports the results of the linear path experiment from Section 6.2, but it does represent the progress of the training more faithfully to the reality. The results are presented in the following Figures.

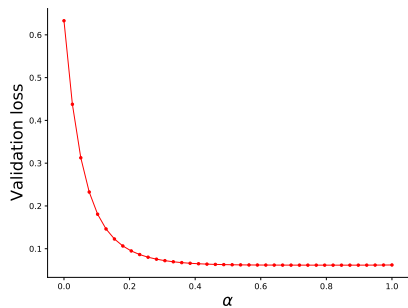


(a)

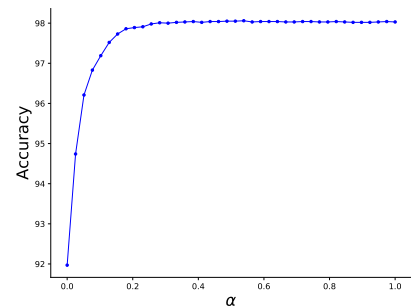


(b)

Figure 6.10: First Convolutional Layer

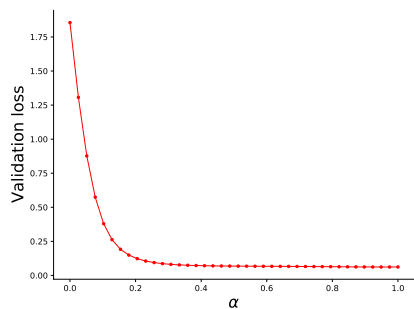


(a)

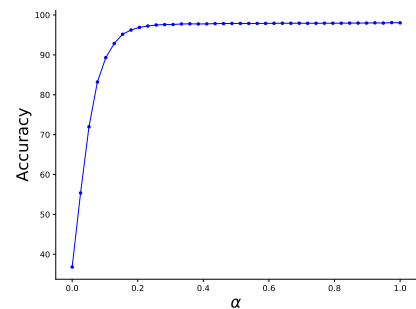


(b)

Figure 6.11: Second Convolutional Layer

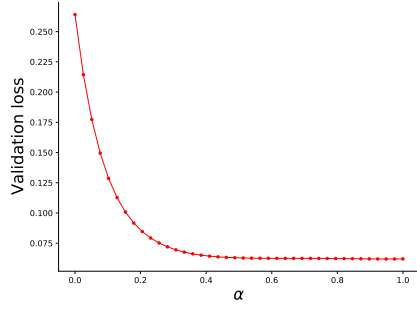


(a)

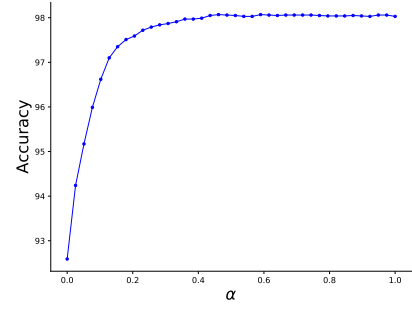


(b)

Figure 6.12: First Fully Connected Layer

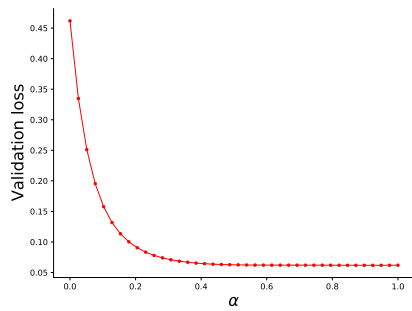


(a)

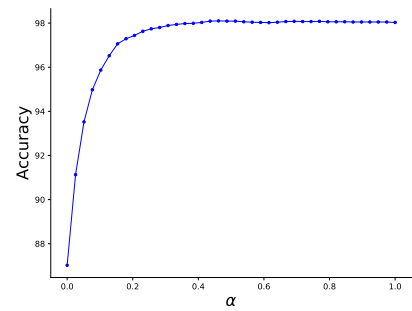


(b)

Figure 6.13: Second Fully Connected Layer



(a)



(b)

Figure 6.14: Third Fully Connected Layer

## Quadratic Path Experiment on the Level of Parameters

The examination of the neural network training progress on the level of the parameters is done for an interpolation coefficient in the range:  $\alpha \in \langle -0.5; 1.5 \rangle$ . The range was chosen to visualize what could happen if the training would proceed.

The experiment is done similarly to the quadratic interpolation of the parameters on the level of layers. The initial parameters  $\Theta_i$ , mid parameters  $\Theta_m$  and final parameters  $\Theta_f$  are obtained. The known points for this experiment are following:

Table 6.4: Known Data for Examining the Parameters

x-axis	0	0.5	1
y-axis	$\Theta_i^{\text{param}}$	$\Theta_m^{\text{param}}$	$\Theta_f^{\text{param}}$

The Lagrange interpolation polynomial is constructed similarly as is for the quadratic path on the level of layers, using the Equation 4.3 and substituting functional values  $\Theta_i^{\text{param}}$  as  $f_0$ ,  $\Theta_m^{\text{param}}$  as  $f_1$  and  $\Theta_f^{\text{param}}$  as  $f_2$ .

Similarly to the previous experiment, the final parameters are loaded into the model. Then during each interpolation step, the interpolated parameters  $\Theta_\alpha^{\text{param}}$  are replacing its equivalent in the parameters loaded in the model. After each interpolation step, the performance of the model is evaluated.

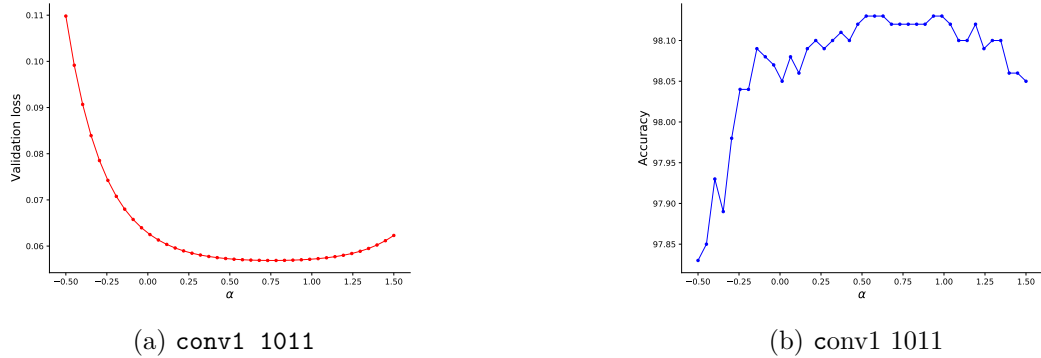
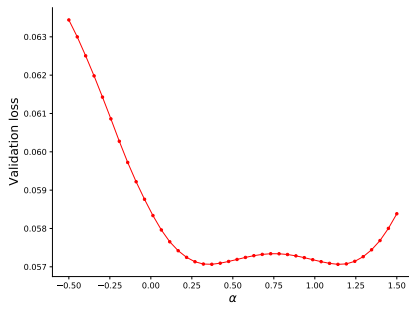


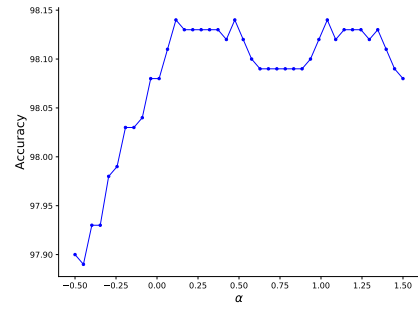
Figure 6.15: Quadratic interpolation of individual parameter of first convolutional layer.

The results are corresponding to the results of the linear path experiment for individual parameters. It can be observed that the quadratic interpolation has more smooth progress. The quadratic path gives similar results for certain patterns of the linear path experiment results. The proposed method diverges from the results of the linear path experiment when extrapolating the parameters.

The results of the experiment executed on the first convolutional layer are presented in Figure 6.15. They show that in the first convolutional layer the parameters are behaving well. The loss function progress has a nice convex shape and for all chosen parameters it has a similar pattern. The accuracies are more interesting in this case. As in the linear path experiment, the change in the accuracy is negligible. When the accuracy does not change much and long flat areas can be observed, the parameter has little impact on the final performance of the model.



(a) conv2 1102

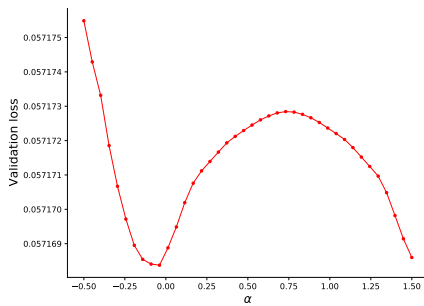


(b) conv2 1102

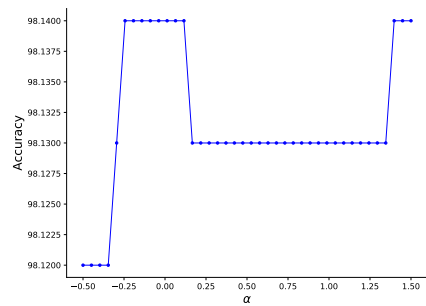
Figure 6.16: Quadratic interpolation of individual parameter of second convolutional layer.

The progress of the training visualized in the parameters of the second convolutional layer is very smooth. The results are shown in Figure 6.16.

The parameters of the first fully connected layer are expected to have a bigger impact on the final performance than the parameters of other layers have. This expectation is based on the results of the layer examination experiments. A closer look at the parameters shows the opposite. Individual parameters of the most important layer of the model have little impact on the final performance. The following results agree and support the conclusion of the linear path experiment executed on the first fully connected layer. The quadratic interpolation of parameters of the first fully connected layer is shown in Figure 6.17



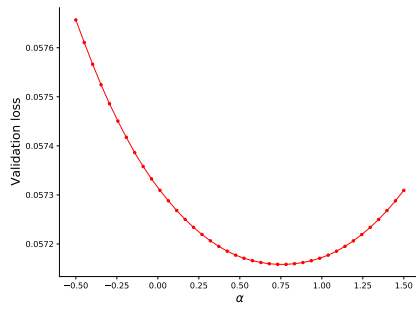
(a) fc1 112265



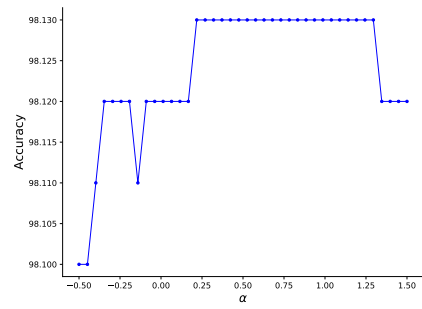
(b) fc1 112265

Figure 6.17: Quadratic interpolation of individual parameter of the first fully connected layer.

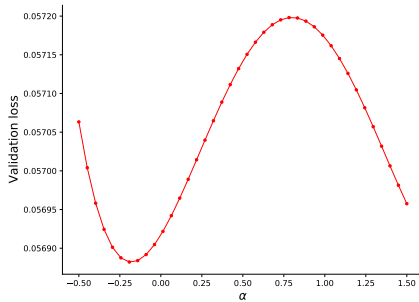
The training progress visualized in the parameters of the second and last fully connected layers is similar to the visualization on the first fully connected layer. The individual parameters have little impact on the final performance as seen in Figure 6.18.



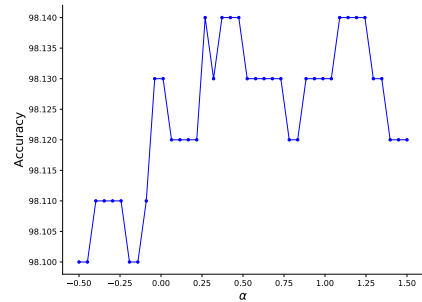
(a) fc2 4772



(b) fc2 4772



(c) fc3 0357



(d) fc3 0357

Figure 6.18: Quadratic interpolation of the parameters of second and third (last) fully connected layers.

## 6.4 Comparison Between the Quadratic and Linear Path

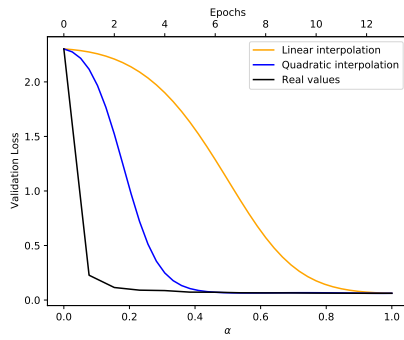
The comparison between the linear and quadratic path methods provides a glimpse on the differences between them.

The comparison was performed on one model. In the visualization, the linear and quadratic paths are displayed in one graph. The graph axes are shared between the two methods. Each figure represents a comparison between linear path and quadratic path examination methods.

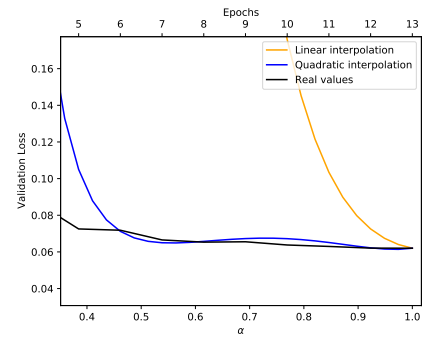
The linear path method shows that the SGD algorithm takes a more steep path than the linear path. Based on this observation, the steepness and shape of the quadratic path was examined.

On the level of the whole model, the quadratic path definitely performs better than the linear path examination method. The Quadratic Path is more faithful to the real validation progress. The SGD in real progress minimizes the validation loss after the first training epoch. Both of the linear and quadratic paths need more time to reach similar values of the validation loss. The linear path intersects the actual path at the end of the interpolation. The quadratic path intersects the actual path much sooner, around the interpolation coefficient  $\alpha = 0.5$ . Which is approximately two times faster than the linear path and thus the quadratic path provides a more accurate visualization of the actual training progress. The results are shown in Figure 6.19.

Several patterns can be identified at the level of parameters. The patterns always depend on both of the training progress visualization methods. The comparison on the



(a) Comparison of the methods.



(b) Detail of the intersection of the methods.

Figure 6.19: Neural network training progress examination methods in comparison to real values of the validation loss. The bottom x-axis represents the progress of the interpolation. The top x-axis represents the progress of the actual training.

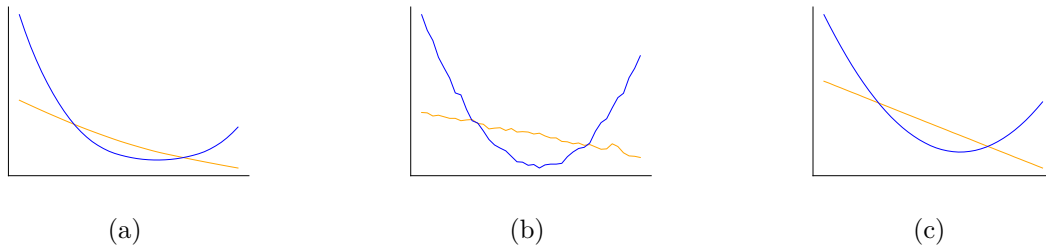


Figure 6.20: Monotonically Descending Linear – Convex Quadratic. When the linear path is monotonically descending, the quadratic path has convex shape. The orange curve is the linear path progress visualization and the blue curve is the quadratic path progress visualization.

level of parameters was done on  $\alpha \in \langle -0.5; 1.5 \rangle$  of the interpolation coefficient  $\alpha$ . The ticks and labels of the visualizations were omitted for the sake of readability. The main purpose of the figures is to show the identified patterns.

Pattern identified as *monotonically descending linear – convex quadratic* is shown in Figure 6.20. When the linear path is monotonically descending, then the quadratic path is convex. Visualization in Figure 6.20b represents a parameter whose value does not change much during the training. The noise that can be observed is caused by an oscillation of the parameter value around an optimum point.

Another observed pattern, identified as *monotonically ascending linear – concave quadratic* is presented in Figure 6.21. The concave shape of the quadratic interpolation is caused by the parameters of the *middle* known point are corresponding to a higher value of validation loss than the parameters of the *end* known point.

A flat region pattern can be observed in Figure 6.22. This pattern appears when the linear path has a convex shape, but the change in the validation loss value is little. The quadratic path contains a flat region due to the relatively fast descent at the start but a small change overall. This pattern can be linked to the *bump in quadratic* pattern, described in the following paragraph.

In Figure 6.23 can be observed that the quadratic path does a little bump when the linear path is convex. The quadratic interpolation reacts too fast and too radically with

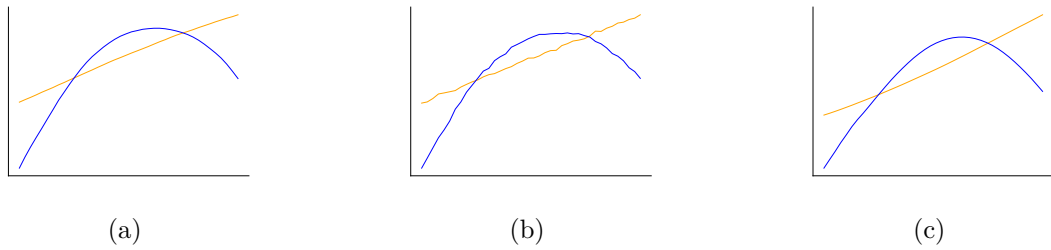


Figure 6.21: Monotonically Ascending Linear – Concave Quadratic. The orange curve represents the linear path, the blue curve represents the quadratic path.

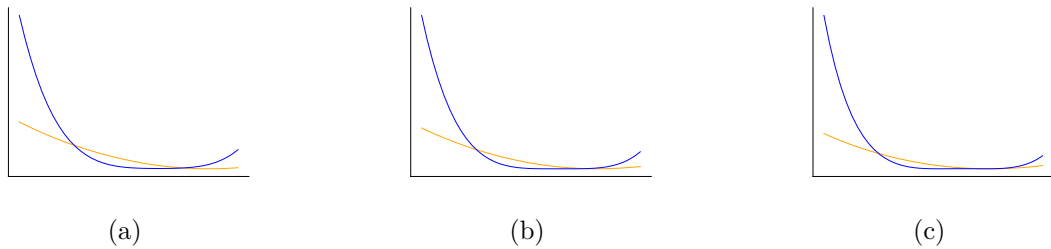


Figure 6.22: Slightly Convex Linear – Flat Region in Quadratic. The orange curve represents the linear path, the blue curve represents the quadratic path.

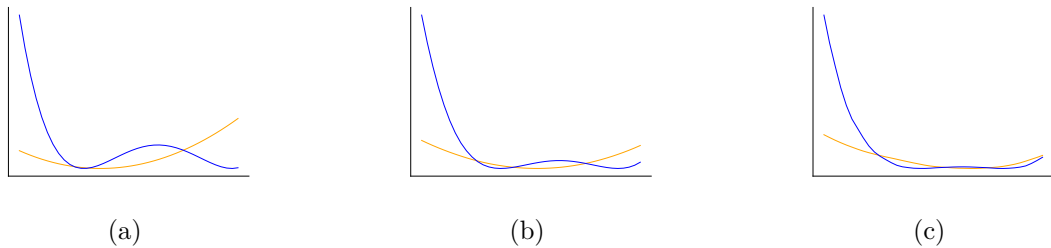


Figure 6.23: Convex Linear – Bump in Quadratic. The orange curve represents the linear path, the blue curve represents the quadratic path.

little changes. This causes the bump to appear. This pattern can be linked to the previous *flat region* pattern. The difference is that in the *flat region* pattern, the linear path is monotonically descending, but in the *bump* pattern, the linear path starts to ascend before it reaches  $\alpha = 1.0$  value of the interpolation coefficient. The detail of the difference is shown in Figure 6.24.

The last identified pattern *Rapidly Ascending Linear – Wave in Quadratic* is shown in Figure 6.25. A wave appears in the quadratic path visualization when the validation loss is growing rapidly. As mentioned before, the quadratic interpolation, through which the parameters of the quadratic path are obtained, reacts too fast when a strong impulse appears. Because of that, the quadratic path has to dramatically change its course when approaching the *end* known point.

Generally, it can be said that the linear path provides more stable results when the interpolation coefficient gets behind the range of known points between  $\langle 0; 1 \rangle$ . The quadratic path starts to *run* too far away. The quadratic interpolation of the parameters when the



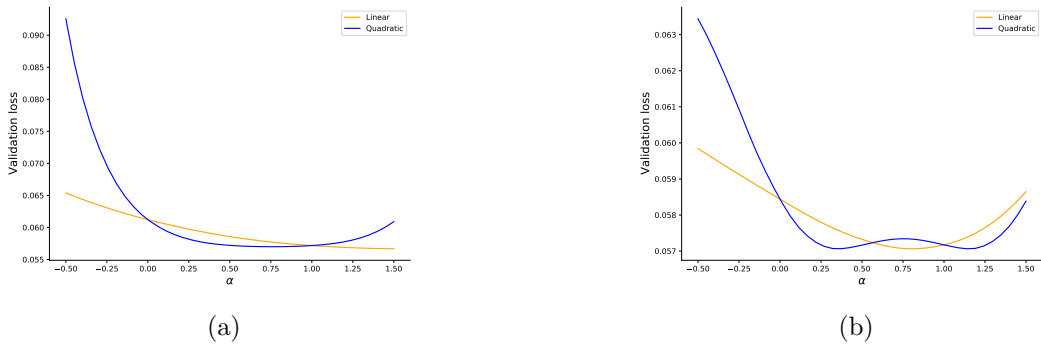


Figure 6.24: Difference between the *flat region* and *bump* patterns.

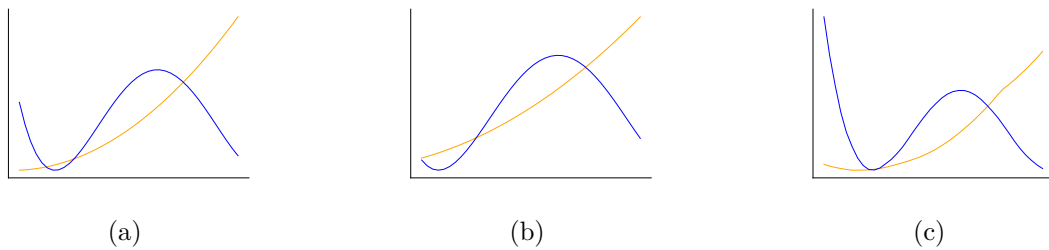


Figure 6.25: Ascending Convex Linear – Wave in Quadratic. The orange curve represents the linear path, the blue curve represents the quadratic path.

interpolation coefficient is  $\alpha \notin \langle 0; 1 \rangle$  takes bigger steps than the linear interpolation at the same values of  $\alpha$  and thus the values of the validation loss are corresponding to different parameters.

The quadratic path method provides results more faithful to the reality, but the method has limitations. The quadratic interpolation reacts too hastily to little changes and starts to „running away“ when the interpolation goes behind the known points. However, when the quadratic path is used for visualization of the training progress inside the range constrained by known points, it does provide more accurate results than the linear path.

## 6.5 Loss Function Surface Visualization

The visualization of the loss function landscape provides a look at the shape of the validation loss around a trained model. The visualization is done by the projection of two random directions. This method provides information about the complexity of the training progress of the model.

To achieve this, two random directions are chosen to create a projection of the loss landscape. Then a *loss grid* is created, which has its center point at the trained state of the model. The surroundings are computed with move around the starting point in random directions with a step size calculated from the resolution.

As discussed in Section 4.3, the projection in the random directions is not ideal for visualizing the path of the optimizer. Different examination method is used for this reason. This method chooses the two most explanatory directions from the multidimensional space, using the principal component analysis. Then the loss landscape around the trained model

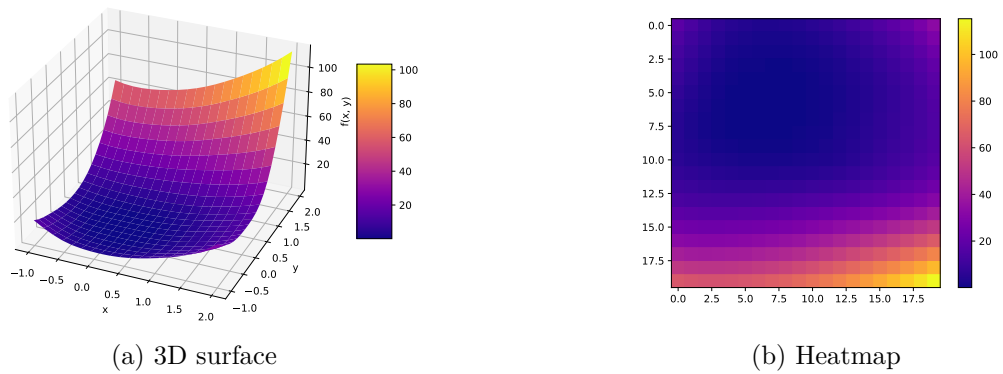


Figure 6.26: Visualization of the loss landscape of simple convolutional neural network model.

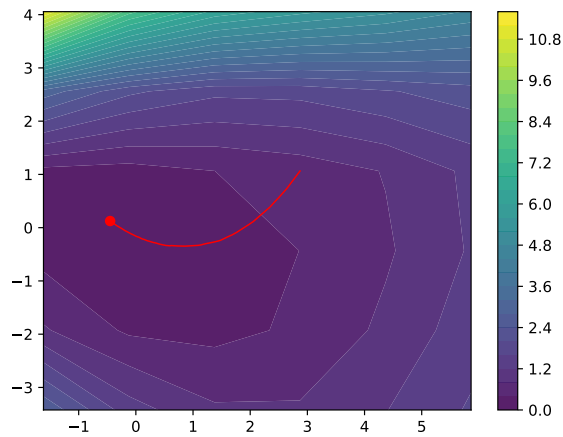


Figure 6.27: Visualization of SGD path on the simple CNN model.

is created, the landscape is visualized up to the initial point of the optimization of the loss function. Finally, the path that the optimizer chooses is visualized in the loss landscape.

The result of the visualization is presented in Figure 6.27. The SGD optimizing algorithm converges to the minima with confidence for the simple CNN model.

# Chapter 7

## Conclusion

The goal of this thesis was to visualize and examine the training progress of neural networks. The training progress is highly computational demanding task and some of its parts are not fully examined. The proposed tool in this thesis provides a number of various methods to visualize the progress. The visualization makes possible to examine the training progress on the level of whole model, layers and individual parameter.

The visualization of the training progress on the level of the model using the linear path examination was successfully reproduced. The results of this experiment show that if the model knew the final parameters, a simple line would do a good job of training. The linear path on the level of layers has revealed and identified the robust and ambient layers. This identification could provide a glimpse of the effectiveness of the examined architecture of a neural network model. The experiment, executed on the level of parameters, provides a detailed look at the training progress on the individual parameters of the model. This can help to identify fast and slow-paced changes in parameters.

The proposed quadratic path experiment enhances the linear path experiment. It provides results that are more faithful to the reality. Especially on the level of the whole model and parameters can be seen the difference in the accuracy of these two methods. This can unveil details about the training progress with bigger confidence. The results of this experiment are corresponding to the results of the linear path experiment.

The loss function surface was visualized using both random directions and the PCA directions. The random directions projection provides a look at the complexity of the neural network model and its training. The PCA directions projection allows visualizing the path that the optimizer algorithm takes during the training.

The implemented tool, providing the experiments, is published under MIT license on Github and as a Python Package on PyPi.

The results of this work were successfully presented at Excel@FIT conference and the paper<sup>1</sup> presented at the conference was awarded by experts committee.

---

<sup>1</sup><http://excel.fit.vutbr.cz/submissions/2021/021/21.pdf>

# Bibliography

- [1] ABDI, H. and WILLIAMS, L. J. Principal component analysis. *WIREs Comp Stat.* 1st ed. 2010, vol. 2, no. 4, p. 433–459. DOI: 10.1002/wics.101.
- [2] DAUPHIN, Y. N., PASCANU, R., GÜLÇEHRE, Ç., CHO, K., GANGULI, S. et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR.* 0th ed. 2014, abs/1406.2572, -. Available at: <http://arxiv.org/abs/1406.2572>.
- [3] EICKENBERG, M., GRAMFORT, A., VAROQUAUX, G. and THIRION, B. Seeing it all: Convolutional network layers map the function of the human visual system. *NeuroImage.* 1st ed. october 2016, vol. 152, -. DOI: 10.1016/j.neuroimage.2016.10.001. ISSN 1053-8119.
- [4] GOODFELLOW, I., BENGIO, Y. and COURVILLE, A. *Deep Learning.* 1st ed. MIT Press, 2016. ISBN 978-0-262-33743-4. Available at: <http://www.deeplearningbook.org>.
- [5] GOODFELLOW, I. J., VINYALS, O. and SAXE, A. M. *Qualitatively characterizing neural network optimization problems.* 2015. Available at: <https://arxiv.org/abs/1412.6544>.
- [6] HEBB, D. O. *The organization of behavior: a neuropsychological theory.* 1st ed. J. Wiley; Chapman & Hall, 1949. ISBN 978-0805843002.
- [7] IM, D. J., TAO, M. and BRANSON, K. *An empirical analysis of the optimization of deep network loss surfaces.* 2017.
- [8] LECUN, Y., BOTTOU, L., BENGIO, Y. and HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE.* 1st ed. Ieee. 1998, vol. 86, no. 11, p. 2278–2324.
- [9] LI, H., XU, Z., TAYLOR, G., STUDER, C. and GOLDSTEIN, T. *Visualizing the Loss Landscape of Neural Nets.* 2018. Available at: <https://proceedings.neurips.cc/paper/2018/file/a41b3bb3e6b050b6c9067c67f663b915-Paper.pdf>.
- [10] LIN, G. and SHEN, W. Research on convolutional neural network based on improved Relu piecewise activation function. *Procedia Computer Science.* 1st ed. january 2018, vol. 131, C, p. 977–984. DOI: 10.1016/j.procs.2018.04.239.
- [11] LIVNI, R., SHALEV SHWARTZ, S. and SHAMIR, O. *On the Computational Efficiency of Training Neural Networks.* 2014.

- [12] MCCULLOCH, W. S. and PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*. 1st ed. Springer. 1943, vol. 5, no. 4, p. 115–133.
- [13] MINSKY, M. and PAPERT, S. *Perceptrons: an introduction to computational geometry*. 1st ed. M.I.T. Press, 1969. ISBN 0-262-13043-2.
- [14] MITCHELL, T. M. *Machine Learning*. 1st ed. Boston: McGraw-Hill, 1997. ISBN 0-07-042807-7.
- [15] O’SHEA, K. and NASH, R. *An Introduction to Convolutional Neural Networks*. 2015. Available at: <https://arxiv.org/abs/1511.08458>.
- [16] PATTERSON, J. and GIBSON, A. *Deep Learning: a practitioner’s approach*. 1st ed. O’Reilly, 2017. ISBN 978-1-491-91425-0.
- [17] RAMACHANDRAN, P., ZOPH, B. and LE, Q. V. *Searching for Activation Functions*. 2017.
- [18] ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*. 1st ed. American Psychological Association. 1958, vol. 65, no. 6, p. 386.
- [19] SKANSI, S. *Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence*. 1st ed. Springer, 2018. ISBN 978-3-319-73004-2.
- [20] SRA, S., NOWOZIN, S. and WRIGHT, S. *Optimization for Machine Learning*. 1st ed. MIT Press, 2012. Neural information processing series. ISBN 9780262016469.