VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Egor Filonov

IMPLEMENTATION OF SOFTWARE FOR MACHINE LEARNING AND DEEP LEARNING

Technology and Communication
2021

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information technology

ABSTRACT

Author	Egor Filonov
Title	Implementation of Software for Machine Learning and Deep Learning
Year	2021
Language	English
Pages	55
Name of Supervisor	Anna-Kaisa Saari, Alex Jung

The thesis studied how to create software for machine learning and deep learning. It also examined what the main components are in this kind of program and how are they implemented.

The central concepts that appear throughout the study are machine learning and deep learning. These abstractions are discussed in more detail during the explanation of the implementation process. Mathematical concepts, such as matrices and optimization algorithms, were used during the research. Moreover, topics from computer science, such as computation graphs, were applied to provide a better understanding of the final algorithm.

The results showed that the implementation of the software for machine learning and deep learning relies on the "training" algorithm and may vary depending on the type of problem approached with the software.

Keywords Machine learning, deep learning, software library, C++

CONTENTS

LIST OF FIGURES AND TABLES

1 INTRODUCTION

Deep learning, as it is primarily used, is essentially a statistical technique for classifying patterns, based on sample data, using neural networks with multiple layers. The first algorithm of deep network was published by Alexey Ivakhnenko and Lapa in 1967 (Ivachnenko & Lapa, 1967). The backpropagation algorithm has been applied to a deep neural network by Yann LeCun for the first time in 1989 with the purpose of recognizing handwritten ZIP codes on mail (LeCun, et al., 1989).

The implementation of deep learning solutions requires knowledge of multiple areas, such as mathematics, statistics and computer science. It is also recommended to have access to a computational device that has appropriate capabilities for the given task. Nowadays, deep learning is gaining popularity due to the increasing computational power available to people around the globe as well as the high availability of knowledge resources.

Due to existing solutions of libraries focused on machine learning and deep learning, it is easier for people without strong mathematical background to explore the possibilities of neural networks. One of the challenges that are encountered by students and researchers is a better understanding of the algorithms that are lying in the center of machine learning and deep learning. It is important to know exactly how the algorithm works to contribute to the area of machine learning and deep learning as well as to escape errors that might be critical in different situations. With the growing popularization of deep learning, there are more and more people whose safety partially relies on machine learning and deep learning algorithms, such as users of self-driving cars and intrusion detection systems.

The aim of this thesis is to provide a complete understanding of the inner structure of a software focused on the design and training of neural networks and deep learning models. There are several parts that are to be covered in this paper: theoretical background, software implementation and applications of the software in the real world. The algorithms related to the machine learning and deep learning areas are

implemented with respect to the theoretical research that has taken place during the early stage of this project. The programming language that has been chosen for this software is C++ as it provides good performance and good technical capacity for further improvement.

2 THEORETICAL BACKGROUND

2.1 Machine Learning

Machine learning (ML) is the study of computer algorithms that improve automatically through experience (Mitchell, 1997). In other words, given a dataset and an optimization goal, the algorithm can produce a correct result with respect to the inputs that were not exposed to it previously.

The machine learning approaches to solve a given problem can be divided into the three categories: supervised learning, unsupervised learning, and reinforcement learning. The supervised learning is the most applied method nowadays as it provides a solution to the most common problems in the industry, such as classification and value prediction. The unsupervised learning algorithms are also widely used for such tasks as denoising and clustering of data. Reinforcement learning is an algorithm that solves the given task by means of trial and error and is commonly used for implementing software capable of solving puzzles, playing videogames, and driving vehicles (Vitelli & Nayebi, 2016). For example, Deep Blue, a chess-playing computer, was able to win against a reigning world champion Garry Kasparov.

2.1.1 Deep Learning and Neural Networks

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction (LeCun, Bengio, & Hinton, 2015). Deep learning algorithms provide possibility to extract complex features from a large amount of data. The computing system that is used to implement a deep learning algorithm is called a neural network. A neural network consists of elementary computational units called neurons, where multiple number of neurons form a layer. Layers in a neural network may be connected to each other in different ways; each of them defines the architecture of the neural network. The example of a neural network architecture is shown in the **Figure 1**.

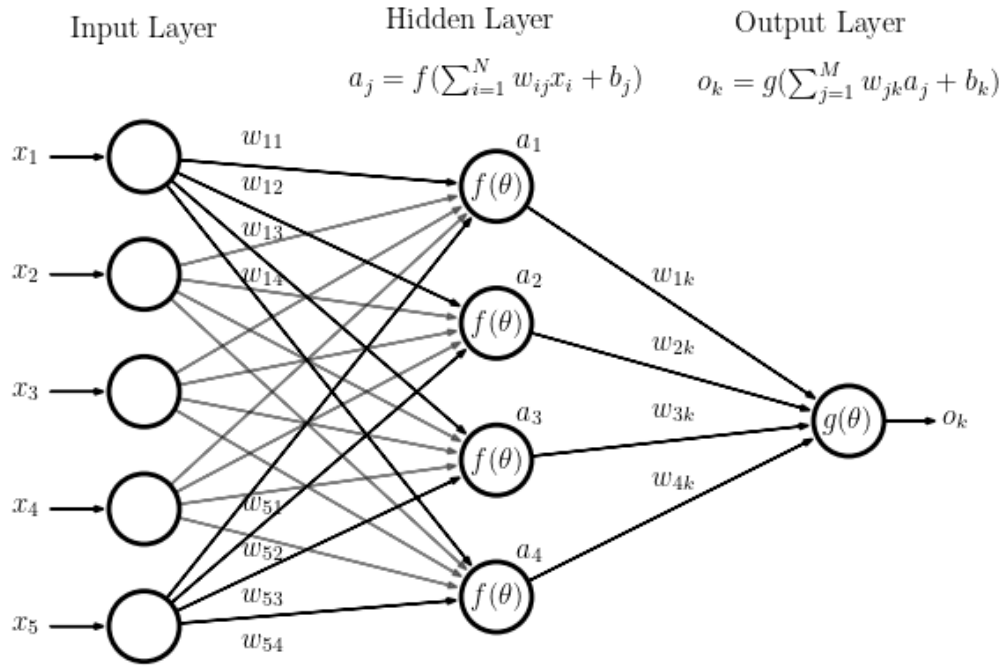


Figure 1. Neural network architecture example (Neural Network Diagram¶).

The algorithm of calculating the output of the neuron is shown in **Figure 2**.

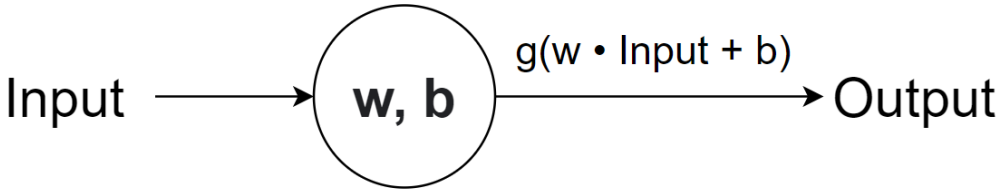


Figure 2. Algorithm of calculating the output of a neuron.

In Figure 2, "Input" is the value obtained from the dataset that is passed to the neuron in order to calculate the "Output" of the neuron. "w" and "b" are the weights of the neuron that will be modified automatically throughout the training process. They are used to calculate the pre-activation of a neuron which is denoted as:

$$p(Input, w, b) = w \cdot Input + b \quad (1)$$

Activation function of a neuron "g" is applied upon the pre-activation of the neuron to calculate the final output of the neuron. There are many activation functions that

are used with respect to different roles of the neuron. Usually, the activation functions of the neurons located on the same layer are chosen to be the same function because it reduces the complexity of the algorithm and provides as good algorithm performance as could be achieved by varying activation functions across the neurons of the layer.

The modification of the weights is done with help of a backpropagation algorithm. To understand how the algorithm is implemented we can look at the **Figure 3**.

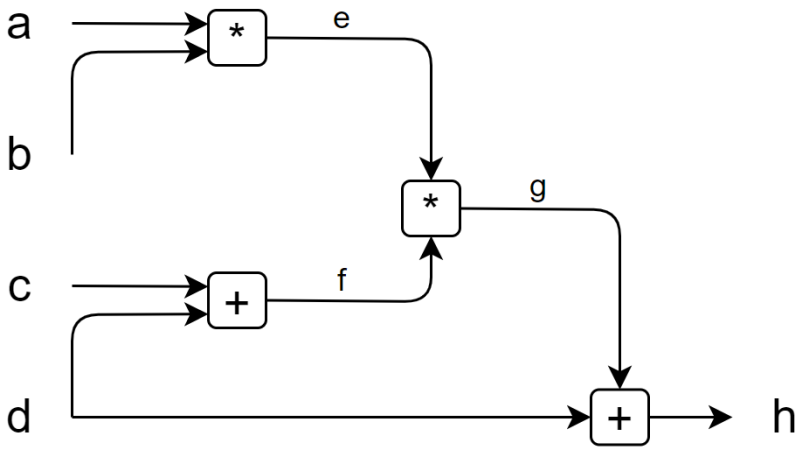


Figure 3. Computational graph.

This figure represents a computational graph, a directed graph where the nodes correspond to mathematical operations. This graph represents the following mathematical expression:

$$(a * b) * (c + d) + d = h \tag{2}$$

By assigning values to the inputs a, b, c and d, we can calculate the result of the forward pass procedure on this graph. Let the values of the inputs be as follows:

$$a = 1,\ \ b = 2,\ \ c = 3,\ \ d = 4$$

Then the output h will be equal to:

$$e = a * b = 2$$

$$f = c + d = 7$$

$$g = e * f = 14$$

$$h = g + d = 18$$

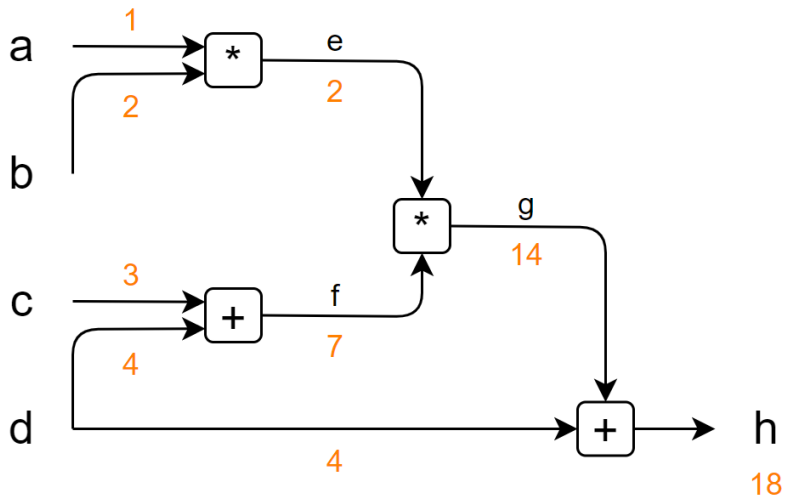


Figure 4. Computational graph with values.

We can perform backpropagation by calculating the partial derivatives of the output with respect the inputs. The first step is to calculate the partial derivative of the output to itself:

$$\frac{\partial h}{\partial h} = 1 \tag{3}$$

The derivative of h to g is:

$$\frac{\partial h}{\partial g} = \frac{\partial (g + d)}{\partial g} = 1 \tag{4}$$

In the same way, the derivative of h to d is also equals to 1.

By utilizing the chain rule, it is possible to calculate the partial derivatives with respect to other values in the following way:

$$\frac{\partial h}{\partial e} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial e} \tag{5}$$

The value of $\frac{\partial h}{\partial g}$ is already calculated, therefore, it is necessary to calculate only the second derivative:

$$\frac{\partial g}{\partial e} = \frac{\partial (e * f)}{\partial e} = f \tag{6}$$

In the same way, the value of $\frac{\partial g}{\partial f}$ is equal to e.

Therefore, the values of the derivatives are calculated as follows:

$$\frac{\partial h}{\partial e} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial e} = 1 * f = f = 7 \tag{7}$$

$$\frac{\partial h}{\partial e} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial f} = 1 * e = e = 2 \tag{8}$$

The last step is to calculate the partial derivatives of h with respect to a, b, c and d, which can be done by applying the same chain rule and using the results of previous calculations.

$$\frac{\partial h}{\partial a} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial e} * \frac{\partial e}{\partial a} \tag{9}$$

$$\frac{\partial h}{\partial b} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial e} * \frac{\partial e}{\partial b} \tag{10}$$

$$\frac{\partial h}{\partial c} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial f} * \frac{\partial f}{\partial c} \tag{11}$$

$$\frac{\partial e}{\partial a} = \frac{\partial (a * b)}{\partial a} = b \tag{12}$$

$$\frac{\partial e}{\partial b} = \frac{\partial (a * b)}{\partial b} = a \tag{13}$$

$$\frac{\partial f}{\partial c} = \frac{\partial (c + d)}{\partial c} = 1 \tag{14}$$

The value of $\frac{\partial h}{\partial d}$ is equal to 1 as it can be calculated by the following equation:

$$\frac{\partial h}{\partial d} = \frac{\partial (g + d)}{\partial d} = 1 \tag{15}$$

As the result, it is possible to calculate the values of the final partial derivatives:

$$\frac{\partial h}{\partial a} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial e} * \frac{\partial e}{\partial a} = 1 * f * b = f * b = 7 * 2 = 14 \tag{16}$$

$$\frac{\partial h}{\partial b} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial e} * \frac{\partial e}{\partial b} = 1 * f * a = f * a = 7 * 1 = 7 \tag{17}$$

$$\frac{\partial h}{\partial c} = \frac{\partial h}{\partial g} * \frac{\partial g}{\partial f} * \frac{\partial f}{\partial c} = 1 * e * 1 = e = 2 \tag{18}$$

$$\frac{\partial h}{\partial d} = 1 \tag{19}$$

By using the same approach, it is possible to calculate the partial derivatives of a loss function applied to a neural network output with respect to the weights and biases for each neuron. The computational graphs for a single neuron would look like in **Figure 5**.

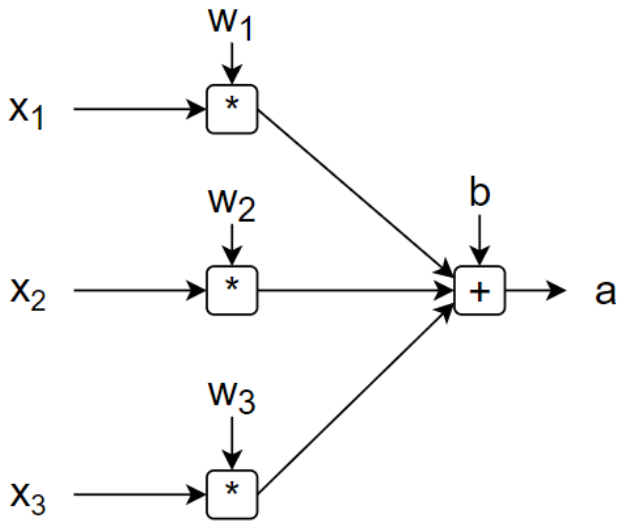


Figure 5. Computational graph of a single neuron.

The loss function is applied to the output of the last layer of a network to determine how inaccurate is the output of the neural network compared to the expected value from the dataset (in case of supervised learning).

2.2 Backpropagation Algorithm

Backpropagation is by far the most complex step of the machine learning algorithm from the mathematical point of view. Moreover, the backpropagation is the core part that enables the existence of machine learning in general.

The backpropagation step of a learning algorithm is based on calculating the partial derivatives of the loss function with respect to the weights and biases of each neuron. These derivatives are then used to calculate the amount of change that must be applied to the values of the weights and biases after the current iteration. The reason behind this approach is based on the idea of minimizing the loss function, which directly reflects the accuracy of the model. The projection of the loss function can be plotted to demonstrate the importance of the derivatives during the backpropagation. **Figure 6** shows that the further the value of the weight from the optimal, the steeper the curve of the loss function, which leads to high value of the derivative

of that function at the given point. The value of the derivative is decreasing when moving closer to the point of global minima, which allows the optimization algorithm to converge.

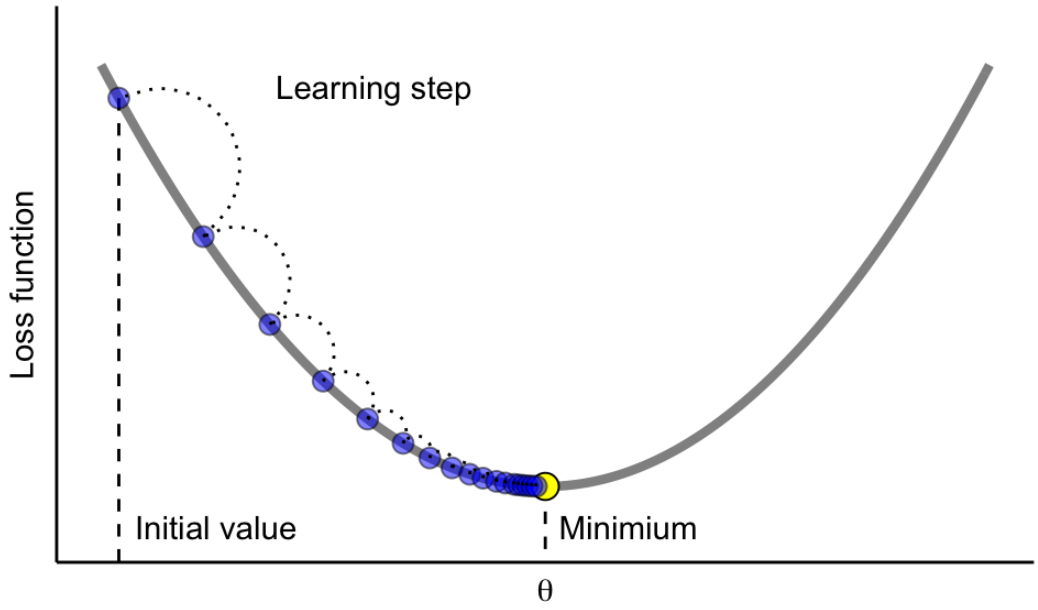


Figure 6. Loss function graph.

The process of backpropagation is illustrated in **Error! Reference source not found.**.

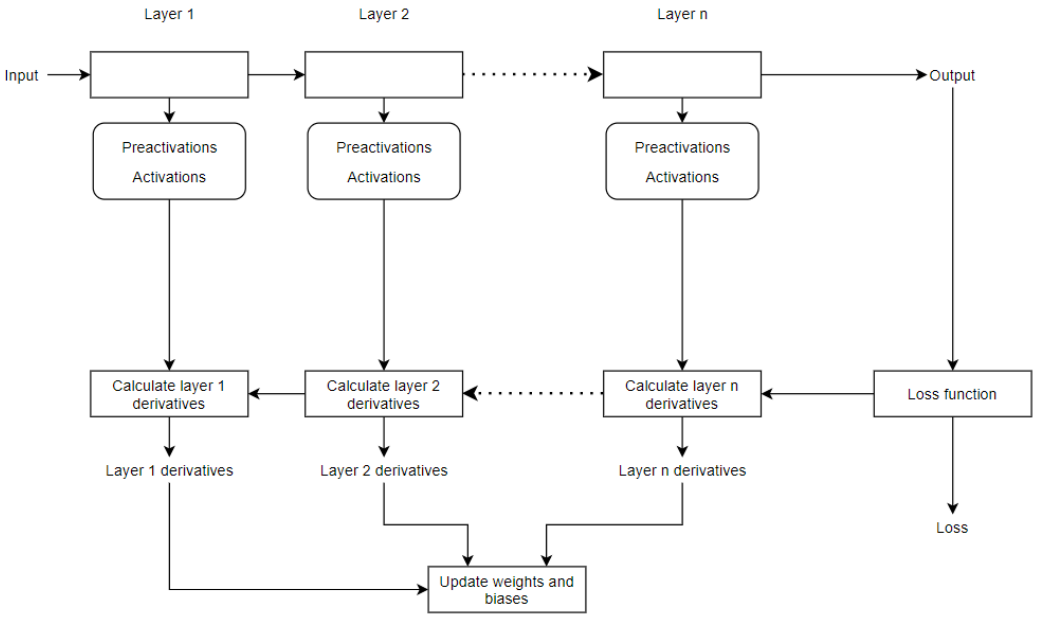


Figure 7. Backpropagation algorithm.

The pre-activation value is the value that is calculated before the activation value of the neuron, when the activation of the neuron is the value that is outputted by an activation function when applied on a pre-activation. As seen in the diagram, the pre-activations and activations of the neurons for each layer must be stored in order to be used by the backpropagation algorithm in the future. At the end of the iteration, the calculated partial derivatives of the loss function with respect to each weight and bias are used to update the weights and biases. The input and output are not necessarily scalars and could also be in a form of vectors when the network accepts multiple values and/or returns multiple values as a result.

Let the index of the current layer be l, the index of the current neuron in the layer be n and the total number of input values be m. Then, the pre-activation of this neuron is calculated using the following formula:

$$p^{(n)[l]} = \sum_{i=1}^{m} x_i * w_i^{(n)[l]} + b^{(n)[l]} \tag{20}$$

Where x_i is the input value with index i,

$w_i^{(n)[l]}$ is the weight of the neuron with the index n for the input with the index i on the layer with the index l and

$b^{(n)[l]}$ is the bias of the neuron n at the layer l.

It is also possible to rewrite the equation above in the vectorized form:

$$p^{(n)[l]} = W^{(n)[l]T} \cdot X + b^{(n)[l]} \tag{21}$$

Where X and $W^{(n)[l]}$ are column vectors of the form:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \qquad W = \begin{bmatrix} w_1^{(n)[l]} \\ w_2^{(n)[l]} \\ \vdots \\ w_m^{(n)[l]} \end{bmatrix} \tag{22}$$

In the equation (21), the superscript T of the W vector indicates that the transpose operation is performed on it.

It is possible to calculate the values of all the neurons one by one, however, in practice it is more computationally efficient to perform calculations in a vectorized form instead of iterating over many atomic mathematical operations. The vectorized form of equation to calculate pre-activations of all the neurons on the current layer will look as follows:

$$P^{[l]} = W^{[l]T} \cdot X + B^{[l]} \tag{23}$$

Where $W^{[l]}$ and $B^{[l]}$ are in the forms of the following matrices:

$$W^{[l]} = \begin{bmatrix} W_1^{(1)[l]} & W_1^{(2)[l]} & \cdots & W_1^{(n)[l]} \\ W_2^{(1)[l]} & W_2^{(2)[l]} & \cdots & W_2^{(n)[l]} \\ \vdots & \vdots & \cdots & \vdots \\ W_m^{(1)[l]} & W_m^{(2)[l]} & \cdots & W_m^{(n)[l]} \end{bmatrix} \tag{24}$$

$$B^{[l]} = \begin{bmatrix} b^{(1)[l]} \\ b^{(2)[l]} \\ \vdots \\ b^{(n)[l]} \end{bmatrix} \tag{25}$$

To calculate the activation of a neuron the activation function must be applied to the pre-activation of this neuron:

$$a = g(p) \tag{26}$$

The activation function is chosen depending on the main purpose of the layer the neuron is located in. For example, the activation function is often chosen to be the sigmoid function in case the current layer is a classification layer and the number of labels is two. The logic of this decision is because the sigmoid function is a monotonic function that outputs a value between 0 and 1, which makes this function a good candidate for a task with only two possible result labels.

In case of vectorized solution, the activation function is applied to every element of the pre-activation vector, which results in a new vector of the same dimensions.

$$A^{[l]} = g^{[l]}\left(P^{[l]}\right) \tag{27}$$

The next layer of neurons accepts the activations of the neurons on the previous layer as inputs and performs the same calculations of its own pre-activations and activations. Lastly, the activation of the last layer is passed to the loss function, the function that outputs a value that determines the degree of similarity of the output of the network with the expected value from the dataset. In case the neural network is performing linear regression type of task, it would be reasonable to choose our loss function to be the quadratic loss function, which calculates the sum of the squared distances between the output of the network and the expected output. It is also common to denote the output of the network with the symbol y and the expected output with the symbol $\hat{y}$ (“waı hæt”).

$$Loss = L(y, \hat{y}) \tag{28}$$

To start adjusting weights and biases inside the network we must calculate the partial derivatives of the loss function with respect to the weights and biases. The first step would be to calculate these derivatives for the last layer of the network, because it is the closest layer to the loss calculation process.

$$\frac{\partial L}{\partial w_m^{(n)[last]}} = \frac{\partial L}{\partial y_n} * \frac{\partial g^{[last]}}{\partial p^{(n)[last]}} * \frac{\partial p^{(n)[last]}}{\partial w_m^{(n)[last]}} \tag{29}$$

In the equation (29) the layer index $last$ corresponds to the index of the last layer of the network.

Let the loss function L be the squared loss function and the number of neurons in the output layer of the network be N. Then, the loss value of the network for the current output values is expressed as the following:

$$L(y, \hat{y}) = \frac{1}{2N} * \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{30}$$

Let the activation function of the neuron be the linear function, indicating that the problem that we are solving with the current network is the linear regression problem. Moreover, the loss function above is perfectly suitable for such task. The activation function will take the following form:

$$g^{[last]}\left(p^{(n)[last]}\right) = p^{(n)[last]} \tag{31}$$

Now it is possible to calculate each component of the equation (29):

$$\frac{\partial L}{\partial y_n} = \frac{\partial \left(\frac{1}{2N} * \sum_{i=1}^{N} (y_i - \hat{y}_i)^2\right)}{\partial y_n} = \frac{1}{2N} * \ 2 * (y_n - \hat{y}_n) = \frac{1}{N} * (y_n - \hat{y}_n) \tag{32}$$

$$\frac{\partial g^{[last]}}{\partial p^{(n)[last]}} = \frac{\partial p^{(n)[last]}}{\partial p^{(n)[last]}} = 1 \tag{33}$$

Let M be the total number of weights of the neuron n at the layer $last$.

$$\frac{\partial p^{(n)[last]}}{\partial w_m^{(n)[last]}} = \frac{\partial \left(\sum_{i=1}^{M} \left(w_i^{(n)[last]} * a^{(i)[last-1]} \right) + b^{(n)[last]} \right)}{\partial w_m^{(n)[last]}} = a^{(m)[last-1]} \quad (34)$$

To calculate the bias for the same neuron, we only must change the last derivative of the sequence to the following:

$$\frac{\partial p^{(n)[last]}}{\partial b^{(n)[last]}} = \frac{\partial \left(\sum_{i=1}^{M} \left(w_i^{(n)[last]} * a^{(i)[last-1]} \right) + b^{(n)[last]} \right)}{\partial b^{(n)[last]}} = 1 \quad (35)$$

In the result, the complete equations of the partial derivatives with respect to one of the weights and to the bias will take the following form.

$$\frac{\partial L}{\partial w_m^{(n)[last]}} = \frac{1}{n} * (y_n - \hat{y}_n) * a^{(m)[last-1]} \quad (36)$$

$$\frac{\partial L}{\partial b^{(n)[last]}} = \frac{1}{n} * (y_n - \hat{y}_n) \quad (37)$$

The derivatives above can be expressed in vectorized form. To simplify the expression, it is possible to denote the common part of these two derivatives as following:

$$dZ^{[last]} = \begin{bmatrix} \frac{1}{n} * (y_1 - \hat{y}_1) \\ \vdots \\ \frac{1}{n} * (y_n - \hat{y}_n) \end{bmatrix} \quad (38)$$

It is also necessary to present the activations of the previous layer in a form of a vector:

$$A^{[last-1]} = \begin{bmatrix} a^{(1)[last-1]} \\ \vdots \\ a^{(m)[last-1]} \end{bmatrix} \quad (39)$$

Using vectors above, it is now possible to express the derivatives in a vectorized form:

$$\frac{\partial L}{\partial W^{[last]}} = dZ^{[last]} \cdot A^{[last-1]T} = \begin{bmatrix} \frac{\partial L}{\partial w_1^{(1)[last]}} & \cdots & \frac{\partial L}{\partial w_m^{(1)[last]}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial w_1^{(n)[last]}} & \cdots & \frac{\partial L}{\partial w_m^{(n)[last]}} \end{bmatrix} \quad (40)$$

It is common to denote the partial derivatives of the loss function with respect to the weights and biases in a simplified form:

$$\frac{\partial L}{\partial w_m^{(n)[last]}} = dw_m^{(n)[last]} \quad (41)$$

$$\frac{\partial L}{\partial W^{[last]}} = dW^{[last]} \quad (42)$$

The shorter notation is introduced to provide a better readability of the expressions for the reader.

Therefore, the equation (40) will take the following form:

$$dW^{[last]} = dZ^{[last]} \cdot A^{[last-1]T} = \begin{bmatrix} dw_1^{(1)[last]} & \cdots & dw_m^{(1)[last]} \\ \vdots & \ddots & \vdots \\ dw_1^{(n)[last]} & \cdots & dw_m^{(n)[last]} \end{bmatrix} \quad (43)$$

The calculation process of the partial derivatives for the previous layers is like the example above, but instead of performing the calculation again from the beginning, it is possible to take advantage of the previous calculations. We can start by calculating the partial derivatives of the first neuron in the previous layer.

$$dw_m^{(n)[last-1]} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial p^{(i)[last]}} * \frac{\partial p^{(i)[last]}}{\partial a^{(n)[last-1]}} * \frac{\partial a^{(n)[last-1]}}{\partial p^{(n)[last-1]}} * \frac{\partial p^{(n)[last-1]}}{\partial w_m^{(n)[last-1]}} \right) \quad (44)$$

Therefore, the only derivatives that must be calculated are:

$$\frac{\partial p^{(i)[last]}}{\partial a^{(n)[last-1]}} = \frac{\partial \left(\sum_{j=1}^{M} \left(w_j^{(i)[last]} * a^{(j)[last-1]} \right) + b^{(i)[last]} \right)}{\partial a^{(n)[last-1]}} = w_n^{(i)[last]} \quad (45)$$

$$\frac{\partial a^{(n)[last-1]}}{\partial p^{(n)[last-1]}} = \frac{\partial g^{[last-1]}\left(p^{(n)[last-1]}\right)}{\partial p^{(n)[last-1]}} = g^{[last-1]\prime} \tag{46}$$

$$\frac{\partial p^{(n)[last-1]}}{\partial w_m^{(n)[last-1]}} = \frac{\partial\left(\sum_{i=1}^{M}\left(w_i^{(n)[last-1]} * a^{(i)[last-2]}\right) + b^{(n)[last-1]}\right)}{\partial w_m^{(n)[last-1]}} \tag{47}$$

$$\frac{\partial p^{(n)[last-1]}}{\partial w_m^{(n)[last-1]}} = a^{(m)[last-2]} \tag{48}$$

The final equation will take the next form:

$$dw_m^{(n)[last-1]} = \sum_{i=1}^{N}\left(\frac{\partial L}{\partial p^{(i)[last]}} * w_n^{(i)[last]} * g^{[last-1]\prime} * a^{(m)[last-2]}\right) \tag{49}$$

The $g^{[last-1]\prime}$ is the derivative of the activation function of the layer $last-1$. In this example, it is realistic to choose this function to be the sigmoid function. It is important for the network to have non-linear activation functions in its hidden layers to learn more complex features of the provided data, otherwise, the output of the network will be linear, which is not sufficient for the most tasks.

$$g^{[last-1]}(x) = \sigma(x) = \frac{1}{1+e^{-x}} \tag{50}$$

$$g^{[last-1]'}(x) = \frac{1}{1+e^{-x}} * \left(1 - \frac{1}{1+e^{-x}}\right) = \sigma(x)(1-\sigma(x)) \tag{51}$$

The partial derivative for the bias is calculated in the similar way:

$$db^{(n)[last-1]} = \sum_{i=1}^{N}\left(\frac{\partial L}{\partial p^{(i)[last]}} * w_n^{(i)[last]} * g^{[last-1]\prime}\right) \tag{52}$$

Before vectorizing these expressions, we can extract the common part of the equations 49 and 52:

$$dz^{(n)[last-1]} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial p^{(i)[last]}} * w_n^{(i)[last]} * g^{[last-1]\prime} \right) \tag{53}$$

$$dw_m^{(n)[last-1]} = dz^{(n)[last-1]} * a^{(m)[last-2]} \tag{54}$$

$$db^{(n)[last-1]} = dz^{(n)[last-1]} \tag{55}$$

Now we can express the $dZ^{[last-1]}$ in a vector form:

$$dZ^{[last-1]} = \begin{bmatrix} dz^{(1)[last-1]} \\ \vdots \\ dz^{(n)[last-1]} \end{bmatrix} \tag{56}$$

Moreover, it is possible to express the partial derivatives of the weights and biases in a vectorized form:

$$dW_m^{[last-1]} = dZ^{[last-1]} * a^{(m)[last-2]} \tag{57}$$

$$dB^{[last-1]} = dZ^{[last-1]} \tag{58}$$

We can also improve the vectorized form of the weight derivative by introducing new vector:

$$A^{[last-2]} = \begin{bmatrix} a^{(1)[last-2]} \\ \vdots \\ a^{(m)[last-2]} \end{bmatrix} \tag{59}$$

$$dW^{[last-1]} = dZ^{[last-1]} * A^{[last-2]T} \tag{60}$$

After the calculation of the partial derivatives for the weights and biases, the original values of those parameters are to be modified following the formula below:

$$w = w - \alpha * dw \tag{61}$$

$$b = b - \alpha * db \tag{62}$$

The learning rate is denoted by α and is a small positive value often between range 0 and 1. The learning rate controls how quickly or slowly a neural network model

learn the issue. A higher value of the learning rate will lead the model to converge faster but might also result in a suboptimal solution. A lower learning rate might result in a more optimal result; however, it might also negatively affect the speed with which the model reaches the solution. Therefore, the value of the learning rate is chosen based on the learning model as well as on the dataset.

3 SOFTWARE IMPLEMENTATION

We address the implementation of the software for deep learning in this research project. In this part the main implementation details of the software is discussed.

3.1 Running Environment

The software is developed in the Windows 10 Home Edition 64-bit environment. The following list of development environment attributes were present throughout the whole process of development:

- Operating system: Windows 10 Home Edition
- Instruction set: x86-64
- Random-access memory size: 16Gb
- Graphics card: Nvidia GeForce GTX 1060 Ti
- Video Random Access Memory size: 6Gb
- Central processing unit: Intel Core i7-8700

The minimum requirements for a system to build and execute the software are defined by system requirements of Visual Studio 2019 on the official documentation webpage (Anzhou).

3.2 Software Stack

The software is developed in the C++ programming language. The following is the list of additional software that were used during the development:

- Compiler: Microsoft Visual C++ (MSVC)
- Integrated development environment: Microsoft Visual Studio 2019
- Source-code editor: Visual Studio Code
- Version control system: Git

There is an external library that was used in this project – Eigen. “Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related

algorithms.” (Eigen, 2021). This library was chosen for this project because it provides all the necessary functionality and documentation. There is also a big community of people that use this library for their projects, which assures good stability of the library.

3.3 Project Structure

The project is divided in two main parts, library and client. The library is the part of the project that contains all the features that are useful to the user of the machine learning and deep learning software. The client is another part of the project that is created on order to test the functionality of the library. The file and folder structure of the top layer of the project look like the in **Figure 8**.

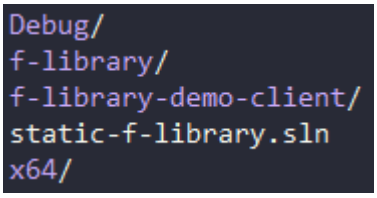


Figure 8. Top-level project structure.

The “f-library” directory contains the library part of the project, the “f-library-demo-client” contains the client that uses the functionality provided by the library and “x64” contains the binaries of both the library and the client built for the x64 architecture.

Figure 9 shows the structure of the library part of the project.

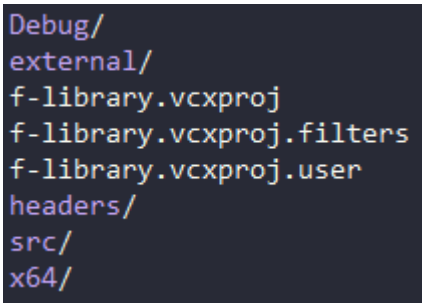


Figure 9. Library structure.

The source code is located under the “src” directory and the header files are located under the “headers” directory. The “external” directory contains external libraries that are used in this project; in this case, there is only one library, Eigen. The “x64” directory contains the compiled source files when the target architecture is set to be “x64”.

A similar structure is present for the client part of the project with minor difference of not having external dependencies and shown in **Figure 10**.

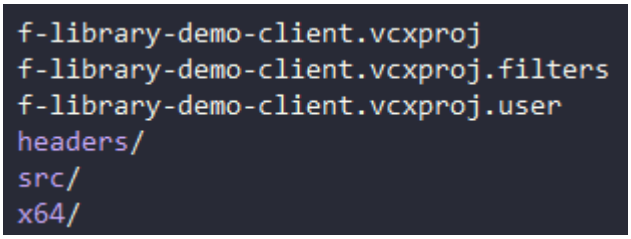


Figure 10. Client structure.

3.4 Library Implementation

The library is divided into several source code files, which are shown in **Figure 11**.

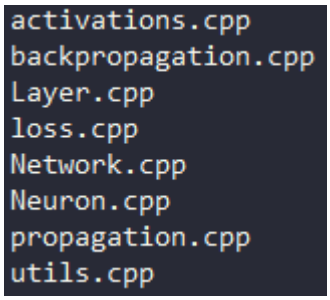


Figure 11. Library source code files.

The files names of which start with a capital letter contain the classes, whereas other files include separate functions that are reused inside the methods of the classes or directly in the client code.

The files containing classes are named after the different parts of the neural network: “Neuron”, “Layer” and “Network”. These classes implement methods for calculating output of neurons, layers, and the network. They also provide functionality to perform back-propagation to update weights correctly.

The other files provide functions that perform mathematical operations that can be reused, such as activation and pre-activation functions, loss functions and their derivatives.

3.4.1 Neuron Class

The neuron class is the smallest unit of the system. The source code of this class is shown in **Figure 12**.

```cpp
#include <algorithm>
#include <cstdlib>
#include <map>
#include <numeric>
#include <random>
#include <string.h>
#include <iostream>

#include "Neuron.h"
#include "activations.h"

Neuron::Neuron(int input_size)
{
    Neuron::parameters.w = Eigen::VectorXd(input_size);

    for (int i = 0; i < input_size; i++)
    {
        Neuron::parameters.w(i) = ((double)std::rand()) / RAND_MAX * std::sqrt(2.0 / input_size);
    }

    Neuron::parameters.b = 0;
}

DataTypes::NeuronParameters Neuron::getParameters() { return Neuron::parameters; }

Eigen::VectorXd Neuron::getW() { return Neuron::parameters.w; }

void Neuron::setW(const Eigen::VectorXd &w) { Neuron::parameters.w = w; }

double Neuron::getB() { return Neuron::parameters.b; }

void Neuron::setB(double b) { Neuron::parameters.b = b; }
```

Figure 12. Neuron class.

The neuron is initialized by providing the input size to the constructor, after which the correct number of weights are instantiated. The weights are initialized with random values ranging from 0 to 1 and the biases are initialized with the values of 0. This method of initialization provides good result during the training process. There are also methods for getting and setting the values of the weights and the bias, which will be necessary during the training process.

The type of the “Neuron::parameters” attribute is shown inside the header file for the Neuron class in **Figure 13**.

```
1   #pragma once
2
3   #include <map>
4   #include "propagation.h"
5   #include "DataTypes.h"
6   #include "Eigen/Dense"
7
8   class Neuron
9   {
10  private:
11      DataTypes::NeuronParameters parameters;
12
13  public:
14      Neuron(int input_size);
15      DataTypes::NeuronParameters getParameters();
16      Eigen::VectorXd getW();
17      void setW(const Eigen::VectorXd &newW);
18      double getB();
19      void setB(double newB);
20  };
```

Figure 13. Neuron class header file.

3.4.2 DataTypes

The “DataTypes” namespace is defined inside the “DataTypes.h” header file. It contains multiple struct definitions, which are used to provide better code structure which are shown in **Figure 14**.

```cpp
#pragma once

#include <vector>
#include "Eigen/Dense"


namespace DataTypes
{
    struct NeuronParameters
    {
        Eigen::VectorXd w;
        double b;
    };

    struct LayerParameters
    {
        Eigen::MatrixXd w;
        Eigen::VectorXd b;
    };

    struct LayerCacheResult
    {
        Eigen::VectorXd preactivations;
        Eigen::VectorXd activations;
    };

    struct NetworkCache
    {
        std::vector<Eigen::VectorXd> preactivations;
        std::vector<Eigen::VectorXd> activations;
    };

    struct Deltas
    {
        std::vector<Eigen::MatrixXd> dw;
        std::vector<Eigen::VectorXd> db;
    };
}
```

Figure 14. DataTypes header file.

The data structures above fulfill the following purpose:

- **NeuronParameters**:
  Stores weights and a bias of the Neuron class instance.
- **LayerParameters**
  Stores weights and biases of the neurons in the layer.
- **LayerCacheResult**
  Stores pre-activations and activations of the neurons in the layer.
- **NetworkCache**
  Stores pre-activations and activations of all the neurons in the network.
- **Deltas**
  Stores the values of the partial derivatives with respect to the weights and biases calculated during the backpropagation step of the training process.

3.4.3 Layer Class

The layer class contains one or more instances of the neuron class, which are stored inside a vector. There is a constructor that accepts multiple parameters (**Figure 15**):

- **layer_size**
  Determines how many neurons will be created for the layer.
- **input_size**
  The number of input values that are accepted by the neuron during the forward pass.
- **activation_function**
  The activation function that will be assigned to each neuron.

```cpp
Layer::Layer(int layer_size, int input_size, ACTIVATION_FUNCTION activation_function)
{
    this->layer_size = layer_size;
    this->activation_function = activation_function;

    neurons.reserve(layer_size);

    for (int i = 0; i < layer_size; i++)
    {
        Neuron *neuron = new Neuron(input_size);
        neurons.push_back(neuron);
    }
}
```

Figure 15. Layer class constructor.

The “forward_prop” method calculates pre-activations and activations of the neurons of the layer and returns these values as a “LayerCacheResult” data structure (**Figure 16**).

```cpp
DataTypes::LayerCacheResult Layer::forward_prop(const Eigen::VectorXd &activations)
{
    DataTypes::LayerParameters layerParams = this->getParams();
    Eigen::MatrixXd w = layerParams.w;
    Eigen::VectorXd b = layerParams.b;

    Eigen::VectorXd layerPreactivations = preactivation(w, activations, b);
    Eigen::VectorXd layerActivations = activation(layerPreactivations, activation_function);

    return {.preactivations = layerPreactivations, .activations = layerActivations};
}
```

Figure 16. Layer class forward propagation method.

The “getParams” method returns the parameters of the neurons in the layer as a “LayerParameters” data structure (**Figure 17**).

```
61  DataTypes::LayerParameters Layer::getParams()
62  {
63      Eigen::MatrixXd w(neurons.size(), neurons.at(0)->getW().size());
64      Eigen::VectorXd b(neurons.size());
65
66      for (std::size_t i = 0; i < neurons.size(); i++)
67      {
68          DataTypes::NeuronParameters parameters = neurons.at(i)->getParameters();
69          w.row(i) = parameters.w;
70          b(i) = parameters.b;
71      }
72
73      return {.w = w, .b = b};
74  }
```

Figure 17. Layer class get parameters method.

The "getNeurons" method returns the vector with the neurons of the layer (**Figure 18**).

```
76  std::vector<Neuron *> Layer::getNeurons()
77  {
78      return neurons;
79  }
```

Figure 18. Layer class get neurons method.

The "setWeights" method allows to set the values of the weights of the neurons of the layer (**Figure 19**).

```
81  void Layer::setWeights(Eigen::MatrixXd weights)
82  {
83      for (size_t i = 0; i < neurons.size(); i++) {
84          neurons.at(i)->setW(weights.row(i));
85      }
86  }
```

Figure 19. Layer class set weights method.

The header file of the layer class is shown in **Figure 20**.

```cpp
#pragma once
#include <map>

#include "Neuron.h"

class Layer
{
private:
    int layer_size;
    std::vector<Neuron *> neurons;

public:
    ACTIVATION_FUNCTION activation_function;
    Layer(int layer_size, int input_size, ACTIVATION_FUNCTION activation_function);
    DataTypes::LayerCacheResult forward_prop(const Eigen::VectorXd &x);
    DataTypes::LayerParameters getParams();
    std::vector<Neuron *> getNeurons();
    void setWeights(Eigen::MatrixXd weights);
};
```

Figure 20. Layer class header file.

3.4.4 Forward Propagation Functions

The functions that are used during the forward propagation are in the "propagation.cpp" file. This file also contains a function that returns a value of a derivative of an activation function which name is provided as an argument.

The function named as "preactivation" has weights, inputs and biases as parameters and returns values of pre-activations as a vector (**Figure 21**).

```cpp
Eigen::VectorXd preactivation(const Eigen::MatrixXd &w, const Eigen::VectorXd &activations, const Eigen::VectorXd &b)
{
    return (w * activations).array() + b.array();
}
```

Figure 21. Pre-activation function.

The function named as "activation" accepts pre-activations and an activation function enumeration type and returns a value that is calculated based on the provided type of the activation function (**Figure 22**).

```
Eigen::VectorXd activation(const Eigen::VectorXd &z, ACTIVATION_FUNCTION activation_function)
{
    switch (activation_function)
    {
    case ACTIVATION_FUNCTION::LINEAR:
        return z;
    case ACTIVATION_FUNCTION::SIGMOID:
        return sigmoid(z);
    case ACTIVATION_FUNCTION::TANH:
        return z.array().tanh();
    case ACTIVATION_FUNCTION::RELU:
        return relu(z);
    default:
        throw "Unrecognized activation function name!";
        break;
    }
}
```

Figure 22. Activation function.

The “activation_prime” function behaves in a similar way to the “activation” function except it returns the output of a derivative of the activation function (**Figure 23**).

```
Eigen::VectorXd activation_prime(const Eigen::VectorXd &z, ACTIVATION_FUNCTION activation_function)
{
    switch (activation_function)
    {
    case ACTIVATION_FUNCTION::LINEAR:
        return Eigen::VectorXd::Ones(z.size());
    case ACTIVATION_FUNCTION::SIGMOID:
        return sigmoid_prime(z);
    case ACTIVATION_FUNCTION::TANH:
        return tanh_prime(z);
    case ACTIVATION_FUNCTION::RELU:
        return relu_prime(z);
    default:
        throw "Unrecognized activation function name!";
        break;
    }
}
```

Figure 23. Activation prime function.

The header file of the propagation file is shown in **Figure 24**.

```cpp
#pragma once
#include <vector>
#include "Eigen/Dense"

enum ACTIVATION_FUNCTION
{
    LINEAR,
    SIGMOID,
    TANH,
    RELU
};

Eigen::VectorXd preactivation(const Eigen::MatrixXd &w, const Eigen::VectorXd &activations, const Eigen::VectorXd &b);

Eigen::VectorXd activation(const Eigen::VectorXd &z, ACTIVATION_FUNCTION activation_function);

Eigen::VectorXd activation_prime(const Eigen::VectorXd &z, ACTIVATION_FUNCTION activation_function);
```

Figure 24. Propagation header file.

3.4.5 Network Class

The network class contains the layers as well as the cache, which stores values of pre-activations and activations of the neurons after the forward propagation step. This class also includes methods to perform forward propagation, backpropagation, updating parameters and cost calculation.

The constructor of the network class accepts layers and a random seed (**Figure 25**).

```cpp
Network::Network(const std::vector<Layer *> &layers, std::optional<unsigned int> random_seed)
{
    // initialize random generator with the passed seed
    std::srand(random_seed.value_or(std::time(nullptr)));

    this->layers = layers;
}
```

Figure 25. Network class constructor.

The “random_seed” parameter controls whether the weights will be initialized with the same pseudo-random values as during the previous execution of the program.

The “getLayers” method returns the vector of layers of the network (**Figure 26**).

```
18  std::vector<Layer *> Network::getLayers()
19  {
20      return this->layers;
21  }
```

Figure 26. Network class get layers method.

The “forward_prop” method calculates the pre-activations and activations of the layers and stores these values in cache. It also returns the activation of the last layer, which is also the activation of the whole network (**Figure 27**).

```
23  Eigen::VectorXd Network::forward_prop(const Eigen::VectorXd &input)
24  {
25      this->cache.preactivations.reserve(layers.size());
26      this->cache.activations.reserve(layers.size() + 1);
27
28      Eigen::VectorXd layerActivations = input;
29
30      cache.activations.push_back(layerActivations);
31
32      for (std::size_t i = 0; i < layers.size(); i++)
33      {
34          DataTypes::LayerCacheResult layerCacheResult = layers.at(i)->forward_prop(layerActivations);
35
36          Eigen::VectorXd layerPreactivations = layerCacheResult.preactivations;
37          layerActivations = layerCacheResult.activations;
38
39          this->cache.preactivations.push_back(layerPreactivations);
40          this->cache.activations.push_back(layerActivations);
41
42          debug("Network::forward_prop > layer " + std::to_string(i) + " weights");
43          debug(layers.at(i)->getParams().w);
44
45          debug("Network::forward_prop > layer " + std::to_string(i) + " biases");
46          debug(layers.at(i)->getParams().b);
47
48          debug("Network::forward_prop > layer " + std::to_string(i) + " preactivations");
49          debug(layerPreactivations);
50
51          debug("Network::forward_prop > layer " + std::to_string(i) + " activations");
52          debug(layerActivations);
53      }
54
55      return layerActivations;
56  }
```

Figure 27. Network class forward propagation method.

The “back_prop” method calculates partial derivatives with respect to the weights and biases of the neurons and returns them as “Deltas” data structure (**Figure 28**, **Figure 29**).

```cpp
DataTypes::Deltas Network::back_prop(const Eigen::VectorXd &y)
{
    debug("Network::back_prop > y");
    debug(y);

    std::vector<Eigen::MatrixXd> dw;
    std::vector<Eigen::VectorXd> db;

    dw.reserve(layers.size());
    db.reserve(layers.size());

    Eigen::VectorXd current_da = mse_prime(this->cache.activations.back(), y);

    debug("Network::back_prop > current_da last");
    debug(current_da);

    Eigen::VectorXd current_dz = current_da.array() * activation_prime(this->cache.preactivations.back(), layers.back()->activation_function).array();

    debug("Network::back_prop > current_dz last");
    debug(current_dz);

```

Figure 28. Network class backpropagation method 1.

```cpp
    for (int i = layers.size() - 1; i >= 0; i--)
    {
        Eigen::MatrixXd current_dw;
        Eigen::VectorXd current_db;

        Layer *current_layer = layers.at(i);

        Eigen::VectorXd a_next = this->cache.activations.at(i);

        debug("Network::back_prop > a_next for layer " + std::to_string(i));
        debug(a_next);

        if (((std::size_t)i) == layers.size() - 1)
        {
            Eigen::MatrixXd current_dz_m = current_dz.matrix();
            Eigen::MatrixXd a_next_m = a_next.matrix();

            current_dw = current_dz_m * a_next_m.transpose();
            current_db = current_dz;
        }
        else
        {
            Layer *prev_layer = layers.at((int)i + 1);

            Eigen::MatrixXd w_prev = prev_layer->getParams().w;
            Eigen::VectorXd z_current = this->cache.preactivations.at(i);

            current_dz = (w_prev.transpose() * current_dz).array() * activation_prime(z_current, current_layer->activation_function).array();

            Eigen::MatrixXd current_dz_m = current_dz.matrix();
            Eigen::MatrixXd a_next_m = a_next.matrix();

            current_dw = current_dz_m * a_next_m.transpose();
            current_db = current_dz;
        }

        dw.push_back(current_dw);
        db.push_back(current_db);
    }

    return {.dw = dw, .db = db};
}
```

Figure 29. Network class backpropagation method 2.

The “update_parameters” method updates the weights and biases with the new values passed as arguments (**Figure 30**).

```
122  void Network::updateParameters(const std::vector<Eigen::MatrixXd> &dw, const std::vector<Eigen::VectorXd> &db, double alpha)
123  {
124      for (std::size_t i = 0; i < layers.size(); i++)
125      {
126          std::vector<Neuron *> neurons = layers.at(i)->getNeurons();
127
128          for (std::size_t j = 0; j < neurons.size(); j++)
129          {
130              Neuron *neuron = neurons.at(j);
131
132              Eigen::VectorXd w = neuron->getW();
133              double b = neuron->getB();
134
135              w = w.array() + alpha * dw.at(layers.size() - 1 - i).row(j).transpose().array();
136              b = b + alpha * db.at(layers.size() - 1 - i)(j);
137
138              neuron->setW(w);
139              neuron->setB(b);
140          }
141      }
142  }
```

Figure 30. Network class update parameters method.

The “calc_cost” method calculates the cost, which shows how close the values produced by the network are to the values in the dataset (**Figure 31**).

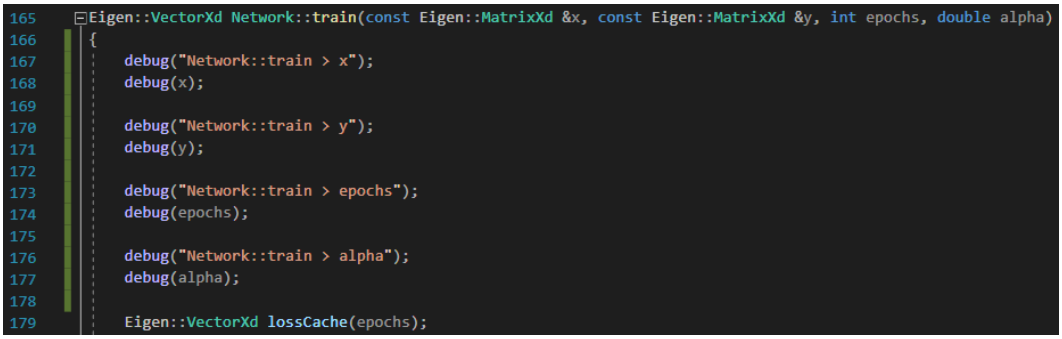
```
144  double Network::calc_cost(const Eigen::VectorXd &x, const Eigen::VectorXd &y)
145  {
146      debug("Network::calc_cost > x");
147      debug(x);
148
149      debug("Network::calc_cost > y");
150      debug(y);
151
152      Eigen::VectorXd activations = forward_prop(x);
153
154      debug("Network::calc_cost > activations");
155      debug(activations);
156
157      double result = mse(activations, y);
158
159      debug("Network::calc_cost > mse");
160      debug(result);
161
162      return result / y.size();
163  }
```

Figure 31. Network class calculate cost method.

The “debug” function just outputs the values provided to it into the text file, which might be useful during debugging.

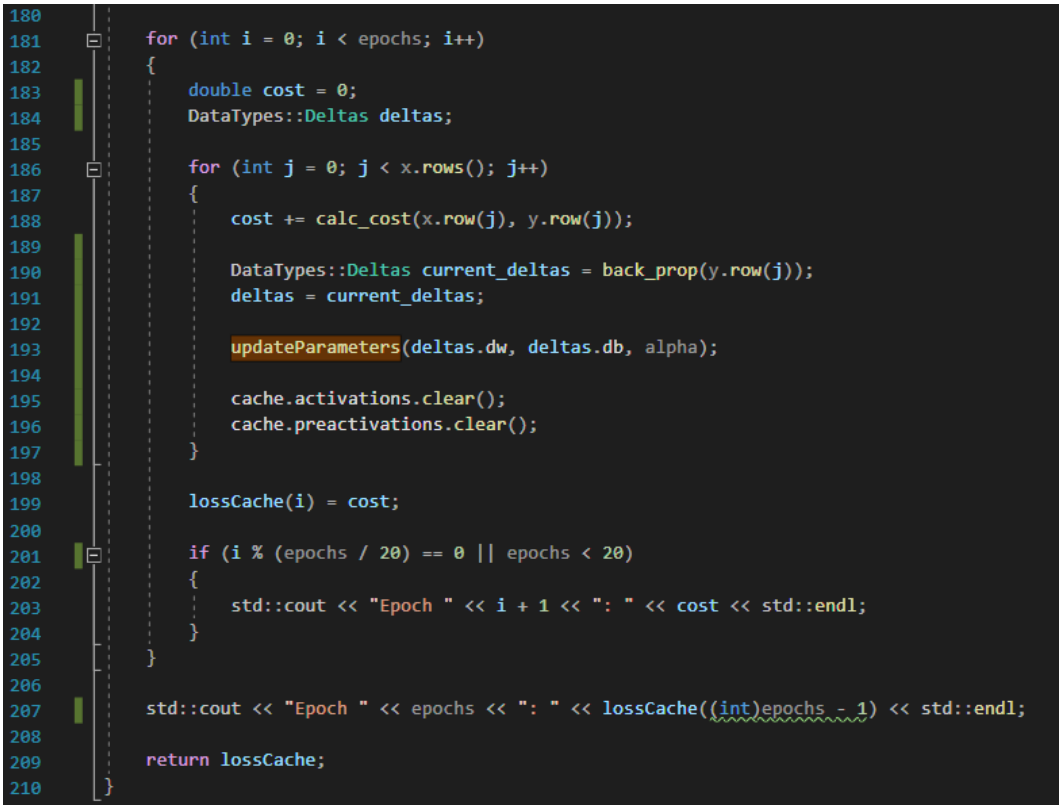
The “train” method is the first method that is called during the training process. It iterates over the epochs and training examples to perform forward propagation,

backpropagation and weight updates. This method accepts dataset values, number of epochs and learning rate as its attributes (**Figure 32**, **Figure 33**).



```
165  Eigen::VectorXd Network::train(const Eigen::MatrixXd &x, const Eigen::MatrixXd &y, int epochs, double alpha)
166  {
167      debug("Network::train > x");
168      debug(x);
169
170      debug("Network::train > y");
171      debug(y);
172
173      debug("Network::train > epochs");
174      debug(epochs);
175
176      debug("Network::train > alpha");
177      debug(alpha);
178
179      Eigen::VectorXd lossCache(epochs);
```

Figure 32. Network class train method 1.



```
180
181      for (int i = 0; i < epochs; i++)
182      {
183          double cost = 0;
184          DataTypes::Deltas deltas;
185
186          for (int j = 0; j < x.rows(); j++)
187          {
188              cost += calc_cost(x.row(j), y.row(j));
189
190              DataTypes::Deltas current_deltas = back_prop(y.row(j));
191              deltas = current_deltas;
192
193              updateParameters(deltas.dw, deltas.db, alpha);
194
195              cache.activations.clear();
196              cache.preactivations.clear();
197          }
198
199          lossCache(i) = cost;
200
201          if (i % (epochs / 20) == 0 || epochs < 20)
202          {
203              std::cout << "Epoch " << i + 1 << ": " << cost << std::endl;
204          }
205      }
206
207      std::cout << "Epoch " << epochs << ": " << lossCache((int)epochs - 1) << std::endl;
208
209      return lossCache;
210  }
```

Figure 33. Network class train method 2.

The header file of the network class is shown in **Figure 34**.

```
#pragma once

#include "Layer.h"
#include "loss.h"
#include <optional>
#include <functional>

class Network
{
private:
    std::vector<Layer *> layers;
    DataTypes::NetworkCache cache;

public:
    Network(const std::vector<Layer *> &layers, std::optional<unsigned int> random_seed = std::nullopt);

    std::vector<Layer *> getLayers();

    Eigen::VectorXd forward_prop(const Eigen::VectorXd &input);

    DataTypes::Deltas back_prop(const Eigen::VectorXd &y);

    void updateParameters(const std::vector<Eigen::MatrixXd> &dw, const std::vector<Eigen::VectorXd> &db, double alpha);

    double calc_cost(const Eigen::VectorXd &x, const Eigen::VectorXd &y);

    Eigen::VectorXd train(const Eigen::MatrixXd &x, const Eigen::MatrixXd &y, int epochs, double alpha);
};
```

Figure 34. Network class header file.

3.5 Example of Library Usage in Client

The client part contains example dataset, creation of the network class instance, training of the network, and drawing functionality to see the results of the training. The main function of the client calls the function responsible for the network training after which the drawing function is called to display the result (**Figure 35**).

```
int WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
    setDebugFile();

    Eigen::VectorXd result = trainModel();

    init_window(hInstance, getAction(result.array()));
}
```

Figure 35. Client main function.

The “trainModel” function contains the variables that store the example dataset, which will be passed to the network. Afterwards, the layers of the network are initialized and put into a vector “layers”. This vector is passed to the network class constructor. The training process is started by calling the “train” method of the network class instance named “network”, this method returns loss values collected during the training process. It is then possible to return loss, predictions or other values from the “trainModel” function (**Figure 36**, **Figure 37**).

```
Eigen::VectorXd trainModel() {
    Eigen::MatrixXd x(100, 1);
    x << Eigen::VectorXd::LinSpaced(100, -50, 50).array() / 50;

    debug("x:");
    debug(x);

    Eigen::MatrixXd y(100, 1);
    y << (x.array() * 50).pow(3) / 125000.0;

    debug("y:");
    debug(y);

    std::vector<Layer*> layers = {
        new Layer(50, 1, ACTIVATION_FUNCTION::TANH),
        new Layer(1, 50, ACTIVATION_FUNCTION::LINEAR)
    };
```

Figure 36. Client train model function 1.

```

    Network network = Network(layers, 0);

    Eigen::VectorXd loss = network.train(x, y, 10000, 0.01, 0.9);

    debug(loss);

    Eigen::VectorXd result(x.rows());

    for (int i = 0; i < x.rows(); i++) {
        result[i] = network.forward_prop(x.row(i))[0] * 125000.0;
    }

    debug("Network result:");
    for (auto item : result) {
        debug(item);
    }

    debug("Expected result:");
    debug(y);

    //return loss;
    return result;
    //return y.col(0);
}
```

Figure 37. Client train model function 2.

The result from the “trainModel” function is displayed on the screen by calling windows API and drawing pixels at the points of data. The algorithm that draws the data points at the correct locations is provided in **Figure 38**.

```
void action(Eigen::VectorXd data) {
    void* buffer_memory = get_buffer_memory_ref();
    int buffer_height = get_buffer_height();
    int buffer_width = get_buffer_width();
    double coef_x = (double)data.size() / buffer_width;
    double coef_y = (data.maxCoeff() - data.minCoeff()) / buffer_height;

    Eigen::VectorXd biased_y = data.array() - data.minCoeff();

    unsigned int* pixel = (unsigned int*)buffer_memory;
    for (int y = 0; y < buffer_height; y++) {
        for (int x = 0; x < buffer_width; x++) {
            int coeffed_x = coef_x * x;

            if (coeffed_x < biased_y.size() && y - (int)(biased_y(coeffed_x) / coef_y) < 0 && y - (int)(biased_y(coeffed_x) / coef_y - 5) > 0) {
                *pixel = 0xff0000;
            }
            else {
                *pixel = 0xffffff;
            }
            pixel++;
        }
    }

    display_buffer();
}
```

Figure 38. Client data drawing delegate.

As a result, the trained network output data is displayed in **Figure 39** as the green curve and original data is displayed as the red curve.

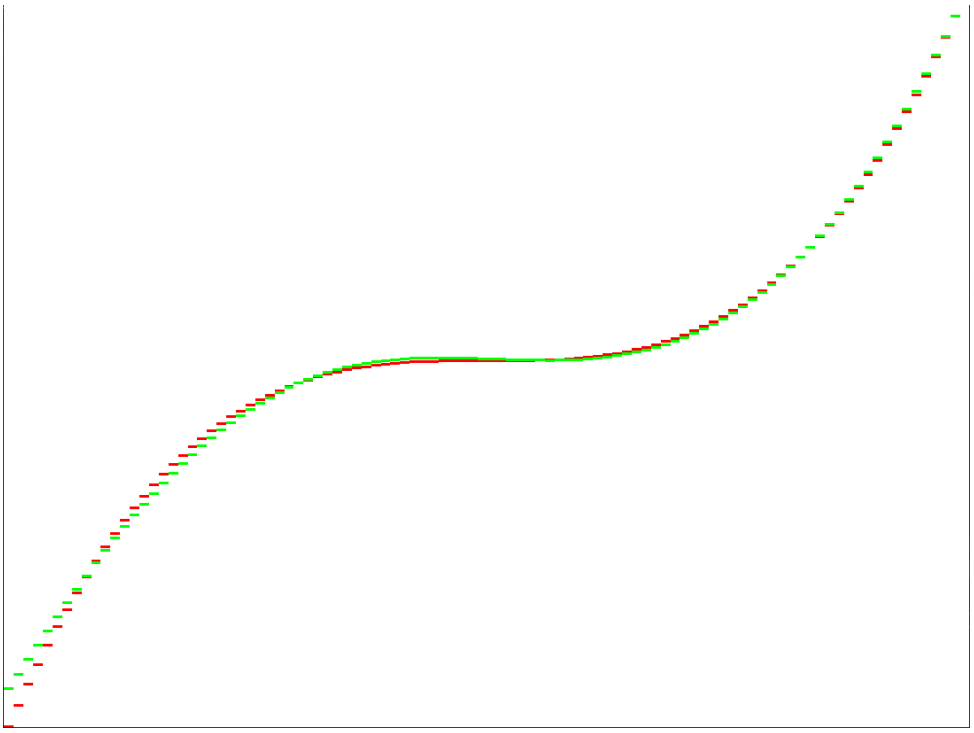


Figure 39. Client network output graph.

The graph of the loss shows that the network learned features successfully and had enough capacity to fit in the provided data (**Figure 40**).

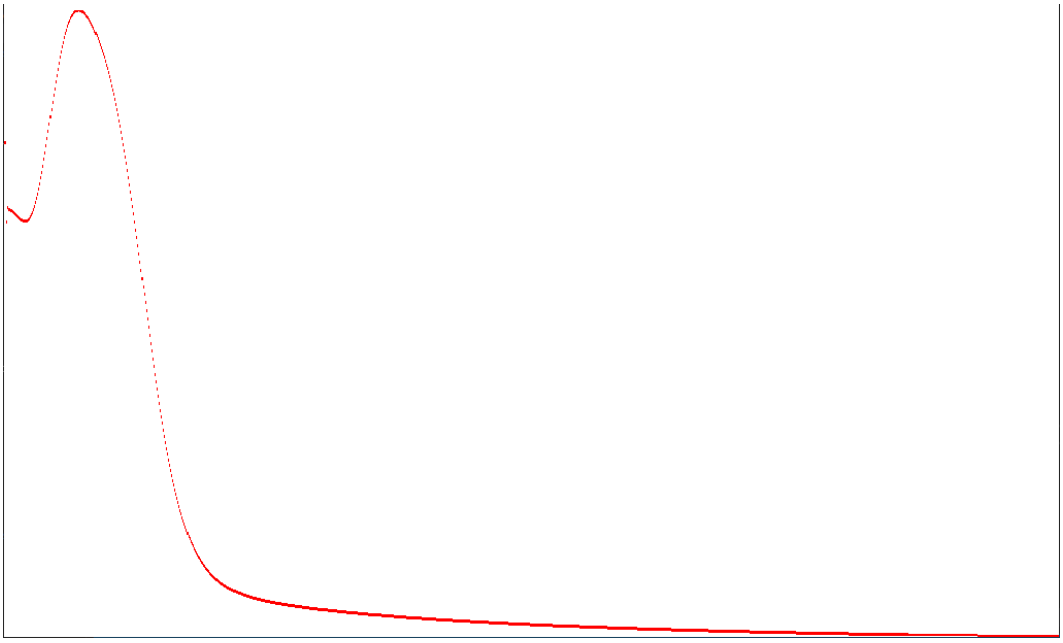


Figure 40. Client network loss graph.

3.6 Example of Library Usage with Complex Data

One of the functionalities of this software is to use multidimensional data as input to predict the output value. In this example, the dataset contains information of the imaginary houses with different properties: area in meters squared, number of rooms, presence of a sauna, presence of a balcony and price. The dataset is partially shown in **Figure 41**.

meters	rooms	sauna	balcony	price
82.12	4.00	1.00	1.00	752862.8
51.98	3.00	1.00	0.00	485352.9
70.79	2.00	1.00	0.00	632968.5
59.25	3.00	1.00	0.00	546273.6
28.96	2.00	0.00	1.00	267595.2
78.31	4.00	0.00	0.00	695927.4
70.84	1.00	0.00	0.00	603318.9
35.31	3.00	0.00	1.00	330720
98.35	2.00	0.00	0.00	843754.8
30.78	1.00	1.00	0.00	287841.2
66.55	2.00	0.00	1.00	582433.9
70.81	1.00	1.00	0.00	623101.6
49.94	3.00	1.00	0.00	468295
34.82	3.00	1.00	1.00	346613.4
80.77	3.00	1.00	1.00	731537.6
85.15	2.00	0.00	0.00	733225.7

Figure 41. Example dataset

The dataset was converted to a csv format and saved in a file.

The dataset was loaded into the program and normalized as shown in **Figure 42**.

```
// Load data
auto dataset = readFile("dataset.csv");

// Data preparation
Eigen::MatrixXd x = std::get<0>(dataset);
Eigen::MatrixXd y = std::get<1>(dataset);

auto normX = normalizeData(x);
auto normY = normalizeData(y);

auto normCoefX = std::get<1>(normX);
auto normCoefY = std::get<1>(normY);

x = std::get<0>(normX);
y = std::get<0>(normY);

writeFile(x, "normX.csv");
writeFile(y, "normY.csv");
```

Figure 42. Data loading and preparation

The normalization coefficients are stored to be able to convert the output values to the original scale. The normalized data was also written to files for further use.

In this example, the neural network contains two layers with one neuron each, which should be sufficient for this task. The first layer accepts four values as inputs because there are four features, besides the price that the dataset contains. The value that is predicted by the network is the price. The structure of the network is shown in **Figure 43**.

```
// Network structure setup
std::vector<Layer*> layers = {
    new Layer(1, 4, ACTIVATION_FUNCTION::SIGMOID),
    new Layer(1, 1, ACTIVATION_FUNCTION::LINEAR)
};
```

Figure 43. Example network structure

The learning rate during the training is set to 0.1 and the number of epochs is set to 100.

As a result, the loss after the last epoch was equal to 0.005430. The graph of the loss function is shown in **Figure 44**.

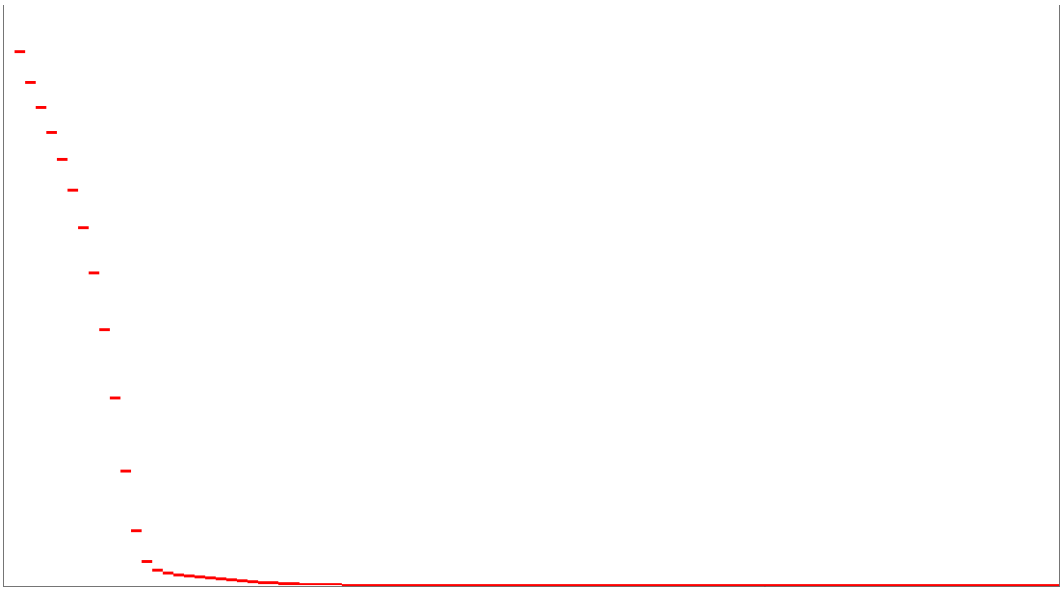


Figure 44. Example loss graph

It is now possible to predict the price of a house with custom properties. For example, we can predict the price of the house with the following properties:

Area: 50 m^2

Number of rooms: 2

Has sauna: No

Has balcony: Yes

The process of custom value prediction is shown in **Figure 45**.

```
log("Custom prediction");
Eigen::VectorXd custom_x(4);

custom_x(0) = 50.0 / normCoefX.at(0);   // area
custom_x(1) = 2.0 / normCoefX.at(1);    // number of rooms
custom_x(2) = 0.0 / normCoefX.at(2);    // has sauna
custom_x(3) = 1.0 / normCoefX.at(3);    // has balcony

double custom_result = network.forward_prop(custom_x)(0) * normCoefY.at(0);
std::cout << std::fixed;
log(custom_result);
```

Figure 45. Example custom prediction

The price of the following house is predicted to be 424 278.

If we perform the prediction of the price for the house with the same properties, but which also contain sauna, the price of the house is predicted to be 437 126.

3.7 Comparison to Existing Solutions

There are some other libraries, such as TensorFlow and Pytorch, which provide functionality to train and utilize neural networks. In this part, the current software is compared to a popular library created by Google called TensorFlow.

To compare the libraries, a neural network model was created with the same architecture in both TensorFlow and the current software. This network is used to be trained on the same dataset. The dataset that was chosen is the same dataset that was used in part 3.6. The code for implementation and training of the neural network in TensorFlow is shown in **Figure 46**, **Figure 47** and **Figure 48**.

```
[68] import tensorflow as tf
     import keras
     import numpy as np
     import pandas as pd
     from google.colab import files
     import io
     from matplotlib import pyplot as plt

[63] uploadedX = files.upload()
     uploadedY = files.upload()

     Choose Files No file chosen   Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
     Saving normX.csv to normX (1).csv
     Choose Files No file chosen   Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
     Saving normY.csv to normY (1).csv

[65] normX = pd.read_csv(io.BytesIO(uploadedX['normX.csv']))
     normY = pd.read_csv(io.BytesIO(uploadedY['normY.csv']))

[66] x = normX.to_numpy()
     y = normY.to_numpy()

[94] model = tf.keras.models.Sequential([
       tf.keras.layers.Dense(1, input_shape=(1, 4), activation='sigmoid'),
       tf.keras.layers.Dense(1)
     ])
```

Figure 46. TensorFlow example 1

```
[95] optimizer = keras.optimizers.Adam(learning_rate=0.1);
     model.compile(optimizer=optimizer, loss=keras.losses.mean_squared_error)
     print(model.summary())

     Model: "sequential_14"
     _________________________________________________________________
     Layer (type)                 Output Shape              Param #
     =================================================================
     dense_30 (Dense)             (None, 1, 1)              5
     _________________________________________________________________
     dense_31 (Dense)             (None, 1, 1)              2
     =================================================================
     Total params: 7
     Trainable params: 7
     Non-trainable params: 0
     _________________________________________________________________
     None
```

Figure 47. TensorFlow example 2

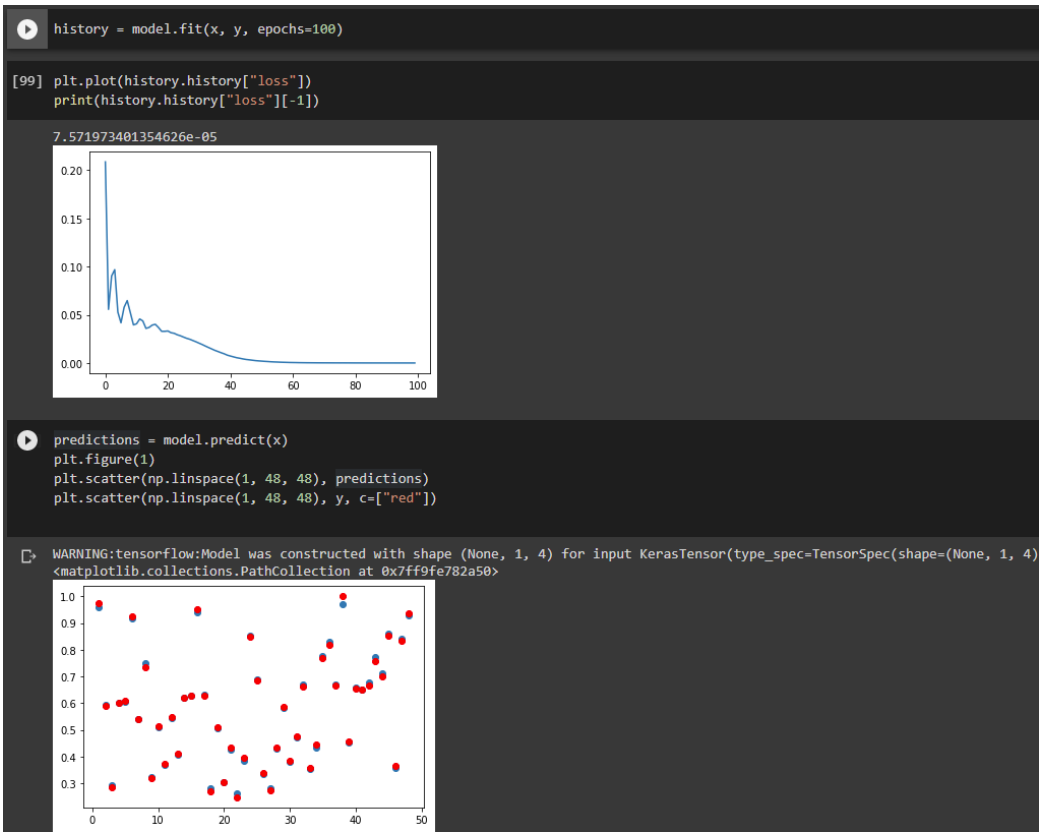


Figure 48. TensorFlow example 3

The number of layers in the network is two with one neuron in each layer. The learning rate is set to 0.1 and the number of epochs is set to 100. The loss that the model was able to achieve is 0.0000757. The loss graph is shown in **Figure 48**.

The software implemented during this project has shown slightly worse results in terms of loss value, which have reached the value of 0.005430. The reason why the TensorFlow model was able to achieve a better loss value is due to the better optimization algorithm that was used. The loss graph of the project software is shown in **Figure 44**.

There are other differences that make TensorFlow better software than the current implementation, such as GPU utilization and multithreading. These features require more time than was available for this project and were not feasible to implement.

Most of the existing libraries would perform better than the current implementation because of the GPU and multithreading support, which are vital to have in order to perform computationally heavy tasks, such as image recognition.

4 CONCLUSIONS

To summarize the results, the software for machine learning and deep learning applications is working according to the theoretical findings.

Both theory and implementation required attention to small details, which made this project very difficult and interesting at the same time. There are many resources, such as online courses, research papers and videos that I had to study in order gain necessary knowledge which I could apply. Moreover, I cannot understate the importance of the mathematics classes that I have taken at university, they were very helpful for this project.

The programming language that I have chosen for the practical part of the project, C++, was new to me when I started. One of my goals was to extend my knowledge of this programming language; using it in such a project was a perfect opportunity to study it as well.

The final version of the software has its flaws, such as utilizing only one thread of the processor, not utilizing the graphical processing unit and insufficient software optimizations. Unfortunately, these milestones were unlikely for me to achieve within the given time constraints. Due to low performance of the software, the number of applications of the software, such as image recognition and voice recognition, were impractical with the current solution due to the amount of time it would take to train the model.

There are many points that must be improved for this software to reach the level of quality and performance that is required in the industry; however, it was important to understand the basic ideas behind machine learning and to build a working solution based on those ideas.

REFERENCES

Visual Studio 2019 System Requirements. Accessed 26.4.2021 https://docs.microsoft.com/en-us/visualstudio/releases/2019/system-requirements

Eigen. Accessed 2.3. 2021: https://eigen.tuxfamily.org/

Ivachnenko, A. G., & Lapa, V. G. (1967). Cybernetics and forecasting techniques. New York: American Elsevier.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521, 436–444. doi:10.1038/nature14539

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. Neural Computation, 1, 541-551. doi:10.1162/neco.1989.1.4.541

Mitchell, T. M. (1997). Machine learning. Singapore: McGraw-Hill.

Neural Network Diagram. Accessed 26.4.2021 https://www.astroml.org/book_figures/chapter9/fig_neural_network.html

Vitelli, M., & Nayebi, A. (2016). CARMA : A Deep Reinforcement Learning Approach to Autonomous Driving.