Tillmann Fehrenbach

DiCo & CoViz

A differentiable programming engine for Deep Learning

Bachelor Thesis

Supervisors: Dr. Diaaeldin Taha, Prof. Dr. Felix Joos

Submitted in partial fulfillment of the requirements for the Bachelor of Science degree in 'Informatik' at the Faculty of Mathematics and Computer Science of the University of Heidelberg.

Heidelberg, 2023-06-23

Contents

List of Figures

List of Tables

Abstract

DiCo is a differentiable programming framework based on computation graphs, designed specifically for deep learning. With its client-side and server-less CPU runtime, DiCo-Wasm, and GPU runtime, DiCo-GPU, DiCo can be easily integrated into any JavaScript project. Additionally, we present CoViz, a proof-of-concept for a user-friendly, no-code visual interface to the GPU runtime, which serves as an accessible Neural Network Playground on the Web. This report delves into the theoretical foundations of Computation Graphs, Neural Networks, Automatic Differentiation, and gradient-based optimization algorithms. It also provides a detailed examination of DiCo's implementation using both WebAssembly and WebGPU, as well as the design choices behind CoViz and its development with React.js. To explore CoViz, please visit `https://covizdemo.vercel.app/`. The source code can be accessed on GitHub at `https://github.com/TilliFe/CoViz`.

1 Introduction

DiCo is a computation-graph-based differentiable programming framework that consists of two parts: DiCo-Wasm and DiCo-GPU. Both DiCo-Wasm and DiCo-GPU are frameworks with similar functionality and JavaScript programming interfaces that automate the computation of derivatives in deep learning models. While established deep learning frameworks like PyTorch and TensorFlow have accelerated machine learning research by abstracting away most computationally complex aspects, many people are left wondering about the inner workings of their Neural Network architectures. This report aims to shed some light on their most essential features and shows that is not difficult to replicate and extend the core functionalities.
With the recent advancements in AI research and its rapid adoption by society, many people with no Computer Science background become curious about the technology behind it. However, the technical expertise required to fully understand it can be daunting, especially when it involves interacting with computer code. To address this, we introduce CoViz, a no code programming platform that serves as a visual interface to DiCo-GPU. Rather than presenting users with a black box, CoViz offers a more interactive and intuitive experience through its node-editor-like interface on the web. It may be thought of as as a Lego set for building Neural Networks: users can plug and play with different components, experiment with various architectures, and tweak parameters to see how they affect the network's output. It's a hands-on approach that demystifies the inner workings of deep learning and makes it accessible to anyone interested in learning more about AI. It is important to note that this project is a work in progress and will continue to evolve over time. CoViz is presented as a proof-of-concept.

This report is divided into two sections: one that explores the theoretical concepts behind DiCo and another that delves into various aspects of its design and development process. While the concepts and implementation of DiCo were inspired by other deep learning frameworks such as TensorFlow and PyTorch, this report is not intended to be a detailed case study of those frameworks. Instead, the motivation behind the development of DiCo-Wasm and DiCo-GPU was to experiment with well-established concepts, as discussed in Chapter 2, in two new environments: WebAssembly and WebGPU. Additionally, the goal was to make these concepts more accessible through the user-friendly visual interface of CoViz.

Chapter 2, the Theoretical Background section, provides an overview of the fundamental concepts and techniques underlying a deep learning framework. It starts with an introduction to Computation Graphs, which are used to represent Neural Networks

as a series of interconnected mathematical operations.
The chapter also covers automatic differentiation, an efficient technique for computing gradients of the loss function with respect to model parameters. Gradients play a crucial role in optimizing Neural Networks by providing a measure of how the loss function changes with respect to changes in model parameters. The chapter concludes by discussing stochastic gradient descent and its most popular enhancements.
The presentation of the the different concepts represent our approach to understanding the underlying theory and lay the groundwork for subsequent sections focused on implementing the ideas discussed.

Chapter 3, the Design and Development section, provides an overview of the implementation of DiCo. It starts by realizing the Neural Network Computation Graph by splitting it into three key components: the Node Class (aka. Tensor Class), the Model Class, and the Data Class. The Node Class serves as the fundamental building block of the Computation Graph, encapsulating mathematical operations. The Model Class represents the entire Neural Network and manages forward and backward propagation. The Data Class handles external input data.
DiCo aims to run alongside other JavaScript code on the client-side in a server-less manner utilizing the available hardware on the clients machine. We achieved this goal with two different implementation approaches. The first implementation uses C++/WebAssembly (we will refer to this implementation as DiCo-Wasm), the second one uses WGSL/WebGPU (we will refer to this implementation as DiCo-GPU). WebAssembly is a compilation target for languages other than JavaScript to build high-performance web applications. WebGPU is an emerging graphics API that provides a low-level interface to the clients graphics hardware and enables general purpose programming on the GPU via Compute Shaders. The chapter explains how these technologies were utilized to implement and optimize the Computation Graph's performance. Our discussion on the C++/WebAssembly implementation is brief because we quickly discovered during development that its performance was inferior to that of the WebGPU implementation. However, we still include it in our discussion because it allowed us to implement the concepts of Chapter 2 at a high level before diving into the more complex implementation using Compute Shaders and low-level buffer manipulation in WebGPU.
Finally, we discuss the development of CoViz via React.js, MUI and ReactFlow and give an overview of the available UI components. We present a walk-through on how to use CoViz in a Web Browser to build and train a Neural Network on some exemplary data. For more information about CoViz please visit `https://covizdemo.vercel.app/`.

2 Theoretical Background

In this chapter, we delve into the essential theoretical concepts that form the foundation of the implementation of DiCo, which we will explore in greater detail in the next chapter. We will examine the concept of Computation Graphs, the mathematical representation of Neural Networks, the intricacies of automatic differentiation in both forward and reverse mode, the computation of Neural Network gradients, and a range of gradient-based optimization algorithms. The structure of the presentation of the concepts discussed here mirrors our approach to implementation: while our final application may only utilize a subset of the methods introduced in this chapter, it was this logical progression that guided our understanding and enabled us to construct a more robust and well-organized framework.

2.1 Computation Graphs

In order to introduce the idea of a Computation Graph, consider the following equation $f : \mathbb{R}^2 \to \mathbb{R}$:

$$f(x_1, x_2) = \cos(x_1 x_2) \ln(x_2) \tag{1}$$

The evaluation of the formula can be expressed as a sequence of single operations such as the multiplication or addition of two intermediate results. Table 1 shows the intermediate variables v_i on the left and their corresponding evaluations on the right. It displays the evaluation trace of $f(x_1, x_2)$ at $(x_1, x_2) = (3, 4)$.

Start	v_i	Operation	Evaluation
↓	v_0	$= x_1$	$= 3$
	v_1	$= x_2$	$= 4$
	v_2	$= v_0 v_1$	$= 12$
	v_3	$= \cos(v_2)$	$= 0.84$
	v_4	$= \ln(v_1)$	$= 1.39$
End	v_5	$= v_3 v_4$	$= 1.17$

Table 1: The Computation Trace of Equation (1) evaluated at $(x_1, x_2) = (3, 4)$

We can visualize the trace of these operations in a directed acyclic graph $G = (V, E)$ The Nodes V in a Computation Graph represent intermediate variables, while the directed edges E represent the flow of data between the nodes. Each node has a data value associated with it, which is computed based on the data values of its parent

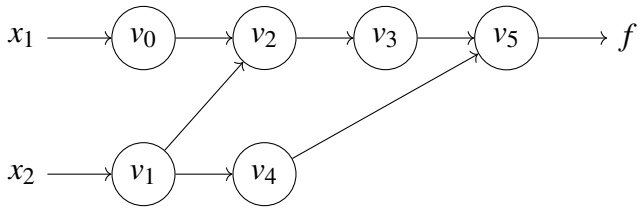

Figure 1: The Computation Graph of Equation (1). Each node represents an elementary operation on the data of its parent nodes.

nodes and any internally defined operation.

The concept of interpreting a computation as a graph is essential for Deep Learning. Computation Graphs have several benefits: They can be optimized for faster and more efficient computation by finding parts of the Graph that can be computed in parallel while others are waiting for the result, yielding a perfect fit for GPU Programming. Many deep learning frameworks have built-in graph optimization techniques that can optimize the graph structure to reduce computation time and memory usage, both during the forward propagation and backward propagation. Additionally, Computation Graphs provide a convenient way to visualize the computation process, which was one of the main motivations behind building CoViz.
There are also inherent limitations when dealing with Computation Graphs: They can be limited in their ability to handle recursion, loops and branching. Loops can cause the graph to become infinitely large, and branching can lead to duplicate nodes and edges, making the graph more complex and difficult to optimize. Hence they are best suited for representing functions that can be broken down into a sequence of elementary operations. Moreover, the overhead of dealing with a graph should not be underestimated since complex graphs might require additional memory and processing power to manipulate.

In the above example we followed the structure of the Graph after finding a sequence of intermediate evaluations. How can we go in the opposite direction? Given a Graph $G = (V,E)$, how do we derive the actual trace of evaluations that we need to perform such that all inputs are considered and no intermediate result gets neglected and is only computed once? The answer to that will be important, since CoViz is inherently visual: The user is able to drag and drop nodes of the Computation Graph and control the flow of data by actually drawing edges between the nodes. A simple Depth First Search starting from the node that we want to compute the value of is sufficient. In our example from above that means starting from f, we perform a Topological Ordering. Sorting the nodes with respect to their Topological ordering yields a sequence for the actual computation. Algorithm 1 explores this in more detail. In order to have access

Input: Graph $G = (V, E)$
Output: Forward Trace of Computation Graph

Reverse the edges of G to obtain $G_{reversed}$;
Initialize an empty list to store the topological ordering;
Initialize a dictionary to store the in-degree of each vertex;
Calculate the in-degree of each vertex;
Create a queue to store vertices with in-degree 0;
while *queue is not empty* **do**
    Dequeue a vertex and add it to the topological ordering;
    Update the in-degree of its neighbors and enqueue any with in-degree 0;
end
Compute the Forward Trace as a list of vertices sorted by their topological number in increasing order;
return Forward Trace;

Algorithm 1: High level pseudocode for computing the sequence of intermediate computations, the 'Forward Trace' of a Computation Graph via non-recursive Topological Ordering.

to all parents of a node, we perform the Topological Ordering on an altered Graph $G_{reversed}$, which is equal to G with all edges reversed.
As we see in the next subsection, such a Forward Trace will be used for a 'Forward Pass' in a Neural Network.

2.2 Neural Networks as Computation Graphs

As we have seen in the last subsection, a Computation Graph can separate a long and convoluted program into smaller intermediate evaluations. We have discussed a way to determine the order of evaluations on such a Graph. We will now discuss how that concept naturally occurs in the context of Multilayer Perceptrons (MLP), which we just call Neural Networks (NN) from now on.
Neural Networks are computational models inspired by the structure and function of animal brains. They are composed of nodes, also known as neurons or units, which receive input from other nodes or external sources and produce an output based on their internal state and the input received [11] . The connections between nodes have associated weights that determine the strength of the influence of one node on another. The layout of a Neural Network can vary, but a common architecture consists of an input layer, one or more hidden layers with activation functions, and an output layer. Activation functions are mostly node-wise functions which introduce non-linearity into the network. In the context of animal brains the purpose of activation functions come close to the threshold an incoming signal at a neuron needs to surpass in order to be

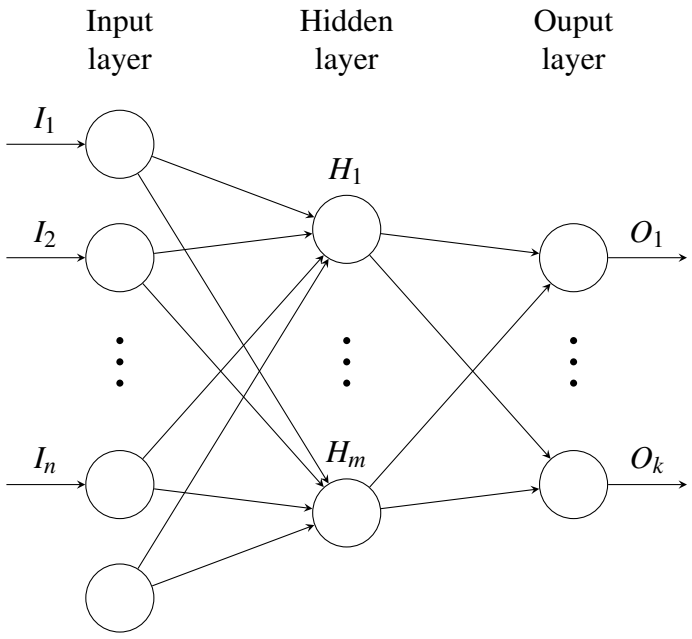

Figure 2: A (single-layer) Neural Network, also known as a multilayer perceptron. The node in the bottom left represents a bias node with an internal value of one. Each edge has an associated weight. During the learning process, these weight parameters are adjusted to approximate the underlying function that generated the data.

passed further to other neurons. Without such activation functions, a Neural Network is equivalent to a linear model, which would limit its ability to learn and represent complex data patterns. For a more detailed discussion on activation functions we refer the reader to [18].

Figure 2 displays the structure of an exemplary artificial Neural Network, assuming that the activation is included in each node. An additional bias node without external input is added in the first layer, which shifts the activation function, improving flexibility and generalization by reducing variance.

We can interpret the functionality of the Neural Network in more abstract terms. The connection of nodes in neighbouring layers can be described in an Adjacency Matrix W. The entry in the i'th row and j'th column of the adjacency matrix of this densly connected sub-graph holds the weight of the edge between the two neurons n_i (in layer l-1) and n_j (in layer l). To compute the vector of all node values z in layer l, the values of the nodes x in the previous layer (l-1) get multiplied with the weighted adjacency matrix, i.e. $z = Wx$. This mathematical abstraction is sometimes called vectorization and is a way of expressing operations on entire arrays rather than individual elements. One major advantage is that it makes the program more concise, the other is that we can use optimization techniques for Linear Algebra Operations such as well established Matrix multiplication Algorithms in order to speed up the computation. We can now begin to talk about the actual problem a Neural Network is meant to solve.

A Neural Network (NN) is a type of function approximator [15] . It can learn to map inputs to outputs in a way that approximates the underlying function that generated the data. Informally speaking, the general objective in Supervised Deep Learning is to find the entries θ of a sequence of (mostly linear) operators such that the difference between the computed output and some ground truth is minimized. In its essence, Neural Network learning is a minimization problem that aims to find:

$$\underset{\theta}{\operatorname{argmin}}\, L(f(x_i;\theta), y_i) \tag{2}$$

where $f(x_i;\theta)$ is the predicted output of the model for input x_i, y_i is the true output for input x_i, and L is some loss function that measures the difference between predicted output and the ground truth. The objective is to minimize this loss function over all training examples.

We can describe the simple one layer NN from Figure 2 in a purely vectorized form. Let $x_i \in \mathbb{R}^n$ be one sample of some training data. Instead of a single sample x_i usually a randomized subset of (batchSize) data samples is considered per input, also called a mini-batch, which generally yields faster learning. We write them as the columns of the Input Matrix $X \in \mathbb{R}^{n \times batchSize}$. Moreover consider the weight matrix $W_1 \in \mathbb{R}^{kxn}$, an activation function such as the Rectified Linear Unit

$$\text{ReLU}\colon \mathbb{R}^{k \times batchSize} \longrightarrow \mathbb{R}^{k \times batchSize} \tag{3}$$

$$x \longmapsto \max\{0, x\} \tag{4}$$

a Bias $b \in \mathbb{R}^{k \times 1}$, an all-ones vector $\mathbf{1} \in \mathbb{R}^{1 \times batchSize}$ and another weight matrix $W_2 \in \mathbb{R}^{m \times k}$. The composition of these operators yields the result of one forward pass through the network:

$$f\colon \mathbb{R}^{n \times batchSize} \longrightarrow \mathbb{R}^{m \times batchSize} \tag{5}$$

$$X \longmapsto W_2(\text{ReLU}(W_1 X + b\mathbf{1})) \tag{6}$$

Let $\hat{y} \in \mathbb{R}^{m \times batchSize}$ be the predicted value, let $y \in \mathbb{R}^{m \times batchSize}$ be some ground truth In order to measure the accuracy of the predicted value, a Loss function, in this exemplary case the mean squared error (MSE) is defined as:

$$\text{MSE}\colon \mathbb{R}^{m \times batchSize} \times \mathbb{R}^{m \times batchSize} \longrightarrow \mathbb{R} \tag{7}$$

$$(\hat{y}, y) \longmapsto \frac{1}{batchSize}\frac{1}{m}||\hat{\mathbf{y}} - \mathbf{y}||_2^2 \tag{8}$$

This yields the function that we want to minimize:

$$L\colon \mathbb{R}^n \times R^n \longrightarrow \mathbb{R} \tag{9}$$
$$(X,y) \longmapsto \mathrm{MSE}(f(X),y) \tag{10}$$

We display the full (forward) evaluation trace of Equation (10) in Table 2.

Start	v_i	Operation
	v_0	$= X$
	v_1	$= W_1$
	v_2	$= b$
	v_3	$= \mathbf{1}$
	v_4	$= W_2$
	v_5	$= y$
	v_6	$= v_1 v_0$
	v_7	$= v_2 v_3$
	v_8	$= v_6 + v_7$
	v_9	$= \mathrm{ReLU}(v_8)$
	v_{10}	$= v_4 v_9$
end	v_{11}	$= \mathrm{MSE}(v_{10}, v_5)$

Table 2: The evaluation Trace of Equation (10).

This sequence of operations can also be understood as a directed acyclic graph as displayed in Figure 3.

2.3 Automatic Differentiation

A common optimization technique for the minimization problem (equation 2) is Gradient Descent, an iterative optimization algorithm that computes the gradients of the objective function with respect to its input parameters and takes small steps towards the negative direction of these gradients. While defining the trace of a computation is relatively simple (section 2.1), finding those gradients is not trivial and is hence mostly automated by modern deep learning frameworks.
The concepts introduced in this section are largely based on the insightful discussion presented in [3] . The approach that we are discussing here to find derivatives is not

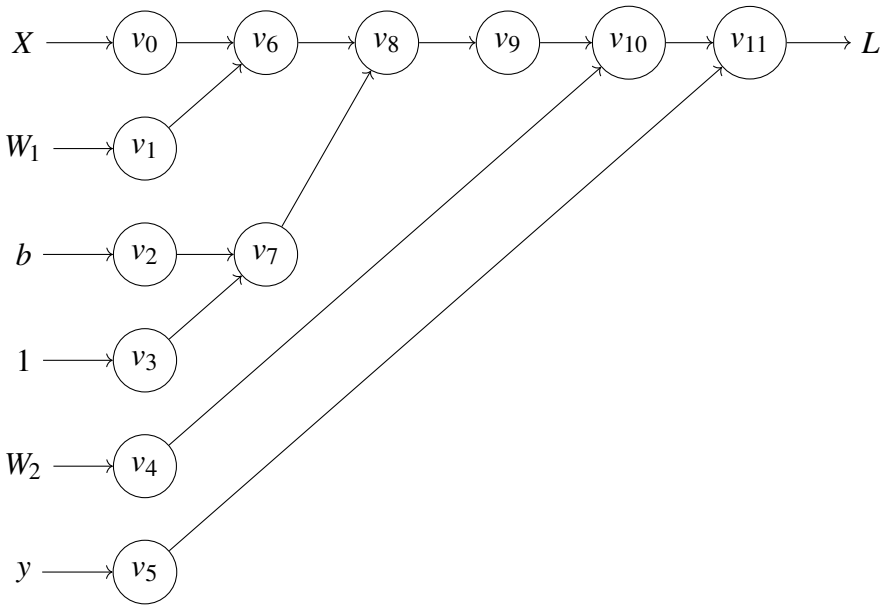

Figure 3: The Computation Graph of Equation (10).

tailored towards Neural Network optimization but is ubiquitous in most of Machine Learning and Scientific Computing. There are three commonly used techniques for computing derivatives in computer programs:

- Numerical differentiation: A technique for approximating the derivative of a function using finite differences. It is based on the limit definition of a derivative and can be used to approximate the gradient of a multivariate function. For example, given a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, one can approximate the gradient $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$ using the following equation:

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x)}{h} \tag{11}$$

  While this method is straightforward to implement, it does have its drawbacks. For instance, it necessitates $\mathcal{O}(n)$ evaluations of f for a gradient in n dimensions and the step size h must be chosen with caution. This makes it infeasible for large (non-vectorized) Computation Graphs or Neural Networks with millions of trainable parameters. Additionally, the accuracy of the approximation can be affected by round-off and truncation errors caused by the limitations of floating point accuracy in computers. A more detailed discussion can be found in any introductory textbook to Numerical Methods such as [5]

- Symbolic differentiation: A method for calculating derivatives by manipulating expressions. It is used in computer algebra systems such as Mathematica and overcomes the limitations of numerical methods. However, it can result in com-

plicated and obscure expressions that are prone to "expression swell." For more information, see [12].

- Automatic differentiation (AD): AD is a method for computing derivatives that guarantees accuracy similar to symbolic differentiation while being relatively simple to implement. There are two modes of automatic differentiation: forward mode and reverse mode. Both modes have their advantages and disadvantages depending on the dimensionality of the given problem. In essence, AD implements the Chain Rule from calculus on complex Computation Graphs.
  The key difference between forward and reverse mode AD lies in their execution order. Both modes initially evaluate all nodes of the Computation Graph from the input nodes to the last node as discussed in section 2.1. Forward mode AD then computes the derivatives of the output with respect to any other node in the same order, often simultaneously with the actual evaluation of the Graph. In contrast, reverse mode AD first performs the evaluation and then computes the gradients from the lastly evaluated node to all the input nodes by propagating them backward through the computation graph.

2.3.1 Forward mode automatic differentiation

Forward mode automatic differentiation (FAD) is a technique for computing the derivative of a function by augmenting the standard computation with the calculation of different derivatives. This is achieved by applying the chain rule to each elementary operation in the evaluation trace of a function.
Consider a function y = f(x) that can be expressed as a composition of m elementary functions $g_1, g_2, \ldots, g_m$, such that $y = g_m(g_{m-1}(\ldots(g_2(g_1(x)))\ldots))$. Let v_i represent the intermediate result of evaluating g_i, so that $v_1 = g_1(x)$ and $v_m = y$. The derivative of y with respect to x can then be computed using the chain rule as follows:

$$\frac{\partial y}{\partial x} = \frac{\partial v_m}{\partial x} = \frac{\partial v_m}{\partial v_{m-1}} \frac{\partial v_{m-1}}{\partial v_{m-2}} \cdots \frac{\partial v_2}{\partial v_1} \frac{\partial v_1}{\partial x} \tag{12}$$

In FAD, the derivative of y with respect to x is computed by evaluating the intermediate derivatives $\frac{\partial v_i}{\partial v_{i-1}}$ alongside the intermediate results v_i. This is achieved by augmenting each elementary operation in the evaluation trace with its corresponding derivative operation. This approach can be generalized to functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with n independent input variables x_i and m output variables y_i. To compute the derivative of the output variables with respect to a specific x_i, $\dot{x}_i$ is set to 1 and the rest to 0. In terms of the Jacobian matrix of f, this means that each forward run computes one column of the matrix. Forward mode automatic differentiation is efficient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ where $n \ll m$. It can be easily implemented by augmenting or overloading operations with their known derivatives.

Example Consider equation (1) $f : \mathbb{R}^2 \to \mathbb{R}$: $f(x_1, x_2) = \cos(x_1 x_2)\ln(x_2)$. We augment the evaluation $f(x_1, x_2)$ at $(x_1, x_2) = (3, 4)$ by its partial derivatives in order to compute $\frac{\partial v_i}{\partial v_{i-1}}$ by setting $\dot{x}_i = 1$. The full evaluation is displayed in Table 3. As we can see, by traversing through the Computation Graph once, we only get the derivative of f with respect to one input parameter.

Direction	v_i	Operation	Evaluation	$\dot{v}_i$	Operation	Evaluation (Deriv.)
Start	v_0	$= x_1$	$= 3$	$\dot{v}_0$	$= \dot{x}_1$	$= 1$
↓	v_1	$= x_2$	$= 4$	$\dot{v}_1$	$= \dot{x}_2$	$= 0$
↓	v_2	$= v_0 v_1$	$= 12$	$\dot{v}_2$	$= \dot{v}_0 v_1 + v_0 \dot{v}_1$	$= 4$
↓	v_3	$= \cos(v_2)$	$= 0.84$	$\dot{v}_3$	$= -\sin(v_2)\dot{v}_2$	$= 2.15$
↓	v_4	$= \ln(v_1)$	$= 1.39$	$\dot{v}_4$	$= \frac{\dot{v}_1}{v_1}$	$= 0$
End	v_5	$= v_3 v_4$	$= 1.17$	$\dot{v}_5$	$= \dot{v}_3 v_4 + v_3 \dot{v}_4$	$= 2.98$

Table 3: FAD Example: The evaluation of Equation (1) at $(x_1, x_2) = (3, 4)$ and the simultaneous computation of the derivative via Forward Mode Automatic Differentiation of f with respect to x_1 which is stored in $\dot{v}_5$ at the end of the forward pass.

In the context of Neural Networks, we usually have a lot of vectorized input parameters and the only output that we evaluate the derivative of is the the Loss function which usually is a scalar. Hence, for the case $f : \mathbb{R}^n \to \mathbb{R}^m$ where n $\gg$m we need a different approach for finding derivatives, which we discuss in the next section.

2.3.2 Reverse mode automatic differentiation

We augment the multivariate chain rule in equation (12), yielding the foundation of reverse mode automatic differentiation (RAD):

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial f}{\partial v_j} \frac{\partial v_j}{\partial x_i} \tag{13}$$

where f is a function of m variables v_1, v_2, ..., v_m, and each v_j is a function of n variables x_1, x_2, ..., x_n.

RAD computes derivatives in two phases. In the first phase, the function code for f is executed in the forward direction to evaluate intermediate variables v_i. In the second phase, adjoints $\bar{v}_i$ are propagated in reverse from the outputs of f to the inputs to calculate the derivatives. We make use of the multivariate chain rule, to compute the

derivative of each node:

$$\bar{v}_i = \frac{\partial f}{\partial v_i} = \sum_{v_j \in \text{children of } v_i} \frac{\partial f}{\partial v_j}\frac{\partial v_j}{\partial v_i} = \sum_{v_j \in \text{children of } v_i} \bar{v}_j \frac{\partial v_j}{\partial v_i} \tag{14}$$

Direction	v_i	Operation	Evaluation	$\bar{v}_i$	Operation	Evaluation
Start Fw	v_0	$= x_1$	$= 3$	$\bar{v}_0$		$= 0$
↓	v_1	$= x_2$	$= 4$	$\bar{v}_1$		$= 0$
	v_2	$= v_0 v_1$	$= 12$	$\bar{v}_2$		$= 0$
	v_3	$= \cos(v_2)$	$= 0.84$	$\bar{v}_3$		$= 0$
	v_4	$= \ln(v_1)$	$= 1.39$	$\bar{v}_4$		$= 0$
End Fw	v_5	$= v_3 v_4$	$= 1.17$	$\bar{v}_5$		$= 0$
Start Bw.				$\bar{v}_5$	$= 1$	$= 1$
↓				$\bar{v}_4$	$= \bar{v}_4 + \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \bar{v}_5 v_3$	$= 0.84$
				$\bar{v}_3$	$= \bar{v}_3 + \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \bar{v}_5 v_4$	$= 1.39$
				$\bar{v}_2$	$= \bar{v}_2 + \bar{v}_3 \frac{\partial v_3}{\partial v_2} = -\bar{v}_3 \sin(v_2)$	$= 0.745$
				$\bar{v}_1$	$= \bar{v}_1 + \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \bar{v}_4 \frac{1}{v_1}$	$= 0.186$
				$\bar{v}_1$	$= \bar{v}_1 + \bar{v}_2 \frac{\partial v_2}{\partial v_1} = \bar{v}_1 + \bar{v}_2 * v_1$	$= 3.16$
End Bw				$\bar{v}_0$	$= \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} = \bar{v}_2 v_1$	$= 2.98$

Table 4: RAD example: The evaluation of Equation (1) at $(x_1, x_2) = (3, 4)$ (Fw) and the subsequent computation of the derivatives via Reverse Mode Automatic Differentiation of f with respect to x_1 AND x_2 which is stored in $\bar{v}_0$ and $\bar{v}_1$ respectively at the end of the backward pass (Bw).

In Table 4 we show the forward (Fw) and backward (Bw) trace of the Computation Graph of equation (1). In the forward pass we set all derivatives $\bar{v}_i$ to zero. Starting from the last node we then perform a modified Breadth first Search, allowing nodes to be visited as often as their respective number of children. In each visit of a node, we compute the intermediate derivative with equation (8). Compared to FAD, we now automatically compute the derivative to all n inputs. In terms of the Jacobian matrix of f, this means that for each backward run we compute one row of the matrix. This solves the problem of forward mode AD for Neural Networks with a lot of vectorized input parameters and only one output.

2.4 Gradients of Neural Networks

We will now delve into the detailed calculation of derivatives for vectorized operations within a Computation Graph. This includes finding the partial derivatives of matrix-matrix multiplications, element-wise functions like ReLU and softmax, as well as two commonly used loss functions: The mean squared error loss (MSE) which measures accuracy of predicted values in regression problems, and cross entropy loss (CE) which measures accuracy in classification problems. The goal of this section is to provide a practical understanding without being mathematically rigorous in every step of the computations, yielding to some abuses of notation. The 'Theorems' and their high level Proofs are largely based on Stanfords cs224n's lecture notes[6].

Theorem 2.1. *Let z be a vector such that $z = x$. Then the Jacobian matrix of z with respect to x is the identity matrix, i.e., $\frac{\partial z}{\partial x} = I$.*

Proof. Since $z_i = x_i$, we have

$$(\frac{\partial z}{\partial x})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} x_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Thus, the Jacobian matrix $\frac{\partial z}{\partial x}$ is a diagonal matrix with entries equal to 1 on the main diagonal, i.e. $\frac{\partial z}{\partial x} = I$. □

Theorem 2.2. *Let $z = f(x)$ be a vector where f is applied element-wise. Then the Jacobian matrix of z with respect to x is a diagonal matrix with entries equal to the derivative of f applied to the corresponding entries of x, i.e., $\frac{\partial z}{\partial x} = diag(f'(x))$.*

Proof. Since f is applied element-wise, we have $z_i = f(x_i)$. Thus,

$$(\frac{\partial z}{\partial x})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} f(x_i) = \begin{cases} f'(x_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Hence, the Jacobian matrix $\frac{\partial z}{\partial x}$ is a diagonal matrix where the entry at (i,i) is the derivative of f applied to x_i. We can write this as $\frac{\partial z}{\partial x} = \text{diag}(f'(x))$. □

Theorem 2.3. *Let $z = Wx$ where $W \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^{n \times 1}$, let $\delta = \frac{\partial L}{\partial z}$ with $\delta \in \mathbb{R}^{m \times 1}$ and J some (Loss) function. then $\frac{\partial L}{\partial W} = \delta x^T$*

Proof. We aim to compute the derivative of a (scalar) loss function L with respect to the the entries of the matrix $W \in \mathbb{R}^{m \times n}$. Instead of computing the Jacobian $\frac{\partial L}{\partial W}$ which would be $1 \times nm$ vector, it is more useful to arrange the gradient in an $m \times n$ matrix which is a slight abuse of notation, however, since this matrix has the same shape as

W, we can easily subtract it later (times the learning rate) from W when doing gradient descent:

$$\frac{\partial L}{\partial W} = \begin{bmatrix} \frac{\partial L}{\partial W_{11}} & \cdots & \frac{\partial L}{\partial W_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial W_{m1}} & \cdots & \frac{\partial L}{\partial W_{mn}} \end{bmatrix}$$

We now compute $\frac{\partial z}{\partial W_{ij}}$:

$$z_k = \sum_{l=1}^{n} W_{kl} x_l,$$

so

$$\frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^{n} x_l \frac{\partial}{\partial W_{ij}} W_{kl}$$

Note that $\frac{\partial}{\partial W_{ij}} W_{kl} = 1$ if $i = k$ and $j = l$, and 0 otherwise. Thus,

$$\frac{\partial z_k}{\partial W_{ij}} = x_j \text{ if } k = i \text{ and 0 if otherwise.}$$

therefore

$$\frac{\partial J}{\partial W_{ij}} = \delta \frac{\partial z}{\partial W_{ij}} = \sum_{k=1}^{n} \delta_k \frac{\partial z_k}{\partial W_{ij}} = \delta_i x_j.$$

We can therefore write $\frac{\partial J}{\partial W}$ as an outer product of δ and x:

$$\frac{\partial L}{\partial W} = \delta x^T$$

□

In the setting of a Neural Network $x \in \mathbb{R}^{n \times 1}$ holds one sample of data that is being transformed by a weight matrix $W \in \mathbb{R}^{m \times n}$ from one Network Layer to the next. In order to speed up the learning process one usually takes several samples of data at a time, also called a mini-batch. It is common to take a weighted average of the derivatives of each sample of the mini-batch. We can therefor augment our above theorem to:

Theorem 2.4. *Let $Z = WX$ where $W \in \mathbb{R}^{m \times n}$ and $X \in \mathbb{R}^{n \times batchSize}$, let $\Delta = \frac{\partial L}{\partial Z}$ with $\Delta \in \mathbb{R}^{m \times batchSize}$ and L some (scalar) Loss function. then*

$$\frac{\partial L}{\partial W} = \frac{1}{batchSize} \Delta X^T$$

Theorem 2.5. *Let $Z = WX$ where $W \in \mathbb{R}^{m \times n}$ and $X \in \mathbb{R}^{n \times batchSize}$, let $\Delta = \frac{\partial L}{\partial Z}$ with $\Delta \in \mathbb{R}^{m \times batchSize}$ and L some (Loss) function then the weighted average of derivatives*

over all samples of a single batch is

$$\frac{\partial L}{\partial X} = \frac{1}{batchSize} W^T \Delta$$

Theorem 2.6. *Let* $\mathbf{y} \in \mathbb{R}^{n \times batchSize}$ *be the matrix of true values and* $\hat{\mathbf{y}} \in \mathbb{R}^{n \times batchSize}$ *be the matrix of predicted values. The gradient of the mean squared error (MSE) loss function with respect to the predicted values* $\hat{\mathbf{y}}$ *is given by:*

$$\frac{\partial MSE}{\partial \hat{\mathbf{y}}} = \frac{1}{n} \frac{2}{batchSize} (\hat{\mathbf{y}} - \mathbf{y})$$

Proof. The MSE can be written as:

$$\text{MSE} = \frac{1}{n} \frac{1}{batchSize} ||\hat{\mathbf{y}} - \mathbf{y}||_2^2$$

where $||\cdot||_2$ denotes the L2 norm. The partial derivative of the MSE with respect to the predicted values $\hat{\mathbf{y}}$ is:

$$\frac{\partial \text{MSE}}{\partial \hat{\mathbf{y}}} = \frac{\partial}{\partial \hat{\mathbf{y}}} \left(\frac{1}{n} \frac{1}{batchSize} ||\hat{\mathbf{y}} - \mathbf{y}||_2^2 \right)$$

Using the chain rule, we can expand this as:

$$\frac{\partial \text{MSE}}{\partial \hat{\mathbf{y}}} = \frac{1}{n} \frac{2}{batchSize} (\hat{\mathbf{y}} - \mathbf{y})$$

□

Theorem 2.7. *Let* $z \in \mathbb{R}^n$ *and* $\sigma(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}$ *be the softmax function. Then the derivative of* $\sigma(z)$ *with respect to* z_j *is given by*

$$\frac{\partial \sigma(z)_i}{\partial z_j} = \sigma(z)_i (\delta_{ij} - \sigma(z)_j),$$

where δ_{ij} *is the Kronecker delta.*

Proof. We start by computing the derivative of $\sigma(z)_i$ with respect to z_j:

$$\begin{aligned}
\frac{\partial \sigma(z)_i}{\partial z_j} &= \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}\right) \\
&= \frac{\partial}{\partial z_j}\left(e^{z_i}\left(\sum_{k=1}^{n} e^{z_k}\right)^{-1}\right) \\
&= e^{z_i}\left(\sum_{k=1}^{n} e^{z_k}\right)^{-2}(-e^{z_j}) \\
&= -e^{z_i}e^{z_j}\left(\sum_{k=1}^{n} e^{z_k}\right)^{-2}.
\end{aligned}$$

Now we consider two cases: $i = j$ and $i \neq j$. If $i = j$, then

$$\begin{aligned}
\frac{\partial \sigma(z)_i}{\partial z_j} &= -e^{2z_i}\left(\sum_{k=1}^{n} e^{z_k}\right)^{-2} + e^{z_i}\left(\sum_{k=1}^{n} e^{z_k}\right)^{-1} \\
&= -\sigma(z)_i^2 + \sigma(z)_i \\
&= \sigma(z)_i(1-\sigma(z)_j).
\end{aligned}$$

If $i \neq j$, then

$$\frac{\partial \sigma(z)_i}{\partial z_j} = -\sigma(z)_i\sigma(z)_j.$$

Thus, we have shown that

$$\frac{\partial \sigma(z)_i}{\partial z_j} = \sigma(z)_i(\delta_{ij} - \sigma(z)_j).$$

as desired. □

Theorem 2.8. *Let $y, \hat{y}, z \in \mathbb{R}^n$ represent the ground truth, predicted values, and weighted input, respectively. Let σ denote the softmax function, defined as $\sigma(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$. Let $\hat{y} = \sigma(z)$. Let CE_j denote the cross-entropy loss for the j-the output, defined as $CE_j = -y_j \ln(\hat{y}_j) + (1-y_j)\ln(1-\hat{y}_j)$. The partial derivative of CE_j with respect to z_j can be simplified to $\hat{y}_j - y_j$. In vectorized notation, this can be written as*

$$\frac{\partial CE}{\partial z} = \hat{y} - y.$$

Proof. The partial derivative of CE_j with respect to z_j is:

$$\frac{\partial CE_j}{\partial z_j} = \frac{\partial CE_j}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial z_j}$$

$$= \left[-\frac{y_j}{\hat{y}_j} + \frac{1-y_j}{1-\hat{y}_j} \right] \sigma'(z_j)$$

Since we're using σ as the activation function, we have:

$$\sigma'(z_j) \overset{\text{Th.2.8}}{=} \hat{y}_j(1-\hat{y}_j)$$

Substituting this expression into our equation for $\frac{\partial CE_j}{\partial z_j}$, we get:

$$\begin{aligned} \frac{\partial CE_j}{\partial z_j} &= \left[-\frac{y_j}{\hat{y}_j} + \frac{1-y_j}{1-\hat{y}_j} \right] \hat{y}_j(1-\hat{y}_j) \\ &= [-y_j + y_j\hat{y}_j + \hat{y}_j - y_j\hat{y}_j] \\ &= \hat{y}_j - y_j \end{aligned}$$

In vectorized notation, this can be written as $\frac{\partial CE}{\partial z} = \hat{y} - y$. □

Theorem 2.9. *Let $Y, \hat{Y}, Z \in \mathbb{R}^{n \times batchSize}$ represent the ground truth, predicted values, and weighted input matrices, respectively. Let σ denote the softmax function applied element-wise to a matrix. Let $\hat{Y} = \sigma(Z)$. In vectorized notation the partial derivative of the cross-entropy loss matrix CE with respect to Z is*

$$\frac{\partial CE}{\partial Z} = \frac{1}{batchSize}(\hat{Y} - Y).$$

2.5 Computing the Gradients of a NN - a small example

We go back to the simple Neural Network described in equation (10). Using the discussed theorems in the previous section and the concept of reverse mode automatic differentiation (RAD), we can now compute the derivatives of the Loss function wrt. to all input parameters. Finding the derivative in this simple example naturally extends to more complex (deeper and more parallel/less sequential) Neural Network models. We start by doing a forward propagation through the network with the Loss function as the last node of the Computation Graph and set the derivatives wrt. all intermediate variables to zero. In the backward propagation we accumulate the gradients for all nodes as we did in the simple example in Table 4 by multiplying gradients and partial derivatives. The full trace of evaluations (Forward Pass (Fw) and Backward Pass (Bw)) is displayed in Table 5. We can see, that for all the five independent inputs (X, W_1, b, W_2, y) only one additional backward pass was necessary to compute all respective gradients.

Direction	v_i	Operation	$\bar{v}_i$	Evaluation
Start Fw	v_0	$= X$	$\bar{v}_0$	$= 0$
	v_1	$= W_1$	$\bar{v}_1$	$= 0$
	v_2	$= b$	$\bar{v}_2$	$= 0$
	v_3	$= \mathbf{1}$	$\bar{v}_3$	$= 0$
	v_4	$= W_2$	$\bar{v}_4$	$= 0$
	v_5	$= y$	$\bar{v}_5$	$= 0$
	v_6	$= v_1 v_0$	$\bar{v}_6$	$= 0$
	v_7	$= v_2 v_3$	$\bar{v}_7$	$= 0$
	v_8	$= v_6 + v_7$	$\bar{v}_8$	$= 0$
	v_9	$= \text{ReLU}(v_8)$	$\bar{v}_9$	$= 0$
	v_{10}	$= v_4 v_9$	$\bar{v}_{10}$	$= 0$
End Fw	v_{11}	$= \text{MSE}(v_{10}, v_5)$	$\bar{v}_{11}$	$= 0$
Start Bw			$\bar{v}_{11}$	$= 1$
			$\bar{v}_{10}$	$\overset{\text{Th.2.6}}{=} \bar{v}_1 0 + \bar{v}_{11}\text{MSE'}(v_{10}, v_5)$
			$\bar{v}_5$	$\overset{\text{Th.2.6}}{=} \bar{v}_5 + \bar{v}_{11}\text{MSE'}(v_{10}, v_5)$
			$\bar{v}_9$	$\overset{\text{Th.2.5}}{=} \bar{v}_9 + v_4^T \bar{v}_{10}$
			$\bar{v}_4$	$\overset{\text{Th.2.4}}{=} \bar{v}_4 + \bar{v}_{10} v_9^T$
			$\bar{v}_8$	$\overset{\text{Th.2.2}}{=} \bar{v}_8 + \text{ReLU'}(\bar{v}_9)$
			$\bar{v}_6$	$\overset{\text{Th.2.1}}{=} \bar{v}_6 + \bar{v}_8$
			$\bar{v}_7$	$\overset{\text{Th.2.1}}{=} \bar{v}_7 + \bar{v}_8$
			$\bar{v}_0$	$\overset{\text{Th.2.5}}{=} \bar{v}_0 + v_1^T \bar{v}_6$
			$\bar{v}_1$	$\overset{\text{Th.2.4}}{=} \bar{v}_1 + \bar{v}_6 v_0^T$
			$\bar{v}_2$	$\overset{\text{Th.2.4}}{=} \bar{v}_2 + \bar{v}_7 v_3^T$
End Bw			$\bar{v}_3$	$\overset{\text{Th.2.5}}{=} \bar{v}_3 + v_2^T \bar{v}_7$

Table 5: The evaluation of a simple Neural Network as described in Equation (10) (the Forward Pass Fw) and the subsequent computation (the Backward Pass Bw) of the derivatives via Reverse Mode Automatic Differentiation with respect to ALL input Parameters

2.6 NN initialization and Optimization Algorithms

We have discussed the theoretical foundation for performing a full forward and backward pass in a Neural Network. In the following, we will assume that the parameters

of a Neural Network (i.e., the entries of the adjacency matrix between neighboring layers of the NN) have been randomly initialized. The choice of initialization method can vary and can greatly affect the learning performance. As a default, we use "He" initialization for a NN with ReLU activation functions [14]. In He initialization, the weights of a NN layer with n neurons are initialized using a normal distribution with a mean of 0 and a standard deviation of $\sqrt{\frac{2}{n}}$.
To iteratively minimize the loss of a NN, the procedure of performing a forward pass, backward pass with gradient computation, and parameter update is repeated.
As a side note, it is neither necessary to send the entire training dataset through the network in every training iteration, nor is it recommended to only use one sample. Instead, a mini-batch, which is a randomly selected subset of the training data, is typically used in each iteration. The algorithm that computes and updates the weights based on this random selection of data samples is called Stochastic Gradient Descent (SGD). Several improvements to this algorithm have been proposed, such as SGD with Momentum, RMSProp, and Adam, which we will briefly discuss here and present pseudocode that resembles our implementation. In our deep learning framework, we allow the user to specify the optimization algorithm to be used.

- Stochastic Gradient Descent (SGD) is an optimization technique used to minimize a given function by iteratively moving towards the direction of steepest descent, as determined by the negative gradient. It is frequently used in the training of Neural Networks. SGD was first introduced in a paper on stochastic approximation by Robbins and Monro in 1951 [25]. The pseudocode for this optimizer is presented in Algorithm 2.

- SGD with momentum is a variation of SGD that uses an exponentially weighted average of past gradients to update the parameters [20]. . This helps to accelerate the optimization process by dampening oscillations and allowing faster convergence. The momentum term can be seen as a ball rolling down a hill, gaining speed in directions with persistent gradients. Pseudocode for this optimizer can be seen in Algorithm 3.

- RMSprop is an optimization technique that normalizes the gradient by utilizing an exponentially weighted moving average of squared gradients. This process helps to balance the step size by decreasing it for large gradients to prevent exploding and increasing it for small gradients to prevent vanishing. Instead of treating the learning rate as a hyper-parameter, RMSprop employs an adaptive learning rate. The pseudocode for this optimizer is shown in Algorithm 4.

- Adam: Adam is an optimization algorithm that combines the advantages of Gra-

dient Descent with momentum and RMSprop [16] . Adam has been shown to work well in practice and is widely used in deep learning. Pseudocode for this optimizer can be seen in Algorithm 5.

Result: Updated parameters θ
Initialize θ, the parameters of the Neural Network;
Set hyper-parameter: learning rate α;
for *each epoch* **do**
    for *each mini-batch* **do**
        Compute gradient g of the loss w.r.t. θ;
        Update parameters: $\theta \leftarrow \theta - \alpha g$;
    end
end

Algorithm 2: Stochastic Gradient Descent

Result: Updated parameters θ
Initialize θ, the parameters of the Neural Network;
Initialize v, the velocity vector;
Set hyper-parameters: learning rate α, momentum β;
for *each epoch* **do**
    for *each mini-batch* **do**
        Compute gradient g of the loss w.r.t. θ;
        Update velocity: $v \leftarrow \beta v + (1-\beta)g$;
        Update parameters: $\theta \leftarrow \theta - \alpha v$;
    end
end

Algorithm 3: Stochastic Gradient Descent with Momentum

Result: Updated parameters θ
Initialize θ, the parameters of the Neural Network;
Initialize r, the running average of squared gradients;
Set hyperparameters: learning rate α, decay rate ρ;
for *each epoch* **do**
  for *each mini-batch* **do**
    Compute gradient g of the loss w.r.t. θ;
    Update running average: $r \leftarrow \rho r + (1-\rho)g^2$;
    Compute update: $\Delta\theta = -\frac{\alpha}{\sqrt{r}+\varepsilon}g$;
    Update parameters: $\theta \leftarrow \theta + \Delta\theta$;
  end
end

Algorithm 4: RMSprop

Result: Updated parameters θ
Initialize θ, the parameters of the Neural Network;
Initialize m, the first moment vector;
Initialize v, the second moment vector;
Set hyper-parameters: learning rate α, decay rates β_1 and β_2;
$t \leftarrow 0$;
for *each epoch* **do**
  for *each mini-batch* **do**
    Compute gradient g of the loss w.r.t. θ;
    Update biased first moment estimate: $m \leftarrow \beta_1 m + (1-\beta_1)g$;
    Update biased second moment estimate: $v \leftarrow \beta_2 v + (1-\beta_2)g^2$;
    Compute bias-corrected first moment estimate: $\hat{m} = \frac{m}{1-\beta_1^t}$;
    Compute bias-corrected second moment estimate: $\hat{v} = \frac{v}{1-\beta_2^t}$;
    Compute update: $\Delta\theta = -\alpha\frac{\hat{m}}{\sqrt{\hat{v}}+\varepsilon}$;
    Update parameters: $\theta \leftarrow \theta + \Delta\theta$;
  end
  $t \leftarrow t+1$;
end

Algorithm 5: Adam

3 Design and Development of the Framework

This chapter delves into the design and development of DiCo. It begins by examining the general realization of the Computation Graph and the data held by each node. The chapter then explores the implementation of the framework through both C++/Web-Assembly (DiCo-Wasm) and WebGPU (DiCo-GPU), which are referred to as the framework's backend. Finally, the chapter takes a closer look at the design and development of CoViz, a visual interface built for DicoGPU with React.js [24] which we also refer to as the framework's frontend, and demonstrates its capabilities by building and training a small Neural Network for regression and classification problems. The hierarchy of the used technologies and the respective abstraction layers can be seen in Figure 4.

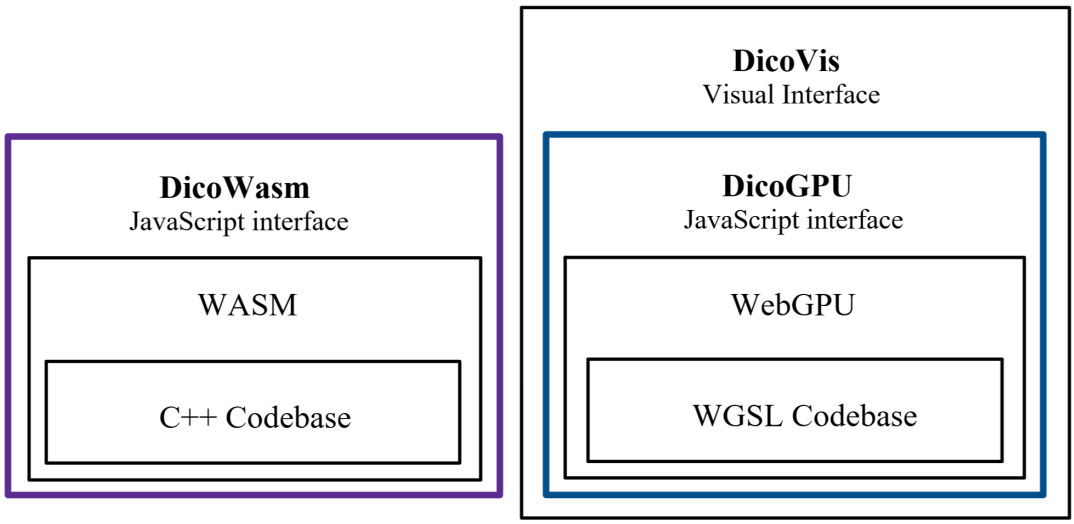

Figure 4: Hierarchy of Technologies used in DiCo. There are two implementations of the differentiable Computation Graph, in WebAssembly (Wasm) with a C++ code base and WebGPU with a WGSL (shader) code base. The respective next layer of abstraction is a JavaScript interface. DiCo-GPU additionally provides a visual interface, called CoViz.

3.1 Computation Graph: Architecture and Implementation

The differentiable Computation Graph of a Neural Network comprises nodes that represent intermediate data and edges that represent the flow of data between nodes. We now present some details of the three main objects that occur in both of our frameworks backend implementations: the 'Node' (aka. 'Tensor') class, the 'Model' class and the 'Data' class.

3.1.1 Node Class (aka. Tensor Class)

Nodes can contain static data, data that changes during the learning process, or data resulting from an operation performed on its parent nodes. In our framework, we sometimes refer to nodes as 'tensors', similar to other popular frameworks such as TensorFlow [1] and PyTorch [22] . Each node has the following attributes:

- **Id**: A unique identifier for the node.
- **Type**: A node in the Graph can either hold data or be an operator that performs an operation on its parent nodes' data and stores the result in its own data matrix. Currently implemented operators include addition, multiplication, softmax, activation functions (ReLU and Tanh), and loss functions (mean squared error and cross-entropy error). A convolution operation is underway.
- **Rows m and Columns n + MetaDims**: The dimensions of both the data and gradient data matrices. It is important to keep track of the dimensionality, since all the data of a node is flattened to a one dimensional buffer if it is sent to the GPU. If the tensor shall not represent an actual matrix such as the weight matrix but rather the result of some operation, the number of columns represent the batchSize of the training process and each column hold the result of one sample of a mini-batch. For example in image processing with convolutional filters, it is necessary to have access to the original dimensions of each sample (e.g. width a_1 and height a_2 etc. of an image) in the columns of the matrix, hence MetaDims is another array of number $[a_1, a_2, .., a_n]$ such that $a_1 * a_2 * ... * a_n = m$. Rows m, Columns n and MetaDims are currently adjustable by the user of DiCo. In other frameworks like PyTorch, the user only specifies MetaDims for a tensor and everything else is handles internally. Eventually this is our goal as well.
- **Data**: A real-valued $m \times n$ matrix holding data based on the type of the node. If the type of a node is set to 'none' which means that it has no input but is only a data provider, we can choose between different initialization modes of the data matrix: 'random' (random values sampled from a Normal or Gaussian distribution), 'all Zeros' and 'all Ones'. Alternatively the data node can provide some external data (see Data Class).
- **Gradient Data**: A real-valued $m \times n$ matrix that gets filled with derivative values during a backward propagation.
- **First moment estimate** (Velocity): A real-valued $m \times n$ matrix used for dampening oscillations and allowing faster convergence when using SGD with momentum and Adam optimization.

- **Second order estimate**: A real-valued $m \times n$ matrix used for RMSProp and Adam optimization.
- **Parents**: Nodes in the Computation Graph that have an edge pointing to the current node.
- **Children**: Nodes that the current node points to.
- **Requires Gradient**: A boolean value that determines whether the partial derivative with respect to the current node and the gradient of the current node should be computed. This can greatly reduce redundant gradient computations for nodes that do not need to be updated, such as the input node or the node that holds the ground truth. The user of the Framework can choose if this attribute is set to true or false. If a node has no parents, it is set to true by default. If a node has parents, the node performs a logical OR between the 'Requires Gradient' Attributes of all Parent Nodes: If the 'Requires Gradient' attribute of at least one node in Parents is true, then the error must be propagated through the current node as well.

3.1.2 Model Class

All nodes with their respective attributes and relationships are stored in an object of type 'Model'. The Model keeps track of the architecture of the Neural Network and stores the hyperparameter values. It also defines which node in the Graph shall be computed i.e. it stores the id of the last node in the computation trace. If the last node is an operator such as 'Mean Squared Error' or 'Cross Entropy' which measure the accuracy of the predicted value given some ground truth, we can select 'training mode' to perform the iterative optimization via Gradient Descent, i.e. let the Neural Network learn. The specific attributes of the class 'Model' are:

- **Nodes** (or Tensors): A list of all the nodes in the Computation Graph with their respective attributes and relationships.
- **Input Tensor Id**: The Id of the Node which repeatedly provides new Data which is then passed through the network.
- **Last Tensor Id**: The Id of the computed output node, i.e the last node in the computation trace.
- **Ground Truth Tensor Id**: Id of the node which hold the labels/ground truth with respect to the input data.
- **Outputs Tensors Ids**: The set of Ids of the tensors whose attributes are repeatedly printed out or visualized during a learning process.
- other **Hyperparameters**: learning rate, first order momentum (scalar ratio), second order momentum (scalar ratio) etc.

3.1.3 Data Class

If the 'Input Tensors Id' and the 'Ground Truth Tensor id' of the Model object are set, an object of type 'Data' is being created which repeatedly provides new input data to the Input Tensor and its respective true Values to the Ground Truth Tensor. In the current implementation of DiCo-Wasm and DiCo-GPU, data on which a Neural Network is trained on, is read from a local CSV file. In CoViz data is provided directly via URL.

3.2 Implementation in C++/WebAssembly

We examine DiCo-Wasm, our first implementation of the classes discussed in the previous section, using the concepts introduced in Chapter 2. Our initial plan was to write the whole Framework in C++ (without speed ups through parallelization by the GPU) and then compile it via the Emscripten [10] Compiler to WebAssembly [27] code in order to make it run inside a Browser. Even though we achieved this goal and successfully trained a Neural Network to carry out the 'MNIST' [17] digit recognition task as a 'Hello World program' in the Browser, we stopped its development early, since it performed poorly compared to our later WebGPU implementation. However, we still present the key concepts and an example of how to use the Framework here, since the ideas naturally transfer to the later implementation.
C++ is an object oriented programming language that enables low memory access. This comes in handy when dealing with large data such as large matrices as they appear in the Computation Graph of a Neural Network. We implemented the Model, Node and Data class as described in the previous section. The edges in the Computation Graph (Model) are implemented via Smart Pointers in the list of parents and children of a each node (Tensor). This allows a node to compute new data based on its parents or children by directly accessing the respective data without performing deep copies, which yields great speed ups. We make use of the Boost Library [4], providing access to highly optimized Linear Algebra operations such as fast dense matrix multiplications. The forward pass through a pragmatically defined Computation Graph is performed as described in Section 2.1 and during training the backward pass performs the Parameter update via Reverse Mode Automatic Differentiation as described in Section 2.3,2.5 and 2.6 without interference by the user.

We present the creation of a Neural Network in Figure 5 using our C++ implementation with the following specifications.

- Initialization of the Model and specification of the data that is used, here the 'MNIST' data set. One input sample is of size 784 (the number of pixels per image of a handwritten digit) and the ground truth or 'reference data' is of size

1 (a digit form zero to nine).

- A tensor 'input' gets a batch of data and passes it through two 'dense' layers of the Neural Network of 15 neurons each. This architecture uses Cross entropy as the Loss function, which is commonly used for measuring the Error of binary classification problems.

- The model perform 1000 iterations on the data set, that means we perform 1000 forward and backward passes on the Computation Graph, with a batch of randomly selected samples from the data set respectively. The learning rate is 0.1 and the model uses Stochastic Gradient Descent with momentum (set to 0.9 by default). The batch Size is 64 which means we compute the classification of 64 images of handwritten digits in every iteration and compute the weighted average Loss respectively. The number 50 in the train() function specifies the frequency of printing out the error of a forward pass. This helps to keep track of the learning process from the console.

```
#include "./include/model.hh"
#include "./include/tensor.hh"

int main(){

  // initalize and configure the Model
  auto model = nn::Model();
  model.setBatchSize(64);
  model.setInputData("MNIST.csv",1,42000,1,784);
  model.setReferenceData("MNIST.csv",1,42000,0,0);
  model.setRandomInput();

  // define the models architecture
  auto input = model.getInputBatch();
  auto trueValue = model.getReferenceBatch();
  auto L1 = model.Dense(15,"ReLU",input);
  auto L2 = model.Dense(15,"ReLU",L1);
  auto output = model.Dense(10,"none",L2);
  auto out = model.Loss(output,trueValue,"softmax_CE");

  // train the model
  model.train(1000,0.1,"SGD_momentum", 50);
}
```

Figure 5: Example of defining a NN in the C++ implementation. The keyword 'auto' is a smart pointer to the node/tensor object returned by the model object after the respective creation. This returned smart-pointer can then be used by other nodes/tensors to build up the Computation Graph. The model class keeps track of the creation of all objects (hence the smart-pointers) and manages all complex computations such as the Forward and Backward Pass during training.

A class such as 'Model', 'Data' and 'Tensor' including its methods can be fully com-

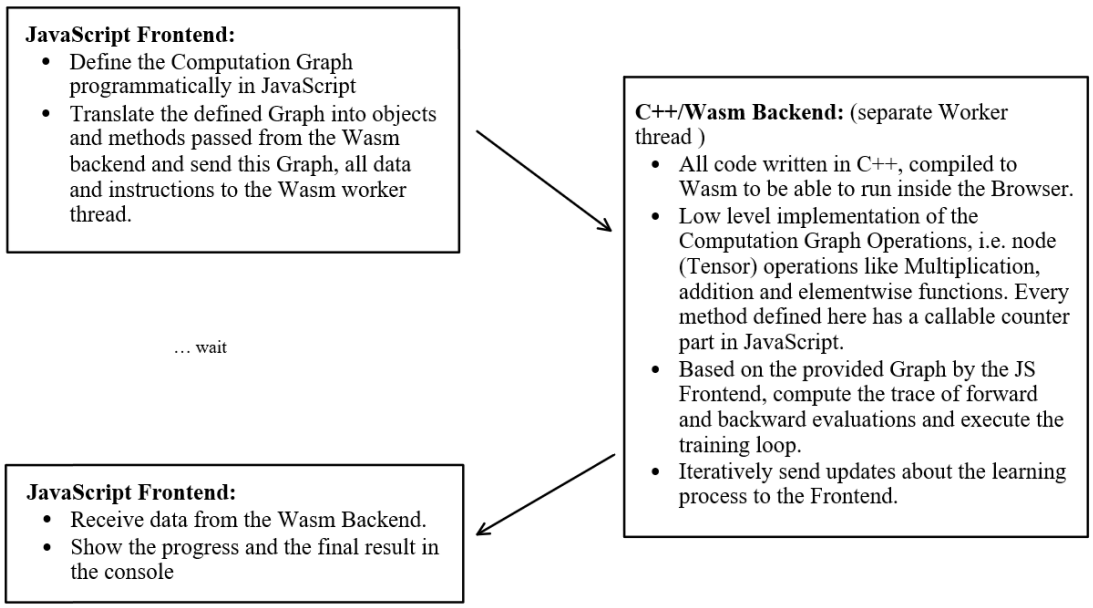

Figure 6: Data flow between a JavaScript and WebAssembly worker.

piled to Objects in WebAssembly. WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. It is designed as a portable compilation target for several programming languages such as C/C++, Python and Rust which enables the code to run in most modern web browsers. It is also built to operate with JavaScript, enabling them to function in tandem.
Hence there is a one to one correlation between the C++ code specified in Figure 5 and our JavaScript interface. The code would look almost identical except using the JavaScript variable declarations such as 'let' or 'const' instead of 'auto'. Even the smart pointers are passed through the JavaScript interface returned by an operation on the 'model'. A high level data flow visualization between the JavaScript frontend and the Wasm backend is shown in Figure 6.

3.3 Implementation via WebGPU

3.3.1 What is WebGPU?

To accelerate Neural Network computations, GPUs are utilized to parallelize specific operations. As a result, we transitioned from our implementation in C++ (in combination with WebAssembly) to WebGPU [28] , an emerging standard for GPU programming in web browsers. One of WebGPU's benefits is the ability to create Compute Shaders, which was not possible with earlier versions of Web Graphics APIs such as WebGL.
In general, a shader is a small program that is sent from the CPU to the GPU as a

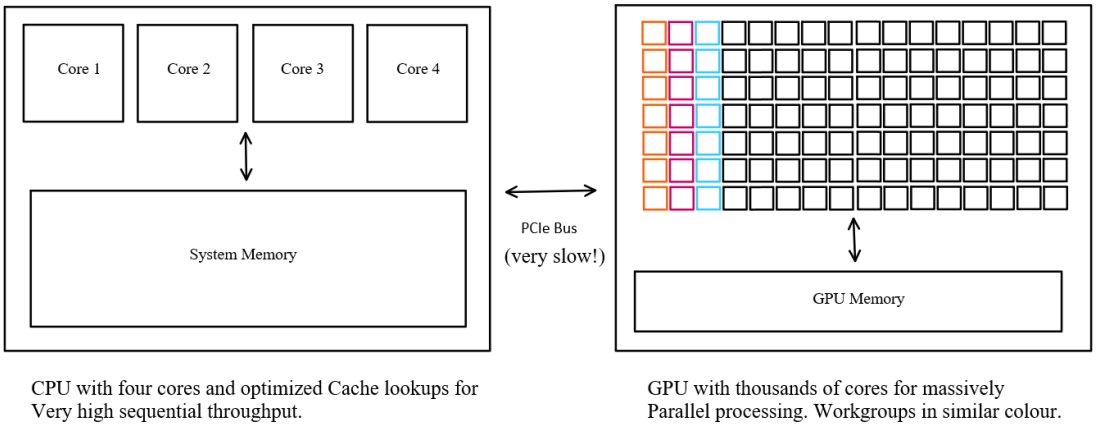

Figure 7: Abstract visualization of CPU and GPU.

string of code. This code is then compiled and executed on one of the many threads (also known as kernels) on the GPU. There are different types of shaders, including compute shaders and regular shaders (such as vertex and fragment shaders), each with their own advantages and use cases. Compute shaders are used for performing arbitrary computations and can be used for tasks that are not directly related to rendering graphics. Unlike CPUs, GPUs typically have thousands of threads running simultaneously, enabling massively parallelized operations known as SIMD (Single Instruction Multiple Data) operations. A classic example of a SIMD operation is matrix multiplication. In the following section we will discuss the basic building blocks of a GPU and its memory management and the communication with the CPU.

3.3.2 Workitems, Workgroups and Workloads

A GPU has similar building blocks as a CPU, the distribution of its elements is however vastly different [2] . A high level comparison between both architectures can be seen in Figure 6. A CPU usually consist of a couple of cores, each which can run exactly one computation at a time. A CPU has a hierarchical Memory structure with different layers of Caches for efficient memory access and large cores with very fast clock speeds. It is optimized for throughput i.e. execute as many (different) tasks sequentially as possible. A CPU can communicate with a GPU by sending data through a memory bus, also called a PCIe express bus. The GPU itself is less optimized for sequential computation. Instead of optimizing throughput, it optimizes parallel task execution by having thousands of cores that can run in parallel, each of them executing (mostly) the same task at a time on different data, also called a workitem. Modern Graphics APIs such as WebGPU provide a level of abstraction to the hierarchical structure of memory access inside the GPU:

Function `matrixMultiplication`(*Matrix W, Matrix X, Matrix Z*)**:**
  $Z \leftarrow 0_{m \times k}$;
  for $i = 1$ *to* m **do**
    for $j = 1$ *to* k **do**
      for $l = 1$ *to* n **do**
        $Z_{i,j} \leftarrow Z_{i,j} + W_{i,l} * X_{l,j}$;
      end
    end
  end

Algorithm 6: Naive Matrix Multiplication executed on the CPU

A **Workload** refers to the entire set of work items that must be processed. This workload is divided into workgroups, with each workgroup containing a number of work items that are scheduled to run concurrently. In WebGPU, the workload is represented as a 3D grid, where each individual "cube" represents a single work item. These work items are then grouped into larger cuboids to form a **Workgroup**. While the cores of a GPU are not actually arranged in three dimensions, this 3D grid representation makes it easier to write parallel code and improves locality for faster access to cache memory. When dispatching a compute pass from the CPU side in WebGPU, the 'workgroupCount' parameter is used to specify the number of workgroups to create along each dimension. This means that the total number of times the compute shader will be invoked is equal to $workgroupCount.x * workgroupCount.y * workgroupCount.z * workgroupSize.x * workgroupSize.y * workgroupSize.z$, where 'workgroupSize' represents the size of each dimension of a workgroup. If the specific GPU hardware being used is unknown, a common Workgroup size is $8 * 8 * 1 = 64$.

3.3.3 Parallel Matrix Multiplication

Consider the following task: Given a matrix $W \in \mathbb{R}^{m \times n}$ and a matrix $X \in \mathbb{R}^{n \times k}$, we aim to compute $Z = WX$. Algorithm 5 shows a naive way of how to implement a matrix-matrix multiplication. The time complexity of this matrix multiplication algorithm is $\mathcal{O}(m \times n \times k)$ due to the algorithms three nested loops. For large matrices as they occur in large Neural Networks this algorithm is infeasible.

Fortunately, the multiplication of two matrices is an 'embarrassingly parallel' process, meaning that the computation of each element of the resulting matrix can be performed independently of the others, which is also true for the addition of matrices and element wise functions [13] . This allows for easy parallelization of the algorithm on multiple processors or cores, as the workload can be easily divided among them with little to no communication overhead.

```
Function shader(Array W, Array X, Array Z, int m, int n, int k):
    if (gId_x >= m and gId_y >= k) or gId_z > 1) then
        return
    end
    row ← gId_y;
    col ← gId_x;
    sum ← 0;
    for i ← 0 to n do
        sum ← sum + W[row * n + i] * X[i * k + col];
    end
    Z[row * k + col] ← sum;
```

Algorithm 7: Parallel Matrix Multiplication Shader executed on the GPU

We now simulate a simplified workflow for executing this task in parallel on the GPU. As an important side note, all specifications of how to use the GPU generally happen on the CPU. First we specify the size of a Workgroup. As discussed earlier, a GPU API such as WebGPU lets us define the size of a Workgroup in three dimensions x, y and z. Since we deal with two dimensional objects (matrices), we specify x and y appropriately and let z be equal to One. Not considering a GPU's specific architecture, we might for example chose x and y to 8 and 8, yielding a set of Workgroups of 64 cores/workItems respectively. We also need to specify the workgroupCount. In the case of the matrix multiplication at most $m * k$ workitems are needed to compute each entry of the resulting matrix. Hence we choose $workgroupCount.x = \lceil \frac{m}{8} \rceil$, $workgroupCount.y = \lceil \frac{k}{8} \rceil$ and $workgroupCount.z = 1$.
We write a compute shader with respect to a global Id / global coordinate (gId_x, gId_y, gId_z), the Matrices W,X and Z are flattened to one dimensional buffers, hence we need to provide the original dimensions to the shader as well. the pseudo-code for the parallel matrix multiplication can be see in Algorithm 6. Due to our ceiling operation when specifying the workgroupCount, we might have too many workitems available on the GPU, hence for the respective workitem coordinates, we return from the shader immediately.

Assuming that we have an unlimited amount of cores inside a GPU, this algorithm reduces the computation of even large scale matrix multiplication to linear time complexity, i.e. $\mathcal{O}(n)$. Better algorithms for this have been invented, for example a recursive block wise multiplication of the matrices. In a future version of the Framework, we might also implement these improvements.
Figure 8 shows a more detailed comparison between the speed of multiplying two squared matrices of increasing size in both C++/Wasm and WGSL/WebGPU, showing that the gained speed ups through parallelization are tremendous, even on low powered

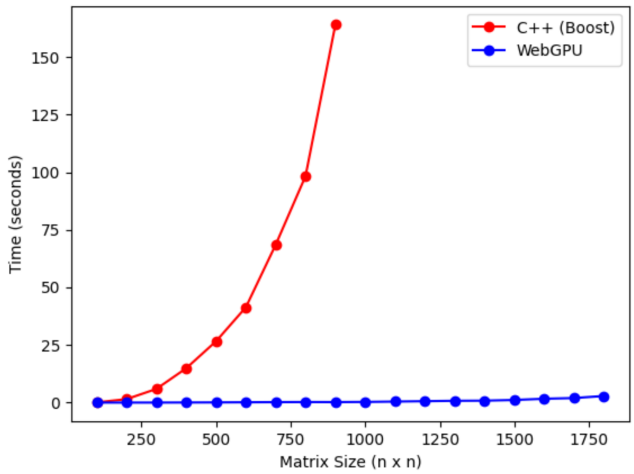

Figure 8: A performance comparison between the C++/Wasm and the WGSL/WebGPU backend. We compute the matrix multiplication between two square $(n \times n)$ matrices. In C++/Wasm the Boost library (compiled with the highest optimization Flag) is used, which uses a well optimized but sequential algorithm. As expected the Compute time grows cubic. The blue curve shows the almost linear time it needs to to the same Job on the GPU with Algorithm 6. In both cases we only measure the pure compute time and neglect other memory transfers. The result clearly shows the advantage of using WebGPU over C++/Wasm, considering the underlying hardware: Intel(R) Core(TM) i5-8250U CPU and its integrated Intel UHD Graphics Card.

machines.

Without providing an equally detailed description it should be noted that other computations such as the addition of matrices and any complex element wise function can also be paralellized in a similar fashion.

3.3.4 Workload Balancing between the CPU and GPU

Another important aspect when using the GPU is the representation of the Computation Graph such that the GPU can deal with it. We cannot simply send a matrix or a complex object such as one of our nodes to the GPU. The programming language of WebGPU is called WGSL (WebGPU Shading Language), which has a Rust like syntax, but does not support object oriented programming. The only datatype that the GPU 'understands' are one dimensional Buffers of arbitrary length. Hence we need to flatten out all the Data of the Computation Graph.

As mentioned earlier, the GPU and the CPU connect through a bus. This data transfer is usually the main bottleneck when doing GPU programming. In order to reduce the data transfer, we aim to send and request data from the CPU side to the GPU side as

little as possible.

Inside the **CPU** we do the following:

- Setup the Computation Graph with all nodes and data and optimize it by finding parts that can be executed at the same time.
- Then send the flattened Graph Data to the GPU memory once.
- Control the execution on the GPU, which includes specifying the the current node which shall be executed. All control parameters are written into a small Buffer that is then sent to the GPU. This is a bottleneck but it is a small one, compared to sending whole subsets of Graph's data in each iteration.
- Request access to the computed data which is stored inside the GPU's memory from time to time, such as the current Loss or the computed data of a specific node. This is a rather big bottleneck, but it is rarely done. The CPU only request full access to the Computation Graph on the GPU once the Learning has finished.

Inside the **GPU** we do the following:

- Receive one or several Buffers that hold all the data of the Computation Graph.
- Receive the shaders and compile them.
- Repeatedly receive the control buffer which is then being interpreted in order to execute the proper computations/shaders on the Graph in parallel.
- Stop writing to or reading from buffers as long as the CPU is actively sending or requesting data.

This workload balancing is a design choice that occurred due to the still limited capabilities of WebGPU. The initial goal was to have the full Computation Graph on the GPU with little to no interaction towards the CPU. Theoretically, this would yield a much faster learning Process and is actually possible on modern NVIDIA GPUs via CUDA [8]. In WebGPU this would require the ability to synchronize executions globally on the GPU, in order to well define a computation trace and to disable race conditions. Unfortunately, a global Synchronization via a global 'Barriers' is not part of the WebGPU specification. At the moment only Barriers in a single Workgroup are possible, but this is not enough to synchronize all needed computations on the Computation Graph. Hence we take the detour of interacting with the CPU for data management. With a future specification of WebGPU this might be improved.

Overall this leaves us with a programmatic interface to define and manipulate the Computation Graph in JavaScript similar to the one we have seen in C++/WebAssembly in section 3.2. This includes defining the training data (through a Data Object), the architecture of the network through a Model object that stores all nodes (tensors) of the network and the actual training process including the number of iterations, the learning rate and an optimization algorithm etc.

3.4 Building the Visual Interface in React.js

3.4.1 Choosing between the Wasm and WebGPU implementation

We have seen how to theoretically describe a Neural Network (Chapter 2) and have visited some of the key implementation details. We are now left with two implementations of a Deep Learning Framework, one utilizing the CPU via C++ and speed ups through optimized Linear Algebra Libraries and good low level memory management, the other utilizing the GPU to speed up computation by parallelization. Both have a similar programmatic interface in JavaScript.
For this project we aim to provide a visual interface on top of the programmatic part, called CoViz, and we chose to permanently use our WebGPU implementation, due to its vastly better performance (compare Figure 8). If there is no dedicated Graphics card available, WebGPU utilizes the integrated Graphics card of the CPU. In our tests this still performed better than the non-parallel C++ implementation.
CoViz in its current implementation should be seen as a proof-of-concept.

3.4.2 UI Design

For the visual interface to our Framework we make use of Next.js/React.js, a popular JavaScript Framework for building interactive UIs. Compared to regular JavaScript, React.js enables software developers to write reusable code through components. A component is a JavaScript function with a HTML return type. In order to visualize the Computation Graph we use ReactFlow, a library for building node-based applications [23] . ReactFlow also defines a graph of nodes and edges which we will later need to translate into our Neural Network Computation Graph (i.e. Model object). Each node in the ReactFlow Graph is a node of our Computation Graph of the Neural Network. This is true until we introduce more levels of abstraction such as the concept of a 'Dense' layer, which includes the matrix multiplication and the addition with a bias matrix.
A node can been created with 'Ctrl + Left Click', a node can be dragged on the canvas by clicking on the colored rectangle on top, two nodes can be connected through an edge by 'Left click' on the outgoing handle of the the one node and a following Drag motion towards one of the incoming handles of the other node, which creates a data flow between both nodes from the left to the right. Nodes and Edges can be

permanently deleted by holding 'ALT + Left Click' while hovering the mouse of them respectively. All other manipulation of the nodes attributes is implemented via Components of Googles 'MUI' library, being a standard in modern Web Design [19] .
Each node displays different changeable attributes, depending on its type. Besides the operations that represent exactly one node in the final Computation Graph such as a node for addition of multiplication etc., we also began to add types that are of higher abstraction such as a 'Dense' layer with or without a bias since this is a frequently used component in a Neural Network and is fairly tedious to build by hand every time. We examine all the different types in Figure 9,10,11 and 12.
The Application is currently deployed via Vercel [26].

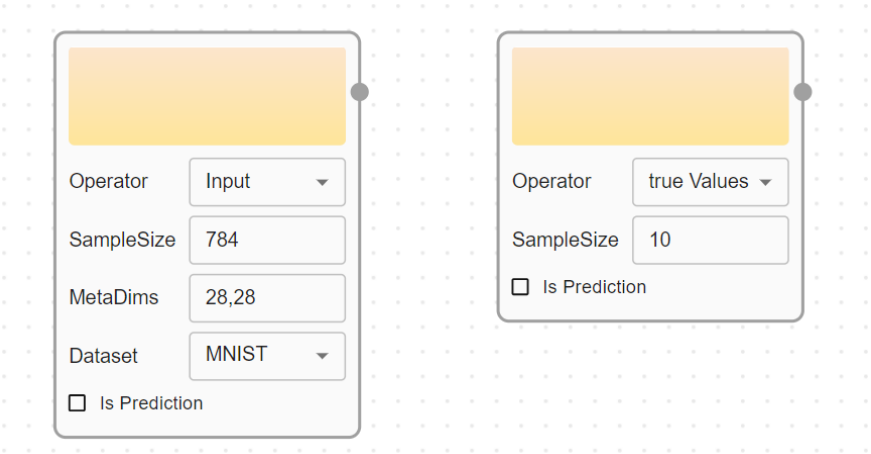

Figure 9: Single Node Types in CoViz: The Input Node (left) that provides mini-batches of randomly selected data samples to subsequent nodes in the Computation Graph. The 'true Values' node (right) provides the corresponding ground truth values, e.g. here the input provides a flattened image of $28 * 28 = 784$ pixels, the true values provide a One-Hot-encoding of the corresponding labels. In other cases such as a regression problems, the 'true Values' node provide normalized real values.

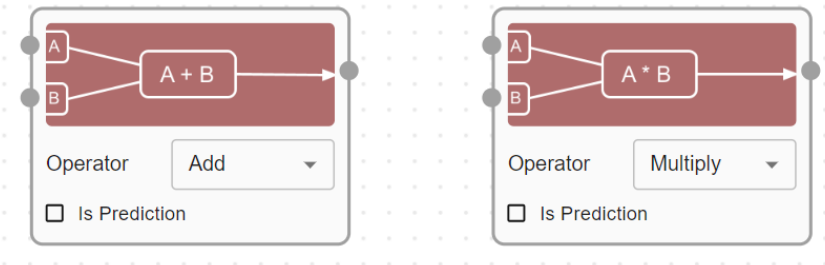

Figure 10: Single Node Types in CoViz: On the left, the 'Add' node adds the entries of the data matrix of its parent nodes if the dimensions of both parent nodes are the same. In this case the order of the incoming nodes does not matter. On the right, the node multiplies the corresponding data matrices of its parent nodes, assuming that the number of columns of the left node's data matrix is equal to the number of rows of the right node's data matrix. In this case the order of the incoming nodes matter.

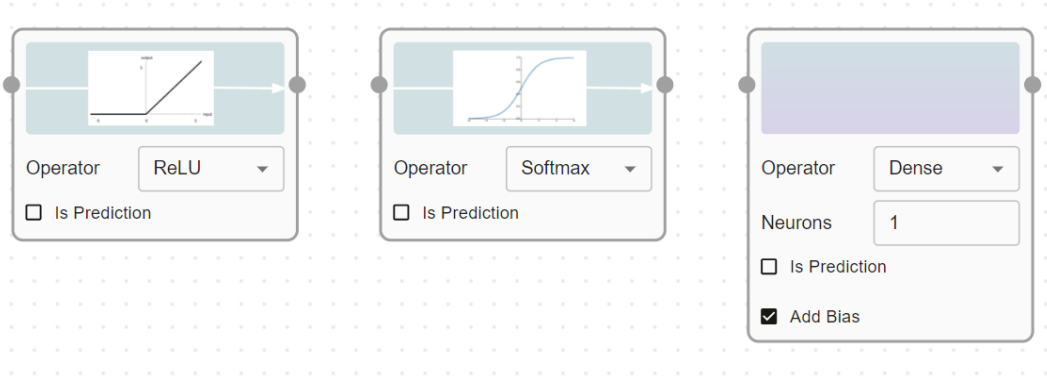

Figure 11: Single Node Types in CoViz: 'ReLU' and 'Softmax' are node types that perform element-wise operations on the incoming data. 'Dense' describes layer of neurons, the user has the choice to add a bias. The Dense layer is a layer of abstraction that needs to be reversed before the actual Computation Graph is defined and sent to the GPU.

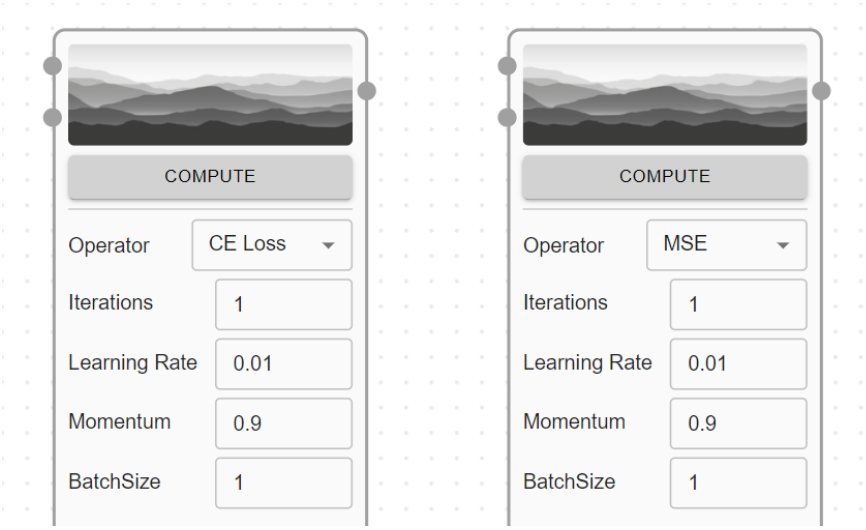

Figure 12: Single Node Types in CoViz: The two available Loss functions i.e. Cross Entropy Loss (aka: 'CE Loss') and Mean Squared Error Loss (aka. 'MSE Loss'). Here the user has the choice to define the number of forward/backward passes, the learning rate, the momentum and the batchSize which describes the number of samples per forward/backward pass. Both node type scan display the error decay over time in a plot (not shown here).

3.4.3 Building Neural Networks with the Visual Interface CoViz

We have presented the different types of nodes (tensors) to create a computation visually. At the same time of designing the architecture of the Neural Network, the user is able to visualize the predicted data and the loss via the D3.js library [9] . The ability to visualize data is currently limited to 2D plots. In a future version of this project we aim to provide visualization of more interesting data types as well, such as Images, Graph Data, 3D plots etc. So far the networks only show the training procedure and can't do just inference on some trained models. This means we can train a model but currently

do not provide a storage for the the model parameters for later usage.

In Figure 13 we present a simple Neural Network that performs regression on the Sine function. The input dataset consists of changing batches of random real valued samples x in the interval from 0 to 1. The true values are provided by computing $sin(10x)$ for each input sample x. The Loss is computed via mean squared error loss. We can activate the visualization of the node that holds the predicted data, by marking the checkbox 'Is Prediction' as true. For these types of regression problems we can add a bias to a 'Dense' layer of neurons for better convergence. Building the Neural Network by adding and connecting nodes can be done in less than a minute.

In addition to simple regression problems, CoViz can also handle binary classification problems on two-dimensional input data. To do so, the model architecture from Figure 13 needs to be slightly altered by changing the loss function from Mean Squared Error (MSE) to Cross-Entropy (CE) and passing the output of the last Dense Layer through a softmax function. This yields a measure for the most probable label. The visualization type can also be changed from 'Regression' to 'Classification', which displays the classification for the two available labels in two different colors. The ground truth for respective areas is shown using a Voronoi Diagram. Figure 14 shows the full network during training.

Figure 13: Training a Neural Network on randomly selected values of the sine function $sin(10x)$ with $x \in [0,1]$. In the output node the true Values (the actual sine function) is plotted in blue and the predicted values are plotted in orange. In the node of type 'MSE' we can also visualize the respective data being computed which shows how the error decreases over time.

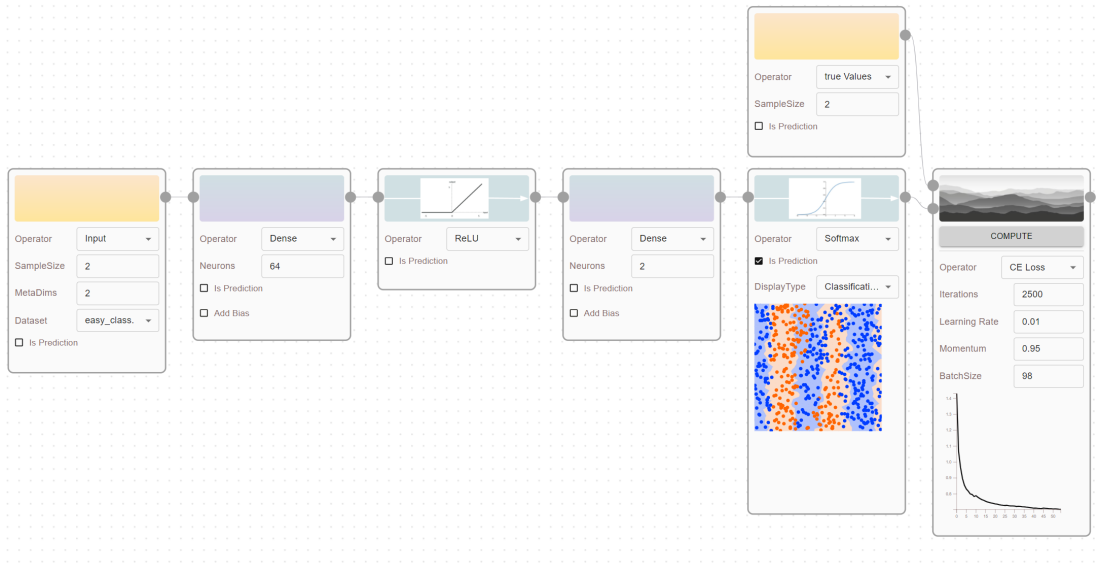

Figure 14: Architecture of a Neural Network for a binary classification of tuples $(x,y) \in [0,1]^2$. The ground truth for a tuple (x,y), which is displayed in the background as a Voronoi diagram, is orange if $cos(15 * x) > 0$, blue otherwise. The colors of a dot at coordinate (x,y) represent the predicted labels.

4 Discussion

4.1 Challenges and Solutions

The incentive of building DiCo was not to provide a perfect deep learning framework like PyTorch or TensorFlow. This could have been done by copying the publicly available source code from GitHub. It is worth mentioning that these well established Frameworks are only the tip of an iceberg of many layers of technologies that have emerged over the last decade epitomized by the rapid development of CUDA/OpenCL [7] [21] for GPGPU programming. By building DiCo we aimed to get a glimpse into most of these layers, starting at counting bits for the GPU buffers up to providing a very high level visual interface. Our goal was to experiment with the concepts discussed in Chapter 2 and come up with ways to integrate those in a new environment. Overall this resulted in a less optimized framework but also allowed us to be creative and learn a lot.
Our initial implementation of the ideas discussed in Chapter 2 was done using WebAssembly. However, after experimenting with WebGPU, we found that it performed much better. As a result, we paused development on the C++/WebAssembly implementation and focused on implementing new features using WebGPU.

One of the main challenges faced while building DiCo with WebAssembly and WebGPU was the scarcity of learning materials. They are still not widely used and since WebGPU is not even supported on all modern web browsers yet, it was challenging to find resources and documentation on how to use them effectively. The documentation that was available was sometimes incomplete or outdated due to the rapidly changing nature of new technologies. Integrating both WebAssembly and WebGPU into a vanilla JavaScript environment is well-documented, but making it work in React.js/Next.js required a lot of experimentation. The lack of external libraries that would facilitate the use of GPGPU programming with WebGPU meant that we had to work with raw buffer manipulation using compute shaders. To overcome these challenges, a significant amount of self-learning and time consuming low level programming was required. Many developers prefer to stick to sophisticated JavaScript libraries, which unfortunately holds back further development and causes them to lag behind in features compared to non-web-based technologies.
The key reason for using Web-Assembly is that it opens the door for building more than just pretty interfaces in client-side applications. It enables developers with a strong background in other languages like C++ to build for the Web and hence make their ideas more accessible to others. The reason for using WebGPU was, though not being very mature, its capability to be completely device independent. Bringing Compute

Shaders to Web Applications enables users to have access to complex scientific computing frameworks by the click of a button. This is amazing progress and building DiCo without these advancements would not have been possible 5 years ago.

4.2 Unique Features

While tools like TensorBoard offer additional visualization for models defined in TensorFlow and PyTorch, our approach with CoViz places a strong emphasis on visualization. In fact, to the best of our knowledge, there is no other free visual deep learning framework that allows users to define a full neural network through a node editor.
Machine learning in the browser is a relatively new field and our framework is among the first to fully leverage the capabilities of WebGPU to build a complete application. By using compute shaders in WebGPU, we have created a performant framework that is also user-friendly for those without programming experience. As mentioned earlier, one major advantage of our implementation with WebGPU is its platform independence. This means that a model defined in the browser on a desktop PC with a dedicated GPU can also run on the browser of a smartphone with only an integrated graphics card.

4.3 Limitations and Future Work

The current framework, i.e. DiCo-GPU with CoViz, can handle basic operations such as addition, multiplication, several activation functions, and a few loss functions. This is enough to build a basic neural network for simple classification or regression problems. Theoretically, if the input data and labeled data are provided in the appropriate format, the framework can deal with any type of independent and identically distributed data. The framework currently does not support convolutional filters or manipulation of spatial or sequential data, it does not support layer and batch normalization and many other features that are relevant to build more complex models. There are limited types of data that can be visualized inside a node of the Computation Graph inside CoViz, mainly 2D data such as plots for 2D regression problems.
Furthermore it should be mentioned that Deep learning on a single device is limiting, this is true for using the Browser as the environment in order to build and train a Neural Network client-side, but this holds true even when using other deep learning frameworks even if a dedicated GPU is available. In order to increase training performance, it might be possible to do networking over the browser between different devices to enable distributed learning. Our framework is less optimized than TensorFlow or PyTorch since we have not put enough effort in optimizing the computation graph before and during execution and our implementation of the parallel matrix mul-

tiplication and other operations is rather naive.

To address these shortcomings of our Framework, a future report needs to do an explicit performance comparison between our implementation and more sophisticated ones. Finally, we plan to add more optimization algorithms and regularization such as Dropout Layers and a Layer Norm. Then DiCo would have enough features to build more complex models like a Transformer. A convolutional filter is in active development at the moment, which will enable the creation of CNNs. Moreover, it might be convenient to have additional predefined blocks of the Network similar to the discussed 'Dense' layer in order to facilitate the building of complex network models and to enable more advanced experimentation. The visualizations might be easily expanded in the future, by utilizing other data visualization tools of the D3.js library.

5 Conclusion

In this thesis, we explored the fundamental concepts underlying modern deep learning frameworks, with a focus on the computational graph paradigm for constructing and training neural networks. We provided the theoretical foundation necessary to understand computation graphs and the use of automatic differentiation for gradient computation. We also described the design and development of DiCo-Wasm and DiCo-GPU, which form the core of our differentiable programming engine. To demonstrate our engine's capabilities, we applied it to simple regression and classification tasks. Additionally, we presented CoViz as a proof-of-concept visual interface that allows non-programmers to experiment with neural networks.

It is important to emphasize that this project is an ongoing effort with much work still to be done. Our goal was not to create a perfect deep learning framework, but rather to showcase the potential of WebGPU and WebAssembly while demystifying established frameworks such as TensorFlow and PyTorch.

Overall, we believe that CoViz provides an engaging and educational experience for its users by helping them gain a better intuition for how neural networks are built. The data visualization capabilities of CoViz offer valuable insights into the effects of different neural network sizes and architectures.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, and Z. Chen. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016.

[2] M. Arora. The architecture and evolution of cpu-gpu systems for general purpose computing. In *Name of Conference Proceedings*, 2012.

[3] A. G. Baydin, B. A. Pearlmutter, and A. A. Radul. Automatic differentiation in machine learning: a survey. *CoRR*, abs/1502.05767, 2015.

[4] Boost. `https://www.boost.org/`. Accessed: March 1st, 2023.

[5] R. L. Burden and J. D. Faires. *Numerical Analysis*. Brooks/Cole, 2001.

[6] Computing neural network gradients. `https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/readings/gradient-notes.pdf`. Accessed: March 1st, 2023.

[7] Cuda. `https://docs.nvidia.com/cuda/`. Accessed: April 1st, 2023.

[8] Getting started with cuda graphs. `https://developer.nvidia.com/blog/cuda-graphs/`. Accessed: April 1st, 2023.

[9] D3. `https://d3js.org/`. Accessed: March 1st, 2023.

[10] Emscripten. `https://emscripten.org/`. Accessed: March 1st, 2023.

[11] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[12] J. Grabmeier and E. Kaltofen. *Computer Algebra Handbook: Foundations, Applications, Systems*. Springer, 2003.

[13] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2003.

[14] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[15] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[16] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

[17] Y. LeCun, C. Cortes, and C. J. Burges. Mnist handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010.

[18] J. Lederer. Activation functions in artificial neural networks: A systematic overview, 2021.

[19] Mui. `https://mui.com/`. Accessed: March 1st, 2023.

[20] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Science & Business Media, 2006.

[21] Opencl. `https://www.khronos.org/opencl/`. Accessed: April 1st, 2023.

[22] A. Paszke, S. Gross, F. Massa, and A. Lerer. Pytorch: An imperative style, high-performance deep learning library, 2019.

[23] React flow. `https://reactflow.dev/`. Accessed: March 1st, 2023.

[24] React. `https://reactjs.org/`. Accessed: March 1st, 2023.

[25] H. Robbins and S. Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951.

[26] Vercel. `https://vercel.com/`. Accessed: April 1st, 2023.

[27] Wasm. `https://webassembly.org/`. Accessed: March 1st, 2023.

[28] Webgpu. `https://www.w3.org/TR/webgpu/`. Accessed: March 1st, 2023.

Declaration of Academic Honesty

I hereby declare under oath that I have completed this work independently and without the use of any resources other than those specified. All passages taken verbatim or in substance from published or unpublished writings are marked as such. This work has not been submitted in the same or similar form or in part for any other examination. I certify that the electronic version submitted is identical to the print version.
I am aware of the criminal liability of a false affidavit, namely the threat of punishment under § 156 StGB of up to three years imprisonment or a fine for intentional commission of the offense or under § 161 (1) StGB of up to one year imprisonment or a fine for negligent commission.

Heidelberg 23. 06. 2023

Tillmann Fehrenbach

Declaration of Consent for Plagiarism Checks

Name: Fehrenbach
First Name: Tillmann
Matriculation Number: 4012922
Type of Work: Bachelor Thesis

I hereby agree that my thesis submitted to the Mathematics and Computer Science Department at the University of Heidelberg may be subjected to automated plagiarism checks using www.turnitin.com or similar plagiarism checking tools.
The review of the work will be carried out exclusively by staff of the department and will only take place anonymously and without permanent storage in the database of the plagiarism checking tool.
I have been advised that a result of the plagiarism check that indicates the use of unacknowledged foreign sources constitutes an attempt at deception. In this case, the work will be graded as a failed examination performance. Further measures of an examination or criminal law nature may be initiated in consultation with the examination office.

Heidelberg 23. 06. 2023

Tillmann Fehrenbach