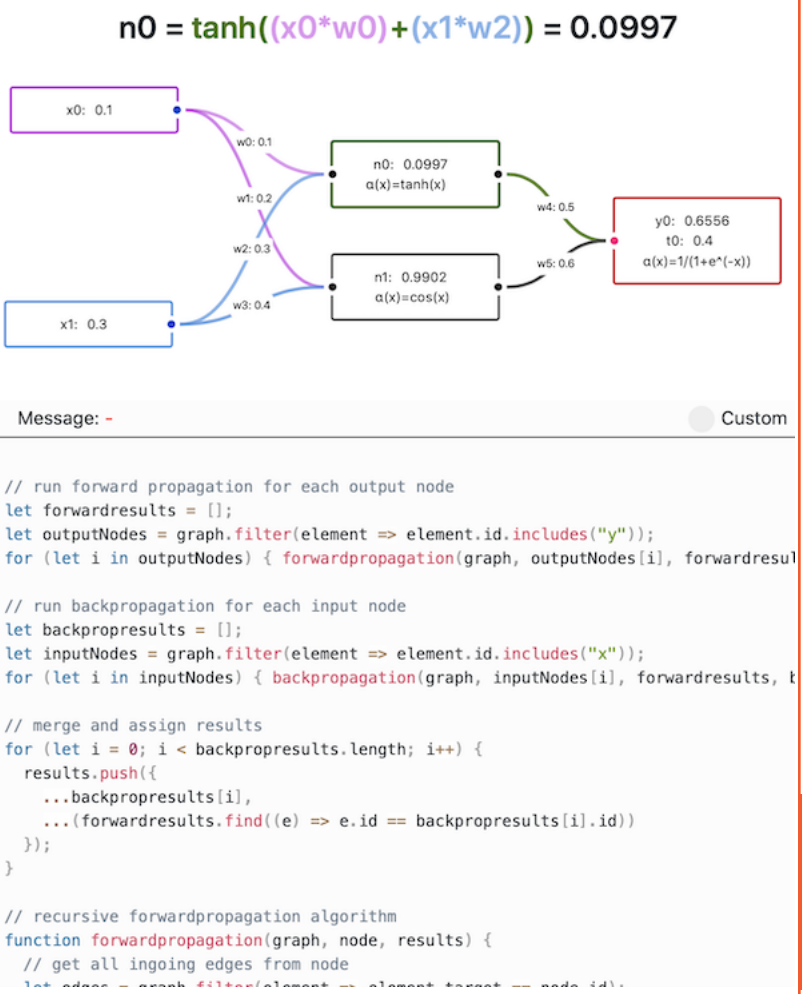
DeepEq - An educational platform to help students develop a mathematical intuition for deep learning



Master's Thesis

People and Computing Lab
Department of Informatics
University of Zurich

by
Peter Giger 14-915-383

University of Zurich[UZH]

PAC

Supervised by
Prof. Dr. Chat Wacharamanotham
Prof. Dr. Dominik Petko

Submission: 21 January 2021

Contents

List of Figures

List of Tables

Abstract

Understanding the flow of gradients in deep learning is essential. Without a gradient, there is no learning in an artificial neural network. Platforms such as Cognimates and Google's TensorFlow Playground lower the barriers to machine learning and encourage users to tinker with different parameters. However, these platforms often hide the inner workings and equations of the algorithms. Backpropagation and gradient descent, the workhorses behind deep learning, remain hidden from the students. For this reason, I have created DeepEq, an educational platform to help students develop a mathematical intuition for deep learning. DeepEq allows users to create a small neural network by joining perceptrons, automatically provides the underlying equations, and lets users implement their own backpropagation algorithm. A complementary interactive tutorial based on the 4C/ID model serves as a guide throughout the learning process.

Acknowledgements

I would like to thank Prof. Chat Wacharamanotham for his excellent supervision, insightful feedback, and guidance throughout my master's studies. Moreover, I would like to thank Prof. Dominik Petko for his valuable inputs from the educational point of view. Last but not least, a special thanks to my friends and family for their support.

Chapter 1

Introduction

Understanding the flow of gradients in deep learning is essential. Without a gradient, there is no learning in an artificial neural network (ANN). For instance, choosing the wrong weights or activation functions can lead to a vanishing gradient (from [Hochreiter, 1998]) and break the learning. In real-world deep learning projects, the role of the gradient is often forgotten because frameworks such as TensorFlow (from [Abadi et al., 2015]) successfully hide the learning algorithms. This can be a double-edged sword: It simplifies the process of building machine learning models, but it might lead users to think that the learning "just works". The problem arises when the network does not learn as expected. Without a clear understanding of the underlying algorithms, debugging becomes a matter of trial and error. In order to avoid this, it is important to have an in-depth understanding of the main algorithms (backpropagation from [Rumelhart et al., 1986], gradient descent from [Ruder, 2016]) and their implications. Platforms such as Google's TensorFlow Playground (from [Smilkov et al., 2017]) provide interactive visualizations for a better intuition about ANNs but do not cover details about backpropagation nor gradient descent. For this reason, I present "DeepEq", an educational platform to help students develop a mathematical intuition for deep learning.

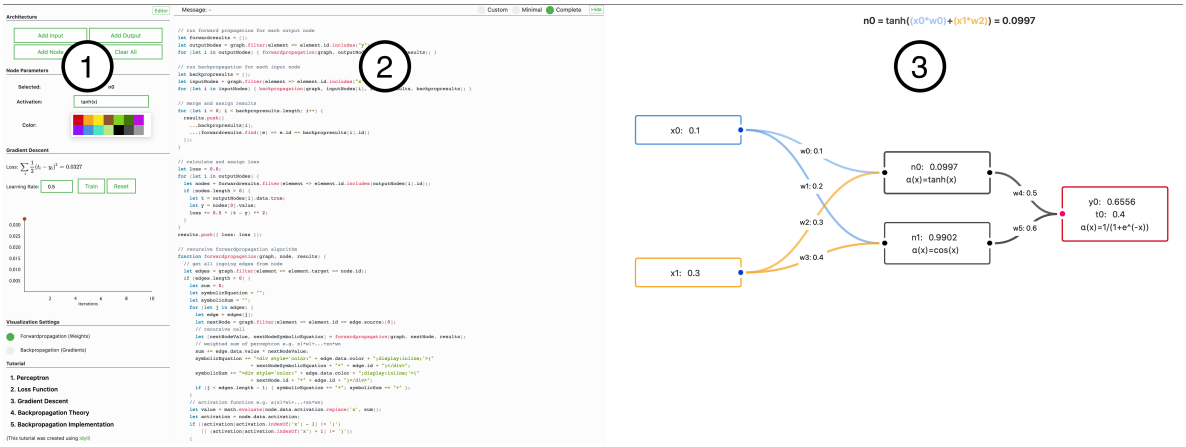


Figure 1.1: DeepEq User Interface - ① Define a neural network architecture and set parameters such as the input value or activation function ② Integrated code editor for implementing forward- and backpropagation in JavsScript ③ Flow-based visualization for connecting nodes (weights) and inspecting the colorized math equations.

From an educational point of view, only little is known about how to teach artificial intelligence (AI) effectively (from [Ko, 2017]). Visualizations such as from Zeiler and Fergus [2014] are readily used in AI courses to explain the inner workings of ANNs. But, however good visualizations may be, truly understanding (not just using) backpropagation requires some mathematics. For this reason, the primary goal of DeepEq is to create a bridge between deep learning theory and practice and allow users to reason about the mathematical implications of backpropagation in an interactive way. DeepEq is intended to facilitate traditional "pen and paper thought experiments" and tries to support users in building a mental model more easily. All equations are kept as simple as possible to make them accessible to high school level students and above. The contributions of DeepEq (see figure 1.1) are as follows:

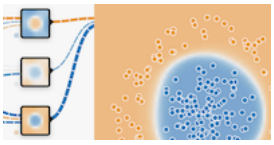
- It allows users to create a small artificial neural network "by hand" and reveals the underlying equations for forward- and backpropagation
- It encourages users to tinker with and implement their own backpropagation algorithm in JavaScript
- A complementary interactive tutorial based on the 4C/ID model serves as a guide throughout the learning process

The following chapters are structured as follows: Chapter 2 goes over the related work and their limitations. Chapter 3 introduces the main algorithms used in deep learning and serves as a basis for the interactive tutorial. Chapter 4 contains the 4C/ID blueprints of the tutorial as well as other implementation details. Moreover, two usage examples/walkthroughs of DeepEq are provided. Chapter 5 contains details about the evaluation method and the results of the think-aloud sessions. Last but not least, Chapter 6 provides a summary of the work and discusses possible future work.

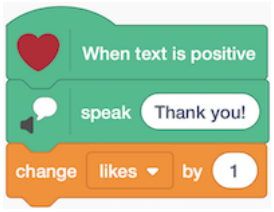
Chapter 2

Related Work

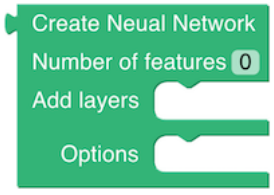
To teach and tinker with AI, several tools and products exist. Targeting the general population, Google [2017] created "Teachable Machine", a hands-on experiment for training and detecting arbitrary objects shown to the webcam. Due to the low entry barriers, it is suitable for everyone and provides a surface level understanding of machine learning. Moreover, Smilkov et al. [2017] (working at Google) released a neural network playground called TensorFlow Playground. It is an interactive tool for visualizing artificial neural networks and requires prior deep learning knowledge. For the educational sector, tools tend to build on block-based visual programming languages such as Scratch (from [Resnick et al., 2009]). Block-based languages improve learnability and make programming more accessible, especially to younger students (from [Bau et al., 2017]). Based on their success, Lane [2017] added blocks for interacting with AI models. His platform is called "Machine Learning for Kids" and was first introduced in 2017. One year later, Druga [2018] released Cognimates, a platform for training AI models in Scratch. Both, ML4Kids and Cognimates, allow younger students to train ML models by connecting visual blocks together. Using a similar approach but with high school students in mind, Zhu [2019] developed several machine learning extensions for the MIT App Inventor (from [Pokress and Veiga, 2013]), a block-based platform for building apps. Zhu covered more advanced topics ranging from regressions to convolutional neural networks. Targeting university students, Rao et al. [2018] introduced Milo, a block-based visual programming environment for data science education. Milo offers a complete data science workflow based on blocks. However, what all these tools have in common is their in-



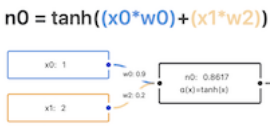
TF Playground - Feature visualization



Cognimates - Feeling recognition



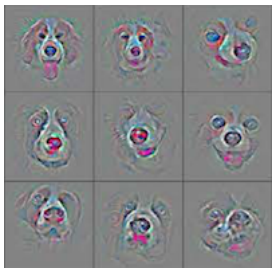
Milo - Neural network block



DeepEq - Equation

transparency when it comes to the underlying algorithms. For example, artificial neural networks are merely treated as black boxes (Druga [2018], Google [2017], Lane [2017], Rao et al. [2018], Zhu [2019]) with some tools providing a visual glimpse into the black box (Smilkov et al. [2017]). Backpropagation and gradient descent, the workhorses behind deep learning, remain hidden from the students. DeepEq tries to open the black box by directly merging deep learning theory into the tool.

From an educational point of view, effectively teaching AI concepts is an emerging challenge with little existing work (from [Sulmont et al., 2019]). As shown by Marques et al. [2020], there is no shortage of AI courses. They have found and analyzed 30 Instructional Units (IUs) on basic ML concepts and criticized the lack of rigorous scientific development and evaluation of the IUs. Based on a workshop at ICER 2017, Ko [2017] published a blog post underlining the lack of knowledge about how to teach ML effectively. She proposed the use of pedagogical content knowledge (PCK) for teaching machine learning. Two years later, Sulmont et al. [2019] presented the first paper exploring PCK in the context of non-computer science majors. They have discovered insights about preconceptions, barriers, and pedagogical tactics for teaching ML. With a focus on K-12 students, Touretzky et al. [2019] highlighted a lack of guidance for teaching AI and proposed curriculum guidelines. However, despite all the efforts, the field of AI education is still in its infancy. DeepEq contributes by providing an interactive deep learning tool and tutorial based on the 4C/ID model by Van Merriënboer et al. [2002].



Zeiler - CNN Visualization

Understanding AI not only concerns educators but machine learning experts as well. In fact, an entire field is devoted to the eXplainability of AI (XAI). As machine learning algorithms are getting more complex, transparency (opposite of black box) and interpretability are all the more important e.g. understanding the algorithm of an autonomous vehicle (from [Arrieta et al., 2020]). Well-known work such as Zeiler and Fergus [2014] is readily used in AI courses to explain and visualize the inner workings of a large Convolutional Neural Network (CNN). However, even though DeepEq visualizes an ANN and tries to open the AI black box, it does provide only little value to the field of XAI because interpreting small networks was never a problem to begin with. Rather, DeepEq is intended to facilitate traditional "pen and paper thought experiments" and thus, tries to improve the productivity of the learner.

Chapter 3

Deep Learning Theory

This chapter introduces the main concepts and algorithms used in training a feedforward neural network. The interactive tutorial (see chapter 4) is based on a simplified version of this chapter.

3.1 Perceptron

The perceptron is an artificial neuron and was first introduced by Rosenblatt [1958]. In his paper, Rosenblatt [1958] showed that a perceptron can learn from (and respond to) stimuli. The notation has changed since then but the concepts remain and serve as the foundation for today's artificial neural networks. A modern perceptron (e.g. from [Rumelhart et al., 1986]) multiplies an input $x_i \in \mathbb{R}$ with its corresponding parameter $w_i \in \mathbb{R}$. This parameter is called a "weight" because it prioritizes the inputs for all $i \in \{1, ..., n\}$. For example, it could be that x_1 detects animal fur and x_2 detects colors. A snake-detecting perceptron should prioritize the color detector (because snakes have no fur) and, therefore, it should multiply x_2 with a large weight w_2 and x_1 with a small weight w_1. Finally, the results are combined $\sum_i$ and an activation function $a : \mathbb{R} \to \mathbb{R}$ is applied. Figure 3.1 illustrates the workings of a perceptron.

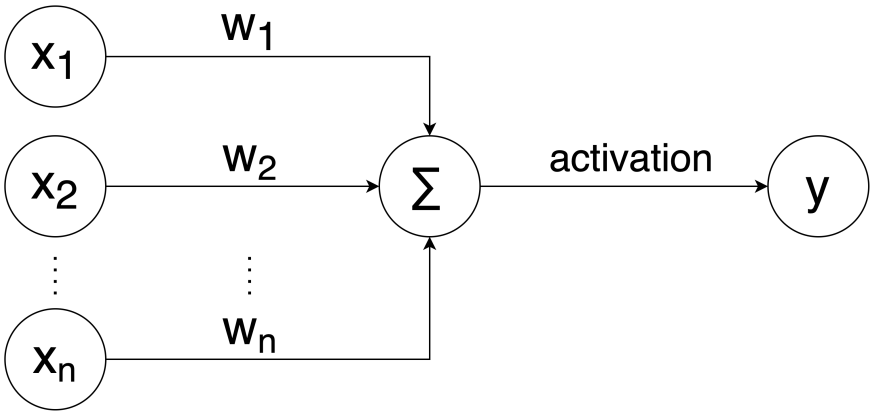


Figure 3.1: Perceptron

The universal approximation theorem states that a **non-linear** artificial neural network can approximate any function

The purpose of the activation function is to add non-linearity, and thus, make complex behavior possible in the first place. This is because chaining linear perceptrons again results in a linear perceptron, therefore, does not provide a better prediction. The proof follows from the definition of a linear function. A function $f : \mathbb{R} \to \mathbb{R}$ is linear if the following properties hold:

$$f(cu) = cf(u) \; \forall u \in \mathbb{R}, c \in \mathbb{R}$$

$$f(u + v) = f(u) + f(v) \; \forall u \in \mathbb{R}, v \in \mathbb{R}$$

A perceptron with output $y \in \mathbb{R}$ can be written as:

$$y = a(\sum_i x_i w_i)$$

Chained perceptrons (single-input in this case) can be represented as a map between input x and output y:

$$y(x) = f_1(...f_n(x)) = (f_1 \circ ... \circ f_n)(x)$$

where each function $f_j : \mathbb{R} \to \mathbb{R}$ is the intermediate output $j \in \{1, ..., n\}$ of input x_j, weight w_j and activation a_j:

$$f_j = a_j(x_j w_j)$$

Thus, without an activation function, chained perceptrons remain linear:

$$\begin{aligned}(f_1 \circ ... \circ f_n)(cu) &= c(\prod_j w)u \\ &= c(f_1 \circ ... \circ f_n)(u)\end{aligned}$$

$$\begin{aligned}(f_1 \circ ... \circ f_n)(u+y) &= (\prod_j w)(u+v) \\ &= (\prod_j w)u + (\prod_j w)v \\ &= (f_1 \circ ... \circ f_n)(u) + (f_1 \circ ... \circ f_n)(v)\end{aligned}$$

□

Because of that, activation functions are almost always used. According to Nwankpa et al. [2018], common activation functions are:

Sigmoid: $a(x) = \sigma(x) = \frac{1}{1+e^{-x}},\ x \in \mathbb{R}$

Tanh: $a(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}},\ x \in \mathbb{R}$

ReLU: $a(x) = max(0, x),\ x \in \mathbb{R}$

Note: Due to the difficult analytical derivative of ReLU and other rectified activation functions, they will not be used in this thesis

3.2 Loss Function

Training a perceptron requires a function $l : \mathbb{R}^n \to \mathbb{R}$ (mostly $\mathbb{R}^n \to \mathbb{R}_{\geq 0}$). This function is called a loss function and imitates "learning by example" e.g. teach a child what a dog is by showing a dog nearby. A commonly used loss function is defined as follows (from [Rumelhart et al., 1986]):

$$loss = \sum_i \frac{1}{2}(t_i - y_i)^2, i \in \{1, ..., n\}$$

where t_i is a true label provided by a human and y_i is the output/prediction of the perceptron. The further apart the true label and the prediction, the higher the loss. There is no particular reason to choose this exact loss function (e.g. $\sum_i |t_i - y_i|$ works as well), but the mathemat-

ical properties make it easier to work with. In particular, $\frac{1}{2}$ simplifies the derivative, and the square creates a lower boundary for the loss.

3.3 Gradient Descent

Gradient descent minimizes the loss function by updating the weights step-by-step in the opposite direction of the gradient (from [Ruder, 2016]). It is one of the most commonly used optimization algorithm in artificial neural networks and responsible for the actual "learning" (from [Ruder, 2016]). The general problem is as follows (from [Ruder, 2016]):

$$\min_{w_1,...,w_n} loss(w_1, ..., w_n)$$

and iteratively update the weights:

$$w_i^{new} = w_i - \eta \nabla_{w_i} loss(w_1, ..., w_n)$$

where $\eta \in \mathbb{R}_{>0}$ is the learning rate (e.g. 0.8), $\nabla_{w_i} loss(w_1, ..., w_n)$ is the gradient of the loss function with respect to the weight w_i, and w_i^{new} is the updated weight.

3.4 Backpropagation Theory

Backpropagation (from [Rumelhart et al., 1986]) is a systematic way to calculate the gradients efficiently, and thus essential for large artificial neural networks. Mathematically, it is the repeated application of the chain rule (from [Rumelhart et al., 1986]):

$$f'(x) = g'(h(x))h'(x)$$

where $f(x) = g(h(x))$, g and h are differentiable with respect to x, and $f'(x)$ is the derivative. It is important to note that multiple notations of differentiation exist. For example, in Leibniz's notation, the chain rule can be expressed as $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$. However, due to the common usage of the Lagrange's notation in high schools, a prime mark f' will be used to indicate a derivative from now on. Moreover, f'_x will be used to express partial derivatives.

To give an example, assume the following network (see figure 3.2):

$$n = sin((x_0 w_0) + (x_1 w_1))$$

$$y = sin(n w_2)$$

The goal is to calculate the derivative with respect to each weight (how much the loss changes when the weight changes). Due to the chain rule, it is reasonable to start at the outermost weight and continue till reaching the innermost weight. The first step is to calculate the derivative with respect to w_2.

$$loss = \frac{1}{2}(t - y)^2$$

$$loss'_y = (t - y)(-y') = (t - y)(-cos(n w_2))$$

$$loss'_{w_2} = loss'_y n$$

Note that $loss'_y$ represents an intermediate result and is, strictly speaking, not formally correct because y is not a differentiable variable. However, it highlights the fact that the derivative of the weight w_2 is based on the derivative of the node y. The second step is to calculate the derivative with respect to w_0 and w_1:

$$loss'_n = loss'_y cos((x0w0) + (x1w1))w_2$$

$$loss'_{w_0} = loss'_n x_0$$

$$loss'_{w_1} = loss'_n x_1$$

Again, (formally) there is no $loss'_n$ but it shows that the derivatives build on each other which is why it is called backpropagation (backward propagation of the errors). The derivative of the weight w_0 is based on the derivative of the node n which is based on the derivative of the weight w_2 which is based on the derivative of the node y.

3.5 Backpropagation Implementation

Calculating derivatives by hand is cumbersome and error-prone. For this reason, it is a good idea to automate this process. The algorithm is usually divided into ”forwardpropagation” and ”backpropagation”. The forwardpropagation starts from the inputs, calculates the value of the perceptron, and repeats this till every perceptron has a value. Backpropagation builds on the results of forwardpropagation and calculates the gradient of each weight.

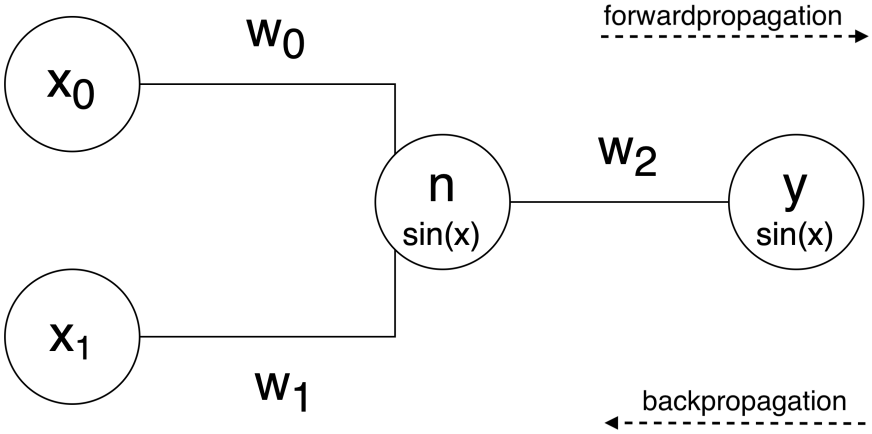


Figure 3.2: Backpropagation Example

More formally, given a graph G with vertices/nodes V and edges/weights E, forwardpropagation calculates the output of every node by multiplying its ingoing edges ins with their corresponding source node s and applies the summation as well as the activation function (analogous to the calculations of a perceptron). The pseudo-code can be found in algorithm 1.

Algorithm 1 Forwardpropagation

Require: Graph $G(V, E)$ representing an artificial neural network
  $Y \leftarrow$ output nodes of V
  for all $y \in Y$ **do**
    FORWARDPROPAGATION(y)
  end for
  function FORWARDPROPAGATION(v)
    $ins \leftarrow$ ingoing edges of v
    if $ins \neq \{\}$ **then**
      $sum \leftarrow 0$
      for all $in \in ins$ **do**
        $s \leftarrow$ source node of in
        $sum \leftarrow sum + in *$ FORWARDPROPAGATION(s)
      end for
      return $activation(sum)$
    else
      return v ▷ exit recursion when reaching input node x
    end if
  end function

Backpropagation builds on the results of forwardpropagation and calculates the gradient of each edge/weight ∇_{out}. It is important to note that the weight gradient ∇_{out} differs from the propagated gradient ∇_{total}. Hence, only ∇_{total} has to be returned (back-propagated) and ∇_{out} has to be saved. The pseudo-code can be found in algorithm 2.

Algorithm 2 Backpropagation

Require: Graph $G(V, E)$ representing an artificial neural network
(includes the values from forwardprop and a true value t)
$X \leftarrow$ input nodes of V
for all $x \in X$ **do**
BACKPROPAGATION(x)
end for
function BACKPROPAGATION(v)
$outs \leftarrow$ outgoing edges of v
if $outs \neq \{\}$ **then**
$\nabla_{total} \leftarrow 0$
for all $out \in outs$ **do**
$s \leftarrow$ sink node of out
$sum \leftarrow$ pre-activation value of node v
$\nabla_{total} \leftarrow \nabla_{total} * activation'(sum) * out *$
BACKPROPAGATION(s)
$\nabla_{out} \leftarrow v *$ BACKPROPAGATION(s) ▷ weight gradient
end for
return ∇_{total}
else
$sum \leftarrow$ pre-activation value of node v
return $-1 * (t - v) * activation'(sum)$
end if
end function

It should be noted that neither algorithm 1 nor algorithm 2 are suitable for large neural networks. This is because of the non-optimized recursions which re-calculate the values multiple times (keyword: dynamic programming). However, due to the educational focus of this project, simplicity and compactness are more important than efficiency.

The concrete JavaScript implementation of algorithm 1 and algorithm 2 can be found in the appendix A

Chapter 4

Design And Implementation

This project can roughly be divided into two parts, an educational tutorial and an interactive component providing the main functionality. The implementation relies on Idyll (from [Conlen and Heer, 2018]), a toolkit for interactive articles and explorable explanations. With Idyll, it was possible to write the interactive tutorial in a clean Markdown-like language while having the freedom to create custom components in ReactJS providing the main functionality.

4.1 Tutorial

The content of the tutorial is based on a simplified version of chapter 3 and can be found in appendix B. The goal was to keep it as simple as possible and make it accessible to high school level students and above. It consists of the following five parts:

1. Perceptron
2. Loss Function
3. Gradient Descent
4. Backpropagation Theory
5. Backpropagation Implementation

The tutorial is based on the 4C/ID model by Van Merriënboer et al. [2002] which is well suited for complex learning (e.g. real-life tasks that require the integration of different skills), and thus is a good fit for the problem-solving skills required in deep learning. The model defines four central components which are then used to structure blueprints for complex learning (from [Van Merriënboer et al., 2002]): **Learning Task** (e.g. find the best answer for a given question), **Supportive Information** (e.g. conceptual model of a search engine, library etc.), **Just-In-Time-Information** (e.g. how to use a search engine, how to access the library etc.), and **Part-Task Practice** (e.g. regular expressions). The blueprints of the tutorials (increasing in complexity) can be found below.

Perceptron

Supportive Information: Mental model and example
- Conceptual model of a perceptron

Learning Task: Conventional
Learners have to create a perceptron and try different input values, weights, and activation functions. They are required to give an interpretation of the equations and the flow of information.

Part-Task Practice: Interactive Elements
Two interactive elements provide hands-on practice on the concept of a perceptron and a pixel.

Just-In-Time-Information: Demonstration
Procedures (GIF) for using the application

Loss Function

Supportive Information: Mental model and example
- Conceptual model of the loss function

Learning Task: Conventional
Learners have to try different true values, input values, weights, and activation functions, and reason about the effect they have on the loss.

Learning Task: Conventional
Learners have to explain the circumstances under which a loss might never approach zero.

Part-Task Practice: Interactive Elements
Two interactive elements provide hands-on practice on the concept of a loss function.

Just-In-Time-Information: Demonstration
Procedures (GIF) for using the application

Gradient Descent

Supportive Information: Mental model and example
- Conceptual model of gradient descent

Learning Task: Conventional
Learners have to try different learning rates and think about how the gradients are used to adjust the weights.

Learning Task: Conventional
Learners have to explain the circumstances under which a gradient can be too small or too large.

Part-Task Practice: Interactive Elements
One interactive element provides hands-on practice on the concept of minimizing a loss function.

Just-In-Time-Information: Demonstration
Procedures (GIF) for using the application

Backpropagation Theory

Supportive Information: Mental model and example
- Conceptual model of backpropagation
- Worked-out example of backpropagation

Learning Task: Conventional
Learners have to try different network topologies and reason about the flow of the gradient.

Just-In-Time-Information: Demonstration
Procedures (GIF) for using the application

Backpropagation Implementation

Supportive Information: Example
- Pseudo-code of the backpropagation algorithm

Learning Task: Conventional
Learners have to implement their own version of backpropagation based on the knowledge of the previous tasks. Working code examples are given in order to facilitate the task.

Just-In-Time-Information: Demonstration
Procedures (GIF) for using the application

The implementation of the tutorial relies on Idyll by Conlen and Heer [2018]. Idyll provides a Markdown-like language in which ReactJS components can be embedded to provide the interactivity. Built-in components (e.g. sliders or text-boxes) reduce the effort and programming skills required. However, for this project, the amount of time and effort required for creating interactive elements was nevertheless high. The reason lies in the variation of visualization problems. For example, built-in components such as text-boxes and circles cannot sufficiently model a perceptron. The solution is simple yet time consuming: 1. Draw a perceptron 2. Import into Idyll 3. Add interactivity. These three steps have been found to be the easiest method for creating non-standard interactive components.

As an example, the process for creating an interactive perceptron in Idyll is as follows: First, the overall structure of the perceptron has to be created (without the interactive elements). The most straight forward way is to use an SVG (Scalable Vector Graphics) editor such as Inkscape. Alternatively, it can be hand-coded since SVG is a human-readable format similar to HTML. Figure 4.1 shows the outcome when using an SVG editor. Second, the SVG format has to be converted (e.g. <line x1=4../> to [line x1:4../]) due to the different syntax of Idyll. This process can largely be automated with regular expressions. Third, the interactive components have to be created. This can be done by linking the Idyll components (e.g. [var name:"x1" value:1/] defines a variable) with the corresponding SVG components (e.g. [SvgText value:x1/]). An example code can be found in figure 4.2. Last but not least, the Idyll compiler has to generate the interactive element. The final interactive component can be seen in figure 4.3.

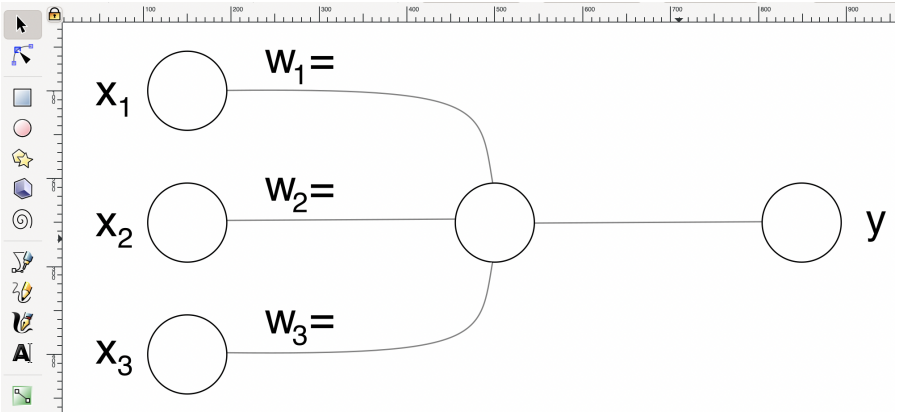


Figure 4.1: Interactive Perceptron Structure

```
1  [var name:"perceptronX1" value:1 /]
2  X1: [customRange className:"slider" style:`{width: '22%'}` value:perceptronX1 min:0 max:3 step:1/]
3  ...
4  [derived name:"perceptronY" value:`Math.tanh(Number(perceptronX1*perceptronW1+...+perceptronW3)).toFixed(1)`/]
5
6  [SVG viewBox:"0 0 1000 500"]
7    [ellipse fill:"#fff" stroke:"#000" strokeWidth:"1.5" cx:"150" cy:"100" rx:"45" ry:"45"/]
8    [SvgText value:perceptronX1 y:"115.76917" x:"138.07674" /]
9    ...
10   [SvgText value:perceptronY y:"265" x:"820" /]
11 [/SVG]
```

Figure 4.2: Interactive Perceptron Idyll Code

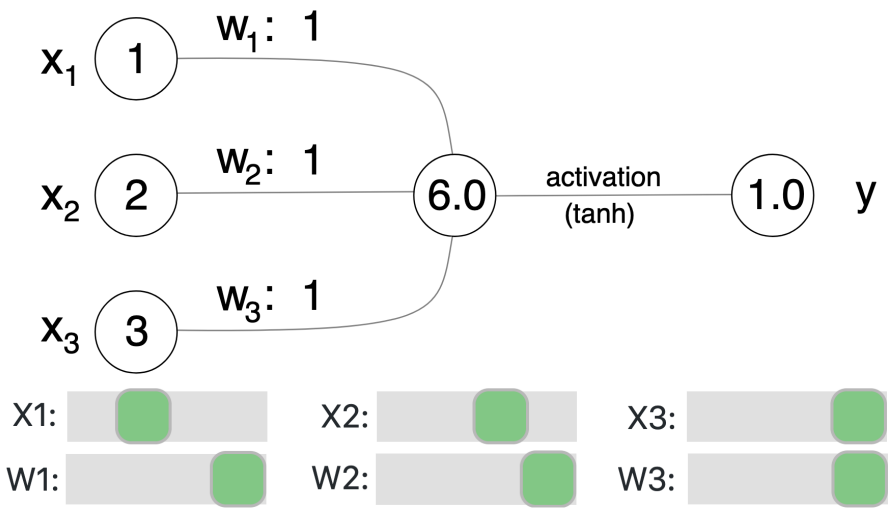


Figure 4.3: Interactive Perceptron With Sliders

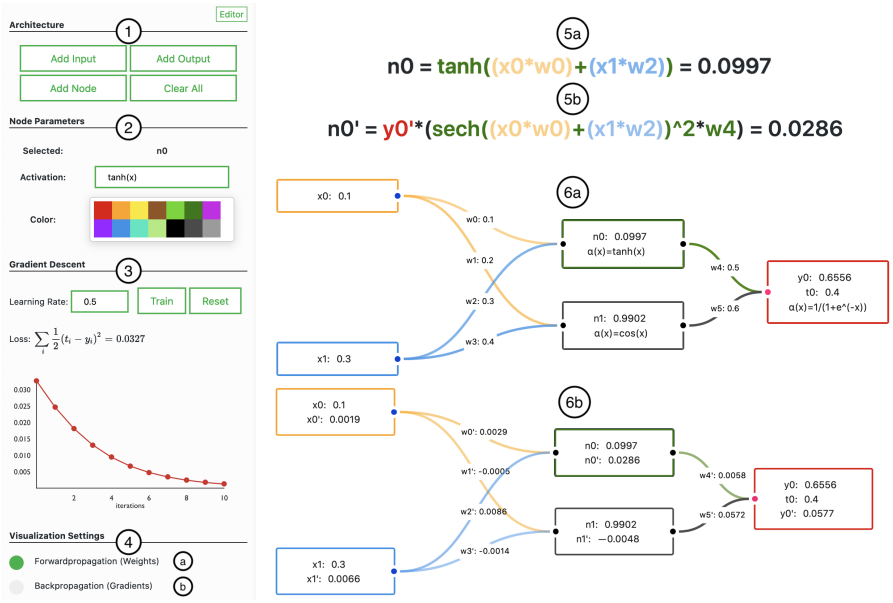


Figure 4.4: Interactive Component - ① Definition of the neural network architecture ② Parameters of the selected node e.g. color or activation function ③ Training of the neural network ④ Visualization settings for a) forwardpropagation b) backpropagation ⑤ Colorized math equations of the selected node ⑥ Flow-based visualization of the network

4.2 Interactive Component

For the main functionality, a custom ReactJS component was created. The reason for using ReactJS is because Idyll does not support any other method for creating custom components. The main interface of the component can be seen in figure 4.4. It allows users to create and train a small neural network while showing the underlying equations for forward- and backpropagation. The advantage is that users can quickly try different parameters or network topologies and reason about the flow of information at a deep level. Without in-depth an knowledge of backpropagation, their intuition might break down at some point. Such cases are extremely valuable and one of the main reasons why uncovering the backpropagation black box is important in the first place. Moreover, understanding and recognizing similar situations in a real-world deep learning project might save days of debugging. Two examples of such non-intuitive cases are given below.

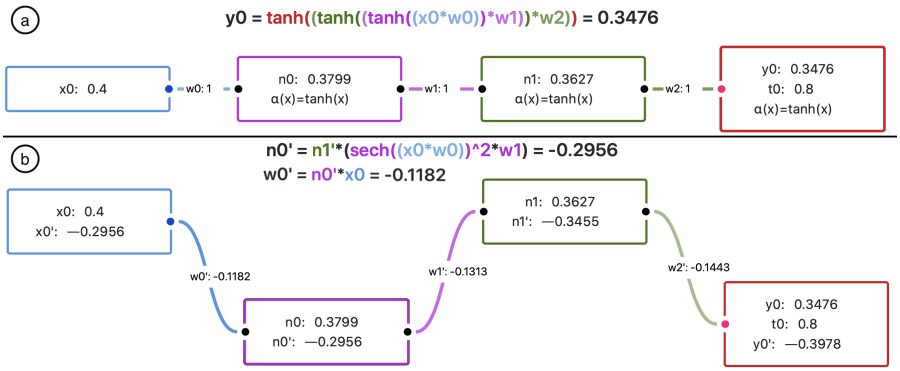


Figure 4.5: Example - Vanishing Gradient Problem (a) Forwardpropagation-View (b) Backpropagation-View

Example 1: Figure 4.5 shows two representations (a) and (b) of the same network. The view (a) shows the values for forwardpropagation, thus, input $x_0 = 0.4$, weights $w_i = 1$, activation functions $a(x) = tanh(x)$, and true value $t_0 = 0.8$. The second view (b) shows the values of interest for backpropagation, namely the gradients of all nodes and weights. But why does the magnitude of the gradient decrease $|w_0'| < |w_1'| < |w_2'|$ with each layer? After all, the weights and activation functions are the same for each layer. This non-intuitive behavior is called the vanishing gradient problem (from [Hochreiter, 1998]). The problem occurs because the gradients depend on the gradient of the previous layer. For example (see figure 4.5), the gradient of the weight $w_0' = n_0' x_0$ depends on the gradient of the previous layer $n_0' = n_1' sech(x_0 w_0)^2 w_1$ which, again, depends on the gradient of its previous layer $n_1' = ...$ etc. Due to the derivative of the activation function $tanh(x)' = sech(x)^2$, the gradient is always smaller or equal to one. When multiplying numbers less than one, the result is even smaller. This is the reason why gradients can vanish and magnitude of the effect varies with the chosen activation functions. DeepEq shows all the necessary equations to understand the vanishing gradient problem and, additionally, lets users try different activation functions and network topologies.

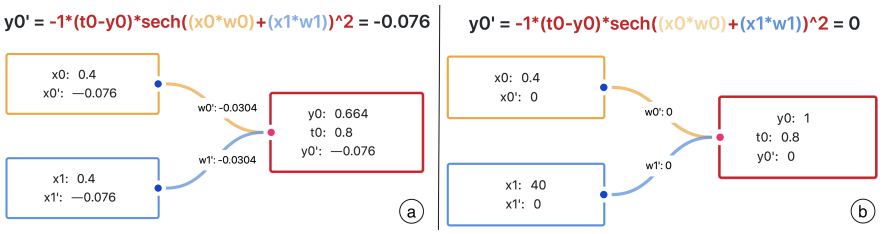


Figure 4.6: Example - Large Values (a) $x_1 = 0.4$ (b) $x_1 = 40$

Example 2: Figure 4.6 shows the gradients of the same network with $w_i = 1$ and $a(x) = tanh(x)$, the only difference being the input values (a) $x_1 = 0.4$ (b) $x_1 = 40$. But why are all gradients in (b) zero? After all, the only difference is the increase of x_1 from 0.4 to 40. The reason lies in the derivative of the activation function $tanh(x)' = sech(x)^2$. As already mentioned, the maximal value of the derivative is one. However, the minimal value rapidly approaches zero for large values. This is the reason why large input values or weights can break the gradients, and thus hinder the learning of a network significantly. With DeepEq, users can try different network topologies and see how learning is hindered by large values.

Apart from letting users try and reason about different parameters and network topologies, DeepEq also includes an integrated code editor that allows users to tinker with their own backpropagation algorithm. By implementing backpropagation, users can strengthen their understanding following a ”learning by doing” approach. The API is simple: The users can access the current graph of the network and they have to return an updated graph. There are no other restrictions aside from having to use JavaScript. This gives the users the freedom to use whatever they are comfortable with e.g. ”recursive vs. iterative”, ”hard-coded vs. generalized” etc. DeepEq already provides two implementations, a ”complete” and a ”minimal” version. The ”complete” version provides all the functionality whereas the ”minimal” only provides the numerical values without the color-highlighted equations. Both implementations have been tested against TensorFlow (from [Abadi et al., 2015]) to ensure their correctness. Figure 4.7 shows the integrated code editor with two code snippets.

Figure 4.7: Integrated Code Editor

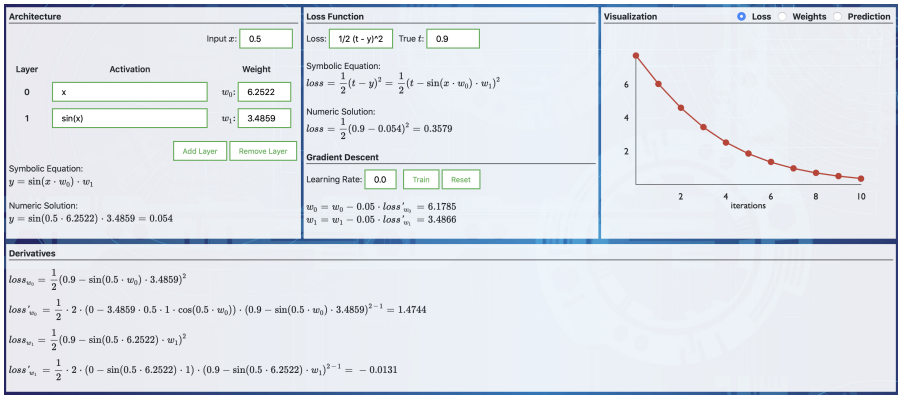


Figure 4.8: Preliminary Prototype

But why is DeepEq designed the way it is? From the very beginning of the development phase, it was a priority to include the underlying mathematics. Why? Because however good visualizations and examples may be, they do not replace adequate mathematical knowledge. Truly understanding (not just using) backpropagation requires mathematics because, in the end, it is an effective way of representing abstract concepts. Early prototypes of DeepEq (see figure 4.8) were almost purely based on mathematics and far less intuitive. Later prototypes used a flow-based visualization to better represent the flow of information. The reason for using a flow-based visualization was to build on the knowledge of visual programming languages and thus, it should support users in building a mental model more easily (from [Navarro-Prieto and Cañas, 2001, Bau et al., 2017]).

Chapter 5

Evaluation

The aim of the evaluation was to gain insights into the pedagogical effectiveness and usability of DeepEq. For this reason, the think-aloud protocol by Lewis [1982] was used as the primary method of evaluation. Lewis [1982] describes the think-aloud method as follows:

> *"The think-aloud method is a form of detailed observation of what the user is doing with the system and documentation...The special feature of the think-aloud method is that while working or reading the participant is asked to keep up a running commentary of his or her thoughts: what s/he is trying to do, what questions or confusions s/he is concerned about, what s/he expects will happen next, and the like. An observer is present to prompt the participant to keep up the flow of comments."*

The advantage of the think-aloud method is its great expressiveness while only requiring a small number of participants (from [Lewis, 1982]). The following two sections cover the method and the results.

5.1 Method

All participants were recruited from personal connections. The inclusion criteria were as follows: (1) high school level students and above (2) little or no deep learning experience. The demographics of the participants can be found in table 5.1.

ID	Gender	Age	Education Level	Deep Learning Novice
PH	Female	18	High School Student	Strongly Agree
PB	Male	25	BSc Student (CS)	Strongly Agree
PM	Male	26	MSc Student (CS)	Agree

Table 5.1: Participant Demographics

The think-aloud sessions lasted 2 hours per participant and were conducted in-person. All observations were video recorded (screen + body) whenever possible. Before the sessions, the participants were asked to fill out a pre-questionnaire about their demographics and prior deep learning knowledge. Afterward, a description of the general procedure and about the tasks they had to complete was given. Both, tasks and questionnaire, can be found in the appendix C. During the session, the observer guided the participants and decided "on-the-spot" which subtasks they had to complete. This guidance was necessary due to the limited time, difference in prior knowledge, and difference in comprehension speed. The goal was to complete all tasks rather than focusing on a single one. For example, participant PM quickly understood the concept of a perceptron, thus, he was asked to continue with the next task rather than wasting time on trivialities. On the contrary, participant PH did not know what a derivative was, thus, a short explanation was provided to be able to continue with the tasks. After the session, the participants were asked to fill out a post-questionnaire about their current deep learning knowledge and their experience with DeepEq.

In the tasks, the participants were asked to complete the first four tutorials (see chapter 4 and appendix B): "Perceptron", "Loss Function", "Gradient Descent", and "Backpropagation Theory". The last tutorial "Backpropagation Implementation" was not included due to the complexity and time limit. For a consistent setup, the participants were provided with a MacBook Pro 15" 2018. Moreover, an external mouse was given to non-Mac users because in the pilot study I found that some users may have trouble with the zoom- and scroll-behavior of the trackpad.

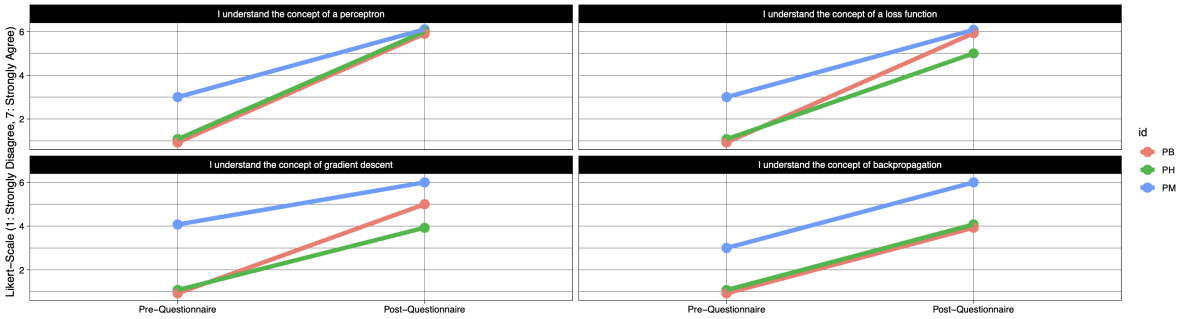


Figure 5.1: Pre-Post-Understanding Comparison

For the analysis, interesting observations were noted and summarized e.g. when the users struggled with or particularly liked something. The results were then grouped into (1) pedagogical and (2) usability aspects. Usability aspects were further sub-grouped because most participants struggled with similar issues. Pedagogical aspects, on the other hand, were used "as is" (e.g. quotes) because the aim was to gain insights rather than discovering patterns.

5.2 Results

The result section is divided into pedagogical and usability aspects. In terms of pedagogical effectiveness, the participants found DeepEq to be a valuable learning tool with a median rating of 6 out of 7. Regarding the usability, participants found DeepEq easy to use with a median rating of 6 out of a 7-point Likert-scale. Nevertheless, the think-aloud method uncovered usability issues ranging from zoom-problems to suboptimal page layouts.

5.2.1 Pedagogical Aspects

Pre-Post-Understanding: The participants were asked to rate their understanding of different deep learning topics before and after the use of DeepEq. The median within-subjects rating difference was 3.75 points which is a substantial increase in the (subjective) understanding of the topic. However, there is a possibility that the observer's guidance (e.g. giving an explanation of derivatives) influenced the results. Figure 5.1 shows the results for each participant.

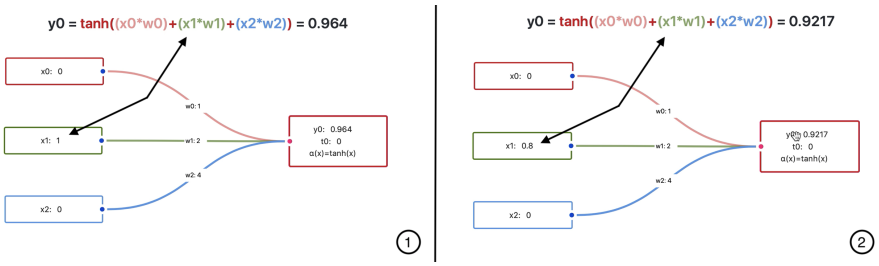


Figure 5.2: Interactive Equations

Interactivity: When the participants were asked about their opinion on the interactivity of DeepEq, most responses were positive. The participants liked the color-highlighted equations and how they could try different values themselves. On the contrary, some participants struggled with the added degree of freedom. For example, participant PM did not find the sliders (see figure 4.3) intuitive and reported that a "Try it yourself" hint would have been useful. Moreover, an improvement point was revealed by the think-aloud observations. Because of the symbolic math equations, a change of a numerical value (e.g. an input value) was not reflected in the equations (see figure 5.2 ① vs. ②). From a usability perspective, this lead to a gulf of evaluation because the current state of the system was not sufficiently communicated. But more importantly, it changed the interpretation of the equation. For example, participant PH explained the equation of a perceptron as follows:

> *"The activation function is just tan, cos, sin, or whatever one wants to use. The rest of the equation stays the same and only the value back here (points to the numerical output value) changes."*

Although the equation does indeed not change, there is a change in the flow of information (numerical values) which is why this interpretation is problematic. To avoid such problems and misinterpretations, it is advisable to always communicate the current state of the system (e.g. by using numerical values) when the user requests a change. Last but not least, participant PB mentioned after the session that the interactivity reduced the amount of time he had to read the tutorial because "trying things out" was quicker. Although this was only one statement, it might still be a good idea to have one interactive element for each important topic because users might only skim over the text.

Figure 5.3: Open-Ended Activation Function

Open-ended interaction: To support complex learning and avoid forcing students into a specific way of thinking, the tool and tutorial were kept "open" with little step-by-step instructions. However, there are downsides to this design choice. For example, participant PM wanted to use a "softmax(x)" and "logit(x)" activation function which were not supported. This lead to a gulf of execution because he would have needed to use the full mathematical equations (e.g. $log(x/(1-x))$). On the contrary, participant PH solely used the activation functions provided in the example and the observer had to guide the participant to try different activation functions e.g. trigonometric functions. Last but not least, participant PB complained that the questions of the tutorial were "too open". Yet, a good example showing the advantages of open-questions was given by the very same participant. The situation was as follows (see figure 5.3): The participant created a perceptron with three inputs (representing red, green, and blue) and he had to give an interpretation of the output. The following thoughts were recorded:

> *"The interpretation is how bright an LED shines... but then the number is over 255. At an exam, I would probably fix the problem using the activation function $a(x) = x/1.5$ (this way, the output is smaller than 255) because it is not clearly defined in the question."*

He knew that a value over 255 did not make any sense. Moreover, he knew that the inputs were fixed and he assumed that the weights were ranging from $[0, 1]$. Thus, he had to ensure that $(255 + 125)/x \leq 255$ which is ≈ 1.5. This way, the output is scaled to a value smaller than 255. This creative solution is a consequence of the vague question and shows that the participant deeply thought about (and understood) the concept of an activation function.

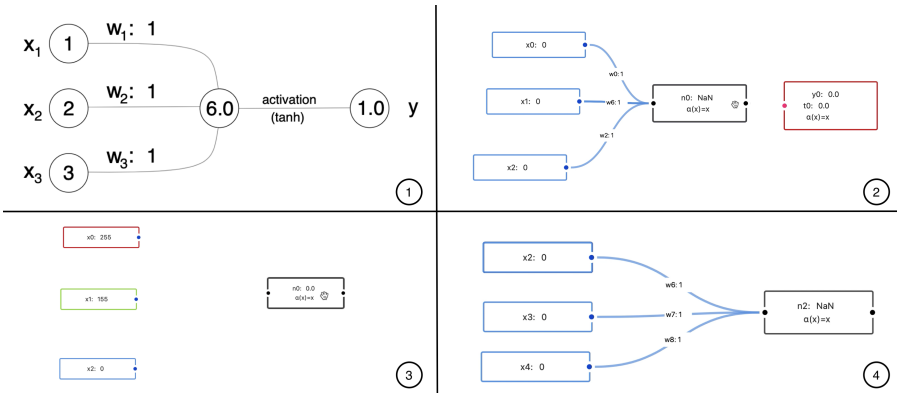


Figure 5.4: Perceptron Representations - ① Tutorial ② Participant PB ③ Participant PH ④ Participant PM

Different representations: While trying to create their first perceptron in DeepEq, all participants struggled with the representation of the perceptron. In the tutorial, the perceptron was introduced with a pre- and post-activation value (see figure 5.4 ①). However, DeepEq itself did not distinguish between pre- and post-activation value. Thus, all participants created an unnecessary middle-node (description-similarity slip). For example, the following question was recorded from participant PB (see figure 5.4 ②):

> *"Isn't this (points to the middle-node) the activation function I have to define and only afterward can I append the output at the end (creates an output node)?"*

In this case (and in a similar situation with participant PM), the observer answered the question to help the participant. In the case of participant PH, the observer only asked to carefully re-read the learning task which solved the problem. However, it still might be advisable not to use multiple similar representations of the same topic.

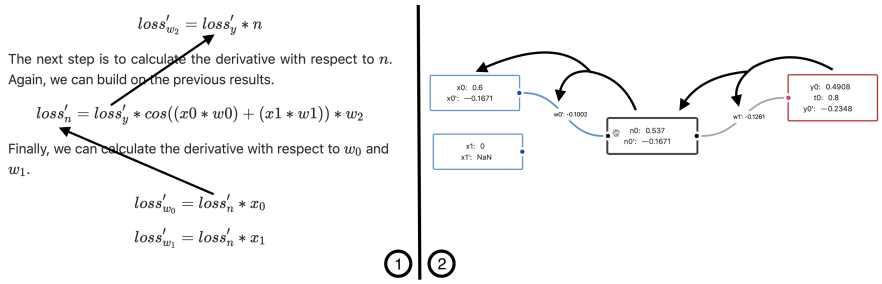


Figure 5.5: Backpropagation Thoughts - ① Tutorial ② DeepEq

Level of Difficulty: The level of difficulty increased with each topic, and thus the understanding decreased accordingly (see figure 5.2). The topics "Perceptron" and "Loss Function" were quickly and well understood by the participants. On the other hand, "Gradient Descent" and "Backpropagation" were more challenging according to the participants. That was to be expected due to the mathematical nature of those topics. The third-year high school student PH had a particular disadvantage because she did not know what a derivative (or the chain rule) was. However, she still understood that the equations of backpropagation depend on each other and that the error is propagated backward. For example, the following thoughts were recorded while reading the tutorial (see figure 5.5 ①):

> *"You start with one equation, plug it into the next one, and again into the next one....$loss'_n$ is from above and $loss'_y$ from here (points to the uppermost equation)"*

...and the thoughts while using the visualization (see figure 5.5 ②):

> *"This (middle node) times this (weight) equals this (output node). And here, it takes this (input node) multiplied with this (weight) which equals this (middle node)....Actually, you can do this indefinitely and it can always take the value from the front."*

Interestingly, her mental model does not seem to be far off from the mathematical description, presumably because DeepEq shows all the necessary equations without having to manually calculate the derivatives (which she was not able to do).

Question	Median
It is a useful tool for learning about “deep learning”	6
The tool improves the effectiveness of my learning	6
The tool helped me connect the mathematical equations with their corresponding visual representations	6
The tutorial was useful and easy to understand	6
The interactive elements of the tutorial helped me understand the topic better	7

Table 5.2: Participant Questionnaire Results 7-point Likert (1: Strongly Disagree 7: Strongly Agree)

Questionnaire: Last but not least, the participants were asked to rate several aspects of DeepEq. In general, the responses were positive which indicates that DeepEq was an effective tool for learning. The tutorial and its' interactive elements received particularly high ratings. The results can be found in table 5.2.

5.2.2 Usability Aspects

Reset problem: One issue participants frequently encountered was during a reset of the network. Sometimes, the participants changed the activation function of a node and pressed ”reset” afterward because they wanted to reset the weight values. However, due to the implementation of the reset function, the whole network (including the activation function) was set to the last known state. This ”whole reset” was unnecessary for the use cases that the participants have encountered which is why the reset behavior has been restricted to weights only.

Selection highlighting: When the participants had to change network parameters such as weights or activation functions, the selected node or edge was not highlighted. This lead to a gulf of evaluation because the users thought that they had missed the node/edge and clicked again. This minor problem was solved by changing the border color of the selected node or edge.

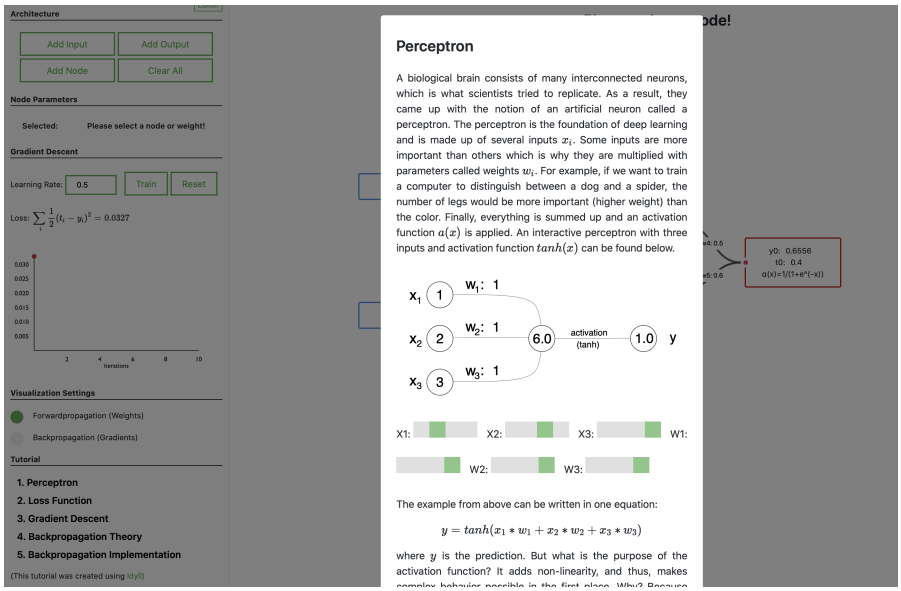


Figure 5.6: Tutorial Visibility

Layout: When the participants were asked about certain sub-questions of the tutorial (during the task because e.g. the participants skipped one), they (mostly) had to re-open the tutorial and look for the specific information. Moreover, some sub-questions were completely skipped/forgotten by the participants. Figure 5.6 illustrates the issue. The tutorial was implemented as a modal that blocks the main view when opened. This was a deliberate design choice because modals require only a minimal amount of space. However, for a better user experience, references should always be accessible to the user. For example, a retractable sidebar might be an alternative with similar space requirements but with the option to keep the tutorial visible at all times.

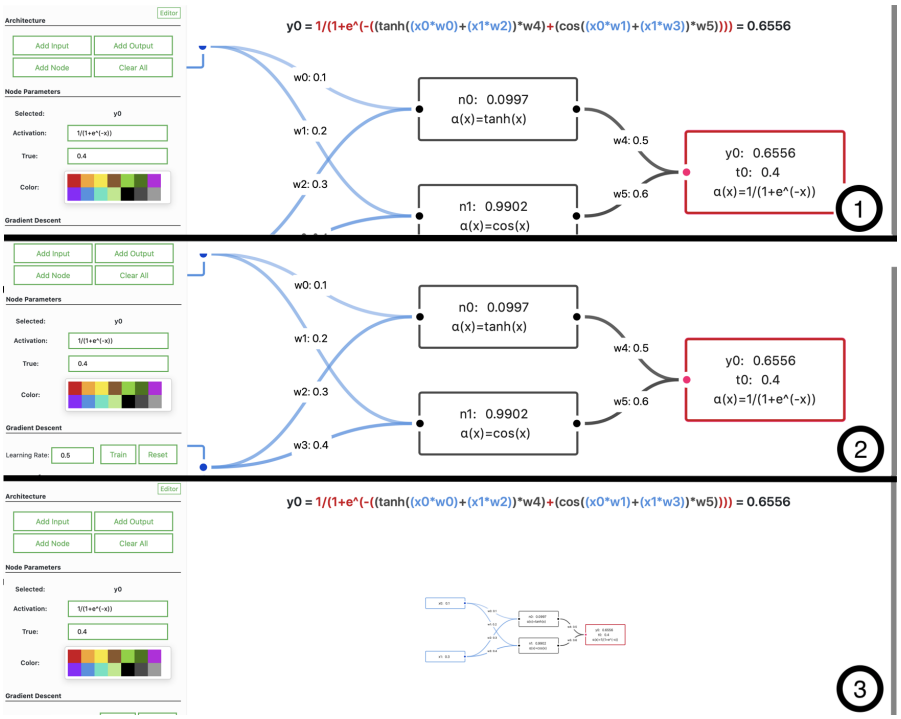


Figure 5.7: Zoom And Scrolling Behavior - ① Original view ② Hidden equation due to scrolling ③ Change of graph size due to scrolling

Zoom and scrolling behavior: Participants often had to scroll down the page because selecting certain nodes increased the vertical size of the sidebar (see left-hand side of figure 5.7 ①). This was not a problem as long as the participants used the scroll-wheel on the sidebar. However, when the scroll-wheel was used on the graph view (see right-hand side of figure 5.7 ①), no scrolling action was triggered. Instead, the graph changed in size similar to the behavior of Google Maps. This was problematic in two different ways: 1) Certain elements e.g. equations remained hidden (mostly unconscious) from the user (see figure 5.7 ②) 2) The users had to undo the resizing of the graph and/or were confused about the sudden disappearance of the graph (disappearance = extremely small, see figure 5.7 ③). At first, participants frequently encountered this problem but they quickly adapted to the interaction method. Nevertheless, to separate zoom and scroll behavior, ”zoom buttons” were added. Moreover, by overriding the scroll-wheel zoom behavior, it is now possible to use the whole page as a scroll area.

Chapter 6

Summary

In this thesis, I have created DeepEq, an educational platform to help students develop a mathematical intuition for deep learning. DeepEq allows users to create a small artificial neural network by hand, shows the underlying equations, lets users change and reason about different parameters, and provides a safe environment for implementing one's own backpropagation algorithm. Moreover, I have created a complementary interactive tutorial based on the 4C/ID model. Last but not least, I have conducted three think-aloud sessions to gain insights into the pedagogical effectiveness of DeepEq.

6.1 Limitations

Regarding the evaluation, several limitations exist. First, there are limitations to the think-aloud method itself e.g. participants may not be used to "speaking while thinking". This was particularly noticeable because participant PB was more talkative and could express himself better than the other participants. Second, due to the limited time of the think-aloud sessions, the observer had to guide the participants and decide which subtasks they had to complete. This guidance likely influenced the results, especially of the questionnaire e.g. due to unequal time distribution per topic. Third, the low number of participants limits the generalizability of the results. In terms of the implementation, there is one major limitation. Due to the use of Idyll/ReactJS/JavaScript, bugs are highly likely and maintainability issues are inevitable. It would have been preferable to use purely-functional and type-safe languages such as Haskell/PureScript.

6.2 Future Work

I strongly believe that explorable explanations and other interactive media will become even more important in the future. However, as I have seen myself, the entry barriers for creating explorable explanations are quite high and the amount of time required is enormous. Idyll by Conlen and Heer [2018] certainly lowered the entry barrier for interactive articles but non-standard interactive elements (e.g. the main component of this thesis) still require good programming skills. After all, a custom Idyll component is simply a wrapper around ReactJS. Moreover, there does not seem to be a standardized and efficient way of creating explorable explanations which is, in my opinion, the biggest issue yet to be solved (e.g. by a programming language for explorable explanations). Until then, I can't recommend extending DeepEq or other explorable explanations because most of them have been built on a completely different codebase which requires a disproportionate amount of effort to familiarize oneself with. Apart from that, there are still many features I would have liked to implement e.g. support for datasets, bias value, an efficient method for creating larger networks, support for convolutional/recurrent neural networks, etc. Moreover, I think that newer topics in computer science (e.g. Blockchain) would also benefit from interactive explanations.

Appendix A

JavaScript Implementation Of Backpropagation

```
1
2  /** Description of the variables
3
4   * graph
5
6    Description: "An array of objects containing the data
         of all nodes and edges"
7    Note: Use the filter function to traverse the graph e
         .g. graph.filter(element => element.target == "n0
         ")
8    Values Node:
9     - id: the id of the node
10    - data.activation: a string of the activation
          function e.g. tanh(x)
11    - data.sum: value assigned by you, see "result"
12    - data.value: value assigned by you, see "result"
13    - data.gradient: value assigned by you, see "result"
14    - data.symbolicEquation: value assigned by you, see
          "result"
15    - data.symbolicGradient: value assigned by you, see
          "result"
16   Example Node:
17    {id: "n0", data: {sum: 0.1, value: 0.0997, gradient:
          0.0285, activation: "tanh(x)" }}
18   Values Edge:
19    - id: id of the edge
20    - source: id of the starting point of the edge
21    - target: if of the end point of the edge
```

```
22    - data.value: the value of the weight/edge
23    - data.gradient: value assigned by you, see "result"
24    - data.symbolicEquation: value assigned by you, see
        "result"
25    - data.symbolicGradient: value assigned by you, see
        "result"
26   Example Edge:
27    {id: "w2", source: "x0", target: "n1", data: {value:
        0.2, gradient: -0.0005 }}
28
29
30  * math
31
32   Description: "A reference to math.js, an extensive
       math library for JavaScript"
33   Example: derivative of tanh: math.derivative('tanh(x)
       ', 'x');
34
35
36  * results
37
38   Description: "An empty array of objects to be filled
       with your results"
39   Values:
40    - id: the id of the node or edge (mandatory)
41    - sum: the weighted sum e.g. x1*w1+...+xn*wn (
        optional)
42    - value: value after the activation function e.g. a(
        x1*w1+...+xn*wn) (optional)
43    - gradient: the value of the calculated gradient (
        optional)
44    - symbolicEquation: the mathematical formula of the
        forwardpropagation pass (optional)
45    - symbolicGradient: the mathematical formula of the
        backpropagation pass (optional)
46   Example:
47    let result =
48    {
49      id: "n1", sum: 0.14, value: 0.99, gradient:
          -0.005,
50      symbolicEquation: "cos((x0*w1) + (x1*w3))",
          symbolicGradient: "y0'*(-sin....)"
51    });
52    results.push(result);
53
54  */
55
56
57
```

```
58 // run forward propagation for each output node
59 let forwardresults = [];
60 let outputNodes = graph.filter(element => element.id.
       includes("y"));
61 for (let i in outputNodes) { forwardpropagation(graph,
       outputNodes[i], forwardresults); }
62
63
64 // run backpropagation for each input node
65 let backpropresults = [];
66 let inputNodes = graph.filter(element => element.id.
       includes("x"));
67 for (let i in inputNodes) { backpropagation(graph,
       inputNodes[i], forwardresults, backpropresults); }
68
69
70 // merge and assign results
71 for (let i = 0; i < backpropresults.length; i++) {
72   results.push({
73     ...backpropresults[i],
74     ...(forwardresults.find((e) => e.id ==
          backpropresults[i].id))
75   });
76 }
77
78
79 // recursive forwardpropagation algorithm
80 function forwardpropagation(graph, node, results) {
81   // get all ingoing edges from node
82   let edges = graph.filter(element => element.target ==
         node.id);
83   if (edges.length > 0) {
84     let sum = 0;
85     for (let j in edges) {
86       let edge = edges[j];
87       let nextNode = graph.filter(element => element.id
           == edge.source)[0];
88       // recursive call
89       let nextNodeValue = forwardpropagation(graph,
           nextNode, results);
90       // calculate weighted sum of perceptron e.g. x1*
           w1+...+xn*wn
91       sum += edge.data.value * nextNodeValue;
92     }
93     // apply activation function e.g. a(x1*w1+...+xn*wn
         )
94     let value = math.evaluate(node.data.activation.
         replace('x', sum));
95     // save result
```

```
96      let result = { id: node.id, sum: sum, value: value
            };
97      results.push(result);
98      // return value for the next recursive call
99      return value;
100   } else if (node.id.includes("x")) {
101     // stop recursion when reaching an input node
102     let result = { id: node.id, value: node.data.value
            };
103     results.push(result);
104     return node.data.value;
105   } else {
106     // stop recursion when reaching a node without
          connections
107     return NaN;
108   }
109 }
110
111
112 // recursive backpropagation algorithm
113 function backpropagation(graph, node, forwardresults,
      results) {
114   // get all outgoing edges from node
115   let edges = graph.filter(element => element.source ==
        node.id);
116   if (edges.length > 0) {
117     let totalGradient = 0.0;
118     for (let j in edges) {
119       let edge = edges[j];
120       let nextNode = graph.filter(element => element.id
            == edge.target)[0];
121       // recursive call
122       let nextNodeGradient = backpropagation(graph,
            nextNode, forwardresults, results);
123       // use latest data from forward propagation
124       let tmp = forwardresults.filter(element =>
            element.id == node.id)[0];
125       let nodeSum = tmp.sum;
126       let nodeValue = tmp.value;
127       // calculate gradient
128       let outerDerivative = math.derivative(node.data.
            activation, 'x');
129       totalGradient += nextNodeGradient *
            outerDerivative.evaluate({ x: nodeSum }) *
            edge.data.value;
130       // save result for edge
131       let edgeGradient = { id: edge.id, gradient:
            nextNodeGradient * nodeValue };
132       results.push(edgeGradient);
```

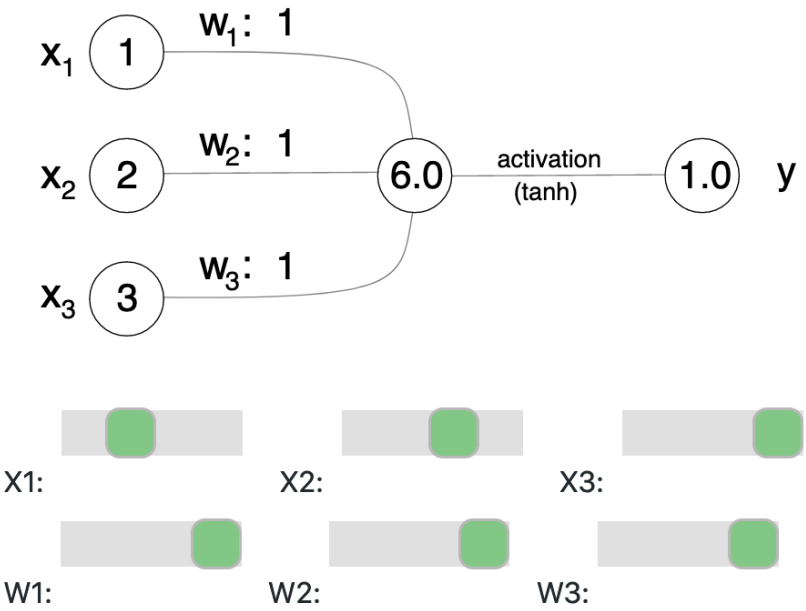
```
133       }
134       // save result for node
135       let nodeGradient = { id: node.id, gradient:
              totalGradient };
136       results.push(nodeGradient);
137       // return values for the next recursive call
138       return totalGradient;
139     } else if (node.id.includes("y")) {
140       // stop recursion when reaching an output node
141       let tmp = forwardresults.filter(element => element.
              id == node.id)[0];
142       let nodeSum = tmp.sum;
143       let nodeValue = tmp.value;
144       let outerDerivative = math.derivative(node.data.
              activation, 'x');
145       // save result
146       let nodeGradient = {
147         id: node.id,
148         gradient: -1 * (node.data.true - nodeValue) *
                outerDerivative.evaluate({ x: nodeSum }),
149       };
150       results.push(nodeGradient);
151       return nodeGradient.gradient;
152     } else {
153       // stop recursion when reaching a node without
              connections
154       return NaN;
155     }
156   }
```

Appendix B

Tutorial

Perceptron

A biological brain consists of many interconnected neurons, which is what scientists tried to replicate. As a result, they came up with the notion of an artificial neuron called a perceptron. The perceptron is the foundation of deep learning and is made up of several inputs x_i. Some inputs are more important than others which is why they are multiplied with parameters called weights w_i. For example, if we want to train a computer to distinguish between a dog and a spider, the number of legs would be more important (higher weight) than the color. Finally, everything is summed up and an activation function $a(x)$ is applied. An interactive perceptron with three inputs and activation function $tanh(x)$ can be found below.



The example from above can be written in one equation:

$$y = tanh(x_1 * w_1 + x_2 * w_2 + x_3 * w_3)$$

where y is the prediction. But what is the purpose of the activation function? It adds non-linearity, and thus, makes complex behavior possible in the first place. Why? Because the result of chaining two linear perceptrons together is still linear, thus, does not provide a better prediction:

$$y_1 = x_1 * w_1$$

$$y_2 = y_1 * w_2 = x_1 * \overbrace{w_1 * w_2}^{w}$$

It is important to keep in mind that that the input x represents anything a computer can measure e.g. a pixel. Generally, a pixel can have a value between 0 and 255 where zero means that the pixel is turned off. By mixing red, green, and blue pixels, nearly every color can be created e.g. (red=255, green=255, blue=0) results in yellow. A (digital) image is simply a composition of millions of pixels.

It is important to keep in mind that that the input x represents anything a computer can measure e.g. a pixel. Generally, a pixel can have a value between 0 and 255 where zero means that the pixel is turned off. By mixing red, green, and blue pixels, nearly every color can be created e.g. (red=255, green=255, blue=0) results in yellow. A (digital) image is simply a composition of millions of pixels.

Value: 128

Learning Task

Create a perceptron with red, green, and blue pixels as inputs and connect them to an output node. Try different input values, weights, and activation functions and think about the effect they have on the flow of information.

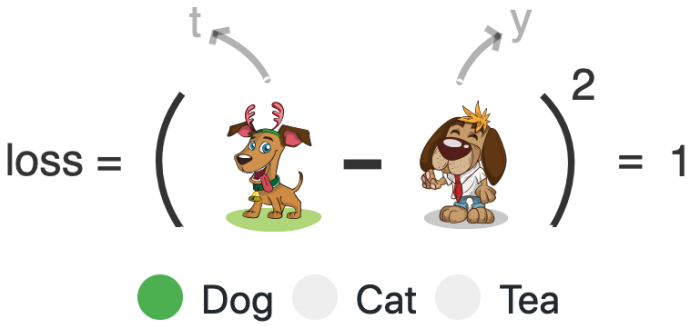
- What is the equation of the perceptron?
- Use the equation to calculate the prediction value of e.g. red=255, green=120, blue=0, weights=0.5, a(x)=x. What is the interpretation?
- Change the weight of the red pixel to 1 and all other weights to 0. What is the interpretation?
- Use the activation function $a(x) = tanh(x)$. What is the value of the prediction? How does the interpretation change? When is $tanh(x)$ -1 or 1?
- Another popular activation function is called "sigmoid" ($a(x) = 1/(1 + e^{-x})$). Draw the sigmoid function as well as $a(x) = x$ and $a(x) = tanh(x)$. What range can they take?

Loss Function

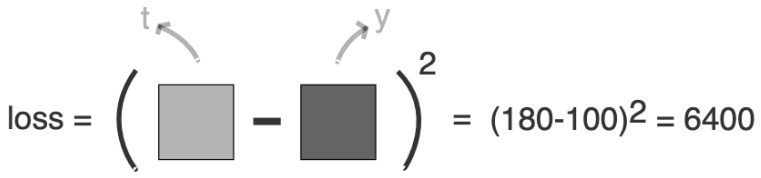
Identifying and naming objects in an image is no challenge for humans. Every child can identify a dog and, if not, learn it in an instant e.g. by showing a dog nearby. To translate this "learning by example" to a computer, it is necessary to quantify (express as numbers) this process. Just like grades in school, the computer has to know how good it is. For example, when a computer identifies a car instead of a dog, the grade should be low. When it mistakes a Yorkshire Terrier for a West Highland Terrier, it should get a comparatively high grade. In machine learning terms, this is called a loss function. As opposed to school grades - the smaller the loss, the better:

$$loss = \frac{1}{2}(t - y)^2$$

where t is the true label a human provides, and y is the prediction of the computer. Why is it multiplied by $1/2$ and squared? Because $1/2$ simplifies the derivative (next section), and the square creates a lower boundary for the loss (deep learning minimizes the loss, next section). There is no particular reason to choose this exact loss function (e.g. $|t - y|$ works as well), but the mathematical properties make it easier to work with.



As mentioned in the previous section, an image consists of millions of pixels. Hence, the loss has to be calculated (and summed together) for each of those pixels. For example, a loss function with one pixel might look like this:



True:

Predicted:

Learning Task

Use the perceptron from the previous task. Try different true values t and think about how they affect the loss. Moreover, think about how the input, weights, and activation function change the loss.

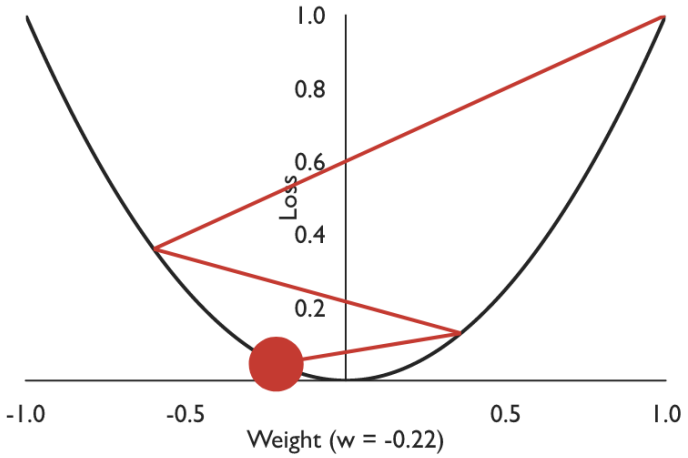
- Because of the range of $tanh(x)$ (between -1 and 1), it is better to use inputs with a similar range. For example, a pixel can be represented by values between zero and one where 1 means completely turned on).
- When is it possible for a loss to never approach zero? Try different true values and activation functions.

Gradient Descent

Just like a student, the computer should try to improve its' grades (the actual "learning"). In other words, it should try to minimize the loss function, because, the smaller the loss, the better the prediction. But how to minimize a function? The most straightforward answer is by taking the derivative and set it equal to zero $loss' = 0$. However, this analytical approach does not work well for complex functions with millions of weights as used in deep learning. For this reason, an iterative approach called "gradient descent" is used. The word "gradient" is just another word for "derivative" (with multiple variables). Instead of setting the derivative equal to zero, the weights are adjusted step-by-step by going in the opposite direction of the derivative. As you may have guessed, "learning" is nothing more than adjusting the weights w such that the loss gets smaller (better grades).

$$w_{new} = w_{old} - LearningRate * loss'$$

For example, given $loss = w^2$ with derivative $loss' = 2w$, we have to choose a starting weight (e.g. $w = 1$) and a learning rate (e.g. $LearningRate = 0.8$). In the first step, the weight is adjusted to $w_{new} = 1 - 0.8 * (2 * 1) = -0.6$, in the second step to $w_{new} = -0.6 - 0.8 * (2 * -0.6) = 0.36$ etc. As you see, the weight gradually approaches zero because this is where the loss is the smallest. Similar to a ball rolling off a mountain, the weights change the fastest where the slope is the steepest. Keep in mind that the learning rate has to be large enough to make a difference each step but not too large because it can overshoot. A visualization with adjustable learning rate (e.g. 0.1 is relatively slow and 1.1 overshoots) can be found below.



Learning Rate: 0.80 Train Reset Show Line:

Learning Task

Use the perceptron from the previous task and visualize the gradients. Train the perceptron, use different learning rates and think about how the gradient is used to adjust the weight.

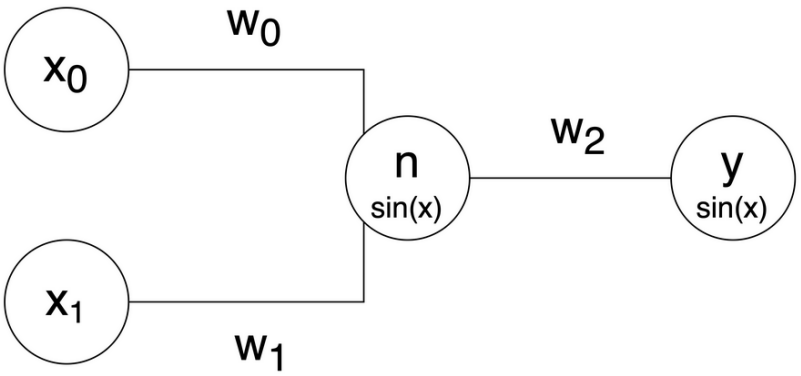
- What happens when the learning rate is either too small or too large?
- What happens to the gradient when t changes? Why is the gradient zero when $t = y$?
- What happens to the gradient when x changes? Why is the gradient zero when $x = 0$?
- What happens to the gradient when $a(x)$ changes? Why is the gradient zero with $a(x) = tanh(x)$ and a large input (e.g. 255)?
- Choose one specific weight and predict how it will change when you train the perceptron.
  $w_{new} = w_{old} - LearningRate * loss'$

Backpropagation Theory

Gradient descent uses the derivative to minimize the loss function. But how to calculate the derivative of multiple chained perceptrons (layers)? This is where backpropagation is used. Mathematically, backpropagation is simply the repeated application of the chain rule:

$$f'(x) = g'(h(x)) * h'(x)$$

For example, given the following network:



$$n = sin((x_0 * w_0) + (x_1 * w_1))$$

$$y = sin(n * w_2)$$

The goal is to calculate the derivative of each weight (how much the loss changes when the weight changes). Due to the chain rule, it makes sense to start at the outermost weight and continue till reaching the innermost weight. First, we have to calculate the derivative with respect to y. This means that everything except y is treated as a constant (e.g. insert numerical values such as $t = 5$).

$$loss = \frac{1}{2}(t - y)^2$$

$$loss'_y = (t - y) * -y' = (t - y) * -cos(n * w_2)$$

Now we can calculate the derivative with respect to w_2. Since we already have $loss'_y$, there is no need to start from scratch. We can simply continue at the derivative of $n * w_2$, and because everything except w_2 is treated as a constant, the derivative is just n.

$$loss'_{w_2} = loss'_y * n$$

The next step is to calculate the derivative with respect to n. Again, we can build on the previous results.

$$loss'_n = loss'_y * cos((x0 * w0) + (x1 * w1)) * w_2$$

Finally, we can calculate the derivative with respect to w_0 and w_1.

$$loss'_{w_0} = loss'_n * x_0$$

$$loss'_{w_1} = loss'_n * x_1$$

As we can see, the derivatives build on each other which is why it is called backpropagation (backward propagation of the errors). In a way, backpropagation is similar to a group project with dependent tasks. The first student has to finish his/her task and passes it on to the next student (dividing the learning).

Learning Task

For real-world deep learning projects, an understanding of the gradients is essential, especially for debugging. When the gradient is too small or too large, the network will not be able to learn. Try different network topologies and see how they change the flow of the gradient.

- What happens to the gradient when multiple perceptrons with $tanh(x)$ activation function are chained together? (Keyword: vanishing gradient)
- What happens to the gradient when a weight is extremely large? Why can it break the learning of a network? (Keyword: random weights)

Backpropagation Implementation

Calculating derivatives by hand is cumbersome and error-prone. For this reason, it is a good idea to automate this process. The algorithm is usually divided into forwardpropagation and backpropagation. The forwardpropagation starts from the inputs, calculates the value of the perceptron, and repeats this till every perceptron has a value. The backpropagation builds on the forwardpropagation results and calculates the derivatives of each weight.

For example, a simple recursive forwardpropagation function can be found below. In general, the forwardpropagation algorithm is easy to implement (iteratively or recursively) because it does not rely on higher mathematics. Note that the provided code-examples do not save results nor apply optimization techniques (for simplicity reasons).

```
function forwardpropagation(graph, node, results) {
  // get all ingoing edges from node
  let edges = graph.filter(e => e.target == node.id);
  if (edges.length > 0) {
    let sum = 0;
    for (let j in edges) {
      let edge = edges[j];
      let nextNode = graph.filter(e => e.id == edge.sou
      // recursive call
      let nextNodeValue = forwardpropagation(graph, ne
      // calculate weighted sum of perceptron e.g. x1*
      sum += edge.data.value * nextNodeValue;
    }
    // result --> apply activation function e.g. a(x1*
    let value = math.evaluate(node.data.activation.rep
    return value;
  } else if (node.id.includes("x")) {
    // stop recursion when reaching an input node
    return node.data.value;
  } else {
    return NaN;
  }
}
```

```
function backpropagation(graph, node, forwardresults, 
  // get all outgoing edges from node
  let edges = graph.filter(element => element.source ==
  if (edges.length > 0) {
    let totalGradient = 0.0;
    for (let j in edges) {
      let edge = edges[j];
      let nextNode = graph.filter(element => element.id
      // recursive call
      let nextNodeGradient = backpropagation(graph, nex
      // calculate gradient
      let outerDerivative = math.derivative(node.data.a
      totalGradient += nextNodeGradient * outerDerivati
      // result --> the gradient of the weight
      let edgeGradient = nextNodeGradient * node.data.v
    }
    // result --> the total gradient is returned (propa
    return totalGradient;
  } else if (node.id.includes("y")) {
    // stop recursion when reaching an output node
    let outerDerivative = math.derivative(node.data.act
    return  -1 * (node.data.true - node.data.value) * o
  } else {
    return NaN;
  }
}
```

Learning Task

Implement your own version of backpropagation because it is the best way to deepen your understanding.

- Feel free to have a look at the provided implementations. “Complete” supports symbolic equations whereas “Minimal” is kept as simple as possible.
- Remember the theory! It is almost a 1:1 implementation of the theory.

$$loss'_y = (t - y) * -y'$$

```
let outerDerivative = math.derivative(node.data.activat
return  -1 * (node.data.true - node.data.value) * outer
```

Appendix C

Questionnaire

Think-aloud

Say everything that comes to your mind aloud while completing the tasks below. For example: "First, I have to create a perceptron...I don't understand why..."

Tasks

1) Open: https://pgigeruzh.github.io/DeepEq
2) Complete the first four tutorials (1-4)

Pre-Questionnaire

		Strongly Disagree			**Neither**		**Strongly Agree**	
		1	**2**	**3**	**4**	**5**	**6**	**7**
01	I am a deep learning novice (7: if you can't draw a "perceptron")							
02	I understand the concept of a **perceptron** and its implications							
03	I understand the concept of a **loss function** and its implications							
04	I understand the concept of **gradient descent** and its implications							
05	I understand the concept of **backpropagation** and its implications							
06	What is your gender?							
07	How old are you?							
08	What is your current education level (e.g. MSc CS)?							

Post-Questionnaire:

		Strongly Disagree			**Neither**		**Strongly Agree**	
		1	**2**	**3**	**4**	**5**	**6**	**7**
09	I understand the concept of a **perceptron** and its implications							
10	I understand the concept of a **loss function** and its implications							
11	I understand the concept of **gradient descent** and its implications							
12	I understand the concept of **backpropagation** and its implications							
13	The software was easy to use							
14	It is a useful tool for learning about “deep learning”							
15	The tool improves the effectiveness of my learning							
16	The tool helped me connect the mathematical equations with their corresponding visual representations							
17	The tool helped me develop a mathematical intuition for deep learning							
18	The tutorial was useful and easy to understand							
19	The interactive elements of the tutorial helped me understand the topic better							
20	Comments:							

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`.

Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador García, Sergio Gil-López, Daniel Molina, Richard Benjamins, et al. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion*, 58: 82–115, 2020.

David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. Learnable programming: Blocks and beyond. *Commun. ACM*, 60(6):72–80, May 2017. ISSN 0001-0782. doi: 10.1145/3015455. URL `https://doi.org/10.1145/3015455`.

Matthew Conlen and Jeffrey Heer. Idyll: A markup language for authoring and publishing interactive articles on the web. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, page 977–989. Association for Computing Machinery, 2018. ISBN 9781450359481. doi: 10.1145/3242587.3242600. URL `https://doi.org/10.1145/3242587.3242600`.

Stefania Druga. *Growing up with AI: Cognimates: from coding to teaching machines.* PhD thesis, Massachusetts Institute of Technology, 2018.

Google. Teachable machine, 2017. URL https://teachablemachine.withgoogle.com. Accessed: 2020-08-11.

Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

Amy J. Ko. We need to learn how to teach machine learning, 2017. URL https://medium.com/bits-and-behavior/we-need-to-learn-how-to-teach-machine-learning-acc78bac3ff8. Accessed: 2020-08-11.

Dale Lane. Machine learning for kids, 2017. URL https://machinelearningforkids.co.uk/. Accessed: 2020-08-24.

C. Lewis. *Using the "thinking Aloud" Method in Cognitive Interface Design*. Research report. IBM T.J. Watson Research Center, 1982.

Lívia S Marques, Christiane Gresse von Wangenheim, and Jean CR HAUCK. Teaching machine learning in school: A systematic mapping of the state of the art. *Informatics in Education*, 19(2):283–321, 2020.

Raquel Navarro-Prieto and Jose J Cañas. Are visual programming languages better? the role of imagery in program comprehension. *International Journal of Human-Computer Studies*, 54(6):799–829, 2001.

Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

Shaileen Crawford Pokress and José Juan Dominguez Veiga. Mit app inventor: Enabling personal mobile computing, 2013.

A. Rao, A. Bihani, and M. Nair. Milo: A visual programming environment for data science education. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 211–215, 2018.

Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, November 2009. ISSN 0001-0782. doi: 10.1145/1592761.1592779. URL http://doi.acm.org/10.1145/1592761.1592779.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 (6):386, 1958.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323 (6088):533–536, 1986. doi: 10.1038/323533a0. URL `http://www.nature.com/articles/323533a0`.

Daniel Smilkov, Shan Carter, D. Sculley, Fernanda B. Viégas, and Martin Wattenberg. Direct-manipulation visualization of deep networks, 2017.

Elisabeth Sulmont, Elizabeth Patitsas, and Jeremy R. Cooperstock. Can you teach me to machine learn? In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, SIGCSE '19, page 948–954, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450358903. doi: 10.1145/3287324.3287392. URL `https://doi.org/10.1145/3287324.3287392`.

David S. Touretzky, Christina Gardner-McCune, Fred Martin, and Deborah W. Seehorn. Envisioning ai for k-12: What should every child know about ai? In *AAAI*, 2019.

Jeroen JG Van Merriënboer, Richard E Clark, and Marcel BM De Croock. Blueprints for complex learning: The 4c/id-model. *Educational technology research and development*, 50(2):39–61, 2002.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

Kevin Zhu. An educational approach to machine learning with mobile applications. Master's thesis, Massachusetts Institute of Technology, 2019.

Typeset January 21, 2021