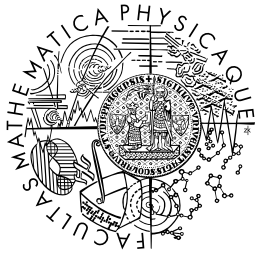


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Martin Smolík

Artificial Intelligence and Automated Reasoning

Department of Algebra

Supervisor of the master thesis:	Mgr. Josef Urban, Ph.D.
Study programme:	Mathematical structures
Study branch:	study branch

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Dedication.

Title: Artificial Intelligence and Automated Reasoning

Author: Martin Smolík

Department: Department of Algebra

Supervisor: Mgr. Josef Urban, Ph.D., Department of Algebra

Abstract: In this thesis we aim to build algebraic models in computer using machine learning methods. We start with a set of axioms that describe functions and constants and use them to train neural networks approximating them. Every element is represented as a real vector, so that neural networks can operate on them. We also explore and compare different representations. Main focus in this thesis are groups. We build representations for cyclic (the simplest) and symmetric (the most complex) groups. Another part of this thesis is the aim to extend these built models by introducing new "algebraic" elements, not unlike the classic extension of rational numbers $\mathbb{Q}[\sqrt{2}]$.

Keywords: Machine learning Automated reasoning Model theory Neural networks

Contents

Introduction **2**

1 Background **3**
1.1 Model theory 3
1.1.1 Basic definitions 3
1.1.2 Skolemization 5
1.2 Neural networks 7
1.2.1 What is a neural network 7
1.2.2 Example: XOR 8
1.2.3 Other activation functions 10
1.2.4 Learning and optimization 11
1.2.5 Back-propagation 13
1.2.6 Adam optimizer 14

2 Implementation **17**

3 Results **18**

Conclusion **19**

Bibliography **20**

Introduction

In the world of automated theorem proving, there had always been a disconnect between how computers and humans do mathematics. One of the reasons why computers have not been able to emulate human reasoning is the human capacity for intuition. When working with a well known structure (e.g. field of real numbers), they operate with its mental image. Therefore they can usually correctly guess whether or not will a given sentence hold or not. This assessment is possible even if the human can not prove the sentence using a formal proof system. Computers have so far been unable to do this estimation very well, mostly because they operate only with axioms of the structures and have no such image.

We try to build this image, so that it can later be used to do these estimations. The ultimate aim is to have an oracle that guesses "truthiness" of given sentences based on a number of pre-trained models of the theory.

Another crucial aspect of intuitive understanding of structures is the ability to naturally extend them. Most structures have extensions that can be called "algebraic", i.e. those that arise when we add solutions to an equation that has no solution in the structure itself. Best known examples are algebraic extensions of rings, for example using the equation $x^2 - 2 = 0$ in $\mathbb{Z}$ or $x^2 + 1 = 0$ in $\mathbb{R}$.

The structures that we work with, however, can be quite complex. So complex in fact, that handcrafting a model that the computer can work with gets quite challenging. That is why we want to use machine learning. Namely we use the universal property of the neural network. Neural network learning, however, needs to work with differentiable functions, that do not exist in most structures. Therefore we choose a representation of the structure elements in $\mathbb{R}^n$. Since this is not an embedding in the true algebraic sense (it does not necessarily preserve structure), we will call it *Grounding*.

Here we will use exclusively handpicked groundings that give us better insight into the preformance of the model. In the future, however, we would like to move to self-found groundings. They might be found using recursive neural networks or other machine learning algorithms.

Using this neural modeling technique and the framework Tensorflow (Abadi et al. [2015]) we have built models and extensions of groups, namely $\mathbb{Z}_n$ and S_n. Results of this experimentation can be found in chapter 3. The framework built for this purpose is robust in the sense that representing other structures will not be much trouble. However, it still can not represent relations. It would be possible to build such an extension, but that is beyond the scope of this thesis.

1. Background

In this section we will discuss the fundamentals of both model theory and neural networks. Both are essential to understanding the rest of the thesis, but the reader well acquainted with them may skip this chapter.

1.1 Model theory

Model theory is the area of mathematics that studies mathematical structures through the lens of mathematical logic.

1.1.1 Basic definitions

Most of these definitions are paraphrased from Weiss and D'Mello [2015].

Definition 1. *A **language** $\mathcal{L}$ is a set of function, constant and relation symbols with associated arities.*

Definition 2. *A **term** in the language $\mathcal{L}$ is a finite sequence of function and constant symbols from $\mathcal{L}$ and variables that is defined recursively:*

1. *A variable is a term.*
2. *A constant is a term.*
3. *If f is n-ary function and $t_1, \dots t_n$ are terms then $f(t_1, \dots t_n)$ is also a term.*

Only sequences that can be obtained using this constructions are terms.

Definition 3. *A **Formula** in the language $\mathcal{L}$ is a finite sequence of any symbols in $\mathcal{L}$, logical operations $(\wedge, \vee, \neg, \rightarrow)$ and the equality symbol $=$. It is also defined recursively:*

1. *If t_1 and t_2 are terms then $t_1 = t_2$ is a formula.*
2. *If R is an n-ary relation symbol and $t_1, \dots t_n$ are terms then $R(t_1, \dots t_n)$ is a formula.*
3. *If φ is a formula then $\neg\varphi$ is also a formula.*
4. *If ϕ and ψ are formulas, then $\varphi \wedge \psi$, $\varphi \vee \psi$ and $\varphi \rightarrow \psi$ are also formulas.*

The definition usually also includes the quantifiers $\exists$ and $\forall$.

5. *If φ is a formula and x is a variable that is not already used with a quantifier then $(\exists x)\varphi$ and $(\forall x)\varphi$ are formulas.*

   $\forall$ is called the universal and $\exists$ is called the existential quantifier

*Once again, only sequences obtained by this construction are formulas. All subsequences of a formula that are also formulas are called **subformulas**.*

Definition 4. *Given a formula φ, a **free variable** is a variable in φ that is not used in a quantifier. A variable that is used in a quantifier is called **bound**. If $x_1, \dots x_n$ are all the free variables of φ then we usually write φ as $\varphi(x_1, \dots x_n)$ to show that it has free variables. A **sentence** is a formula that has no free variables. A **free formula** is a formula that has only free variables.*

Definition 5. *An **$\mathcal{L}$-theory** is a set of sentences in the language $\mathcal{L}$.*

Note that theory can be also infinite. Also note that the cardinality of a theory of a finite language is at most countable.

Definition 6. *A **structure $\mathcal{S}$** is a collection $(S, \mathcal{I})$ where S is a nonempty set (called the universe of $\mathcal{S}$) and $\mathcal{I}$ is an assignment that assigns interpretations to the elements of $\mathcal{L}$. Naturally it assigns function symbols to functions on S, constant symbols to elements of S and relation symbols to relations on S with their appropriate arities.*

Now we will need to know what does it mean for a sentence to be true in a structure. In order to do that we first need to know the values of terms.

Definition 7. *Let $t(v_1 \dots v_n)$ be an $\mathcal{L}$-term and $\phi : \{v_1, \dots v_n\} \longrightarrow S$ be an assignments of the variables to an $\mathcal{L}$-structure $\mathcal{S}$. Then we recursively assign values to subterms of t.*

1. *Value of v_i is $\phi(v_i)$*
2. *Value of c where c is a constant is $\mathcal{I}(c)$*
3. *If $t_1 \dots t_m$ are terms with assigned values $s_1 \dots s_m$ and f is a function then $f(t_1, \dots t_m)$ is assigned the value $\mathcal{I}(f)(s_1, \dots s_m)$.*

Definition 8. *Let $\varphi(x_1, \dots x_n)$ be an $\mathcal{L}$-formula and $\mathcal{S}$ an $\mathcal{L}$-structure. Then given a variable assignment $\phi : \{x_1, \dots x_n\} \longrightarrow S$ we assign the truth value of all subformulas of φ as follows:*

1. *If the subformula is of the form $t_1(x_1, \dots x_m) = t_2(x_1, \dots x_m)$ where t_1 and t_2 are terms, then it is true if and only if t_1 and t_2 have the same value assigned with variable assignment ϕ.*
2. *If the subformula is of the form $R(t_1, \dots t_r)$ where $t_1, \dots t_r$ are terms with assigned values $s_1, \dots s_r)$ then it is true if and only if $\mathcal{I}(R)(s_1, \dots s_r)$ holds in $\mathcal{S}$.*
3. *Any logical operators work as normal, e.g. $\neg\psi$ is true if and only if ψ is false or $\psi_1 \wedge \psi_2$ is true if and only if both ψ_1 and ψ_2 are true in this assignment.*
4. *If the subformula is of the form $(\forall y)\psi(y, x_1, \dots x_m)$ then it is true if and only if for any extension ϕ' of ϕ that includes also assigns y does $\psi(y, x_1, \dots x_m)$ hold (with assignment ϕ').*

   Alternatively if the subformula is of the form $(\exists y)\psi(y, x_1, \dots x_m)$ then we only require that there exists at least one such ϕ'.

Definition 9. *Given an $\mathcal{L}$-formula $\varphi(x_1, \cdots x_n)$ we say that φ **holds** in an $\mathcal{L}$-structure $\mathcal{S}$ if it is true for any variable assignment. We denote $\mathcal{S} \models \varphi(x_1, \ldots x_n)$. A formula φ is satisfiable if there exists a structure $\mathcal{S}$ in the same language where $\mathcal{S} \models \varphi$*

Note that this definition also works if φ is a sentence.

Definition 10. *Given an $\mathcal{L}$-theory T, we say that an $\mathcal{L}$-structure $\mathcal{S}$ is a **model** of T if $T \models \varphi$ for every $\varphi \in T$.*

Definition 11. *Formulas φ and ψ are **equivalent** if they are both true under the same variable assignments of their free variables in any of their models*[1].

This $\mathcal{S}$ is by no means unique. In fact, most theories have vastly different models.

Note that this definition does not include semantic differences, e.g. renaming of variables. For this we use a different definition:

Definition 12. *We say that $\mathcal{L}$-formulas φ and ψ are **equisatistfiable** if either both are satisfiable or both are not.*

The models of φ and ψ can be different, they might not even be in the same language. Every double of equivalent formulas is also equisatisfiable, since they share all models [2]. This term is used almost exclusively in formula manipulations.

Important things to notice in this section are that the universe S has no restrictions. Here we will use S as a subset of $\mathbb{R}^n$ in order to allow working with neural networks.

1.1.2 Skolemization

The existence of existence quantifiers has been a major hurdle for automatic theorem provers to overcome, since it adds a lot of complexity to the proving algorithm. For example if a formula starts with $\forall x \exists y \forall z$ then y can be completely different for each x, but does not depend on z. To "remember" this dependency in further proofs, the theorem prover needs to do some processing. We call this process **Skolemization** by it's inventor, Thoralf Skolem.

Definition 13. *We say that a formula φ is in **prenex normal form** if it is written as a sequence of quantifiers followed by a free formula.*

Theorem 1. *Every formula is equivalent to a formula in prenex normal form.*

Proof. We recursively apply the quantifier equivalence rules (we also assume that there exists at least one element):

- $(Qx\ \varphi) \wedge \psi$ is equivalent to $Qx\ (\varphi \wedge \psi)$ where Q is either $\exists$ or $\forall$.

[1] This definition is not the standard one, since we only define equivalence with respect to models. Normally equivalence of formulas is a semantic property, regardless of models or even satisfiability. However, to define equivalence properly, we would need to define multiple other terms from mathematical logic, which we will not do for better readability of this section. Classic definition can be found in Mendelson [1997].

[2] This is also true with the classical definition of equivalence, even though it does not require the formulas to be satisfiable.

- $(Qx\ \varphi) \vee \psi$ is equivalent to $Qx\ (\varphi \vee \psi)$ where Q is either $\exists$ or $\forall$.
- $\neg(\exists x\varphi)$ is equivalent to $\forall x\neg\varphi$
- $\neg(\forall x\varphi)$ is equivalent to $\exists x\neg\varphi$

If we treat implication $\varphi \to \psi$ as equivalent to $\neg\varphi \vee \psi$ we are done. □

Definition 14. *We say that a formula φ is in **Skolem normal form** if it is in prenex normal form and all quantifiers are universal.*

Skolemization is a process that turns formulas from prenex normal forms to Skolem normal form by introducing new function and constant symbols. We repeatedly apply this step:

Let φ be a formula that has the form

$$\forall x_1 \forall x_2, \ldots \forall x_i \exists y \psi(x_1, \ldots x_i, y)$$

where $x_1, \ldots x_n$ are all universally quantified and ψ is a formula in prenex normal form. Then Skolemized version of φ is obtained by introducing a new i-ary function symbol f_y and replacing all occurences of y in ψ with $f_y(x_1, \ldots, x_i)$. Thus we get

$$\forall x_1 \forall x_2, \ldots \forall x_i \psi(x_1, \ldots x_i, f_y(x_1, \ldots, x_i))$$

We say that f_y is the Skolem function of y.

If φ has the form $\exists y\psi(y)$, i.e. there are no universal quantifiers at the start we introduce a Skolem constant c_y and replace all occurences of y in ψ by c_y, obtaining $\psi(c_y)$. This is consistent with the notion of constants being 0-ary functions.

Theorem 2. *Every formula φ is equisatisfiable to the formula ϕ that is obtained by the Skolemization process.*

Proof. We need to prove that we can build a model of ϕ from a model of φ and vice versa. Let $\mathcal{S}$ be a model of φ and WLOG let φ be in prenex normal form $Q_1 x_1, \ldots Q_n x_n \psi(x_1, \ldots x_n)$ where Q-s are quantifiers and ψ is a free formula. We need to add interpretations of all Skolem functions and constants to build the model $\mathcal{S}' \models \phi$.

First, let us assume that y is the first existentially quantified variable in φ and $f_y(x_1, \ldots, x_i)$ its Skolem function (to include Skolem constants we permit $i = 0$). Since $\mathcal{S}$ is a model of φ, we know that for every $x_1, \ldots x_i \in S$ there exists an y that is the "witness" that φ holds. We define $f_y(x_1, \ldots x_i) = y$. Since there is such y for every tuple of x-es, this is a well-defined function. And we can easily see, $\psi(x_1, \ldots x_i, y, x_{i+2}, \ldots x_n)$ has the same truth value in $\mathcal{S}$ as $\psi(x_1, \ldots x_i, f_y(x_1, \ldots x_i), x_{i+2}, \ldots x_n)$ does in $\mathcal{S}'$, therefore $\mathcal{S}' \models \phi$. We repeat this process of defining Skolem functions for every step of the Skolemization process, and we get the whole $\mathcal{S}'$.

The other way around is just as easy. Since $\psi(x_1, \ldots x_i, f_y(x_1, \ldots x_i), x_{i+1}, \ldots, x_n)$ holds in $\mathcal{S}'$, there obviously exists an $y \in S$ that $\psi(x_1, \ldots x_i, y, x_{i+1}, \ldots, x_n)$, that being $y = f_y(x_1, \ldots x_i)$. □

This process had first been introduced to prove a general theorem in model theory, but we will use it to enable manipulation with existential quantifiers. More about the usage of this theorem can be found in Mendelson [1997] and Weiss and D'Mello [2015].

1.2 Neural networks

Neural networks have been one of the most important and well developed methods in machine learning in recent years. The main advantage of their usage is their universal property - the fact that given an $\epsilon > 0$ there exists for every smooth function a neural network that approximates it with the error of at most ϵ. Assuming that we have a structure whose universe is a subset of $\mathbb{R}^n$, we can approximate any of its functions with arbitrary precision. How exactly do we do that will be discussed in chapter 2. More about this subject can be found in Goodfellow et al. [2016]. This publication is also the principal source for this section.

1.2.1 What is a neural network

In this section we will describe a **feedforward neural network**. We call it feedforward because it lacks feedback connections. It is therefore simpler, although it does not have any "memory". If a network has feedback connections, it is called **recurrent**. They are also widely used in machine learning, they are not used here, thus are beyond the scope of this thesis.

A network is usually represented by chaining of functions, called **layers**. Thus if we have a network $f(x) = f_n(f_{n-1}(\dots f_1(x))\dots)$ we call f_1 the first layer, f_2 second and so on. The number of these layers is called **depth** of a model. The last layer is called **output layer**, while the rest are called **hidden layers** since we usually can not interpret what does the data they are using mean.

In general the output f_i does not need to have the same dimension as the input x or the output y. But for the sake of simplicity, all hidden layers usually tend to have the same dimension between them. This is called the **width** of the network. Both depth and width can have a profound effect on the network's performance.

These networks are called neural, because they are inspired by biological neurons. In each layer the singular neurons are scalar functions from each element of input to each element of the output. These elements are also commonly referred to as *nodes*. When we add together all neurons that lead to a node of the output, we get a vector to scalar function, that is the building block of a layer. This is better illustrated in Figure 1.1.

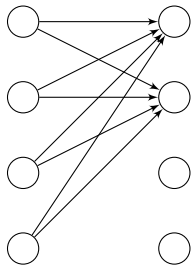
The simplest neural networks are linear models, e.g. linear regression. In these models each neuron is a simple multiplication by a parameter. These models however have many drawbacks, for instance that they can only model linear functions. That is because the layers are linear functions, therefore several layers in a sequence also form a linear function.

We overcome this by altering the layer input in a non-linear fashion, ideally preserving the most important parts of it. This is an important area of ongoing research.

One of the most widely used methods to break the linearity is the use of so-called activation functions. Activation function is a non-linear $\mathbb{R} \to \mathbb{R}$ function that we perform on each node to know if it "activates". There are several functions that fulfill this purpose, each with their advantages and disadvantages. The simplest is *ReLU*: the **Re**ctified **L**inear **U**nit (Nair and Hinton [2010], Jarrett

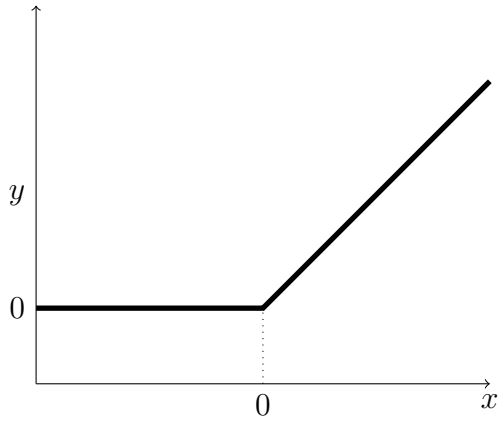
Figure 1.1: Neurons in a layer

Each arrow represents a single neuron-a scalar to scalar function



et al. [2009], Hahnloser et al. [2000]). It is defined as $ReLU(x) = max\{x, 0\}$. It is the most used activation function because it is the simplest function that also preserves most of what makes linear functions easy to work with (simple derivation). It has however some drawbacks that we will discuss later.

Figure 1.2: ReLU



1.2.2 Example: XOR

One of the simplest examples of the usefulness of this non-linearity is the binary operation XOR: the e**X**clusive **OR**. It is a function $\{0, 1\}^2 \to \{0, 1\}$ that is 1 if and only if exactly one of the inputs is 1. This function is impossible to approximate with linear regression, but there exists a simple network with $ReLU$ activation functions that exactly approximates it. This example network is the same as can be found in Goodfellow et al. [2016].

Let us start with linear regression. As the loss function we will use the most common one, mean square error. It has the form

$$J(\theta) = \frac{1}{n}\sum(\widehat{f}(\mathbf{x};\theta) - f(\mathbf{x}))^2$$

where f is the function we try to estimate and $\widehat{f}$ is the learned function. Since we are now using linear regression, $\widehat{f}$ has the form $\widehat{f}(\mathbf{x};\mathbf{w},b) = \mathbf{x}^\top\mathbf{w} + b$ where

$\mathbf{w}$ is called the weight vector and b is bias (in this case scalar). Using the least squares method we get $\mathbf{w} = (0, 0)^\top$ and $b = \frac{1}{2}$. This is obviously not a good fit as it just outputs $\frac{1}{2}$ regardless of the input.

If we however permit the usage of activation functions (ReLU), we can handcraft a network that will fit the function perfectly. Now our network will have the form

$$\widehat{f}(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^\top \max\{0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}\} + b$$

where

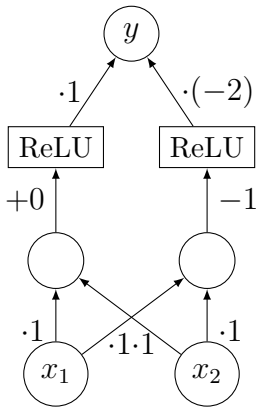
$$\mathbf{W} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix},$$

$$\mathbf{c} = \begin{pmatrix} 0 \\ -1 \end{pmatrix},$$

$$\mathbf{w} = \begin{pmatrix} 1 \\ -2 \end{pmatrix},$$

and $b = 0$. This network is illustrated in Figure 1.3. This is also not the only network that estimates XOR perfectly.

Figure 1.3: XOR network



Let us see what happens when we apply the input data to it in matrix form

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

After the first layer multiplication by the weight matrix $\mathbf{W}$ we have

$$\begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{pmatrix}.$$

When we add the bias vector $\mathbf{c}$ we get

$$\begin{pmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{pmatrix}.$$

Now we apply ReLU:

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{pmatrix}.$$

This concludes the first layer of the network. Then we multiply each row with $\mathbf{w}^\top$ and we get

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

which is exactly the desired output.

In this case we got the hand-crafted solution from a book. Normally the weights and biases are found using an optimization method such as gradient descent. Here we found the perfect solution, a global minimum to the loss function. But that is not generally possible.

1.2.3 Other activation functions

ReLU, while being the most popular activation function, is not the only one used here, because it also has some drawbacks. The fact that its gradient is 0 on all negative inputs means that gradient descent optimization methods might get us to the state where a node "dies" - all gradients regardless of data are 0. This tends to happen if our input and output contain few non-zero elements. In that case it is advised to use a different activation function.

Before the introduction of ReLU the default activation functions were sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

and hyperbolic tangent

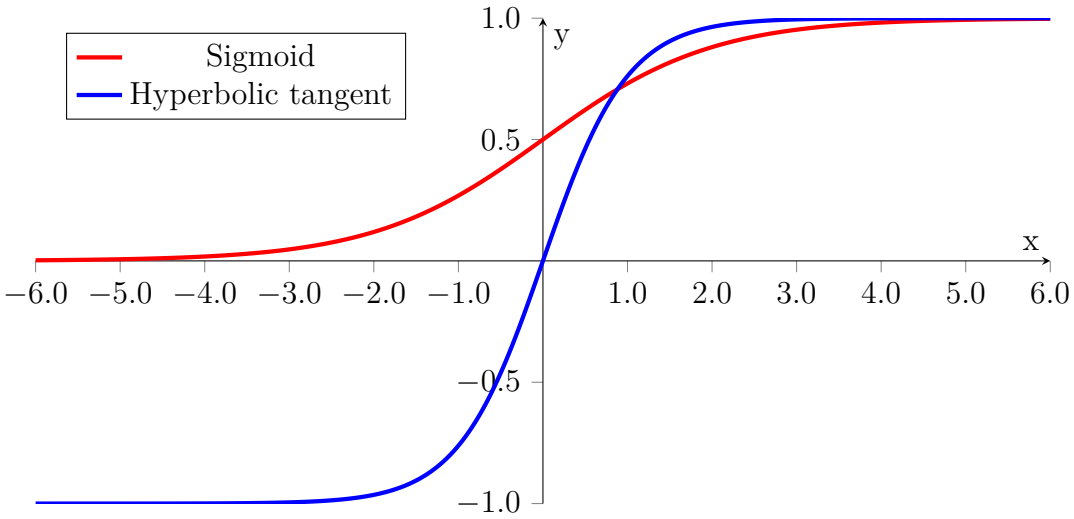
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

functions. They are related to each other since $\tanh(x) = 2\sigma(2x) - 1$. The main advantage of these functions is that they only output values in an open interval, $(0, 1)$ for sigmoid and $(-1, 1)$ for tanh. They also have nonzero gradient over their whole domain, which ensures that gradient descent optimization always works. However, as can be seen in Figure 1.4 if the inputs are far enough from 0, the gradients become very small. This results in gradient descent optimizers taking longer time (Krizhevsky et al. [2012]). If this happens, we say that the functions **saturate**. This is also the reason why these functions fell out of use.

There had been also a number of attempts to fix the "dead" node problem with ReLU. Most of them try to preserve the piecewise linearity, since it is the ReLU's strongest advantage. Among examples are Leaky ReLU (Maas et al. [2013]), Parametric ReLU (He et al. [2015]) or Exponetial LU (Clevert et al. [2015]). They all try to combat this problem by redefining ReLU on negative numbers. They can be seen in Figure 1.5

They are defined as follows:

Figure 1.4: Sigmoid and tanh functions



Leaky ReLU: $\text{LReLU}(x) = \max\{0.01 \cdot x, x\}$

Parametric ReLU: $\text{PReLU}(x) = \begin{cases} a \cdot x & x \leq 0 \\ x & x \geq 0 \end{cases}$ where a is a learned parameter

Exponential LU: $\text{ELU}(x) = \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x \geq 0 \end{cases}$ where α is a learned parameter

There is to this day ongoing search for better activation functions, for example in paper Ramachandran et al. [2018] they propose a different function called Swish, defined as $x \cdot \sigma(\beta \cdot x)$ where β is a learned parameter. They were able to find this function by crafting a set of basic functions and rules to combine them, thus greatly enlarging the set of activation functions tried. This Swish function is presented as a compromise between ReLU and the sigmoid:

> If $\beta = 0$, Swish becomes the scaled linear function $f(x) = \frac{x}{2}$. As $\beta \to \infty$, the sigmoid component approaches a $0 - 1$ function, so Swish becomes like the ReLU function. This suggests that Swish can be loosely viewed as a smooth function which nonlinearly interpolates between the linear function and the ReLU function.(p.5)

This paper also contains performance comparisons for all above mentioned activation functions.

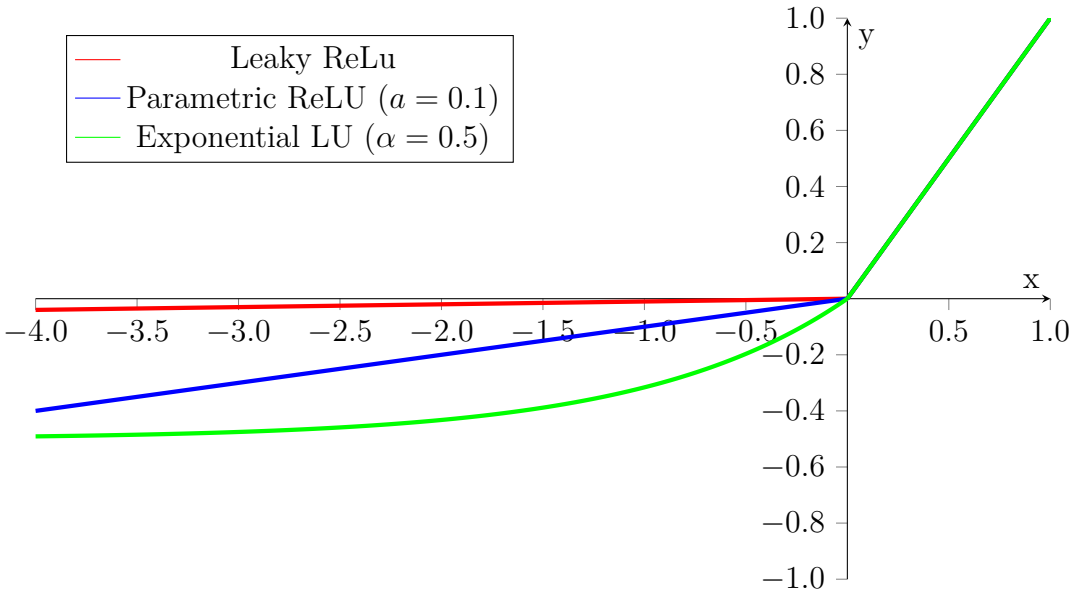
1.2.4 Learning and optimization

Mathematical optimization is a branch of applied mathematics that is about finding local extremes in functions. We use it to find parameters for our neural network.

To use it in practice we first define a function that enumerates how far we are from the desired output,

$$J : \Theta \to \mathbb{R}$$

Figure 1.5: ReLU variations



where Θ is the universe of all possible sets of parameters for our estimator. We will call J the **loss** function. The set of parameters that is the minimum of J is the set that leads to the best estimator. Using optimization on J we seek to find this minimum.

With neural networks we use the general form

$$J(\theta) = \frac{1}{|\mathbf{X}|} \sum_{\mathbf{x} \in \mathbf{X}} j(\widehat{f}(\mathbf{x}; \theta))$$

where j is the loss on a singular input. Note that the universe of inputs may be infinite. In those cases we only use a finite subset $\mathbf{X}$ in each of the optimization steps.

A very common loss function is the mean squared difference first introduced above. There $j = (\widehat{f}(\mathbf{x}; \theta) - f(\mathbf{x}))^2$. It is widely used because of its simplicity, but it also has some drawbacks, e.g. sensitivity to outliers. It was also used to optimize the networks built for this thesis.

After we have a loss function, we choose an optimization method. A very basic one is **gradient descent**. This method is commonly introduced as a solution to the foggy hill problem. In this problem we have a traveller that wants to reach the summit of a hill, but because of the fog he can only see his immediate vincinity. He therefore always goes in the direction of the steepest climb and if he can not see any climb, he declares that he had reached the summit. In this problem we seek the maximum height above sea level as a function of longitude and latitude.

Gradient descent is an algorithm that can be used on functions that are differentiable with respect to θ. It works in steps. It starts at a random $\theta_0 \in \Theta$. In each step t it computes the gradient ∇_t of $J(\theta_t)$ and then sets

$$\theta_{t+1} = \theta_t - \epsilon \cdot \nabla_t,$$

since $-\nabla_t$ is the direction of the steepest descent. ϵ is a parameter of the algorithm that describes how "long" the steps are. It is also called the **learning rate**. A careful consideration needs to go into the choice of this parameter. Too large ϵ can lead to the algorithm "overshooting" the minimum, while with a small ϵ, the finding of the minimum can be very slow.

Usually the universe the function operates on is infinite, or too large to use efficiently. In those cases we do the gradient descent in batches. As mentioned before, we take a sample subset $\mathbf{X}$ of the universe (called **batch**) and we do the descent step using that with the loss function J. It is important to note that the gradient would be different with every $\mathbf{X}$ we choose. That may lead (although rarely) to the situations where in one step we have the gradient that is opposite of the previous one, thus undoing the previous step. Another disadvantage of gradient descent is that it only finds local minima, so any small valley would stop the algorithm. To combat this, we usually make some alterations. One approach is discussed in subsection 1.2.6.

1.2.5 Back-propagation

Next problem in optimizing a neural network is computing the gradients in each step. Since the input dimension can be quite large, numerical computation of gradients, although possible, is usually slow. One of the fundamental properties of neural networks is the simplicity of their individual components that allows us to compute gradients more efficiently. The most widely used method is called **Back-propagation** (Rumelhart et al. [1986]) or **Backprop**.

This name comes from the inverse of the term **Forward-propagation** - usage of the feedforward network to compute an output. Input propagates forward through the network until it produces an output, and subsequently the loss $J(\theta)$. Back-propagation is taking this loss and propagating it backwards to produce a gradient.

At the heart of this method lies the *chain rule of calculus*. Let f and g be $\mathbb{R} \to \mathbb{R}$ differentiable functions and $z = f(y) = f(g(x))$. Then

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}.$$

This also holds in higher dimensions, where it is generalized to

$$\frac{\partial z}{\partial x_i} = \sum_{j=0}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

where $\frac{\partial z}{\partial y_j}$ are partial derivations of f in j-th coordinate and $\frac{\partial y_j}{\partial x_i}$ are partial derivations of g_j - the j-th coordinate of g with respect to its inputs i-th coordinate. Written with a Jacobian matrix J of g it is

$$\nabla_x z = J^\top \nabla_y z.$$

As we can see, if Jacobians of vector to vector functions and gradients of vector to scalar or scalar to scalar functions in our network are known, we can compute gradients with respect to any subset of inputs starting from any point

Algorithm 1 A basic backpropagation algorithm for the most basic feedforward neural networks. We assume that each layer of our network is an affine function $y^{(i)} = f^{(i)}(x^{(i-1)}) = W^{(i)}x^{(i-1)} + b^{(i)}$ with activation function a on each element: $x^{(i)} = a(y^{(i)})$. x-es here are states between layers and y-s are states before applying activation functions for nonlinearity. Here $x^{(0)} = y^{(0)} =$ input and $f^{(1)}$ is the first layer.

Require: $\widehat{y}, y$: Computed and expected result respectively. $\widehat{y} = x^{(n)}$
Require: $J(\widehat{y}, y)$: A loss function
Require: $\{x^{(i)}, y^{(i)}\}$: Computed values for all layers
Require: $\{W^{(i)}, b^{(i)}\}$: Network weights and biases
  $g \leftarrow \nabla_{\widehat{y}} J(\widehat{y}, y)$: Initialization of gradient with respect to the computed value
  for $i = n-1, \ldots, 0$ **do**
    $g \leftarrow \nabla_{y^{(i)}} J = g \odot a'(y^{(i)})$
    Undo the derivation of the activation function. For the sake of simplicity we assume that a has no parameters, but if it had we could save their gradients here.

    $\nabla_{b^{(i)}} J = g$
    $\nabla_{W^{(i)}} J = g{x^{(i)}}^\top$ //g is a column vector and ${x^{(i)}}^\top$ is a line
    Compute the gradients with respect to this layer's parameters

    $g \leftarrow \nabla_{x^{(i-1)}} J = {W^{(i)}}^\top g$
    Propagate the gradient to previous layer
  end for

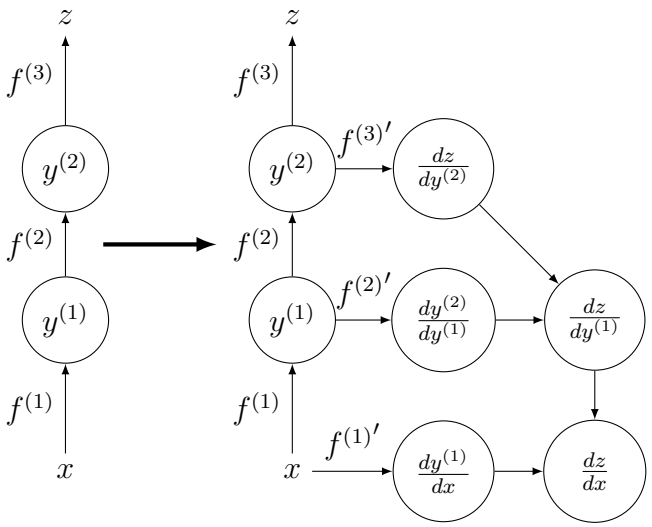
in our network. We can accomplish this by applying the chain rule recursively (and taking the rest of the inputs as constants).

However, this algorithm computes the gradient for only one input. If our batch size is higher, we compute all gradients and add them together. This can get computationally difficult, so most software uses procedures to speed it up. For example Tensorflow framework uses what Goodfellow et al. [2016] calls the symbol to symbol approach. The framework adds new nodes to the network that are used to compute the derivations. This way the backpropagation can be done by the same engine. Specifically we add nodes for gradients of each layer by itself and other nodes for computing chain rule. Then if we treat the batch as a matrix, we can compute gradients during only one run through the graph. Figure 1.6 illustrates this process.

1.2.6 Adam optimizer

In the framework built for this thesis we use Adam (**Ada**ptive **m**oment estimation) optimizer (Kingma and Ba). It is based on gradient descent, but it also conserves the momentum. In each learning step it uses a weighted average of previous gradients, thus it is not as dependent on the exact batch chosen for the step. It also helps overcome local minima, since it takes longer to reverse the direction of descent. It uses 4 parameters: α - learning rate, β_1 - gradient momentum decay, β_2 - second gradient moment momentum decay and ϵ - a small

Figure 1.6: An example of the symbol to symbol approach. Note that $f^{(3)}$ here can also be the loss function



parameter to avoid division by zero. The description of algorithm can be found in Algorithm 2.

The paper recommends using $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. These values are also the default values in Tensorflow framework and were also used for building the models for this thesis.

This optimizer is best suited for very noisy functions or functions with a lot of local minima. In the step 7 of the loop we update each parameter with a different step size, dependent on the second moment. The effective step size for each element is $\Delta_t = \alpha \cdot \widehat{m}_t / \sqrt{\widehat{v}_t}$. Therefore the step size gets higher if the space is sparse, i.e. it only has several large gradients. If the gradients are closer to each other, we get smaller steps. We generally expect there to be a wild variance when we are further from the optimum (due to noise) that diminishes the closer we get. This way Adam can tune its step size based on how close we estimate that we are.

In the 5^{th} and 6^{th} step of the loop we perform bias corrections. This is because the model without these corrections is naturally biased towards the initial value. As an example, let us assume that in the gradient g_1 has one element value 10. Then m_1 has for this element $10 \cdot (1 - \beta_1)$, that is 1 if we use recommended parameters. The average, however, should obviously be 10. If we do the correction, this is exactly what we have.

Another nice property of Adam is its invariance towards rescaling. If we rescale the gradients with a positive constant c, it cancels out: $\Delta_t = c \cdot \widehat{m}_t / \sqrt{\widehat{v}_t \cdot c^2} = \widehat{m}_t / \sqrt{\widehat{v}_t}$.

More information about this algorithm, including the convergence analysis, proof of convergence or extensions can be found in the original paper.

Algorithm 2 Adam optimizer. This algorithm is exactly like it can be found in the original paper. g_t^2 denotes element-wise square. All vector operations are elemet-wise. f_t represents the loss function f realized over the training batch in the step t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
  $v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
  $t_0 \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m_t} \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v_t} \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m_t} / (\sqrt{\widehat{v_t}} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

2. Implementation

3. Results

Conclusion

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/`. Software available from tensorflow.org.

Djork-Arn é Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning byexponential linear units (elus). 2015. arXiv preprint arXiv:1511.07289.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789), 2000.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers:surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on com-puter vision*, page 1026–1034, 2015.

Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? *2009 IEEE 12th International Conference on Computer Vision,*, 2009.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *Journal of the American Statistical Association*. URL `https://arxiv.org/pdf/1412.6980.pdf`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neuralnetworks. *Advances in Neural Information Processing Systems*, page 1097–1105, 2012.

Andrew L Maas, Awni Y Hannun, , and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. *International Conference on Machine Learning*, 30, 2013.

Elliot Mendelson. *Introduction to Mathematical Logic*. Fourth edition. Chapman & Hall, 1997. ISBN 0 412 80830 7.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. *International Conference on Machine Learning*, 2010.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions, 2018. URL https://openreview.net/forum?id=SkBYYyZRZ.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

William Weiss and Cherie D'Mello. *Fundamentals of Model Theory*. 2015. URL http://www.math.toronto.edu/weiss/model_theory.pdf.