FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Martin Smolík

Neural modelling of mathematical structures and their extensions

Department of Algebra

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

Prague 2019

I want to thank my supervisor for persuading me to continue studying.

Title: Neural modelling of mathematical structures and their extensions

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 and in particular neural networks. We start with a set of axioms that describe functions, constants and relations 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. The main focus in this thesis are groups. We train neural representations for cyclic (the simplest) and symmetric (the most complex) groups. Another part of this thesis are experiments with extending such trained 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

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 a given sentence will 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.

In this thesis, we do first experiments with trying to build such mental image, so that it can later be used to do these estimations. The ultimate aim is to have an oracle that guesses validity 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 mathematicians 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 rely on the machine learning, namely we rely the universal property of the neural networks - their ability to approximate any continuous function with arbitrary precision. Neural network learning, however, needs to work with differentiable functions, that do not exist in most structures. To enable their usage we choose a representation of the structure elements in $\mathbb{R}^n$, which we will call *grounding*. Here we will use exclusively handpicked groundings that give us better insight into the performance of the model.

We try to teach the networks using the propositions that are true/false in the structures. The details can be found in Chapter 2. Using this approach with the framework Tensorflow (Abadi et al. [2015]) we have built neural models and extensions of groups, namely $\mathbb{Z}_n$ and S_n. Results of these experiments can be found in Chapter 3. However, we have so far not tried any experimentation on other structures. Experiments with other structures are possible, however beyond the scope of this thesis.

1. Background

In this section we will discuss the fundamentals of both model theory and neural networks. Since this thesis seeks to unify two vastly different fields of mathematics and computer science, understanding of both of them is essential. A reader acquainted with those fields should not find any surprises here.

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[1].

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 the universe of 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 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'.

[1]Assuming either countable set of variables, or taking the formulas "up to renaming of variables".

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. This is denoted as $\mathcal{S} \models \varphi(x_1, \ldots x_n)$. 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 $\varphi \in T$.*

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

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[2].*

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 pair of equivalent formulas is also equisatisfiable, since they share all models [3]. This term is used almost exclusively in formula manipulations.

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

1.1.2 Skolemization

Existential quantifiers have been a major hurdle for automatic theorem provers, since they add a lot of complexity to the proving algorithms. 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 its 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):

[2]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].

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

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

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

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

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

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 occurrences 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 $Q_1 x_1, \dots Q_n x_n \psi(x_1, \dots x_n)$ 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 $f_y(x_1, \dots, x_i)$ its Skolem function (to include Skolem constants we permit $i = 0$). Since $\mathcal{S}$ is a model of φ, we know that for every $x_1, \dots x_i \in S$ there exists an y that is the "witness" that φ holds. We define $f_y(x_1, \dots x_i) = y$. Since there is such y for every tuple of x-es, this is a well-defined function. And we can easily see, $\psi(x_1, \dots x_i, y, x_{i+2}, \dots x_n)$ has the same truth value in $\mathcal{S}$ as $\psi(x_1, \dots x_i, f_y(x_1, \dots x_i), x_{i+2}, \dots x_n)$ 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 $\psi(x_1, \dots x_i, f_y(x_1, \dots x_i), x_{i+1}, \dots, x_n)$ holds in $\mathcal{S}'$, there obviously exists an $y \in S$ that $\psi(x_1, \dots x_i, y, x_{i+1}, \dots, x_n)$, that being $y = f_y(x_1, \dots x_i)$. □

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].

1.1.3 Substructures and extensions

In order to do any constructions in model theory, we need to know how models relate to each other. As it is right now, our models exist completely separately from each other, with the only comparisons being possible through the lens of formulas. However, if two structures are models of the same theory, we can not differentiate between them using just that theory. That is why we need the notions of substructures and extensions.

Definition 15. *Let $\mathcal{S}, \mathcal{T}$ be structures in the same language $\mathcal{L}$. Then a **mapping** of $\mathcal{S}$ to $\mathcal{T}$ is any function $f : S \to T$ that satisfies the following (for any $x_1, .., x_n \in S$):*

1. *If F is an n-ary function in $\mathcal{L}$ then $f(F_{\mathcal{S}}(x_1, \ldots, x_n)) = F_{\mathcal{T}}(f(x_1), \ldots, f(x_n))$.*
2. *If c is a constant in $\mathcal{L}$ then $f(c_{\mathcal{S}}) = c_{\mathcal{T}}$.*
3. *If R is an n-ary relation in $\mathcal{L}$ then $R_{\mathcal{S}}(x_1, \ldots, x_n)$ holds if and only if $R_{\mathcal{T}}(f(x_1), \ldots, f(x_n))$ holds.*

*If it also holds that f is one-to-one, then f is called an **embedding**.*

Definition 16. *We say that a structure $\mathcal{S}$ is a **substructure** of $\mathcal{S}'$ if they are in the same language, $S \subseteq S'$ and all symbol interpretations agree on S. That means that $(S, \mathcal{I}|_S)$ is a structure. Here $\mathcal{I}|_S$ means the assignments of symbols is the same but restricted to S.*

*We also say that $\mathcal{S}'$ is an **extension** of $\mathcal{S}$*

There are not many things that hold in extensions for general. For example a carthesian product of two structures with naturally defined functions, constants and relations is an extension of both of them, but their elements do not necessarily "interact" with each other. To ensure this interaction we need new definitions.

Definition 17. *Let $\mathcal{S}$ be an $\mathcal{L}$-structure and $\mathcal{S}'$ its extension. Let $t_1(x)$ and $t_2(x)$ be $\mathcal{L}'$-terms with only one variable where $\mathcal{L}'$ is the language $\mathcal{L}$ with added names for each element of S. We say that the predicate[4] $\varphi(x)$: "$t_1(x) = t_2(x)$" is an **equation** in $\mathcal{S}$. If $\varphi(x)$ holds for some $x \in S'$, we say that x is the **solution** of φ. If the if S is not a subset of the set of solutions of φ, we say that the equation is **non-zero**.*

Definition 18. *Let $\mathcal{S}$ be a structure, $\mathcal{S}'$ its extension and let $s \in S'$ be an element. If there exists a non-zero equation $\varphi(x)$ in $\mathcal{S}$ for which s is a solution then s is **algebraic**[5]. If there is no such equation, then s is **transcendental**.*

[4]Predicate is a formula with only one free variable

[5]Normally we do not require the φ to be an equation, but for the purposes of our model we will make this assumption

We see that every element $s \in S$ is algebraic. We can take $x = s$ as the equation.

A reader acquainted with field extensions will recognize these terms. Indeed, these are generalizations of those terms, so that the theory can be modified and applied to other structures than fields. A crucial difference is that this definition does not guarantee the existence of an extension. An example of this is $0 \cdot x = 1 \cdot x$ in any field. However, there are classes of structures where every equation yields an algebraic extension. One such example are groups.

1.1.4 Groups and their algebraic extensions

Definition 19. *A **group** is a structure in the language $\{\cdot, ^{-1}, e\}$ (where $\cdot$ is binary function, $^{-1}$ is unary and e is a constant) that also models the following theory*[6]

1. $\forall a, b, c \; (a \cdot b) \cdot c = a \cdot (b \cdot c)$
2. $\forall a \; a \cdot e = e \cdot a = a$
3. $\forall a \; a \cdot a^{-1} = a^{-1} a = e$

*We will call $\cdot$ **composition**, $^{-1}$ **inverse** and e shall be **unit**.*

Note that this is the skolemized version of the theory that has $\exists e \forall a \; e \cdot a = a \cdot e = a$ or $\exists e(\forall a \; e \cdot a = a \cdot e = a \wedge \forall a \exists b \; a \cdot b = b \cdot a = e)$[7].

Definition 20. *A substructure of a group is a **subgroup** (denoted $G \leq G'$). We say that a subgroup is **normal** if $\forall x \in G' \; \forall g \in G \; x^{-1} \cdot g \cdot x \in G$, or simply $x^{-1} G x \leq G$. We denote it as $G \trianglelefteq G'$.*

It can be proven that this subgroup is of the form $\{t(b_1, \dots b_n) | b_1, \dots b_n \in B, t \text{ is a term}\}$. It is also the intersection of all groups that contain B (Drápal [2000]). Note that if we use this as a definition, we can expand the notion of a generated substructure to any model.

Definition 21. *Let G be a group and B a set of elements of G. Then a subgroup **generated** by B is the smallest subgroup of G that contains B. We denote it as $\langle B \rangle_G$.*

Definition 22. *Let $G \leq H$ be two groups. Then the sets $hG = \{h \cdot g | g \in G\}$ for any $h \in H$ are called **left cosets** of G. If G is a normal subgroup, then we can induce a composition on these cosets: $hG \cdot h'G = (h \cdot h')G$ (a proof that this is really a group can be found in Drápal [2000]). We will call this group the **quotient group** H/G.*

As with algebraic extensions over fields, we can use this quotient group to craft an algebraic extension to any equation.

First, however, we need a group that we can do quotient of.

Definition 23. *Let G be a group. Then we will define an extension $G[x]$ as such:*

[6]For the sake of simplicity we will use multiple = signs in the definitions. To be absolutely correct, we could re-write these as $a = b = c \longrightarrow a = b \wedge b = c \wedge c = a$.

[7]Once again we use simplified notation.

- *Elements of $G[x]$ are formal strings of the form $g_1x^{n_1}g_2x^{n_2}\dots g_kx^{n_k}g_{k+1}$ where k can be any integer (including 0-in that case we have an element of g), g_i are elements of G (all except g_1 and g_{k+1} have to be different from e) and n_i are non-zero integers.*
- *$\cdot$ is defined as concatenation of strings with adjustment for inverses. We do this adjustment by repeating the following:*

  1. *If we there is a substring g_ig_{i+1} we replace it with $(g_i \cdot g_{i+1})$.*
  2. *If there is a substring $x^{n_{i-1}}ex^{n_i}$ we replace it by $x^{n_{i-1}+n_i}$.*
  3. *If there is x^0, we replace it by e.*

  If none of these replacements is possible, we have a valid element.
- *e is the same as in G.*
- *If $g = g_1x^{n_1}g_2x^{n_2}\dots g_kx^{n_k}g_{k+1}$ then $g^{-1} = g_{k+1}^{-1}x^{-n_k}\dots x^{-n_1}g_1^{-1}$. We can easily verify that $g \cdot g^{-1}$ is indeed equal to e.*

Without loss of generality we can assume that any equation φ has the form $t(x) = e$, where t is a term. If $\varphi(x)$: "$t_1(x) = t_2(x)$" would have a different form, then $\varphi'(x)$: "$t_1(x) \cdot (t_2(x))^{-1} = e$" has exactly the same solutions.

Theorem 3. *Let G be a group and $\varphi(x)$ an equation of the form $t(x) = e$ where $t(x)$ has at least one occurrence of x. Then there exists an extension G' of G such that there exists $g \in G'$ that is a solution to φ.*

Proof. If we apply simplifications from definition 23, t has the form $g_1x^{n_1}g_2x^{n_2}\dots g_kx^{n_k}g_{k+1}$. This modification of φ still has the same solutions.

This way t can be seen as an element of $G[x]$. Let us then define $B = \{gtg^{-1} | g \in G[x]\}$. $\langle B\rangle$ is a normal subgroup of $G[x]$. We immediately see that $gbg^{-1} \in \langle B\rangle$, (even $gbg^{-1} \in B$) for all $b \in B, g \in G[x]$. Now if $b = b_1b_2\dots b_n$ where $b_i \in B$ then

$$gbg^{-1} = gb_1b_2\dots b_ng^{-1} = gb_1g^{-1}gb_2g^{-1}\dots gb_ng^{-1}.$$

We define $b_i{}' = gb_ig^{-1}$ for $i \in \{1, \dots, n\}$. Now

$$gbg^{-1} = (gb_1g^{-1})(gb_2g^{-1})\dots(gb_ng^{-1}) = b_1{}'b_2{}'\dots b_n{}'.$$

Since $\forall i\ b_i{}' \in B$, also $gbg^{-1} \in \langle B\rangle$.

Now we know that $G[x]/\langle B\rangle$ is a group. In it, all elements of G form different cosets, since $g\langle B\rangle$ always has the element $g \cdot t(x)$ and no two $g \in G$ can produce the same $g \cdot t(x)$. Also, since $t(x)$ has at least one occurrence of x, we know that $g \notin \langle B\rangle\ \forall g \in G$. Therefore $G \leq G[x]/\langle B\rangle$.

The last thing we need to show is that $x\langle B\rangle$ is a solution to $\varphi(x)$. Indeed, since $t(x) \in \langle B\rangle$,

$$t(x\langle B\rangle) = t(x)\langle B\rangle = \langle B\rangle = e_{G[x]/\langle B\rangle}.$$

Therefore $G[x]/\langle B\rangle$ is the desired extension. □

A reader acquainted with field extensions will immediately recognize this construction. Indeed, it had been the inspiration for this proof. Worth noticing is that this construction can also naturally lead to the notions of Galois groups and indices of group extensions, but that is beyond the scope of this thesis.

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 [8]. 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**. Such networks are also widely used in machine learning, however they are not used here, and are beyond the scope of this thesis.

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

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.

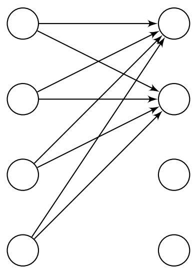
We overcome this by altering the layer input in a non-linear fashion, ideally preserving the most important parts of it. This is an important area of ongoing research.

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

[8] Assuming a suitable extension of the functions to facilitate the smoothness.

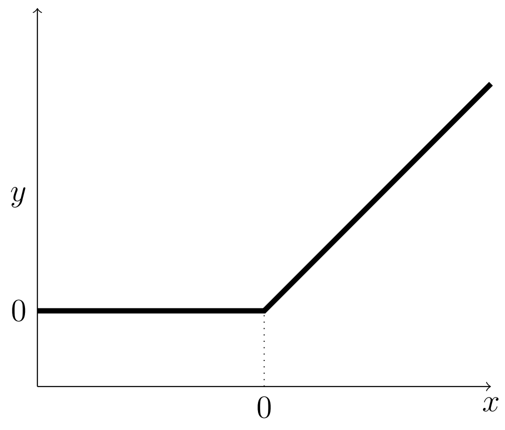
Figure 1.1: Neurons in a layer

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



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 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[9] we will use the most

[9]Loss function and its notation with parameters θ is introduced in Subsection 1.2.5.

common one, the 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 the *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

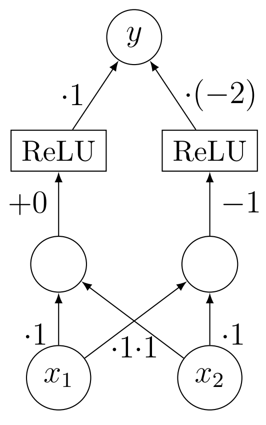
$$\mathbf{W} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix},$$

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

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

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 piece-wise linearity, since it is the

Figure 1.4: Sigmoid and tanh functions

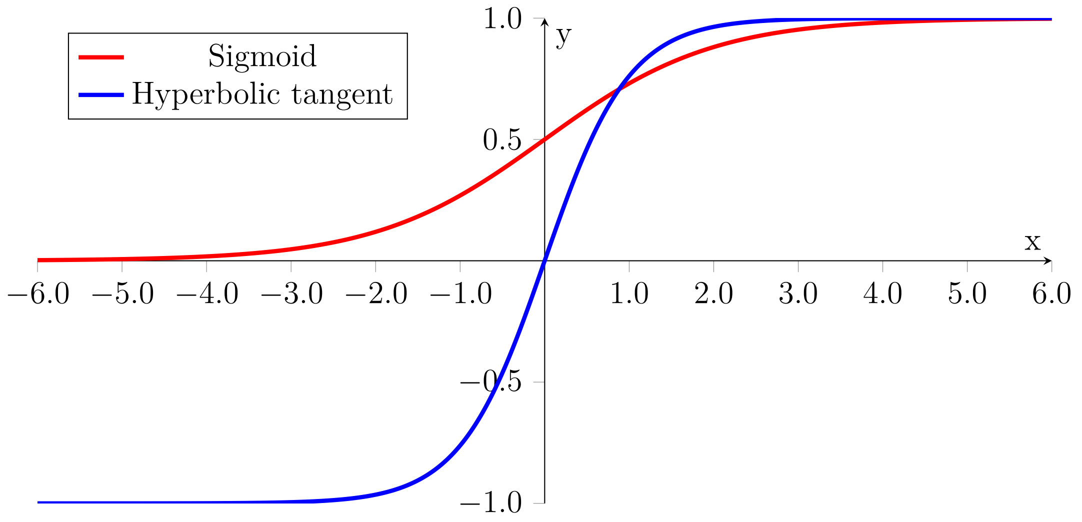
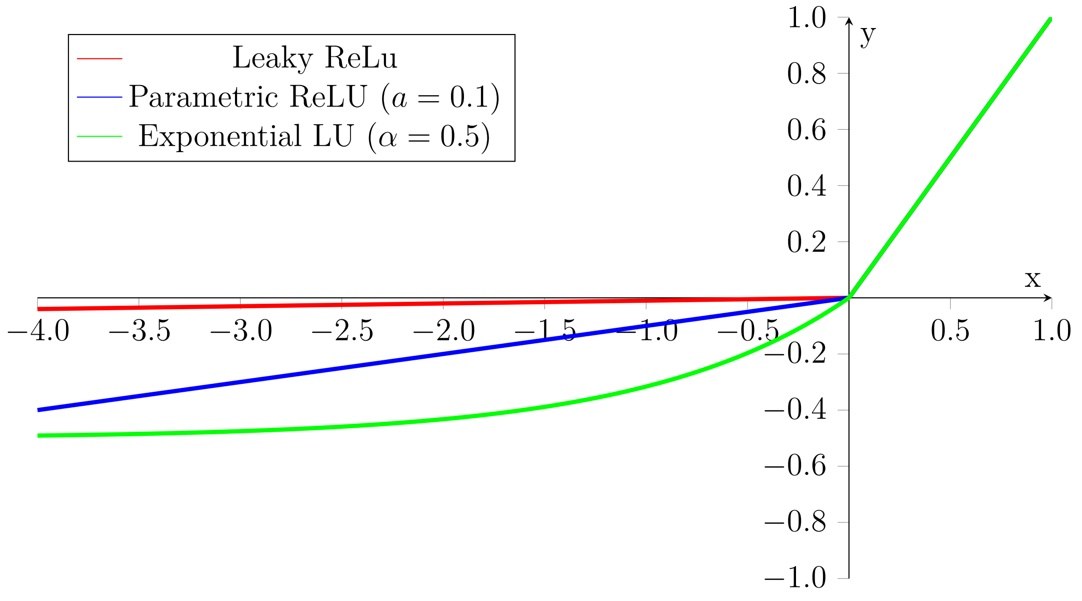


Figure 1.5: ReLU variations



ReLU's strongest advantage. Among examples are Leaky ReLU (Maas et al. [2013]), Parametric ReLU (He et al. [2015]) or Exponential 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:

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

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

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

There is to this day ongoing search for better activation functions, for example in 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. The 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 non-linearly interpolates between the linear function and the ReLU function.(p.5)

Ramachandran et al. [2018] also contains performance comparisons for all above mentioned activation functions.

1.2.4 Universal approximation

The most useful property of the neural networks is their ability to approximate any continuous function (Cybenko [1989]). The universal approximation theorem is stated thusly:

Definition 24. *Let I_n be the unit cube $[0,1]^n$ and μ a measure on I_n. Then $\sigma : \mathbb{R} \to \mathbb{R}$ is **discriminatory** if for any $\theta \in \mathbb{R}$ and $y \in I_n$*

$$\int_{I_n} \sigma(y^\top x + \theta) d\mu(x) = 0$$

if and only if $\mu = 0$.

Theorem 4. *Let σ be a discriminatory function. Then the finite sums of the form*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^\top x + \theta_j)$$

are dense in $C(I_n)$ (space of all continuous function on I_n).

This means that for any $f \in C(I_n)$ and $\epsilon > 0$ there is a sum $G(x)$ of the form as above such that $\forall x \in I_n$ $|G(x) - f(x)| < \epsilon$.

The proof of this theorem, along with the proof that sigmoid-like functions are discriminatory can be found in the original paper.

There is however no bound on the width of the network, which may pose a problem for the computers. Fortunately, however, Lu et al. [2017] prove that using a ReLU activation function we only need the width $n+4$ to achieve universal approximation on compact subsets of $\mathbb{R}^n$. On the flipside, this width-bound also erases the depth-boundedness seen in the theorem above.

The relation between depth and width had not yet been fully established. Partial results show that there are networks whose decrease in depth would require an exponential increase in width in order to keep the same accuracy (Eldan and Shamir [2016]). This leads to preference of deep neural networks, rather than wide. The other way - decreasing width and increasing length - had not yet been fully explored, but Lu et al. [2017] also prove that this increase in some cases has to be more than polynomial.

This universal approximation is the reason that the usage of neural networks is possible in model building. For example if $S \subset \mathbb{R}^{\ltimes}$ is the universe of a model and $f : S^m \to S$ is a function, we can approximate it by first introducing an arbitrary smooth function $f' : \mathbb{R}^{nm} \to \mathbb{R}^n$ such that $f'|_S = f$, and then finding a neural network that approximates f'.

1.2.5 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,

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

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 traveler that wants to reach the summit of a hill, but because of the fog he can only see his immediate vicinity. 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 1.2.7.

1.2.6 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 back-propagation 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 non-linearity. Here $x^{(0)} = y^{(0)} =$ input and $f^{(1)}$ is the first layer.

```
Require: $\hat{y}, y$: Computed and expected result respectively. $\hat{y} = x^{(n)}$
Require: $J(\hat{y}, y)$: A loss function
Require: $\{x^{(i)}, y^{(i)}\}$: Computed values for all layers
Require: $\{W^{(i)}, b^{(i)}\}$: Network weights and biases
  $g \leftarrow \nabla_{\hat{y}} J(\hat{y}, y)$: Initialization of gradient with respect to the computed value
  for $i = n-1, \dots, 0$ 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
```

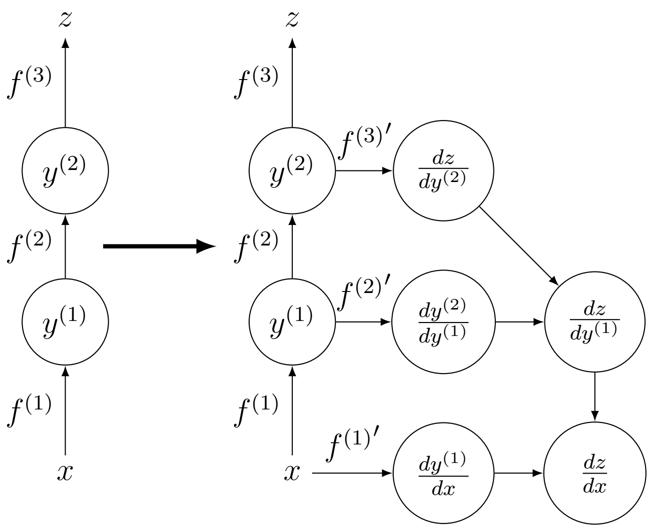
in our network. We can accomplish this by applying the chain rule recursively (and taking the rest of the inputs as constants).

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 the 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 back-propagation can be done by the same engine. Specifically we add nodes for gradients of each layer and other nodes for computing with the 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.

1.2.7 Adam optimizer

In the framework built for this thesis we use the 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 5th and 6th 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 element-wise. f_t represents the loss function f realized over the training batch in the step t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t_0 \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ 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 $\theta_t$ (Resulting parameters)
```

2. Neural modelling

This thesis is inspired by the work of Serafini and d'Avila Garcez [2016]. In their paper they propose the creation of *Real Logic*, a framework that uses tensor networks [1] to process sentences and assign them truth value in the interval $[0, 1]$ (we call this "uncertainty" **fuzzy logic**). They propose to do this by introducing a **grounding**: a function that *grounds* a structure in real vector space, including all elements, functions, constants and even relations. This grounding is approximated using neural networks.

The place where this thesis differs from this work is the type of the structure we try to emulate. In the original paper they model a relational structure, while we forego the relations completely and only implement functions and constants.

Definition 25. *Let $\mathcal{L}$ be a language. Then a* ***grounding*** *is a function $\mathcal{G}$ that satisfies:*

1. *$\mathcal{G}(c) \in \mathbb{R}^n$ for every constant c.*
2. *$\mathcal{G}(f)$ where f is a function symbol is a map $\mathbb{R}^{nm} \to \mathbb{R}^n$ where m is the arity of f.*
3. *$\mathcal{G}(r)$ where r is a relation symbol is a function $\mathbb{R}^{nm} \to [0, 1]$ where m is the arity of r.*

Grounding is then naturally extended to any term and atomic formula as such:

$$\mathcal{G}(f(t_1, \ldots, t_n)) = \mathcal{G}(f)(\mathcal{G}(t_1), \ldots \mathcal{G}(t_n))$$

$$\mathcal{G}(r(t_1, \ldots, t_n)) = \mathcal{G}(r)(\mathcal{G}(t_1), \ldots \mathcal{G}(t_n))$$

There are also rules for logic on formulas:

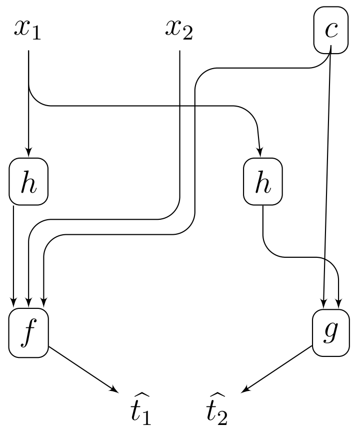
$$\mathcal{G}(\neg r(t_1, \ldots, t_n)) = 1 - \mathcal{G}(r(t_1, \ldots, t_n))$$
$$\mathcal{G}(\phi_1, \vee \cdots \vee \phi_n) = \mu(\mathcal{G}(\phi_1), \ldots, \mathcal{G}(\phi_n))$$

Where μ is a co-t-norm: an operator that joins the "probabilities" of the sentences being true into one. A t-form (or a triangle form) is a function in fuzzy logic that replaces conjunction, e.g. $\top_{\text{min}}(a, b) = min\{a, b\}$ or $\top_{\text{Luk}}(a, b) = \max\{0, a + b - 1\}$. A co-t-form is a dual that does the same with disjunction ($\mu(a, b) = 1 - \top(1 - a, 1 - b)$ because $a \vee b = \neg(\neg a \wedge \neg b)$). For more about fuzzy logic see Pelletier [2000].

In this thesis, however, we will be dealing with specific structures. Therefore we also define $\mathcal{G}(x)$ for each element of the structure. The rules for extending to terms still have to hold.

[1] A type of neural networks

Figure 2.1: An example network obtained from a literal $f(h(x_1), x_2, c) = g(c, h(x_1))$ where x_1 and x_2 are inputs, f, g, h are functions and c is a constant. Loss is computed from the difference of $\widehat{t_1}$ and $\widehat{t_2}$.



2.1 Building a neural model

First we will discuss the idea behind the general neural models. However, because of the limited scope of the experiments not every concept introduced in this section was implemented. To see what was done in practice, refer to Section 2.3 and Chapter 3.

Let T be an $\mathcal{L}$-theory that describes a class of structures. We assume that T is finite. We also assume that all $\varphi \in T$ are in Skolem normal form. If not, we use the skolemized version of T. We also assume that $\mathcal{L}$ has no relation symbols. How these could be implemented is discussed in Chapter 4.

We pick a grounding for the elements. We take an appropriate subset of $\mathbb{R}^n$ to represent the elements. Here we assume that the grounding is handpicked, i.e. it is given externally. Other possibilities are also discussed in Chapter 4.

Next we build a neural network for each of the symbols following the rules of groundings (see Definition 25). They are initialized randomly. Then we take these neural networks as building blocks and we build a network for each axiom of T, as illustrated in Figure 2.1. Since we assume that there are no relation symbols, each axiom has to have at least one equality sign. If the axiom is an equality between two terms, we use this as the basis for our loss function.

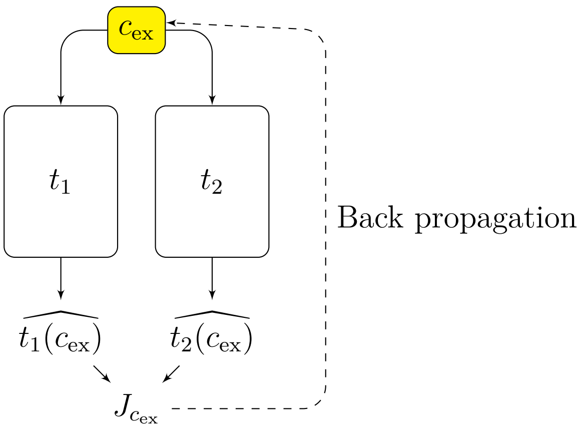
In the case that the axiom has more such equalities (or negations of them) we do some adjustments. Because we do not assign a truth value (0-1) to the atomic formulas, the methods of aggreggating subformulas in Definition 25 could prove problematic. Therefore we will need different methods to combine the mean squared differences.

The loss function of a negation of a subformula φ should be computable from its own loss, e.g.

$$J_{\neg\varphi} = \frac{1}{J_\varphi}.$$

If φ_1, φ_2 are subformulas, loss of $\varphi_1 \vee \varphi_2$ will be e.g. $\min\{J_{\varphi_1}, J_{\varphi_2}\}$ and loss of $\varphi_1 \wedge \varphi_2$ will be e.g. $J_{\varphi_1} + J_{\varphi_2}$. However, these operations are for now just a speculation and further research is warranted. For example $\vee$ could use a function with a gradient that depends on both parts of the equation, while $\wedge$

Figure 2.2: Network used for learning an extension defined by the equation $t_1(x) = t_2(x)$. Note that only c_{ex} is optimized, everything else is treated as a function.


c_{ex}
t_1
t_2
Back propagation
$\widehat{t_1(c_{\text{ex}})}$
$\widehat{t_2(c_{\text{ex}})}$
$J_{c_{\text{ex}}}$

might perform better using a different method of aggregation (e.g. $2\sqrt{J_{\varphi_1}^2 + J_{\varphi_2}^2}$).

It may be a good idea to also penalize straying from the groundings of the elements, since this may produce functions that are not exactly $S \to S$. Possible ways to achieve this are discussed in Chapter 4.

After computing the loss for each axiom, we aggregate them like with $\wedge$ and use back-propagation from there. This is, however, very costly with regards to computation time. Therefore in our experiments we used a different, simplified approach (see Section 2.3).

2.2 Neural model extension

Another part of this thesis is learning model extensions. In Subsection 1.1.3 we have discussed which type of extensions are we interested in.

Assume we have already built a model $\mathcal{G}$ for a structure $\mathcal{S}$ and we want to extend it to include a solution of an equation $t_1(x) = t_2(x)$[2]. Note that t_1 and t_2 are allowed to have parameters from $\mathcal{S}$. Since we assume that the representation is already settled, the usage of parameters is not a problem. We will treat the equation as an axiom $\exists x\ t_1(x) = t_2(x)$. In its skolemized version it is $t_1(c_{\text{ex}}) = t_2(c_{textex})$. We already have a framework to find the grounding of such a constant (Figure 2.2). However, we need to remove any possible penalties for straying away from the representations of S.

During the learning of this constant, it is vital that the optimization process does not change any parameters of other functions and constants. Otherwise we would not get an extension of the already built structure, only its modification.

After the optimizer found a minimum, we evaluate various terms with this new constant to test if it behaves as expected.

[2]We assume that it has no solution in S.

2.3 Implementation for groups

Axioms of groups can be used differently, since the definitions of symbols stem from one another. e is defined from $\cdot$, and $^{-1}$ is defined from e and $\cdot$. This is because we can regard e and $^{-1}$ as Skolem symbols (Subsection 1.1.4). However, this ordering of symbols is not generally possible.

There are many different types of groups, that have vastly different complexity levels. We have built models for the cyclic groups (namely $\mathbb{Z}_{10}$ and $\mathbb{Z}_{20}$) and symmetric groups (S_3 and S_4). These groups have been selected for being the simplest and the most complex finite groups respectively.

To better see if the computer is able to represent these groups correctly, we have opted to use the multiplication table for the first experiments with the learning of $\cdot$. The multiplication table is a table of pre-computed values for each pair of the elements. This is a simplification of learning the model only based on some propositions valid in it, but it is an essential one for human interpretability of the results. To demonstrate the ability of the network to generalize, and to a degree ”understand” the composition function, several entries (up to 10%) had been selected as testing data - they are never used during the optimization process.

Both $\cdot$ and $^{-1}$ are 4-layer feed-forward neural networks with all connections. The width of each layer is $3n$ where n is the size of the grounding. The activation functions on each layer are leaky ReLU. The loss function for each network is the mean squared difference. There is no penalty for straying too far from the element representations, since this is already covered by the multiplication table approach. Each network is optimized using the Adam optimizer with default settings. Inputs are given in batches of 25 random pairs for $\cdot$ and 5 elements for $^{-1}$ and e.

Each of the group functions is trained with a different network, with a separate optimizer. Because of the difficulty of determining whether a network is trained well, all of them were trained at the same time. The network used for learning the composition is described in Figure 2.3, unit in Figure 2.4 and inverse in Figure 2.5.

Note that e had been used during the training of composition as a part of the training data. However, this knowledge had not been passed to the optimizer seeking to find this e.

Figure 2.3: Learning of composition. An incomplete multiplication table is used to determine the loss. x_1, x_2 is a randomly selected pair.

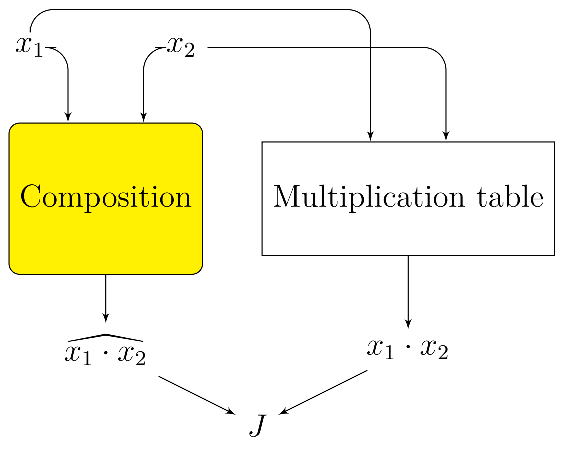


Figure 2.4: Learning of the unit. Parameters in the composition network are not altered in this optimization process. Note that the unit element is also present in the multiplication table used for the learning of the composition. However, this information is not shared, and the optimizer has to find it by itself. Learning is based on the axiom $\forall a \; a \cdot e = a$. The dual axiom $e \cdot a$ had been disregarded for the sake of efficiency.

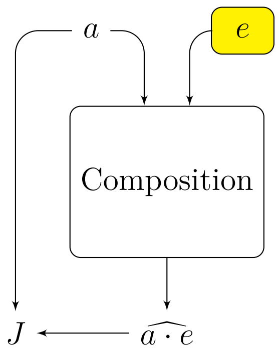
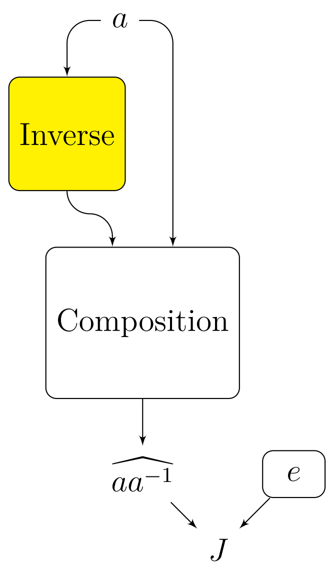


Figure 2.5: Learning of the inverse. Here we use the learned unit and composition. We use the axiom $a^{-1}a = e$. The dual $aa^{-1} = e$ is disregarded.



3. Results

In this chapter we will discuss the results of our experiments with groups. Note that the error rate depicted in the graphs is *not* the loss, but rather its square root. This is done as to represent errors on the original scale.

3.1 Cyclic groups

Cyclic groups are the simplest groups possible. They are defined as groups generated by only one element x. They are all isomorphic to either $(\mathbb{Z}, +, -, 0)$ - integers with addition - or $(\mathbb{Z}_n, +, -, 0)$ - $\{0, 1, 2, \ldots n-1\}$ with addition modulo n.

3.1.1 $\mathbb{Z}_{10}$

The group selected first was $\mathbb{Z}_{10}$. All elements were represented as themselves in $\mathbb{R}^1$. Since there is no element a such that $a + a = 1$, this was selected as the equation through which we will extend this group. Expected extension is $\mathbb{Z}_{20}$, where the embedding is $a = 1$, therefore $1_{\mathbb{Z}_{10}} \mapsto 2_{\mathbb{Z}_{20}}$. We will call this a the *half*.

The results of one experiment with training the composition, inverse and the unit are depicted in Figure 3.1. As we can see, we can get a very good approximation of composition and the unit. The inverse lags behind a little, but that is expected because it is dependent on both of the other functions, therefore the errors in them reflect in the inverse much more.

The inverse also appears smoother. That is because the testing set for the inverse network contains 10% of all data, therefore here it is only one element. That means that all testing batches are the same (although that is not true for thes training batches).

One unexpected result was the fact that while in most runs the unit tended to be around 0, sometimes it also settled around 10. That could also be considered a right answer, although 10 is not in the chosen representation.

While learning the extension, most of the time the "half" settled around 5.5. This is of course one of two possible intuitive solutions to $a + a = 1$, the other of course being 0.5. Table 3.2 shows how the extension settled in several different runs. Note that the runs had different lengths, but the *half* always settled so early that the length had no effect.

Table 3.3 shows how two different runs generated $\mathbb{Z}_{20}$ using iterated (learned) composition on the "half" element.

3.1.2 $\mathbb{Z}_{20}$

The second group we experimented with was $\mathbb{Z}_{20}$. Because of its larger size, the experiments with it were longer.

Once again, the equation by which we extend the group was $a + a = 1$. The results of one experiment can be seen in Figure 3.4. Once again, 10% was excluded from the training. The trend lines are similar to $\mathbb{Z}_{10}$ (Figure 3.1), with composition and inverse training quickly, while inverse lags behind. The inverse line is much noisier, because now the testing data has 2 elements, out of which 5 are chosen for testing (with repetition).

Values for the half in different runs can be seen in Figure 3.5. Once again, the algorithm usually found one of the two intuitive answers, 0.5 and 10.5. A graph showing the iterated composition of the half is depicted in Figure 3.6. As we see, the learned structure appears at first as very similar to the expected $\mathbb{Z}_{40}$, however we start seeing very large deviation after 40 compositions. The cause of this is unknown, but we suspect it could be caused by the network not learning the associativity axiom. Indeed, $19 + 1 \doteq$19+(half+half) outputs correctly 0, however (19+half)+half does not output 0.

3.1.3 Infinite group $\mathbb{Z}$

The last cyclic group experiment was with the infinite group $(\mathbb{Z}, +)$. The training set comprised of integers from an interval $[-10000, 10000]$ in order to avoid overflow errors. Because neural network learning is already hard on such a diverse set, we opted to not exclude any data for training.

$a + a = 1$ still has no solution in this group. In contrast to the previous cases, there is only one intuitive solution and that is $\frac{1}{2}$. The group generated by this $\frac{1}{2}$ is isomorphic to the original. Unfortunately, we were unable to get to this point. The learned half was -1.1713637. When we tried to generate some elements of the extension, we obtained

1	2	3	4	5	6
-1.1713637	1.0000439	2.2069557	2.8777812	3.2506382	3.4578807

7	8	9	10
3.57307	3.637094	3.6726806	3.6924593

Indeed, the learned half+half=1, but the composition fails for larger iterations.

3.2 Symmetric groups

Symmetric groups, or permutation groups are the most complex finite groups. Indeed, a corollary from Cayley's theorem is that every finite group is isomorphic to a symmetric group of large enough size (D.L.Johnson [1971]). This is why they have been chosen for further experiments.

Every symmetric group is generated by the set of transpositions of two elements. Each element can therefore be written as a sequence of transpositions. Although this sequence is not unique, all such sequences have the same length. For each permutation σ we define $\operatorname{sgn}(\sigma) = (-1)^l$ where l is this number of transpositions. It also holds that $\operatorname{sgn}(\sigma_1 \circ \sigma_2) = \operatorname{sgn}(\sigma_1)\operatorname{sgn}(\sigma_2)$. Therefore $\operatorname{sgn}(\sigma \circ \sigma) = 1$. (proofs in Hladík [2019])

For this reason the equation that we sought to find the extension for is $h \circ h = (0, 1)$ where $(0, 1)$ denotes the transposition of elements 0 and 1. $\operatorname{sgn}((0, 1)) = -1$, therefore we know that h can not be in the original group.

The structure of this extension is suspected to be some form of the semidirect product of $S_n \times \mathbb{Z}_2$, although the basic semidirect product appears to be insufficient.

3.2.1 S_4 with a basic grounding

S_4 is the group of permutations of 4 elements. A ”naïve” grounding of elements was chosen first. It assigns each permutation (a, b, c, d)[1] the element $[a, b, c, d] \in \mathbb{R}^4$. The advantage of this grounding is its simplicity and shortness. The main disadvantage, however, is that the mean squared error loss function is biased. We can see this on an example where $[0, 1, 2, 3]$ is one transposition away from both $[1, 0, 2, 3]$ and $[3, 1, 2, 0]$, however in the latter case the mean squared error is considerably higher.

For the training we once again use only 90% of the data. During the training we expectedly observe significantly higher training times, as exemplified in Figure 3.8. This is also compounded by having slightly more elements (24). We are also unable to achieve the same precision. Learning of the unit was again very precise, see table 3.9.

When it comes to learning of h, we see even more problems than in the case of cyclic groups. Some half elements are seen in table 3.10. They are noticeably different from each other, something we did not observe before. The reason why these were the vectors found is unknown.

However, as we see in table 3.11, generating even a small subgroup results in a failure. Since $h \circ h = (0, 1)$, we expect that $h^4 = (h \circ h) \circ (h \circ h) = (0, 1) \circ (0, 1) = e$. Unfortunately, we were never able to get this equation to hold with $h \circ (h \circ (h \circ h))))$. Once again, reason for this is unknown. But because $(h \circ h) \circ (h \circ h)$ yielded reasonable results, we suspect that again the associativity is the problem.

3.2.2 S_4 with matrix grounding

To fix the problem with the bias of the loss function we add the ”one-of-n” representation. This representation is used for elements that are equally different from each other. This representation attaches to each of the n elements a vector from the canonical basis of $\mathbb{R}^n$. If we take the basic representation of S_4 and change every element to its one-of-n representation, we get a vector in $\mathbb{R}^{16}$ that has exactly four ones and the rest are zeroes. For example

$$(1, 2, 0, 3) \to ((0, 1, 0, 0), (0, 0, 1, 0), (1, 0, 0, 0), (0, 0, 0, 1))$$

This erases the loss function bias, because now the mean squared difference between any two permutations depends only on the number of elements switched. One-to-n representation can also be seen as the matrix representation. Indeed, $(1, 2, 0, 3)$ can be represented as

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} b \\ c \\ a \\ d \end{pmatrix}.$$

[1]This being the full representation of the permutation.

A proof of this is an easy exercise in linear algebra.

This new grounding, however, brought several new problems. For example 3/4-ths of the expected output are zeroes. This created a problem early on in the experimentation with the network only outputting a zero vector on any input. That is the reason why leaky ReLU had been used as an activation function as opposed to standard ReLU. This change solved this problem, since they avoid the "neuron death" - a situation where input to ReLU is so negative that it always outputs 0 and thus has no gradient.

Another problem was the size of the vectors relative to the number of elements, e.g. S_3 has only 6 elements but this representation has size 9. However, since the size of the group S_n grows exponentially while the matrix representation only quadraticaly, for large enough n the benefits of a less biased loss function outweigh the costs. Unfortunately, because of the amount of computation power needed to get to this point we were not able to verify this hypothesis.

As we can see in Figure 3.12, this representation is still efficient for composition, but the inverse is clearly wrong. Furthermore, as we see in table 3.13, the unit was usually significantly different from what was expected. This is the place where a penalty for straying away from an element could help significantly, as this problem most likely emerged as a consequence of the increase in dimension.

These two factors lead to the consequence that the inverse is not able to output elements of the grounding. If it did, composition would also output $a^{-1}a$ in grounding, but the learned e is not in it.

The extension element h is shown in table 3.14 along with parts of the generated subgroup. In the same vein as before, $h \circ h$ shows promise, but h^4 breaks down.

Figure 3.1: Error rate for the learning of composition, inverse and unit in $\mathbb{Z}_{10}$. Testing data percentage is 10%

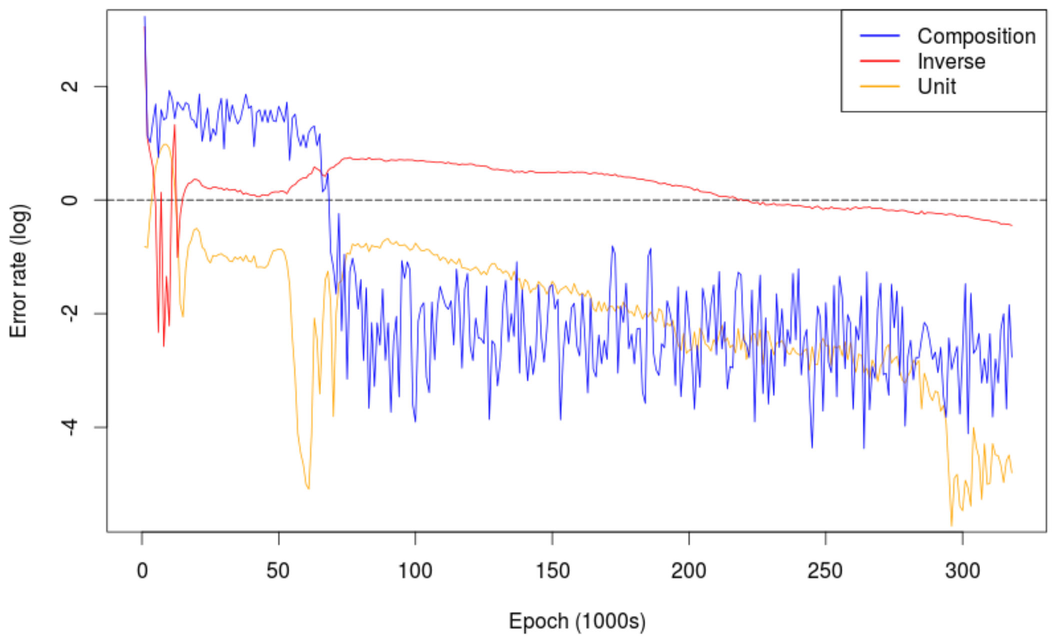


Figure 3.2: Different values of "half" found during different runs. The first run (red) was very peculiar since it had over 2 700 000 epochs, but this representation settled already around epoch 300 000. It is not clear how this interacts with the rest of the elements.

-0.9417969	5.5017343	5.500125	0.5014977	5.500004	5.507943

Figure 3.3: Extension groups generated in two runs (read left to right, top to bottom). The blue elements are supposed to be in the original embedding, i.e. they should be 1...9 ascending with the last 3 elements being 0, "half", 1 respectively. In the first run we see that we have had very low success, even though the first half+half looks promising. The second run had much better success and with the exception of 9 it hit all original elements reasonably well, even those last 3.

5.500004	0.99998194	10.919293	6.419118	1.9191066
9.268123	4.7680063	0.26805452	12.234171	7.7339497
3.2338893	8.733828	4.233731	1.9053116	9.292908
4.792793	0.2928396	12.189646	7.6894255	3.1893663
8.689303	4.189211			

5.5069175	0.99456155	6.5232387	2.0135	7.5396843
3.032562	8.556266	4.051762	3.872835	5.4972463
0.9848655	6.5135655	2.0038028	7.530016	3.022867
8.546589	4.04206	3.9609222	4.6975384	0.18309715
5.713754	1.20193			

Figure 3.4: A $\mathbb{Z}_{20}$ learning run. We see that the trends we observed in $\mathbb{Z}_{10}$ continue, and it appears that extra time is not needed. However, in different runs we encountered difficulties with learning the inverse function.

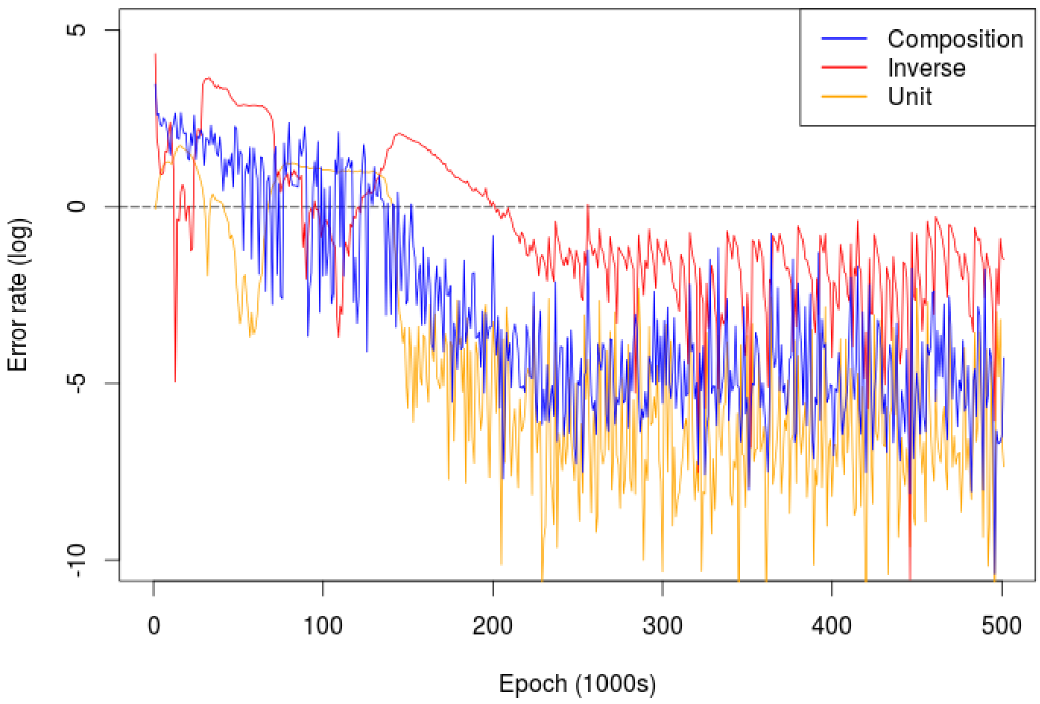


Figure 3.5: Values for the half in different $\mathbb{Z}_{20}$ runs. Once again, the reason for why the red one is so different is unknown. The tendency towards 0.5 instead of 10.5 can be attributed to the fact that the initial parameters have been restricted to avoid overflow errors.

0.4999506	-6.5685954	10.500707	0.49987993	0.5000777
0.49967808	0.49978873	0.49993014	0.50047106	10.499506

Figure 3.6: An example of a group generated from the "half" element. We see that there were no problems with addition of the half, except the modulus is not applied correctly.

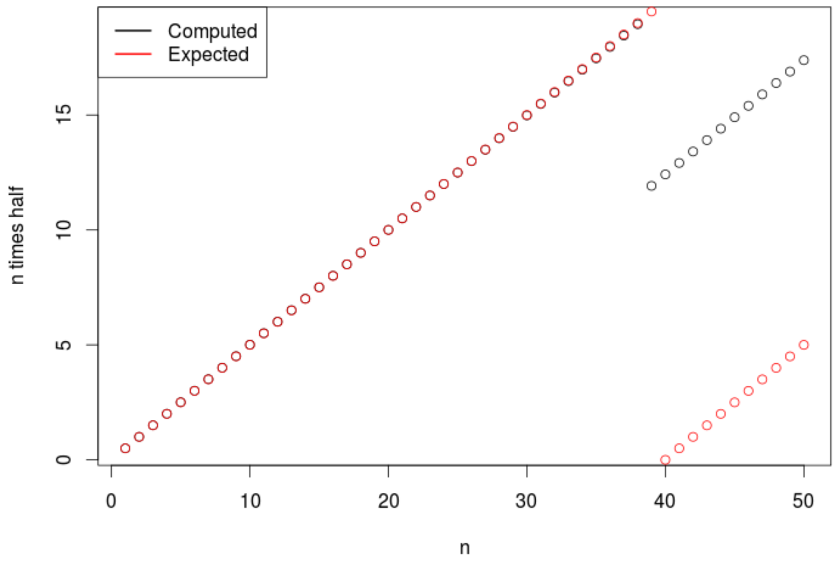


Figure 3.7: $\mathbb{Z}$ as an infinite group. Because of limitations of the software, we only trained on the interval $[-10000, 10000]$ (hence the trained group is not really infinite). There was no testing set, but the network seemed to generalize well even outside of the training interval. Examples are computed $11000^{-1} = -11001.614$ or $15000 + (-12000) = 2999.9941$.

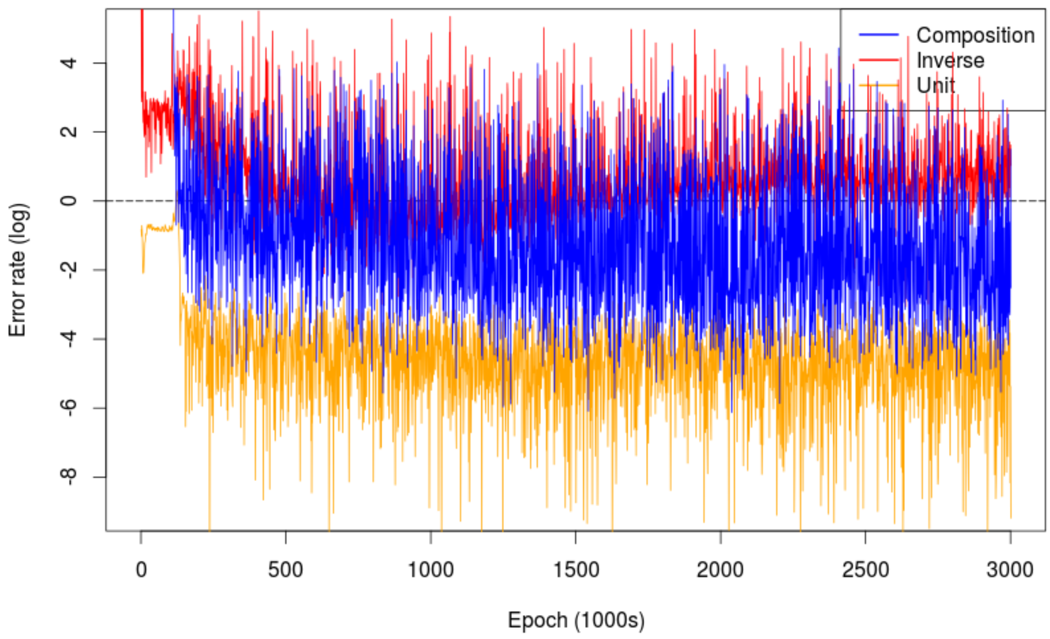


Figure 3.8: One run of learning the S_4 with the basic grounding. We see that the training is expectedly much slower than in the cyclic groups.

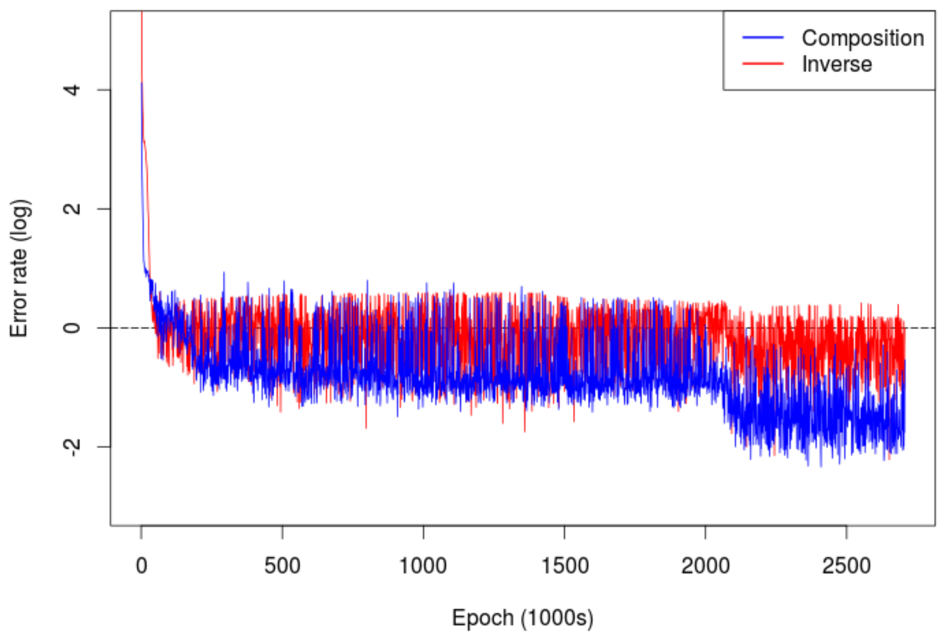


Figure 3.9: Units learned in S_4 with basic grounding. The expected output was $[0, 1, 2, 3]$. As we see, they are very precise.

1.)	0.03967234	1.0745986	1.957876	2.9761093
2.)	0.00685234	1.0557759	2.1151063	3.2438033
3.)	0.01609807	0.9784863	2.011951	3.0008512

Figure 3.10: h-s found in several runs. As expected, they do not look like anything.

1.)	-0.9288303	1.8216157	0.17884427	2.0224957
2.)	1.2200519	0.5469334	3.7773933	-0.22675751
3.)	2.9790108	2.5041845	1.9638568	-1.186425

Figure 3.11: h composed with itself several times. h^2 and h^6 should both be $[1, 0, 2, 3]$. h^4 should be $[0, 1, 2, 3]$.

h	2.9790108	2.5041845	1.9638568	-1.186425
h^2	1.0137038	0.6440371	1.960084	3.0298772
h^3	1.899735	0.6546894	2.3109381	2.1213117
h^4	1.4959755	0.52787334	2.564281	2.4759564
h^5	1.3819728	0.7297171	2.8694618	2.1578472
h^6	1.1029358	0.97653824	3.1764572	1.9179444

Figure 3.12: S_4 with matrix grounding. The composition is very successful, but the inverse is not.

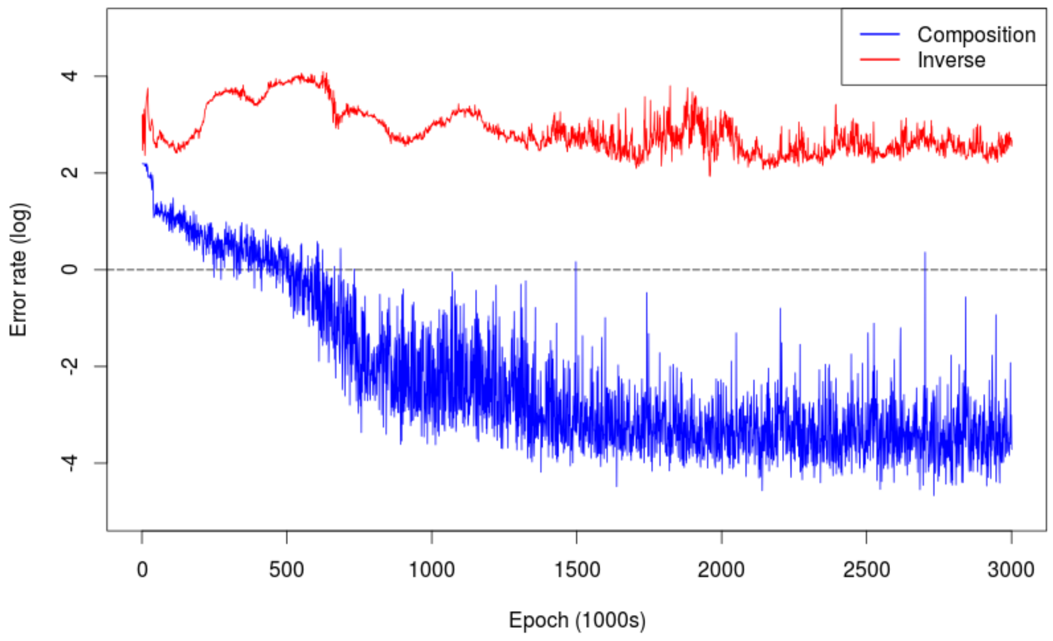


Figure 3.13: Units found in S_4 with matrix representation. An identity matrix was expected. Places where 1 was expected are blue, black ones are for 0.

0.9291287	0.39432997	-0.09073094	-0.00462165
-0.49930313	2.5813785	-1.0001388	-0.51179993
0.38528794	0.32126167	1.4879085	-0.3122037
0.16110185	-0.10436057	-0.3780875	1.356762

0.36581007	0.11518399	-0.29025748	0.7275856
0.38638538	0.95944846	-0.47597495	-0.14604506
0.203323	-0.15147878	0.65808797	-0.03862157
0.4341608	-0.57306635	-0.15940993	1.5463064

Figure 3.14: An extension attempt for S_4 with matrix representation. The blue numbers are expected to be 1 and black ones 0. As we see, h^2 is pretty much exactly what we wanted, but h^4 breaks down.

h	-0.44275028	0.45813385	0.84849375	-0.4929848
	0.29338264	0.25557452	0.701611	-0.33617198
	0.5497755	0.75910103	-0.16280994	-0.17575327
	0.20515643	0.25966993	0.10358979	1.0272595
$h \circ h$	0.0016880417	0.99703968	-0.0002135747	-0.0009868203
	0.99901026	-0.0018832732	-0.0010174632	0.0011361403
	-0.0027710588	-0.0013556076	1.0073379	-0.001112761
	-0.0005431428	0.0049326816	-0.0010595275	1.00323
h^4	0.72058374	0.17218184	0.04241377	0.04261543
	0.4558762	-0.00405501	0.55539876	0.00293861
	-0.02832983	0.10163078	0.4973646	0.4502491
	-0.02180864	0.6911213	-0.02542126	0.4643306

4. Comments and discussion

The main difference of our approach and the approach of Serafini and d'Avila Garcez [2016] is the exclusion of relations. Relations should also be neural networks that take a tuple of elements as input and output a number in $[0, 1]$. Relation is, just like an equality between two terms, an atomic formula. In Chapter 2 we have discussed how to combine different atomic formulas with fuzzy logic operators.

The loss functions shown in Chapter 2 were based on mean squared difference, which is very inefficient when it comes to 0-1 functions. Here we could use some modifications of cross-entropy function that are robust even with noisy labels (Zhang and Sabuncu [2018]), since we can not generally be sure of the expected truth value of an atomic subformula.

Another challenge is posed by the axioms that combine both a relation and a term equality, e.g. $R(\dots) \implies t_1(\dots) = t_2(\dots)$. Here we would have to combine the two different loss functions. One workaround is to use a different loss on the equality subformulas, something that is more related to the cross-entropy.

Our preliminary research showed that to learn each relation we expectedly need both positive and negative examples. We have tried to learn the Sheffer stroke (Sheffer [1913]) - a function on booleans for which all sentences of the type

$$((U|(V|W))|((Y|(Y|Y))|((X|V)|((U|X)|(U|X)))))$$

are true for all subformulas U, V, W, X, Y. The expected result is the XOR function. The axiomatic approach however led to the network always yielding 1, since there were no examples where it should output 0.

Another big challenge in the neural modelling is the choice of grounding. As we have seen with S_4, the choice can profoundly impact the learning process. For the models described in this thesis we have used handpicked groundings, but those require prior knowledge of the structure. In order to eliminate this requirement, we need to use a self-found representation of elements. We could do this using recurrent neural networks that have been extensively used in feature extraction, even in logic itself. For example Wang et al. [2014] have successfully embedded a knowledge graph (a set of 3-ary relations) to a continuous space, where similar relations are spatially closer to each other. This, under some modification, could prove a promising start for further research.

We have also encountered a big problem when training S_4 with the matrix grounding, due to the fact that the learned e was not in the original grounding. Although the axiom $a \cdot e = a$ was satisfied for all a in the grounding, e not being an element prevented the inverse function from being an $S \to S$ function. This leads to the conclusion that we should utilize some penalty for the constants (and maybe functions) that are too far from the given elements. One approach could be to alternate between learning the constants axiomatically and pushing them towards the nearest element ("grid-fitting"). With some fine-tuning of the learning rates this could provide us with a state of equilibrium where a constant settles on an established element. However if the learning rates are configured badly, we could end up in a state where the axiomatic optimizer seeks to abandon an element, but is continually pushed back by the ”grid-fitter”.

One of the main features of the Adam optimizer used in our experiments is the variable learning rate. Generally speaking, it learns quicker when it is far from the minimum and slower when it is near. We could utilize this and use an inverse learning rate for the ”grid-fitting” optimizer. This would lead to the Adam being dominant during the search for the minimum, and when the minimum is closer, the ”grid-fitter” would gain precedence and force the constant to be closer to one of the structure elements. This is, however, only speculation.

Conclusion

We have seen some promising results with regards to using neural networks to simulate particular mathematical models by learning on propositions that are true/false in them. We have successfully learned neural representations of groups, namely the cyclic and symmetric groups. Another focus of the work was building extensions to those models, relying on the learned functions.

For every model built here we used the multiplication table to learn the composition operation. Even despite the fact that the whole table is not needed (we have had good results even with 10% of the table missing), prior knowledge of the structure is still required. In order to truly follow the ideas of the model theory, we would need to drop the table altogether. How to do this is currently not known and would be a subject to further experimentation.

Another place to improve the method shown here is the grounding, specifically the usage of handpicked representations. This would ideally also be eliminated, since it is another essential part of the model that relies on prior knowledge of the structures. For the self-finding of the groundings we could use recurrent neural networks, which are widely used for feature extraction. Another method that could improve performance is mutable grounding, i.e. grounding that could change during the learning process to better reflect the structure learned. However, this might be quite nontrivial.

A very large part of this thesis is model extension. Despite initial optimism stemming from the successes of the finite cyclic groups, the results have been rather lackluster. We speculate that this is caused by the fact that we used the multiplication table rather than the associativity axiom. The associativity obviously holds in the original universe, since the multiplication table had been learned quite efficiently. One way to ensure associativity on the extension as well would be introduction of axioms such as $(h \cdot a) \cdot b = h \cdot (a \cdot b)$ where h is the extension element. However, training for general associativity - i.e. associativity on the whole domain $\mathbb{R}^n$ might slow down the learning process severely.

Because the work shown here is very early, there was little focus on the end goal - building an oracle that would gauge the probability that a given sentence is true. This would be a boon to the automated theorem proving community. Current trend is to use machine learning on the sentences themselves, thus skipping the models altogether. This approach has considerable limitations.

Unfortunately, the model-building process is very slow (order of hours on a home computer), therefore building an array of models for every problem would take some time. Classical Automated theorem proving competitions run in relatively short times (minutes), rendering this method rather unwieldy for usage in the state it is in right now. There is however a feasible niche for this model-based oracle in recent large-theory competitions and benchmarks and in theory building, i.e. expanding a given theory without a set goal. Large-theory benchmarks such as CASC LTB and the MPTP Challenge provide a large global time limit (days) for solving many related problems. Machine learning of useful models for predicting the validity of lemmas and conjectures could be very useful there.

Another area where the neural models could be useful is axiom selection, like SRASS described in Urban et al. [2008]. Proving a conjecture C from a large knowledge base usually needs only a subset of the axioms. Since automated theorem provers perform better on smaller axiom sets, finding such a subset is crucial. SRASS attempts this by numbering the axioms in the knowledge base $\varphi_1, \varphi_2, \ldots$ and then finding a model for $\{\neg C, \varphi_1, \ldots \varphi_n\}$ for ever larger n. If no such model exists, $\varphi_1, \ldots \varphi_n$ is the desired subset on which the theorem prover can run.

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.

G. Cybenko. Approximations by superpositions of sigmoidal functions. *Mathematics of Control, Signals, and Systems*, 4(2):251–257, 1989.

D.L.Johnson. Minimal permutation representations of finite groups. *American Journal of Mathematics*, 93:857–866, 1971.

Aleš Drápal. *Teorie grup: základní aspekty.* Karolinum, 2000. ISBN 80-246-0162-1.

Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. *Conference on Learning Theory*, page 907–940, 2016.

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.

Milan Hladík. *lineární algebra (nejen) pro informatiky.* 2019. URL `https://kam.mff.cuni.cz/~hladik/LA/text_la.pdf`.

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.

Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6231–6239. Curran Associates, Inc., 2017. URL `http://papers.nips.cc/paper/7203-the-expressive-power-of-neural-networks-a-view-from-the-width.pdf`.

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.

Francis Jeffry Pelletier. Petr hájek. metamathematics of fuzzy logic. trends in logic, vol. 4. kluwer academic publishers, dordrecht, boston, and london, 1998, viii 297 pp. *Bulletin of Symbolic Logic*, 6(3):342–346, 2000. doi: 10.2307/421060.

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.

Luciano Serafini and Artur S. d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. *CoRR*, abs/1606.04422, 2016. URL `http://arxiv.org/abs/1606.04422`.

H.M. Sheffer. A set of five independent postulates for boolean algebras, with application to logical constants. *Transactions of the American Mathematical Society*, 14:481–488, 1913.

Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. Malarea sg1 - machine learner for automated reasoning with semantic guidance. In *Automated Reasoning*, pages 441–456. Springer Berlin Heidelberg, 2008.

Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph embedding by translating on hyperplanes. 06 2014.

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

Zhilu Zhang and Mert R. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. *CoRR*, abs/1805.07836, 2018. URL `http://arxiv.org/abs/1805.07836`.