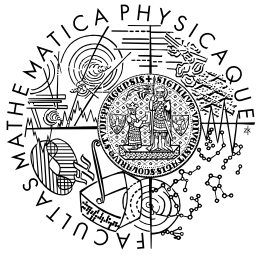


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 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

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 this thesis we propose and try to build a new tool for automated theorem proving. Theorem provers, that is algorithms that given a theory and a conjecture find a formal proof, mostly use syntactic rules to manipulate formulas. This approach is very different from how human mathematicians think about mathematics. When a human is working with a theory, they usually do so with some sort of mental images of the structures that satisfy it. We aim to build these models for automatic provers, thus giving them the ability to also leverage the semantic meaning of the given theory. These models could be used as an oracle that guesses validity of given sentences by trying whether or not they hold in the trained models.

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 machine learning, namely 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.

The main subject and the main contribution of this thesis is the description and testing of the network architecture (inspired by Serafini and d'Avila Garcez [2016]) that enables the networks approximating the structure functions to learn using the propositions that are true in the structures. The description of this architecture can be found in Chapter 2. Using 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.

Another minor contribution is the proof that the algebraic extensions of groups are always possible (in Subsection 1.1.4). It is very likely that this was proven before, but we were unable to locate any such proof, thus we present this proof as our own.

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 $\mathcal{S} \models \varphi(x_1, \dots x_n)$. 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

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

where $x_1, \ldots x_n$ are all universally quantified and ψ is a formula in prenex normal form. Then 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 $f_y(x_1, \ldots, x_i)$. 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 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 $Q_1 x_1, \ldots Q_n x_n \psi(x_1, \ldots x_n)$ 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].

1.1.3 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** over $\mathcal{S}$. If $\varphi(x)$ holds for some $x \in S'$, we say that x is the **solution** of φ. 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 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 $x^{-1}Gx \leq G$. 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} where g_i and g_{i+1} are elements of G, we replace it with their product in G: $(g_i \cdot g_{i+1})$.*
  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.*

We will still treat G as a subgroup of $G[x]$, by identifying all elements of G with a string made up by just this element.

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. We can also assume that every term $t(x)$ is equivalent to an element of $G[x]$, because the simplifications described in Definition 23 produce terms that are equivalent, i.e. they yield the same values under the same variable assignment.

Definition 24. *Let t be an element of $G[x]$ of the form $g_1x^{n_1}g_2x^{n_2}\dots g_kx^{n_k}g_{k+1}$. The **degree** of t is $\sum_{i=1}^{k} n_i$.*

We also naturally assign degrees to terms. For every term $t(x)$ that corresponds to $t \in G[x]$, we say that degree of $t(x)$ is the degree of t as defined above.

Lemma 3. *For any $t, t' \in G[x]$ with degrees n, m. the following hold:*

1. *$t \cdot t'$ has the degree $n + m$.*

2. *t^{-1} has the degree $-n$*

Proof. The part 2 is easily seen from the definitions of $^{-1}$ in $C[x]$ and the degree.

We observe that the adjustments described in Definition 23 do not change the sum of exponents of the x-es. Thus $t \cdot t'$ has the same degree as the sum of exponents of x-es in concatenation of strings t and t'. This sum is obviously $n + m$. □

Theorem 4. *Let G be a group and $\varphi(x)$ an equation of the form $t(x) = e$ with a non-zero degree d. Then there exists an extension G' of G such that there exists $g \in G'$ that is a solution to φ.*

Proof. Without loss of generality we asume that $t(x)$ 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.

Let t be an element of $G[x]$ corresponding to $t(x)$. Let us then define $B = \{ptp^{-1}|p \in G[x]\}$. $\langle B\rangle$ is a normal subgroup of $G[x]$. We immediately see that $pbp^{-1} \in \langle B\rangle$, (even $pbp^{-1} \in B$) for all $b \in B, p \in G[x]$. Now if $b = b_1b_2\dots b_n$ where $b_i \in B$ then

$$pbp^{-1} = pb_1b_2\dots b_np^{-1} = pb_1p^{-1}pb_2p^{-1}\dots pb_np^{-1}.$$

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

$$pbp^{-1} = (pb_1p^{-1})(pb_2p^{-1})\dots(pb_np^{-1}) = b_1{}'b_2{}'\dots b_n{}'.$$

Since $\forall i\ b_i{}' \in B$, also $pbp^{-1} \in \langle B\rangle$ and thus $\langle B\rangle$ is normal.

Also note that by Lemma 3 all elements of B have degree d. Thus no elements of $G[x]$ that have degree 0 are in B. Therefore every element of B contains at least one occurrence of x^n (where $n \neq 0$).

Since $\langle B\rangle$ is a normal subgroup of $G[x]$, we know that $G[x]/\langle B\rangle$ is a group. In it, all the elements of G form different cosets, since $g\langle B\rangle$ always has the element $g \cdot t$ and no two $g \in G$ can produce the same $g \cdot t$. Assuming that $g \notin \langle B\rangle\ \forall e \neq g \in G$, G is isomorphic to a subgroup of $G[x]/\langle B\rangle$, and thus it can be seen as an extension of G.

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

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

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

Lastly, we need to prove that $g \notin \langle B\rangle\ \forall g \in G, g \neq e$. Let g be a counterexample, meaning that $g = \beta_1\beta_2\dots\beta_n$ where $\forall i$ either $\beta_i \in B$ or $\beta_i^{-1} \in B$. We can assume that β_1 starts with g and β_k ends with e (or rather, a power of x). That is because if β_1 starts with k and β_2 ends with l, where $kl = g$ in G (the only possibility of obtaining g is if all occurences of x annihilate each other), we take β_i' to be $l\beta_il^{-1}$ and we get the desired start. Then β_1 ends with g^{-1}. β_2 therefore has to start with g. By induction we see that $\forall i\beta_i$ has to start with g and end with g^{-1}. This is a contradiction with β_k ending with a power of x. □

A reader acquainted with field extensions will probably recognize this construction. Indeed, it had been the inspiration for this proof.

The main difference, however, is that the extensions are not automatically unique, i.e. the groups that we can consider to be algebraic extensions are not necessarily isomorphic to each other. The property that ensures the uniqueness in field extensions is commutativity. We will show that commutative groups have unique comutative extensions.

Definition 25. *We say that a group is* ***commutative****, or* ***Abelian*** *if $a \cdot b = b \cdot a$ for any a, b.*

Theorem 5. *Let G be a commutative group and $\varphi : t(x) = e$ an equation. Then there exists a commutative extension G' of G for which the following hold:*

1. *G' contains a solution for φ*

2. *If H is a commutative extension of G that contains a solution for φ, then G' is isomporphic to a subgroup of H.*

Proof. We can redefine $G[x]$ to also be commutative (denoted $G_A[x]$), by adding a fourth inverse adjustment rule: replace gx^n with $x^n g$. With this rule in place, we can see that all elements of $G_A[x]$ have the form $x^n g$ where $n \in \mathbb{Z}, g \in G$.

We can see that if we assume commutativity in $t(x)$, it is also equivalent (in the same sense as above) to an element $t \in G_A[x]$.

Therefore we may without loss of generality assume that $t(x)$ has the form $x^n c$.

If we construct $B = \{ptp^{-1} | p \in G_A[x]\}$ as above, we observe that due to commutativity it is just $\{t\}$. Therefore $\langle B \rangle = \{t^k | k \in \mathbb{Z}\}$. We once again construct $G_A[x]/\langle B \rangle$ that has the solution for φ and is an extension of G. Any factor of a commutative group is commutative (Drápal [2000]). This group will be denoted G'.

Let us explore the structure of G'. For the sake of simplicity we will denote the elements as $x^k g$.[8] We know that $x^n \cdot c = e$ in G'. Therefore $x^{-1} = x^{n-1}c$. Also, if $k > n$, then $x^k g = x^{k-n} c^{-1} g$. Therefore we can assume that for any element $x^k g$ it holds that $0 \leq k < n$ (because every factor set has exactly one element of this form).

Now we shall prove the uniqueness of G'. Let H be a commutative extension of G that has an element h that solves φ. Then $h^{-1} = h^{n-1}c$, since $h^n c = e$, just as well as $h^n = c^{-1}$. Therefore any element in this subgroup has the form $h^k g$ where $0 \leq k < n$ and $g \in G$. We can easily see that this subgroup is isomorphic to C' described above. □

These are however not the only extensions of commutative groups. The general construction only produces infinite non-commutative groups, since they will always contain infinite number of elements of the form $xgxgxg...xg$ (assuming t has not the same form for the same g).

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 $\epsilon > 0$ for every smooth function there exists 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[9]. How exactly do we do that will be discussed in Chapter 2. This section will introduce the reader to the basics of neural networks. 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

[8] Rather than the factor sets

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

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 a type of a $\mathbb{R}^n \to \mathbb{R}^m$ function that is smooth almost everywhere and has the form $f(x) = f_n(f_{n-1}(\dots(f_1(\mathbf{x}))\dots)$, where f_i are real vector functions, called **layers**. 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 of f_i does not need to have the same dimension as the input $\mathbf{x}$ or the output $\mathbf{y}$. 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.

There is no overarching definition describing which functions are layers, thus every type of network may have a different type of layer function. However, in neural networks, there we almost always use a certain type of the layer function, which will describe here.

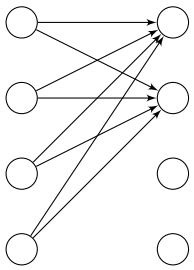
These layer functions consist of an affine transformation weight matrix $\mathbf{W}$, bias vector $\mathbf{b}$ and some activation function $\varphi : \mathbb{R} \to \mathbb{R}$. They are combined as

$$f_i(\mathbf{x}) = \varphi \odot (\mathbf{W}\mathbf{x} + \mathbf{b})$$

where $\varphi \odot (v_1, ..., v_n)^\top$ denotes $(\varphi(v_1), ...\varphi(v_n))^\top$, the element-wise application of φ. Activation functions and our requirements for them are further discussed below.

The networks using this type of layers are called neural, because they are inspired by biological neurons. Figure 1.1 illustrates the *informal* intuition behind this. Each arrow represents a $\mathbb{R} \to \mathbb{R}$ function between two nodes. Each node represents a neuron that either "activates" or not, based on the sum of the functions leading to it and the activation function. The biological inspiration comes from the fact that biological neurons also either send a signal or not, based on the signals from surrounding neurons.

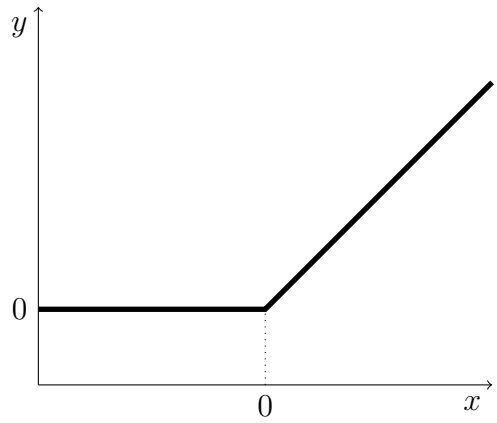
Figure 1.1: Neurons in a layer



The simplest neural networks are linear models, e.g. linear regression. In these models there is no activation function. These models are simple, however they have many drawbacks. The main one is their ability to only model linear functions. Indeed, if the layers are linear functions, several layers in a sequence also form a linear function.

We overcome this by introducing non-linearities via the activation functions. Activation function therefore has to be a *non-linear* $\mathbb{R} \to \mathbb{R}$ function. We generally also require some degree of smoothness to enable optimalisation (see Subsection 1.2.5). 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 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 non-linear 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, along with other used alternatives.

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. A linear regression model of a function $f : \mathbb{R}^n \to \mathbb{R}$ is a function $\widehat{f}_\theta$ that is affine, i.e. it inputs a vector $\mathbf{x} = (x_1, ..., x_n)$ and outputs a scalar $a_1x_1 + ... + a_nx_n + b$ where $\{a_1, ..., a_n, b\} = \theta$ are some real parameters. The points $(x_1, ...x_n, \widehat{f}_\theta(\mathbf{x}))$ form an affine plane in $\mathbb{R}^{n+1}$. A plane of a linear regression model approximating XOR needs to intersect points $(0,0,0); (1,0,1); (0,1,1)$ and $(1,1,0)$. The only plane that intersects the first three is defined by the equation $x_1 + x_2 - y = 0$. However, this does not intersect the last point.

Linear regression is usually optimized using the least squares method. This means that the parameters $\theta = \{a_1, ..., a_n, b\}$ are defined as

$$\theta = \text{argmin}_\theta \left\{ \sum_{i=1}^{k} \left(f(\mathbf{x}^{(i)}) - \widehat{f}_\theta(\mathbf{x}^{(i)}) \right)^2 \right\}$$

where $\{\mathbf{x}^{(i)}\}_{i=1}^{k}$ is a finite set of points that we seek to approximate f on. Using this method on our 4 points, the linear regression model of XOR function has parameters $a_1 = a_2 = 0$ and $b = \frac{1}{2}$. This is a constant function $\widehat{f}(\mathbf{x}) = \frac{1}{2}$, an obviously unsatisfactory approximation.

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

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

where

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

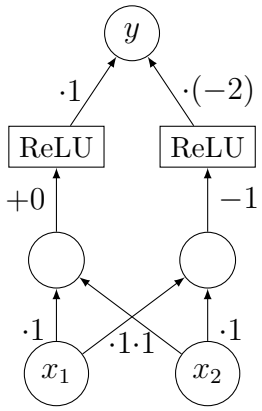
$$\mathbf{c} = \begin{pmatrix} 0 \\ -1 \end{pmatrix},$$

and

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

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 input the points 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 (more about optimization can be found in Subsection 1.2.5). Here we found the perfect solution, a global minimum to the loss function. However that is not generally possible.

1.2.3 Other activation functions

ReLU, while being the most popular activation function, is not the one used in the models built for this thesis, because it also has some drawbacks. The fact that its gradient is 0 on all negative inputs means that the optimization process may lead 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 therefore 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 largely fell out of use.

Figure 1.4: Sigmoid and tanh functions

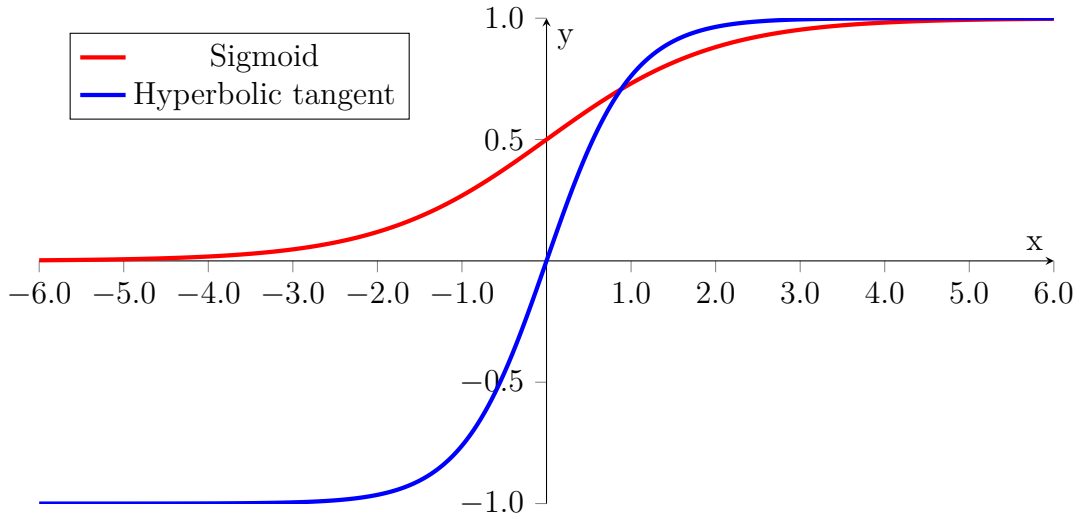
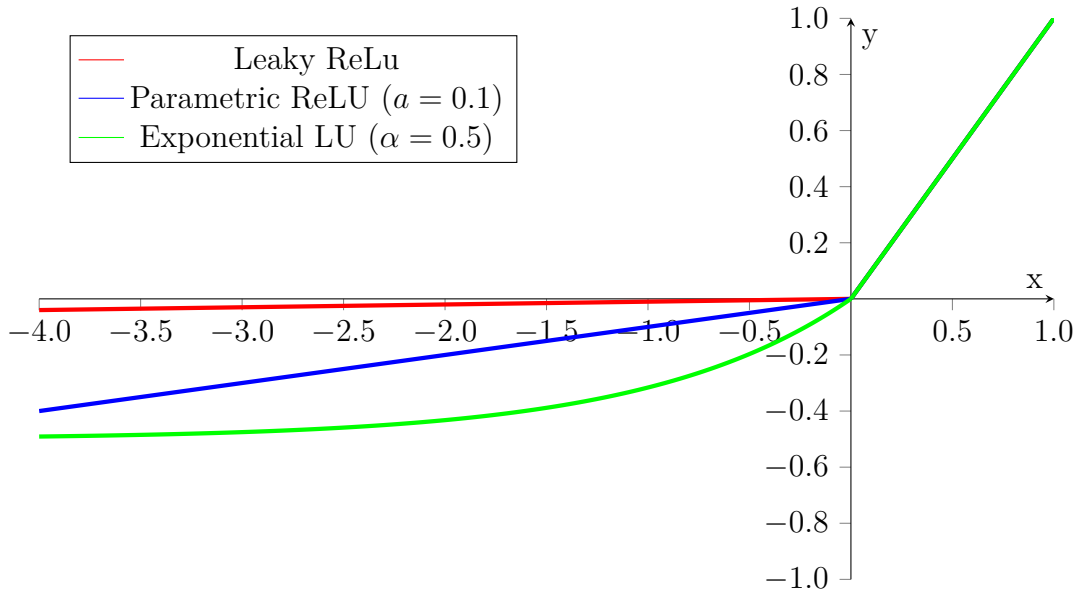


Figure 1.5: ReLU variations



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 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 and σ is the sigmoid function. 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])[10]. The universal approximation theorem is stated thusly:

Definition 26. *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*

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

for any $\theta \in \mathbb{R}$ and $y \in I_n$ implies that $\mu = 0$.

Theorem 6. *Let σ be a discriminatory function. Then the finite sums of the form*

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

are dense in $C(I_n)$ (space of all continuous function on I_n) with respect to the supremum norm.

In other words, 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 are too technical for the purposes of this introduction and can be found in the original paper.

[10] The statement of the Universal approximation theorem contains some non-trivial definitions that are beyond the scope of this thesis, as we will not use them elsewhere. However, they can be found in Taylor [1973] along with an introduction needed to understand the proof. This subsection only serves to illustrate the very technical statement of this theorem as shown in Cybenko [1989] and its consequences.

There is however no bound on the width of the network, which may pose a problem for 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 depth - had not yet been fully explored, although 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 $S \subset \mathbb{R}^n$ is the universe of a mathematical structure and $f : S^m \to S$ is one of its functions, we can approximate f 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. In neural networks the parameters $\theta \in \Theta$ represent the elements of the weight matrices and bias vectors, as well as activation function parameters, if any. 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 optimizationm methods on J we seek to find this minimum.

With neural networks we use the general form for loss

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

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 $\mathbf{X}$ (also called **batch**) in each of the optimization steps (also called **epoch**).

A very common loss function is the mean squared difference. 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[11]. It was also used to optimize the networks built for this thesis.

[11] Data points that are markedly different from others. There is no formal definition.

After we have chosen 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 optimization process can be very slow.

Usually the universe the function operates on is infinite, or too large to use efficiently. In those cases we perform the gradient descent in batches. As mentioned before, we take a (random) finite 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, which may lead to premature termination in a non-convex function. To combat this, we usually make some alterations. One approach is discussed in **??**.

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

Algorithm 1 A basic back-propagation algorithm for most of the basic feed-forward 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 a parameterless 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 to achieve non-linearity. 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
  $g \leftarrow \nabla_{\hat{y}} J(\widehat{y}, y)$: 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.

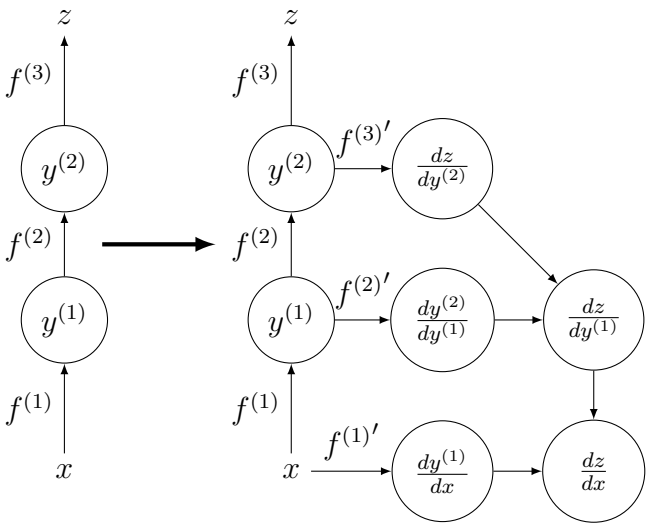
    $\nabla_{b^{(i)}} J = g$
    $\nabla_{W^{(i)}} J = g{x^{(i)}}^\top$ //g is a column vector and ${x^{(i)}}^\top$ is a line
    Compute the gradients with respect to this layer's parameters

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

Algorithm 1 is the implementation of this back-propagation that computes the gradient for one input. If our batch size is higher, we can 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 pass through the graph. Figure 1.6 illustrates this process.

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



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 [2017]). It is based on gradient descent, but it also conserves momentum. This means that in each optimization step it uses a weighted average of previous gradients, thus it is not as dependent on the specific batch chosen for the step. It also helps overcome local minima, since it takes several steps to reverse direction. It uses 4 parameters: α - learning rate, β_1 - gradient momentum decay, β_2 - second gradient moment momentum decay and ϵ - a small 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[12] functions or other functions with numerous local minima. In the last step 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.

[12]A function f is informally called noisy if it has many local minima and maxima, that follow a general trend g that has few extremes. Then $f = g + h$ where h is a comparatively small "noise" function that is unpredictable. During optimization for these noisy functions, we want to avoid these local minima, while converging to the minimum of the trend function.

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: 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)

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 however do the correction, we obtain the correct value.

Another useful property of Adam is its invariance towards rescaling. If we rescale (multiply) 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.

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 first order 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 27. *Let $\mathcal{L}$ be a language. Then a **grounding** in the space $\mathbb{R}^n$ 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:

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

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

Where "$\mathcal{G}(t_1), \ldots, \mathcal{G}(t_n)$" means concatenation[2] of the vectors outputted by the functions $\mathcal{G}(t_i)$

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-norm (or a triangle form) is a function in fuzzy logic that replaces conjunction, e.g. $\top_{\min}(a, b) = min\{a, b\}$ or $\top_{\text{Luk}}(a, b) = \max\{0, a + b - 1\}$. A co-t-norm 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 Hájek [1998].

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

[2] Concatenation of vectors $u = (u_1, \ldots, u_n)$ and $v = (v_1, \ldots, v_m)$ is the vector $(u_1, \ldots, u_n, v_1, \ldots, v_n)$

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 (see Subsection 1.1.2). 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 $S \subseteq \mathbb{R}^n$ 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 with input and output dimensions following the rules of groundings (see Definition 27). They are initialized randomly. Then we take these neural networks as building blocks and we build a network for each axiom of T. 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 (i.e. it has no $\wedge, \vee$ nor $\neg$), we use this as the basis for our loss function[3].

Formally speaking, we assume that the axiom has the form $t_1(\mathbf{x}) = t_2(\mathbf{x})$ where $\mathbf{x}$ is a set of m variables and t_1, t_2 are terms. Then $\mathcal{G}(t_1)$ and $\mathcal{G}(t_2)$ are functions $\mathbb{R}^{nm} \to \mathbb{R}^n$. They are constructed using the rules for groundings of terms (see above). This means that we have a network $\mathcal{G}(f)$ for every function f (we will denote it $\widehat{f}$), and if $t(\mathbf{x}) = f(s_1(\mathbf{x}), \ldots, s_k(\mathbf{x}))$ for some terms $s_1, \ldots, s_k$ then $\mathcal{G}(t) = \widehat{f}(\mathcal{G}(s_1), \ldots, \mathcal{G}(s_k))$. This rule is applied recursively, in effect replacing each symbol with its neural network equivalent.

A visual representation of this supernetwork can be seen in Figure 2.1.

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 27 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. $J_{\varphi_1} + J_{\varphi_2}$.

The networks constructed here are not forced to produce $S^m \to S$ functions, nor are the constants forced to be near any element of S. To remedy this, we will need to alter our loss function. Possible ways to do this are discussed in Chapter 4.

[3] See Subsection 1.2.5

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. $\widehat{f} = \mathcal{G}(f), \widehat{g} = \mathcal{G}(g), \widehat{h} = \mathcal{G}(h)$ are all neural networks and $\widehat{c} = \mathcal{G}(c)$ is a vector in $\mathbb{R}^n$. $\widehat{f}$ and $\widehat{g}$ concatenate all (3 and 2 respectively) of their input vectors. Loss is computed from the difference of $\widehat{t_1}$ and $\widehat{t_2}$.

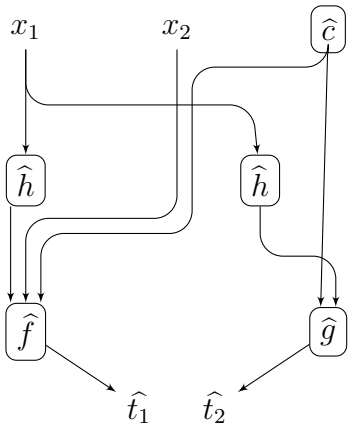
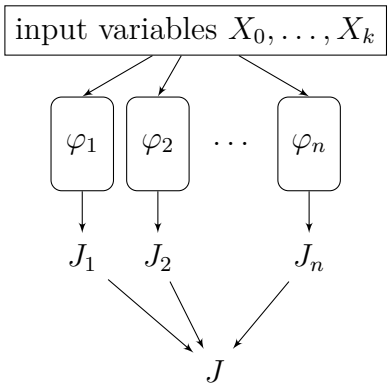


Figure 2.2: Network aggreggating. If $\varphi_1, \ldots, \varphi_n$ are axioms, we first build the supernetworks for these axioms as above and aggreggate them to an "overall loss" that we use the optimizing algorithm on. For the ease of use, the input variables are shared between these supernetworks.



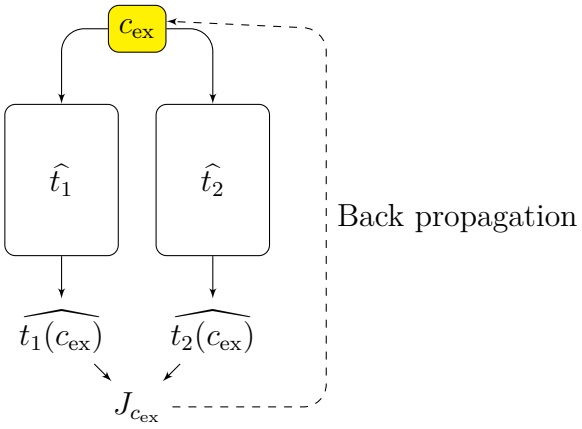
After computing the loss for each axiom, we aggregate them like with $\wedge$ and use an optimization algorithm (see Subsection 1.2.5) using back-propagation (see Subsection 1.2.6) from this aggreggated loss (this aggreggating is illustrated in Figure 2.2). The framework built for this thesis uses

$$J_{\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n} = \sqrt{J_{\varphi_1}^2 + J_{\varphi_2}^2 + \cdots + J_{\varphi_n}^2}$$

rather than the aforementioned $J_{\varphi_1} + \cdots + J_{\varphi_n}$ This is done to ensure that the networks that are less successful (have larger losses) are prioritized by the optimizer (as they have larger influence on the gradient).

Source code for this framework along with some examples of usage are included in Appendix 1.

Figure 2.3: Network used for learning an extension defined by the equation $t_1(x) = t_2(x)$. $\widehat{t_1}$ and $\widehat{t_2}$ represent networks built based on the terms, just like in previous section. Note that only c_{ex} is optimized, everything else is treated as a function.



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 $\mathcal{S}$ using the method described above. Now we want to extend it to include a solution of an equation $t_1(x) = t_2(x)$ (we assume that it has no solution in S). Note that t_1 and t_2 are allowed to have parameters from $\mathcal{S}$. Since we assume that the universe S is already fixed, 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 $t_1(c_{\text{ex}}) = t_2(c_{\text{ex}})$. We already have a framework to find the grounding of such a constant (Figure 2.3).

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.

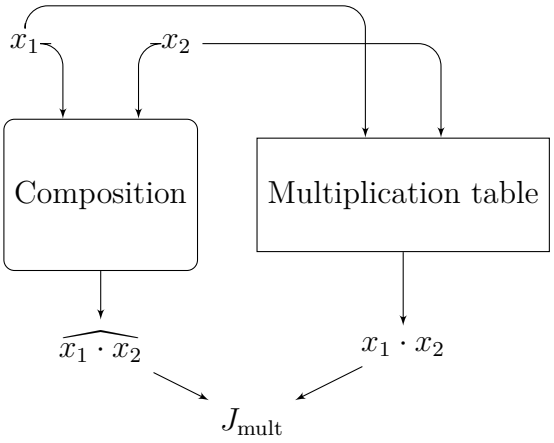
2.3 Our neural architecture implementing groups

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, S_4 and S_5). 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. The usage is illustrated in Figure 2.4. The loss function is computed from the difference between the computed value and the value in the table.

This is a simplification, contrary to learning the model only based on some propositions valid in it, but it is an essential one for human interpretability of the

Figure 2.4: Learning of composition. An incomplete multiplication table is used to determine the loss. x_1, x_2 is a randomly selected pair.



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.

Network architecture: Both $\cdot$ and $^{-1}$ are 4-layer feed-forward neural networks (see Subsection 1.2.1). The width of each layer is $3n$ where n is the size of the grounding[4]. The activation functions on each layer are leaky ReLU (see Subsection 1.2.3). The loss function for each network is the mean squared difference (see Subsection 1.2.5). The networks are optimized using the Adam optimizer with default settings (see subsection 1.2.7). Inputs are given in batches of 25 random pairs of elements.

The networks are then used as building blocks for supernetworks, as described in Section 2.1. The supernetwork used for learning the **composition** is described in Figure 2.4, **unit** in Figure 2.5 and **inverse** in Figure 2.6. All of these networks calculate their own losses $J_{\text{mult}}, J_{\text{unit}}, J_{\text{inv}}$ respectively. The final loss function that the optimizer seeks to minimize is $\sqrt{J_{\text{mult}}^2 + J_{\text{unit}}^2 + J_{\text{inv}}^2}$.

[4]This makes them $\mathbb{R}^{an} \to \mathbb{R}^{3n} \to \mathbb{R}^{3n} \to \mathbb{R}^{3n} \to \mathbb{R}^{3n} \to \mathbb{R}^n$ where a is 2 for composition and 1 for inverse.

Figure 2.5: Learning of the unit. 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. The composition node represents the learned composition network.

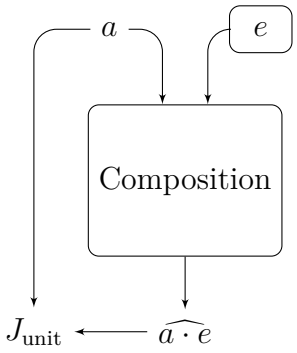
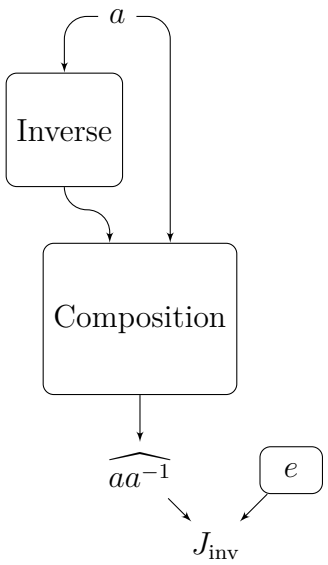


Figure 2.6: Learning of the inverse. Here we also use the learned unit and the network for composition. We use the axiom $a^{-1}a = e$. The dual $aa^{-1} = e$ is disregarded.



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 in order 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 $(\mathbb{Z}_n, +, -, 0) : \{0, 1, 2, \dots n-1\}$ with addition modulo n. They are all commutative.

3.1.1 Example network for addition in $\mathbb{Z}_n$

Here we will show an example of how a network that computes Z_n may look like. One such network for $n = 2$ is illustrated in Subsection 1.2.2, since Xor on $\{0, 1\}$ is identical to addition modulo 2. For other $n \in \mathbb{N}$, we will use a different network.

This network will have two layers, therefore it has the form

$$f(\mathbf{x}) = \mathbf{w}^\top \alpha \left(\mathbf{W}_2(\alpha(\mathbf{W}_1\mathbf{x} + \mathbf{c}_1)) + \mathbf{c}_2\right)$$

where $\mathbf{W}_1, \mathbf{W}_2$ are weight matrices, $\mathbf{c}_1, \mathbf{c}_2$ are bias vectors and α is an activation function.

We will set

$$\mathbf{W}_1 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \mathbf{W}_2 = \begin{pmatrix} 1 & -2n \\ 0 & 1 \end{pmatrix}$$

and

$$\mathbf{c}_1 = \begin{pmatrix} 0 \\ -n+1 \end{pmatrix}, \mathbf{c}_2 = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \mathbf{w} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

We set α to be the ReLU function, just like in the XOR example.

This network exactly computes $a + b \pmod{10}$, where $\mathbf{x} = (a, b)^\top$ and $0 \leq a, b < n$.

The first layer is $\alpha(\mathbf{W}_1\mathbf{x} + \mathbf{c}_1)$. That is

$$\alpha\left(\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} 0 \\ -n+1 \end{pmatrix}\right) = \alpha\left(\begin{pmatrix} a+b \\ a+b \end{pmatrix} + \begin{pmatrix} 0 \\ -n+1 \end{pmatrix}\right) =$$

$$= \alpha\left(\begin{pmatrix} a+b \\ a+b-n+1 \end{pmatrix}\right)$$

We will call the output of the first layer $(y_1, y_2)^\top$. We have two possibilities for $a+b$. Either $a+b \geq n$, then both $y_1, y_2 > 0$, or $a+b < n$, then $a+b-n+1 \leq 0$, so $y_2 = 0$. We see that after the application of α, at most one of the elements y_1, y_2 is non-zero.

Next we apply the second layer:

$$\alpha\left(\begin{pmatrix}1 & -2n\\ 0 & 1\end{pmatrix}\begin{pmatrix}y_1\\ y_2\end{pmatrix}+\begin{pmatrix}0\\ -1\end{pmatrix}\right)=\alpha\left(\begin{pmatrix}y_1-2ny_2\\ y_2\end{pmatrix}+\begin{pmatrix}0\\ -1\end{pmatrix}\right)=$$

$$=\alpha\left(\begin{pmatrix}y_1-2ny_2\\ y_2-1\end{pmatrix}\right)$$

We will call output of this layer $(z_1, z_2)^\top$.

If we have $a+b \geq n$, therefore $y_1 = a+b, y_2 = a+b-n+1$, we have $y_1 - 2ny_2 < 0$ (because $y_2 \geq 1$ and $a+b < 2n$), therefore $z_1 = 0$. In this case also $z_2 = a+b-n = (a+b)\text{mod } n$.

If we have $a+b < n$, therefore $y_1 = a+b, y_2 = 0$, we have $z_1 = a+b = (a+b)\text{mod } n$, $z_2 = 0$ (because $y_2 - 1 < 0$).

We see that one of z_1, z_2 always has the desired result, while the other is zero. Therefore the output layer $\mathbf{w}^\top \begin{pmatrix}z_1\\ z_2\end{pmatrix} = z_1 + z_2$ always gives the correct value.

Since we did not assume that a, b are natural numbers, this network is also stable under commutative algebraic extensions of $\mathbb{Z}_n$, i.e. it does not need any modification to work correctly with fractions (or even irrationals).

3.1.2 $\mathbb{Z}_{10}$

Elements: Integers $0, 1, 2, \ldots, 9$

Operations: Composition is addition modulo 10, inverse of a is $10 - a$ and the unit is 0.

Grounding: As themselves in $\mathbb{R}^1$

Extension: Using the equation $h + h = 1$. Because $h + h$ is "even", we can see that the base group has no solution. We call this h "half".

Notes: Since $\mathbb{Z}_{10}$ is commutative, we want the extension to be commutative as well (see Theorem 5). We expect to get an extension isomporphic to $\mathbb{Z}_{20}$.

The learning progress of one experiment with training the composition, inverse and the unit is depicted in Figure 3.1. The (logarithmic) error rate on the Y axis is the square root of loss (to closer resemble Euclidean distance).

The curve depicting the error rate of the inverse 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 $a + a = 1(\text{mod}10)$, 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 the extension (isomorphic to $\mathbb{Z}_{20}$) using iterated (learned) composition on the "half" element.

3.1.3 $\mathbb{Z}_{20}$

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

Elements: Integers $0, 1, 2, \ldots, 19$

Operations: Composition is addition modulo 20, inverse of a is $20 - a$ and the unit is 0.

Grounding: As themselves in $\mathbb{R}^1$

Extension: Using the equation $h + h = 1$. As with $\mathbb{Z}_{10}$, this has no solution in $\mathbb{Z}_{20}$.

Notes: Once again we expect a commutative extension, this time isomporphic to $\mathbb{Z}_{40}$.

The error rates during 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 than with $\mathbb{Z}_{10}$, 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, $19 + 1 \doteq 19+(\text{half}+\text{half})$ outputs correctly 0, however $(19+\text{half})+\text{half}$ does not output 0.

3.1.4 Infinite group $\mathbb{Z}$

The last cyclic group experiment was with the infinite group $(\mathbb{Z}, +)$.

Elements: Integers $\cdots -2, -1, 0, 1, 2, \ldots$

Operations: Composition is classic addition, inverse is - and unit is 0.

Grounding: As themselves in $\mathbb{R}^1$

Extension: Using the equation $h+h = 1$. Because of parity, there is no solution in $\mathbb{Z}$.

Notes: The extension should be also isomorphic to $\mathbb{Z}$

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.

For the extension $h + h = 1$, there is only one intuitive solution: $\frac{1}{2}$. 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 $h + h = 1$, 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[1]. Although this sequence is not unique, all such sequences have the same length. For each permutation σ we define $\text{sgn}(\sigma) = (-1)^l$ where l is this number of transpositions. It also holds that $\text{sgn}(\sigma_1 \circ \sigma_2) = \text{sgn}(\sigma_1)\text{sgn}(\sigma_2)$. Therefore $\text{sgn}(\sigma \circ \sigma) = 1$. (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. $\text{sgn}((0, 1)) = -1$, therefore we know that h can not be in the original group.

Since the symmetric groups are not commutative, we expect this extension to be infinite (see the note after Theorem 5).

3.2.1 S_4 with a basic grounding

S_4 is the group of permutations of 4 elements.

Elements: Permutations of a set of 4 elements. We will denote the set as $\{0, 1, 2, 3\}$ and each permutation as (a, b, c, d), where $\{a, b, c, d\} = \{0, 1, 2, 3\}$

Operations: Composition is the chaining of permutations, the inverse of a can be for example obtained by writing a as a sequence of transpositions and reversing their order. The unit is the identity permutation $(0, 1, 2, 3)$.

Grounding: Basic grounding means that we encode a permutation (a, b, c, d) as $(a, b, c, d)^\top \in \mathbb{R}^4$.

Extension: Using the equation $h \circ h = (1, 0, 2, 3)$. As mentioned above, this has no solution in S_4.

[1] A transposition is the swap of two elements.

Notes:

1. We expect this extension to be infinite, therefore we will not test it all. However, h^4 should be $(0, 1, 2, 3)$, which we can use to verify the correctness of the extension.
2. The grounding used here is very short and simple. However, it has a disadvantage in the fact 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.

However, as we see in table 3.11, generating even a small subgroup results in a failure. We expect that $h^4 = e$, but unfortunately we were never able to get this equation to hold with $h \circ (h \circ (h \circ h)))$. However, because $(h \circ h) \circ (h \circ h)$ yielded reasonable results, we suspect that is the problem is with associativity.

3.2.2 S_4 with matrix grounding

To fix the problem with the bias of the loss function we change the grounding to the matrix representation.

Elements: As above

Operations: As above

Grounding: A permutation (a, b, c, d) is first encoded as a 4x4 matrix that has ones at the coordinates $(1, a + 1), (2, b + 1), (3, c + 1), (4, d + 1)$ and zeros elsewhere. This matrix is then "flattened" into a vector in $\mathbb{R}^{16}$.

Extension: As above, $h \circ h = (1, 0, 2, 3)$.

Notes:

1. This grounding erases the loss function bias, because the mean squared difference between any two permutations now only depends on the number of elements switched.
2. Composition of the elements corresponds with matrix multiplication of their groundings, i.e. $\mathcal{G}(p \circ q) = \mathcal{G}(p)\mathcal{G}(q)$.

With this grounding, we might encounter a problem with the extension. Assuming the neural network for composition perfectly mimics the matrix multiplication, there is no h in $\mathbb{R}^{4x4}$ that would solve the equation. We can show this using diagonalisation. We can write any (invertible) matrix A as the product SDS^{-1} where D is diagonal and S is orthogonal, where this decomposition is unique (Hladík [2019]). From this uniqueness follows that if $B^2 = A$ then B

is $S\sqrt{D}S^{-1}$ in its diagonalised form, where $\sqrt{D}$ is a diagonal matrix with the elements being the square roots of the elements of D.

We will look at the diagonalisation of the matrix grounding of the element $(1, 0, 2, 3)$.

$$\mathcal{G}(1,0,2,3) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -\frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{pmatrix}$$

Unfortunately, D has negative number in it. Therefore $\sqrt{D}$ does not exist in $\mathbb{R}^{4x4}$. Therefore if we are using this grounding we must hope that the optimization process actually avoids the perfect solution for matrix multiplication. This had been fortunately avoided, as seen in the table 3.14.

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 (an identity matrix). This is a consequence of the networks not being forced to produce $S^m \to S$ functions, as discussed in Section 2.1.

As a consequence, 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. This is probably the reason for the inverse having such a bad error rate.

The learned extension element h is shown in table 3.14 along with parts of the generated subgroup. Just like with basic grounding, $h \circ h$ shows promise, but h^4 breaks down.

Figure 3.1: Error rate for the learning of composition, inverse and unit in $\mathbb{Z}_{10}$ on the testing data. Testing data percentage is 10%. Each epoch describes one optimization step.

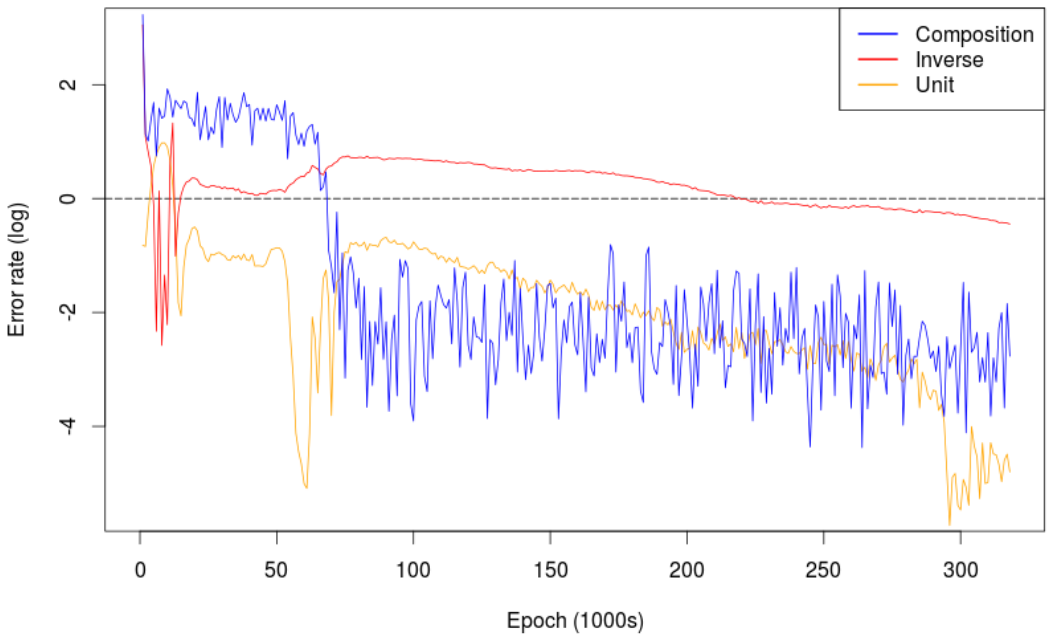


Figure 3.2: Different values of "half" found during different runs. The first run (red) was very peculiar since it had over 2 700 000 epochs, but this representation settled already around epoch 300 000. It is not clear how this interacts with the rest of the elements.

-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. These results are from 10% testing data. 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.

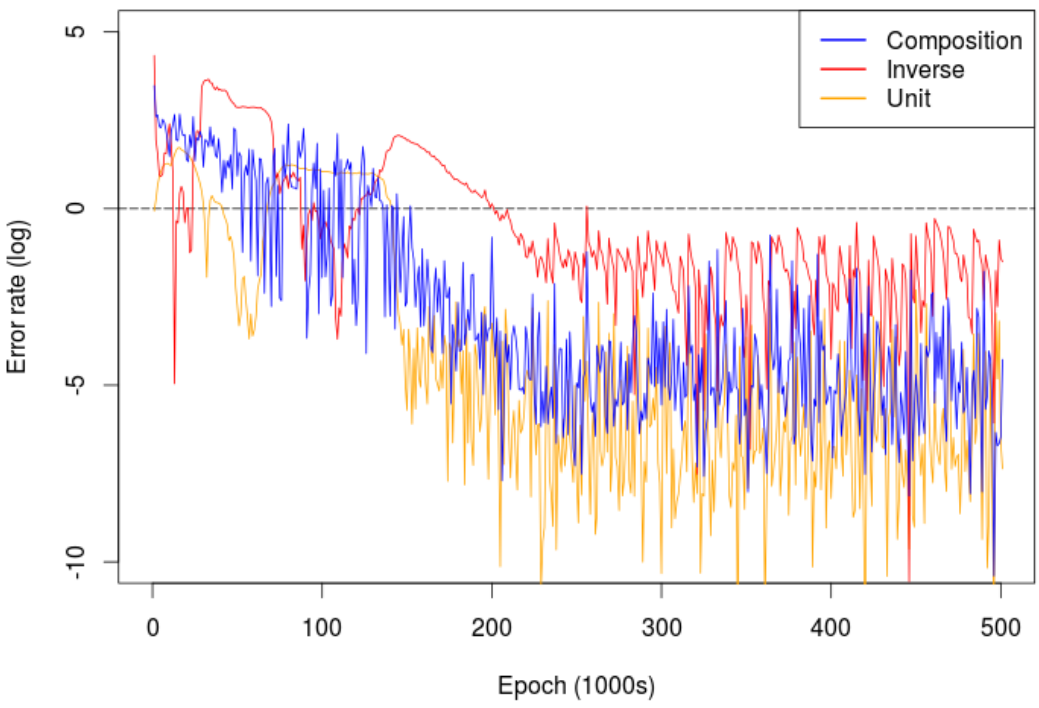


Figure 3.5: Values for the half in different $\mathbb{Z}_{20}$ runs. Once again, the reason for why the red one is so different is unknown. The tendency towards 0.5 instead of 10.5 can be attributed to the fact that the initial parameters have been restricted to avoid overflow errors.

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.

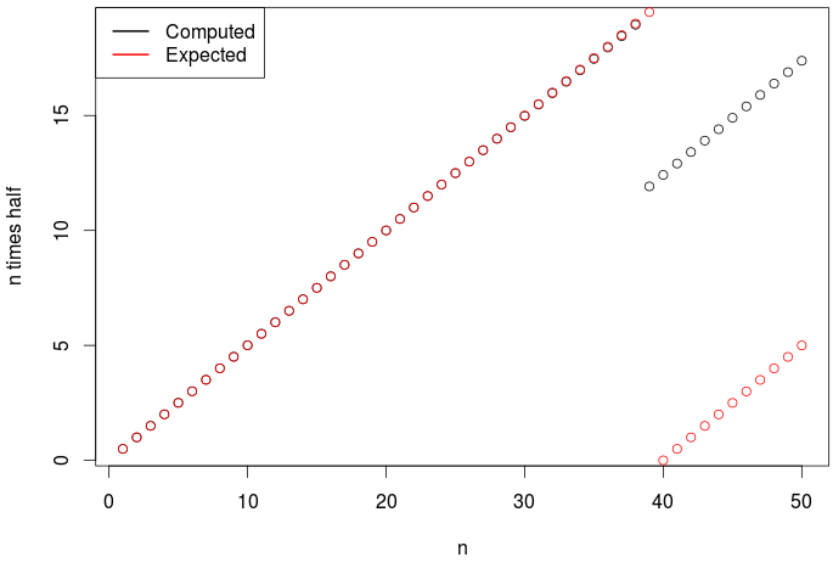


Figure 3.7: $\mathbb{Z}$ as an infinite group. Because of limitations of the software, we only trained on the interval $[-10000, 10000]$ (hence the trained group is not really infinite). There was no testing set, but the network seemed to generalize well even outside of the training interval. Examples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

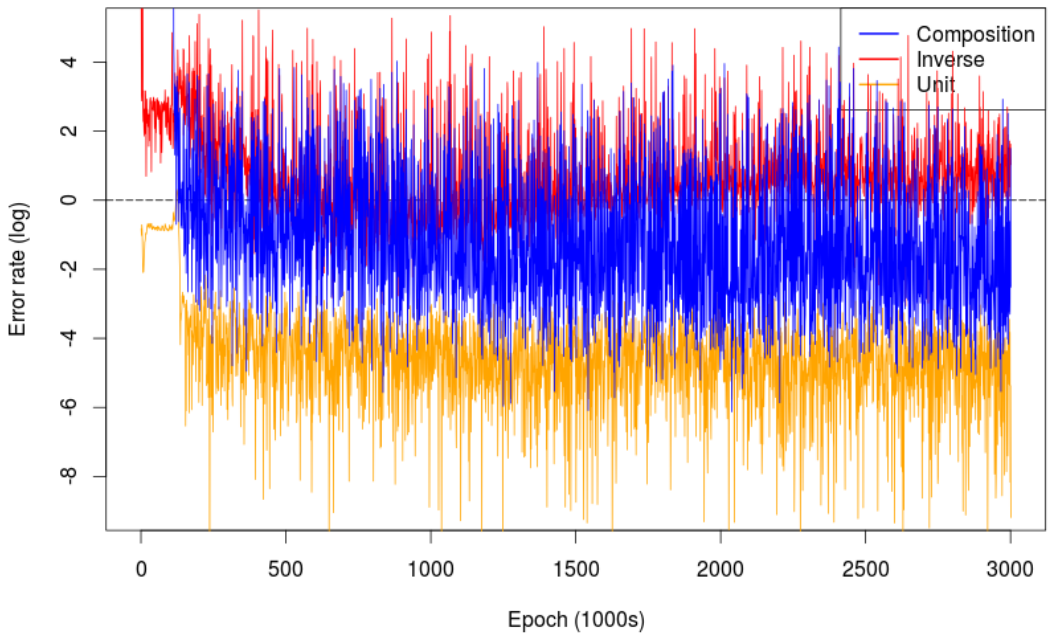


Figure 3.8: One run of learning the S_4 with the basic grounding. This graph shows error rates on 10% testing data. We see that the training is expectedly much slower than in the cyclic groups.

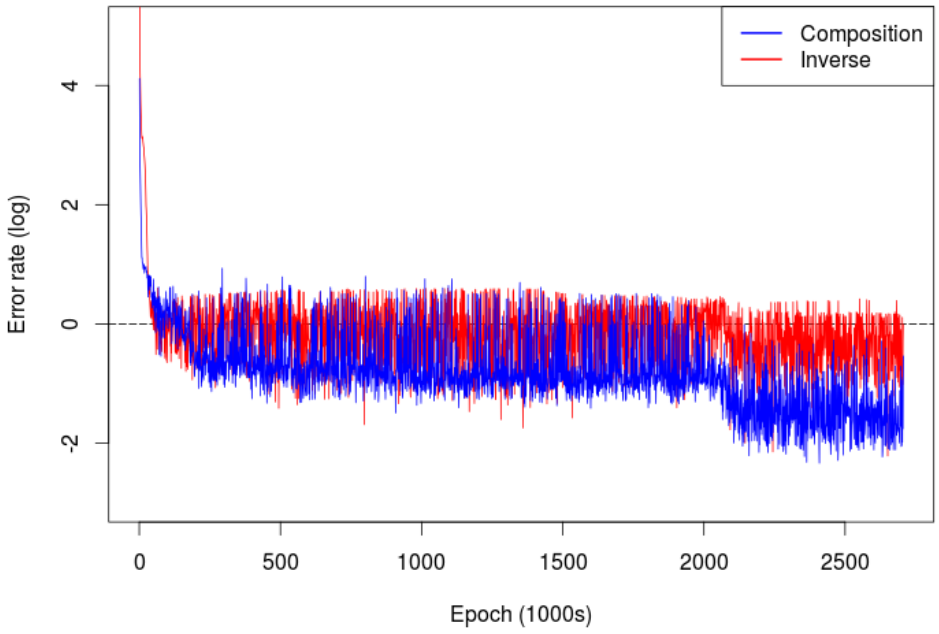


Figure 3.9: Units learned in S_4 with basic grounding. The expected output was $[0, 1, 2, 3]$. As we see, they are very precise.

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 testing data percentage is 5% The composition is very successful, but the inverse is not.

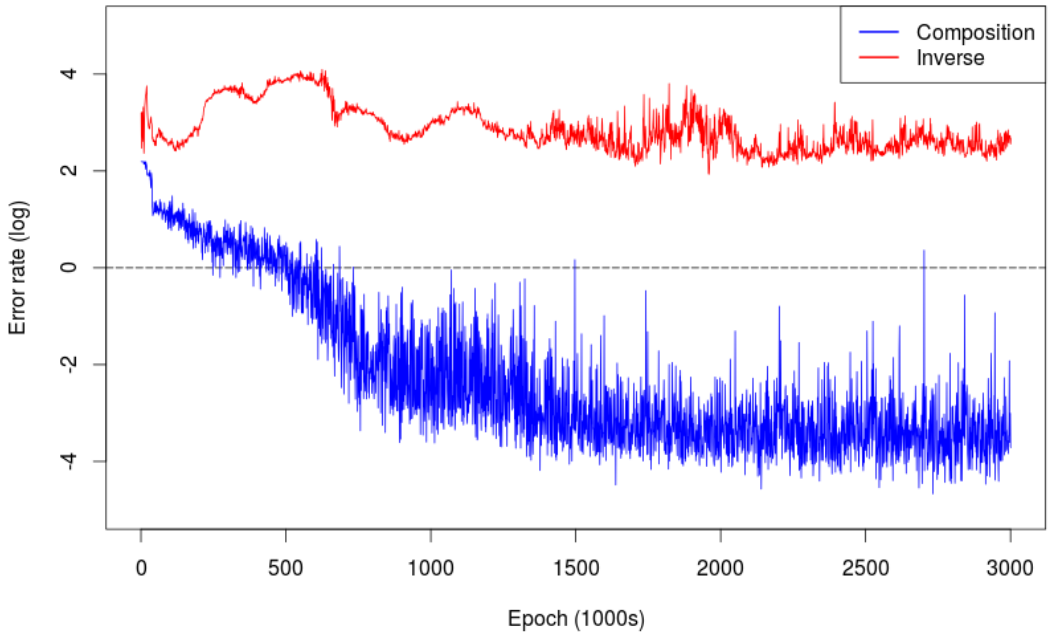


Figure 3.13: Units found in S_4 with matrix representation. An identity matrix was expected. Places where 1 was expected are blue, black ones are for 0.

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 related research

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[1] (e.g. Zhang and Sabuncu [2018]).

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 or even make some extensions impossible. 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 would need to use a self-found representation of elements. Representing various entities in vector spaces is a very active area of research. 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 modify the loss function to incur some penalty for the constants (and maybe functions) that are too far from the given

[1] Cross entropy of a non-deterministic function $f : \mathbb{R}^n \to \{0, 1\}$ is defined as $-y \log \widehat{y} - (1 - y) \log(1 - \widehat{y})$ where y is the probability that $f(x) = 1$, while $\widehat{y}$ is the computed probability that $f(x) = 1$. Cross-entropy is the lowest when $\widehat{y} = y$ for any given y. If f is deterministic, then $y = f(x)$.

elements. Other 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 end in an equilibrium where the "found" 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 of the axiom loss function, 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, still only speculation.

Related research and comparison

At the time of the writing of this thesis, we are unaware of any other model building frameworks that utilize neural networks. In Claessen and Sorensson [2003] they describe a model builder called Paradox, which determines whether or not is a given theory satisfiable. The program is very fast on smaller models, but it struggles with larger structures. It transforms all axioms into clauses[2] that it then tries to satisfy utilizing a SAT solver. Since SAT is known to be a hard problem, especially on large sets, efficiency of this approach is very dependent on the universe size. For example Paradox is unable to learn the permutation group S_5 from the axiomatized multiplication table[3].

[2]A clause is a formula of the form $\varphi_1 \vee \varphi_2 \cdots \vee \varphi_n$ where φ_i are atomic.

[3]Axiomatized multiplication table is a set of axioms that describe constants $c_{(a,b,c,d)}$ as labels for each element. There are axioms that say that the elements are different: $c_{(a,b,c,d)} \neq c_{(k,l,m,n)}$ if (a, b, c, d) is different from (k, l, m, n). Also, there are axioms that describe the multiplication: $c_{p_1} \circ c_{p_2} = c_{p_1 \circ p_2}$

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.

K. Claessen and N. Sorensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

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`.

Petr Hájek. Metamathematics of fuzzy logic. 6, 1998. doi: 10.2307/421060.

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*, 2017. 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.

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.

S.J. Taylor. *Introduction to measure and integration*. Cambridge University Press, 1973. ISBN 978-0-521-09804-5.

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.