MASTER THESIS

Martin Smolík

Artificial Intelligence and Automated Reasoning

Department of Algebra

Prague 2019

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

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

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

Dedication.

Title: Artificial Intelligence and Automated Reasoning

Author: Martin Smolík

Department: Department of Algebra

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

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

Contents

Introduction

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

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

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

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

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

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

1. Background

In this section we will discuss the fundamentals of both model theory and neural networks. Since this thesis seeks to unify two vastly different fields of mathematics, 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 $t_1, \ldots t_n$ are terms then $f(t_1, \ldots t_n)$ 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 $t_1, \ldots t_n$ are terms then $R(t_1, \ldots t_n)$ 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 $x_1, \ldots x_n$ are all the free variables of φ then we usually write φ as $\varphi(x_1, \ldots x_n)$ to show that it has free variables. A **sentence** is a formula that has no free variables. A **free formula** is a formula that has only free variables.*

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

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

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

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

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

1. *Value of v_i is $\phi(v_i)$*

2. *Value of c where c is a constant is $\mathcal{I}(c)$*

3. *If $t_1 \ldots t_m$ are terms with assigned values $s_1 \ldots s_m$ and f is a function then $f(t_1, \ldots t_m)$ is assigned the value $\mathcal{I}(f)(s_1, \ldots s_m)$.*

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

1. *If the subformula is of the form $t_1(x_1, \ldots x_m) = t_2(x_1, \ldots x_m)$ 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 $R(t_1, \ldots t_r)$ where $t_1, \ldots t_r$ are terms with assigned values $s_1, \ldots s_r)$ then it is true if and only if $\mathcal{I}(R)(s_1, \ldots s_r)$ 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 $(\forall y)\psi(y, x_1, \ldots x_m)$ then it is true if and only if for any extension ϕ' of ϕ that includes also assigns y does $\psi(y, x_1, \ldots x_m)$ hold (with assignment ϕ').*

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

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

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

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

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

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

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

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

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

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

1.1.2 Skolemization

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

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

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

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

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

---

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

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

- $(Qx\ \varphi) \lor \psi$ is equivalent to $Qx\ (\varphi \lor \psi)$ 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 $\neg\varphi \lor \psi$ we are done. $\square$

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

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

Let φ be a formula that has the form

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

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

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

We say that f_y is the Skolem function of y.

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

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

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

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

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

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 separate 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 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[3] $\varphi(x) :$ "$t_1(x) = t_2(x)$" is an **equation** in $\mathcal{S}$. If $\varphi(x)$ holds for some x, we say that that x is the **solution** of φ. If the set of solutions of φ is not S, we say that the equation is **non-zero**.*

Definition 18. *Let $s \in S'$ be an element. If is there exists a non-zero equation $\varphi(x)$ in $\mathcal{S}$ for which s is a solution then s is **algebraic**[4]. If there is no such equation, then s is **transcendental**.*

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.

---

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

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

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[5]*

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 $\exists e(\forall a \ e \cdot a = a \cdot e = a \wedge \forall a \exists b \ a \cdot b = b \cdot a = e)$[6].

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 $\{t(b_1,\ldots b_n)|b_1,\ldots b_n \in B, t \text{ is a term}\}$. 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:*

- *Elements of $G[x]$ are formal strings of the form $g_1 x^{n_1} g_2 x^{n_2} \ldots g_k x^{n_k} g_{k+1}$ 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 $g_i g_{i+1}$ we replace it with $(g_i \cdot g_{i+1})$.*

---

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

[6]Once again we use simplified notation.

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 $g = g_1 x^{n_1} g_2 x^{n_2} \ldots g_k x^{n_k} g_{k+1}$ then $g^{-1} = g_{k+1}^{-1} x^{-n_k} \ldots x^{-n_1} g_1^{-1}$. We can easily verify that $g \cdot g^{-1}$ is indeed equal to e.*

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

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

Proof. If we apply simplifications from definition 23, t has the form $g_1 x^{n_1} g_2 x^{n_2} \ldots g_k x^{n_k} g_{k+1}$. This modification of φ still has the same solutions.

This way t can be seen as an element of $G[x]$. Let us then define $B = \{gtg^{-1} | g \in G[x]\}$. $\langle B \rangle$ is a normal subgroup of $G[x]$. We immediately see that $gbg^{-1} \in \langle B \rangle$, (even $gbg^{-1} \in B$) for all $b \in B, g \in G[x]$. Now if $b = b_1 b_2 \ldots b_n$ where $b_i \in B$ then

$$gbg^{-1} = gb_1 b_2 \ldots b_n g^{-1} = gb_1 g^{-1} gb_2 g^{-1} \ldots gb_n g^{-1}.$$

We define $b_i' = gb_i g^{-1}$ for $i \in \{1, \ldots, n\}$. Now

$$gbg^{-1} = (gb_1 g^{-1})(gb_2 g^{-1}) \ldots (gb_n g^{-1}) = b_1' b_2' \ldots b_n'.$$

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

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

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

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

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

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

In order to describe some of the extensions of groups that are obtained in this thesis, we need a new type of construction: a **semidirect product**. There are two different versions of this construction, we will use the so-called outer one.

Definition 24. *Let H, K be groups and $\varphi : K \to Aut(H)$[7] a homomorphism (for the sake of clarity we will refer to $\varphi(k)$ as φ_k). Then the semidirect product $H \rtimes_\varphi K$ is defined as such:*

- *Universe is the set $H \times K$.*

- *$(h_1, k_1) \cdot (h_2, k_2)$ is defined as $(h_1 \varphi_{k_1}(h_2), k_1 k_2)$*

- *$(h, k)^{-1}$ is defined as $(\varphi_k^{-1}(h^{-1}), k^{-1})$*

- *$e = (e_H, e_K)$*

Proof that this is indeed a group can be found in Drápal [2000].

1.2 Neural networks

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

1.2.1 What is a neural network

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

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

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

These networks are called neural, because they are inspired by biological neurons. In each layer the singular neurons are scalar functions from each element of input to each element of the output. These elements are also commonly referred

---

[7]For the definition of homomorphisms and automorphisms of groups, refer to Drápal [2000]. In short, a homomorphism is a mapping $f : G \to H$ such that $f(g_1 g_2) = f(g_1)f(g_2)$ and automorphism is an invertible homomorphism $G \to G$.

to as *nodes*. When we add together all neurons that lead to a node of the output, we get a vector to scalar function, that is the building block of a layer. This is better illustrated in Figure 1.1.

Figure 1.1: Neurons in a layer

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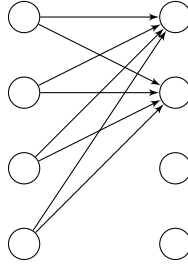
The simplest neural networks are linear models, e.g. linear regression. In these models each neuron is a simple multiplication by a parameter. These models however have many drawbacks, for instance that they can only model linear functions. That is because the layers are linear functions, therefore several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

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

One of the most widely used methods to break the linearity is the use of so-called activation functions. Activation function is a non-linear $\mathbb{R} \to \mathbb{R}$ function that we perform on each node to know if it "activates". There are several functions that fulfill this purpose, each with their advantages and disadvantages. The simplest is *ReLU*: the **Re**ctified **L**inear **U**nit (Nair and Hinton [2010], Jarrett et al. [2009], Hahnloser et al. [2000]). It is defined as $ReLU(x) = max\{x, 0\}$. It is the most used activation function because it is the simplest function that also preserves most of what makes linear functions easy to work with (simple derivation). It has however some drawbacks that we will discuss later.

1.2.2   Example: XOR

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

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

$$J(\theta) = \frac{1}{n} \sum (\widehat{f}(\mathbf{x}; \theta) - f(\mathbf{x}))^2$$
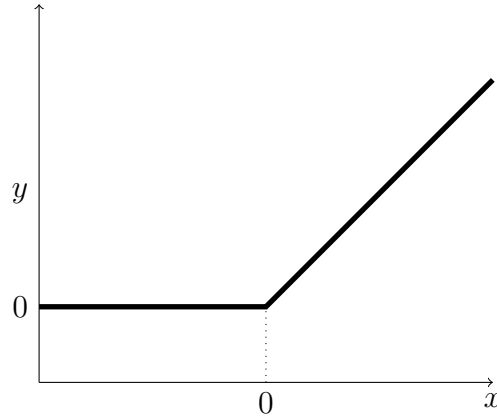
where f is the function we try to estimate and $\widehat{f}$ is the learned function. Since we are now using linear regression, $\widehat{f}$ has the form $\widehat{f}(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^\top \mathbf{w} + b$ where $\mathbf{w}$ is called the weight vector and b is bias (in this case scalar). Using the least squares method we get $\mathbf{w} = (0, 0)^\top$ and $b = \frac{1}{2}$. This is obviously not a good fit as it just outputs $\frac{1}{2}$ regardless of the input.

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

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

where

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

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

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

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

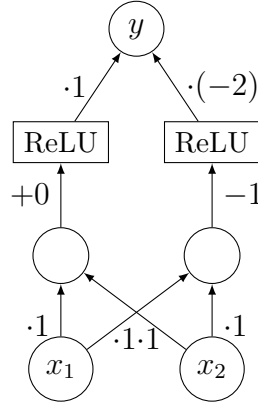Let us see what happens when we apply the input data to it in matrix form

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

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

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

Figure 1.3: XOR network

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

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

Now we apply ReLU:

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

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

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

which is exactly the desired output.

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

1.2.3 Other activation functions

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

Before the introduction of ReLU the default activation functions were sigmoid

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

Figure 1.4: Sigmoid and tanh functions
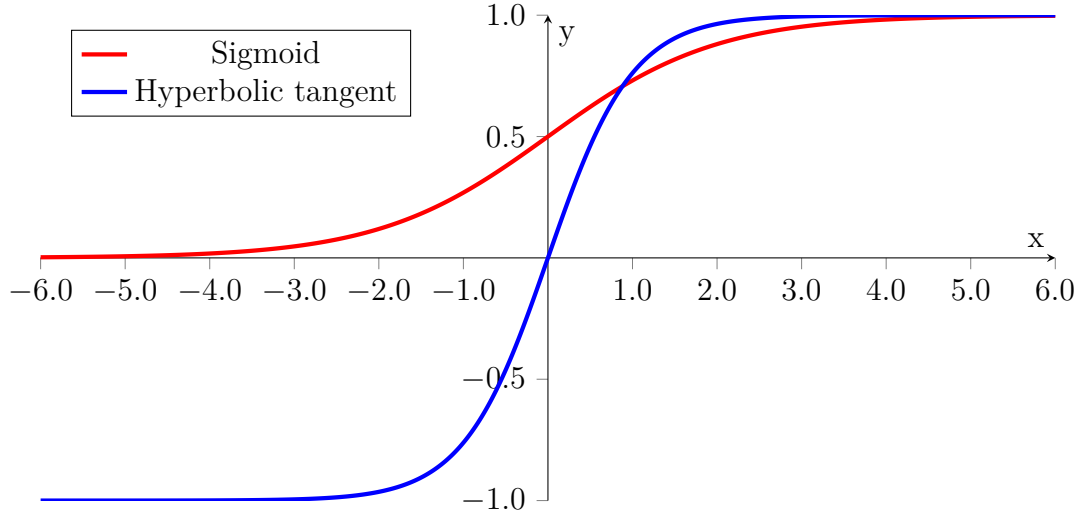
and hyperbolic tangent

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

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

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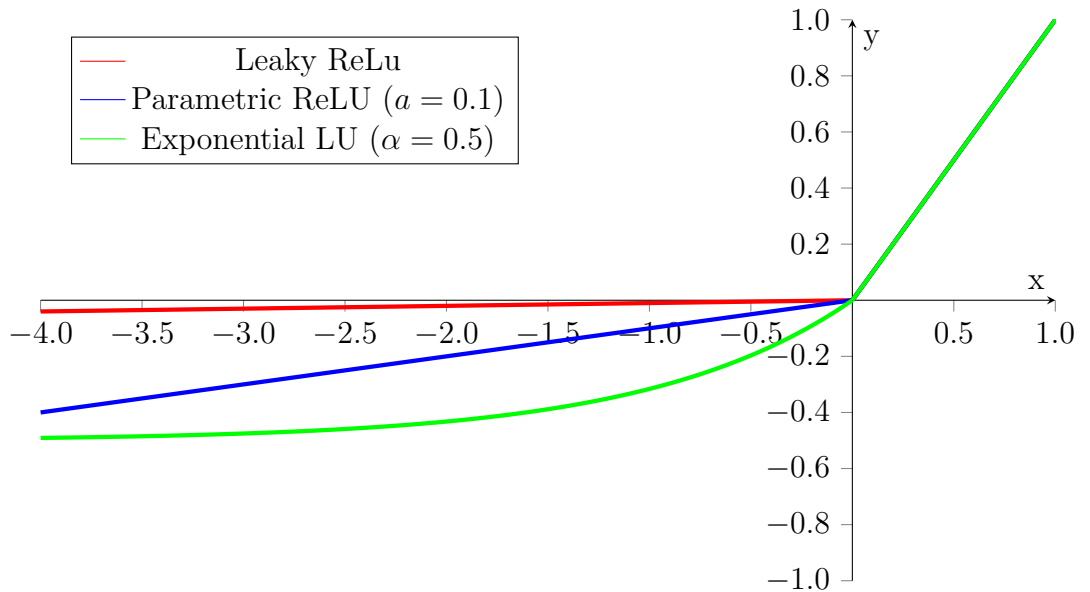
There had been also a number of attempts to fix the "dead" node problem with ReLU. Most of them try to preserve the piecewise linearity, since it is the ReLU's strongest advantage. Among examples are Leaky ReLU (Maas et al. [2013]), Parametric ReLU (He et al. [2015]) or Exponetial LU (Clevert et al.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

They are defined as follows:

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

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

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

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

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

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

1.2.4   Learning and optimization

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

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

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

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

With neural networks we use the general form

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

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

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

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

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

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

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

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

1.2.5 Back-propagation

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

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

Back-propagation is taking this loss and propagating it backwards to produce a gradient.

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

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

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

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

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

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

As we can see, if Jacobians of vector to vector functions and gradients of vector to scalar or scalar to scalar functions in our network are known, we can compute gradients with respect to any subset of inputs starting from any point in our network. We can accomplish this by applying the chain rule recursively (and taking the rest of the inputs as constants).

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

1.2.6   Adam optimizer

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

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

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

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

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

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

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

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

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

Figure 1.6: An example of the symbol to symbol approach. Note that $f^{(3)}$ here can also be the loss function
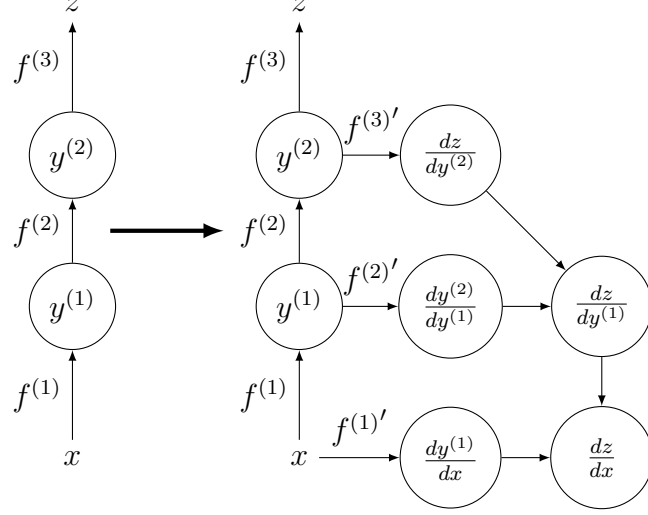
Algorithm 2 Adam optimizer. This algorithm is exactly like it can be found in the original paper. g_t^2 denotes element-wise square. All vector operations are elemet-wise. f_t represents the loss function f realized over the training batch in the step t.

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

---

2. Neural modelling

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

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

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

1. *$\mathcal{G}(c) \in \mathbb{R}^n$ for every constant c.*

2. *$\mathcal{G}(f)$ where f is a function symbol is a map $\mathbb{R}^{nm} \to \mathbb{R}^n$ where m is the arity of f.*

3. *$\mathcal{G}(r)$ where r is a relation symbol is a function $\mathbb{R}^{nm} \to [0, 1]$ where m is the arity of r.*

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

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

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

There are also rules for logic on formulas:

$$\mathcal{G}(\neg r(t_1, \ldots, t_n)) = 1 - \mathcal{G}(r(t_1, \ldots, t_n))$$

$$\mathcal{G}(\phi_1, \vee \cdots \vee \phi_n) = \mu(\mathcal{G}(\phi_1), \ldots, \mathcal{G}(\phi_n))$$

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

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

---

[1] A type of neural networks

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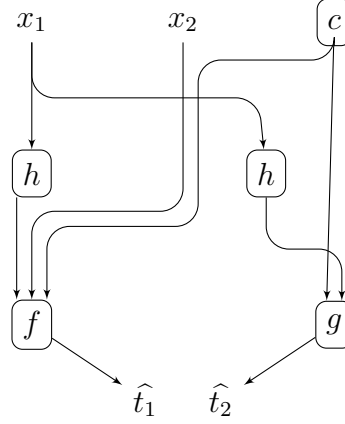
2.1 Building a neural model

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

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

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

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

In the case that the axiom has more such equalities (or negations of them) we do some adjustments. Loss function of negation of a subformula should be computable from its own loss, e.g.

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

If φ_1, φ_2 are subformulas, loss of $\varphi_1 \vee \varphi_2$ will be e.g. $\min\{J_{\varphi_1}, J_{\varphi_2}\}$ and loss of $\varphi_1 \wedge \varphi_2$ will be e.g. $J_{\varphi_1} + J_{\varphi_2}$. However, these operations are for now just a speculation and further research is warranted. For example $\vee$ could use a function with a gradient that depends on both parts of the equation, while $\wedge$ might perform better using a different method of aggreggation (e.g. $2\sqrt{J_{\varphi_1}^2 + J_{\varphi_2}^2}$).

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

Figure 2.2: Network used for learning an extension defined by the equation $t_1(x) = t_2(x)$. Note that only c_{ex} is optimized, everything else is treated as a function.
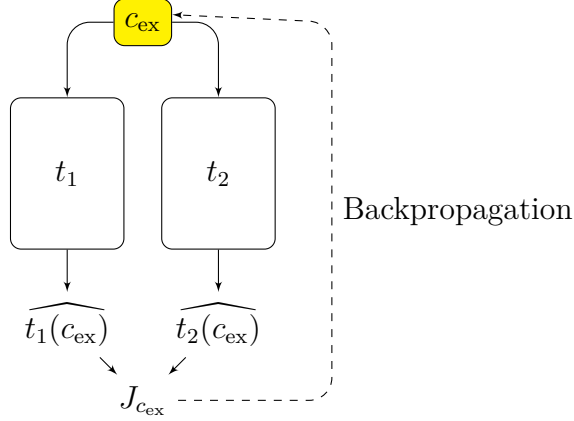
After computing the loss for each axiom, we aggregate them like with $\wedge$ and use backpropagation from there. This is, however, very costly with regards to computation time. Therefore in our experiments we used a different, simplified approach.

2.2 Neural model extension

Another big part of this thesis is model extension. In subsection 1.1.3 we have discussed which type of extensions are we interested in.

Assume we have already built a model $\mathcal{G}$ for a structure $\mathcal{S}$ and we want to extend it to include a solution of an equation $t_1(x) = t_2(x)^2$. Note that t_1 and t_2 are allowed to have parameters from $\mathcal{S}$. Since we assume that the representation is already settled, the usage of parameters is 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 $t_1(c_{\text{ex}}) = t_2(c_{xtex})$. We already have a framework to find such a constant (Figure 2.2). However, we need to remove any possible penalties for straying away from the representations of S.

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

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

2.3 Implementation for groups

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

---

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

Figure 2.3: Learning of composition. An incomplete lookup table is used to determine the loss. x_1, x_2 is a randomly selected double.
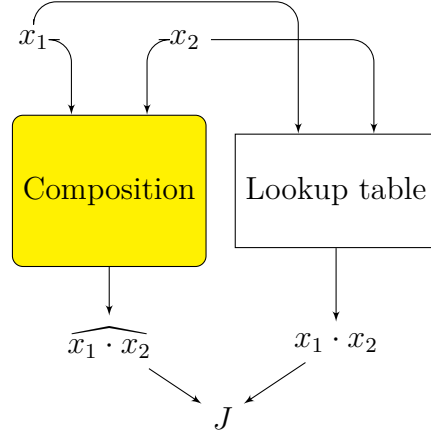
There are many different types of groups, that have vastly different complexity levels. We have built models for the cyclic groups (namely $\mathbb{Z}_{10}$ and $\mathbb{Z}_{20}$) and symmetric groups (S_3 and S_4). These groups have been selected for being the simplest and the most complex finite groups respectively.

To better see if the computer is able to represent these groups correctly, we have opted to use a lookup table for the learning of $\cdot$. A lookup table is a table of pre-computed values for each double of the elements. This runs counter to the ideas described above, but is essential for human interpretability of the results. To demonstrate the ability of the network to generalize, and to a degree "understand" the composition function, several entries (up to 10%) had been selected as testing data - they are never used during the optimization process.

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

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

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

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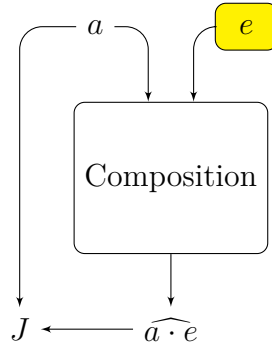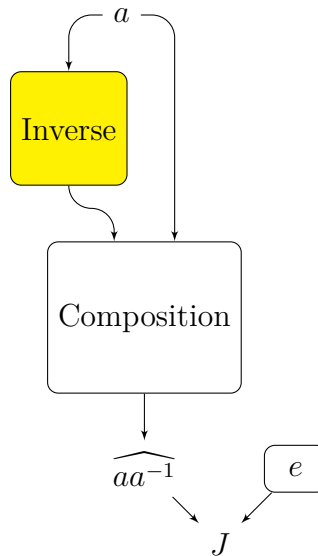

Figure 2.5: Learning of the inverse. Here we use the learned unit and composition. We use the axiom $a^{-1}a = e$. The dual $aa^{-1} = e$ is disregarded.

3. Results

Conclusion

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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