Deep Reasoning: Hardware Accelerated Artificial Intelligence

Student project

for the

Bachelor of Engineering

from the Course of Studies IT-Automotive

at the Cooperative State University Baden-Württemberg Stuttgart

by

Phillip Lippe

4. June 2018

Time of Project	16. October 2017 to 4. June 2018
Student ID, Course	9045534, IT-Automotive 2015
Supervisor	Prof. Dr. rer. nat. Stephan Schulz

Abstract

An automated theorem prover is a computer program that shows that a statement, called the conjecture, can be logically implied by a set of axioms. These systems are applied for deducting mathematical proofs, but also for the verification of software and hardware given a proper formulation of the problem as axioms and conjecture. Most modern theorem provers like Vampire and E rely on first order logic where implications of clauses are computed in order to generate the empty clause. When to use which clauses for interference is a key aspect of automated theorem proving and crucial for the system's performance.

For this task of clause selection, a new heuristic based on artificial neural networks is presented in this project, as deep learning techniques have currently demonstrated great success in several domains. First, the clauses are embedded by a vocabulary that is learned during training and shares features among similar symbols. The following compression to a fixed size is implemented by combining the concepts of dilated convolutions and a dense block to ensure a wide context understanding. Next, the features of a clause that should be evaluated are concatenated with those from the negated conjecture, and processed by a small classifier network. Moreover, a knowledge base extraction for initial clauses is developed and used as an additional input to the classifier. The training is optimized by extensive negative mining and a novel loss function. The heuristic is integrated and tested in the state of the art theorem prover E. In experiments, the network correctly classifies over 90% of all clauses in a test dataset of solved proofs if they are either likely to be useful for the proof or not. As this project is one of the first in this area, only one from a sample of 25 new proofs can be solved with the developed technique leaving open challenges for future approaches.

Contents

List of Figures

List of Equations

List of abbreviations

ATP	Automated Theorem Proving
BCE	Binary Cross Entropy
CNF	Conjunctive normal form
CNN	Convolutional Neural Network
FIFO	First-In/First-Out
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit [8]
LSTM	Long Short-Term Memory [21]
ReLU	Rectified Linear Unit [47]
RNN	Recurrent Neural Network
TPTP	Thousands of Problems for Theorem Provers library [71]

List of Tables

1. Introduction

1.1. Motivation

To proof a hypothesis in mathematics, a sequence of logical statements has to be found that explains why the given statement is true. The proof relies on a set of self-evident statements, known as axioms, along with accepted rules of inference. To solve the task of finding a proof, a certain ability of formal reasoning is needed. Humans have demonstrated their skills several times, as at least every mathematical proof before the 20th century was done by hand. Today, computers are superior to humans on many automatable tasks. So, this raises the questions if machines could also deduct proofs, especially the ones that no one could solve yet?

Computer programs dealing with this task are known as automated theorem provers. In comparison to humans, such systems can compute implications much faster and therefore test more possible proof paths. One of the most popular successes was settling of the Robbins problem by the automated theorem proving system EQP in October 1996 [41]. The problem was based on the conjecture that a particular group of axioms form a basis for Boolean algebra proposed in 1933, but neither the author, Herbert Robbins, nor anyone else could prove this statement until the solution by EQP. The proof search by EQP took about 8 days on an RS/6000 processor [31].

But not only for mathematics is automated theorem proving relevant. A key modern application of theorem provers is in the verification of hardware and software designs [29]. Digital computers represent abstract discrete state machines, and can naturally be formalized in mathematical logic. Thus, an automated theorem prover is able to find bugs and proof the correctness of complex software and hardware architectures given the formal descriptions of such systems. Many well known software bugs resulted in costly consequences including various space and military aircraft crashes underlining the viability of verification systems. Further applications of automated theorem prover involve i.e. geometric proofs and artificial intelligence [46].

The language in which modern automated theorem provers operate often is first order logic, like Vampire [55, 32] and E [60, 61]. The advantage of first order logic is the good compromise between expressiveness (how easily can computable problems be encoded in this language) and automation of the proof search. To transform a problem into first order logic, the axioms are converted into a set of clauses that also includes the negation of the given conjecture or hypothesis. The proof idea is to apply inference rules until either the emtpy clause is derived witnessing the correctness of the conjecture, or no new clause can be generated. When to compute which implications is a key aspect of automated theorem proving and mostly organized by a variation of the given clause algorithm. Its concept is to split the clauses into a processed set, in which all possible implications are computed, and an unprocessed set from which one clause is processed at each iteration. The choice of the chronological order, in which the clauses are processed, is crucial for the provers' performance.

This task is also known as the clause or premise selection problem defined by:

> “(Premise selection problem). Given a large set of premises P, an ATP system A with given resource limits, and a new conjecture C, predict those premises from P that will most likely lead to an automatically constructed proof of C by A” [4].

Classic approaches like selecting the oldest clause first (FIFO: First-In/First-Out) or preferring small clauses (Clauseweight) are more likely to find proofs due to the greatly amount of clauses they can process. - SATZ VON INTRO AUS LIPPE/SCHULZ PAPER - However, the function and predicate symbols are uninterpreted and dependencies between different clauses are rarely considered as it constitutes a very complex problem. Hence, the best heuristic depends on the problem but is not obvious even to human experts [7].

In the last years, algorithms in the area of machine learning have successfully proved their capabilities to model such relationships which are too complex for analysis [17]. Especially deep learning revolutionized the concepts of domains like Computer Vision [33, 54], Natural Language Processing [16, 58] and Planning [65, 75]. Most approaches in combining machine learning with automated theorem proving concentrated on modifying an heuristic regarding previous proofs or shallow feature selection [7, 11, 12]. However, deep learning has reached a state where even an evaluation of clauses with end-to-end learning is reasonably practicable as experiments by [4, 39] have shown. Therefore, this project investigates a deeper assessment on how to optimize neural networks for the clause selection problem.

1.2. Proposed Approach

For this project, the basic concept of the deep learning heuristic is based on [39]. The inputs for the evaluation are the clause that should be evaluated, and the negated conjecture of the proof. As neural networks mostly run on fixed size data, both need to be embedded into a feature vector. The network that is responsible for this step is called the embedding network and is one of the major parts of the architecture. After the clause and the negated conjecture are embedded into a fixed size feature vector, both are concatenated and processed by a second model, the combiner network. The result is a single number between 0 and 1 constituting whether the clause is likely to be useful (0) or not (1) for the proof attempt.

This concept has many choice points and open challenges that need to be addressed. Therefore, this project concentrates on four main aspects of the architecture and proposes new approaches for a better clause heuristic:

Vocabulary The first step of evaluating a clause is representing its symbols as features so that it can be processed by a neural network. For this, techniques from the domain of Natural Language Processing are used where the symbol embeddings are learned in a vocabulary within the standard training process [16]. However, symbols can share some characteristics (like if it is a constant, a variable or a parentheses) so that also their features should be similar. Therefore, this project proposes an approach where the feature embedding of each symbol is split into a part for its name and another for its arity, or rather its type, which is shared among similar words.

Embedding network Embedding a clause might be the hardest problem in this architecture as a sequence of symbols with arbitrary length has to be compressed into a small, fixed size feature vector that must contain all information necessary for the evaluation. Different approaches like shallow convolutional and recurrent networks, but also deeper architectures like recursive Tree-LSTMs and WaveNet were already tested in related works [4, 39]. Models based on convolutions demonstrated the best results but suffered by a lack of context information of the clause. To address this problem, a new network architecture is proposed that reasonably increases the local view of each neuron and actively supports the combination of different filter sizes.

Combiner network The combiner network is the last step of the heuristic and results in a final evaluation of the clause. The inputs for this model are the feature vectors of the negated conjecture and the clause that should be evaluated. However, these inputs are not sufficient for an optimal clause selection. Therefore, a third input is developed in this projects that constitutes a knowledge base extracted from the initial clauses of a proof. Based on this, the network can learn dependencies between clauses and compare the clause to be evaluated with possible others.

Performance measurement The last part deals with the question how to give the network a feedback on its predictions. Optimally, the heuristic would be applied in E and trained on its own proof searches. However, as neural networks need many iterations for training and proof attempts can take several minutes, this approach is not practicable. The alternative that is used in this project and also in [39], is training a network in a supervised manner on proofs that are deducted by classic heuristics. This guarantees a faster learning process and sufficient amount of examples.

Nevertheless, the challenge coming with this task is the significantly imbalanced data. The amount of examples for clauses that are not useful for a proof are up to 100 times higher than the one for useful clauses. Thus, a new technique is developed in this project combining extensive mining of hard examples with an adapted loss function that increases the loss for misclassifications of positive examples.

1.3. Outline

This report is structured as follows. The first section gives a brief introduction to the field of automated theorem proving and machine learning. A detailed discussion of the proposed approach is presented in section 3 and the corresponding experiments are described in section 4. The report concludes with an outlook on remaining open challenges and an overall summary.

2. Foundations

This section gives a brief introduction to the field of machine learning and theorem proving. The basics of first-order theorem proving are explained in the first subsection, followed by an introduction to E [60]. Next, a review of machine learning techniques is given, while the following subsection discusses artificial neural networks as a part of learning algorithms. As the last part of this section related works are presented and summarized.

2.1. First-Order Theorem Proving

The goal of an automated theorem prover (ATP) is to solve the question if a hypothesis H can be implied by a set axioms $A = \{A_1, A_2, ..., A_n\}$: $A \models H$ [59]. Mostly, the problem that should be solved has an application in other domains like mathematics. To process those with automated theorem provers, the basic rules of the domain need to be formalized into a set of axioms as well as the problem as a conjecture or hypothesis. First order logic has established as a standard for this formalization as automated theorem provers can deducts proofs with methods like resolution and superposition, and its syntax quite similar to word formulations of rules [49, 59]. If the ATP has found a proof, it can be used to solve the given problem in the real domain like proving a mathematical hypothesis. This concept is visualized in figure 1.

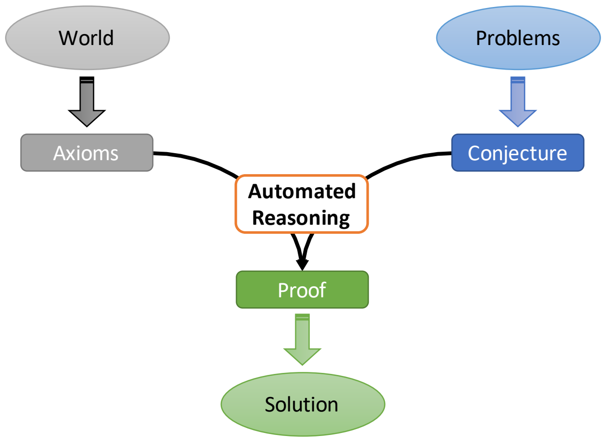


Figure 1: The concept of automated reasoning is to formalize a problem as a conjecture. Inspired by [72]

Today, automated theorem provers like Vampire [55, 32], Prover9 [43], SPASS [80] and E [60, 61] define the state of the art and are based on superposition and saturation subsuming other calculi like resolution, paramodulation and unfailing completion [62]. To introduce automated theorem provers, the first paragraph gives an short overview of first order logic with equality. The next section describes the general concept of saturation-based theorem provers, before E [60, 61] is presented in more detail in section 2.3.3.

2.1.1. First Order Logic with Equality

First order logic extends propositional logic to predicate logic where a predicate is a function of terms. Despite a proposition which is either true or false for the whole domain, the truth value of a predicate depends on its parameters. These parameters, or also called terms, can be constants, functions or variables. General statements that apply for all terms can be expressed by quantified variables like $\exists X$ (for at least one element) and $\forall X$ (for all elements) [32].

Overall, a signature for predicate logic is a triple of (P, F, V) where P defines the predicate symbols, F the functions with arbitrary arity, and V the variables. Functions and predicate symbols are uninterpreted so that they do not have any semantic value. However, first order logic can include equality which is interpreted as a congruence relation [49].

To clarify the syntax used in this report, the following definitions are stated:

Formulae A formulae specifies a logical statement and is defined recursively:

- Literals, also called elementary statements, are formulae
- Boolean combinations ($\vee$, $\wedge$, $\rightarrow$, $\leftrightarrow$)
- If F is a formula, $\forall X : F$ and $\exists X : F$ are formulae of formulae are formulae

An example for a formula is $\forall X : (\forall Y : ((odd(X) \wedge odd(Y)) \rightarrow X \not\simeq add(Y, 1)))$. The individual parts of the formula are explained in the next definitions.

Atom and Literal An atom is the smallest statement that has either the logical value true or false. In first order logic, a predicate with specified parameters is an atom. A literal constitutes a positive or negative atom. In the example, $odd(X)$ and $odd(Y)$ are both positive atoms while $X \not\simeq add(Y, 1)$ is a negative atom due to the negated equality. However, all three atoms are also literals.

Terms A term is a variable or a function with specified parameters. The parameters are recursively defined as terms and can be functions as well. Functions have different arities where a function with no parameters is called a constant. In the example above, X, Y, 1, $add(Y, 1)$ are terms where X and Y variables and 1 is a constant term, and $add(Y, 1)$ is a composite term with proper subterms 1 and Y.

Clauses A clause is a multiset that is written and interpreted as disjunctions of literals. All variables in a clause are implicitly universally quantified ($\forall X$). The previous example of the formulae can be rewritten as a clause: $X \not\simeq add(Y, 1) \vee odd(X) \vee odd(Y)$.

Equality Many conjectures based on practical applications involve equality. As equality has a high semantic value, it can be included as part of the logic itself and interpreted as a congruence relation. The properties of equality can be expressed by the congruence axioms for reflexivity, symmetry and transitivity [49]:

$$\begin{aligned} \rightarrow x \simeq x \quad &\text{(reflexivity)} \\ x \simeq y \rightarrow y \simeq x \quad &\text{(symmetry)} \\ x \simeq y \land y \simeq z \rightarrow x \simeq z \quad &\text{(transitivity)} \end{aligned} \tag{2.1}$$

The extended resolution of first-order logic with equality is called superposition and contains inference rules based on the congruence axioms.

2.1.2. Saturation-Based Theorem Proving

The first step for a first-order theorem prover is clausification. Therefore, the axioms A and hypothesis H as first-order predicate logic formulae are transformed into a clause set $\{C_1, C_2, ..., C_n\}$ in clause normal form (CNF) such that the set is unsatisfiable if and only if $A \models H$ holds [50, 59]. The success of the ATP can greatly depend on the quality of clausification. This is why there are various techniques like Formula renaming, Skolemization and Simplification to improve this transformation [50].

Proving, that the resulting clause set is satisfiable or not, can be done by saturation. Saturation means to compute the closure of a given set of formulas or clauses under a given set of inference rules [14]. Therefore, new clauses are derived by applying inference rules on one or more existing clauses. Afterwards, the new clause is added to the original clause set and can be used as a premise again for deriving new clauses. If no new non-redundant clause can be derived by the clause set, it is saturated up to redundancy and the process stops. Saturation comes also to an end, if the empty clause is derived. This would prove that the clause set is unsatisfiable and witnesses the implication of the hypothesis H by the original axioms A [62].

During the process, current calculi simplify clauses by for example replacing terms by smaller ones (called *rewriting*) or removing clauses that are implied by a more general clause (called *subsumption*). This reduces the size of the clause set without loosing non-redundant information and could speed up the proof search. However, saturation can derive an infinite number of consequences in most cases. To guarantee that if the clause set is unsatisfiable the empty clause is derived in a finite runtime, no non-redundant inference should be delayed infinitely. This principle is called *fairness* [62].

One of the key aspects in ATP is to design an algorithm that determines when to compute which inference, as there is a great number of possible derivations. One of the simplest selection rules is *level selection* which derives all possible clauses from the original clause set before adding them into the set. This ensures the fairness of the algorithm, but does not support any heuristic guidance. The counterpart to this process is a *single step* algorithm. For this, only one inference is performed at a time and the derived clauses are instantaneously added to the clause set. Although heuristics can be used to select the next step, they have to be applied on inferences instead of concrete clauses. In addition, every computed inference must be recorded to prevent repetitions, but requires a great amount of memory for larger proof searches.

Today, most provers like Vampire [55, 32], Prover9 [43], SPASS [80] and E [60, 61] rely on variants of the *given clause* algorithm. For this, the clause set is divided into the subsets U for the unprocessed and P for the processed clauses. Initially, P is empty and U contains all clauses. In each iteration, one clause g from U is selected by a heuristic and added to the subset P. Next, all inferences using g and other premises from P are computed and included in U. With these steps it can be verified that all inferences between clauses in P have been performed and if P is unsatisfiable the empty clause must have been derived. The algorithm terminates as well if U is empty what would witness the satisfiability of the clause set.

The two most popular variants of the given clause algorithm are the Otter loop and DISCOUNT loop [10, 42]. The main difference between those algorithms is that Otter loop uses all clauses for simplification while only processed clauses are used in the DISCOUNT loop. This leads to a faster iteration runtime for the DISCOUNT loop, but the Otter loop typically needs less iterations to find a proof. So, both variants have advantages and disadvantages, and none was found to be systematically superior of the other. Still, the heuristic for selecting the given clause g for each iteration has a great effect and is one of the main criteria for the provers' performance [62].

The concrete implementation of the DISCOUNT loop in E is illustrated in figure 2 and further discussed in the next section 2.2.1.

2.2. E Prover

In this project, the state of the art theorem prover E [60, 61] is used as a foundation. It is a saturation-based prover based on first order logic with equality. The four key aspects of E are its calculus (variant of superposition), the search organization (DISCOUNT loop), the heuristic control (clause and literal selection, term ordering) and its inference engine (efficient rewriting). While developing a new heuristic, the most interesting parts are search organization and the heuristic control. Therefore, these aspects are presented in the following sections based on [60, 61].

2.2.1. Search Organization

In E, the proof search is based on the DISCOUNT loop. As introduced in section 2.1.2 the DISCOUNT loop is a variation of the given clause algorithm and splits all clauses into the subset processed P and unprocessed U. At the beginning of the proof search, P is empty and all initial clauses are in U. Afterwards the algorithm moves one clause from U to P one at a time and stops if either P contains the empty clause (set is unsatisfiable) or U is empty (set is satisfiable). E has implemented five steps to add a clause from U to P which are illustrated in figure 2:

1. The first step for each iteration is to select the given clause g. This is done by evaluating every clause with heuristics and choosing the best one. As clause selection is one of the major aspects of automated theorem proving, section 2.2.2 presents some heuristics and their implementation structure in E.

2. After a clause g is selected, it is simplified. Therefore, all inference rules are applied based on g and potential other premises from P that modify g. If g is the

empty clause, the algorithm stops and returns the unsatisfiability of the clause set. Otherwise, the simplified version of g will be used for all further steps.

3. The next step handles the counterpart simplification. If a clause c in P could be simplified by using g, it is moved back to U after reduction. Moreover, if c is fully subsumed by g or redundant it can be removed without loosing completeness. Otherwise the clause stays unmodified in P.

4. With the simplified g inference rules are implied to derive new clauses between g and P. Superposition, equality factoring and resolution is performed for the generation step.

5. Finally, the generated and prior clauses have to be simplified with regards to P. However, as this can be an expensive task, only those inference rules are performed that E efficiently implements. Despite the simplification of step 2 and 3, the clauses do not need to be fully simplified to guarantee completeness as they can be processed again including step 2 later.

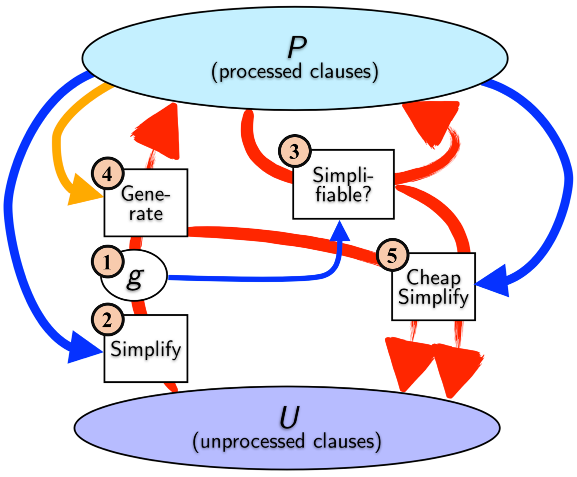


Figure 2: Illustration of the DISCOUNT loop as implemented in E. An iteration can be separated into five steps performing selection, simplifications or derivations. The figure is adapted from [59].

2.2.2. Heuristic Control

Search heuristics have a great impact on the provers' performance, especially when using the DISCOUNT loop instead of the Otter loop. E provides a generic framework for integrating and implementing different heuristics for clause selection, literal selection and selection of term ordering. In this project, a new heuristic for clause selection is developed by using Deep Learning. Therefore, this section gives an overview of current clause selection heuristics [60, 61].

For each unprocessed clause, all heuristics compute an evaluation only once after it was generated. Priority Queue

FIFOweight A fair and simple heuristic is First-In/First-Out (FIFO). Older clauses are selected first ensuring completeness and fairness of the proof search. The evaluation is based on a counter that is increased for each generated clause.

Clauseweight This heuristic evaluates a clause by the number of symbols. It is parameterized by a weight for function (including constants) and predicate symbols, a weight for variables and a factor for positive literals. The clauseweight heuristic prefers small clauses that typically generate fewer descendants and likely present general concepts [60].

Refinedweight Refinedweight works similar to clauseweight but modifies the symbol counting by increasing the weight for maximal terms and literals.

In E, the most successful clause heuristics are combinations of i.e. symbol weight or FIFO, also called hybrid heuristics. The evaluation by which the given clause is selected alternates in a round-robin fashion.

2.3. Machine Learning

The field of machine learning summarizes algorithms that become more accurate in predicting outcomes without being explicitly programmed [34]. The basic concept is using statistical analysis on received input data and its corresponding correct output. But how can an algorithm learn? A succinct definition for that is given by [44]:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measures P, if its performance at tasks in T, as measured by P, improves with experience E." [44, p. 2]

The interaction between task, performance and experience is shown in figure 3 and described in the following sections. The layout is inspired by [17, 44] to give a brief introduction to the huge area of machine learning.

2.3.1. Task *T*

A learning algorithm is designed to solve a specific task. This task defines how an input should be processed and what the result of this algorithm looks like. As an example if a robot should be able to walk than the task it tries to solve is walking. The input could be a RGB image of its surrounding while the output is the leg's movement.

While the algorithm is learning on a task the input it gets is called an *example*. An example $\boldsymbol{x}$ is a collection of features that have been measured as a sample input of the algorithm's system. Mathematically the example is represented as a vector $\boldsymbol{x} \in \mathbb{R}^n$ which

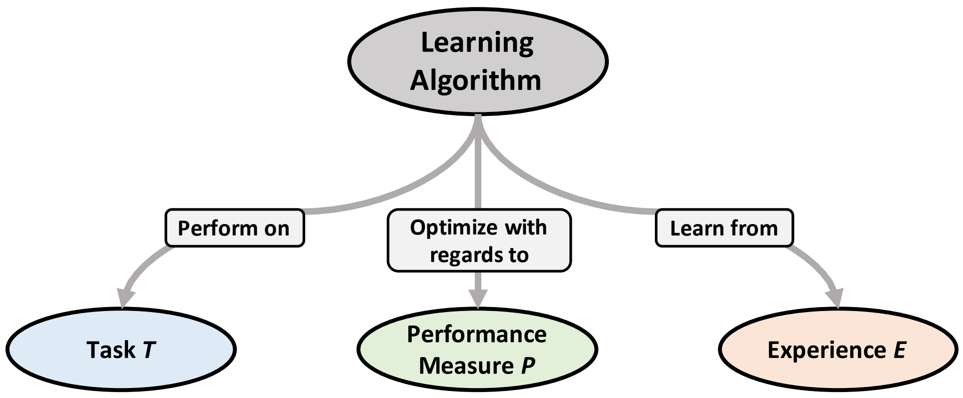


Figure 3: Overview of the relations between the learning algorithm and its task T, performance measure P and experience E. While T specifies the process, E defines how the datasets look like and P the way algorithm learns from the data.

elements are features. For an image this means that the pixel values are the features of the example.

After defining what a task is and how an input looks like they could be distinguished in different kinds. Due to the huge variation of possible tasks the following ones just represent a small subset of the most common machine learning tasks. For a more detailed list see [17].

Classification One of the basic tasks of learning algorithms is specifying the input's category among multiple possibilities. This means that the algorithm represents a function $f : \mathbb{R}^n \rightarrow \{1, ..., k\}$ that classifies the input example with its features to one of k categories. The output for choosing a category is usually defined by a numerical value $y = f(\boldsymbol{x})$ or an one-hot vector $y \in [0, 1]^k$ for which f calculates a probability distribution over classes. Typical use cases of this kind of tasks are object and handwriting recognition on images [36, 57].

Regression Some tasks require continual numerical values instead of discrete classes as output. In comparison to classification the represented function of the algorithm is defined by $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Regression is used for various motor control tasks [48] and object localization in images [38, 54].

Synthesis and sampling A different kind of task is when the algorithm is asked to generate new data examples that are similar to those in the training dataset. The input of this task is flexible and depends on the specific application. Current approaches deal with Generative Adversarial Networks [18] which try to imitate the ground truth distribution so that another network can not distinguish between real and generated data.

2.3.2. Performance Measure *P*

In order a algorithm can learn it has to get a feedback of how good its predicted output was. So a quantitative measurement of its performance must be designed, usually specific to the belonging task T. In addition different algorithms can easily be compared using the same performance measurement. In the learning process the goal of the algorithm is to optimize its parameter towards performance gain.

A common way to measure the performance on tasks like classification is to define the accuracy of a model. Accuracy is the proportion of examples for which the model predicts the correct output. The opposite of it would be the error rate, the proportion of examples for which the model predict an incorrect output. During training, the performance is rated more precisely to optimize the internal parameters so that the output is close to the labels. The difference between prediction and ground truth is measured by a loss function which the learning algorithm tries to minimize. For classification, a popular loss function is Cross Entropy [17]. It quantifies the difference between two probability distributions p and q by adding the Kullback-Leibniz distance $D_{KL}(p||q)$ to the entropy of p. Over a set of values $y \in \Omega$, the Cross Entropy is defined as:

$$H(p,q) = H(p) + D_{KL}(p||q) = H(p) + \int_{y \in \Omega} p(y) * \log \frac{p(y)}{q(y)} \tag{2.2}$$

However, if both distributions are discrete and only defined for certain values, the formula can be simplified by replacing the integral by a sum and eliminating the entropy of p. In the case of classification, p is the ground truth distribution and q the prediction while both depend on an input vector x. The value y is one of the output categories and therefore discrete. The average cross entropy over a set of N examples and M classes can be calculated as:

$$H(p,q) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{M} p(y_k|x_i) \log q(y_k|x_i) \tag{2.3}$$

The performance of a learning algorithm would be ideal if $p = q$ where the Kullback-Leibniz distance is minimal at a value of 0. As the entropy $H(p)$ can not be affected by q, the cross entropy also has a minimum for $p = q$. This is why learning algorithms are often optimized to minimize the cross entropy of its prediction and the ground truth.

If only labels of 1/100% or 0/0% are used, the ground truth distribution is limited to $p(0|x_i) = z_i$ and $p(1|x_i) = 1 - z_i$ for the label z, and the prediction $\hat{z}$ equally to $q(0|x_i) = \hat{z}$, and $q(1|x_i) = 1 - \hat{z}$. Therefore, the loss function can be simplified as follows [17]:

$$\text{BCE} = -\frac{1}{N} \sum_{i=1}^{N} (z_i \log \hat{z}_i + (1 - z_i) \log (1 - \hat{z}_i)) \tag{2.4}$$

This simplification of cross entropy is called Binary Cross Entropy (BCE), as it uses binary labels. A variation of Binary Cross Entropy will be used in section 3 to measure the performance of predicting the relevance of a clause during training, as it is 1 if the clause should not be used, and 0 otherwise. However, the penalty function often depends on the application of the system and has to be adjusted for every task.

2.3.3. Experience *E*

As the last part of the machine learning basics the source from which the algorithm learns should be explained. The basic component of experience is a dataset consisting out of a collection of examples. Mostly it is split up into a training and testing part. The training dataset is used for the algorithm to learn from while on the testing dataset the performance is measured regarding to new examples which the algorithm has not seen for training. This prevents overfitting and tests the generalization of the system.

Three common different kinds of datasets could be distinguished depending on the way the algorithm gains experience of the data:

Supervised learning The easiest way for an algorithm to learn how the estimated function looks like is having for every example the correct output, called label or ground truth, and optimize it towards that. Looking from a statistical perspective supervised learning has a random vector $\mathbf{x}$ as input and learns to predict the correct label $\mathbf{y}$ by estimating $p(\mathbf{y}|\mathbf{x})$. The structure of the ground truth depends on the task which should be learned. This approach is called Supervised Learning, because a teacher is needed to show the network the correct answers. For example, to get a network classifying objects into different classes like fruits, cars and ships, it is trained to predict the class fruit for the input object apple. However, the ground truth has to be generated by an instructor that is mostly human.

Unsupervised learning In unsupervised learning the ground truth is missing. The algorithm has to learn useful properties of the dataset structure without explicitly showing it. Usually for deep learning the aim is to capture the entire probability distribution either explicitly, as in density estimation, or implicitly, for generating new image at the task of synthesis and sampling. Another possible application after capturing the distribution is clustering the dataset in different categories by self-explored structures. So it learns a joint distribution of all features in the input vector $\mathbf{x} \in \mathbb{R}^n$ [17]:

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(x_i|x_1, ..., x_{i-1}) \tag{2.5}$$

Although the distribution estimation and the way of learning looks different to supervised learning they both could not be clearly separated. Given the ground truth every unsupervised task could be converted to supervised, and on the other hand the conditional probability of the supervised way can be solved by learning the joint distribution of $p(\mathbf{x}, y)$ by unsupervised methods [17]:

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')} \tag{2.6}$$

A learn paradigm taking some of supervised and unsupervised is the semi-supervised learning where for example only some examples have label but not all of them. Unsupervised learning mostly takes longer until the algorithm has fully learned the distribution because it has to explore the structure by itself. But as a huge advantage no instructor is needed and so much more data is easily available. Applications of unsupervised learning are speech recognition/generation [58, 78] and video prediction [13, 79].

Reinforcement learning A common human way to learn new things is by "trial and error" [23, 73]. Simulating this behavior reinforcement learning is a paradigm in which the algorithm interacts with its environment and learns from feedback or reward that is returned. Therefore the system selects actions in an environment in order to maximize the expected reward. Figure 4 illustrates this framework.

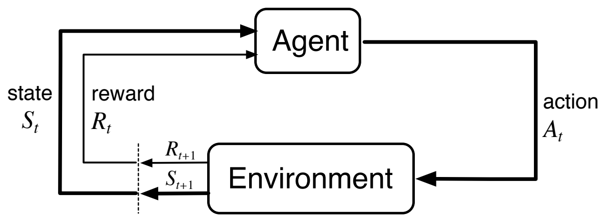


Figure 4: The learning algorithm is represented by an agent that takes an action A_t and receives a reward R_{t+1} from the environment. Its decision is based on the current state S_t and reward R_t. In games the state usually is a RGB image of the scene the users sees and reward is given by the score [73].

An intuitive application of such learning problems is playing games [45, 27]. In classical computer games like Atari [45] there are a limited number of possible actions the player can take. The algorithm has to learn the expected reward of an action based on the input that is mostly the pixel values of the game. Approaches like AlphaGo [65] have successfully shown the possibilities of reinforcement learning while beating humans performance.

While games have always a certain reward signal (the simplest one is winning or loosing) some applications are missing that out. For this inverse reinforcement learning (IRL) is usable where an expert demonstrates the task that the algorithm should learn [2]. Reinforcement learning applies for computer games [45] and motion planning [66].

2.4. Artificial Neural Networks

Artificial neural networks are graph based models that are inspired by the human cognition [9]. A single neuron represents the smallest unit or node of such a network interlinked by many interconnections with other neurons using its activation to communicate. This behavior is simulated by simplified mathematical models for which the neurons are represented as graph nodes and the connections are weighted directed edges.

To understand the mechanism of artificial neural networks the following section is divided into three parts. The first paragraph deepens the mathematical model of a neuron while the second handles with the basic architecture of a network and the third introduces the common convolutional neural network structure.

2.4.1. Artificial neurons

Figure 5 shows the mathematical model of a single neuron. It has multiple external inputs $\{x_1, x_2, ..., x_n\}$ which are usually the output of other connected neurons. All of them are weighted by a corresponding parameter of $\{w_1, w_2, ..., w_n\}$ defining the relevance of an input for this neuron. Finally the weighted inputs are summed up to u for which an additional parameter b is taken into account. This variable is called "bias" and shifts the evaluated sum with a constant value. The neuron's output y is calculated by taking u as input for an activation function $g(u)$ that defines the output's mapping to the input. When a neuron learns it adjusts its parameter including input weights and bias towards the expected output.

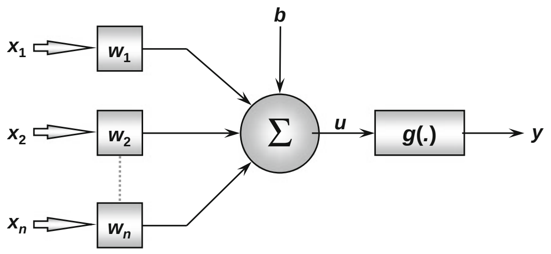


Figure 5: The mathematical model of a neuron consists of a weighted sum of its inputs that is shifted by a constant bias. The sum is given as input to an activation function to get the final output [9].

Summarizing an artificial neuron is represented by the following equation [9]:

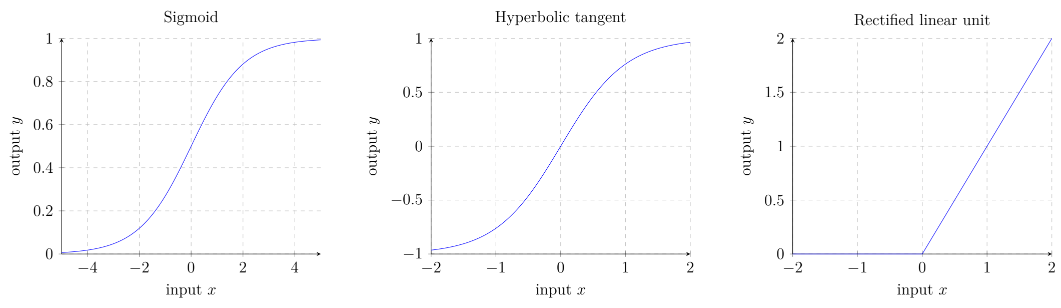
$$y = g(b + \sum_i x_i \cdot w_i) \tag{2.7}$$

The activation function's goal is basically reducing the output range of values because the weighted sum u could theoretically be anything between $-\inf$ and $+\inf$. If many neurons are cascaded the output value could easily blow up. Furthermore from a biological point of view the output of a neuron is the activation that is also fixed to specific borders. There are several activation functions which can be used but only three should be described further here: sigmoid, tanh and ReLU [47].

The sigmoid function is defined as $\sigma(x) = \frac{1}{1+e^{-x}}$ and plotted in figure 6a. It is fully differentiable and nonlinear while having a range of values between 0 and 1. Due to it's output points are very steep near to 0 it tends to push all values to either 0 or 1. So it is a common activation function for the classification task.

Figure 6b shows the hyperbolic tangent that is defined as $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. It can be written by using the sigmoid function as $\tanh(x) = 2 \cdot \sigma(2x) - 1$ and shares most of the properties with the sigmoid function. Due to the output range of values is between -1 and 1 it is often used as output activation function for tasks like image generation or inside a network.

Rectified linear unit (ReLU) [47] is another more simpler activation function that is plotted in figure 6c. It is defined by the equation $\text{ReLU}(x) = \max(0, x)$ and limits the output to a range of $[0, \inf)$ what can still blow up. However one big advantage of this function is that it is less computational expensive and the activations are sparse what makes the network more efficient and faster to learn. ReLU is everywhere differentiable except at $x = 0$ but has no gradients for $x < 0$.



(a) $\sigma(x)$ (b) $\tanh(x)$ (c) $\text{ReLU}(x)$

Figure 6: (a) The sigmoid function maps the inputs to a range of 0 to 1 while having high gradients near to $y = 0$ to bring the output more to either 0 or 1. (b) The hyperbolic tangent is similar to the sigmoid function but has a output range of -1 to 1. (c) A rectified linear unit (ReLU) is 0 for all input lower than 0. All other values are processed linear so that they do not change.

Overall there is not the one activation function that can be used for every network. Moreover sigmoid is better to use for classification output, tanh for synthesizing and ReLU inside the network [33, 34].

2.4.2. Deep Feedforward Networks

After analyzing the concept of a single neuron this paragraph deals with the interconnection and combination of multiple neurons to a network. The quintessential artificial neural architectures are deep feedforward networks, or multilayer perceptrons [17]. These models are called feedforward because information flows through the whole network without any connections in which outputs of the model are fed back into itself. Networks with such connections are called Recurrent Neural Networks (RNN) and are described in section 2.5.

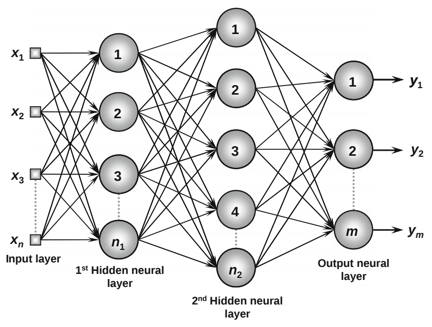
In general an artificial neural network consists of multiple layers that can be divided into three main parts:

Input layer The first layer of a network receives information (data) from the external environment. Usually these inputs are normalized between -1 and 1 for better numerical precision of mathematical operations and to keep the network's weight in a certain range [9, 17].

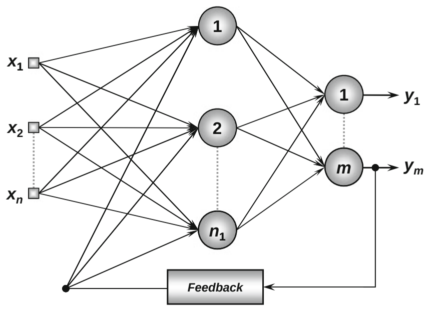
Hidden layers The main computing layers of a neural network are called "hidden layers" because they are invisible for the external environment. The structure consists of multiple stacked neurons that are responsible for extracting patterns associated to the input. Neurons are only connected between layers but not within a single layer.

Output layer The last layer of a network is composed of neurons that are responsible for predicting the final output based on the previous hidden layers. Similar to the input of the network the output is usually normalized as well.

The structure of a simple feedforward network with two hidden layers is shown in figure 7a. The input is processed straight forward through the network and there is no recurrent connection. In comparison the model in figure 7b has a feedback from the last output layer back to the network. With this the next calculated output depends on the result of the previous step and is able to learn from a time component (for details see section 2.5).



(a) Feedforward Network



(b) Feedback/Recurrent Network

Figure 7: (a) This figure illustrates fully connected feedforward network with two hidden layers. All neurons of one layer are connected to every neuron of the next layer with no feedback connection [9]. (b) In comparison to feedforward network a RNN has a feedback connection from one later neuron back to one in a earlier layer [9].

In the example of figure 7a all input nodes are fully connected with all neurons of the first hidden layer. Every neuron of this layer has therefore n inputs and n corresponding weights next to one bias that have to be adjusted by learning. With n_1 neurons the first hidden layer contains $(n + 1) \cdot n_1$ learnable parameters. Layer two has therefore $(n_1 + 1) \cdot n_2$ and the output layer $(n_2 + 1) \cdot m$ parameters. With an increasing number of neurons the amount of parameters blows up what gets computational expensive and hard to learn. Sparser layers are for example convolutions which are described in the next paragraph.

2.4.3. Convolutional Neural Networks

A special kind of networks for processing data that has a known, grid-like topology like images, are Convolutional Neural Networks (CNN) [35]. They are based on the primary visual cortex of the brain and strongly inspired by neuroscientific research. CNNs usually consist of two important layers: the convolution layer that is a variation of the mathematical operation called "convolution", and the pooling layer that reduces the inner width and height dimension of the data.

An example to motivate the usage of convolutions is having an RGB image of 64x64 pixels as input to a network. If the first hidden layer would consist of a fully connected layer with for example 1024 neurons, the needed amount of parameters would be $64 \cdot 64 \cdot 3 \cdot 1024 = 12,582,912$. One way to reduce this number is by considering the geometry of the image. A pixel correlates much more with its close neighbors than with far off pixels [3]. Since the correlation of inputs is represented in the weights of a neuron the connections could supposedly be reduced to only a local region of the data, called kernel or patch, like 5x5 pixels (see figure 8). This leads to that the structure of the hidden layer is a similar grid to the input. Convolutional layers are going even further by sharing the weights between all neurons. In addition one layer does not only consist of one weight set, but multiple neuron grids are stacked on each other which are called channels. For example, applying a standard convolution layer with a 5x5 kernel and 64 channels on an input image of the size 64x64px, the first hidden layer would consist of $64 \cdot 64 \cdot 64 = 262,144$ neurons but only have $5 \cdot 5 \cdot 3 \cdot 64 = 4,800$ weight parameters.

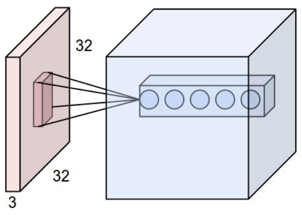


Figure 8: A neuron in a convolutional layer is only connected to a local region of the input. To apply different weight setting on the same kernel, multiple neurons are stacked behind each other which do not share the weights. However all neurons in one channel have the same to reduce the number of parameters [28].

The same layer can also be applied on different data types which might be one dimensional. Therefore a kernel of the size 3x3 is transformed into a kernel of 3x1 and only has a local region over one dimension. To expand the receptive field without increasing the number of parameters, input neurons could be skipped. This is described by dilated convolutions [82] with a dilation factor d which determines the neurons that should be skipped. If for example $d = 2$, only every second input is used for the convolution. Figure 9 illustrates the comparison between standard and dilated convolutions.

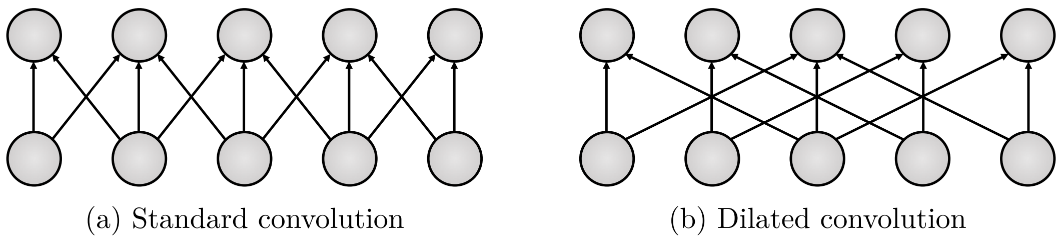


Figure 9: (a) A simple one-dimensional convolution with a kernel size of 3. (b) Dilated convolutions skip nodes in the local region. The example shows a dilation factor of 2 where the immediate neighbours are left out. Image inspired by [39].

The resulting output has the same size as the input, but might not be so rich of information anymore, especially when using ReLU as activation function (many outputs are 0). To concentrate on the important results, a pooling layer [34] reduces the height and width dimension by applying operations like max on all inputs of its kernel. It is worth mentioning that pooling is done on each channel separately. The max pooling operation is inspired by the biology where neurons that are strongly activated, exceeds their neighbors and weaken their influence. Figure 10 illustrates the mechanism of a max pooling layer.

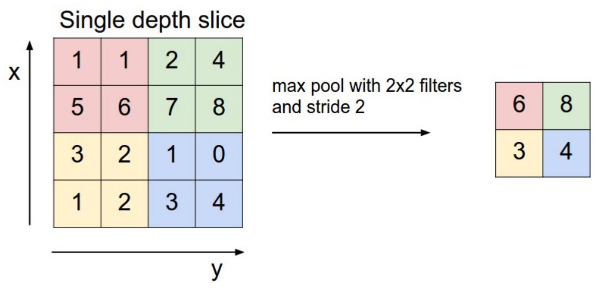


Figure 10: This figure illustrates the procedure of a max pooling layer with a kernel size 2 and stride 2. On every 2x2 pixel patch a max operation is applied to extract only the highest values. This reduces the dimensionality of the input by the factor of 2 [28].

The overall architecture of Convolutional Neural Networks consists of multiple stacked convolutions and pooling. Due to the output structure depends on the task of the network the final layers are adjusted to it. For example the task of classification ends up into a 1D-vector, so that the final layer usually is fully connected.

2.5. Recurrent Neural Networks

Next to the Convolutional Neural Networks that are specialized for processing a grid of values, Recurrent Neural Networks (RNN) [56] are focused on processing a sequence of values $x_0, ..., x_n$. Of course such a sequence could also just be stacked together to one big input to a multilayer network, but this would not be parameter efficient. For example considering a model for language understanding that should extract a year out of a sentence. Two samples are "I went to Nepal in 2009" and "In 2009, I went to Nepal" [17]. The corresponding word appears in the first sentence at sixth and in the other at second position. Still the parameters could be shared between these two time steps.

The following paragraphs deepens the topic of RNN by describing how recurrence could be modeled in a neural network and presenting the Long Short-Term Memory (LSTM) [21] for long-time dependencies.

2.5.1. Modeling recurrence

Recurrent Neural Networks have, as in section 2.4.2 described, feedback connections from a neuron's output back into the network. The values of these connections are held over time steps, so that processing input x_t depends on computation results of x_{t-1}.

A better understanding of a recurrent network is unfolding it over multiple time steps, as illustrated in figure 11. Network A gets x_t as input to generate its output h_t. In addition there is a connection over time steps, so that x_{t-1} influences the prediction. Due to x_{t-1} is therefore influenced by x_{t-2} and so on, h_t depends on the whole previous sequence $x_0, ..., x_t$.

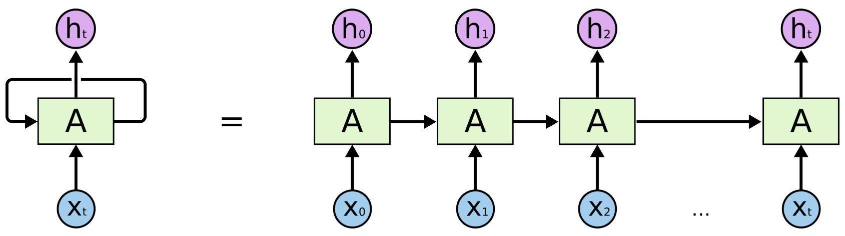


Figure 11: Recurrent Network A can be seen as a chain of multiple time steps with feedback connections between them. Due to it is the same network the same weights are used for processing the input and the recurrent output of previous step influences the final prediction [51].

The design and position of the feedback connection is based on the task. Some examples are:

- RNNs that produce an output h_t at every time step and have recurrent connections between hidden units as shown in figure 11.
- RNNs that produce an output h_t at every time step that is taken as input for the next time step. A common application for this design is video prediction [13, 79].
- RNNs that only produce an final output after reading a whole sequence. For this hidden units have recurrent connections in between to process and extract useful information out of every single input.

The detailed structure inside the network is not specified. Convolutional Neural Networks can be combined with RNN by taking the output of one convolution as input in the next time steps. However the computation of x_t strongly depends on x_{t-1}, but if long time dependencies are needed like for example in complex language understanding, other methods outperform for this task. One example are the Long Short-Term Memory Networks (LSTM) [21] that are explained in the next paragraph.

2.5.2. Long Short-Term Memory Networks

As [6, 20] have shown it is difficult to train recurrent neural networks to store information over a long time. Gradients flowing backwards in time tend to either vanish, what causes a need of training for very long time or does not work at all, or blow up, so that the weights are not stable anymore. To overcome these problems [21] proposed a much more efficient gradient-based method called "Long Short-Term Memory" (LSTM).

LSTMs are a special kind of RNN which enable an constant error flow over multiple time steps. Between the time steps a cell state and the last output is passed through building up in a chain like structure (see figure 12). Inside the module four neural networks layers interact with each other to decide which information will passed further (remembered) and which will be added. The following paragraph describes this behavior more detailed using the name assignments of [51].

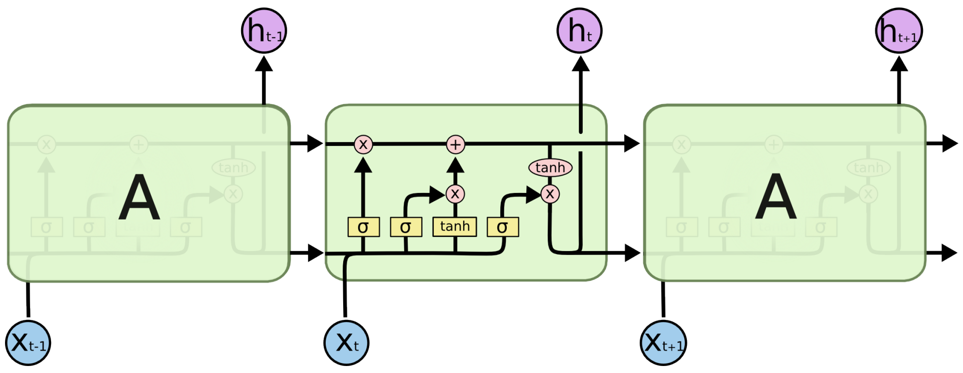


Figure 12: LSTMs are build up in a chain-like structure over time where they pass the cell state c_t and hidden state h_t to the next time step. The inner structure consists of a "forget gate layer", that selects which states should stay, "input gate layer", that chooses the new information which should be added to the states, and "output gate layer" for calculating the next hidden state [51].

The first step in a LSTM is the "forget gate layer". In here a mask of numbers between 0 and 1 are calculated in respect to the current input x_t and the last output/hidden state h_{t-1} to determine which information of the cell state c_{t-1} should be kept (respectively 1) and which to get rid of (respectively 0). Mathematically this layer could be written as follows [21]:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.8}$$

Next the module has to decide which information should be updated based on the current time step. This is done by a sigmoid layer i_t, calculating an update mask similar to the forget gate layer, and by a tanh layer to create the new cell state values $\tilde{C}_t$. Combining the mask and the new states with a point-wise multiplication to an "input gate" the cell state c_t can be determined in the following way [21]:

$$\begin{aligned} i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\ C_t &= f_t * C_{t-1} + i_t * \tilde{C}_t \end{aligned} \tag{2.9}$$

The last step is to calculate the "output gate" for the current time step based on the new cell state C_t. As the previous layers a filter is implemented by a sigmoid layer taking x_t and h_{t-1} into account and used on the cell state which was pushed through a tanh function to scale the values between -1 and 1. The final output of the LSTM is described by the following equation [21]:

$$\begin{aligned} o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\ h_t &= o_t * \tanh(C_t) \end{aligned} \tag{2.10}$$

There are many variants on LSTMs. [15] adds "peephole connections" to every gate so that they also take the cell state into account. Another variation is the Gated Recurrent Unit (GRU) [8] which simplifies the LSTM a bit. The changes contain merging the forget and input gate into a single "update gate" and combining the cell state C_t and hidden state h_t.

LSTMs outperform on tasks where recognizing long-term dependencies is an essential key like speech recognition [58, 16] and language modeling [68, 70]. The application of LSTMs are extended by using convolutional LSTMs [64] so that they can also be used for video prediction [79].

2.6. Related work

The following section briefly introduces related work which was taken into account for this project. The first paper, DeepMath [4], deals with fundamental research of embedding clauses to evaluate them with several neural network architectures. The second subsection presents another paper that is based on the previous one and extends the research on networks for clause embedding to more complex and deeper models.

2.6.1. DeepMath - Deep Sequence Models for Premise Selection

The first attempt to combine automated theorem proving with deep learning was made by [4]. The general idea is that a network processes pairs of clauses and negated conjecture, and predicts whether the clause is likely to be useful for the proof or not. As clauses have an arbitrary size, an embedder network is developed that compresses them into a fixed size feature space. Besides, the negated conjecture is also embedded with a similar network. Both feature embeddings are concatenated, and two consecutive fully-connected layers are applied ending in a single final output. This output determines the evaluation of the input clause. The overall architecture is visualized in figure 13.

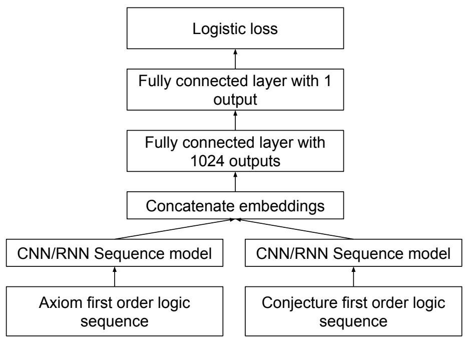

Logistic loss
Fully connected layer with 1 output
Fully connected layer with 1024 outputs
Concatenate embeddings
CNN/RNN Sequence model
CNN/RNN Sequence model
Axiom first order logic sequence
Conjecture first order logic sequence

Figure 13: The basic structure for premise selection with deep learning proposed by [4] contains two embedding networks, one for the clause to be evaluated, and the other for the negated conjecture of the current problem. Both feature vectors have a fixed size and can, therefore, be processed by two fully connected layers. The final output constitutes the evaluation of the clause.

The performance of this heuristic concept strongly depends on the network architecture used for the clause and conjecture embedding. Therefore, different models of Convolutional and Recurrent Neural Networks are tested in [4]. The first step was comparing architectures on character level that treat formulas as sequences of single characters embedded by an 80-dimensional one-hot vector, and word level, where embeddings for constants, functions and other symbols are learned by embedding the axioms of their defining statements. The second approach has the advantage that it uses smaller inputs, especially for functions with long names, but needs an extensive vocabulary for datasets with many different symbols.

The network architectures evaluated in [4] on character level were an LSTM [22] and a GRU [8] model, a pure convolutional architecture with final max-pooling over channels, and a combined network of convolutions and LSTM. For word-level, only CNNs were tested. The overall result was that all networks improved the proof search, but convolutional models outperformed recurrent networks. Especially the LSTM needed a massive amount of training steps before reaching its final performance. As a conclusion, deep learning can help to find new proofs and learn heuristics, even with shallow networks.

2.6.2. Deep Network Guided Proof Search

Another related work considered for this project is "Deep Network Guided Proof Search" [39]. The ideas of the paper are similar to [4], but expand the evaluation of clause embedding networks to more complex and deeper models. All networks rely on a vocabulary that is used for embedding clauses on word level. The features of a symbol are learned during the training process. Overall, three different architectures are compared that should be shortly presented here:

Simple convolutional model The first model consists of three stacked convolutions with a patch size of 5 and 1024 channels. ReLU is used as an activation function. Although the network is relatively small and might not be able to learn great feature complexity, it can be computed very fast and can process more clauses than networks with a higher complexity at the same time.

WaveNet Another approach is the WaveNet [77] that was originally designed for processing raw audio. The network consists of three stacked WaveNet blocks that each contains seven dilated convolutions with increasing dilation by 2 ($d = 1, 2, 4, ..., 64$). Instead of ReLU, gated activation functions $\tanh(x) * \sigma(x')$ are applied where x and x' are computed on the same input but different weights for the layer. The output of a WaveNet block is added to its input by using residual connections. Moreover, dropout [69] is used between blocks for generalization.

Recursive neural network The third network architecture is based on recursive parser networks [67]. The networks' structure is a function of the parse tree of the first-order logic formula and so adjusts to the input. Four different layer types are distinguished:

- **or** for a disjunction of two inputs
- **and** for a conjunction of two inputs
- **not** for a single input
- **apply** for applying a function on two inputs

All of the above operations are implemented by fully connected layers or tree LSTMs [74]. The weights for each type are shared across the same tree.

For all approaches, the embedding architecture is shared for the clauses to be evaluated and the negated conjecture but with separate weights.

The networks are also evaluated on first-order problems of the Mizar Mathematical Library (MML) [19]. Firstly, to check if the networks can distinguish between useful and not useful clauses at all, the accuracy is evaluated on a set of proofs that are already solved by automated theorem provers. All approaches show an accuracy of approximately 77% up to 81.5%. However, the recursive networks had the worst results and are also inefficient in the aspect of execution time, as the network structure has to be constructed for each clause. The WaveNet models slightly outperform the simple convolutional networks in accuracy.

When applying the networks in E [60, 61], the proof search was more often successful, if faster heuristics like the auto-mode of E are used in parallel. This is done either in a hybrid approach, alternating between the priority queues of the network and existing heuristics, or a switched approach where the network is solely used at the beginning, and the auto mode takes over after, i.e. 20 minutes. In experiments, the simple CNNs outperform the WaveNet models due to faster evaluation. Overall, 7.36% of all hard statements are solved by several approaches in union that constitutes an improvement of 4.7% with deep learning guidance.

3. Network model

Selecting the right clauses requires a wide context. To apply deep learning techniques for this complex task, the problem needs to be formalized. The network should extract significant features of a clause and be able to determine its value for a certain proof task. Like other clause selection heuristics, the network separately predicts a weight for every unprocessed clause and computes it only once. The inputs used to compute an evaluation ideally are the current clause, the conjecture which should be disproved, and all processed clauses.

However, there needs to be a trade-off between accuracy and computational effort. A network selecting the right clauses for every proof without making any mistakes might be perfect, but is not possible with current techniques. So, to use a network with lower accuracy, it has to be small and fast to still find all relevant clauses. Yet, the task stays complex and requires more layers with a higher computational effort. Finding the right trade-off between these two counterparts is a key aspect for creating clause heuristics with deep learning.

To introduce the network concept that was developed in this student project, the overall architecture is described in section 3.1.1. Next, the vocabulary (3.1.2), embedding network (3.2.1) and combination network (3.2.2) are discussed in more detail and the section finishes with an evaluation of the performance measurement in 3.3.

3.1. General concept

As presented in section 2.6, the current state of the art uses a two step approach with an embedding and subsequent combiner network to evaluate clauses with deep learning. In this project, a similar approach is applied but significantly differs in the vocabulary, network models and training techniques. This section gives an overview of the adjusted architecture and discusses the changes of the vocabulary part.

3.1.1. Architecture

The network architecture is based on [4, 39] and illustrated in Figure 14. To make it easier to clarify the concept, an example retrieved from the proof GRP004-2 of the TPTP library [71] is added in the figure and explained step by step in the following paragraphs.

The basic inputs to the network are the clause which should be weighted and the negation of the conjecture to be proved. For instance, the clause is *prod(X,identity)=X* and the negated conjecture *prod(a,inv(a))=identity* where X is a variable and a and *identity* are constants. The purpose of the network is now to decide whether the clause is likely to be useful for the proof or not.

The first step is to embed all symbols into a feature vector. This is done by a table lookup of a vocabulary that contains a trainable feature vector for every constant, variable, function name and parentheses. Both the clause and conjecture share the same vocabulary. In the example of the figure with a vocabulary vector size of 512, the

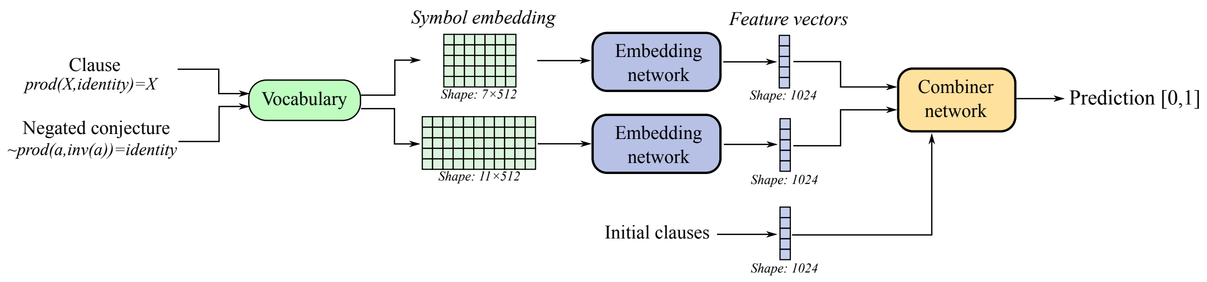


Figure 14: The overall network architecture with an example retrieved from GRP004-2 [71]. The clause and conjecture are embedded by a vocabulary into a feature matrix, whose shapes are based on the examples. These are processed by two networks with the same architecture but different weights, resulting in a fixed size vector of, for instance, 1024. A combiner network combines the clause's, conjecture's and original axioms' features to obtain a final score. In this case, the optimal prediction would be 1 (not useful), because the positive conjecture is already one of the initial clauses.

clause has seven symbols (*prod*, *(*, X, *identity*, *)*, $=$, X) resulting in a feature matrix of 7×512, and the conjecture similarly with eleven symbols. A further discussion about the vocabulary is given in section 3.1.2.

Next, an embedding network is applied on the clause and conjecture features. Both networks have the same architecture but different weights. As the input can vary in its first dimension, the network mainly consists of convolutions to process different input sizes. The output is a one-dimensional feature vector of a fixed size generated by a max pool over features on the last layer. In section 3.2.1, a dilated dense block for the embedding network is proposed combining a broad context with a complex and fast architecture.

After the clause and conjecture are reduced to a fixed size, a combiner network processes both vectors to get a final score between 0 (useful) and 1 (not useful) for the clause selection heuristic. Since the dimensions of the inputs are constant, fully connected layers can be applied. However, the evaluation would ideally depend on the current set of processed and unprocessed clauses. As a compromise to the extensive runtime that would be needed, a knowledge base of the original axioms is extracted at the beginning of every proof search and considered as an additional input into the combiner network. Thus, the prediction of a single clause depends on a wider proof understanding than only the conjecture. In section 3.2.2, the details of the knowledge base extraction are further explained.

3.1.2. Embedding vocabulary

The vocabulary lookup is the first step of the embedding process. The approach described in [39] collects all constants, function and variable names, logical operations and parentheses into a vocabulary during the data collection and learns a separate feature vector for each symbol. In this project, the same approach is used, but some features are shared between similar symbols.

For a good understanding of clauses, the network must be able to separate between for example constants and variables. To support this, the vocabulary is splitted into the symbol names itself and its type with each half of the original channel size (256 for the example in Figure 14). This means, that all operations, constants, variables and functions are clustered, and allocated to a shared feature vector. Functions of different arities are considered as distinct types and get separate features. As an example, when the clause *less(x,negative(y))* with x and y as variables is embedded, *less* and *negative* do not share their type feature vector because they have different arities, but x and y do. Figure 15 illustrates this approach with a second example.

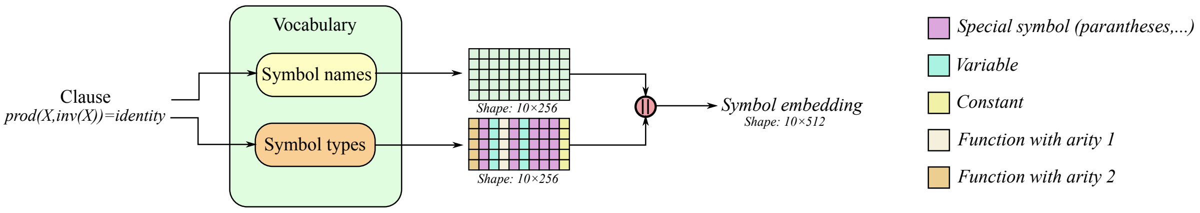


Figure 15: When embedding a clause or conjecture, the features of the symbols' names and their types are looked up. Therefore, all variables share half of their feature vector as well as constants, special characters and functions distinguished by arity. The colors of the type embedding are explained on the right.

3.2. Network architecture

Evaluating a clause resembles the task of natural language processing where a sequence of an arbitrary number of words has to be understood and interpreted. Mostly, recurrent networks are applied that process each symbol separate and pass internal features as an input to the next iteration [34, 17]. However, convolutional and recursive networks were also tested by [39] for clause evaluation. In this project, a novel architecture for the embedding network is proposed that combines a dense block [24] with multiscale dilated convolutions [82]. This model is introduced in section 3.2.1. The architecture of the combiner network is expanded by a knowledge base extraction using RNNs and discussed in section 3.2.2.

3.2.1. Dilated Dense Embedding Network

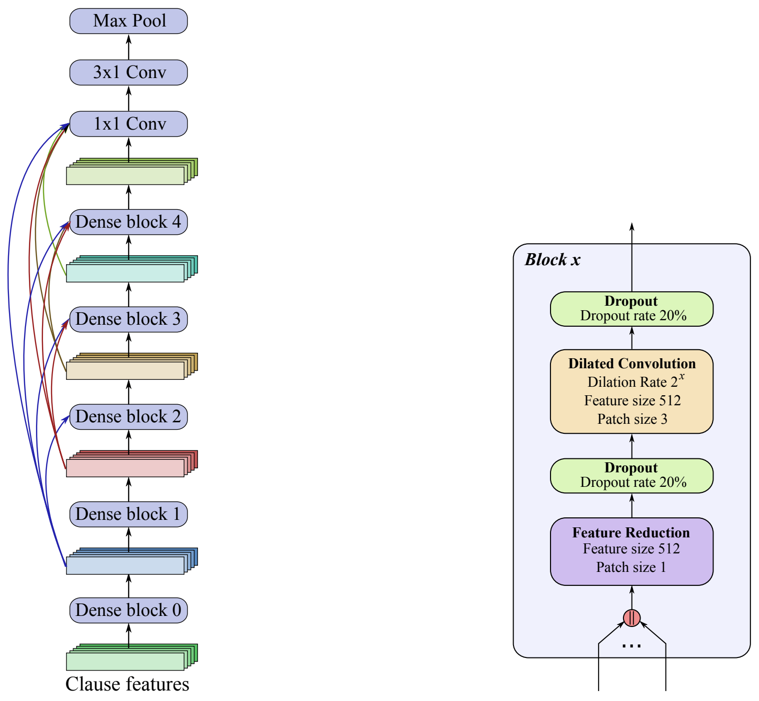
The greatest challenge in learning ATP heuristics is to develop a suitable network architecture that can compress an arbitrary clause into a fixed size feature vector based on which it can be decided whether to use the clause for a proof attempt or not. The complexity is constrained by the runtime performance as every generated clause must be weighted during the proof search.

Experiments of [4, 39] have shown that recurrent and recursive networks perform badly on the task of clause embedding in comparison to convolutional models. In addition, the computational effort is significantly greater as every symbol has to be sequentially processed by i.e. an LSTM cell. However, shallow convolutional models only consider a small number of anchor symbols as their patch size is limited. In this project, a new network architecture is developed that increases the receptive field and actively supports feature sharing between different patch sizes.

The principal component of the embedding network is a 5-layer dense block [25] which connects each layer to every other in a feed-forward fashion. The feature maps of every previous layer are used as input into all subsequent layers whereas 1x1 convolutions reduce the number of input channels before each convolution with patch size 3. The feature reduction decreases the number of parameters that have to be learned without loosing important context information like max pooling might. As an example, the last block has an input size of 4×1024 channels. Using a convolution with a kernel size of 3, the layer contains $3 \times 4 \times 1024 = 12,288$ parameters. In comparison, a feature reduction needs only a third of the weights. The input to the following convolutional layer has a fourth of the original channel size, so that only $1 \times 4 \times 1024 + 3 \times 1024 = 7,168$ parameters need to be optimized. This concept is even more efficient, if multidimensional convolutions (i.e. 2D for image processing) are applied on the input [25]. Having a great amount of parameters has the problem that the network easier overfits and training issues like gradient vanishing and normalization are enhanced, especially for small datasets [26, 17].

However, stacked receptive fields of 3 do not actively increase the field of view so that each final neuron has a theoretical receptive field of only 13 although the effective field is even smaller [40]. Considering an average clause length of about 30, a broader contexts needs to be provided for the embedded clause vector. Therefore, every convolution in the dense block is replaced with dilated ones [82] increasing the dilation rate by factor 2 throughout the layers. This leads to an overall receptive field of almost 64 so that much greater clauses can effectively be analyzed. Still, using dilated convolutions often comes with the disadvantage of losing local information. With the concept of a dense block, the feature maps are shared among all layers, and so local information is passed through the network as well as those from greater receptive fields are. Moreover, the last layer is a weighting of features where the network can learn the best combination of different field of views. With a post-processing convolution of patch size 3, the complexity of the network is increased which allows a last local view on much richer features in contrast to the first convolution, which is directly applied on the vocabulary tokens. A final max pool over channels reduces the output to a fixed size vector. Figure 16 illustrates the embedding architecture.

Compared to the WaveNet [77] used by [39] the proposed network has fewer layers and therefore a smaller computational effort. The three WaveNet blocks contain each 7 gated units that consists of two convolutions with a patch size of 3. Considering only the multiplications of weights and inputs for the layers, the overall number of operations would be $\#Blocks \times (\#Gated\ Units \times 2) \times PatchSize \times \#InputChannels \times \#OutputChannels = 132,120,576$. In contrast, the dilated dense block needs by using feature reduction less multiplications per block, summing up to $34,603,008$ operations overall.



(a) Overall embedding architecture (b) Dense layer

Figure 16: (a) The embedding network consists of 5 dilated dense blocks that share their output features with all successors. The first block does not contain an feature reduction as it is applied on the vocab. A final 1x1 convolution combines all feature maps with a last local view of an patch size of 3. (b) Every dense layer starts with a 1x1 convolution reducing the number of features and parameters. To increase the receptive field, the subsequent convolution has a dilation rate between 1 and 16, based on its position in the embedding network. 20% feature-wise dropout [69] between the layers for regularization and a feature size of 512 is used.

In addition, the Dilated-Dense block explicitly supports the feature sharing between different receptive field sizes. Also, by using 1x1 convolutions over all features of several fields of view, the network can learn even more complex functions for combining those. So, smaller terms or axioms like $odd(X)$ can easily be analyzed and reused for larger context like the example in Figure 14. The WaveNet constrains the information gain per receptive field with gated units whose output is added to the input features, because otherwise local information might be lost due to the large dilation factor.

Nevertheless, a simple 3-layer CNN is still much faster than a dilated dense block. To outperform these shallow models for finding new proofs, the accuracy of the proposed network is expected to be higher than the one from [39]. that is discussed in section 4.3.1.

3.2.2. Combiner network

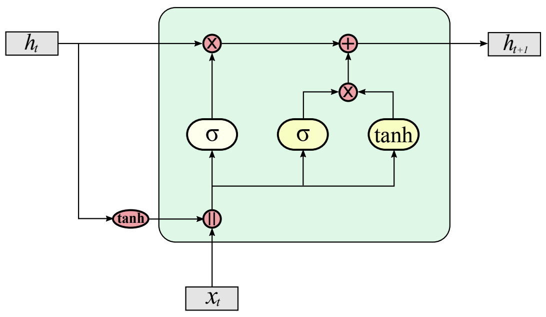
The final step of the network architecture is the combination of the clause and conjecture features to predict a heuristic score. As the size of both inputs are known and determinate, fully connected layers with ReLU [47] activation and a channel size of 1024 are used within the combiner network. The last layer ends up in only one output constituting the final prediction. In order to obtain a score between 0 and 1, a sigmoid activation is applied instead of a ReLU. To summarize, the combiner network has overall three fully connected layers and is illustrated in Figure 17b. Nevertheless, a third input, the knowledge base, is enclosed which is discussed in the next paragraphs.

When selecting the next clause for a proof, the decision would ideally depend on the full proof state i.e. both the current processed and unprocessed clause set. However, as both sets dynamically change and increase over time, it would be too computationally expensive to consider the full current state for all clause weight calculations. Therefore, only the initial clauses are processed once at the beginning of the proof to generate a solid foundation for better clause heuristic. Theoretically, an analysis of the initial clauses can be sufficient to find a proof as all possible derivations are based on these clauses, but this task would be even more complex for a network. However, without knowing any dependencies between clauses, no substantial prediction for a proof search can be made. A good example for this situation is the clause in figure 14. If the positive conjecture would not be in the initial unprocessed set, this clause might be useful for a proof. Moreover, if the training dataset contains different initial clause sets for the same negated conjecture, the network is simultaneously trained to predict 0 and 1 for the exact same inputs. So, without considering the initial clauses for a clause evaluation, the prediction strongly depends on the set of initial clauses given in the training dataset and the network could never reach a sufficient state for a random set of clauses.

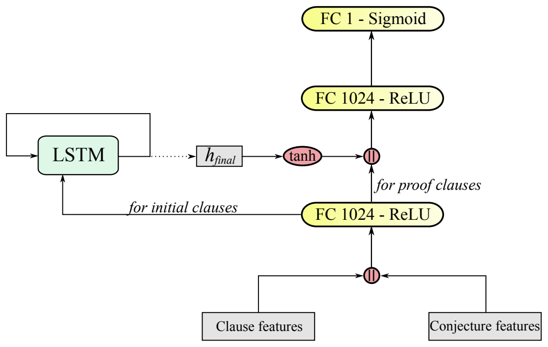
In order the implement such an approach in a neural network, the initial clauses are embedded with the same network that is used for the clauses to be evaluated followed by the first fully connected layer. To combine the features of different clauses and end up in a constant state size, a recurrent neural network is sequentially applied to all clauses. An LSTM structure [21] for the RNN provides an efficient gradient backflow over multiple timesteps (see figure 17a). Moreover, the gate units only pass important information of each clause to the state, so that the knowledge base is refined every step. In contrast to the standard application of a LSTM, just the final state is considered instead of a continuous output, and the state should be independent of the clause order. Therefore, the output gate is removed and the current state is replaced by the hidden state with a `tanh` activation.

To evaluate a clause, a `tanh` activation function is applied on the final state to reduce the values to the range -1 and 1, and concatenate it with the features of the first fully connected layer of the combiner network. A second consecutive layer combines the knowledge base with the clause's features, before the final score is calculated.

The evaluation time of a clause does not reasonably increase through this approach as only one layer with few parameters is added to the network. The computational effort is negligible as the embedding network has significantly more expensive layers. Still, an



(a) Simplified LSTM architecture.



(b) Combination network

Figure 17: (a) Compared to the LSTM introduced by [21], the architecture does not have an output gate and only one hidden state. The input features are extended by the last state with a `tanh` activation to compress the input between -1 and 1. Based on these, the hidden state is updated via the forget and update gate. (b) The combination network is built up similar as in [39] and consists of 3 fully connected layers. For initial clauses, the output of the first layer is directed into the LSTM. Otherwise, the final hidden state is merged to the current features and processed by two additional layers.

overview of the initial clauses are crucial for good clause heuristics and is expected to support the network to evaluate clauses.

3.3. Performance Measurement

Performance measurement is a key factor for training neural networks. Ideally, actual proof deductions would be simulated at every training step and the network is trained on its own selections. However, this concept would take very long to train, especially for hard proofs, and so, the evaluation has to be broken down to a simpler and faster loss calculation. For this, E [60, 61] was used to generate positive and negative examples for all proofs in the dataset (details in section 4.1 and train the network on these clauses. Still, the data is strongly imbalanced because every proof only has a few positive clauses, but the number of negatives could be nearly unlimited. This section gives an overview of the methods used to encounter this problem: handling imbalanced data with closely selecting which examples to show next, and adjusting the loss.

3.3.1. Handling imbalanced data

Having imbalanced training data can cause a network to only focus on a subset of classes and therefore performing badly on the rare ones as shown in [63, 81]. Therefore it is very important to consider how the training data is handled to optimize the network. In this project a subset of methods proposed by [81] are used to support the training process and described inclusive their deployment on proofing clauses in the following section.

One common technique to handle imbalanced data is to just train the rare examples more often than the easy ones. As the network classifies a given clause if it might be helpful for a proof or not, the balance could be achieved for positive or negative clause examples. However, there are significantly more examples of clauses that are not useful for a proof, as mostly, only a small subset of all possible derivations already leads to a proof [62]. Training positive examples as often as negatives, they have to be repeated very often and the network easily overfits. Therefore, every batch consists of 3 times more clauses that are not useful for a proof than positive, and the loss function is adjusted to this imbalance which is discussed in the next section. A static data sampler is needed for the training process where the proportion of positive and negative examples of a batch stays the same during the whole training. Otherwise, the loss gets inconsistent and the network might be unstable [81].

However, many negative examples resemble one another and are therefore easy to learn. To concentrate on the hard ones, a dynamic data sampler is implemented that records the losses lately obtained for every single clause. One fourth of all negative examples of a batch are determined by having the highest loss in a proof. A similar approach was introduced by [4] and improved the accuracy of the model from 61% to about 66%.

Another, more important data balance has to be achieved for the negative conjectures. Its embedding network has much less examples than the one for clauses and therefore it is even more important that the small training data is not biased. Simple static data sampling methods are not sufficient because the negative conjectures can not simply classified into rare and common examples as it can only be guessed which conjectures hold the most information for the network. This is why the data sampling methods Random Upsampling (RUS) and Downsampling (RDS) [63] are used over proofs. Random undersampling randomly removes examples from majority classes until the desired balance is achieved, while random downsampling duplicates difficult and rare examples on which the network has performed badly and got a high loss. With this method the data is dynamically balanced during the training and adjusts to the current loss. As some proofs contain much more clauses than other, the initial distribution of the training data is determined by the number of negative and positive clauses combined while the positive ones are weighted stronger.

3.3.2. Loss evaluation

As mentioned in the previous section, the loss function for the score prediction has to be adjusted to the strong class imbalance.

One possible loss function is as proposed in [39] a logistic loss on whether the clause is needed for the proof, for which the output is trained to be $p = 0$, or not, respectively $p = 1$. The labels are switched compared to [39] to be consistent with other clause heuristics for E. However, there is a significant difference in selecting an unnecessary or missing out an important clause. If the network falsely predicts a score of $p = 0$ for a clause, it will be processed as well, but may not increase the runtime noticeably. Despite, if the network calculates a score near to $p = 1$ for a clause that is essential for the proof, the search will not end until all clauses with a lower score are processed. Therefore, the loss function is adjusted to this dilemma.

First of all, the loss function is based on the Binary Cross Entropy (BCE) and therefore depends on the clause's label y. For clauses that should not be processed ($y = 1$) a logistic loss is used, but restricted between 0 and 1. The loss is prevented to convert to infinity for predictions close to zero, because it makes no difference if the network predicts 0.05 or 0.01 for a clause as it would be probably processed anyway. In contrast for clauses with $y = 0$, it is important to reduce false predictions as much as possible because the difference between $p = 0.98$ and $p = 0.99$ can cause many unnecessary clauses to be processed before the clause needed for the proof. This is the logistic loss is not changed here.

When the dataset contains many easy examples compared to a few hard ones, the network should concentrate on the hard ones, especially for clauses with $y = 0$. To handle this imbalance between easy and hard examples, the Focal Loss [37] is adapted for this application to down-weight well classified examples. Overall, the loss function can be summarized as follows:

$$\mathcal{L}(p) = \begin{cases} -\alpha_1 * \log\left(\frac{1+p*(e-1)}{e}\right) * (1-p)^{\gamma} & \text{for } y = 1 \\ -\alpha_0 * \log(1-p) * p^{\gamma} & \text{for } y = 0 \end{cases} \tag{3.1}$$

In experiments the parameters $\alpha_0 = 1.25$, $\alpha_1 = 1$ and $\gamma = 0.5$ lead to the best results.

4. Experiments

After introducing the network architecture, this section presents the implementation and experiments of the model. Therefore, the first part deals with the dataset that is used to train and evaluate the network on. Implementation details of the training and integration in E [60, 61] are discussed in the second subsection. Finally, the experiments of the network on the evaluation dataset are described, and the results on new, previously unsolved proofs in E presented.

4.1. Dataset

The dataset for training and evaluation plays a significant role in the networks' performance. In this project, the TPTP Problem Library [71] is used that contains several test problems for automated theorem proving. The network is trained in a supervised manner for which proofs are deducted by using E. To limit the size of the vocabulary, only a subset of TPTP is selected. Section 4.1.1 reviews the selection process while the next passage deals with the data generated by proof deduction.

4.1.1. Data Selection

As the network learns the features of the vocabulary by itself, it is important that every symbol appears more than once in the dataset. The worst case would be that a symbol is solely included in positive examples. The network would save this in the vocabulary features and ignores the rest of the clauses, although in other, unseen proofs the symbol might appear in negative examples as well.

Therefore, the major selection criterium is to have as many proofs as possible with a minimal vocab. Examples of problem domains like Semantic Web (SWB) and Software Verification (SWV) generate a huge vocabulary with many symbols that are rarely used. Also, small domains like Biology (BIO) and Geography (GEG) are excluded because of too few examples. The selected domains for the dataset are:

- The domains of algebra: LDA (left distributive algebra), REL (relation algebra) and ROB (Robbins algebra).
- The domains of logic: COL (combinatory logic) and LCL (logic calculi).
- Other mathematical problems: FLD (field theory), GRP (group theory), NUM (number theory), RNG (ring theory) and SET (set theory).
- The domains NLP (natural language processing) and PRO (processes). Problems in natural language processing were only selected if they do not include the WordNet axiom set with over 380,000 constants.

The distribution of file numbers can be found in table 1.

As another selection criteria, proof examples deducted by E are also excluded from the training dataset if the proof contains more than 150 positive examples or 100 initial clauses. Examples with too many positive clauses are hard to train providing a high loss, and as a result, the network concentrates on this proof reducing the accuracy for negative examples on all other examples. The initial clauses are also limited because the LSTM only processes 32 clauses per proof in a batch. If a proof contains more initial clauses, context is missing, and the weights would rather be trained to ignore the LSTM output than to pay attention to the knowledge base.

4.1.2. Data Generation

After selecting the problems, proof examples have to be generated. Therefore, E [60, 61] is configured with the three different heuristic settings: FIFO, Clauseweight and Auto208. The First-In/First-Out heuristic has the advantage to find short proves while clauseweight prefers small clauses [62]. Both configurations help the network to learn, as short proves imply fewer positive clauses and less context to extract, while for short clauses the network does not need a very wide receptive field. However, the third configuration, Auto208, deducts the most proves and therefore provides the most examples. It is a hybrid approach of 5 different heuristics with the following settings:

```
Auto208 - configuration G_E__208_C18_F1_SE_CS_SP_PS_SOY
1*ConjectureRelativeSymbolWeight(SimulateSOS,0.5,100,100,100,100,1.5,1.5,1),
4*ConjectureRelativeSymbolWeight(ConstPrio,0.1,100,100,100,100,1.5,1.5,1.5),
1*FIFOWeight(PreferProcessed),
1*ConjectureRelativeSymbolWeight(PreferNoGoals,0.5,100,100,100,100,1.5,1.5,1),
4*Refinedweight(SimulateSOS,3,2,2,1.5,2)
```

The number in front of each heuristic determines the frequency with which a clause is selected from its priority queue. The heuristic *ConjectureRelativeSymbolWeight* takes clauses that are similar to the conjecture, while *Refinedweight* prefers smaller clauses. Additionally, *FIFO* checks that no clause is delayed infinitely. After the heuristics' name, more settings are individually specified for each heuristic (for details see [61]). In general, hybrid approaches are likely to deduct more proofs than single heuristics [60].

Table 1 summarizes the dataset with the number of generated proofs for each heuristic[1]. All proof attempts were executed with a memory limit of 128GB and a CPU time limit of 1800 seconds using the StarExec cluster of the University of Iowa [76]. When applying the FIFO heuristic, a proof can only be found for 824 examples. This set is too small to train a network from scratch as the proofs might be biased. The clauseweight dataset contains more than twice as many examples which might be sufficient for a first training. However, over 3,000 proofs can be generated with the heuristic Auto208. The disadvantage of this dataset might be that the proof structure differs between different problems as multiple heuristics are applied. The network is not able to adjust to one structure and gets a higher loss.

[1]Number of proofs after applying the second selection criteria of maximum 150 positives or 100 initial clauses

To evaluate the effect of different datasets on the model's performance, the network was trained on both Clauseweight and Auto208. The results are presented in the following section 4.3.

Folder	No. Files	FIFO	Clauseweight	Auto208
COL	239	95	129	169
FLD	281	40	90	126
GRP	982	252	489	805
KLE	241	26	62	160
LCL	884	135	294	481
LDA	50	4	6	6
NLP	516	8	39	40
NUM	933	77	284	480
PRO	72	0	27	41
REL	220	9	77	124
RNG	263	25	71	181
ROB	45	9	14	20
SET	1268	144	441	632
Total	5994	824	2023	3265

Table 1: The selected dataset consists of approximately 6000 examples from 13 different domains of the TPTP dataset [71]. Proof attempts in E [60, 61] were performed with the heuristic settings FIFO, Clauseweight and Auto208. The hybrid approach significantly outperforms the other two heuristics finding a proof for more than 50% of all examples. The fewest proofs can be deducted with FIFO only achieving a result of 14%.

4.2. Network implementation in TensorFlow

The following section describes various implementation details of the network architecture. First, the training process is reviewed focusing on the batch processing and applied regularization techniques like data augmentation and normalization. In the second part, the integration of deep learning heuristics in E [60, 61] are explained and parameter settings discussed.

4.2.1. Training

The network is trained on a NVIDIA TitanX Maxwell GPU with 12GB RAM and implemented in the framework TensorFlow [1] using the programming language Python.

A training batch consists of 6 proofs that contain 8 positive and 24 negative examples to align with the dataset distribution. In combination with the adjusted loss function introduced in section 3.3.2 we can still ensure that the lowest loss for a random prediction is close to 0.5 and so the decision is not biased, although much more negative than positive examples are trained. In order to process a batch efficiently, every embedded clause needs to have the same size. Therefore, smaller clauses are padded with a default symbol that is solely used for this purpose. Before the max pooling over channels takes place, the padded symbols are removed so that only the features of the original clause are taken into account. After this step, all clauses consist of 1024 features independent to their size. However, the clause sizes are limited to a maximum of 120 symbols because otherwise the memory usage strongly varies and a smaller batch must be chosen. As only a small proportion of the examples are larger than 120 (less than 1%), the loss of information can be neglected here.

To gain the knowledge base of the LSTM as described in section 3.2.2, the initial clauses need to be embedded as well. Therefore, a subset of 32 initial clauses is appended to every proof that is processed by the embedding network along to the positive and negative examples but is used as an input for the LSTM. The amount is chosen based on the fact that over 90% of all proof examples have equal or less than 32 initial clauses. If a proof includes more than 32, then a randomly selected subset is extracted and used for the current batch. For proofs with less initial clauses, random inputs are added at the end of a batch, but only the final state after the last original clause is used, and further iterations of the LSTM are ignored. To improve the generalization of the knowledge base extraction, the final state is computed several times for the same proof but with a different, randomly shuffled order of the embedded initial clauses. The resulting knowledge bases are spread over the clauses to be evaluated so that in case of 8 different shuffles, each different state is used for one positive and three negative examples. The computational effort does not significantly increase as the LSTM runs with a small batch size (proofs $\times$ shuffles) and has very few layers with fewer features compared to the embedding network.

Another factor that can greatly influence the networks' performance is regularization [17, 33, 52]. Regularization can be defined as "any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error" [17, p. 117]. There are multiple methods to do this, and one of the most important is data augmentation. If the dataset is quite small which is the case especially for positive examples here, the network can easily overfit so that it has a high accuracy on the training data but does not generalize and perform poorly on the test data. To prevent this, the data can be slightly alternated without losing the important information, also called data augmentation. In the context of first-order logic, the literals in a disjunction can be permutated by using the symmetry axiom and also the terms of an equality as defined in equation 2.1. Moreover, variables can be alternated as well because a variable constitutes a quantifier $\forall x$ and the name of the variable is irrelevant. However, it is important that all variables with the same name are changed the same way, as $(odd(X) \rightarrow even(add(X, 1))) \not\simeq (odd(X) \rightarrow even(add(Y, 1)))$. To ensure that the network can distinguish between different vocab and extract important terms/literals based on initial clauses, random clauses with different vocabs are generated and trained as negative. As these examples are easy to learn a low probability of 2% is used that a negative example is randomly generated instead of using it from the dataset.

Next to data augmentation, the weights of the network need to be regularized as well. One approach for that is dropout [69]. The idea is to set the output of some neurons randomly to zero. The network is forced to learn the same features over different neurons so that a better generalization emerges and the risk of overfitting falls. In addition, to limit the weight and output values (optimum of sigmoid input is +/- infinity), a small Mean Squared Error loss of $\text{MSE} = 2 * 10^{-6} * \sum_{i=1}^{n}(w_i^2 - 0)$ is added to the overall training loss. Advanced regularization methods like Batch Normalization [26] or Layer Normalization [5] are, due to lack of memory, only possible with much smaller batch sizes that reduce the generalization again. Experiments with Self-Normalizing Linear Units (SELU) [30] instead of ReLU seem to have a negative effect as the network does not convert anymore and the accuracy is much lower. However, the network is not stable during training, also when using all techniques as described. When analyzing the gradient backflow, the negated conjecture has 32 times higher gradients than the clauses to be evaluated as it is the same for the whole proof. Therefore, the gradients of the negated conjecture are manually weighted down by the factor of 16, and also the gradients of the vocab are reduced by 32. This results in a much more stable training and gains greater accuracy.

4.2.2. Inference

To use a network as clause selection heuristic in E [60, 61], the training network architecture is alternated so that the feature vector of the negated conjecture and the knowledge base of the initial clauses can be extracted at the beginning of the proof search. For later clause evaluations, these features are fed back into the network. This reduces the computational effort to a minimum and removes redundant operations. As a second step, the graph and the trained weights are frozen. Freezing means that the network is exported and can be loaded as a black box where only the required inputs and outputs need to be specified. After freezing, the network file can simply be imported into E using the C-API of TensorFlow [1].

Processing every clause separate takes much more time than gathering multiple clauses and process them together as a batch. The reason for this is that the architecture of a GPU as a SIMD processor (single instruction multiple data) is optimized to process a significant amount of data in parallel and not sequentially [1, 53]. Therefore, the clause evaluation in E is alternated to a batch processing where all newly generated clauses of one DISCOUNT loop iteration are collected first and then evaluated in parallel. As the deep learning heuristic takes several milliseconds to compute for every batch, the network evaluation could also be done asynchronously to the proof search. However, this can cause the network to be many steps delayed to the ATP so that a clause is selected although there could be a better choice in the remaining, not yet evaluated clauses.

When the network is frozen, a fix batch size has to be specified. With a small batch size, the network can adjust the computational effort to the number of clauses the best and has, therefore, less overhead. In contrast, a great batch size has the advantage of a faster evaluation per clause but might process additional clauses as well if the number of clauses is not a multiple of the batch size. During the proof search, the number of generated clauses per iteration greatly deviates from 0 up to more than 2000 so that sometimes great batch sizes of 128 or similar are more efficient and sometimes smaller of 16 or 32. Figure 18 evaluates the average runtime per clause for networks with different batch sizes. If the full batch size can be exploited, the minimum runtime is achieved with 128 clauses per batch. Larger batches seem to exceed the GPU memory so that the runtime slightly increases again. However, as mentioned before, if the number of clauses is not a multiple of the batch size, only a subset of the processed clauses can be used. The computational overhead is determined by recording the number of processed clauses for several different proof searches, and comparing it to the batch size. For 256, the overhead was more than 30%, so that the effective runtime per clause rises to 0.54ms. The fastest evaluation is achieved for a batch size of 128 with an average runtime of 0.46ms per clause, although the computational overhead is about 15%. Nevertheless, to not waste computation time and use this overhead efficiently, free spots are filled up with different augmentations of original clauses of the batch. The evaluation of a clause arises from the average of all its variations preventing random false classifications.

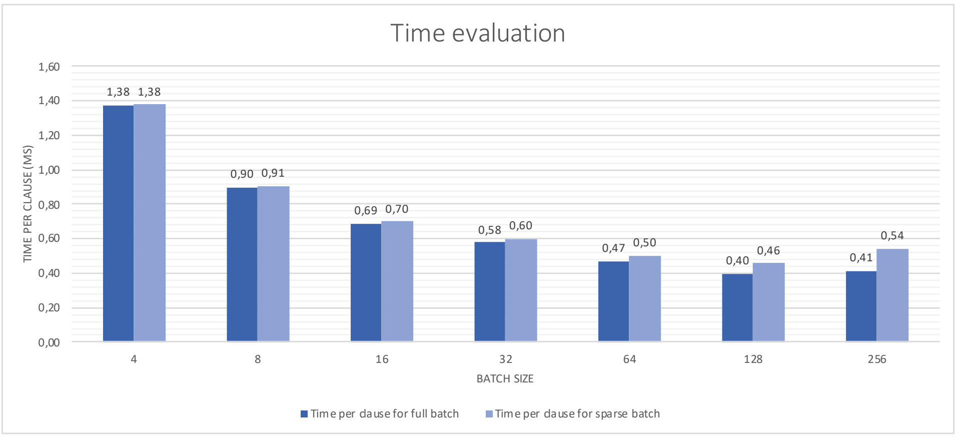


Figure 18: The diagram summarizes the average runtime needed to evaluate a clause when using a certain batch size on a NVIDIA TitanX Maxwell GPU. The dark blue columns visualize the case that all clauses in a batch need to be evaluated. In contrast, examples of numbers of clauses that need to be processed per step are used to evaluate the effective runtime, presented by the light blue columns. Overall, a batch size of 128 constitutes the minimal effective runtime of 0.46ms per clause.

4.3. Experimental results

This section presents experimental results of the deep learning heuristic by examining two tasks: Is the network able to determine if a clause is useful for a proof or not for unseen problems which are solved by classic heuristics like in the training dataset? Moreover, can the approach find new proofs for problems that are not solved by heuristics yet? Therefore, the first paragraph compares the accuracy of models trained on different datasets. The second subsection discusses the results on new, unsolved proofs.

4.3.1. Accuracy on test data

The trained networks are tested on selected positive and negative examples that were not used for training. The performance is examined on all examples of a proof to simulate a full proof search without using E. Thus, the corresponding dataset consists of 11 proofs (FLD010-1, GRP001-4, GRP018-1, GRP150-1, KLE144+1, LCL110-2, LDA007-2, NUM559+1, RNG090+1, SET840-2, SET602+4) including 296 positive and 1659 negative examples. The network calculates a score for each clause and the prediction is classified as positive, if the evaluation is lower than 0.5, and negative otherwise. The accuracy of a classifier for a certain class c is defined as [17]:

$$ACC_c = \frac{TP_c + TN_c}{TP_c + TN_c + FP_c + FN_c} \tag{4.1}$$

where TP (true positives) is the number of examples that are correctly classified as this class, TN (true negatives) the examples that are correctly classified as not this class, FP (false positive) the examples that are falsely classified as this class, and FN (false negatives) the examples that are falsely classified as not this class. As the task of clause selection only has the two categories “useful” (0) and “not useful” (1), the FN of one class is the FP of the other class, and similar for TP and TN. Thus, the average accuracy of both classes can be simplified to:

$$ACC_{avg} = \frac{TP_0 + TP_1}{P_0 + P_1} \tag{4.2}$$

with P_c as the number of positive examples for class c. Since the test dataset is reasonably biased towards negative examples, the overall accuracy is the average of both classes normalized. This constitutes a 50-50% split of negative and positive examples, as used in [39].

Table 2 shows the results for the network architecture presented in section 3. Three datasets were used for training that differ in the heuristic and example settings. The first one was solely trained on examples generated by using clauseweight as heuristic in E. The accuracy exceeds the baseline of the CNN and WaveNet from [39], especially for positive clauses. However, the accuracies cannot be compared in detail because both networks from [39] were trained on a different dataset (Mizar [19]) and, thus, tested on different proofs.

When applying the first model for proof search, many wrong clauses are selected so that even proofs on which the network was trained are solved slower. The problem is that the network has only seen clauses that the heuristic has selected in the training

Network/Dataset	Accuracy Positive	Accuracy Negative	Accuracy overall
Auto - CNN [39]	-	-	80.3%
Auto - WaveNet [39]	-	-	81.5%
Clauseweight	86.4%	81.8%	84.1%
Clauseweight with Unprocessed	88.7%	85.3%	87.0%
Auto208 with Unprocessed	**92.6%**	**89.2%**	**90.9%**

Table 2: The CNN and WaveNet architecture by [39] achieved an accuracy of 80% to 81.5% on a 50-50% split of positive and negative examples of the Mizar dataset. The presented network of section 3 is evaluated on unseen examples of the generated TPTP dataset trained on either the results of the heuristic "clauseweight" or "auto208". The best model reaches an accuracy of 90.9% reducing the error by almost half.

dataset. Clauses that stay unprocessed for the whole proof are ignored since [39] has reported that learning these as negatives reduces the performance. However, during inference, the network evaluates such clauses and might falsely classify them as useful. Therefore, a second model is trained with unprocessed clauses included as negative. The test dataset is not updated with unprocessed clauses as this would manipulate the results and For the network in this project, the accuracy is improved by using this dataset for training. Especially the performance on negative clauses rises by 3.5% resulting in an overall accuracy of 87.0%.

As discussed in section 4.1.2, most example proofs were generated by using the Auto208 heuristic in E. Therefore, a third model was trained on this dataset including unprocessed examples. This configuration outperforms the previous ones by about 4% in accuracy for both negative and positive examples resulting in 90.9% correct classifications. The error rate is reduced by almost half compared to the best model of [39].

Nevertheless, even if the high accuracy seems impressive, the performance has to be interpreted in the context of proof deduction. Recognizing over 90% of all positive clauses might be enough when using it in combination with other heuristics like Auto208. The proportion of correct classified negatives is the greater problem. When processing a clause, up to 2000 and more new clauses are generated. As most proofs consists of less than 100 positive clauses, most of the generated ones are not useful. If 90% of them are correctly classified, the network still suggests approximately 200 of these to be useful and should be processed next. The few correctly positive clauses might never be selected, because some examples are processed before them and generate many more clauses that are placed in the queue in front of them again.

This theoretical view on applying the heuristic might conclude that the accuracy on negative clauses have to be sharply increased. However, the accuracy only constitutes the number of clauses with an weight of more or less than 0.5. If the positive clauses have a much lower score than the false classified ones, the performance might still be sufficient. Therefore, the next section discusses the results when actually applying a trained model as a clause heuristic in E.

4.3.2. Proof search in E

As proposed in [39], a switched approach with network guidance at the beginning and auto heuristic afterwards is implemented in E. The first phase has a hybrid heuristic with a frequency of 20 for the deep learning algorithm and 1 for each other heuristic of Auto208. So, the network guidance dominates but still little impacts can be given by the classic heuristics. After 20 minutes, the frequencies are alternated to 2 for the network and to the standard values of Auto208 for the other heuristics. During the second phase, no new generated clauses are processed by the neural network and the evaluations for these are set to 1 (maximum of not useful). Still, the clauses that were already evaluated are used and processed in their priority order for the network heuristic as the evaluations might still be valuable.

First, the network guidance was tested on the dataset on which the accuracy of the model was determined in section 4.3.1. All of the eleven proofs were solved by the network trained on the auto heuristic dataset in the time limit of 30 minutes and about two thirds when solely applying the deep learning heuristic. However, the model trained without unprocessed clauses was only able to solve the problem NUM559+1 when using purely network guidance although its accuracy on the test dataset was about 84%. This underlines the relevance of unprocessed clauses in the training process for the future application.

For testing the models on new problems for which none heuristic could find a proof, the following 25 files were randomly selected:

- Domain “Field Theory” (FLD): FLD082-1 and FLD085-3
- Domain “Group Theory” (GRP): GRP124-1, GRP207-1 and GRP781-1
- Domain “Left Distributive Algebra” (LDA): LDA005-1, LDA015-1, LDA025-1 and LDA026-1
- Domain “Processes” (PRO): PRO003+1, PRO004+1, PRO007+2 and PRO014+1
- Domain “Relation Algebra” (REL): REL019+2, REL029+4, REL029-2, REL039+1 and REL053+1
- Domain “Robbins Algebra” (ROB): ROB007-3, ROB012-1, ROB032-1 and ROB033-1
- Domain “Set Theory” (SET): SET069+1, SET652+3 and SET993+1

The model of the clauseweight dataset had no success on any of the problems, even when including unprocessed clauses in the training process. However, it was conspicuous that the network often selected clauses that contain a part of a different positive clause. For example, in case of the problem SET652+3, the clause `subset(X1,X3) | ~subset(X2,X3) | ~subset(X1,X2)` was selected as a positive clause. As a result, similar clauses like `subset(X1,X2) | ~subset(X1,X3) | ~subset(X4,X2) | ~subset(X3,X4)` and `subset(X1,X2) | ~subset(X1,X3) | ~subset(X3,X4) | ~subset(X5,X2) | ~subset(X4,X5)` with increasing size (up to 12 literals) were selected as well. The problem is that the network architecture is based on a convolutional model and only has a local view although the receptive field is greatly increased by the dilated convolutions. The final features of a symbol therefore depend on a part of a clause and not all of it. Moreover, the max pooling over channels basically ignores symbols that have small feature values. The network might represent unnecessary parts with low features as they are not so relevant for the evaluation, but also whole literals can be cut out due to the max pooling. Applying an extensive mining of negative examples in the training process as described in section 3.3.1, reduces the number of such selections as these get high losses and trained more often. However, this is a workaround and can be efficiently implemented in the network architecture. So, more work has to be invested in the final feature compression to prevent this problem.

When using a model trained on the Auto208 dataset, at least one new proof can be found. The problem file REL019+2 was solved in about 22 minutes, shortly after the heuristic switched. All 60 clauses that were useful for the proof can be found in the appendix A. 58 of them were selected in the network guided phase, whereas only the last two clauses were chosen by the substantial auto heuristic. This shows that the deep learning approach has a great impact and can on the proof search. Still, it also supports the hypothesis that the switch approach outperform pure network guidance as the final clauses were selected by the auto heuristic only 2 minutes after the frequencies changed.

However, the sample of 25 proof attempts that was limited by the lack of resources with a single GPU, is not sufficient for a detailed evaluation. Proving one out of 25 can be a lucky hit, or might even be lower than the actual performance of the network. Nevertheless, the first proof on new problems and the great accuracy of unseen proofs demonstrate the potential of deep learning heuristics for automated theorem proving.

5. Outlook

This project was one of the first steps integrating deep learning in automated theorem proving. The proposed network architecture was able to correctly classify 90% of all clauses in the test dataset whether they are likely to be useful for a proof or not. However, applying the network as a heuristic for clause selection on new, unsolved problems, has brought little success due to multiple reasons. Thus, there is still a lot of space for improvement, that is discussed in this section.

First of all, the network contains many hyperparameters that have to be optimized. The structure of a training batch regarding number of proofs, examples and initial clauses, can significantly affect the network's training process and its final performance. In comparison to a network without a knowledge base extraction, the batches are strongly biased as it only contains 6 different negated conjectures and 192 initial clauses that are not trained explicitly. Moreover, the learning rate and the corresponding gradient flow is hard to control for this network, as the input vocabulary is trained as well and the dilated dense block enables multiple backpropagation paths. The weights for embedding the negated conjecture are trained way less than the ones for the clauses to be evaluated because of the same conjecture for multiple examples in a batch. Reducing the gradients for this path stabilizes the network, but might not be the optimal solution. This drawback of the knowledge base extraction has to be examined more, but not all experiments could be executed due to limited resources with a single GPU.

In addition, the approach of the knowledge base extraction is not fully developed. The idea is that the network gets an overview of all initial clauses and learn dependencies. Nevertheless, it can not be checked if the network really does that as the features are not readable for humans and analyzing a neural network on what it has learned is mostly hardly possible [34]. Still, if the network would learn something from the initial clauses, its performance should improve. Therefore, the same network architecture excluding the LSTM and the knowledge base extraction has to be trained and compared to the current one. This experiments will demonstrate whether the intial clauses are helpful for the network or even decrease its performance due to the constraints of the training process.

Another constraint for a successful knowledge base extraction is that the resulting features are independent from the chronological order in which the initial clauses are processed. To ensure this, several different sequences are computed in a training iteration and randomly assigned to examples. However, the features might still deviate as they are not compared with each other. Thus, an additional loss like the Kullback Leibniz distance $\sum_i p_i \log \frac{p_i}{q_i}$ between feature vectors of different orders has to be considered that checks the similarity of the knowledge bases. The weight of this loss compared to the evaluations has to optimized and constitutes a new hyperparameter. If the loss is too high, the network might ignore the knowledge base for evaluation and computes the same features for all proofs. On the other side, if the loss is too low, it will not effect the network at all.

Not only the network architecture can be improved. Also the task of how to represent a clause as an input to the embedding network is a key aspect. Currently, the input solely consists of a table lookup of features for the symbol's name and arity. However, clauses have a strong syntax that gets lost and has no representation in the network. To implement the syntax and more information of a clause in the input, the following parameters can added:

Negative literals One input that determines whether the symbol is in a negative (-1) or positive (1) literal. This reduces the long-term dependency to the negated symbol for large literals.

One-hot vector for the symbol depth The depth of a symbol is the number of functions around it. For example, in the clause $odd(add(X, 1))$, odd has the depth 0, add the depth 1, and X and 1 the depth 2. It is encoded with a one-hot vector with a maximum value of 4 to ensure that only one depth is taken at a time. The network can easier recognize parameters of a function, especially if the parameters are functions as well, and parentheses might be redundant reducing the input size a lot.

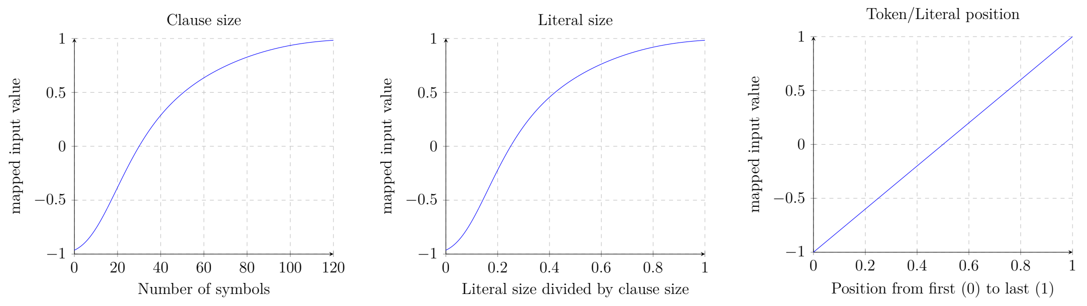
One-hot vector for the literal identity Every literal in a clause is assigned to a different id. For example, in the clause $odd(X) \vee even(X)$, both literals have different ids. This ensures that large receptive fields like the last dilated convolution in the embedding network can still distinguish between symbols of different literals. The size of the one-hot vector is limited to for example 4 different ids that do not give any information of the literal itself but just constitutes a difference to the neighbors. If the clause contains more than 4 literals, ids are repeated, but with a maximum distance.

Clause size The size of the whole clause can also improve the input features. As the inputs should be normalized for the best performance [17], a function with a maximum of 1 and minimum of -1 is applied to the clause size. However, as the average length is at about 30, the function should also be close to 0 at 30 to get a greater resolution for small clauses. A possible function is $\tanh(2 * \frac{(x-30)}{(x+30)} * (1 + (\frac{x}{120})^3))$ as it is everywhere differentiable and meets the constraints above. The last factor of $(1 + (\frac{x}{120})^3)$ increases the resolution for large clauses. The function is shown in figure 19a.

Literal size Next to the clause, also the size of the literals might be important. Therefore, a similar function is used to map the value between -1 and 1. Nevertheless, the literal size and its average depends on the clause size. Thus, the function is 0 for the average literal size (clause size divided by number of literals) and its inputs are normalized to the clause size. Figure 19b illustrates this function.

Position Next to the literal size, it is important to know at which position the symbol is in the literal. This information can be mapped with a linear interpolation from -1 if the symbol is the first one of the literal, and 1 for the last (see figure 19c). A non-linear function is not needed here as the average of the interpolation is always

0. With the symbol position, the network can orientate itself better within a literal and improves long-term dependencies.



(a) Clause size (b) Literal size (c) Symbol position

Figure 19: (a) The clause size is mapped by the function $\tanh(2 * \frac{(x-30)}{(x+30)} * (1 + (\frac{x}{120})^3))$ into a range between -1 and 1. The resolution for is for smaller values greater as the average clause size is about 30. (b) The same function is used for the literal size, but it is normalized to the clause size. The average length at which the function is 0 is 1 divided by the number of literals in the clause. (c) The position of a symbol in a literal is the interpolated value between -1 for the first symbol and 1 for the last.

Adding these twelve inputs to the input would barely help as the 512 other input features clearly overweight. Moreover, the regularization techniques limit weights to a certain boundary and tries to level them so that none stand out. As an alternative, a 1x1 convolution can be used to scale the twelve inputs up to 256 features which are equally trained with the vocabulary. In addition, the network can learn to combine those inputs and learn new features from it. This approach might help the network to understand the syntax of a clause and lead to a better performance.

Another key aspect for the network's performance that is easily ignored is the dataset although it is very important as stated by many researches [34, 17, 81, 52]:

"Your neural network is only as good as the data you feed it." [52]

In this project, proofs of the TPTP problem library [71] were generated by E [60, 61] with different heuristic settings and trained in a supervised manner. The dataset is currently quite small as networks overfit on clausweight data after only 30.000 iterations. One of the biggest problems might be the divergence of the different domains and their vocabulary. Many vocabs are only used in one or two domains, and amounts of examples significantly differ over domains so that the vocabulary is biased. The dataset would be improved by using only one large domain with enough examples like the Mizar dataset [19]. As another approach, to use the same vocabulary for almost every proof, a standard vocabulary of 100 constants, 100 functions, 50 variables and more can be created, on which he proof vocabulary is mapped dynamically. The number of vocab per type is determined by the maximum usage of a proof in the testing and training dataset. At training time, the vocabulary mapping is shuffled for every proof in a batch so that the network learns the structure of a proof independent of its vocabulary. However, as the name of a vocab is now irrelevant, the vocabulary is not trained anymore and initialized it with a random distribution of -1, 0 and 1, because it can lead to a unstable training.

Nevertheless, supervised learning is still a simplification of the real environment. The future of deep learning with automated theorem proving lays in reinforcement learning where the network is trained on proof searches that are generated by its own selections. Each iteration consists of a proof attempt conducted by the network, and when a proof is found, the model is retrained on the correct and wrong choices it made. Also more complex and stronger network architectures that actively implement planning like the Value Iteration Network [75] can be used with reinforcement learning. This approach is very difficult to realize as proof attempts last several minutes and extremely slows down the training. But only with reinforcement learning, the model is independent to other heuristics, optimizing itself until saturation and can achieve a performance way beyond current state-of-the-art on supervised learning.

6. Conclusion

Automated theorem provers are computer programs that try to find formal proofs of theorems. The hypothesis is represented as a conjecture that should be deducted from a set of ground rules, called axioms. In first order logic, the axioms are converted into a clause set. In combination with the negation of the conjecture, new clauses are derived until the empty clause is generated witnessing the implication of the theorem by the axioms, or no other non-redundant implications can be made. As there is a great number of possible derivations, one of the key aspects in ATP is to design an algorithm that decides when to compute which inference.

Modern automated theorem provers like Vampire [55, 32], Prover9 [43], SPASS [80] and E [60, 61], rely on variants of the *given clause* algorithm, where the clause set is divided into a processed and an unprocessed part. At each iteration, a clause from the unprocessed set is selected and all inferences with previously processed clauses are computed. The chronological order, in which the clauses are moved to the processed set, is determined by a clause selection heuristic like FIFO or Clauseweight, and has a great impact on the prover's performance. Nevertheless, classic heuristics show moderate success on efficient proof searches. As deep learning has shown great improvements in several domains like Computer Vision [33, 54], Natural Language Processing [16, 58] and Planning [65, 75], this project deals with the development of a neural network architecture for a clause section heuristic.

In this project, the clause to be evaluated and the negated conjecture are compressed to a fixed sized feature vector by an embedder network. In order to do that, a vocabulary is used to replace symbols with features, a dilated dense block is applied as principal component and a final max pooling over features reduces the size to 1024. Afterwards, a second model, the combiner network, computes the final evaluation of the clause based on both feature vectors. Moreover, a knowledge base, extracted from the initial clauses by an adjusted LSTM [22], is used as an additional input. Another focus of this work is handling the imbalance of positive and negative examples by using extensive mining and sampler techniques as well as an new, adapted loss function.

The network is trained in a supervised manner on proofs of a subset of the TPTP problems library [71] generated by different heuristic settings of E. Unprocessed clauses are included as negative examples, as this seem to be crucial for later application in E. In experiments on unseen proofs, the model has demonstrated a classification accuracy of over 90% clearly outperforming current state-of-the-art architecture proposed by [4, 39]. Nevertheless, when applying the network as clause selection heuristics on problems that could not be previously solved by classic heuristics, the network guidance shows moderate success with only one newly solved proof out of a sample of 25 problems. Although the results are not as good as expected, the great accuracy show the potential of deep learning algorithms for heuristics.

However, many open challenges regarding the network architecture, data handling and training methods remain that are future steps to aim creating a deep learning clause selection heuristic beyond state of the art.

References

[1] Abadi, M. *et al.*: Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Abbeel, P. and Ng, A.Y.: Apprenticeship learning via inverse reinforcement learning. In: Proceedings of the twenty-first international conference on Machine learning, p. 1. ACM, 2004.

[3] Aghdam, H.H. and Heravi, E.J.: Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification. Springer International Publishing, Cham (Switzerland), first edition, 2017.

[4] Alemi, A.A. *et al.*: DeepMath - Deep Sequence Models for Premise Selection. In: D.D. Lee/ M. Sugiyama/ U.V. Luxburg/ I. Guyon and R. Garnett, editors, Advances in Neural Information Processing Systems, 29, pp. 2235–2243. Curran Associates, Inc., 2016. `1606.04442`.

[5] Ba, J.L./ Kiros, J.R. and Hinton, G.E.: Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[6] Bengio, Y./ Simard, P. and Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, volume 5, no. 2, pp. 157–166, March 1994.

[7] Bridge, J.P.: Machine learning and automated theorem proving. Ph.D. thesis, University of Cambridge, Computer Laboratory, 2010.

[8] Cho, K. *et al.*: On the properties of neural machine translation: Encoder-decoder approaches. *Computing Research Repository (CoRR)*, volume abs/1409.1259, 2014.

[9] da Silva, I.N. *et al.*: Artificial Neural Networks: A Practical Course. Springer International Publishing, Switzerland, first edition, 2017.

[10] Denzinger, J./ Kronenburg, M. and Schulz, S.: DISCOUNT: A Distributed and Learning Equational Prover. *Journal of Automated Reasoning*, volume 18, no. 2, pp. 189–198, 1997. Special Issue on the CADE 13 ATP System Competition.

[11] Denzinger, J. and Schulz, S.: Learning Domain Knowledge to Improve Theorem Proving. In: M. McRobbie and J. Slaney, editors, Proc. of the 13th CADE, New Brunswick, *LNAI*, volume 1104, pp. 62–76. Springer, 1996.

[12] Denzinger, J. *et al.*: Learning from Previous Proof Experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999.

[13] Finn, C./ Goodfellow, I.J. and Levine, S.: Unsupervised learning for physical interaction through video prediction. *Computing Research Repository (CoRR)*, volume abs/1605.07157, 2016.

[14] Ganzinger, H.: Saturation-based theorem proving. In: International Colloquium on Automata, Languages, and Programming, pp. 1–3. Springer, 1996.

[15] Gers, F.A. and Schmidhuber, J.: Recurrent nets that time and count. In: IJCNN (3), pp. 189–194, 2000.

[16] Goldberg, Y.: A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, volume 57, pp. 345–420, 2016.

[17] Goodfellow, I./ Bengio, Y. and Courville, A.: Deep Learning. MIT Press, Cambridge, Massachusetts, first edition, 2016.

[18] Goodfellow, I. *et al.*: Generative adversarial nets. In: Z. Ghahramani/ M. Welling/ C. Cortes/ N.D. Lawrence and K.Q. Weinberger, editors, Advances in Neural Information Processing Systems 27, pp. 2672–2680. Curran Associates, Inc., 2014.

[19] Grabowski, A./ Korniłowicz, A. and Naumowicz, A.: Four decades of mizar. *Journal of Automated Reasoning*, volume 55, no. 3, pp. 191–198, 2015.

[20] Hochreiter, S.: Untersuchungen zu dynamischen neuronalen netzen. *Master's thesis, Institut fuer Informatik, Technische Universitaet, Muenchen*, 1991.

[21] Hochreiter, S. and Schmidhuber, J.: Long short-term memory. *Neural Computation*, volume 9, no. 8, pp. 1735–1780, 1997.

[22] Hochreiter, S. and Schmidhuber, J.: Long short-term memory. *Neural Computation*, volume 9, no. 8, pp. 1735–1780, 1997.

[23] Holroyd, C.B. and Coles, M.G.: The neural basis of human error processing: reinforcement learning, dopamine, and the error-related negativity. *Psychological review*, volume 109, no. 4, p. 679, 2002.

[24] Huang, G. *et al.*: Densely connected convolutional networks. *arXiv preprint arXiv:1608.06993*, 2016.

[25] Huang, G. *et al.*: Densely connected convolutional networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2261–2269, 2017.

[26] Ioffe, S. and Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: F. Bach and D. Blei, editors, Proceedings of the 32nd International Conference on Machine Learning, *Proceedings of Machine Learning Research*, volume 37, pp. 448–456. PMLR, Lille, France, 07–09 Jul 2015.

[27] Kaelbling, L.P./ Littman, M.L. and Moore, A.W.: Reinforcement learning: A survey. *Journal of artificial intelligence research*, volume 4, pp. 237–285, 1996.

[28] Karpathy, A.: Cs231n convolutional neural networks for visual recognition, 2017. Retrieved from http://cs231n.github.io/convolutional-networks/, last viewed on 14 May 2018.

[29] Kaufmann, M. and Moore, J.: Some key research problems in automated theorem proving for hardware and software verification. volume 98, 01 2004.

[30] Klambauer, G. *et al.*: Self-Normalizing Neural Networks. In: I. Guyon/ U.V. Luxburg/ S. Bengio/ H. Wallach/ R. Fergus/ S. Vishwanathan and R. Garnett, editors, Advances in Neural Information Processing Systems, volume 30, pp. 971–980. Curran Associates, Inc., 2017. 1706.02515.

[31] Kolata, G.: Computer math proof shows reasoning power (the new york times), December 10, 1996. Retrieved from https://archive.nytimes.com/www.nytimes.com/library/cyber/week/1210math.html, last viewed on 3 June 2018.

[32] Kovács, L. and Voronkov, A.: First-order theorem proving and Vampire. In: N. Sharygina and H. Veith, editors, Proc. of the 25th CAV, *LNCS*, volume 8044, pp. 1–35. Springer, 2013.

[33] Krizhevsky, A./ Sutskever, I. and Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp. 1097–1105, 2012.

[34] LeCun, Y./ Bengio, Y. and Hinton, G.: Deep learning. *Nature*, volume 521, no. 7553, pp. 436–444, 2015.

[35] LeCun, Y. *et al.*: Backpropagation applied to handwritten zip code recognition. *Neural computation*, volume 1, no. 4, pp. 541–551, 1989.

[36] LeCun, Y. *et al.*: Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, volume 86, no. 11, pp. 2278–2324, 1998.

[37] Lin, T. *et al.*: Focal loss for dense object detection. In: IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017, pp. 2999–3007, 2017.

[38] Liu, W. *et al.*: Ssd: Single shot multibox detector. In: European conference on computer vision, pp. 21–37. Springer, 2016.

[39] Loos, S. *et al.*: Deep Network Guided Proof Search. In: T. Eiter and D. Sands, editors, Proc. 21st LPAR, Maun, Botswana, *EPiC Series in Computing*, volume 46, pp. 85–105, 2017.

[40] Luo, W. *et al.*: Understanding the effective receptive field in deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 4898–4906, 2016.

[41] McCune, W.: Solution of the robbins problem. *Journal of Automated Reasoning*, volume 19, no. 3, pp. 263–276, 1997.

[42] McCune, W. and Wos, L.: Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, volume 18, no. 2, pp. 211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[43] McCune, W.W.: Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010. (acccessed 2016-03-29).

[44] Mitchell, T.M.: Machine Learning. McGraw-Hill, Inc., New York, NY, USA, first edition, 1997.

[45] Mnih, V. *et al.*: Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[46] Mueller, E.T.: Commonsense reasoning: an event calculus based approach. Morgan Kaufmann, 2014.

[47] Nair, V. and Hinton, G.E.: Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning*, , no. 3, pp. 807–814, 2010. `1111.6189v1`.

[48] Nguyen-Tuong, D. and Peters, J.: Model learning for robot control: a survey. *Cognitive processing*, volume 12, no. 4, pp. 319–340, 2011.

[49] Nieuwenhuis, R. and Rubio, A.: Chapter 7 – Paramodulation-Based Theorem Proving. In: Handbook of Automated Reasoning, pp. 371–443, 2001.

[50] Nonnengart, A. and Weidenbach, C.: Computing Small Clause Normal Forms. In: A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning, volume I, chapter 5, pp. 335–367. Elsevier Science and MIT Press, 2001.

[51] Olah, C.: Understanding lstm networks, August 2015. Retrieved from http://colah.github.io/posts/2015-08-Understanding-LSTMs/, last viewed on 16 May 2018.

[52] Raj, B.: Data augmentation, how to use deep learning when you have limited data - part 2, 2018. Retrieved from https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced, last viewed on 22 May 2018.

[53] Rauber, T. and Rünger, G.: Parallele Programmierung. eXamen.press. Springer Berlin Heidelberg, 2012.

[54] Redmon, J. *et al.*: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 779–788, 2016.

[55] Riazanov, A. and Voronkov, A.: The Design and Implementation of VAMPIRE. *Journal of AI Communications*, volume 15, no. 2/3, pp. 91–110, 2002.

[56] Rumelhart, D.E. *et al.*: Learning representations by back-propagating errors. *Cognitive modeling*, volume 5, no. 3, p. 1, 1988.

[57] Russakovsky, O. *et al.*: ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, volume 115, no. 3, pp. 211–252, 2015.

[58] Sak, H./ Senior, A. and Beaufays, F.: Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*, 2014.

[59] Schulz, S.: Where, what, and how? lessons from the evolution of e. Invited talk at the 19th LPAR, 2013, Stellenbosch, South Africa.

[60] Schulz, S.: E – A Brainiac Theorem Prover. *Journal of AI Communications*, volume 15, no. 2/3, pp. 111–126, 2002.

[61] Schulz, S.: System Description: E 1.8. In: K. McMillan/ A. Middeldorp and A. Voronkov, editors, Proc. of the 19th LPAR, Stellenbosch, *LNCS*, volume 8312, pp. 735–743. Springer, 2013.

[62] Schulz, S. and Möhrmann, M.: Performance of clause selection heuristics for saturation-based theorem proving. In: N. Olivetti and A. Tiwari, editors, Proc. of the 8th IJCAR, Coimbra, *LNAI*, volume 9706, pp. 330–345. Springer, 2016.

[63] Seiffert, C. *et al.*: Rusboost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, volume 40, no. 1, pp. 185–197, 2010.

[64] Shi, X. *et al.*: Convolutional LSTM network: A machine learning approach for precipitation nowcasting. *Computing Research Repository (CoRR)*, volume abs/1506.04214, 2015.

[65] Silver, D. *et al.*: Mastering the game of Go with deep neural networks and tree search. *Nature*, volume 529, no. 7587, pp. 484–489, January 2016.

[66] Singh, S.P. *et al.*: Robust reinforcement learning in motion planning. In: Advances in neural information processing systems, pp. 655–662, 1994.

[67] Socher, R. and Lin, C.: Parsing natural scenes and natural language with recursive neural networks. In: Proceedings of the 28th international conference on machine learning (ICML-11), pp. 129–136, 2011. `arXiv:1207.6324`.

[68] Soutner, D. and Müller, L.: Application of LSTM Neural Networks in Language Modelling, pp. 105–112. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[69] Srivastava, N. *et al.*: Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, volume 15, pp. 1929–1958, 2014. `1102.4807`.

[70] Sundermeyer, M./ Schlüter, R. and Ney, H.: Lstm neural networks for language modeling. In: Thirteenth Annual Conference of the International Speech Communication Association, 2012.

[71] Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, volume 59, no. 4, pp. 483–502, 2017.

[72] Sutcliffe, G.: The world of automated reasoning, (n.d.). Retrieved from http://www.cs.miami.edu/ tptp/Seminars/ATP/WorldOfARGIF.html, last viewed on 23 May 2018.

[73] Sutton, R.S. and Barto, A.G.: Introduction to Reinforcement Learning. MIT Press, Cambridge, MA, USA, first edition, 1998.

[74] Tai, K.S./ Socher, R. and Manning, C.D.: Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *arXiv preprint arXiv:1503.00075*, 2015. 1503.00075.

[75] Tamar, A. *et al.*: Value iteration networks. In: D.D. Lee/ M. Sugiyama/ U.V. Luxburg/ I. Guyon and R. Garnett, editors, Advances in Neural Information Processing Systems, 29, pp. 2154–2162. Curran Associates, Inc., 2016. 1602.02867.

[76] The University of Iowa: Starexec, 2018. Retrieved from https://www.starexec.org/starexec/public/about.jsp, last viewed on 24 May 2018.

[77] van den Oord, A. *et al.*: WaveNet: A generative model for raw audio. *CoRR*, volume abs/1609.03499, 2016. 1609.03499.

[78] Versteegh, M. *et al.*: The zero resource speech challenge 2015. In: Interspeech, pp. 3169–3173, 2015.

[79] Villegas, R. *et al.*: Decomposing motion and content for natural video sequence prediction. In: Conference on Computer Vision and Pattern Recognition (CVPR), 2017.

[80] Weidenbach, C. *et al.*: SPASS Version 3.0. In: F. Pfenning, editor, Proc. of the 21st CADE, Bremen, *LNAI*, volume 4603, pp. 514–520. Springer, 2007.

[81] Weiss, G.M.: Mining with rarity: a unifying framework. *ACM Sigkdd Explorations Newsletter*, volume 6, no. 1, pp. 7–19, 2004.

[82] Yu, F. and Koltun, V.: Multi-scale context aggregation by dilated convolutions. *Computing Research Repository (CoRR)*, volume abs/1511.07122, 2015.

A. Network guided proof example

Here, the full proof for the problem REL019+2 of the TPTP library [71] found by network guidance is presented. The proof attempt took about 22 minutes and consists of 60 positive and 2614 negative examples. Over 95% of positive clauses were selected during the first phase mainly by the network guidance (highlighted in blue) while the hybrid heuristic Auto208 is responsible for the last two clauses (highlighted in green).

```
cnf(c-0-34, plain, (join(X1,X2)=join(X2,X1))).
cnf(c-0-19, plain, (converse(converse(X1))=X1)).
cnf(c-0-18, plain, (composition(converse(X2),converse(X1))=converse(composition(X1,X2)))).
cnf(c-0-24, plain, (composition(X1,one)=X1)).
cnf(c-0-43, plain, (join(X1,complement(X1))=top)).
cnf(c-0-23, plain, (converse(composition(converse(X2),X1))=composition(converse(X1),X2))).
cnf(c-0-29, plain, (composition(converse(one),X1)=X1)).
cnf(c-0-33, plain, (join(composition(converse(X1),complement(composition(X1,X2))),complement(X2))=complement(X2))).
cnf(c-0-42, plain, (complement(join(complement(X1),complement(complement(X1))))=zero)).
cnf(c-0-35, plain, (converse(one)=one)).
cnf(c-0-41, plain, (composition(one,X1)=X1)).
cnf(c-0-46, plain, (join(complement(X1),complement(X1))=complement(X1))).
cnf(c-0-50, plain, (join(zero,zero)=zero)).
cnf(c-0-108, negatedconjecture, (composition(complement(join(complement(esk10),complement(esk20))),top)!=
complement(join(complement(esk10),complement(esk20))))).
cnf(c-0-49, plain, (join(join(X1,X2),X3)=join(X1,join(X2,X3)))).
cnf(c-0-52, plain, (join(zero,join(zero,X1))=join(zero,X1))).
cnf(c-0-48, plain, (join(complement(join(complement(X1),complement(X2))),complement(join(complement(X1),X2)))=X1)).
cnf(c-0-53, plain, (join(zero,complement(complement(X1)))=X1)).
cnf(c-0-54, plain, (join(zero,X1)=X1)).
cnf(c-0-95, negatedconjecture, (composition(esk10,top)=esk10)).
cnf(c-0-90, negatedconjecture, (composition(esk20,top)=esk20)).
cnf(c-0-55, plain, (complement(complement(X1))=X1)).
cnf(c-0-63, plain, (join(converse(X1),converse(X2))=converse(join(X1,X2)))).
cnf(c-0-56, plain, (join(X1,X1)=X1)).
cnf(c-0-57, plain, (join(X1,join(X1,X2))=join(X1,X2))).
cnf(c-0-59, plain, (join(X1,top)=top)).
cnf(c-0-62, plain, (join(top,X1)=top)).
cnf(c-0-61, plain, (join(X1,join(complement(X1),X2))=join(top,X2))).
cnf(c-0-88, plain, (join(composition(X1,X3),composition(X2,X3))=composition(join(X1,X2),X3))).
cnf(c-0-91, plain, (join(X1,composition(X2,X1))=composition(join(one,X2),X1))).
cnf(c-0-64, plain, (join(complement(join(complement(X1),X2)),complement(join(complement(X2),complement(X1))))=X1)).
cnf(c-0-67, plain, (complement(join(complement(X1),complement(join(X1,X2))))=X1)).
cnf(c-0-93, plain, (converse(composition(X2,converse(X1)))=composition(X1,converse(X2)))).
cnf(c-0-69, plain, (join(complement(X1),complement(join(X1,X2)))=complement(X1))).
cnf(c-0-71, plain, (join(complement(X1),complement(join(X2,X1)))=complement(X1))).
cnf(c-0-73, plain, (join(X2,complement(composition(converse(X1),complement(composition(X1,X2)))))=
complement(composition(converse(X1),complement(composition(X1,X2)))))).
cnf(c-0-75, plain, (join(composition(converse(X2),complement(composition(X2,complement(X1)))),
complement(join(complement(X1),composition(converse(X2),complement(composition(X2,complement(X1)))))))=X1)).
cnf(c-0-66, plain, (converse(join(converse(X1),X2))=join(X1,converse(X2)))).
cnf(c-0-68, plain, (join(X1,converse(complement(converse(X1))))=converse(top))).
cnf(c-0-70, plain, (converse(top)=top)).
cnf(c-0-72, plain, (join(X1,converse(complement(converse(X1))))=top)).
cnf(c-0-74, plain, (complement(join(complement(X1),complement(converse(complement(converse(complement(X1)))))))=X1)).
cnf(c-0-76, plain, (complement(join(X1,complement(converse(complement(converse(X1))))))=complement(X1))).
cnf(c-0-79, plain, (join(complement(X1),converse(complement(converse(X1))))=converse(complement(converse(X1))))).
cnf(c-0-77, plain, (join(X2,complement(converse(complement(converse(X2)))))=X2)).
cnf(c-0-78, plain, (join(X1,converse(complement(converse(complement(X1)))))=X1)).
cnf(c-0-82, plain, (converse(complement(converse(X1)))=complement(X1))).
cnf(c-0-86, plain, (complement(converse(X1))=converse(complement(X1)))).
cnf(c-0-85, plain, (join(complement(converse(X1)),composition(X2,complement(converse(composition(X1,X2)))))=
complement(converse(X1)))).
cnf(c-0-105, plain, (join(complement(X1),composition(complement(composition(X1,X2)),converse(X2)))=complement(X1))).
cnf(c-0-94, negatedconjecture, (composition(top,converse(complement(esk20)))=converse(complement(esk20)))).
cnf(c-0-96, negatedconjecture, (composition(complement(esk20),top)=complement(esk20))).
cnf(c-0-97, negatedconjecture, (composition(top,converse(complement(esk10)))=converse(complement(esk10)))).
cnf(c-0-102, negatedconjecture, (composition(complement(esk10),top)=complement(esk10))).
cnf(c-0-98, plain, (converse(join(X1,composition(converse(X2),X3)))=join(converse(X1),composition(converse(X3),X2)))).
cnf(c-0-99, plain, (join(converse(composition(X2,X1)),composition(X3,converse(X2)))=
composition(join(converse(X1),X3),converse(X2)))).
cnf(c-0-103, plain, (join(composition(X1,X2),composition(X1,X3))=composition(X1,join(X2,X3)))).
cnf(c-0-107, plain, (join(X1,composition(X1,X2))=composition(X1,join(one,X2)))).
cnf(c-0-101, negatedconjecture, (join(complement(esk20),composition(X1,top))=
composition(join(complement(esk20),X1),top))).
cnf(c-0-106, negatedconjecture, (composition(join(complement(esk10),complement(esk20)),top)=
join(complement(esk10),complement(esk20)))).
```