Evolving Neural Networks for Natural Deduction Proofs

Sebastian Schulze

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2016

Abstract

This project aims to explore new approaches for exploiting the generalisation properties of neural networks in the general proof construction domain. Compared to previous work, the main differences lie in the choice of feature representation and underlying network architectures. Rather than constructing a set of hand-crafted features, the task of feature learning is incorporated in the training process. Additionally the implemented framework aims to utilise the knowledge embedded in existing interactive theorem proving tools as an alternative to constructing new corpora for machine learning applications. Because of the novelty of these approaches, a genetic algorithm capable of optimising hyper-parameters was proposed to train neural networks. Experiments assessing the benefits of this approach were conducted on the derivation of natural deduction proofs over intuitionistic logic statements.

Acknowledgements

I would like to express my gratitude to my supervisor Dr Jacques Fleuriot for his support and help in the design of the work described in this report. None of the described work would have been possible to carry out without his ongoing guidance and feedback throughout the course of this project.

Furthermore, the system implemented for carrying out experiments described in this report is loosely based on the NEAT implementation by Cesar Gomes and makes use of the Isabelle-light module written by Petros Papapanagiotou for the HOL-Light theorem prover. I would therefore like to the two developers for making their work openly available. In particular Petros deserves my thanks for revising and expanding his work to provide certain additional features on request.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Sebastian Schulze*)

Table of Contents

Chapter 1

Introduction

One of the oldest goals of artificial intelligence research is to imbue computers with the ability of reasoning about domains of interest.

In the pursuit of a solution to this problem, the field of automated reasoning has concerned itself with the questions of what it means for arguments to be valid and how to derive these automatically since the 1950s.

Both the validity of statements and the arguments reasoning about them have almost exclusively been regarded as synonymous with similar notions in mathematical logic. In particular, the notions of symbolic representations of statements and the construction of reasonable inference rules for their manipulation have been hugely influential.

It is therefore not surprising that much time and effort has been spent on the exploration of automated construction of formal proofs as a means to answer the second enquiry regarding the automatic derivation of such arguments. This gave rise to the notion of automated theorem proving and related fields which have consequently developed into what is probably the best explored sub field of automated reasoning. This development has further been supported by the manifold applications in mathematics itself, but more prominently in system verification and circuit design.

To this day, a range of automated theorem proving tools has been developed. However, only a minority of these is fully automated in the sense of users supplying a statement to be proven and being presented with a proof. In many cases this kind of reasoning is yet beyond the capabilities of the machines. In particular, it is often uncertain whether a machine could actually find a valid proof due to results such as the incompleteness theorem. Instead, a more collaborative approach is taken to avoid the need for an extensive search over possible proof steps and allow humans to intervene when trying to reason about unprovable or false conjectures. In the so called interactive theorem proving, the theorem prover provides a framework for the user to provide high-level guidance for the inference process. In this context the full automation of the proof procedure is regarded as less important and the machine's main task is to enable the user to complete

proofs as efficiently as possible by providing ‘bookkeeping‘ mechanisms, tracking the progress of different steps and formulating the final proof in a formally correct fashion. [1] [2]

This process was inspired by the fact, that humans make use of their intuition when constructing proofs by hand. Thereby, they are often able to develop an understanding outperforming the most complex hand-crafted search heuristics despite having encountered only a few example instances. This observation holds across many domains and the field of machine learning aims to imbue computers with the same ability to automatically extrapolate decision heuristics from example data. While existing approaches generally require a vast amounts of training data when compared to humans, astonishing advances in a variety of application fields have been made.

In this project, I aim to combine the two fields outlined above and examine whether or not machine learning approaches can be used to construct heuristics fulfilling the same role as humans in the context of interactive theorem proving. The particular machine learning model I am interested in imbuing with the ability to reason about simple symbolic proofs are neural networks. These were initially developed in the 1950s and have recently achieved human-like performance on a broad spectrum of tasks. One key property to these success is their ability to develop abstract feature descriptions of problem instances during the training process which lends them the needed flexibility to generalise from an incomplete set of basic examples [3]. It stands to reason, that they are able to learn heuristics for the process of proof construction which would allow them to guide an interactive theorem prover.

Moreover, neural networks are inspired by the human brain and have often been used as models of human cognition. In particular, they are often used to try and emulate human-like processes such as intuition or creativity which are also needed in the process of mathematical proof construction. Recent achievement seem to confirm the capability of neural networks to emulate such processes. The program AlphaGo [4] developed by Google is based on neural networks and was the first to overcome human professional Go players in a series of even matches. Because of the large space of possible games, Go is considered to be a game requiring not only logical reasoning about the arising positions, but also an intuition for moves are good or bad. Prior to this Google released ‘Deep Dream‘ [5] capable of creating surreal and more importantly novel images.

This report describes my approach to training neural networks constructing natural deduction proofs [6] in the domain of propositional logic [7]. In the next chapter, I provide a short survey of other approaches to imitating human creativity or intuition, before going into more detail regarding interactive theorem proving and the natural deduction calculus. I then proceed to give a brief description of typical ways to employ neural networks and how these found their application in this project in chapter 3. This part of the report is mostly based on the deliberation during the early stages of the project during which I was aiming to formulate a clear problem statement as well as potential evaluation criteria.

As such, it is mainly founded in a cursory exploration of various fields potentially relating neural networks and automated reasoning.

Chapter 4 describes the considerations made during the construction of the framework for combining neural networks implemented in python with the interactive theorem proving system HOL-light [8]. Chapter 5 contains the experiments I ran within this framework as well as their results. Further discussion of these together with potential extensions of this project can be found in the last chapter.

Chapter 2

Mathematical Creativity and Automated Theorem Proving

As described above, this project aims to train neural networks imitating human intuition for the purpose of providing heuristics for the construction of formal proofs.

Numerous attempts at defining the concept of human intuition have been made. While the Oxford dictionary describes it as 'the ability to understand something instinctively, without the need for conscious reasoning', the field of psychology establishes links to our physical senses and the unused parts of our brain and others in turn relate it to the existence of a spiritual aspect or soul. As manifold as these definitions are, none of them have been able to give a convincing explanation of how it works and it has consequently been notoriously difficult to construct even the simplest mathematical models.

Nonetheless, researchers in artificial intelligence have tried to imitate this ability computationally. Links can be drawn to a number of related fields such as computational creativity, artificial imagination and artificial intuition. These attempt to recreate the corresponding human abilities in various ways.

Approaches founded on rule-based systems in which (usually recursive) sets of rules give rise to a potentially unlimited number of new creations have been used as early as 1973 with the development of AARON by H. Cohen [9]. This program is capable of creating original artistic images based on previously provided abstract description of objects characterising their sub parts and the relationships between them. However, with the rise of connectionism [10] in the late 1980s, neural networks and genetic algorithms became attractive alternatives for such tasks [11]. Being used for applications such as music generation [12], they have since become a popular object of study in these areas because of their ability to generalise from training examples presented to them.

Modern developments in these areas have produced projects such as the Painting Fool [13] [14] and CoInvent [15] exploring the creative generation of art or abstract concept descriptions. Both of these are characterised by combining rule-based

approaches (in particular constraint satisfaction problems) providing high-level guidance within the target domain to machine learning and genetic algorithms producing randomness in the created artefacts. Unfortunately, many of these projects are targeting unstructured application domains such as the visual arts, music, poetry etc. in which it is hard to assess the success of a computer program due to the inherent subjectiveness of their experience [16].

While there are a number of projects aiming towards the creation of new conjectures [17] or concepts [18], to my knowledge no attempts have been made to apply these concepts in the domain of automated reasoning. While the connection to the rigour of mathematical reasoning may initially suggest little room for intuition in the construction of proofs, further consideration shows that this is not actually the case.

Firstly, there is a notion of elegance and beauty of proofs within the mathematical community referring to the aesthetic qualities associated with various theorems but more importantly their proofs. While this is a subjective and therefore elusive criterion, there is a consensus between mathematicians regarding its existence. Computer generated proofs are often seen as lacking this quality as they tend to rely on automatised procedures to exhaustively work through a large number of cases humans cannot verify by hand and are often characterised by laborious calculations.

Secondly, a connection to the actual proof construction process can be drawn. Many domains - even those considered to be simplistic and well-explored - give rise to problems incapable of being solved through exhaustive searches across an entire space of solutions or proof steps. One of the probably simplest examples is given by propositional logic. Proving or disproving the truth value of these statements has been proven to be Co-NP-complete [19] and it is therefore believed that only exponential-time algorithms exist for general proof tasks. Consequently, formal proofs in these and more complex settings rely on good heuristics when exploring possible steps towards proof construction. Particularly in complex proofs it is often unpredictable which course of action will lead to actual proof completion and humans have to rely on their intuition rather than exhaustively working through all available option.

Due to restricted computational resources and time limitations, I decided to restrict myself to the domain of intuitionistic logic [20] over propositional logic statements [7]. This provides an obvious assessment criterion in the form of the mathematical correctness of the suggested proof steps. While there already exist automated proof procedures for propositional logic such as resolution [21], these rely on exhaustive search procedures hardly resembling human reasoning. A more human-like framework is provided by the natural deduction calculus [6] described in the next subsection. Proofs therein transform the target statements by breaking up connectives or introducing them as needed via appropriate rules. Consequently, reasoning continues on the resulting (simplified) statements.

A second, softer criterion is given by the assessment of proofs regarding their elegance. This is partially incorporated by the choice of natural deduction calculus

which was developed with the intention of imitating human reasoning. However, beyond that the brevity of a proof (number of proof steps until proof completion) and the avoidance of steps reversing the effect of previous ones can provide softer measures thereof.

2.1 Natural Deduction Proof Calculus and propositional logic

The following section gives a brief overview of the propositional logic system [7] and the natural deduction calculus [6]. Even though the later is a system for general (logic) proofs, I will mainly stay within the context of propositional logic as this is the domain chosen for this project.

The propositional logic system is designed to reason about the truth value of well-formed statements. Thereby, a distinction between atomic clauses and more complex propositions is made. The former are indivisible statements with fixed truth values not containing any connectives usually represented through a set of boolean variables. Additionally, well-formed formulae can be composed via a set of connectives defining relations between the truth-values of the sub-clauses and their resulting combination.

Definition: A well-formed propositional logic formulae is either

- one of the two constants True ('T' or '$\top$') or False ('F' or '$\bot$')
- a variable representing an atomic clause
- the proposition '(A)' for some well-formed formulae A
- the negation $\neg A$ of some well-formed formulae A
- the combination A * B of some two well-formed formulas A and B via a connective $* \in \{\wedge, \vee, \longrightarrow, \leftrightarrow\}$

However, it has been shown that differing sets of connectives can be substituted in the definition given above without restricting the expressiveness of the resulting system. Hence, a number of additional operators such as the alternative denial (NAND), exclusive disjunction(XOR) and more have been defined - the definition above only lists the most commonly considered ones. In this project, I am considering the set containing only the connectives for disjunction, conjunction and implication. Other connectives can be defined indirectly e.g.

$$\neg A \quad \text{as} \quad (A \Rightarrow \bot) \quad \text{and}$$
$$A \Longleftrightarrow B \quad \text{as} \quad (A \Rightarrow B) \wedge (B \Rightarrow A)$$

The truth value of any well-formed formula is then given according to the definitions of the operators used and the truth values of the atomic clauses. A particular property of interest well-formed statements can possess is to be tautologically true regardless of the circumstances encoded in the set of atomic clauses.

Definition: A well-formed formula ϕ is a tautology iff for all possible assignments of truth values assigned to atomic clauses part of ϕ its truth value is true.

The class of such universal statements provable within a logical system is governed by a set of applicable inference rules transforming logical statements and a set of axioms regarded as absolute truths. A particular such system is given by the natural deduction calculus.

The natural deduction calculus was first invented in the early 1930s with the aim of constructing a system to specify formal arguments or deductions in a way closely relating to the 'natural' way of human thinking. This was a reaction to previous attempts at developing such a system for the description of mathematical proofs which followed more axiomatic approaches. These required any statement to be directly derived from a set of core axioms via inference rules. However, this does not mirror the ways in which less formal, yet correct proofs are formulated in the mathematical community and in particular does not allow for indirect proofs or proofs by contradiction temporarily introducing premises of unknown truth value.

Natural deduction arguments overcome this shortcoming by arguing from a set of premises serving as the basis to be manipulated via inference rules. In contrast to axiomatic systems which are characterised by very few powerful inference rules, natural deduction makes use of a multitude of simpler inference rules for the composition and decomposition of statements. Depending on whether these rules compose two sub statements into a single one by introducing a connective or subdivide a single statement by removing a connective, these rules are referred to as introduction and elimination rules. Depending on the particular setting there exists at least one introduction and one elimination rule for each connective.

In the context of propositional logic, this requires one rule for the introduction of each connective except for disjunction which requires two separate rules for first and second disjunct. As depicted below, the premises are listed above the conclusions separated by a bar labelled with the name of the rule. The three dots in the IMP_I rule represent a proof of B from the set of premises [A].

$$\frac{\mathrm{A}\quad \mathrm{B}}{A \wedge B}\, CONJ_I \qquad \frac{\mathrm{A}}{A \vee B}\, DISJ_{I1} \qquad \frac{\mathrm{A}}{B \vee A}\, DISJ_{I2} \qquad \begin{array}{c}[A]\\ \vdots \\ \mathrm{B}\end{array}\; IMP_I \quad A \Rightarrow B$$

Figure 2.1: Introduction rules for propositional logic

Furthermore there is a matching inference rule for the elimination of each connection with the exception of conjunction which requires a separate rule per conjunct:

$$\frac{A \wedge B}{A}\, CONJ_{E1} \qquad \frac{A \wedge B}{B}\, CONJ_{E2} \qquad \frac{[A] \quad [B]}{\begin{array}{cc} \vdots & \vdots \\ [A \vee B] & \mathrm{C} \quad \mathrm{C} \end{array}\; C}\, DISJ_E \qquad \frac{\mathrm{A} \qquad A \Rightarrow B}{B}\, IMP_E$$

Figure 2.2: Elimination rules for propositional logic

The inference rules given so far constitute a sound and complete proof system for intuitionistic logic [20]. This means that only statements accepted as true within that logical system can actually be proven and at the same time all such statements can be proven. In particular, intuitionistic logic rejects the law of the excluded middle ($A \vee \neg A$). To extend the system to more powerful logic systems such as classic logic, further inference rules need to be added.

Because of the way the calculus is structured, arising proofs generally aim to break up connectives in the premises as needed via elimination rules and reassemble the resulting statements to derive the desired conclusion using introduction rules. There are several ways to depict these proofs. The most common one employs a graphical tree with nodes representing intermediate statements and connections the composition and decomposition via inference rules. The root of the tree contains the desired conclusion while leaves are labelled with the (discharged) premises. Both the premises and applied introduction rules are numbered to show when premises were introduced.

$$\dfrac{\dfrac{\dfrac{\dfrac{(A \Rightarrow (B \wedge C))^1 \qquad A^2}{B \wedge C}\, IMP_E}{B}\, CONJ_{E1}}{(A \Rightarrow B)}\, IMP_I^2 \qquad \dfrac{\dfrac{\dfrac{(A \Rightarrow (B \wedge C))^1 \qquad A^3}{B \wedge C}\, IMP_E}{C}\, CONJ_{E2}}{(A \Rightarrow C)}\, IMP_I^3}{((A \Rightarrow B) \wedge (A \Rightarrow C))}\, CONJ_I}{(A \Rightarrow (B \wedge C)) \Rightarrow ((A \Rightarrow B) \wedge (A \Rightarrow C))}\, IMP_I^1$$

Figure 2.3: Example of a natural deduction tree representation of a proof in propositional logic

This style of representation clearly reflects the dependencies between individual formulas and and emphasises their linkage via inference rules. However, it is relatively convoluted, and potentially involves duplication of similar or even identical proof branches and premises. When working by hand, it is therefore usual to resort to more linear representation merely reflecting the core steps in the different proof branches. On a computer, however, it is unproblematic to transfer this 'bookkeeping' of what premises are 'alive' in which branches and what the current sub-goals are to the machine. The IsabelleLight extension [22] of the HOL-Light theorem prover [8] provides such possibilities. At each stage, the user is presented with a set of premises for a particular sub-goal resulting from previous inference steps. When a sub-goal is successfully proven, the theorem prover automatically

advances to the next remaining sub-goal or reports the completion of the proof. The following section discusses the field of interactive theorem proving as well as the syntax and usage of the developed tools in more detail and describes the environment employed in this particular project.

2.2 Interactive Theorem Proving and HOL-Light

As outlined previously, the field of interactive theorem proving evolved from the field of automated theorem proving in an attempt to overcome certain weaknesses of fully automated theorem provers. Automated theorem proving itself is motivated by fact that proofs published in the mathematical community are not only growing in numbers but also in complexity, making it harder and harder to verify or guarantee their correctness. With the arrival of computer supported proofs in 1976 (proof of the 4-colour theorem) this trend has only been strengthened, calling for tools capable of constructing and verifying proofs in arbitrary mathematical domains. Additionally, it is possible to describe many practical, complex applications in mathematical terms allowing to reason about desirable properties - for instance, both AMD and Intel verify chip designs using automated theorem proving procedures [23] [24].

Where automated theorem proving focusses on enabling computer systems to discover mathematical truths completely autonomously without human intervention at any point (other than supplying the initial problem), interactive theorem proving aims to enable the user of such a system to construct formal proofs in collaboration with the machine. Thereby, it is the machines main task to track the low-level inference steps and to construct the final formal proof whilst guaranteeing its correctness. Automation is merely provided in the form of subroutines simplifying intermediate problem statements and solving eventually arising decidable sub problems. This leaves the user free to creatively construct the main proof on an appropriate level of abstraction far from the axiomatic foundation of the respective theories. Furthermore, he can access and reuse previously proven results contained by the tool whilst being certain of their mathematical rigour. [23]

During proof construction, the user interacts with the system via a (usually interactively interpreted) programming like interface. Most available tools use their own proof languages constituting a mixture programming language paradigms and mathematical notation and supply corresponding editors. Depending on the particular tool different styles of proofs are specifiable:

- Procedural languages emphasize proof methods (i.e. application of theorems, rewriting, proof by induction etc.), thereby losing precise descriptions of the intermediate proof states which are implementation dependent.
- Declarative languages on the other hand emphasise these states but are slightly less precise about the logical justification of the gap between one state and the next one which is left to the tool itself.

In either paradigm a proof of a theorem is constructed by iteratively modifying a current goal or sub-goal through a sequence of linking steps which can be roughly divided into two categories. On the one hand they can invoke automated procedures provided by the theorem prover which then try to discharge the current sub-goal by (exhaustively) searching for a solution. These procedures are generally derived from automated theorem provers and remain mostly domain specific. On the other, they can rewrite the current goal state in terms of a lemma provided, making it possible to apply general (verified) procedures such as complete induction or case distinctions but also previous results defined by the user.

Beyond this 'proving environment', the theorem proving tools supply large libraries providing the necessary background definitions for specific domains (e.g. natural number theory, linear algebra) which can be loaded on demand. Each tool derives its mathematical justification from a specific core set of axioms from which all other theorems are derived e.g. set theory for MIZAR and type theory for Isabelle and HOL [23].

2.2.1 HOL-Light

This project is based on the HOL Light theorem prover which is light-weight offspring from the HOL family of theorem provers. These in turn are derived from the LCF theorem prover developed in Edinburgh [25] [26]. All of these are written in the ML language or dialects thereof. This general purpose programming language was designed to accommodate the development of abstract proof tactics representing the inference steps described above. The language is an impure functional programming language allowing for side-effects and defines abstract objects types for statements (terms), theorems and proofs. Tactics and inference rules in turn are defined as functions operating over these types.

The HOL Light theorem prover itself uses the OCaml language and supports mostly procedural style proofs. These can either be stored in script files and be (verified and) loaded at run time or developed in the interpreter. The interpreter itself only provides a low-level shell interface for users in which they can call the provided inference rules or formulate valid OCaml statements to interact with the theorem prover. At run time the prover tracks which theorems have been proven rather than whether or not specific theorems are true or false. More specifically it will accept unprovable conjectures as potential inputs under the assumption that these will never be verified within the axiomatic reference frame provided by the theorem prover itself.

Beyond that, the tool maintains a goal stack for the most recent (unfinished) proof. This consists of a number of sub-goals yet to be proven or further broken down via the inference mechanisms provided by HOL-Light or previously defined by the user. This goal stack remains unchanged if any invalid inference steps are proposed by the user in which case an error response is given. If a valid inference rule is suggested, the goal stack is changed accordingly and potentially completed

sub-goals are discharged before the topmost remaining sub-goal is displayed or if none remain the proofs completion is reported.

Of particular interest for this project was easy programmability of the HOL-Light system. It is distributed as open source software with freely available source code. OCaml scripts can be loaded into the theorem prover at any point and have full access to the internal representations of proven theorems, the goal stack and other objects. Since all interaction with the tool is intended to happen through the call of appropriate tactics (which are function objects in the OCaml language) or OCaml expressions, it is relatively straight forward to interface with the tool once it is running.

Despite its easy programmability and low-level interface, the tool is intended for use by a human operator. As such it has a relatively large response time to commands in terms of machine time scales (several milliseconds). Furthermore, it provides a large set of theories users can access at runtime in form of scripts loaded during initialisation. Unfortunately, this leads to long response times (2-5mins) at start up. Both of these shortcomings are severe problems for machine learning applications requiring large amounts of interactions to learn from. This issue was eventually overcome by running multiple instances of the theorem prover in parallel and only load a subset of the available theories during the start-up.

2.2.2 Isabelle Light

As mentioned previously, this project heavily builds on the IsabelleLight library [22]. This module provides a procedural environment for the formulation of natural deduction proofs aiming to mimic the natural deduction proof commands available in the Isabelle theorem prover. In particular, it defines theorems containing tactics and inference rules for the manipulation of logical statement according to the introduction and elimination rules specified above. The proof process is initiated by supplying a propositional (or first-order) logic statement. The proof steps needed to be specified by the user consist of two parts - a tactic and an inference rule. The inference rules available are the standard natural deduction inference rules for propositional logic introduced above. These can be combined with the four tactics 'rule', 'erule','drule' and 'frule'. The tactics guide the unification of the sub-formulas with the current goal state. While 'rule' only matches the current goal's conclusion with the conclusion of the inference rule and introduces the inference rule's premises as sub-goals to be established, the other tactics establish links between the current premises and these sub-goals. A final tactic called assumption not requiring an additional inference rule as an argument is used to match the conclusion with an assumption at the end of a proof or sub proof. The proof shown in the previous section is conducted via the following sequence of commands:

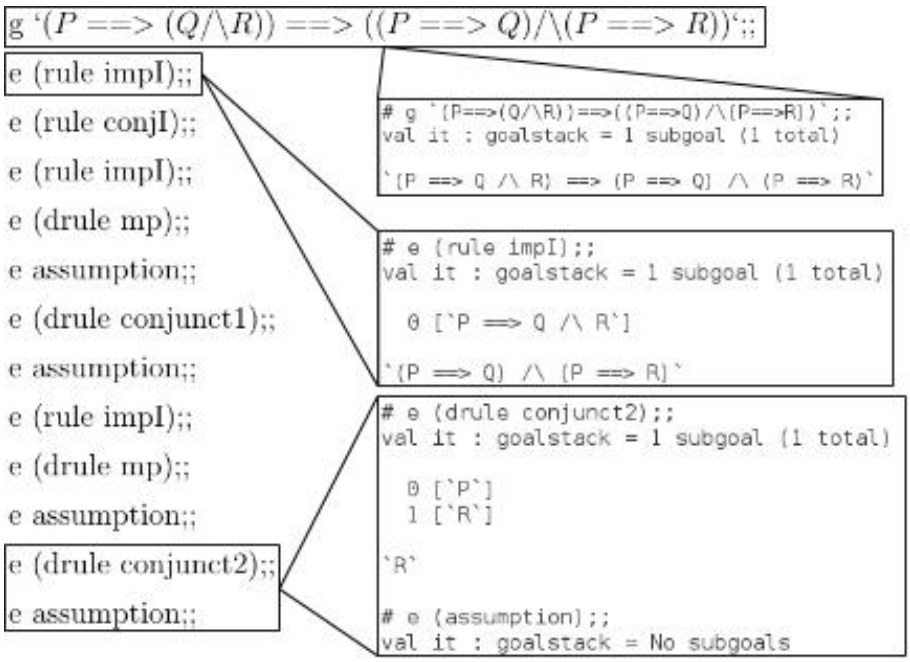

Figure 2.4: IsabelleLight proof and response to selected proof steps

Proof of concept

At this point, I would like to point out the existence of general proof procedures for the problem of deciding whether a propositional logic statement is satisfiable, contradiction or tautology. An example is given by the resolution algorithm proposed in the 1960s [21]. However, rather than seeing this project as an attempt at improving over these methods, I would like to encourage the reader to regard it is an exploration of concept. The focus lies on the question of finding ways to train neural networks to conduct more general theorem proving tasks inspired by human theorem proving. Neural networks already achieve (super-)human performance in a range of domains such as classification problems in vision and speech recognition but also in more abstract tasks such as competing in arcade games [27] and most recently Go [4]. In my opinion the problem of learning heuristics for the choice of optimal inference steps resembles game settings much more closely than classification tasks, since intermediate steps within a proof relate to each other and often there is more than a single correct solution. I therefore propose to view this problem as a game setting in which the neural network is presented with a problem statement and a range of inference rules from which it has to choose. Upon making a decision, it receives feedback from the interactive theorem prover and evolves in response. In order to iteratively improve its performance, the neural network should need to learn the logical connections between the problem statements and inference rules effectively learning the abstract knowledge previously embedded in the theorem prover. It thereby makes sense, to consider simplistic settings prior to more complicated ones in order to facilitate learning. The next section discusses this learning process and the general capabilities of neural networks as well as their connection to this project's setting in more detail.

Chapter 3

Neural networks and further problem analysis

The term neural network refers to a class of graphical models originally inspired by biological neural networks such as the human brain [28]. In the last decade neural networks have been receiving an increasing amount of attention and once again risen to provide state-of-the-art solutions for many machine learning tasks under the re-brand 'Deep learning'. In particular, they have been used to solve a wide range of classification tasks in vision domains (handwriting and object recognition), speech recognition and other applications [29]. Moreover, they have been used as policy networks controlling more complex, non-stationary problems including game settings such as arcade games and Go. [27] [4].

3.1 Problem specification

As presented in previous chapters the interactive theorem proving tool - or more specifically IsabelleLight - supplies a problem statement and in turn expects instructions consisting of tactic and inference rule to be applied next as its input. Constructed neural networks should therefore map from the space of possible goal states to the space of tactic-term pairs.

This description classifies the problem as an instance of supervised learning - a general class of problems concerned with the approximation of a target function from a set of labelled examples. The following sections give an overview regarding neural networks in general and approaches to their training in more detail. Furthermore, the approaches chosen for experiments conducted in this project are motivated from past research in different areas.

3.2 Neural network specification

At their core neural networks imitate the functionality of neurons in the human brain. This is done by considering systems of units connected via weighted links along which 'messages' are passed. A neural network is generally defined in terms of

1. The activation function processing incoming messages at individual neurons
2. The graph structure of the network i.e. number of units, link patterns and weights
3. A learning algorithm adjusting the link weights of the network in response to observed training data

3.2.1 Activation functions

Each unit or neuron in a neural network receives messages from its incoming connections which are accumulated into a weighted sum. The unit then applies an activation function to determine its own output to be passed on to other units. Without this function application, every neural network could be described as a single linear transformation and would effectively be no more powerful than a linear model (e.g. a linear regression).

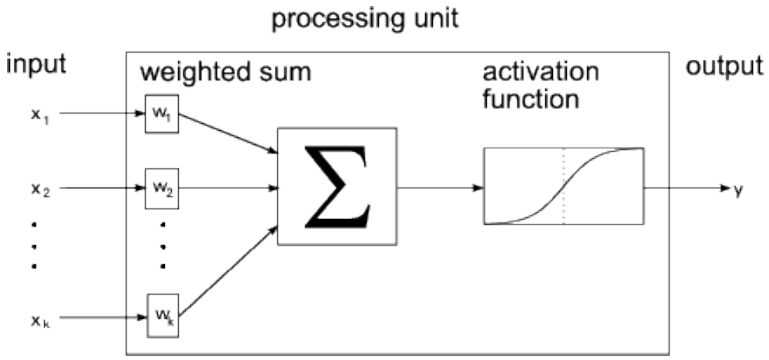

Figure 3.1: Neural network unit schematic representation,
Source: Springer "Industrial Image Processing"

In biologically inspired models, the activation function often chosen is a step function yielding no response before a particular threshold is reached. Machine learning applications typically chose differentiable functions with similar properties, such as the sigmoid or tanh function. However, other functions have been successfully deployed and no activations function appears to have a general advantage over others, making the choice between them context dependent. [30]

3.2.2 Network structures

Neural networks accept input arguments from a specific feature space and output an estimate of the matching function value. In classification tasks this is done by designating a number of input units whose activation levels correspond to particular features and a set of output units whose activation is interpreted as features of the function value, once activations have been propagated through the remaining, 'hidden' units.

Depending on the particular task, various connective patterns may be superimposed on a neural network. The most common approach are the so-called feed-forward networks. Here hidden units are arranged in a hierarchy of layers between the input and output units. Connections only lead from one layer to the next propagating the activations from input to output layer. Individual layers are typically associated with varying feature representations. Connections between layers therefore act as transformations from one feature space to the next. [28]

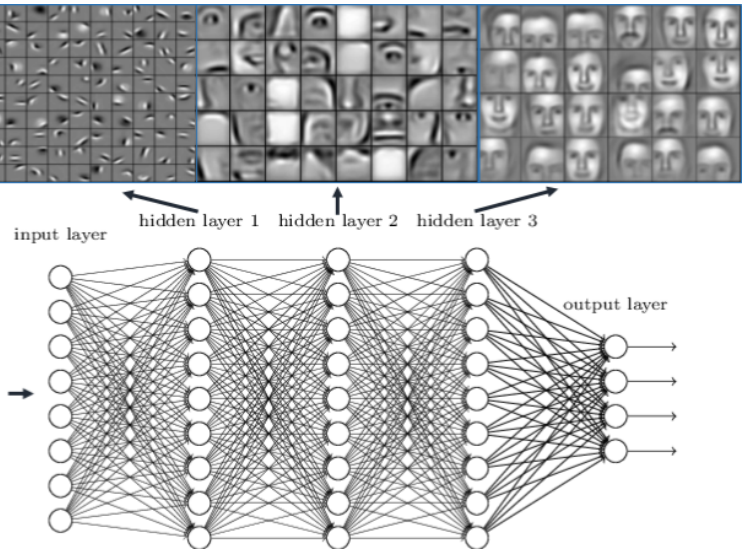

Figure 3.2: Feed-forward architecture and feature spaces
Source: http://www.rsipvision.com/exploring-deep-learning/

If connections feeding back into previous feature representations (layers), are part of a network, it is said to be recurrent. While feed-forward networks can be simulated by a single propagation of activation updates from input to output units, recurrent networks require the addition of a time component. They are simulated in a stepwise fashion - at each stage all units in the network propagate their content through outgoing connections and update their internal state via incoming ones simultaneously. Typically, this class of network is used to analyse a sequence of data points or symbols and at each time step in the simulation the network is presented with a new data point at its input layer. Because earlier results are fed back into lower representation, they provide a context for the processing of later parts of the sequence. The layers in these models therefore not only represent more and more abstract feature representations but instead

provide summaries for of a sequence up to the current point. [31] [32]

3.2.3 Training paradigms

A variety of different training mechanisms have been proposed to adjust free parameters of neural network architectures for differing types of tasks.

In supervised learning problems where networks are to approximate a function from a set of labelled training examples, training algorithms are almost exclusively based on gradient descent based methods. These assume a fixed network structure provided by the user and only optimise the connection weights thereof. Thereby, they treat the training process as a convex optimisation problem and aim to minimise the error of the network over the training set based on a loss function comparing the networks predictions to the actual labels. This generally requires the activation functions of individual neurons to be differentiable. A number of extensions to optimise the convergence and generalisation properties of these algorithms have been developed. [28]

A second class of training algorithms available is inspired by genetic algorithms. These are not necessarily bound to a particular network structure other than fixed numbers of input and output units dictated by the task description and an overall connective patterns (e.g. feed-forward network, recurrent network, convolutional network). They semi-randomly explore the space of possible network configurations by evaluating a number of possible structures and weight assignments on the training data with regards to a fitness function. Better-performing networks are combined according to a crossover operator and minor random perturbations or mutations are introduced to create a new set of networks to be evaluated in the next generation. This process is repeated until a network meeting particular performance thresholds is constructed or the maximum number of generations is reached. Just as other genetic algorithms, this approach is inspired by natural evolution and assumes the existence ‘useful‘ traits or ‘building blocks‘ enabling higher network performance. Over the course of training such traits would increase a networks performance, making them more likely to be propagated into the next generation and eventually form an optimal solution. A particular such algorithm called NEAT is described later in the report. [33]

3.3 Related work

3.3.1 Machine learning and automated theorem proving

Machine learning paradigms are increasingly finding applications in automated theorem proving. The main interest here lies in learning good heuristics for exhaustive search algorithms over a number of available proof steps. However, in that particular context neural networks are often dismissed in favour of other

machine learning models - in particular support vector machines. They have therefore only been used in a selected number of projects.

In the 1990s Mareco and Paccanaro were among the first to explore how neural networks could be used to improve automated theorem provers. However, they focused on matching different terms to each other for rewriting-based theorem proving and applied their work on the domain of group theory. While they did derive different term representations suitable for neural networks, these relied on hand-crafted, numerical features. [34]

A form of recurrent neural networks was employed by Goller et al. [35] to estimate the value of available inference steps throughout proof construction for the theorem prover SETHEO. While promising, the work done there was only evaluated on a simplistic domain. Later application to more realistic problems by Blanchard et al. [36] showed, that while yielding improvements, the networks merely memorised training examples instead of generalising to new patterns.

Lastly, Suttner and Ertel employed feed-forward architectures as search heuristics for the SETHEO prover. Their work focused on the area of model elimination and the constructed heuristics led to significant improvements of the search process. Once again, they relied on hand-crafted features as inputs for the created models. However, they commented on the fact that these are domain specific and the generalisation of such an approach is not entirely certain. [37]

Hence, it can be said that the few existing projects have shown the potential of neural networks as search heuristics supporting autonomous or interactive theorem proving tools. Existing approaches focus on the creation of hand-crafted fixed size features and employ neural networks as function approximators of the targeted heuristics. However, this yields models tailored to particular domains which are uncertain to generalise well.

3.3.2 Neural network research

Looking at the wider neural network literature, it becomes apparent that the use of hand-crafted features is less and less prevalent since it has become viable to train deep models. Instead, neural networks are presented with low-level features and meant to discover suitable feature representations in their intermediate layers during the training process by themselves. [38]

Following this train of thought the field of natural language processing becomes of particular interest to this project. This field aims to use neural networks to extract semantic information from character or word sequences and use the obtained information in various applications such as speech-to-text or translation tools. In particular, this can be seen as a building block for the task of proof construction as it is necessary to parse descriptions of the current goal state as provided by the theorem proving tool before choosing the appropriate inference steps.

In this field recurrent neural networks are the dominant architecture and have been shown to be a powerful alternative to other modelling paradigms such as Markov models or n-gram models both on a word and character level. Their main advantage over other neural network topologies is the ability to operate over input sequences of arbitrary length, as long as individual features such as individual letters or words are represented by fixed size representations (e.g. binary vectors of an underlying vocabulary). As described previously, this is done by maintaining a feature summary over the observed points in the sequence to be used for the particular task at hand.

A second subarea of neural network research is neuroevolution. This field was inspired by evolutionary algorithms and overcomes the restriction to fixed network topologies inherent in typical-used gradient-descend based training algorithms. These can be applied to construct both networks of fixed and variable topology. As such, they not only optimise a networks weight parameters but also it's hyper-parameters. The algorithm of particular interest to this project is called NEAT (NeuroEvolution of augmenting topologies). This algorithm was successfully employed to solve a range of reinforcement learning benchmarking problems. Thereby it outperformed other neuroevolutionary methods with regards to training time and quality of the evolved networks. This included tasks such as the approximation of the XOR-function, the pole-balancing problem and controlling a car in a racing simulator. Moreover, it has successfully been used in more complex settings such as the development of networks navigating the game Super Mario, controlling agents in a real-time strategy game called Nero and controlling a roving eye architecture for playing against the program GnuGo on small board sizes [33]. In the later application it has shown exciting generalisation properties. Networks trained for to play on smaller board sizes (5x5) were used to initialise the search for well-performing networks on 7x7 boards. It was shown that this allowed for significant speed up of the training process. Moreover, this application showed NEATs ability to successfully develop recurrent connections for an abstract task.

3.4 NeuroEvolution of augmenting topologies

The NEAT algorithm [33] is a genetic algorithm evolving both structure and weight settings of neural networks. As mentioned above it has been successfully applied to a number of challenging problems. This makes the algorithm an appealing choice for this project as it indicates, that it is able to cope with abstract problems of high complexity. Moreover, it is possible to evolve recurrent structures as they are required in this project.

NEAT follows the general approach of genetic algorithms in that it maintains a population or gene pool of network descriptions or chromosomes. Each network is represented by a collection of genes describing individual nodes and connections (although connection genes can be disabled).

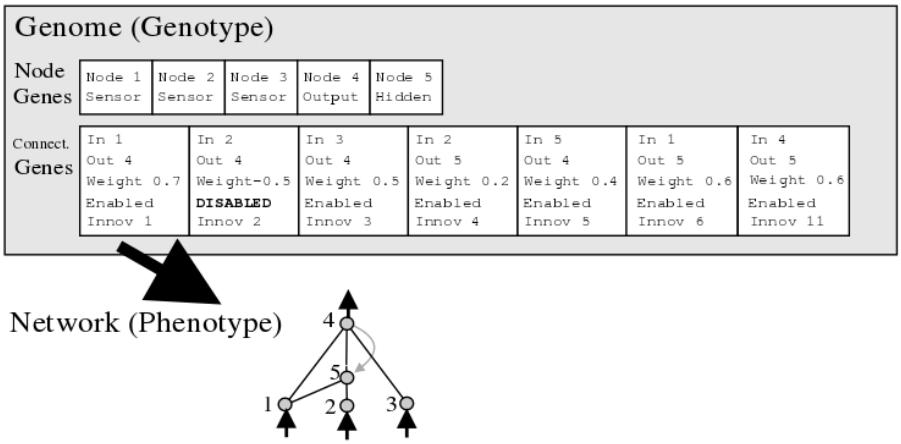

Figure 3.3: Network description as given in [33]

In each simulated generation, the fitness of each network is assessed and a percentage of low-performing networks are pruned from the gene-pool. The remaining networks are recombined with each other using a cross-over operator and small random perturbations or mutations are introduced. This process is repeated until a suitable solution has been found or a time limit is reached. The crossover operator employed by NEAT constructs the combination of two chromosomes as the collection of all their genes - shared genes are taken from the chromosome with a higher fitness or at random.

Three mutation operators are available. The first randomly alters weights and biases of existing links and nodes by a certain amount. The second introduces a random connection by choosing two unconnected nodes at random and constructing the matching gene. The last mutation adds a node by choosing a connection at random, disabling it and adding a new hidden united connected to the start (weight 1.0) and end (old weight) of the initial connection. All three mutations are performed with probabilities set by the user.

Beyond this, the functionality of NEAT is based on three core principles.

Firstly, the algorithm starts with the simplest neural networks containing no hidden units and no or few connections. Starting from this population, the algorithm incrementally creates more complex structures by adding individual units or connections. Thus, simpler solutions are proposed first and the algorithm systematically expands the networks based on the feedback obtained through the fitness function. In particular, this produces solutions which are less prone to have overfit to a particular training set.

Secondly, each of the genes representing a connection is tracked by a historic marker assigned upon its first appearance in the gene pool. This allows to compare and combine different network structures by interpreting genes with the same historic origin to represent the same structural change.

Lastly, networks are clustered into species via a similarity operator and fitnesses are only compared between networks of the same species or amongst entire species

as a whole. This allows newly introduced genes to remain in the gene pool without immediately needing to improve the networks fitness. However, species are penalised and finally removed if they show no improvement in their average fitness over a set period of time. The similarity operator is based on the historical markings indicating which inventions are shared between two networks as well as the average difference in weights between connections in common between two networks. Once again it is the users choice to weight the impact of weight differences against structural differences when determining the similarity of different networks.

There are several extension to the basic NEAT algorithm, however, these aim to extend the evolution process to be simulated in real time problems and to make use of symmetries in the in input space. They are therefore irrelevant for type of task considered in this report.

Chapter 4

System architecture

4.1 Design Choices

4.1.1 Learning paradigm

As mentioned above, supervised learning tasks are characterised by their need for labelled training sets. Unfortunately, such a set of training data is not available in our problem for two reasons. Firstly, few natural deduction proofs in the target domain have been recorded - theorem proving tools only describe automated procedures for solving these, but do not actually contain specific instances. The few cases actually recorded are scattered over libraries contained by a variety of different tools as they serve as examples illustrating particular aspects of the tools in question. Secondly, it is often hard to specify what proof step should be considered to be correct for a particular problem instance. Often the user is faced with a choice of several different connectives to be broken up and the order in which this is done does not influence the future development, hence, it would be difficult to assign labels to such instances.

Any hand-curated corpus could only contain a comparatively small number of proofs, since each proof step would have to be manually specified. This is problematic as it facilitates the over-fitting of networks during the training stage. Rather than generalising from the training data, networks would learn to distinguish between individual training instances, provided the network structure is flexible enough to do so. It therefore seems appropriate to treat this task as an online learning problem in which networks are altered in response to repeated interaction with the theorem prover instead of restricting ourselves to a limited corpus.

A second issue arises from the choice between different network sizes and connective patterns for the targeted structures. Since neural networks have not been employed in this fashion in automated reasoning before, there are no previous results we could draw on to obtain estimates for this task. Gradient descent based methods would require a fixed network structure and would have to be explored

via a costly search over different network sizes.

Genetic algorithms on the other hand only assume a fixed declaration of input and output units and are capable of placing hidden units and connections within the neural network on its own. Thereby, they are able to find their own -often surprising- solutions to relatively vague problem descriptions in the form of fitness functions. Within these fitness functions it is sufficient to describe criteria for a good solution rather than specifying the targeted solution it. Thus, the users choices reduce to a suitable description of the task neural networks are to fulfil and criteria for what is considered to be a good solution. The former is done by describing the two feature spaces in which inputs and outputs of the neural networks reside. The later can be evaluated on the basis of a set of potential problems and the amount of proofs a candidate neural network completes.

Thus, I decided to employ the NEAT algorithm as a means to train networks throughout the project. This also suggested to employ the same activation function that was used in the experiments described in [33]. The two remaining specifications to be made were the feature representations of outputs and inputs and the overall architecture to be built which largely dependent on the features presented to the network. Furthermore, suitable fitness functions had to be defined, however, these were evaluated experimentally.

4.1.2 Feature spaces and network architecture

The output space can easily be described as the combination of two probability distributions - one across the available tactics and the other one across available rules. The options receiving the highest probability mass in the two sets are then interpreted as tactic and inference rule chosen by the network. Note that this representation is only suitable, since there is a fixed number of tactics and rules available at all times.

Problem statements on the other hand are more difficult to describe. They can vastly vary in length and content, and small difference may drastically change the semantic meaning of a statement. Fixed size feature descriptions serving as the basis for feed-forward networks have to be carefully hand-crafted in order to capture this kind of variability. Otherwise, there likely exist differing statements being transformed into the same feature representation and therefore (wrongly) treated the same by the network. Furthermore, these descriptions have to be feature engineered for the particular task and are therefore innately biased by the design choices made.

An alternative is given by regarding propositional logic statements describing the goal states as ‘sentences‘ of logical symbols. These can be interpreted by recurrent structures as described before, allowing for parallels to the modelling of natural languages to be drawn. These models allow for inputs of arbitrary length and individual symbols can be represented by fixed size binary vectors.

Additionally, recurrent structures could interpret any enumerable data structure,

such as lists or trees. In particular, it is possible to enumerate the parse trees of the logical formulae. Hereby, a formulae would be encoded by a tree containing individual variables in the leaves and connectives in the internal nodes. The tree structure is dictated by different connectives merging the respective variables and sub formulae in the order dictated by their precedence properties. In such a representation a sequence corresponding to an enumeration of the tree's nodes according to a depth-first or breadth-first traversal would be passed to the network. Special symbols denoting the transition from a premise to the next or to the final conclusion would be inserted as needed.

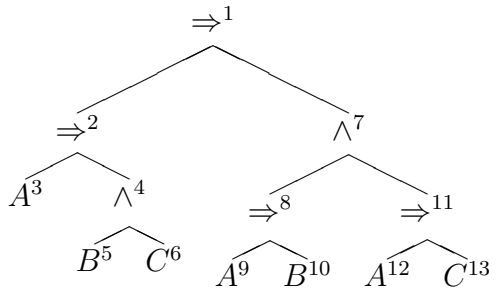

Figure 4.1: Logical parse tree for $(A \Rightarrow (B \wedge C)) \Rightarrow ((A \Rightarrow B) \wedge (A \Rightarrow C))$ and enumeration according to depth-first search

However, we decided to employ the linear representation containing sequences of symbols. While more simplistic, this aligns with many deep learning approaches which tend to pass low-level unprocessed data into the input layer. As neural networks are able to learn new feature spaces on their own, this avoids a potential loss of information or bias inherent in humanly engineered representations. Additionally, we aimed to keep the burden of parsing the logical statements on the neural networks. When humans perform natural deduction proofs they tend to first remove the highest level connectives in the conclusion before proceeding to the corresponding manipulations of the premises. Providing the tree-structure above would already include the parsing of the symbol sequence provided by the theorem prover and expose the main connective to the neural network as the first symbol in a sequence. This potentially means that the network merely resorts to exhaustively breaking up the top connectives instead of constructing a proof in a more goal-oriented fashion.

4.2 Implementation

As outlined in previous sections, the constructed system has two main components - the interactive theorem prover HOL-Light written in OCaml and a python implementation of the NEAT algorithm. Moreover, there are several modules providing simplistic communication between these two as well as several convenience functions for constructing sets of problems to be considered and evaluating the produced networks. The implementation of the NEAT algorithm is loosely based

on an implementation by Cesar Gomes. This was initially intended to verify the experimental results presented in by K. Stanley in [33]. However, profound changes were made including the revision of the provided neural network implementation, selection mechanisms during the crossover process, construction of the initial gene pool and more.

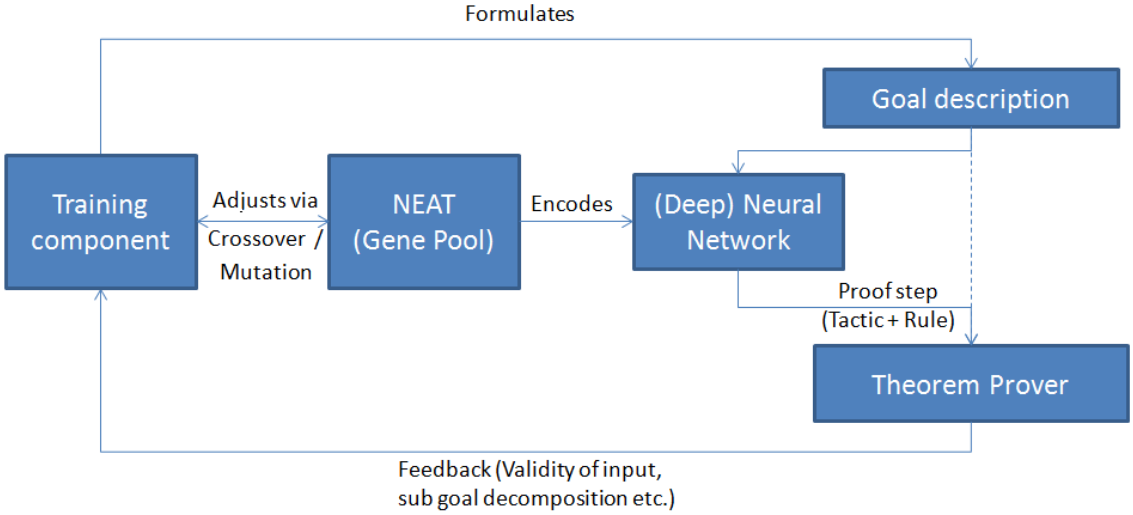

Figure 4.2: Overall system functionality

Within the framework, experiments can be executed via a single python script specifying a particular set of experimental parameters.

4.2.1 Python component

During experimental trials the provided script initialises a population object based on various parameters. The modifiable settings to be provided within said script are:

a) the fitness function of the NEAT algorithm,

b) a function generating the set of problems neural networks are evaluated on,

c) the encoding of the networks output choices and

d) various parameters of the NEAT algorithm

The script then calls a method evolving the networks for a set amount of generation or until a cut-off performance is reached. In each generation, this causes the population object to manipulate representations of individual species, chromosomes and genomes after having evaluated the fitness function for each neural network in the gene pool on the set of generated examples and discarded a percentage of the gene pool.

Since the fitness function is evaluated based on a set of training problems, a method constructing such a set is expected to be provided. However, the framework provides pre-defined functions for the generation of problem statements.

These can be based on propositional logic statements in which case only variable names are exchanged. However, they can also be artificially aim for the construction of neural networks capable of producing certain sequences of inference rules. In that case the target 'solution' sequence is to be provided and a matching propositional statement is designed.

In the formulation of the fitness function users can access the full simulation history of inference rules chosen by a neural network and the resulting intermediate states to assess the networks capability of constructing formal proofs. For each of these methods providing counts of the symbols, connectives and variables can be accessed.

The framework automatically translates between symbolic problem statements accepted by the theorem prover and binary vector representations suitable for neural networks. Each symbol is encoded in a binary representation and a limited amount of 16 variable names has binary code representations as well. This results in input vectors of 14 binary digits encoding the current symbol. Constructed neural networks have 13 output units. The first five of these are interpreted as a probability distribution over the available tactics and the tactic receiving the highest probability mass is considered the networks choice. Similarly, the remaining 8 units are interpreted as a probability distribution over the available inference rules.

Moreover, the system implements the described pre-evolution process by allowing the specification of iteratively more complex problem sets as well as the modifications of the NEAT parameters throughout a trial. The gene pool of networks trained to solve prior (simpler) problems is then reused as the initial starting point for a new evolution process based on the new training and parameter sets.

The numpy package is extensively used to speed up the choice of random elements of the evolution process as well as the simulation of recurrent neural networks. However, no specialist neural network libraries such as Theano or Caffe are employed - these generally require GPU support which was not available. Furthermore, the constructed networks are small and have a quickly changing topology making the use of such libraries ineffective.

A summary of the population statistics as well as an extensive list of the best performing networks is stored in log files and can be transformed into graph representations via provided methods.

4.2.2 Communication with IsabelleLight

To gather feedback regarding a particular networks choices from the theorem prover it is necessary to transmit the statement to be proven and the suggested inference steps to an IsabelleLight process and receive the resulting intermediate goal states.

For this a simple asynchronous socket connection was chosen. A process initialis-

ing the HOL-Light prover and loading the IsabelleLight module is run and opens the server side of a socket connection. Early on I found that the HOL-light theorem prover has a relatively large start-up times (several minutes). To overcome this issue, I initially adopted the recommended practice of manually checkpointing [39] the state of the HOL-Light process. This allows it to re-instantiate the process directly which is considerably quicker (< 10 sec). Later it became necessary to run several HOL-Light instances in parallel. However, every instance to be run requires a separate checkpoint of the tool. Maintaining a collection of such backups is impractical as it consumes a large amount of storage space. Furthermore, checkpoints would have to be updated for any changes made to the OCaml program. Since they are created manually this was too tedious to keep up through the development process. Instead I decided to restrict the theories loaded at the start of the HOL-Light tool to files on which propositional logic and the IsabelleLight extension depend. Since these are relatively basic theories, the start up time was reduced to ca. 7-10 sec.

The resulting server accepts strings of two formats as input messages. The first prompts HOL-Light to verify the well-formedness of a propositional logic statement and if successful substitutes it for the current goal state. The second applies a given tactic-rule-pair to the current goal state and replies either with an error message or the topmost goal in the resulting goal stack. These are implemented by parsing the input string and calling appropriate functions within the HOL-Light framework.

4.2.3 Parallelisation

In the early stages of the experiments it was found that the HOL-light theorem prover has a relatively large response times (several milliseconds) in terms of machine time scales. Since the tool is intended for use by a human operator this is usually not a problem. In a machine learning context however, this is a severe problem since large amounts of interactions are required to test the proposed neural networks. Since the response time is fixed by the tools internal workings, this issue was addressed by running and querying several independent instances of the tool in parallel.

Since individual neural network can be tested independently from each other, it is possible to parallelise this process. The gene pool is divided into a number of subgroups to be assessed in parallel. Each of these is passed to a worker thread associated with a connection to a separate instance of the HOL-Light module running in its own thread. This has resulted in a significant speed-up and allows to scale the systems response time according to the resources available.

Chapter 5

Experimental results

5.1 Experimental design

To explore avenues of evolving networks capable of successfully choosing natural deduction inference steps in the IsabelleLight framework, a row of experiments were conducted. It was observed, that the evolution process is dependent on various aspects of the experimental design. In particular the fitness function governing the feedback from the theorem prover plays a large role. However, speed and stability of the learning process depend on the parameters of the NEAT algorithm such as mutation rates. Finally, the set of examples which neural networks are presented with at training time has an impact as the process can only incorporate patterns observable in the training data. Given the use of the NEAT algorithm two different approaches can be chosen. On the one hand, it is possible to present a fixed set of problems representative of propositional logic statements to the algorithm. This would effectively constitute a corpus from which the algorithm needs to extract features guiding the choice of inference steps. On the other hand, it is possible to present the algorithm with simple subproblems and iteratively extend the set of problems. As shown in the development of a roving eye for the game of Go this initial experience should facilitate learning on the more complicated sets.

I chose the later approach as it is unclear what set of statements would constitute an accurate representation of ‘typical‘ propositional logic statements and the extension to harder problems is easily realisable within the given framework. The simplest proofs constructable within the chosen system are the inference rules themselves. These can be proven within one to three proof steps and formed the initial starting set of 8 training examples.

Unless specified otherwise the following NEAT parameters were used:

Parameter	Value
Population size	1000
Targeted number of species	65
Amount of generations until stagnation	100
Population retained in each generation	20%
Add Link probability	0.1
Add Node probability	0.1
Similarity factor of avg. weight difference	5
Amount of interspecies crossovers per generation	10
Gen mutation probability	0.9
Gen mutation strength	1.5

These parameters are similar to the ones used for experiments on evolving networks for the roving eye for the game of Go and were initially obtained experimentally. The described experiments were generally run for at least 1000 generations (5-8h) which is twice the amount of generations networks needed to converge to acceptable solutions in the Go setting.

Experiments 1 and 2 - Initial Failure

These experiments aimed to construct networks distinguishing between the eight different inference rules for intuitionistic logic. Feedback was purely based on success or failure to prove the inference rules directly. In the first run networks were assessed based on their performance on each of the individual inference rules. Hereby, each network started with a base fitness level of 0 and attempted to proof each inference rule once. Thereby they received the following bonuses adding up to the maximum fitness value of 1.0:

$$bonus = \begin{cases} 0, & \text{if any invalid inference steps were suggested} \\ 1/8 & \text{if a valid proof was produced} \end{cases}$$

The constructed networks all failed to complete any proof. It was believed that networks repeatedly suggested invalid inference steps and even goal directed steps did not receive positive feedback. Hence, the selection mechanisms were unable to promote ‘useful‘ network traits and the NEAT algorithm effectively performed a random search over all possible neural network architectures which is highly unlikely to succeed. To allow for selection mechanisms to perform better a minor fitness boost for the suggestion of applicable inference steps was introduced even if these did not lead to valid proofs:

$$bonus = \begin{cases} 0, & \text{if any invalid inference steps were suggested} \\ 1/16 & \text{if no valid proof was produced within 5 iterations} \\ 1/8 & \text{if a valid proof was produced} \end{cases}$$

Once again constructed networks failed to proof any of the presented inference rules. Instead it was observed that networks repeatedly choose ‘safe‘ inference

rules applicable regardless of the presented goal. In the early stages such networks would be selected over others as they received higher fitness values. However, they are unable to produce complete proofs and the respective fitness boost remain unattainable.

Experiments 3 and 4 - Positive reinforcement

In addition to the assessments described in previous sections, it appeared to be necessary to give positive feedback for smaller contributions towards the proof construction to enable the NEAT algorithm to find good network formations. Since natural deduction proofs constructed by humans typically break down the statement to be proven, fitness boosts were given for individual steps achieving similar results:

$$bonus = \begin{cases} 1/8 & \text{if a valid proof was produced} \\ \sum_{i=0}^{4} B(step_i) & otherwise \end{cases}$$

$$B(step) = \begin{cases} 0, & \text{if the number of metavariables increased} \\ 1/120, & \text{if the number of remaining meta-variables decreased} \\ 1/120, & \text{if the number of remaining subgoals decreased} \\ 1/120, & \text{if the number of remaining connectives decreased} \\ 1/120, & \text{if the number of remaining variables decreased} \\ 1/80, & otherwise \end{cases}$$

The graph below shows the fitness of the best performing network in each generation. It can be noted that the large 'steps' in the depicted developments generally mark successful learning of a particular training example.

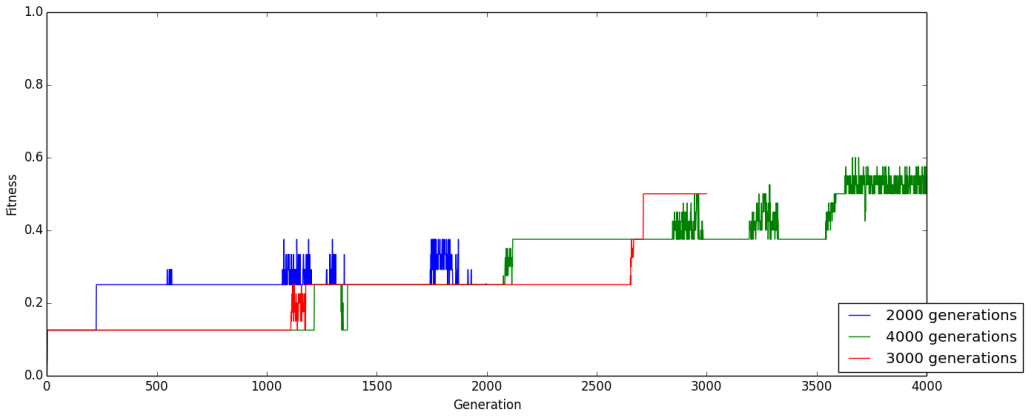

Figure 5.1: Fitness development when promoting minor improvements in the fitness function

While the produced networks failed to distinguish between all different rules, they were able to repeatedly identify 2-3 inference rules correctly after 1000 generations. Increasing the number of generations experiments were run for allowed the construction of networks distinguishing between up to 6 rules (after 6500 generations). Further examination showed, that the networks were able to apply these rules independent of the variable names substituted in the presented goal states.

However, as we can see the learning process is slow even in the seemingly simplistic setting of 8 training examples. The following experiments therefore aimed to identify the sources of these delays and consequently speed up the learning process.

Experiment 5 - Analysing the NEAT parameters

All experiments up to this point had been run with a set of parameters and only minor changes were made. In an effort to determine whether or not there exists a more suitable set of parameters allowing for quicker evolution of these networks.

Four parameters were changed in a grid search fashion, the graphs show how changing them individually changes the overall behaviour of algorithms by depicting the highest fitness in the population at any given point in time.

Graphs 5.2 and 5.3 show experiments with different probabilities of adding a new node or randomly chosen connection to a network during the mutation phase. As we can see higher mutation rates lead to quicker improvements of the results. The network is able to quicker accumulate a sufficient number of hidden units representing different features or evolve features providing more complex summaries across a number of parameters. However, the quicker improvement was found to make the evolution process less stable and predictable when repeating the same experiment.

The other two parameter govern the division of the gene pool into species. Each species is represented by one of its members from the previous generation and is allocated all networks similar enough to this representative. The threshold for these allocations is adjusted dynamically to obtain a constant number of species. The distance between two networks is determined in dependence on the number of innovations not shared between them and the average difference between weights. The impact of the relative weighting factor of that average is depicted in figure 5.4. As we can see, higher values lead to a more beneficial divisions.

Finally, species not improving in fitness are penalised and finally discarded. The second plot shows how the retention time influences the evolution process. In our experiments retaining species for longer than 100 generations seems to give no additional benefits and instead actually hinders improvement (compare figure 5.5).

Thus, it can be noted that the initial estimates found were close to the optimal parameter setting.

Experiment 6 - Pre-evolution

Inspired by pre-evolution process leading to faster learning progress when extending the size of Go boards in [40], it was attempted to speed up the learning process by first challenging the networks to distinguish between simpler rule set. Networks were firstly assessed based on the rules for implication, conjunction or disjunction respectively and the set of considered rules iteratively extend. Ideally, this should benefit the discovery of features distinguishing between inference rules in the simpler settings that can be build upon later in the evolution process. As it can be seen in figure 5.6, this did indeed facilitate quicker learning. However, training progress is still rather slow and no complete solutions were derived in a satisfiable amount of time.

Experiment 7 - Alternate starting points

Since learning in the first 600-800 generations was largely unsuccessful, I suspected that the minimal starting point taken by the NEAT algorithm is actually hindering quick learning in this setting. The lack of hidden units means that no recurrent cycles can be established and sequence analysis fails. Hence, alternate starting points were tested in which the initial pool of neural networks was initialised with a number of hidden units and randomly selected connections resembling the number of genes in the networks evolved so far after 600 and 800 generations. As we can see in figure 5.7, the initial learning process is much quicker than in previous experiments. Networks almost immediately achieve a fitness score of 0.25 indicating the successful completion of two of the presented problems which required at least 300 generations in previous experiments using similar parameter settings. Similarly, a starting point containing even more hidden units and connections leads to the successful completion of an additional training problem after merely 120 generations which is considerably earlier than in previous trials. However, it also becomes apparent that training progress thereafter remains slow indicating that the NEAT algorithm is able to optimise the ‘provided‘ genes but remains incapable to quickly develop additional complex structures needed for further progress.

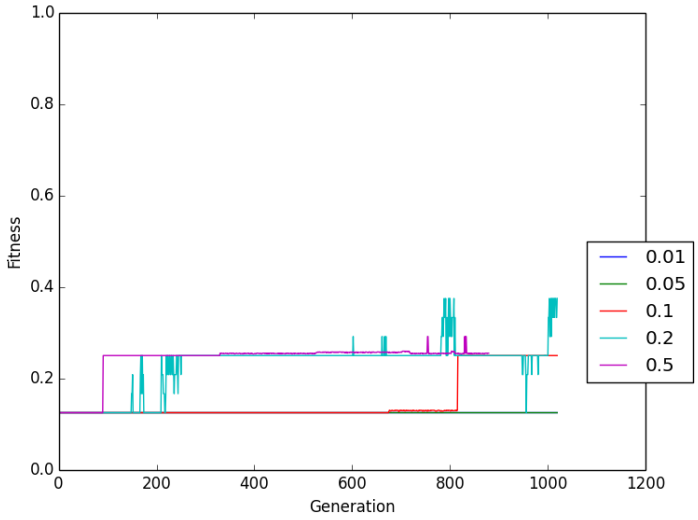

Figure 5.2: Fitness development for varying probabilities for the addition of nodes

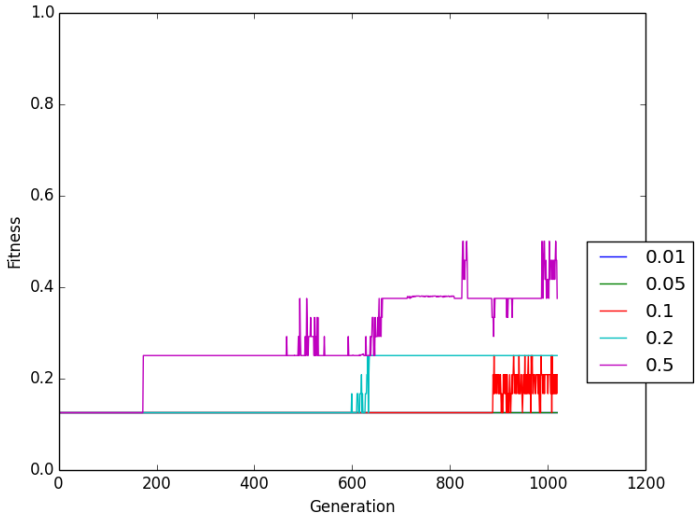

Figure 5.3: Fitness development for varying probabilities for the addition of connections

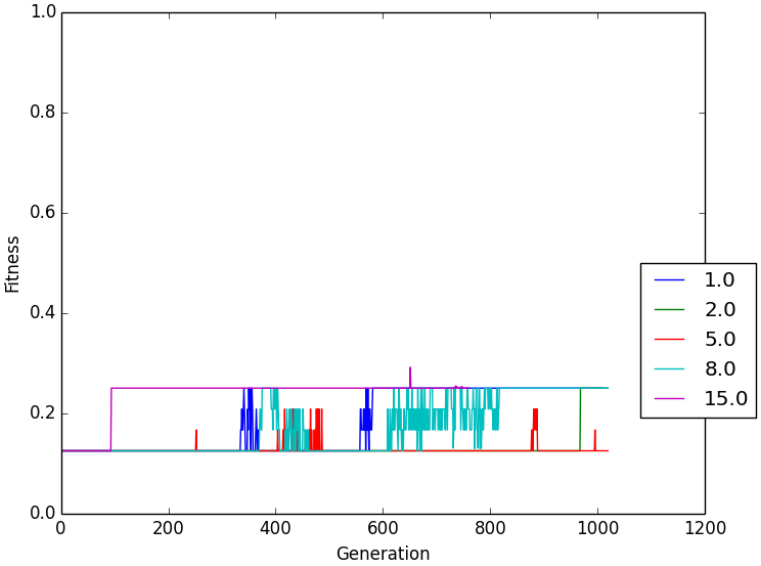

Figure 5.4: Fitness development for varying impacts of connection weights on species allocation

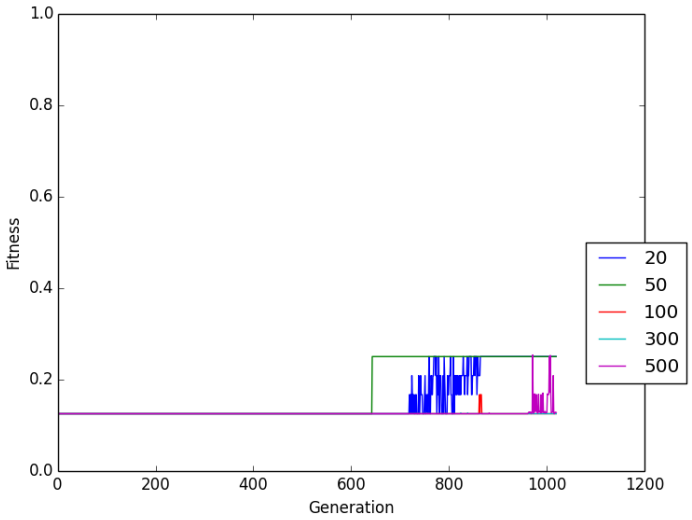

Figure 5.5: Fitness development for varying retention times for stagnant species

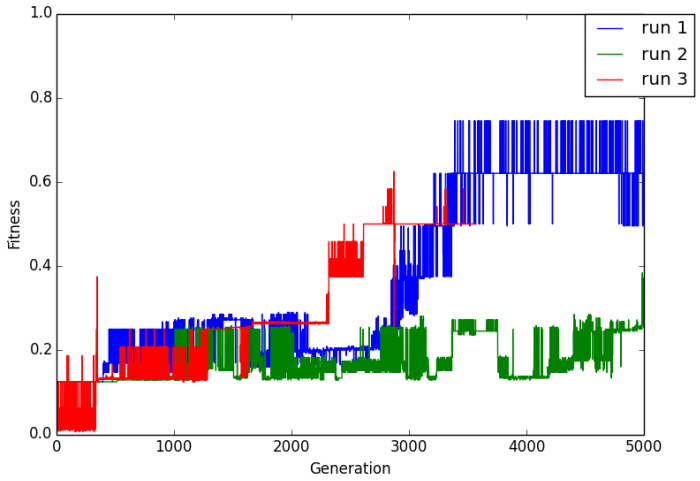

Figure 5.6: Evolution using pre-evolution during training

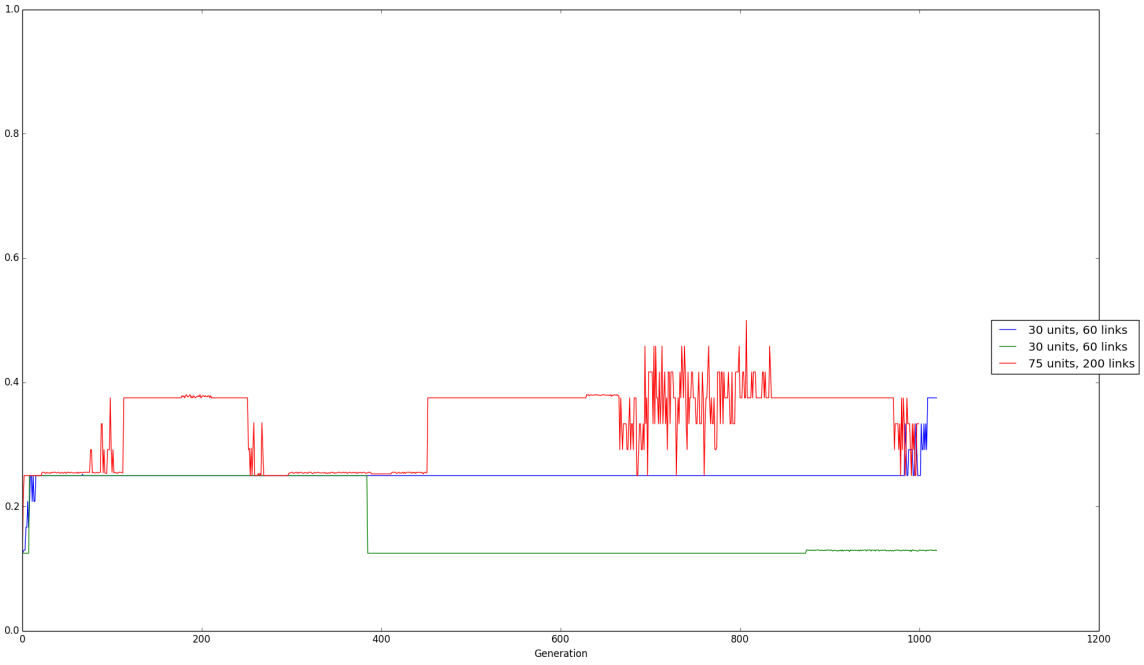

Figure 5.7: Evolution development from alternate starting points

Chapter 6

Discussion and Evaluation

This project explored the construction of neural networks as heuristics over inference steps during the process of formulating natural deduction proofs. This section summarises the findings of the project, and discusses them in the context of future efforts to employ neural networks in the context described.

One of the main difference between this project and previous related research was in the choice of the underlying feature representation of goal states. Where prior efforts generally focussed on the creation of finite, hand-crafted features sets fairly specific to the particular application domain, we chose a symbolic sequence representation for this project. This was inspired by trends in the field of deep learning which tend to solve the problem of feature learning at training time. This has the advantage of making the developed procedures theoretically generalisable to arbitrary dictionaries of underlying symbols. At the same time, it demands the use of recurrent neural networks rather than feed-forward architectures for the processing of symbol sequences.

The second innovation proposed in this project lay in the way a theorem proving tool was directly embedded into the learning process. Theorem proving tools themselves represent an enormous amount of time and effort spent to embed the basis of formal reasoning into computer systems. By training the constructed models on the basis of repeated interactions, the need for a duplication of this effort through the creation of corpora suitable for machine learning purposes is avoided.

Because of the uncertainty regarding the hyper parameters of the heuristics to be constructed we decided to employ the genetic procedure NEAT for the training of neural networks. This training paradigm optimises not only the weights of a neural network but also its other hyper-parameters such as the number and connectedness of hidden units. As it had been successfully employed in other abstract problem settings, it was assumed to be powerful enough to solve the presented optimisation problem. The algorithm additionally aims to discover solutions with minimal amounts of hidden units first, making it less likely to develop overcomplicated solutions capable of overfitting to the training data.

The described innovations were implemented in a framework training neural networks for the construction of natural deduction proofs over intuitionistic propositional logic statements. While experiments found the framework to be incapable of producing fully successful heuristics, partial solutions were derived and we found strong evidence that complete solutions could be constructed given enough time and computational resources. Further experiments suggested, that it was the inability of the NEAT algorithm to quickly construct complex structures to cause the framework to fail. Therefore, it remains inconclusive how the chosen feature representation and network architecture perform when compared to previous approaches. In particular, it remains uncertain how well recurrent neural networks are able to generalise from seen training examples. The fact that produced networks remained invariant towards presented variable names suggests that there is potential to abstract away from given training data as it is necessary to learn useful heuristics for the process of proof construction.

Future projects therefore further explore the possibilities offered by recurrent neural networks as strong heuristics for the task of proof construction. The main problem to overcome is the choice of suitable hyper-parameters for a given problem setting. Hence, it is necessary to identify stronger learning paradigms for the optimisation of these. Here, we can draw on the well-studied field of natural language processing in which recurrent neural networks are routinely employed. While the prevalent approach is to experimentally determine the optimal number of hidden units, a more structured alternative is offered by the field of Baysian optimisation.

Bibliography

[1] Andrea Asperti. A survey on interactive theorem proving. *URL:http://www.cs.unibo.it/ãsperti/SLIDES/itp.pdf*, 2009.

[2] Jeremy Avigad. Interactive theorem proving. *URL:https://www.andrew.cmu.edu/user/avigad/Talks/ioannina.pdf*, 2011.

[3] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends in Machine Learning®*, 2(1):1–127, 2009.

[4] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[5] Alexander Mordvintsev, Michael Tyka, and Christopher Olah. Inceptionism: Going deeper into neural networks, google research blog. *Retreived June*, 17, 2015.

[6] Andrzej Indrzejczak. Natural deduction. *The Internet Encyclopedia of Philosophy (IEP)*, October 2015.

[7] Kevin C. Klement. Propositional logic. In *The Internet Encyclopedia of Philosophy*.

[8] John Harrison. The hol light theorem prover. *URL: https://www.cl.cam.ac.uk/ jrh13/hol-light/*, October 2015.

[9] Harold Cohen. The further exploits of aaron, painter. *Stanford Humanities Review*, 4(2):141–158, 1995.

[10] James Garson. Connectionism. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2015 edition, 2015.

[11] P. M. Gibson and J. A. Byrne. Neurogen, musical composition using genetic algorithms and cooperating neural networks. In *Artificial Neural Networks, 1991., Second International Conference on*, pages 309–313, Nov 1991.

[12] Peter M. Todd. A connectionist approach to algorithmic composition. *Computer Music Journal*, 13(4):27–43, 1989.

[13] Simon Colton. The painting fool: Stories from building an automated painter. In *Computers and creativity*, pages 3–38. Springer, 2012.

[14] Anna Krzeczkowska, Jad El-Hage, Simon Colton, and Stephen Clark. Automated collage generation-with intent. In *Proceedings of the 1st international conference on computational creativity*, page 20, 2010.

[15] Marco Schorlemmer, Alan Smaill, Oliver Kutz, Simon Colton, Emilios Cambouropoulos, Alison Pease, et al. Coinvent: Towards a computational concept invention theory. 2014.

[16] Graeme Ritchie. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines*, 17(1):67–99, 2007.

[17] Simon Colton. The hr program for theorem generation. In *Automated DeductionCADE-18*, pages 285–289. Springer, 2002.

[18] Simon Colton, Alan Bundy, and Toby Walsh. Automatic identification of mathematical concepts. In *ICML*, pages 183–190, 2000.

[19] Samuel R Buss et al. Weak formal systems and connections to computational complexity. *Student-written Lecture Notes for a Topics Course at UC Berkeley*, 1988.

[20] Joan Moschovakis. Intuitionistic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2015 edition, 2015.

[21] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[22] Petros Papapanagiotou and Jacques Fleuriot. Isabelle light: An isabelle-like procedural mode for hol light. 2010.

[23] Jeremy Avigad. Formal verification, interactive theorem proving, and automated reasoning. *URL:https://www.andrew.cmu.edu/user/avigad/Talks/baltimore.pdf*, 2014.

[24] Jeremy Avigad. Interactive theorem proving, automated reasoning, and mathematical computation. *https://icerm.brown.edu/materials/Slides/tw-12-5/Interactive_theorem_proving,_automated_reasoning,_and_mathematical_computation_%5D_Jeremy_Avigad,_Carnegie_Mellon_University.pdf*, 2012.

[25] Robin Milner. Logic for computable functions description of a machine implementation. Technical report, DTIC Document, 1972.

[26] Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–186, 2000.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[28] Michael Nielsen. Neural networks and deep learning. *http://neuralnetworksanddeeplearning.com/*, 2016.

[29] Li Deng and D Yu. Foundations and trends in signal processing. *Signal Processing*, 7:3–4, 2014.

[30] K. Hara and K. Nakayamma. Comparison of activation functions in multilayer neural network for pattern classification. In *Neural Networks, 1994. IEEE World Congress on Computational Intelligence., 1994 IEEE International Conference on*, volume 5, pages 2997–3002 vol.5, Jun 1994.

[31] Michiel Hermans and Benjamin Schrauwen. Training and analysing deep recurrent neural networks. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 190–198. Curran Associates Inc., 2013.

[32] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *URL:http://karpathy.github.io/2015/05/21/rnn-effectiveness*, May 2015.

[33] Kenneth O. Stanley. *Efficient evolution of neural networks through complexification.* PhD thesis, 2004.

[34] C. Mareco and A. Paccanaro. *Computational Methods and Neural Networks*, chapter Using Neural Networks to improve the performance of Automated Theorem Provers, pages 379–404. Dynamic Publishers, Inc., 1998.

[35] Christoph Goller. Learning search-control heuristics for automated deduction systems with folding architecture networks. In *ESANN*, pages 45–50. Citeseer, 1999.

[36] Michael Blanchard, J Horton, and Dawn MacIsaac. Folding architecture networks improve the performance of otter. In *Short Paper at 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2006) Tuesday*, volume 14, 2006.

[37] Christian Suttner and Wolfgang Ertel. *10th International Conference on Automated Deduction: Kaiserslautern, FRG, July 24–27, 1990 Proceedings*, chapter Automatic acquisition of search guiding heuristics, pages 470–484. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.

[38] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, Aug 2013.

[39] K. Arya, G. Cooperman, R. Garg, J. Cao, and A. Polyakov. Dmtcp: Distributed multithreaded checkpointing. *URL: http://dmtcp.sourceforge.net/index.html*, Feb 2016.

[40] Kenneth O Stanley and Risto Miikkulainen. Evolving a roving eye for go. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 1226–1238. Springer, 2004.