Summer Research Program in Industrial and Applied Mathematics

Sponsor

Real AI

Final Report

Towards Automated Theorem Proving: A Deep Learning Approach

Student Members

Gwang Hyeon CHOI, *Seoul National University*

Seung Yong MOON, *Seoul National University*

Zheng PAN, *The University of Hong Kong*

Jiaze SUN, *The University of Hong Kong*

Academic Mentor

Queenie LEE, `goldendragonadviser@gmail.com`

Sponsoring Mentor

Jonathan YAN, `jyan@realai.org`

Date: August 5, 2017

Contents

List of Figures

Abstract

When AlphaGo was revealed in 2016, it shocked everyone with its winning streak against the top Go masters around the world. With computer hardware becomes increasingly powerful, deep learning has shown enormous potential not just in Go playing, but also various other tasks, like proving mathematical theorems. This project works on Holophrasm, an automated theorem prover inspired by AlphaGo. In particular, this report gives a detailed description of Holophrasm's main algorithm, including the concept of proof trees, and the networks involved in the tree search process. Furthermore, this project also uses TFLearn, which is a high-level API based on TensorFlow, to partially improve the original Holophrasm.

Acknowledgments

We would like to express our sincere gratitude for our industrial mentor and director of Real AI Jonathan Yan for all his enlightening guidance and instructions. We are also grateful that our academic mentor Queenie Lee has always been resourceful and supportive. Finally, huge thanks to The Hong Kong University of Science and Technology (HKUST) and Seoul National University for organizing the SPIA program, especially to Dr. Albert Ku and Prof. Shing-Yu Leung from HKUST for coordinating both academic and leisure activities throughout the program.

Chapter 1

Introduction

1.1 Real AI

Real AI is a company based in Hong Kong, and as its name suggests, focuses on the research and development of safe and beneficial artificial general intelligence (AGI) [5]. Adopting the principle of effective altruism, Real AI aims to develop altruistic AI to help solve many of today's problems, such as disease, climate change, or even poverty. Currently, Real AI has two primary functions, one of which is to monitor and assess the latest development in deep learning, and the other is to conduct technical research on AGI.

1.2 Problem Background

Automated Theorem Proving (ATP) is the implementation of computers, or more precisely, artificial intelligence, in producing mathematical proofs. The task of constructing proofs is an arduous one, since it usually requires one to have substantial experience and knowledge in mathematics. It might take a human mind years and perhaps decades of intensive training to be able to successfully produce complex proofs, but with the help of deep learning and modern computer hardware, it is plausible to "teach" an AI mathematics and let it generate proofs for us in a much more efficient and rigorous manner. In addition, the research of ATP is also an important step in the advancement of AGI, as it signifies the next stage for AI evolution, that of reasoning.

1.3 Statement of Problem

The main objective of this project is to study, implement and improve Holophrasm [10], a neural automated theorem prover written in Python. In order for a computer to work on mathematical proofs, we must firstly formalize the mathematics that we humans understand into one that can be processed by the computers, and this is where formal mathematical systems come into play. Within such a system, every step of a proof follows directly from the previous one, and all theorems can be verified algorithmatically. Holophrasm is built using a dataset written in such a formal language called Metamath [3].

The main algorithm of Holophrasm transforms the task of searching for proofs into a tree-search problem, which is somewhat analogous to the core idea of AlphaGo [6]. The main difficulty in this case is that the search space is practically infinite, i.e. the number of possible actions to take at a node is immense. Therefore, three neural networks are implemented to help tackle the problem, which are the relevance, generative, and payoff networks. The details of the networks are covered in Chapter 4.

However, the current version of Holophrasm is still preliminary and yet to achieve its final form. Now it can provide correct proofs to approximately 14% of the test theorems in Metamaths `set.mm` database [10]. With the development of ATP still in its infancy and the emergence of powerful deep learning libraries such as TensorFlow, Holophrasm has much potential for improvements. This project aims to gain a good understanding of Holophrasm, and make possible modifications to the structure and networks of the program using TFLearn, a higher level API built upon TensorFlow [8].

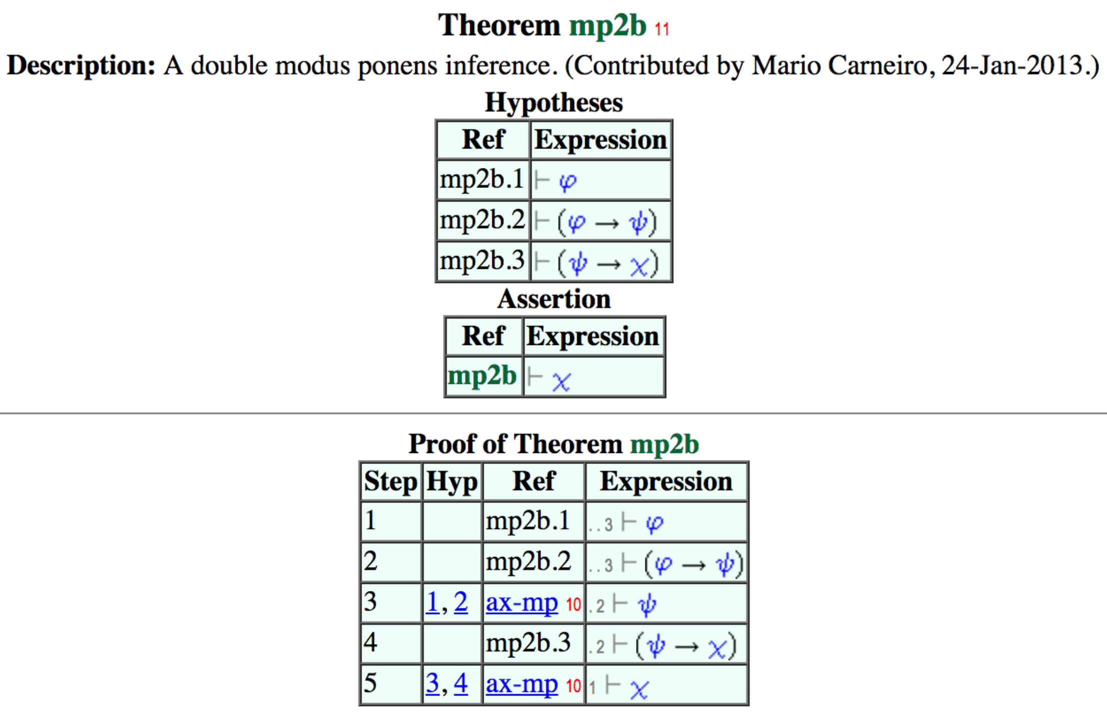
Chapter 2

Metamath: A Formal Mathematical System

A formal mathematical system is a computer language which enables a computer to work on mathematical proofs, such as performing proof verification. Metamath [3], meaning "metavariable mathematics", is such a formal language, and it lays the groundwork for Holophrasm. The Metamath online database includes over 19,000 propositions, all of which are derived from a list of only 22 axioms, which constitute the foundation of mathematics known as the Zermelo-Fraenkel set theory (ZFC). The propositions included in the Metamath database range from simple logical rules like *Modus Ponens* to complicated ones found in complex analysis, abstract algebra and so forth.

2.1 Metamath Database Structure

The Metamath database consists of propositions which either belong to the 22 axioms in the ZFC framework or are derived from them. As shown in Figure 2.1, a proposition C, or context, can have at most three components, a set of hypotheses e_C, an assertion a_C, and a proof. Every proposition must have an assertion; one without hypotheses is tautological; and one without a proof is axiomatic. A hypothesis in e_C contains a label and a statement, and so does a_C except that a_C is labeled by the name of C. A proof can be displayed in a tabular format, where each line is a step. Every step has different components depending on its type, and they can be categorized into two main types. A Type I step is a hypothesis from e_C, thus no theorem is applied in the step, and it contains a label and the statement of that hypothesis (see Figure 2.1 step 1, 2, and 4); a Type II step gives an assertion a that is entailed by a set of some previous steps, e_a, in which case a theorem is applied, and it will contain indices of those previous steps, the name of the theorem being applied, and the statement of a (see Figure 2.1 step 3 and 5). Note that for a Type II step, if e_a is empty, which means the step does not use any previous steps, then a is necessarily tautological.



Theorem mp2b 11

Description: A double modus ponens inference. (Contributed by Mario Carneiro, 24-Jan-2013.)

Hypotheses

Ref	Expression
mp2b.1	$\vdash \varphi$
mp2b.2	$\vdash (\varphi \to \psi)$
mp2b.3	$\vdash (\psi \to \chi)$

Assertion

Ref	Expression
mp2b	$\vdash \chi$

Proof of Theorem mp2b

Step	Hyp	Ref	Expression
1		mp2b.1	..3 $\vdash \varphi$
2		mp2b.2	..3 $\vdash (\varphi \to \psi)$
3	1, 2	ax-mp 10	.2 $\vdash \psi$
4		mp2b.3	.2 $\vdash (\psi \to \chi)$
5	3, 4	ax-mp 10	1 $\vdash \chi$

Figure 2.1: Example proposition in Metamath [4]

2.2 Metamath Proofs

Proving mathematical theorems essentially revolves around applying proven or axiomatic propositions to the current context, via a very straightforward process called substitution. Consider an Type II step with statement a in the proof of a context proposition C. Denote by a_C, e_C, and f_C respectively the context's assertion, hypotheses, and set of variables appearing in a_C or e_C. Let T be the theorem being applied in the step, with a_T, e_T, and f_T being defined similarly. A substitution, ϕ, is an operator that replaces variables in f_T by those in f_C, and satisfies $\phi(a_T) = a$. However, the substitution cannot be carried out casually, as there are different types of variables within the Metamath system, and the variables which replace the free variables must be of the corresponding type, otherwise the substitution is improper and might result in an incorrect proof.

Substitutions can be separated into 2 categories depending on the types of variables they replace. A variable of a given proposition T is constrained if it appears in a_T, and unconstrained otherwise. Hence, a substitution is said to be constrained if it replaces a constrained variable, and unconstrained otherwise. With this concept in mind, given an assertion a and a theorem T to be applied, it can be observed that constrained substitutions are generally much easier to determine, since one only has to compare a with a_T. Unconstrained substitutions, on the other hand, are usually more difficult to produce and require one to have sufficient experience in mathematics.

2.3 A Note on Syntax

The preceding discussion only focuses on the general principles of Metamath and does not involve the actual syntax of the language. To write down a proposition and its proof in Metamath is in fact a much more complicated task. For instance, the hypotheses e_C of a context proposition C as defined in the previous sections are merely its e-type hypotheses; a proposition also has f-type hypotheses which handle the declaration of variables that appear in e_C and a_C. Furthermore, the proof of C is written down in a way analogous to Reverse Polish Notation, where T acts as the operator, e_a contains the operands, and a is the output. The proof verifier regards the proof as instructions on creating statements, and if it is able to create a_C from the proof, it will consider the proof as correct. The book *Metamath: A Computer Language for Pure Mathematics* by Norman Megill [3] gives a very detailed description on the matter.

Chapter 3

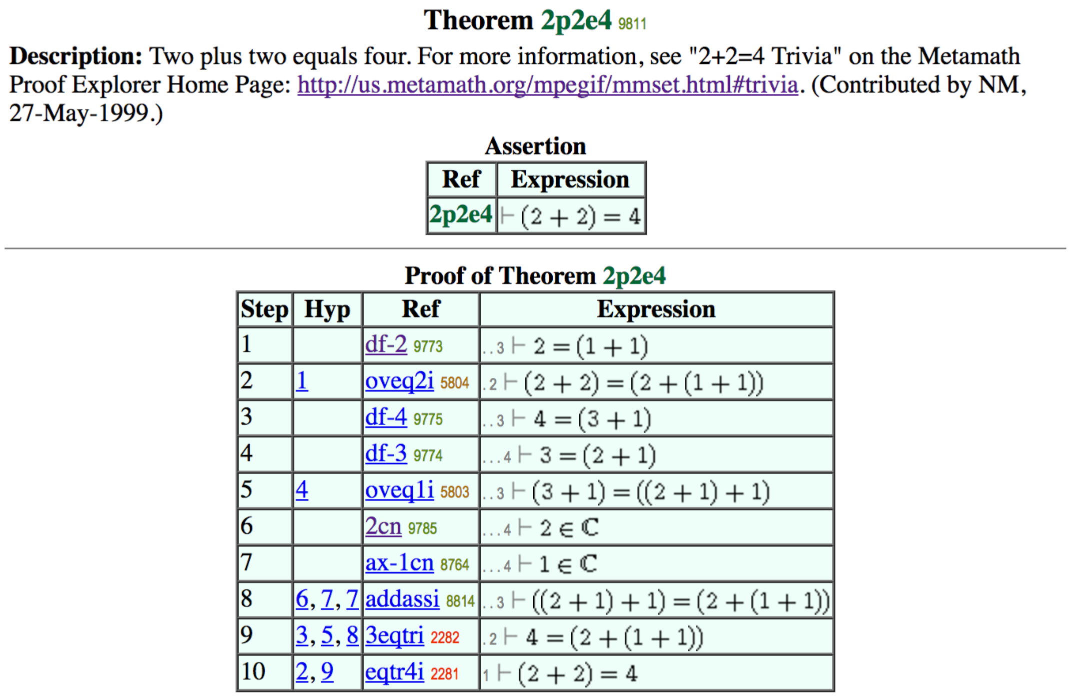
Proof Tree

Metamath's central feature that gives rise to Holophrasm is that under this system, the proof of every proposition can be written down in a tree structure. Thus, proving a proposition is essentially equivalent to finding such a tree. Holophrasm is designed to exploit this property, by combining techniques from tree exploration algorithms and recurrent neural networks.

3.1 Construction of the Proof Tree

Suppose we are given a context proposition C with an assertion a_C, hypotheses e_C, and its proof. To construct the proof tree for C, we start with the last step (which must be a_C) in the proof and go backwards. First of all, we put down a_C as the root node of the tree, label it by a_C and give it the color red. If the last step is of Type I, i.e. it is one of the hypotheses in e_C, then we are done. Suppose it is a Type II step, and is derived using T, we then draw one child node stemming from a_C, label the child node by T, and give it the color blue. If the last step has no steps entailing it, i.e. it is tautological, we can stop the process; otherwise, we suppose a substitution ϕ is carried out. Then for each statement in $\phi(e_T)$, we draw a child node stemming from the blue node labeled by T, label each of them by their corresponding statement and give them the color red again. Now, for each of the bottom-most red nodes, we check if it is in e_C. If it is, we take no action; otherwise it must be a Type II step, so we repeat what we have done with a_C to all of them until we cannot take any further actions. As an illustration, Figure 3.1 shows the contents of Theorem 2p2e4, which asserts that $2 + 2 = 4$; Figure 3.2 shows its proof tree. In this particular example, the theorem itself is a tautology, so all of the bottom-most nodes are blue nodes.

From this description it is easy to observe that there are two types of nodes in the proof tree, red and blue. Red nodes are either hypotheses or assertions, blue nodes are theorems along with, if any, their corresponding substitutions. A red node always has exactly one child blue node, unless it is in e_C, in which case it has no child; a blue node can have anything from zero to several child red nodes. If there exists such a tree, it is a valid proof of C.



Theorem 2p2e4 9811

Description: Two plus two equals four. For more information, see "2+2=4 Trivia" on the Metamath Proof Explorer Home Page: http://us.metamath.org/mpegif/mmset.html#trivia. (Contributed by NM, 27-May-1999.)

Assertion

Ref	Expression
2p2e4	$\vdash (2 + 2) = 4$

Proof of Theorem 2p2e4

Step	Hyp	Ref	Expression
1		df-2 9773	..3 $\vdash 2 = (1 + 1)$
2	1	oveq2i 5804	.2 $\vdash (2 + 2) = (2 + (1 + 1))$
3		df-4 9775	..3 $\vdash 4 = (3 + 1)$
4		df-3 9774	...4 $\vdash 3 = (2 + 1)$
5	4	oveq1i 5803	..3 $\vdash (3 + 1) = ((2 + 1) + 1)$
6		2cn 9785	...4 $\vdash 2 \in \mathbb{C}$
7		ax-1cn 8764	...4 $\vdash 1 \in \mathbb{C}$
8	6, 7, 7	addassi 8814	..3 $\vdash ((2 + 1) + 1) = (2 + (1 + 1))$
9	3, 5, 8	3eqtri 2282	.2 $\vdash 4 = (2 + (1 + 1))$
10	2, 9	eqtr4i 2281	1 $\vdash (2 + 2) = 4$

Figure 3.1: The contents of Theorem 2p2e4 [4]

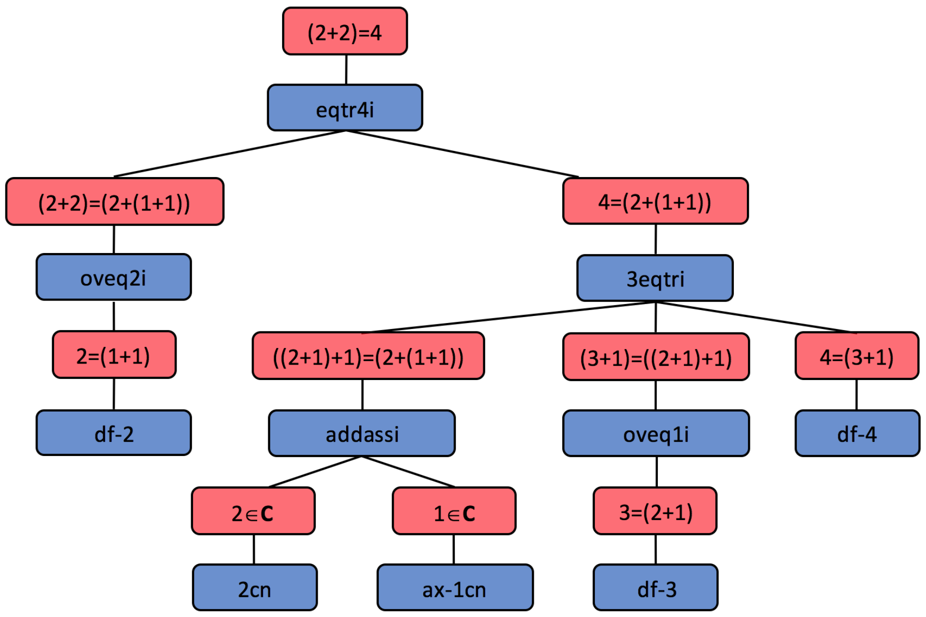


Figure 3.2: The proof tree of Theorem 2p2e4 [4]

3.2 Searching for a Proof Tree

3.2.1 Partial Proof Tree (PPT)

A PPT is essentially an expanded proof tree, and is exactly what the algorithm attempts to construct. In a PPT, red nodes are allowed to have multiple child blue nodes, and to have no children even if it is not in e_C. In addition, every node has a status of being "proven" or "unproven". A red node is proven if one of its child blue node is proven or if it is in e_C; a blue node is proven when all of its child red nodes are proven or if it is tautological. The sub-tree of a proven red node, which is a PPT, can be reduced to a valid proof tree for a_C by removing all unproven blue nodes from that sub-tree.

All nodes in a PPT keep track of their own values and visit counts, and they are initialized upon creation of the nodes and updated at the end of every pass. Every node is given at least one value, which can be interpreted as the likelihood that the given node can be proven. A red node, r, is given an initial value y_r as soon as it is created; they also have three additional items: a total value, t_r, which is the sum of their initial value and the values of their child blue nodes, a visit count, n_r, which is the number of times the node has been visited, and a mean value, m_r, which is the total value divided by the visit count. A blue node, b, has three items, the first two of which correspond to the value and visit count of its worst child red node, i.e. the one with the minimum total value (this is sensible because a blue node is only considered proven when all its child red nodes are proven); the third item is the relevant value, v_b, which measures how likely the theorem is going to be applicable.

To facilitate our following discussion, a few more definitions will be given here:

- A red node is called dead if all its child blue nodes are dead and the algorithm cannot find new theorems to add to its list of children; and a blue node is dead if at least one of its child red nodes is dead.
- The algorithm also avoids circularity, which is the scenario where applying a theorem to a red node leads to creating that same red node or one of its ancestor red nodes.
- A red node a also keeps track of its childless visits, l_a, which is the number of visits where no new child blue node is created.

3.2.2 PPT Exploration and Expansion

The main algorithm of Holophrasm is to construct a PPT over a sequence of passes. Suppose we are given a context proposition C along with its assertion a_C and hypotheses e_C. We first set a_C as the unproven root red node of the PPT, and every hypothesis in e_C as a proven red node. As soon as this is completed, the algorithm will start traversing the tree. In the first pass, the algorithm starts by visiting the root red node. While it is there, it will search in the Metamath database for potential theorems to apply. If the form of a_C cannot be fitted into any theorem in the database, the algorithm will stop and consider this as a fail; otherwise it will return a list of applicable theorems and choose the best one, say T, apply a set of suitable substitutions ϕ, and return $\phi(e_T)$. If applying T does not lead to circularity, then T will be added to the PPT as a child blue node of a_C, while everything in $\phi(e_T)$ will be added to the PPT as child red nodes of T. At this point, the algorithm will update the proven status and value of every node that has been visited in the current pass, including the root node. If the root node becomes proven, the algorithm stops and we have found a proof tree; otherwise, the algorithm will traverse the tree from the root again, until the root is proven, dead, or, the maximum number of passes or time limit has been reached.

Subsequent passes might be different from the first one. Whenever the algorithm encounters a red node r with $n > 0$ child blue nodes, it will first decide whether to create a new child blue node (given there are still applicable theorems to add) by examining the following condition:

$$\frac{n_r}{6} + 0.01 > n + l.$$

The algorithm will create a new blue child if the above condition is true, and will not otherwise. Note that if $n = 0$, the algorithm will ignore the above condition, and will proceed to add a new child blue node if there are still applicable theorems. If the algorithm has decided not to add a new child blue node, it will then need to decide which existing child blue node to visit by calculating

$$\frac{t_b}{n_b} + \beta\frac{v_b}{n_b} + \alpha\sqrt{\frac{\ln n_r}{n_b}}$$

for every child blue node b of r, where α and β are set to be 1 and 0.5 respectively. Upon identifying the child blue node with the highest score, the algorithm will visit in turn the said blue node and its worst child red node, at which point the current pass ends and the algorithm will continue until the a stopping criterion is met.

Chapter 4

The Neural networks in Holophrasm

In the previous chapter we discussed how the algorithm attempts to search for a valid proof tree for a given context proposition. However, three tasks remain difficult to accomplish. The first one is to determine the best theorem to apply for an assertion given a list of applicable theorems; the second is to generate unconstrained substitutions in order for a given theorem to be applied; and the last one is to find the initial value y_r of a given red node r. Hence, three separate neural networks are created to solve respectively these problems, and they are the relevance, generative, and payoff networks.

4.1 Data preprocessing

All three networks takes an assertion and a set of hypotheses as inputs, both of which have tree structures (see Figure 4.1). The nodes of the trees are construction axioms, which consists of logical symbols such as implication ($\rightarrow$) or intersection ($\cap$), and the names of variables. The trees of the statement and the hypotheses are transformed into arrays of 132-dimensional vectors, each of which has the length equal to the number of the nodes in the corresponding tree. Each vector contains an information of each node; the first 128 entries represent its constructor axiom, the last 4 entries represent its depth, parent's arity, its arity and its leaf position. The arrays are then merged into one array, with some special tokens added at the end of each statement to separate them.

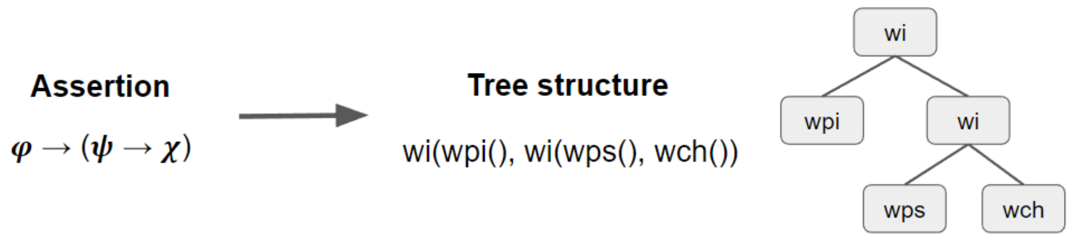

Assertion
$\varphi \rightarrow (\psi \rightarrow \chi)$
Tree structure
wi(wpi(), wi(wps(), wch()))
wi
wpi
wi
wps
wch

Figure 4.1: Tree structure of a statement

4.2 Relevance network

Given a red node with statement a in a context C, and a set of hypotheses e_C, the relevance network attempts to find a proposition that will be used to prove the statement. Since the number of propositions is too large, it is very inefficient to use all the propositions in the training process. To solve the problem, a negative sampling technique, which selects a small number of propositions in each step of the training, is used in the training process. The network consists of two partial networks, both of which use recurrent neural network with GRU cells. The first network takes a statement a and a set of hypotheses e_C as inputs, and outputs an 128-dimensional vector v. In the second network, five propositions are chosen from the Matamath database: one correct proposition T_C and four wrong propositions T_{W_i}, chosen randomly, whose statements are similar to the given statement so that they can be applied to prove the statement. Then, the selected propositions are fed into the network separately. The network takes a statement a_T and a set of hypotheses e_T of a proposition as inputs and outputs an 128-dimensional vector w_T. The relevance network is trained by minimizing the negative sampling loss defined by

$$-\ln(\sigma(l_T)) - \sum_i \ln \sigma(-l_{T_{W_i}}),$$

where σ is the sigmoid function and $l_T = v^T W w_T$ for some 128×128 weight matrix W.

To find an appropriate proposition for a statement using the trained network, all viable propositions are chosen and l_T is calculated for each selected proposition, which are then fed through a softmax function to obtain the probabilities P_T for all the propositions.

4.3 Generative network

Given an assertion a, and a context C, a theorem T has been chosen by the relevance network, we must find suitable substitutions in order to apply it, and the generative network helps determine just what kind of unconstrained substitutions are to be used. The network's inputs are a set of hypotheses of the context e_C, and the substituted hypotheses $\phi(e_T)$ of a theorem T; constrained variables are changed so that the assertion of a theorem after the substitution ($\phi(a_T)$) is equal to the given assertion a. An unconstrained variable is randomly chosen and set as the target, the network then gives a suitable substitution for the target variable. The network repeats this process until all unconstrained variables have substitutions. A sequence-to-sequence model with GRU blocks, as well as an attention model, are used in the network.

4.4 Payoff network

The payoff network measures the likelihood that a given assertion a can be proven by taking a and e_C as inputs, which are fed into a bi-direction network with GRUs.

The network has two fully-connected layers, the outputs from both directions of the first layer are concatenated and subsequently fed into the second layer. Lastly, everything goes through a last fully-connected layer to produce the output value. The previous two layers uses a RELU function while the last layer uses sigmoid. Like the relevance network, the payoff network is trained using correct proof steps as postive examples, and incorrect proof steps produced by the relevance and generative networks as negative examples.

Chapter 5

Implementing TFLearn

Whalen created his own library for Holophrasm, so working on it proved to be a rather challenging task. Hence, we implemented a well-known library, TFLearn, in order to make the whole process more efficient.

5.1 Advantages of TensorFlow

TensorFlow is one of the most widely used deep learning library, thanks to many of its advantages. It was developed by Google Brain and currently used in many academic and industrial deep learning programs, such as drug discovery, translation, and image recognition [1].

5.1.1 Development Supports

Google keeps expanding and maintaining TensorFlow so that developers can use TensorFlow across different operating systems (Linux, MAC OS, Windows, Android) and various computer languages (Python, C++, Java, etc.) [7]. The most recent stable version r1.2 was released on April 17, 2017, and recent developed version r1.3 was released on July 19, 2017. In addition, there is a plethora of examples on how to implement neural networks, including state-of-the-art models. Google's researchers also keep their TensorFlow source codes public on Github [11][9], making them readily available for easy implementation and modifications. As a result, future developers would not have to go through Whalen's own library if they wish to work on

Figure 5.1: TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc. [7]

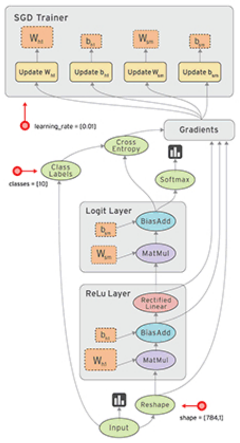


Figure 5.2: TensorBoard, visualization of neural network
[7]

Holophrasm, but simply need knwoledge on TensorFlow.

5.1.2 Hardware Supports

Currently Holophrasm supports only CPU training in a single computer, and its multiprocessing method (in withpool.py file) performs less than ideal on Windows. Fortunately, TensorFlow has APIs for GPU, multiprocessing, and distributed training working in Linux, Windows, Android, and Mac OS [7], thus providing a huge amount of computational resources for us to use larger and more complicated neural networks.

5.1.3 Tensorboard: Visualization of Neural Network

TensorFlow incorporates a utility called Tensorboard, which allows developers to visualize their own neural networks or monitor the training progress, thus making the development and debugging process much more intuitive and efficient. [7].

5.2 TFLearn

TFLearn is a higher-level API based on TensorFlow developed by Aymeric Damien[8], and is a powerful and handy library for implementing TensorFlow.

5.2.1 Layer Construction

TFLearn has many built-in neural network frameworks. With TFLearn, one is able to create embedding layers, convolutional layers, GRU layers, or other kinds of layers in an efficient and intuitive way, significantly reducing our workload.

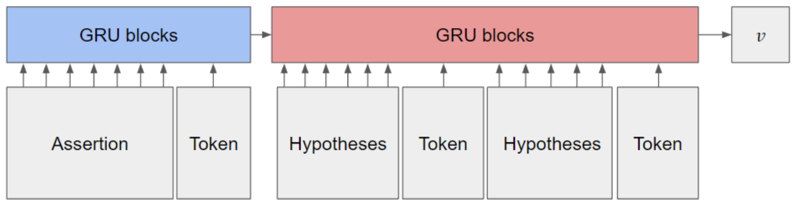


Figure 5.3: The flow of the first network

5.2.2 Automatic Multi-GPU Detection and Training

Although TensorFlow has high-level APIs for GPU training, we need extra works to use GPUs for training, including declaration of GPU to use in desired computation. In training, TFLearn can automatically detect GPUs and distribute computations, which makes future development viable to more researchers.

5.2.3 Easy Construction of TensorBoard

If we want to visualize neural networks with TensorFlow, we have to state all the information to show in TensorBoard for every layer, making it a very tedious task. With TFLearn, however, we can replace these statements with a single integer variable declaration. As a result, we were able to visualize the relevance network with minimum work.

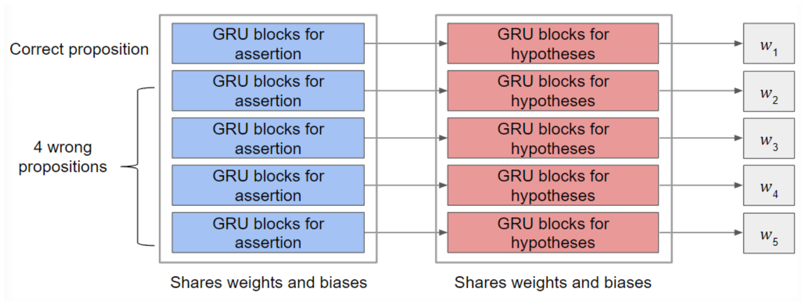
5.2.4 Full Transparency over TensorFlow

TFLearn is not developed by large companies or organizations, so it does not have certain built-in neural network layers, such as attention or sequence-to-sequence models. However, this disadvantage poses little challenge to us as TFLearn supports full transparency over TensorFlow. All TFLearn codes are based on TensorFlow and can be used together. TFLearn website has example codes of the networks mentioned as instances [8]. Holophrasm's relevance network is also different from typical networks, so it was replaced using both TensorFlow and TFLearn.

5.3 Replacing the Relevance Network Using TFLearn

5.3.1 Data Pre-processing

The input data of the relevance network has different lengths. Therefore, we had to use a dynamic RNN, which captures the length of a input automatically, and generates as many GRU blocks as required. However, in order to implement dynamic recurrent neural network using TFLearn, we had to fix the maximum time steps of RNN and add zero vectors to an input so that the length of the input is equal to the maximum time steps. Therefore, 132-dimensional zero vectors were padded on the training data so that each datum had the same length.



-

Figure 5.4: The flow of the second network

5.3.2 Constructing the network

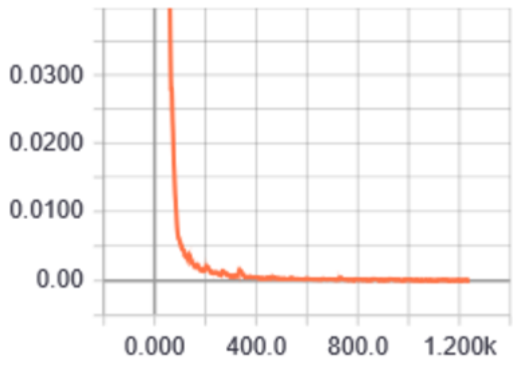
In the first network, two different GRU blocks were used; one for a statement, and the other for a set of hypotheses (see Figure 5.3). The second network has five parallel networks, each of which takes one proposition as an input, and has the same structure as the first network. GRU cells in each network shares their weights and biases with those in the other networks (see Figure 5.4).

5.3.3 Training the network

We used the first 2000 propositions in the Metamath database to create a data set. Proof trees for training were extracted from 80% of the data set, where number of the training data was 6480; the validation data were created from the 10% of the propositions, where the number of the validation data was 857. We used Adam optimizer for stochastic gradient descent optimization. The batch size was 100 and we ran 20 epochs. The loss was shown in Figure 6.4.

5.3.4 Results

we can see that the loss of the network converges to zero. Since we used the negative sampling technique, a separate network must be created for testing. However, we were not able to work on that due to the time limit.



-

Figure 5.5: The training loss of the network

Chapter 6

Conclusion

We have covered Holophrasm, an automated theorem prover which uses a tree search algorithm that is guided by three different neural networks. The program is written based on the Metamath formal language, and is made possible based on the fact that every proof can be organized into a tree structure. The algorithm searches for such a proof tree by first attempting to construct a PPT in which the root node is proven and then trimming it down into a standard proof tree. The three neural networks are created to guide the searching processs. In particular, the relevance network determines the best theorem to apply, the generative network produces unconstrained substitutions for a given theorem, and the payoff network evaluates how likely an assertion can be proven. Each node keeps track of their value and status, and the values are then used by the algorithm to determine which nodes to visit or when to expand the PPT.

Apart from understanding Holophrasm, we also tried to improve it by replacing its relevance network with a well-known library, TensorFlow. Neural networks written in TensorFlow can be supported by Google's consistent updates, huge amount of example models, powerful training tools and visualization. In particular, we used a higher-level API based on TensorFlow, TFLearn, which supports full transparency over TensorFlow. With it, we are able to construct neural networks and visualizations with just a few lines of codes, significantly improving our work efficiency.

There are still substantial improvements to be made with Holophrasm. For example, the generative and payoff networks can also be replaced using the TFLearn library to be made more efficient and better organized. Furthermore, validating and testing the networks are also required in order to improve its performance.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*. `http://tensorflow.org/`, 2015. Software available from tensorflow.org.

[2] D. P. Kingma and J. Ba, *Adam: a method for stochastic optimization*, 2015.

[3] N. Megill, *Metamath: A Computer Language for Pure Mathematics*, Lulu Press, Morrisville, NC, 2007.

[4] Metamath, *Metamath proof explorer home page*. `http://us.metamath.org/mpegif/mmset.html`.

[5] Real AI, *About Real AI*. `http://realai.org/about/`.

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, *Mastering the game of Go with deep neural networks and tree search*, Nature, 529 (2016), pp. 484–489.

[7] TensorFlow. `http://tensorflow.org`.

[8] TFLearn, *TFLearn: Deep learning library featuring a higher-level API for TensorFlow*. `http://tflearn.org/`.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention Is All You Need*, ArXiv e-prints, (2017).

[10] D. P. Whalen, *Holophrasm: a neural automated theorem prover for higher-order logic*, 2016.

[11] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, ukasz Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, *Google's neural machine translation system: Bridging the gap between human and machine translation*, CoRR, abs/1609.08144 (2016).