On Differentiable Interpreters

Matko Bošnjak

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

Department of Computer Science
University College London

February 14, 2021

I, Matko Bošnjak, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

...Katji, Ines i Miri.

...i za ono jedno popodne kad smo sjeli i tipkali tati tezu

```
7m        m..            vtnnbvv c  vnnn                       vvvvvvvdgrtrrfree                  \\\\
 gch ffgfgghtrrreewwwwwwqqqq21qqqa c   xdchh    v x```xxyyu2



 fhmbb                                    xzaa x  zz,.
 ">>>

tyj
 z
```

Acknowledgements

Wow, what a trip this was! While trying to sum this trip up, remembering many people and countless things happening on the way, I am well aware I cannot do right to everyone who in any way shared this trip with me. So, to you who were a part of my journey, even if we just shared a random night of chatting at the Large Common room of Goodenough College or a random bar further away from a conference venue, thank you, as you inevitably helped me get to this point, in some way or another.

I was fortunate enough to have Sebastian Riedel as my supervisor who was an indispensable source of knowledge, inspiration, drive and a rather bottomless source of patience and good vibes. It was fun doing research with you, and I am indebted to you and your calm and utterly positive attitude, all the spot-on support and the unrelenting curiosity. I am thankful to John Shawe-Taylor and his forward-looking insights which helped me guide me in the first stages of my research, while I found many things still unclear. Ed Grefenstette came in at the right time to share his experience, knowledge and well-timed advice and suggestions shaping both the direction of the research and my outlook on industry-led research. I extend my gratitude to my examiners Charles Blundell and Charles Sutton for the remarkably enjoyable in-depth discussion and their keen eyes for details that inevitably improved the quality of this work.

It was inspiring to be a part of the UCL Machine Reading group (now re-branded as the UCL NLP group), not just because of the numerous fantastic researchers and collaborators like Andreas, Andres, Dirk, Guillaume, Isabelle, Jason, Jeff, Pasqale, Pontus, Tim and Thomas, but also much because of the fantastic PhD colleagues whose friendships made it all the more stimulating and lively. Thank you Georgios, Ivan, Johannes, Marzieh and Tim, we surely have had a blast, not just studying and doing research but also having an odd going-out and even a relaxing trip here or there.

I was fortunate enough to do not one but three industry internships during my studies. The first one in Microsoft Redmond with Chris, Hoifung, Kristina and Scott was an eye-opening experience that helped me shape my future research directions and enriched my social life through experiences shared with great many other remarkable PhD students. The second internship at Microsoft Cambridge with Alex, Dimitrios, Marc and Miltos was a unique and motivating research experience that helped me hone my skill set and expand my research interests by providing a fantastic experience to delve further into particular research direction I could have never been able to venture in-depth alone. The third internship at DeepMind with Alex, Arka, Chris, Irina, Loic and Nick was an experience that quickly locked me into the pursuit for AGI by empowering me to ask bigger and bolder questions and providing me with the raw intellectual and computational power to pursue them. This internship turned into a full-time position which actively energises and challenges me on a daily basis, enabling me to tap into the pool of exceptionally bright and interesting colleagues, of which I particularly thank Alex and Loic for their drive and continuous support, as well as Bojan, Ivan and Jovana for being a constant source of encouragement and respite during the sprint towards AGI.

A big thank you to the Goodenough College, the fantastic and energising oasis in the middle of London—a home away from home—where I have spent countless nights in both shallow and in-depth discussions with so many people I find it dizzying just trying to remember some. There I have met acquaintances, friends and friends for life, out of which I thank Adam, Alessandra, Ana, Andres, Ayah, Basia, Boris, Damjan, Duška, Fabio, Ifigeneia, João, Kanu, Kolapo, Laura, Lindsay, Luis, Luisa, Mark, Mary, Matt, Namu, Omar, Pablo, Pavithra, Rani, Rodrigo, Srđan, Stefan, Tariq, Tom, William and Wilhelm for all the highs and lows, for all the moments shared and more yet to come. To this group I add Ilia, Ivan, Ivana, Nikola and "Krkani", as well as my friends from Portugal that I keep stumbling upon everywhere I go. Thank you for all the parties, organisational meetings, random cooking nights and especially for the time spent outside the College on the curb, discussing both everything and nothing. Being away from home is easier with a good bunch of people, but being home with the irreplaceable is incomparable. Thank you Krešo, Martin and Vice for being irreplaceable.

Finally, my never-ending gratitude belongs to the pillars that raised me to

where I am today—my family. An eternal thank you to my mother Mira who uncompromisingly supported me in every single step I made, always pushing me to do the best, and make the best of everything. A monstrous thank you to Ines, my partner in life and my rock, who was with me the whole way, both far and near, both in good and the not so great. I sincerely hope I will be able to repay all your kindness in this lifetime. And lastly, thank you Katia, my little source of wonder and amazement, for showing me that there is a completely new universe there for the three of us where we can enjoy seeing you grow up and build ourselves and our futures as humbler and better people. Hvala vam što me svakodnevno nadahnjujete i činite boljom osobom; neizmjerno vas volim i obožavam!

Abstract

Neural networks have transformed the fields of Machine Learning and Artificial Intelligence with the ability to model complex features and behaviours from raw data. They quickly became instrumental models, achieving numerous state-of-the-art performances across many tasks and domains. Yet the successes of these models often rely on large amounts of data. When data is scarce, resourceful ways of using background knowledge often help. However, though different types of background knowledge can be used to bias the model, it is not clear how one can use algorithmic knowledge to that extent.

In this thesis, we present differentiable interpreters as an effective framework for utilising algorithmic background knowledge as architectural inductive biases of neural networks. By continuously approximating discrete elements of traditional program interpreters, we create differentiable interpreters that, due to the continuous nature of their execution, are amenable to optimisation with gradient descent methods. This enables us to write code mixed with parametric functions, where the code strongly biases the behaviour of the model while enabling the training of parameters and/or input representations from data.

We investigate two such differentiable interpreters and their use cases in this thesis. First, we present a detailed construction of $\partial 4$, a differentiable interpreter for the programming language Forth. We demonstrate the ability of $\partial 4$ to strongly bias neural models with incomplete programs of variable complexity while learning missing pieces of the program with parametrised neural networks. Such models can learn to solve tasks and strongly generalise to out-of-distribution data from small datasets. Second, we present greedy Neural Theorem Provers (gNTPs), a significant improvement of a differentiable Datalog interpreter NTP. gNTPs ameliorate the large computational cost of recursive differentiable interpretation, achieving drastic time and memory speedups while introducing soft reasoning over logic knowledge and natural language.

Impact Statement

This thesis presents differentiable interpreters as a framework for incorporating algorithmic background knowledge into neural networks. We demonstrate two interpreters, $\partial 4$ for incorporating arbitrary algorithmic knowledge, and gNTP, for incorporating scalable reasoning over knowledge bases and text.

By integrating algorithmic knowledge through programs into neural networks, from simple constructs, loops, conditionals, to libraries of well-known algorithms, differentiable interpreters bring strong generalisation of these programs to neural networks. We think the models we present in this thesis are a good testbed for understanding both how much prior algorithmic knowledge a model needs—i.e. how much structure is enough for a learning model to successfully generalise—and the extent of the generalisation programs can bring to neural models. Moreover, since strong generalisation is a well-sought property of neural networks with far-reaching consequences in the application of these models in the real world, our approach could find its use in cases where neural models are controlling critical long-term processes such as control tasks in industrial facilities. Besides, this approach or the elements thereof can benefit tasks involving numerical reasoning and inference such as automatic document understanding involving numbers in areas such as law and medicine.

By incorporating scalable reasoning into neural models, we can build models that scale to large, real-world datasets, and by adding the support for natural language, we open the application of these models on text. This is exciting as we have high hopes that gNTPs or gNTP-inspired, similarly efficient interpretable models, can be utilised for reasoning on large amounts of textual knowledge. Scalable reasoning on text could bring us closer to one of the holy grails of Machine Reading—automatic fact-checking at scale—which could have a tangible societal impact with a potential to transform the way we share and consume information in an information-centric world of today.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Machine Learning (ML) models that enable modelling complex behaviours from data have practically become the focal point of Artificial Intelligence (AI) and its pursuit to build machines that can (learn to) carry out tasks we consider "smart". The ML approach to task learning, epitomised through the *give machines data and let them learn for themselves* guiding principle, is also at the heart of Neural Network (NN) / Deep Learning (DL), models which swept the world of ML fairly recently [Krizhevsky et al., 2012]. It is widely considered that this breakthrough of NNs to the centre stage of ML was enabled by three major advancements: the availability of large datasets [Deng et al., 2009], advances in computing power [Raina et al., 2009, Cireşan et al., 2010] and better learning algorithms [Hinton et al., 2006]. Computing power is still growing, and is becoming increasingly available; learning algorithms are being constantly improved, yet NNs are still *data-hungry*—they mostly still require large datasets for achieving state-of-the-art results. However, in practice, training data is often scarce for all but a small set of problems, so the core question posed is how can we incorporate other information into the model, and enable the model to still learn well even with scarce training data?

As a contribution to answering that question, this thesis studies neural architectures which enable incorporating prior knowledge in a programmatic form. In general, in order to extrapolate beyond the data, a learner requires other sources of information to make up the difference [Tenenbaum et al., 2011]. Among these sources of information, we focus on prior knowledge regarding the task, also known as the *inductive bias*, and its role is to help restrain the space of models considered by the learner [Lake et al., 2017]. Reasonably sized NNs can realise but a fraction of all theoretically possible functions [Kol-

mogorov, 1957, Hornik et al., 1989, Siegelmann and Sontag, 1991], and we use inductive bias to constrain the subset of functions we can effectively learn. Since not all functions are equally useful for a task, we need to find ways to focus on the useful ones—often through the topology of the network.

In general, we can say that NNs have an inductive bias towards minima reachable by gradient-based optimisation.[1] However, the inductive bias depends directly on the model topology, activation functions it uses, and the training regime (including numerous "tricks" employed during training). Here we focus on the inductive bias brought forward by the topology, and to better understand what this means, we take a brief look back at the evolution of inductive biases through historical NNs architectures.

The very first computational model of neural activity is the *Threshold Logic Unit* by McCulloch and Pitts [1943].[2] This simple model sums the equal contribution of the input and activates the output only if the sum reaches a certain threshold. The inductive bias of these units is then characterised by this *all-or-none* process, where each input votes independently (without interactions) towards the final output. The *Perceptron* [Rosenblatt, 1958] relaxed the absolute threshold and the independent contribution of the input of the Threshold Logic unit through the use of per-unit biases and weights. This biases the Perceptron towards the ability to use all inputs to a different (learned) extent. Though the forerunner of today's deep learning, the Perceptron [Minsky and Papert, 1969] could not be learned beyond a single level. However, with the dawn of backpropagation [Linnainmaa, 1970, Rumelhart et al., 1986], deeper architectures, such as the Multi-Layer Perceptron (MLP) [Ivakhnenko and Lapa, 1966] came to light. The use of continuous activation functions biases MLPs towards smooth interpolations between inputs, while the multiple layers bias them towards presenting a more compact representation compared to shallower models [Delalleau and Bengio, 2011, Eldan and Shamir, 2016, Cohen et al., 2017]. Besides a more efficient function representation, the depth also brings significant increases in parameter numbers and training issues.

Convolutional Neural Networks (CNNs) [Fukushima, 1980, LeCun et al., 1989], based on neurophysiological insights of the neural cortex, attack the increase in

[1] The existence of minima unreachable by gradient-based optimisation is quite possible [Gaunt et al., 2016].

[2] This gross oversimplification of the operating principles of a biological neuron is the basis for the much-contested adjective "Neural".

parameter number with a receptive field of convolutional units applied across the image, effectively enabling drastic parameter reduction via parameter sharing. The inductive bias of these models is that the pooling geometry favours interleaved partitions of the image and enables locality and translation invariance, which, in turn, biases the models towards the statistics of natural images [Cohen and Shashua, 2017]. CNNs introduced the concept of sharing weights across function (convolution) applications. This was further brought forward to sequence processing by the Recurrent Neural Networks (RNNs) [Elman, 1990, Jordan, 1997]. These sequential models are temporally invariant models biased towards predicting future behaviour based on the recent past, with model variants such as Long Short-Term Memory (LSTM) further pushing that towards functions preserving contextual information over long sequences. Elaborate modern architectures push the inductive biases even further. Graph Neural Networks incorporate relational inductive biases [Gori et al., 2005, Battaglia et al., 2018]. Transformers [Vaswani et al., 2017] balance the bias towards attending both to the future and the past with the choice between positional variability and invariance, enabling them to avoid the recency bias of RNNs. More elaborate architectures such as Neural GPUs [Kaiser and Sutskever, 2016] and Neural Turing Machines [Graves et al., 2014], are biased towards algorithmic execution.

We notice that, even though there are architectures which enable inductive biases of algorithms, they are either hand-coded for a particular algorithm or they are too general—existing architectures do not support incorporating inductive biases of algorithms designed per task, at least not easily. That is to say, if we know a specific algorithm or the parts of it, we cannot easily influence the model's inductive bias with that knowledge. The emphasis on easy incorporation of knowledge is important here—theoretically, we can influence the inductive bias of a model by changing its weights, e.g. we could change the weights of a Neural Turing Machine to bias its execution with a particular program. However, that is practically impossible because the internal state transitions of such a model are opaque to interpretation, let alone to meaningful and reliable coding and debugging. This inspired us to investigate the problem of incorporating program code as inductive bias.

Why do we focus on the inductive bias of program code? There are several benefits of program code as an inductive bias for neural networks. Program code

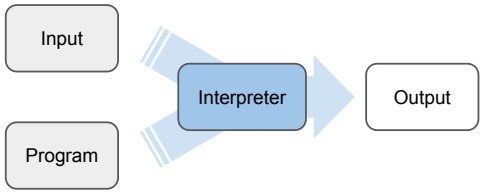


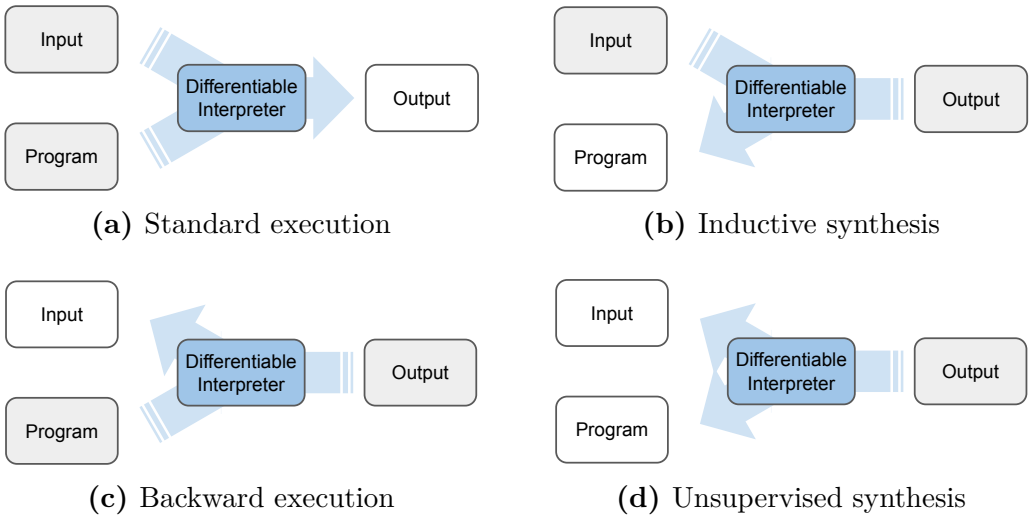
Figure 1.1: A depiction of program interpretation. Grey squares denote given (observed), while the white ones denote computed (inferred) data.

epitomises strong generalisation over input data—correct programs working on small inputs correctly generalise over larger inputs too. They can embody strong generalisation since they support useful constructs such as loops, conditionals, recursions, etc. Programs are recomposable (modular) and enable abstraction and reuse via libraries. Code is the language of computation—it makes it easy to reason about computation and communicate it between humans as well, as the code is both computer- and human-interpretable. We can see it as a useful prior over general computation, but we also need to be aware that though it is beneficial for certain kinds of computation, it is not fit for everything, in particular for ML tasks.

We search for a way to incorporate program code as inductive bias into NNs by turning to standard code execution and interpretation with interpreters. Simply put, an interpreter is a program that reads source code and directly executes it—it performs actions that the code specifies. Interpretation is a strictly one-way deterministic computation process which takes the input and executes the code to produce the output, as depicted in Figure 1.1. In order to bias a NN with program code, the NN should be able to execute said code similarly as an interpreter does. We take this approach of hybridising NNs with code as it enables us to train the model, following the structure of the code, end-to-end from algorithm input-output examples.

Differentiable Interpretation The core insight of this thesis is that the process of interpretation itself, as well as every single code command to be executed, can be continuously relaxed. Instead of running discrete program commands (e.g. variable assignment, memory reading and writing, loops, conditionals, recursion, etc. , we can run their continuously relaxed analogues and unlock the reverse mode of the interpretation process.

Differentiable Interpreters Standard interpreters execute source code written in a particular language by carrying out the actions that each instruction



(a) Standard execution **(b)** Inductive synthesis

(c) Backward execution **(d)** Unsupervised synthesis

Figure 1.2: Differentiable interpreter. Forward mode equals to a) standard (forward) execution (deduction). The reverse mode can correspond to solving different tasks, depending on what is inferred: b) inductive program synthesis (induction), c) backward execution (abduction), and d) unsupervised program synthesis (joint induction and abduction).

describes, according to the semantics of the instructions in that language. The nature of these instructions is discrete—instructions are carried out as discrete functions which operate on discrete states of memory. Differentiable interpreters share the same high-level idea—they execute a source code by carrying the actions of each instruction. However, the instructions here are carried out as continuous functions operating on continuous states of memory. The language of the source code stays the same here, but the semantics of its instructions change from discrete to continuous functions. The fact that these continuous functions are continuous relaxations of the original discrete functions makes these differentiable interpreters continuously relaxed interpreters of original interpreters.

By employing continuous relaxation of an interpreter, we still keep standard interpretation as a forward mode of a NN as we can still run the continuously relaxed commands as we did the discrete ones. However, by employing continuous functions, we can calculate gradients of every command with respect to their inputs and use these gradients to unlock the reverse mode—use gradients to back-propagate signals from the output to the input. Furthermore, if every single command of a program is continuously relaxed, by chaining commands, we can back-propagate error signals from the output of the program to its inputs. This also enables us to freely combine continuously relaxed commands

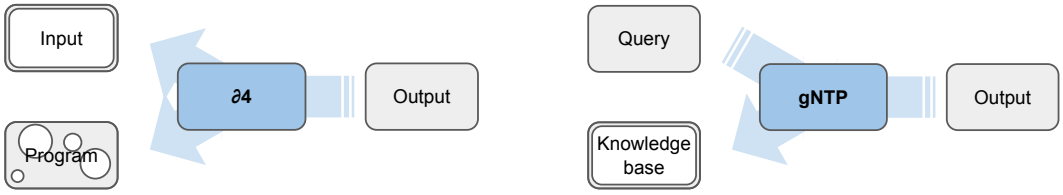
with differentiable parametrised models such as standalone NNs.

Now, besides the standard interpretation via the forward mode of differentiable interpreters (Figure 1.2a), where we deduce the output from the program and the input, we can use the reverse mode to solve interesting tasks that regular interpreters cannot, tasks which require specialised standalone solvers. By providing the input and the output examples, and treating the program as parameters of the model, we can use differentiable interpreters to induce the program (Figure 1.2b)—the inductive program synthesis task [Polozov and Gulwani, 2015]. If we provide the output and the program and treat the input as parameters of the model, we can abduce the input (Figure 1.2c)—the backward execution task [Dinges and Agha, 2014]. In the extreme, we can also envision providing only the output, and jointly inducing the program and abducing the input (Figure 1.2d)—the unsupervised program synthesis task [Ellis et al., 2015, Evans et al., 2019].

1.1 Contributions

Differentiable interpreters are an effective framework for utilising algorithmic background knowledge as architectural inductive biases of neural networks. To support this thesis, besides establishing the notion of differentiable interpretation concurrently with Gaunt et al. [2016], we put forward the following two core contributions:

Contribution 1: ∂4, A Differentiable Forth Interpreter To directly enable the translation of programs into inductive biases, we present ∂4, a differentiable interpreter for the imperative language Forth. We present a continuous relaxation of the Forth dual-stack abstract machine/interpreter, covering a continuously relaxed subset of ANSI Forth commands. This enables us to translate a Forth program into a NN that faithfully executes the program in its forward mode. To capitalise on the reverse mode, we introduce ∂4 sketches, incomplete programs consisting of fully-specified Forth code, which codes up what we know about the computation, and the parametrised NNs which code up the unknown part of the computation. The differentiability of the ∂4 interpreter allows us to not just use the known part of the computation as an inductive bias for a NN but also to train the unknown part and the program input representation via gradient-based optimisation (Figure 1.3a). We then use sketches of two well-known algorithms as a strong inductive bias on



(a) ∂4 **(b)** gNTP

Figure 1.3: Differentiable interpreters studied in this thesis. a) ∂4 can infer missing code in the sketch, as well as the input representations (thick border) and b) gNTP can infer the representation of the knowledge base elements

learning algorithms from data. As the differentiable interpretation is resource-intensive, we present program code optimisations to speed it up. Finally, since the ∂4 sketches are fully end-to-end differentiable, we can pair them up with other NNs. We demonstrate this ability by pairing up a ∂4 sketch with an LSTM model to obtain state-of-the-art for end-to-end reasoning on word algebra problems.

Contribution 2: gNTP, Greedy Neural Theorem Provers Differentiable interpreters exhibit scaling issues, which is particularly evident for the case of the NTP, a differentiable interpreter for the logic language Datalog. We suggest a strategy to significantly scale up NTP, and enable it to deal with compositional language input, thus scaling the inductive bias of differentiable reasoning algorithm and applying it to text-enriched data. We propose gNTP, a computationally effective model which reduces the time and the space load of NTP by drastically reducing the number of proof paths the model observes and lowering the number of parameters for the rule learning process with the attention mechanism. Following the efficiency enhancements, we supplement the model with a compositional reading module which embeds logical facts and natural language texts in the same vector space, enabling learning representations of both the logical and the textual input (Figure 1.3b). We show empirically that the gNTPs significantly outperform NTPs in terms of both time and memory efficiency while achieving the same or better performance on small link-prediction tasks. As a model that now scales, we also contrast gNTPs with related models on large link-prediction tasks showing competitive results. Finally, we qualitatively analyse both the rules induced by the gNTPs models, as well as the best-ranking proof paths that the model picks, suggesting cautiousness in model interpretation.

1.2 Publications

We have published parts of this thesis. Parts of Chapter 3 appeared in the following publications:

- Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a Differentiable Forth Interpreter. In *Proceedings of International Conference on Machine Learning (ICML)*, volume 70, pages 547–556, 2017
  - Also presented at the workshop track of the 5th International Conference on Learning Representations (ICLR), 2017
- Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. A Neural Forth Abstract Machine. In *Neural Abstract Machines & Program Induction (NAMPI) Workshop @ NIPS*, 2016
  - Also presented at the Symposium on Recurrent Neural Networks and Other Machines that Learn Algorithms @ NIPS 2016

Parts of Chapter 4 appeared in the following publications:

- Pasquale Minervini*, Matko Bošnjak*, Tim Rocktaschel, Sebastian Riedel, and Edward Grefenstette. Differentiable Reasoning on Large Knowledge Bases and Natural Language. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2020
  - A lightly extended version of the paper has also been published as an invited book chapter in Ilaria Tiddi, Freddy Lécué, and Pascal Hitzler. *Knowledge Graphs for eXplainable Artificial Intelligence: Foundations, Applications and Challenges*, volume 47 of *Studies on the Semantic Web*. IOS Press, 2020. ISBN 978-1-64368-080-4
- Matko Bošnjak*, Pasquale Minervini*, Andres Campero, Tim Rocktaschel, Edward Grefenstette, and Sebastian Riedel. Neural Theorem Proving on Natural Language . In *The International Conference on Probabilistic Programming*, 2018
- Pasquale Minervini*, Matko Bošnjak*, Tim Rocktäschel, and Sebastian Riedel. Towards Neural Theorem Proving at Scale. In *Neural Abstract Machines & Program Induction v2 (NAMPI_v2) @ ICML*, 2018

I contributed (1) parts of the ideas and the theoretical framework, (2) parts of

the implementation, (3) experimental design and execution of the small-dataset link prediction experiments, run-time evaluation experiments, and parts of the large-dataset link prediction experiments and analyses, (4) qualitative analyses, and (5) writing a third to a half of all the rejected and accepted papers.

1.3 Thesis Outline

Following this introductory chapter, in Chapter 2 we present the relevant preliminaries and technical background material necessary to follow the concepts, notation and the maths of neural networks, first-order logic and program interpretation, necessary for following the coming chapters. In Chapter 3 we introduce $\partial 4$, a differentiable FORTH interpreter—a differentiable construction of the FORTH dual-stack machine—and use it as a means of inducing a strong inductive bias of program code into a neural network. In Chapter 4 we introduce gNTP, a significant time and memory -efficient upgrade of the differentiable Datalog interpreter NTP, which we also expand with a compositional reading module to allow it to use textual information. Finally, in Chapter 5 we conclude the thesis with an outline of future work.

Chapter 2

Preliminaries / Background

The material in this thesis presupposes understanding of neural networks, interpretation and elements of first-order logic. In this chapter we provide the necessary background work for the later expounded models. This includes the concepts and the notation used throughout the thesis, as well as the models and algorithms.

2.1 Neural Networks

Neural Networks are the core ML models in this thesis. We start by defining basic concepts and terminology used in the neural network community [Goodfellow et al., 2016] as well as present the notation we use in the thesis.

2.1.1 Concepts and Notation

Basic Objects We denote *scalars*, i.e. single numbers, with a lowercase variable name typeset in italic, and type them, for example, $x \in \mathbb{R}$, or $t \in \mathbb{N}$.

We denote *vectors*, i.e. ordered arrays of scalars, with a lowercase variable typeset in bold, for example $\mathbf{x} \in \mathbb{R}^m$ indicates a real-valued vector of size m. We refer to a particular element of a vector with *indexing*, e.g. x_i denotes the i-th element of the vector $\mathbf{x}$. Throughout the thesis, mentions of continuous representations, symbol representations or embeddings refer to real-valued column-vectors, and we interpret them as points in an n-dimensional space.

We denote *matrices*, i.e. two-dimensional arrays of scalars, with an uppercase variable typeset in bold, for example, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a real-valued matrix of m rows and n columns.

Tensors, in general, are a generalisation of the objects above; they are d-dimensional arrays of numbers, denoted with an uppercase variable typeset in sans serif font, for example, $\mathsf{A} \in \mathbb{R}^{m \times n \times o}$ indicates a 3-dimensional real-valued tensor. Even though its definition encompasses the definition of scalars ($d = 0$), vectors ($d = 1$), and matrices ($d = 2$), later in the thesis we use the term tensor to denote tensors of $d > 2$.

A *differentiable function* is a mapping from a space of tensors to a space of tensors, such that there exists a derivative at each point in the function's domain, for example, $f \colon \mathbb{R}^{m \times n} \to \mathbb{R}^m$. We often use the function-call notation, presenting such functions as $f(\mathbf{X}) = \mathbf{y}$, with $\mathbf{X} \in \mathbb{R}^{m \times n}$ and $\mathbf{y} \in \mathbb{R}^m$.

Basic operations Throughout our exposition, we rely on standard operations, operators and frequently used functions on scalars, vectors, matrices and tensors. We denote a vector or a matrix *transpose* with $^\intercal$, for example, $\mathbf{x}^\intercal$. We use the general *tensor contraction* to express the summation of products of scalar components of tensors to pairs of indices bound to particular dimensions of tensors, for example, the *vector-matrix product* $(\mathbf{x}^\intercal \mathbf{M})_j = \sum_{i=1}^{I} x_i M_{ij}$, the *matrix-matrix product* $(\mathbf{MN})_{ik} = \sum_{j=1}^{J} M_{ij} N_{jk}$, and the *bilinear tensor product* $(\mathbf{x}^\intercal \mathsf{T} \mathbf{y})_j = \sum_{i=1}^{I} \sum_{j=1}^{J} x_i T_{ijk} y_j$. Moreover, we also use the Hadamard product[1], defined as $(\mathbf{X} \odot \mathbf{Y})_{ij} = (\mathbf{X})_{ij} (\mathbf{Y}_{ij})$, and the outer product of two vectors, $\mathbf{x} \otimes \mathbf{y} = \mathbf{x}\mathbf{y}^\intercal$.

By applying a function f to another function g, we produce a resulting *function composition* $h(x) = f(g(x))$, also denoted as $f(g(x)) = (f \circ g)(x)$. By $\nabla_{\mathbf{x}} f$ we denote the *gradient* (the multi-variable generalisation of the derivative) of a scalar-valued function $f \colon \mathbb{R}^m \to \mathbb{R}$ with respect to $\mathbf{x}$, defined as $(\nabla_{\mathbf{x}} f)_i = \frac{\partial f}{\partial x_i}$. The generalisation of the gradient for a vector-valued function $\mathbf{f} \colon \mathbb{R}^m \to \mathbb{R}^n$ is the *Jacobian* matrix $\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \in \mathbb{R}^{n \times m}$, a matrix of all first-order partial derivatives of $\mathbf{f}$, defined as $(\frac{\partial \mathbf{f}}{\partial \mathbf{x}})_{ij} = \frac{\partial f_i}{\partial y_j}$.

In addition, we use specialised functions, frequently used in the field, such as the *sigmoid* function $\sigma(\mathbf{x}) = \exp(\mathbf{x})/(\exp(\mathbf{x}) + 1)$, the *hyperbolic tangent* $tanh(\mathbf{x}) = (\exp(\mathbf{x}) - \exp(-\mathbf{x}))/(\exp(\mathbf{x}) + \exp(-\mathbf{x}))$ and the *softmax* function $\mathrm{softmax}(\mathbf{x}) = \exp(\mathbf{x})/(\sum_i \exp(\mathbf{x}))$. We often use softmax to ensure that a particular vector sums to one, i.e. $\sum \mathrm{softmax}(\mathbf{x}) = 1$, and $0 \leq \mathrm{softmax}(\mathbf{x})_i \leq 1, \forall \mathrm{softmax}(\mathbf{x})_i$.

[1] also known as the element-wise multiplication

2.1.2 Machine Learning Basics

Machine Learning concerns with models and programs able to learn without explicit programming [Samuel, 1959]. In lieu of explicitly defining learning [Mitchell, 1997], we exemplify it as the process of finding functions that approximate desired behaviours.

Concretely, in this thesis we study finding *parameters* θ of parametrisable differentiable functions $f_\theta \colon \mathcal{X} \to \mathcal{Y}$, such that these functions closely model a set of input-output points called the *training dataset*, $\mathfrak{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$, where $\mathbf{x}_i \in \mathcal{X}$ and $\mathbf{y}_i \in \mathcal{Y}$. Here, by "closely modelling", we imply a requirement that $f_\theta(\mathbf{x}_i) \approx \mathbf{y}_i$, quantifying the deviation of $f_\theta(\mathbf{x}_i)$ from $\mathbf{y}_i$ with a *loss function*, $\mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)$.

In an ideal case, we want to ensure this "close modelling" by optimising the expectation of the loss over the data-generating distribution p_{data}, also called the risk $R(\theta)$:

$$R(\theta) = \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim p_{data}}[\mathcal{L}(f_\theta(\mathbf{x}, \mathbf{y}))]\,. \tag{2.1}$$

However, this is intractable as we do not have access to the data-generating distribution p_{data}, but only to its samples—the training dataset $\mathfrak{D}$. This brings us to the notion of empirical risk, defined over the empirical distribution $\hat{p}_{\mathfrak{D}}$:

$$\begin{aligned} \hat{R}(\theta) &= \mathbb{E}_{(\mathbf{x},\mathbf{y})\sim \hat{p}_{\mathfrak{D}}}[\mathcal{L}(f_\theta(\mathbf{x}, \mathbf{y}))] \\ &= \frac{1}{|\mathfrak{D}|} \sum_{(\mathbf{x}_i,\mathbf{y}_i)\in\mathfrak{D}} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)\,. \end{aligned} \tag{2.2}$$

The goal of the learning procedure is to find the *optimal parameters* which minimise the empirical risk:

$$\begin{aligned} \theta^* &= \arg\min_\theta \hat{R}(\theta) \\ &= \arg\min_\theta \underbrace{\frac{1}{|\mathfrak{D}|} \sum_{(\mathbf{x}_i,\mathbf{y}_i)\in\mathfrak{D}} \mathcal{L}(f_\theta(\mathbf{x}_i), \mathbf{y}_i)}_{\mathcal{L}_\theta(\mathfrak{D})}\,, \end{aligned} \tag{2.3}$$

where $\mathcal{L}_\theta(\mathfrak{D})$ is the overloaded loss function defined on the whole training set, signifying the average of losses per each example from the training set. We refer to this process of optimisation as model *training*.

When $\mathcal{L}_\theta(\mathfrak{D})$ is differentiable with respect to θ, we can use computationally

cheap *gradient-based optimisation* methods, such as Gradient Descent [Cauchy, 1847] to optimise the loss by iteratively updating parameters θ:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \nabla_\theta \mathcal{L}_{\theta^{(t)}}(\mathfrak{D}), \tag{2.4}$$

where η is the *learning rate*, a *hyper-parameter* which determines the update step size, and ∇_θ is the differential operator with respect to θ, making $\nabla_\theta \mathcal{L}_{\theta^{(t)}(\mathfrak{D})}$ the gradient of the loss with respect to θ at $\theta = \theta^{(t)}$.

Since calculating the gradient of the loss on the whole dataset $\mathfrak{D}$ per Equation (2.4) is often slow and even computationally intractable for large datasets, we use variants of Gradient Descent, such as Stochastic Gradient Descent [Robbins and Monro, 1951] and Mini-Batch Gradient Descent, which update parameters on the gradient of the loss on randomly chosen subsets of the dataset at time t, $\mathcal{B}^{(t)} \subset \mathfrak{D}$:

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \eta \frac{1}{|\mathcal{B}^{(t)}|} \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{B}^{(t)}} \nabla_\theta \mathcal{L}(f_{\theta^{(t)}}(\mathbf{x}_i), \mathbf{y}_i). \tag{2.5}$$

The need to calculate this gradient many times throughout the training procedure requires an efficient way of calculation. Enter the back-propagation algorithm.

2.1.3 The Back-propagation Algorithm

To efficiently calculate the gradient of the loss with respect to its parameters and its inputs, we use the *back-propagation algorithm* [Linnainmaa, 1970, Werbos, 1982, Rumelhart et al., 1986].[2]

In its essence, the back-propagation algorithm is an efficient dynamic programming algorithm [Bellmann, 1957] for recursive application of the *chain rule of differentiation* [Leibniz, 1676, de l'Hôpital, 1696, Newton and Whiteside, 2008] on a composite function. The chain rule enables us to compute derivatives of composite functions. Given a composite function $z = f(g(\mathbf{x}))$, where $g(\mathbf{x}) = \mathbf{y}$, $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, and $z \in \mathbb{R}$ the chain rule decomposes the calculation of $\nabla_\mathbf{x} z$ as:

$$\nabla_\mathbf{x} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^\intercal \nabla_\mathbf{y} z, \tag{2.6}$$

[2] Who exactly invented the back-propagation algorithm is a source of surprising contention in the neural network community [Griewank, 2012, Schmidhuber, 2015].

where $\partial \mathbf{y}/\partial \mathbf{x}$ is the Jacobian matrix of g, and $\nabla_{\mathbf{y}} z$ is the gradient of z with respect to $\mathbf{y}$. The efficiency of this algorithm stems from the dynamic programming approach, which avoids recalculation of the same expressions.

Today's modern deep-learning libraries implement reverse-mode automatic differentiation [Linnainmaa, 1970, Baydin et al., 2017], as a generalisation of the back-propagation algorithm, and efficiently calculate the gradients by the computation graph construction [Abadi et al., 2015, Maclaurin et al., 2015b, Goodfellow et al., 2016], or by function transformation [Bradbury et al., 2018].

With back-propagation, we can compute gradients of arbitrarily complex functions, from compositions of simple functions to, as we see in this thesis, compositions of complex functions representing continuously relaxed programs.

2.1.4 Architectures and Mechanism

By now, we presented the task of learning neural networks as function approximators, the learning algorithm, and the efficient way to calculate gradients for the learning algorithm. Here we present the architectures, i.e. the structures, of neural networks used in this thesis.

Multi-Layer Perceptron (MLP) is the archetypal neural network architecture. It is based on so called *layers*, an affine transformation followed by a non-linear transformation:

$$h_\theta(\mathbf{x}) = \varphi(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{2.7}$$

where φ is an element-wise applied non-linear function, often called *activation function*, such as the sigmoid or the hyperbolic tangent, defined earlier, and $\mathbf{W}$ and $\mathbf{b}$ are the parameters of the layer $\theta = \{\mathbf{W}, \mathbf{b}\}$. An MLP is a fixed-length chain composition of such layers, applied on input $\mathbf{x}$:

$$\begin{aligned} \mathrm{MLP}_\theta(\mathbf{x}) &= (h_{\theta_n} \circ \ldots \circ h_{\theta_2} \circ h_{\theta_1})(\mathbf{x}) \\ &= h_{\theta_n}(\ldots(h_{\theta_2}(h_{\theta_1}(\mathbf{x})))) = \mathbf{y}, \end{aligned} \tag{2.8}$$

where the parameters θ of the MLP are the set of all parameters of each layer, i.e. $\theta = \bigcup_{i=1}^{n} \theta_i = \{\mathbf{W}_i, \mathbf{b}_i \mid i \in \{1, \ldots, n\}\}$ and the number of functions composed is called the depth of the MLP. The activation function of the final layer h_n is often chosen based on the task the MLP is trained on, and in the case of

classification, it is often the softmax function.

Since MLPs are a chain composition of layers, they are also called feed-forward networks, as the (transformed) input is fed forward through the network.

Recurrent Neural Network (RNN) [Elman, 1990] is a neural network architecture fit for processing sequential inputs, based on the following recurrence:

$$\mathbf{h}^{(t)} = f_\theta(\mathbf{h}^{(t-1)}), \tag{2.9}$$

where $\mathbf{h}^{(t)}$ is the *hidden state* of the model at time t. This is essentially a no-input dynamical system, which we can rewrite as a chain composition of the recurrence:

$$\begin{aligned} \text{eRNN}_\theta(\mathbf{h}^{(0)}, t) &= (f_\theta \circ \ldots \circ f_\theta \circ f_\theta)(\mathbf{h}^{(0)}) \\ &= f_\theta(...(f_\theta(f_\theta(\mathbf{h}^{(0)})))) = \mathbf{h}^{(t)}. \end{aligned} \tag{2.10}$$

We call this the execution RNN, though it technically is a precursor to the RNN. Contrasting the MLP and the execution RNN, we see that the execution RNN is characterised by a variable number of applications of the same recurrence, as opposed to the application of a fixed number of different layers in the MLP. The standard RNN, on the other hand, is a modification of the execution RNN which accepts a variable-length input $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots \mathbf{x}^{(t)})$, and returns a variable-length output $(\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots \mathbf{y}^{(t)})$:

$$\begin{aligned} \mathbf{h}^{(t)} &= f_\theta(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}) \\ \mathbf{y}^{(t)} &= g_\theta(\mathbf{h}^{(t)}). \end{aligned} \tag{2.11}$$

A concrete example of an often used RNN, dubbed the Vanilla RNN is:

$$\begin{aligned} \mathbf{h}^{(t)} &= \tanh\left(\mathbf{W}\left[\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}\right] + \mathbf{b}\right) \\ \mathbf{y}^{(t)} &= \sigma\left(\mathbf{W}_{out}\mathbf{h}^{(t)}\right), \end{aligned} \tag{2.12}$$

where $[\mathbf{x}; \mathbf{y}] = \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}$ is the column-vector concatenation.

Long Short-Term Memory (LSTM) is an improvement upon RNN. The hidden state $\mathbf{h}^{(t)}$ of the RNN functions as a memory element of the model, enabling it to process variable-length inputs, but it brings forward the issues of exploding and vanishing gradients, leading to problems with long-term depen-

dencies [Bengio et al., 1994, Pascanu et al., 2013]. Hochreiter and Schmidhuber [1997] formulated the LSTM, a special type of RNN that provides solutions to these problems, based on the recurrent application of the LSTM cell defined by:

$$\begin{aligned}
\mathbf{i}^{(t)} &= \sigma(\mathbf{W}_i \left[\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}\right] + \mathbf{b}_i) \\
\mathbf{f}^{(t)} &= \sigma(\mathbf{W}_f [\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}] + \mathbf{b}_f) \\
\mathbf{o}^{(t)} &= \sigma(\mathbf{W}_o [\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}] + \mathbf{b}_o) \\
\mathbf{g}^{(t)} &= \tanh(\mathbf{W}_g [\mathbf{h}^{(t-1)}; \mathbf{x}^{(t)}] + \mathbf{b}_g) \\
\mathbf{c}^{(t)} &= \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \mathbf{g}^{(t)} \\
\mathbf{h}^{(t)} &= \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)}) .
\end{aligned} \tag{2.13}$$

For more details on LSTMs and their inner workings, see Olah [2015].

Bidirectional RNNs (BiRNNs) / Bidirectional LSTMs (BiLSTMs) [Schuster and Paliwal, 1997, Graves et al., 2005] are RNNs that fuse outputs of two independent RNNs from Equation (2.12) running from opposite directions of the input:

$$\begin{aligned}
\overrightarrow{\mathbf{h}}^{(t)} &= \tanh\left(\overrightarrow{\mathbf{W}}\left[\overrightarrow{\mathbf{h}}^{(t-1)}; \overrightarrow{\mathbf{x}}^{(t)}\right]\right) \\
\overrightarrow{\mathbf{y}^{(t)}} &= \sigma\left(\overrightarrow{\mathbf{W}_{out}}\,\overrightarrow{\mathbf{h}}^{(t)}\right) \\
\overleftarrow{\mathbf{h}}^{(t)} &= \tanh\left(\overleftarrow{\mathbf{W}}\left[\overleftarrow{\mathbf{h}}^{(t-1)}; \overleftarrow{\mathbf{x}}^{(t)}\right]\right) \\
\overleftarrow{\mathbf{y}^{(t)}} &= \sigma\left(\overleftarrow{\mathbf{W}_{out}}\,\overleftarrow{\mathbf{h}}^{(t)}\right) \\
\mathbf{y}^{(t)} &= \left[\overrightarrow{\mathbf{y}^{(t)}}; \overleftarrow{\mathbf{y}^{(t)}}\right] .
\end{aligned} \tag{2.14}$$

The concatenation of outputs from opposite directions of the same input enables the output layer to utilise information from both the forward and the backward states at the same time which helps to mitigate the long-term dependency issue. Analogously, BiLSTMs follow the same logic, fusing the output of two independent LSTM cells in opposite direction.

MLP, RNN, LSTM, and BiRNN are depicted in Figure 2.1.

Attention mechanism in neural networks enables models to focus on a particular subset of the data. To focus on the k-th element of a vector $\mathbf{x}$, we could index the vector x_k with a one-hot weight vector, $\mathbf{1}_k$. However, this type of

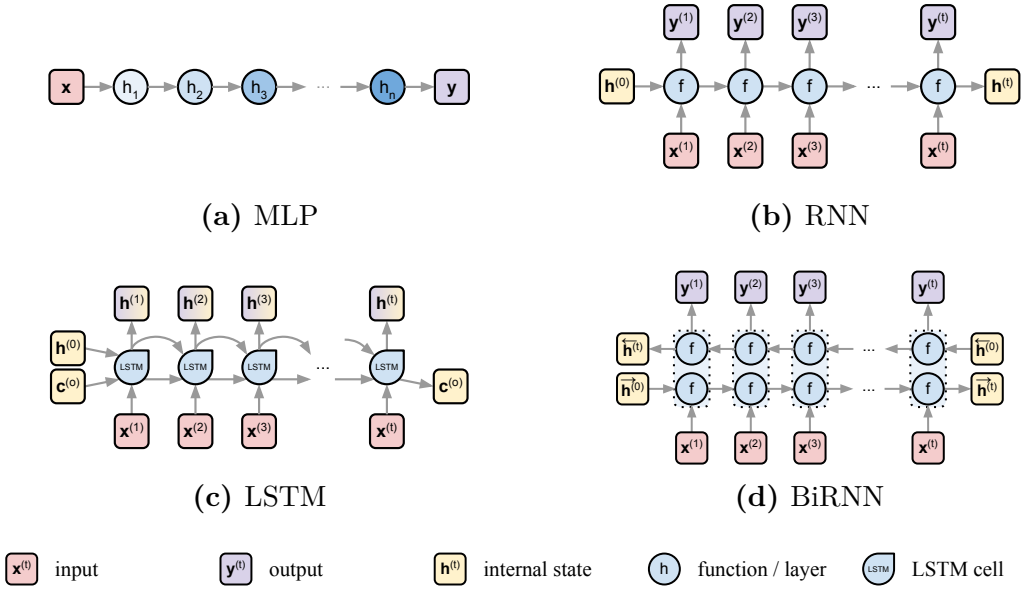


Figure 2.1: A depiction of a) MLP, b) RNN, c) LSTM, and d) BiRNN. BiLSTM is a BiRNN with LSTM cells.

interaction is not differentiable and is difficult to train [Weston et al., 2015, Luong et al., 2015]. To make it differentiable, we focus on the whole vector to different extents, via a convex combination of the elements of the vector, i.e. weighing $\mathbf{x}$ with a vector of weights such that $\sum \mathbf{w} = 1$, and $0 \leq w_i \leq 1, \forall w_i$. This condition is enforced with the softmax function, and is called *soft attention*:

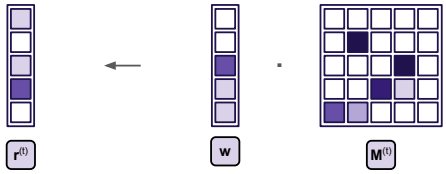
$$\mathbf{y}^\intercal = \text{softmax}(\mathbf{s})^\intercal \mathbf{x}, \tag{2.15}$$

where $\mathbf{s}$ is a vector of arbitrary values called scores which can be calculated in a plethora of ways [Bahdanau et al., 2015, Graves et al., 2014, Luong et al., 2015]. For more details see Olah and Carter [2016].

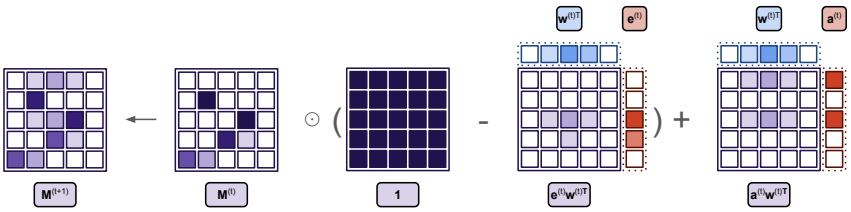
Differentiable Memory is a continuous approximation of discrete memory. The hidden state of RNNs serves as a memory element of the model, enabling better dealing with long-term dependencies. However, the hidden state is an internal memory resource of limited capabilities. Graves et al. [2014] coupled an RNN with a differentiable external memory accessed with an attentional mechanism, to enable the RNN to learn to read and write representations in the memory.

Given a memory bank $\mathbf{M} \in \mathbb{R}^{m \times n}$ representing m n-dimensional row-vector

[2]Each element in the column-vector multiplies the appropriate row of the matrix, and the result is the sum of such weighted rows.



(a) Differentiable Reading / Attention. Note the transposition of w and r in Equation (2.16), but we depict it as a column-vector for compactness and easier graphical understanding.[3]



(b) Differentiable Writing

Figure 2.2: A depiction of a) differentiable reading / attention and b) differentiable writing.

values, attention enables differentiable reading and writing into said memory. *Reading* is realised as a convex combination of all the row-vectors of the memory bank (soft-attention):

$$\mathbf{r}^{(t)\intercal} \leftarrow \mathbf{w}^{(t)\intercal}\mathbf{M}^{(t)}\,, \tag{2.16}$$

with the weight vector $\sum \mathbf{w}^{(t)} = 1$, and $0 \leq w_i^{(t)} \leq 1, \forall i$.

Writing, on the other hand, is a bit more involved:

$$\mathbf{M}^{(t)} \leftarrow \mathbf{M}^{(t-1)} \odot (\mathbf{1} \otimes \mathbf{1} \underbrace{- \mathbf{w}^{(t)} \otimes \mathbf{e}^{(t)}}_{erasure}) + \underbrace{\mathbf{w}^{(t)} \otimes \mathbf{a}^{(t)}}_{addition}, \tag{2.17}$$

where $\mathbf{e}_t$ is the erase vector (i.e. the vector to be erased from the memory), $\mathbf{a}_t$ is the add vector (i.e. the vector to be added to the memory), and $\mathbf{1}$ a vector of ones.

Given that both the erasure and the addition are differentiable, the whole writing operation is differentiable. Figure 2.2 depicts the attention/reading and writing mechanisms presented.

2.2 The Language of First Order Logic

Chapter 4 focuses on a differentiable interpreter for a logic language Datalog, hence we present an excerpt of Datalog notation and the first-order logic framework. We start with the concepts and notations and move towards the backward chaining algorithm, which is at the core of the differentiable interpreter NTP.

2.2.1 Concepts and Notation

Throughout the thesis, we assume standard concepts and terminology from the First-Order Logic (FOL) and logic programming [Russell and Norvig, 2009], yet our notation differs slightly since we borrow it from DATALOG. We briefly define the standard concepts and the terminology as well as notation used.

Symbols We denote *constant* symbols in lowercase letters, like A, B, LONDON, ..., and *variable* symbols in uppercase letters, such as X, Y, Z, ... and typeset both in SMALL CAPS. We denote *predicate*[4] symbols in lower camel case `p`, `q`, `locatedIn`, ... and typeset them in a `typewriter font`. In this work, we consider only the domain of function-free FOL, hence we do not admit the concept of a function.

Expressions Given the domain of function-free FOL, we consider a *term* t to be only a simple term—consisting of a constant or a variable—and typeset terms in the default serif font. An *atom* a is an element of the form $\texttt{p}(t_1, t_2, ..., t_n)$, where `p` is a predicate, and each t_i is a term. The number n in the previous atom is called the *arity* of the predicate `p`, e.g. `locatedIn(X,Y)` is a binary predicate. Without loss of generality, we consider only binary predicates in this work. An atom is *ground* if there are no variables in the atom, e.g. `locatedIn(`LONDON`,`UK`)`.

A *definite clause* is an expression formed of atoms in the form of $a_1 \wedge a_2 \wedge ... \wedge a_n \rightarrow a_{n+1}$, where all a_i are atoms, $\wedge$ is the logical *and* operator, and $\rightarrow$

[4] We also call predicates *rules* and use these two terms interchangeably.

the logical *implication* operator. In this work, we consider all variables in all expressions being universally quantified.

Substitution A *substitution* $\psi = \{X_1/t_1, X_2/t_2 ...\}$ is a mapping from a variable to a term, defined as a set of variable/term pairs. The application of the substitution ψ on an atom a is denoted as ψa, and results in a new atom with variables in the atom substituted by their appropriate terms in ψ. For example, given a substitution $\psi = \{X/\text{LONDON}, Y/Z\}$ and an atom $a = \texttt{locatedIn(X,Y)}$, the result of the substitution application is $\psi a = \texttt{locatedIn(LONDON,Z)}$.

Logic Programming Definite clauses are often written in the reverse form $a_{n+1} \leftarrow a_1 \wedge a_2 \wedge ... \wedge a_n$ in Logic Programming. In DATALOG the (reversed) implication symbol is replaced by :– and the $\wedge$ operator is replaced by a comma, like this a_{n+1} :– $a_1, a_2, ..., a_n$. We call this a DATALOG *clause*. The atom a_{n+1}, left from :–, is called the *head*, whereas the optional, comma-separated list of atoms on its right is called the *body*.

We call a clause without a body, such as a_{n+1} :– {} a *fact* $F \in \mathcal{F}$, and omit the body in writing, e.g. a_{n+1} and denote a set of facts as $\mathcal{F}$. A clause with a body, such as a_{n+1} :– a_1, a_2, a_3, is called a *rule* R, and we denote a set of rules with $\mathcal{R}$. Finally, a logic program, also called a Knowledge Base (KB), is a set of DATALOG clauses—facts and rules—and we denote it as $\mathfrak{K}$.

Lists Finally, we define a *list* as an object consisting of either an empty list [] or a nested ordered pair of an element, called the *head*, and a list, called the *tail* written as $L = [head : tail]$. In addition to the head/tail construction, we also use a modified set-builder notation to define lists, with braces replaced by square brackets, e.g. $L = [l \mid l \in [1,2,3]]$. In general, we use lists instead of sets to denote their importance in practical implementations.

Moreover, we use lists to represent atoms and rules. We write atoms as lists, e.g. $\texttt{locatedIn(LONDON,Y)}$ as $[\texttt{locatedIn}, \text{LONDON}, Y]$, and rules as lists of lists, e.g. $\texttt{locatedIn(X,Y)}$:– $\texttt{locatedIn(X,Z)}, \texttt{locatedIn(Z,Y)}$ as $[[\texttt{locatedIn}, X, Y], [\texttt{locatedIn}, X, Z], [\texttt{locatedIn}, Z, Y]]$.

2.2.2 Backward Chaining

Having knowledge represented as a FOL Knowledge Base (KB) opens the path to deriving new knowledge via inference algorithms. Two such algorithms,

both based on modus ponens [Suppes, 1999], are used in logic programming: forward chaining and backward chaining. In its essence, in *forward chaining*, rules whose bodies are satisfied repeatedly expand the KB with the head of the rule, whereas *backward chaining* works in the opposite direction, finding knowledge which supports a goal.[5]

The backward chaining algorithm can be expressed in an imperative style as in Russell and Norvig [2009][6] or in a functional style as in Rocktaschel and Riedel [2017]. In this work, we adopt the abbreviated functional style from the latter, presented via piecewise-defined functions. We define the intermediate state of the interpretation directly with the substitution set ψ, and omit the exact proof path trail from the formulation as an engineering detail. With `FAIL` we denote a special empty state of unification mismatch; once `FAIL` is reached, no further proof derivations are considered. The goal G we want to prove is formally a term, but in this work we focus on goals as grounded terms only.

The backward chaining algorithm uses Depth-First Search (DFS), executed by mutually recursive `or` and `and` functions to explore the space of all possible proofs while using the `unify` function to ignore paths with incompatible logical expressions. The algorithm ends either with a unification failure, a valid substitution, or a partial result if a prespecified search depth has been reached.

OR The `or` function[7] operates by fetching all clauses $H :- B \in \mathfrak{K}$ (rules and facts) that might unify with the goal G. It then unifies the head H of each clause with the goal G, and calls the `and` function on the resulting substitution and the body B of each of these clauses:

$$\texttt{or}_{\mathfrak{K}}(G, S) = \left[S' \;\middle|\; \begin{array}{l} H \;:-\; B \in \mathfrak{K} \\ S' \in \texttt{and}_{\mathfrak{K}}(B, \texttt{unify}(H, G, S)) \end{array} \right]. \tag{2.18}$$

The `or` function is the starting point for the proving process, starting with a goal G we aim to prove and the starting state being the empty unification set, $\texttt{or}_{\mathfrak{K}}(G, \emptyset)$. We omit the details of variable standardisation and cycle detection [Russell and Norvig, 2009, Van Gelder, 1987] in the interest of brevity.

[5] We use the terms goal and query interchangeably.

[6] For an easy-to-follow implementation in Python, see https://github.com/aimacode/aima-python/blob/master/logic.py#L1438

[7] The query can be proved by any one of the clauses in the KB, hence the name `or`

```
districtIn(bloomsbury, london).
capitalOf(london, uk).
locatedIn(X, Y) :- districtIn(X, Z), capitalOf(Z, Y).
```

Listing 2.1: A running example of a small Datalog knowledge base. The knowledge base represents relationships between Bloomsbury, London and the UK.

AND The and function[8] is invoked on the body of a rule, and in turn, it must prove each atom of the body, while keeping track of the accumulated substitutions. It simply returns the substitution state S if the body is empty, otherwise, it applies the substitution on the subgoal ψg, and calls or on the resulting expression:

$$\mathsf{and}_{\mathfrak{K}}(G,S) = \begin{cases} \mathtt{FAIL} & \text{if } S = \mathtt{FAIL} \\ S & \text{if } G = [] \\ \left[S'' \;\middle|\; \begin{array}{l} S' \in \mathsf{or}_{\mathfrak{K}}(\psi g, S) \\ S'' \in \mathsf{and}_{\mathfrak{K}}(G', S') \end{array} \right] & \text{otherwise } G = [g : G'] \end{cases} . \tag{2.19}$$

Note that here the substitution is equal to the substitution state $\psi = S$, but we separate them here as later we will expand the substitution state to contain the substitution and its score. These two functions lay grounds for the exhaustive search underlying backward chaining, and they rely on the unification for ignoring incorrect proofs.

UNIFY The unify function, used earlier in or, calculates the correspondence of two logical expressions by iterating through pairs of symbols in the two expressions that we want to unify. It fails if the atoms are of different arity:

$$\mathsf{unify}(H,G,S) = \begin{cases} \mathtt{FAIL} & \text{if } S = \mathtt{FAIL} \\ S & \text{if } H = G = [] \\ \mathtt{FAIL} & \text{if } H = [] \vee G = [] \\ \mathsf{unify}(H', G', \mathtt{unify\text{-}var}(h, g, S)) & \text{otherwise } \begin{array}{l} H = [h : H'], \\ G = [g : G'] \end{array} \end{cases} \tag{2.20}$$

[8] The name and stems from the necessity to prove every atom in the body of the clause.

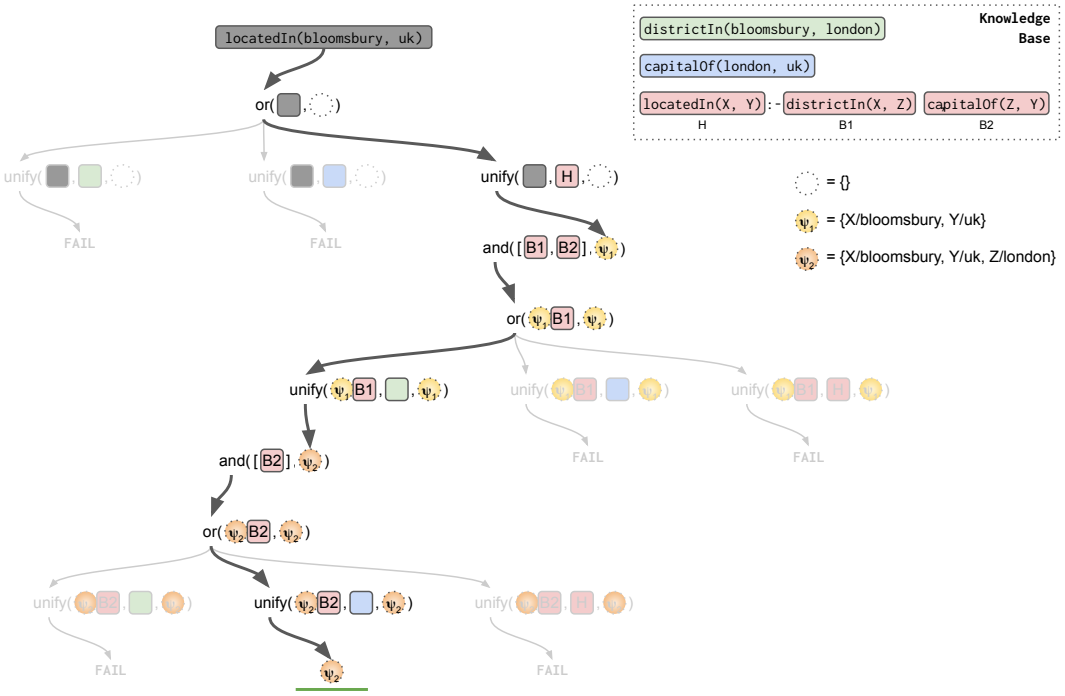


Figure 2.3: An example of the backward chaining execution on a small knowledge base. The knowledge base, presented in Listing 2.1, is also noted in the top right corner. Circles signify substitutions and squares signify atoms (with circles next to squares signifying applying a substitution to an atom). The colour coding of the KB and unifications follow through the example. We omitted some calls to `and` for clarity. Unsuccessful proof paths are greyed out to emphasise the single correct/final proof path in this example.

`unify` uses a helper function `unify-var` which updates the substitution set in case one of the two compared symbols is a variable, otherwise returning failure if two non-variable symbols are not identical:

$$\texttt{unify-var}(h, g, S) = \begin{cases} S \cup \{h/g\} & \text{if } h \in V \\ S \cup \{g/h\} & \text{if } g \in V, h \notin V \\ S & \text{if } g = h \\ \texttt{FAIL} & \text{otherwise} \end{cases}. \tag{2.21}$$

Example Throughout the chapter, we use a small KB example and its derivatives, outlining relationships between the United Kingdom (UK), London and its district Bloomsbury, presented in Listing 2.1.

We depict the full trace of the backward chaining procedure, given the example KB and the goal `locatedIn(`BLOOMSBURY`,`UK`)` in Figure 2.3. The figure

shows a full proof tree for a goal, with all the calls to `unify`, `or` and `and` functions, as well as the states of the substitutions ψ. Note many branches ending in unification failure due to symbol incompatibility.

2.3 Interpretation

Program interpretation / execution is one of the computational processes we aim to continuously approximate in this thesis. We bypass the fundamentals of source code interpretation and compilation [Aho et al., 1986, Mak and Copeland, 1996] to focus on the core concepts and present a simple function notation.

2.3.1 Concepts and Notation

States, instructions, and programs We define a *program state* S as a tuple of values or contents of particular computer resources, such as memory, registers, etc., that uniquely describe the informational makeup of a computer or a program. We assign it a subscript like $S^{(t)}$ to signify a state at a particular time t, and denote the set of all program states with $\mathcal{S}$.

An *instruction* ι is a basic syntactic unit of a programming language. All instructions of a programming language belong to its instruction set, $\iota \in \mathfrak{I}$. In an imperative language, instructions define the units of computational actions, whereas in a declarative language they define the logic of computation. For example, `INC reg` is an x86 instruction [Ferrari et al., 2006] that denotes an increment of the value of the register `reg` it is applied to by 1.

To specify the *semantics* of an instruction, we use the emphatic bracket notation from denotational semantics [Scott and Strachey, 1971, Tennent, 1976]. For example, the semantics of the instruction `INC reg`, where `reg` is an identifier for a register, is the following:

$$[\![\, \texttt{INC reg} \,]\!] \quad \stackrel{\text{def}}{=} \quad \texttt{reg} \leftarrow [\![\, \texttt{reg} \,]\!] + 1\,. \tag{2.22}$$

This reads that the meaning of the $[\![\, \texttt{INC reg} \,]\!]$ instruction is a function that takes the value of the register, $[\![\, \texttt{reg} \,]\!]$, adds one to it and stores it back into the register `reg`. Note here that we implicitly assume the instruction operating on the state S, where `reg` is an element of the state, and that we use the emphatic brackets to both denote the meaning of an instruction $[\![\, \texttt{INC reg} \,]\!]$

and the value of an element of the state $[\![\underline{\texttt{reg}}]\!]$. We see that the semantics of the instruction is a transition function, from one program state to another:

$$[\![\iota]\!] : S \to S \,. \tag{2.23}$$

Admittedly, we do not adhere to the rigour of denotational semantics, as our goal in this work is not to fully formalise the meaning of languages used in this thesis but to present a light and meaningful way to represent the semantics of instructions.

Next, we define a *program* P, from the set of all programs $P \in \mathcal{P}$, as a sequence of instructions:

$$P = (\iota_1, \iota_2, \ldots, \iota_n) \,, \tag{2.24}$$

thus $P \in \mathfrak{I}^*$. In an imperative language, a sequence of instructions details the steps of computation that the program defines, relying on the notion of compositionality:

$$[\![(\iota_1, \iota_2)]\!] = [\![\iota_2]\!] \circ [\![\iota_1]\!] \,. \tag{2.25}$$

On the other hand, in a declarative language, a weaker notion of compositionality applies because in a declarative language the order of instructions does not matter, and a program forms a set of instructions that detail the state of the problem domain, leaving the details of execution to the interpreter.

Interpretation In general, we define an *interpreter* Υ per Figure 1.1 as a function based on the instruction set $\mathfrak{I}$ that, given an input $I \in \mathcal{I}$ and a program $P \in \mathfrak{I}^*$, produces an output $O \in \mathcal{O}$:

$$\Upsilon_{\mathfrak{I}} \colon \mathcal{P} \times \mathcal{I} \to \mathcal{O} \,. \tag{2.26}$$

Treating the input I and the output O as elements of the state, i.e. $I, O \in S$, makes the interpreter:

$$\Upsilon_{\mathfrak{I}} \colon P \times \mathcal{S} \to \mathcal{S} \,. \tag{2.27}$$

Note that albeit the terms abstract machine and interpreter technically differ from one another, in the thesis we use them interchangeably.

2.3.2 Logic Program Interpretation

In Equation (2.26), we defined an interpreter as a function that takes a program and an input and produces an output. Similarly, we can formulate a logic program interpreter Υ_{lp}. In a logic program interpreter, the program P represents the state of the problem domain and is called the knowledge base $\mathfrak{K}$, the goal atom $G \in \mathcal{A}$ is the input, and the output is a set of states $\mathcal{S}$, consisting of either valid proof states or the indication of proving failure:

$$\Upsilon_{\text{lp}} : \mathfrak{K} \times \mathcal{A} \to \mathcal{S} . \tag{2.28}$$

One key difference to an imperative language like FORTH is in the control flow. In FORTH the control flow is explicit and fully specified by the program. In contrast, in logic program interpretation the control flow is pre-defined and consists of a specific theorem-proving strategy—in our case, backward chaining—and the knowledge base $\mathfrak{K}$ specifies the structure of the domain on which the theorem-proving strategy operates. In our case, backward chaining as a theorem-proving strategy is supported by the `or`, `and`, `unify` and the supporting functions, expressing the interpreter as:

$$\Upsilon_{\text{lp}}(\mathfrak{K}, G) = \texttt{or}_{\mathfrak{K}}(G, \emptyset) . \tag{2.29}$$

This reflects the declarative nature of logic programming—the behaviour of the proving procedure of the logic program is not under the control of the programmer.[9]

[9] Though the control is not under the influence of the programmer, there are still many optimisations, engineering details and shortcuts which can be made in concrete implementations of logic program interpreters [Warren, 1983].

Chapter 3

∂4: A Differentiable Forth Interpreter

The overarching goal of AI is the development of machines and algorithms able to effectively master complex behaviours from real-world data. A notable step in this direction is the recent advancement of continuous neural architectures, akin to traditional computers [Graves et al., 2014, Kurach et al., 2016], able to learn algorithms from data. The end-to-end differentiability of these architectures permits learning by gradient-based methods, enabling them to learn these complex behaviours from program traces [Reed and De Freitas, 2016, Mirman et al., 2018] and input-output pairs [Graves et al., 2014, Zaremba and Sutskever, 2015]. However, these models regularly fail to generalise outside of the training distribution and still stay data-intensive for often difficult-to-obtain data [Marcus, 2018, Chollet, 2019].

In Chapter 1 we established that programs can effectively present complex behaviours in a compact representation and enable arbitrary generalisation by design. Enabling structured execution, abstraction and compositionality, fully specified programs could provide strong inductive biases to learning algorithms able to utilise them. The question arising is how can learning algorithms exploit programs as a representation of knowledge, and utilise them as a strong inductive bias for achieving generalisation? We posit the answer lies in differentiable interpreters.

3.1 Programs as Inductive Biases

To address the question of utilising programs as a strong inductive bias in learning algorithms, we turn to differentiable interpreters. They, in turn, enable us to not just adhere to the strong bias of the full program, but also to specify an incomplete program—a *sketch* [Solar-Lezama et al., 2006, Solar-Lezama and Bodik, 2008]—in a traditional programming language, consisting of both fully specified (known) and incomplete (unknown) parts. The known part provides the strong inductive bias for the neural network, whereas the unknown part, which the programmer does not how to appropriately define, is learned from the data.

As explained in Chapter 1, the core insight behind differentiable interpretation relies on the fact that most programming languages can be formulated in terms of an abstract machine or an interpreter that executes commands of the language. We can then implement these machines as neural networks via continuous relaxation of the discrete interpretation. This makes part of the model follow the sketched behaviour, providing the strong inductive bias consistent with the program, and the unknown parts are optimised with respect to the training data.

The host language in this work is FORTH, a simple yet powerful Turing complete stack-based language. We chose FORTH for the following reasons:

1. it is an established, general-purpose high-level language relatively close to machine code [Brodie and FORTH Inc, 1987][1].

2. it promotes highly modular programs through the use of branching, loops and function calls, thus bringing out a good balance between assembly and higher-level languages [Brodie, 2004].

3. its abstract machine is simple enough [Koopman, 1993] for straightforward creation of its continuous approximation.

4. finally, we wanted to strike a good balance between flexibility and simplicity; Turing machines are simple yet too low-level and too abstract, whereas higher-level language such as Python provides greater flexibility, but its interpreter is far too complex for implementation.

[1] FORTH also enables self-modification, a property of interest in the AI research community [Schmidhuber, 2004, 2009].

Underlying FORTH's semantics is a simple dual-stack abstract machine. We introduce $\partial 4$, a continuous relaxation of this machine, differentiable with respect to the transition it executes at each time step, as well as to its distributed input representations. $\partial 4$ enables writing sketches—underspecified programs that partially define machine behaviour. The sketches, albeit conceptually similar to sketches in program synthesis [Solar-Lezama et al., 2006] and to probabilistic programs [van de Meent et al., 2018], have the added benefit of being trained through backpropagation which makes them easy to integrate with any other neural model.

We pose the following research questions:

- Can we use sketching with $\partial 4$ to capture arbitrarily complex inductive biases, including conditionals, loops, functions and recursion, on both discrete and continuous data?
- Does $\partial 4$ enable us to achieve strong generalisation from a small number of input-output data?
- Can $\partial 4$ sketches be jointly trained with other neural models?

We answer these three questions by showing that:

- given input-output pairs, a $\partial 4$ sketch can learn to fill in the missing parts of the sketch and generalise well to problems of unseen size on the sorting and adding neural programming tasks introduced in Reed and De Freitas [2016]
- a $\partial 4$ sketch with basic algorithmic knowledge can be trained jointly with an upstream LSTM [Hochreiter and Schmidhuber, 1997] for solving word algebra problems

The latter demonstrates that $\partial 4$ and the LSTM can "learn to read" natural language narratives with numerical values and reason with them to answer mathematical questions, all without the need for explicit intermediate representations used in previous work.

The contributions in this chapter are the following: i) we present $\partial 4$, a differentiable interpreter—a neural implementation of the FORTH dual-stack machine, ii) we introduce $\partial 4$ sketches for transferring the inductive bias of programs into a neural network, iii) we use these sketches as a strong inductive bias on learning algorithm from data, iv) we introduce program code optimisations to

speed up neural execution, and v) using ∂4 sketches, we obtain state-of-the-art for end-to-end reasoning about quantities expressed in natural language narratives.

3.2 Background: Forth Abstract Machine

FORTH [Moore and Leach, 1970, ANSI, 1994, Koopman, 1993] is a simple, procedural, stack-based programming language, with a long history of application in embedded systems. It is still being actively used today [Furter and Hauser, 2018], although its interesting properties make it also a somewhat esoteric language for today's standards.

FORTH is a semantically defined language, which makes it resemble more to a dictionary of words/commands, where each word is semantically well-defined as a specific operation than to a standard programming language [Knaggs, 1993]. This makes its interpreter easier to build, as it relies on simple lookup operations rather than on an explicit grammar as a formal description of its syntax.[2]

Appropriately, FORTH statements are called *words*,[3] and programs written in FORTH are composed of a sequence[4] of these words [Brodie and FORTH Inc, 1987]. Albeit simple, FORTH is an extensible language which promotes decomposition and abstraction by permitting the definition of new words that, in turn enabling programmers to write elaborate programs.

The semantics of FORTH are defined in terms of the FORTH dual-stack abstract machine, which we define by its machine state, the instruction set and the execution loop [Thomas, 2018], all together forming the FORTH interpreter. For our purposes, the FORTH *machine state* S contains all the memory elements that the language operates on. The FORTH *instruction set* $\mathfrak{F}$ contains the operation semantics of all the core words, and it is the basis of the *interpreter* which executes the program P on a state S. In an imperative language, this boils down to executing a single command of the program before moving on to the next one.

[2] Since the grammar of FORTH can easily change from one part of a program to another, as even core-defined commands can be changed at any moment, it would be inappropriate to model its syntax with a static grammar.

[3] FORTH commands and routines are called words, hence we use all these terms interchangeably.

[4] FORTH is a concatenative language.

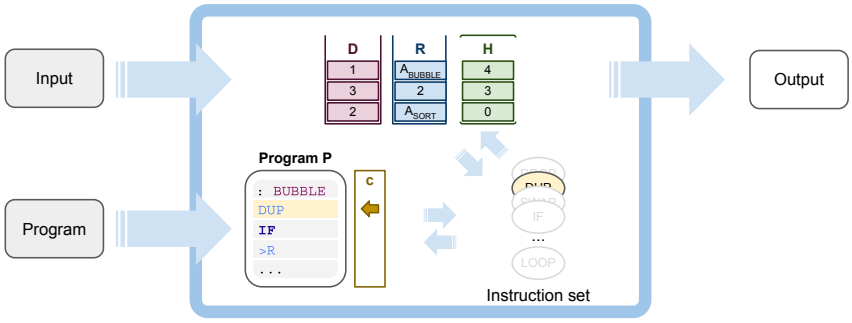


Figure 3.1: A depiction of the FORTH Abstract Machine. The input is written onto the data stack D, the return stack R, and/or the heap H. The program counter c picks the command from the program P that is to be executed. The instruction set contains the definitions of all commands, out of which c picks the current one. The selected command is executed on the machine state S, and the result is written back onto it. The program counter c advances to the next command.

A depiction of the FORTH abstract machine, including its machine state, instruction set and the execution loop, is given in Figure 3.1. Though the exact details of FORTH formalisation depend from a source to source [Knaggs, 1993, Koopman, 1993, Thomas, 2018], we opted for this particular formalism to facilitate grounding the later continuous relaxation of the FORTH abstract machine and its operations.

3.2.1 Forth Machine State

As noted, FORTH abstract machine (the FORTH interpreter $\Upsilon_{\mathfrak{F}}$) contains the execution loop, based on the fixed core instruction set $\mathfrak{F}$, operating on the machine state S. The machine state is a tuple $S = (D, R, H, c)$ consisting of two *stacks*: a data evaluation pushdown stack D (referred to as *data stack*) which holds values for data manipulation, and a return address pushdown stack R (referred to as *return stack*) which stores return addresses for subroutine calls. These two stacks are accompanied by a memory *heap* H (a random access memory buffer) that holds data and the program P, and a *program counter*[5] c that contains the address (i.e. pointer) of the word being currently executed [Koopman, 1993].

Each FORTH word in the instruction set $\mathfrak{F}$ operates on the FORTH machine state and produces a new state. The semantics of the state change depends on each of the words in the instruction set $\mathfrak{F}$.

[5] Also known as the Instruction Pointer (IP) or the Execution Pointer (EP)

3.2.2 Forth Instruction Set

Per Equation (2.23), each FORTH word $\mathtt{w}_i \in \mathfrak{F}$ is an instruction which is semantically a transition function between two machine states:

$$[\![\mathtt{w}_i]\!] : S \to S \,. \tag{3.1}$$

FORTH is a rich language with more than 100 words [ANSI, 1994, Brodie and FORTH Inc, 1987, Brodie, 2004, Thomas, 2018], and implementing an interpreter supporting all of them would take considerable time and effort. We decide to implement a small (but certainly not minimal) subset of all available FORTH words that would make our interpreter fully expressive and would showcase the breadth of applicability of continuous relaxations to discrete commands, while ultimately being useful and enabling users to write short programs. This led us to implement words primarily operating on the data stack, but also including a few words directly operating on the return stack as well as the heap. We put a particular emphasis on complex words for defining and calling subroutines, influencing control flow, and additionally implemented a few words for defining variables to simplify the user experience.

For the purpose of this thesis, we roughly divide the implemented FORTH words into 7 groups: data stack, return stack and heap operations, control, subroutine control, variable creation and other, and we describe them accordingly.

Note that top-of-the-stack (TOS) denotes the value on the top of the stack, and next-on-stack (NOS) denotes the value immediately below the TOS value.

Data stack operations These words include operations which directly manipulate data stack elements.

`int`	pushes integer literal `int` on the (data) stack. We allow only non-zero integer literals in this work.
`DROP`	pops a literal off the stack
`DUP`	duplicates the TOS
`SWAP`	swaps the TOS and NOS
`OVER`	pushes a copy of NOS (on the stack)
`1+, 1-`	increment and decrement of the TOS

+, -, *, /	arithmetic operations applied to NOS and TOS
>, <, =	comparators applied to NOS and TOS; if the result is TRUE, push `1` on stack, otherwise push `0`[6]

Return stack operations These words tend to the transfer of values between the data stack and the return stack.

>R	pop data stack TOS and push it to return stack
R>	pop return stack TOS and push it to data stack
@R	pushes (a copy of) return stack TOS to data stack

Heap operations These words tend to the transfer of values to and from the heap.

@	pushes the value from the data stack TOS heap address on the data stack
!	stores data stack NOS value on the TOS address in the heap

Control statements These words tend to the decision of whether, and if so, how many times will other statements be executed. Please note that each token of these commands carries its own separate semantics in FORTH but we explain them together as their combination easily relates to standard control statements in other languages. "`..`$_x$" denotes a sequence of FORTH words.

IF..THEN	pops the data stack TOS and executes `..` if it is equal to `1`
IF..$_1$**ELSE..**$_2$**THEN**	pops the data stack TOS and executes executes `..`$_1$ if it equals to `1`, otherwise executes `..`$_2$
BEGIN..$_1$**WHILE..**$_2$**LOOP**	keeps executing `..`$_2$ while `..`$_1$ equals to `1`
DO..LOOP	executes `..`, increments index (TOS) and repeats the same while the index does not equal to the limit (NOS)

Subroutine control These words tend to the new word definition and invocation. New words are defined with `: sub .. ;`, where `:` marks the start of

[6] In ANSI FORTH, FALSE is represented with -1.

the subroutine, sub denotes the subroutine / word identifier, .. denotes the subroutine commands, and ; marks the end of the subroutine.

: sub	saves the address of the subroutine for use in subsequent subroutine calls.
sub	calls a subroutine sub
;	exits a subroutine

Variable creation These words tend to the definition of variables.

VARIABLE var	creates a variable var on the heap
CREATE var ALLOT int	creates an array var of size int on the heap
var	variable call; pushes the value of var to data stack

Other These words are not a part of the FORTH specification and were introduced to ease the experience.

NOP	does nothing
MACRO: sub .. ;	treats the identifier sub as a macro: prior to starting the interpretation, the identifier sub is replaced with its defining words ..

These are the (core) words that make up our FORTH instruction set $\mathfrak{F}$. For more detail on the exact semantics of these commands, see Appendix A.1.

Lastly, we support code commenting with two types of comments:

```
\ comment from backslash to end of line


( in-line comment in parentheses )
```

The in-line comments in parentheses are often used as stack comments to indicate the stack invariants by indicating the state of the stack before and after the word execution, separated by `--`. For example, the stack comment `(a1 .. an seq_len -- an .. a1)` for the word `SORT` in Listing 3.1 aims to explain that `SORT` removes the top value of the stack (`seq_len`) and shuffles the elements below (it sorts them, but we could not write that down compactly).

```
1  : BUBBLE   ( a1 .. an seq_len-1 -- a1 .. an seq_len-1 )
2      DUP IF >R
3          OVER OVER < IF SWAP THEN
4          R> SWAP >R 1- BUBBLE R>
5      ELSE
6          DROP
7      THEN
8  ;
9  : SORT   ( a1 .. an seq_len -- an .. a1 )
10   1- DUP 0 DO >R R@ BUBBLE R> LOOP DROP
11 ;
12 2 1 3 3 SORT   \ Example call
```

Listing 3.1: A Bubble sort implementation in FORTH. Note that in ANSI FORTH specification, the word `BUBBLE` in line 4 should be replaced by the word `RECURSE`.

Next, we present the FORTH program, executed by the FORTH interpreter $\Upsilon_{\mathfrak{F}}$ word by word, thus transitioning from a state to a state.

3.2.3 Forth Program

A FORTH program P is a flat sequence of FORTH words per Equation (2.24):

$$P = (\mathtt{w}_1, \mathtt{w}_2, \ldots, \mathtt{w}_n)\,. \tag{3.2}$$

Given the compositionality of semantics in Equation (2.25), a program P is semantically a composition of a transition functions, hence a transition function too:

$$\begin{aligned} [\![\, P \,]\!] &= [\![\, (\mathtt{w}_1, \mathtt{w}_2, \ldots, \mathtt{w}_n) \,]\!] \\ &= [\![\, \mathtt{w}_n \,]\!] \circ \cdots \circ [\![\, \mathtt{w}_2 \,]\!] \circ [\![\, \mathtt{w}_1 \,]\!]\,. \end{aligned} \tag{3.3}$$

Though the FORTH program P, and the state S are usually considered as a part of the heap H, we consider them separate to ease the exposition.

Example An example of a bubble sort algorithm implemented in FORTH is shown in Listing 3.1. We explain this program in the interest of easier understanding of this code as parts of it are used in the experimental section.

The recursive solution for the bubble sort in Listing 3.1 starts at line 12, where the input is defined as a sequence of numbers (`2 1 3`) pushed on the data stack together with the length of the sequence (`3`), after which the word `SORT` is called.

The `SORT` word, defined in lines 9–11, executes the outer loop of the sort. This loop calls the bubbling procedure `BUBBLE` which moves the biggest element of

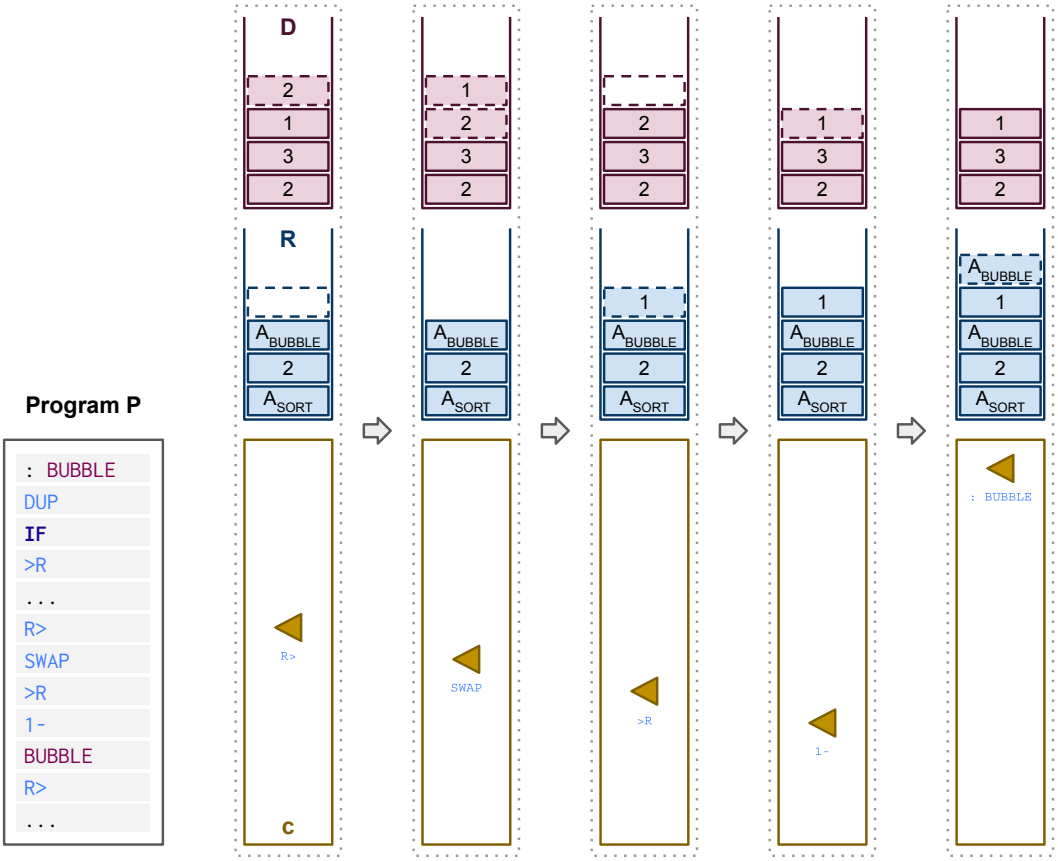


Figure 3.2: Graphical depiction of a part of the machine state (D, R, c) during Bubble sort in Listing 3.1. Stack elements changed between steps marked with a dashed line, left-pointing triangles denote program counters pointing to the currently executed command of the program (also indicated by command below the triangle). We omit the heap as it is not used in the program. See text below for more detail.

the sequence to the bottom of the data stack. Concretely, line 10 sets up the start (`1- DUP`) and the limit (`0`) of the `DO` loop and calls the body of the loop as many times as the input sequence is long, minus 1. The body of the `DO` loop saves the copy of the length of the input sequence on the return stack (`>R R@`), calls the `BUBBLE` word, and returns back the length of the input sequence from the return stack back to the data stack (`R>`). The final `DROP` just removes that length from the data stack after the sequence has been sorted.

The `BUBBLE` word, defined in lines 1–8, executes the recursive logic of the bubbling operation. `DUP` in line 2 duplicates the top of the data stack containing the length of the subsequence, which guides the conditional `IF`. If the length is 0, there are no more values to 'bubble' so the length is dropped with `DROP` in line 6. This effectively means that the element below the length of the

sequence is in its correct place. Otherwise, if the length is non-zero, the length of the subsequence is temporarily moved to the return stack (`>R`). The whole of line 3 is the comparator that ensures that NOS is bigger or equal than TOS. Following the comparison, line 4 returns the length of the subsequence back to the top of the data stack and moves the lesser of the numbers in the comparison on the return stack (`R> SWAP >R`). This denotes one bubbling movement done, and the length of the subsequence is decremented (`1-`), and `BUBBLE` is called recursively again on the subsequence. The machine state of the FORTH interpreter for the line 4 of the `BUBBLE` word, is depicted in Figure 3.2. `R>` after the recursive call makes sure all the temporarily saved non-maximal elements of the subsequence which were saved on the return stack are returned on the data stack for the next bubbling iteration in the `DO` loop in line 10.

3.2.4 Forth Execution Loop and Interpretation

Execution Loop The execution loop performs the execution cycle of FORTH. It executes the transition function of each word, and consequently the whole program, over a sequence of intermediate states, thus making the core loop of the FORTH interpreter $\Upsilon_{\mathfrak{F}}$. Based on the instruction set $\mathfrak{F}$, the loop, given a program P and an input state, produces the next state, per Equation (2.27).

Procedurally, the FORTH execution loop implements a simple three-stage execution. First, it reads the current word $\mathtt{w}_i$ of the program P that the program counter c points to. Second, it searches for the definition of the word $\mathtt{w}_i$ among its core words and the word dictionary, and if found, it executes the word. Third, if the word does not exist in the dictionary, it tries to convert it to a literal, e.g. a number or a variable address, and if that fails, it throws an error.

Concretely, given a program P and a state at time t, $S^{(t)}$, the program counter c of the state $S^{(t)}$ uniquely defines the word $\mathtt{w}^P_{c(t)}$ (where $c(t)$ is the program counter of the state $S^{(t)}$) to be executed, and executes it on the state $S^{(t)}$, leading it to the following state $S^{(t+1)}$ as follows:

$$S^{(t+1)} = \left[\!\left[\mathtt{w}^P_{c(t)} \right]\!\right] (S^{(t)}) \,. \tag{3.4}$$

Interpretation Based on this equation, which defines execution of a single word, and the Equation (3.3), the interpreter $\Upsilon_{\mathfrak{F}}$ presents a recurrent execu-

tion/composition of words of program P, starting at the initial state $S^{(0)}$:

$$\begin{aligned}\Upsilon_{\mathfrak{F}}(P, S^{(0)}) &= (\left[\!\left[\mathtt{w}^P_{c(t)} \right]\!\right] \circ \left[\!\left[\mathtt{w}^P_{c(t-1)} \right]\!\right] \circ \cdots \circ \left[\!\left[\mathtt{w}^P_{c(0)} \right]\!\right])(S^{(0)}) \\ &= \left[\!\left[\mathtt{w}^P_{c(t)} \right]\!\right] (\left[\!\left[\mathtt{w}^P_{c(t-1)} \right]\!\right] (\ldots (\left[\!\left[\mathtt{w}^P_{c(0)} \right]\!\right] (S^{(0)})))) .\end{aligned} \tag{3.5}$$

What is crucial to notice here is that the program counter c decides what is the current word being parsed, looked up and executed, and that the execution of that word determines the next word $\mathtt{w}_{c(t+1)}$. This is directed by FORTH's control statements and subroutine control words, which are ultimately reduced to low-level jumps and conditional jumps (`BRANCH` and `BRANCH0`) to a set of labels tagging code blocks (e.g. invoking a new word is a branch to a label that tags the first word in the definition of the new word).

By now, we presented all the elements of the FORTH interpreter—the machine state, the instruction set, and the program, as well as the execution loop. In the next section, we present the continuous approximation of the FORTH interpreter.

3.3 The Differentiable Forth Abstract Machine ∂4

When writing a FORTH program, programmers define a sequence of FORTH words, i.e. a sequence of known state transition functions. This caters for the case then the programmer knows how they want to model the computational process. To accommodate for cases when the developer's knowledge is incomplete, i.e. when the programmer does not know all the necessary detail how to appropriately define the computational process, or even cases when defining the computational process is impossible, we extend FORTH to support the definition of a *program sketch* [Solar-Lezama et al., 2006]. As with standard FORTH programs (Equation (3.3)), sketches too are semantically a composition of transition functions. However, the important distinction between a sketch and a program is that the sketch may contain transition functions whose behaviour can be learned from data.

We achieve this ability to learn the transition functions by continuous relaxation of the FORTH abstract machine. Continuous relaxation of the full FORTH abstract machine affords us the differentiability of the complete ma-

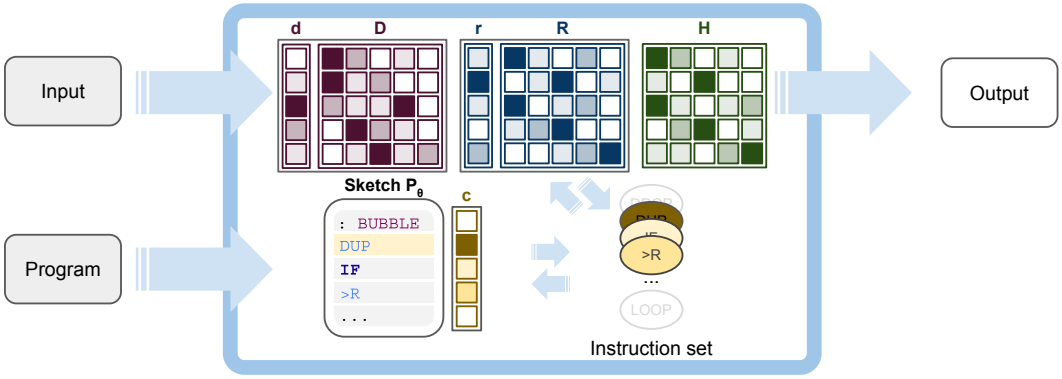


Figure 3.3: A depiction of ∂4, the differentiable FORTH abstract machine. The input is written onto the differentiable data stack $\boldsymbol{\mathcal{D}}$, return stack $\boldsymbol{\mathcal{R}}$ and/or the heap $\mathbf{H}$. Now, the instruction set $\mathfrak{d}4$ contains continuously relaxed ∂4 commands. All of them are executed on the machine state S and the results are weighted by the differentiable program counter $\mathbf{c}$ and written back onto the machine state $\mathbf{S}$. Finally, the differentiable program counter $\mathbf{c}$ is advanced.

chine, which, when combined with gradient optimisation methods, supports the training logic necessary to learn the transition functions and the representations of inputs. We can observe this through Equation (3.5)—a differentiable abstract machine would enable us to choose parametrised transition functions, such as neural networks, for words, as well as enable learning representations of the input state $S^{(0)}$.

To this end, we introduce a differentiable FORTH abstract machine ∂4—a continuous relaxation of the FORTH abstract machine. This abstract machine comprises a continuous machine state representation $\mathbf{S}$, a set of continuously relaxed instructions $\mathfrak{d}4$, parametrised program/sketch $\mathbf{P}_\theta$, and a continuous interpreter $\Upsilon_{\mathfrak{d}4}$ performing continuous program execution.

We implemented ∂4 in TensorFlow [Abadi et al., 2015], and made it freely available under the MIT license[7] at https://github.com/uclnlp/d4/.

3.3.1 ∂4 Machine State Encoding

We map the symbolic machine state $S = (D, R, H, c)$ of FORTH, defined in Section 3.2.1, to a continuous representation $\mathbf{S} = (\boldsymbol{\mathcal{D}}, \boldsymbol{\mathcal{R}}, \mathbf{H}, \mathbf{c})$. This representation consists of two *differentiable stacks*—the data stack $\boldsymbol{\mathcal{D}}$ and the return stack $\boldsymbol{\mathcal{R}}$—a *differentiable heap* $\mathbf{H}$ and the *program counter vector* $\mathbf{c}$. Figure 3.3

[7] https://opensource.org/licenses/MIT

depicts the ∂4 abstract machine together with its elements.

The data stack $\boldsymbol{\mathcal{D}} = (\mathbf{D}, \mathbf{d})$ is a tuple consisting of a data buffer $\mathbf{D}$ and its top-of-the-stack pointer $\mathbf{d}$, and the return stack $\boldsymbol{\mathcal{R}} = (\mathbf{R}, \mathbf{r})$ consists of a return buffer $\mathbf{R}$ and its top-of-the-stack pointer $\mathbf{r}$.

We represent two types of values in ∂4: integers and general dense vectors. Integers are represented with a normalised vector of non-zero values that sum to 1, and we code all addresses and pointers $\mathbf{d}$, $\mathbf{r}$, and $\mathbf{c}$, and literals as integers. General dense vectors, on the other hand, do not have restrictions on their values. That makes each memory buffer $\mathbf{D}$, $\mathbf{R}$, and $\mathbf{H}$ a matrix of row-vectors, where each vector is either an integer vector or a general dense vector.

3.3.1.1 Differentiable Flat Memory Buffers

All three memory structures, the data stack $\boldsymbol{\mathcal{D}}$, the return stack $\boldsymbol{\mathcal{R}}$ and the heap $\mathbf{H}$, are based on a generic differentiable flat memory (see Section 2.1.4) $\mathbf{M} \in \mathbb{R}^{l \times v}$, with l denoting the memory size, i.e. stack length, and v denoting the value size.[8] The difference is only in the data stack and the return stack having their dedicated top-of-the-stack pointers $\mathbf{d}$ and $\mathbf{r}$, respectively.

Read The differentiable flat memory buffer $\mathbf{M}$ has well-defined differentiable reading and writing operations, as defined in Equation (2.16) [Graves et al., 2014, 2016]. Concretely, the reading operation is defined as:

$$\text{read}_{\mathbf{M}}(\mathbf{a})^{\intercal} \leftarrow \mathbf{a}^{\intercal}\mathbf{M}, \tag{3.6}$$

where $\mathbf{a} \in \mathbb{R}^l$ denotes the address vector, and $\mathbf{M} \in \{\mathbf{D}, \mathbf{R}, \mathbf{H}\}$, denote that this operation is applicable to all three memory structures. The address vector $\mathbf{a}$ is represented as an integer with a normalised vector, with $\sum_i a_i = 1$ and $\forall i \quad 0 \leq a_i \leq 1$. In the one-hot vector case, this amounts to returning the exact row-value from the buffer, indexed by the position of 1 in $\mathbf{a}$.

Write The write operation is defined akin to the write operation in Neural Turing Machiness (NTMs) [Graves et al., 2014] (Equation (2.17)):

$$\text{write}_{\mathbf{M}}(\mathbf{x}, \mathbf{a}) : \mathbf{M}^{(t+1)} \leftarrow \mathbf{M}^{(t)} - (\mathbf{a} \otimes \mathbf{1}) \odot \mathbf{M}^{(t)} + \mathbf{x} \otimes \mathbf{a}, \tag{3.7}$$

[8] The equal value size v of all three memory buffers allows a direct transfer of vector representations of values between them.

where **a** is the address pointer.

As opposed to the NTM write formulation in Equation (2.17), we do not explicitly use a custom learnt erase vector, but simply use the full address vector **a** to delete the appropriate values from the memory buffer **M**.

3.3.1.2 Differentiable stack(s)

The general differentiable flat memory buffer formulation can use any network-produced vector as the addressing vector. Differentiable stacks have a dedicated top-of-the-stack (TOS) vector that is taken care of by each read and write operation. Concretely, in addition to the memory buffers **D** and **R**, the data stack and the return stack contain pointers to the current TOS element, $\mathbf{d}, \mathbf{r} \in \mathbb{R}^l$, respectively.

Push This allows us to implement pushing as writing a value **x** into **M** and incrementing the TOS pointer as a side-effect after writing:

$$\text{push}_{\mathbf{M}}(\mathbf{x}) : \text{write}_{\mathbf{M}}(\mathbf{x}, \mathbf{p}), \quad \text{[side-effect: } \mathbf{p} \leftarrow \text{inc}(\mathbf{p})\text{]} \tag{3.8}$$

where $\mathbf{p} \in \{\mathbf{d}, \mathbf{r}\}$ denotes either a data stack TOS or a return stack TOS, and $\text{inc}(\mathbf{p})^\intercal = \mathbf{p}^\intercal \mathbf{O}^{1+}$ is an increment operation with $\mathbf{O}^{1+} \in \mathbb{R}^{l \times l}$ being a left circular shift matrix, i.e. an increment matrix defined as:

$$\mathbf{O}^{1+}_{ij} = \begin{cases} 1 & i + 1 \equiv j \pmod{l} \\ 0 & otherwise \end{cases}.$$

Pop Popping is realised by multiplying the TOS pointer and the memory buffer, and decreasing the TOS pointer as a side-effect after reading:

$$\text{pop}_{\mathbf{M}}(\,) \leftarrow \text{read}_{\mathbf{M}}(\mathbf{p}), \quad \text{[side-effect: } \mathbf{p} \leftarrow \text{dec}(\mathbf{p})\text{]} \tag{3.9}$$

where $\text{dec}(\mathbf{p})^\intercal = \mathbf{p}^\intercal \mathbf{O}^{1-}$, and $\mathbf{O}^{1-}$ is a right circular shift matrix, i.e. a decrement matrix, defined as:

$$\mathbf{O}^{1-}_{ij} = \begin{cases} 1 & i - 1 \equiv j \pmod{l} \\ 0 & otherwise \end{cases}.$$

Note that return values of unary operators such as `1+` and `1-` can too in general be calculated as a vector-matrix multiplication as above with appropriately shaped matrices, $\mathbf{O}^{1+}, \mathbf{O}^{1-} \in \mathbb{R}^{v \times v}$.

By now, we can read from and write to any memory element, and execute unary operations. To be able to construct a fully differentiable stack machine, we need a way to influence the execution with a differentiable program counter $\mathbf{c}$.

3.3.1.3 Differentiable program counter

In a discrete machine, a program counter c points to the word which should be executed, i.e. a program state the interpreter should transition to, whereas in a differentiable interpreter a program counter $\mathbf{c}$ represents a probability distribution over words, i.e. over program states the interpreter can transition to. Concretely, the program counter $\mathbf{c} \in \mathbb{R}^p$, where p is the length of the sketch $\mathbf{P}_\theta$ and $\sum_i \mathbf{c} = 1$, is a vector (i.e. soft-attention, as in Equation (2.15)) denoting a probability distribution over words (state transitions) in the sketch $\mathbf{P}_\theta$. In a case where the program counter $\mathbf{c}$ is a one-hot vector, i.e. all the probability mass is on a single word, the result of the execution would be equivalent to the execution of the discrete machine, i.e. the resulting transition would correspond to a single command. However, in a general case, $\mathbf{c}$ weighs all the possible transition states of a sketch $\mathbf{P}_\theta$ and as a result leads the differentiable interpreter to a mixed machine state which is the convex combination of $\mathbf{c}$ and all the states that each word in $\mathbf{P}_\theta$ leads to, from a starting state.[9]

In order to deal with diverting program flow, e.g. conditionals and loops, which directly operate the program counter $\mathbf{c}$, we need to define two branching operations—the unconditional branch operation, and the conditional branch0 operation.

branch The unconditional branch operator simply sets the $\mathbf{c}$ to a requested address as:

$$\text{branch}(\mathbf{a}) \colon \mathbf{c} \leftarrow \mathbf{a}\,, \tag{3.10}$$

thus diverting the program flow to the $\mathbf{a}$ address of the sketch $\mathbf{P}_\theta$. This operation is used by default with every non-diverting command, to increment

[9] This makes the program counter $\mathbf{c}$ essentially a mean-field approximation of a categorical distribution over all words in a sketch $\mathbf{P}_\theta$.

the program counter by 1 as branch(inc(**c**)), but is also often used elsewhere, for example for subroutine calls.

branch0 [10] The conditional branch operation diverts the program flow, conditioned on the value of the data stack TOS:

$$\text{branch0}(\mathbf{a}): \begin{cases} s \leftarrow =(\text{pop}_{\mathbf{D}}(), \texttt{false}) \\ \mathbf{c} \leftarrow s\mathbf{a} + (1-s)\,\text{inc}(\mathbf{c}) \end{cases}, \tag{3.11}$$

where $\texttt{false} = \mathbf{1}_0$ denotes a one-hot vector with 1 on the 0-th element of the vector, and 0 elsewhere, representing a false evaluation of the comparison. This makes the result of the operation expressing the program counter **c** a convex combination of addresses **a** and the following program counter value inc **c**, hence possibly "splitting" the program counter over two values.[11] Note that the `=` command is defined as a binary operation tensor in the following subsection.

3.3.2 ∂4 Instruction Set

Given the continuous reading, writing and branching defined in the previous section, we can convert FORTH instruction set $\mathfrak{F}$, defined as functions on discrete machine states S in Section 3.2.2, to the ∂4 instruction set $\mathfrak{d}\mathbf{4}$ operating on the continuous machine states **S**.

For example, consider the FORTH word `DUP`, which duplicates the top of the data stack. Akin to the discrete version, the differentiable version of `DUP` does the same by reading off the data stack TOS with $\mathbf{x} \leftarrow \text{read}_{\mathbf{D}}(\mathbf{d})$, and pushes the read value on the data stack $\text{push}_{\mathbf{D}}(\mathbf{x})$.

Akin to the unary operators inc and dec, which are defined as a vector-matrix product, we define a binary operator as a bilinear tensor product:

$$\text{op}(\mathbf{x}, \mathbf{y}) \leftarrow \mathbf{x}^{\intercal} \mathsf{R}^{\text{op}} \mathbf{y}\,, \tag{3.12}$$

[10] The name branch0 signifies branching if a value (on the TOS) is equal to zero.

[11] Albeit the unconditional branch can be presented as a conditional branch ($\text{branch}(\mathbf{a}) = \text{branch0}(\mathbf{a}) \circ \text{push}_{\mathbf{D}}(0)$), we separate these two to lessen the burden on notation.

where $\mathsf{R}^{\text{op}} \in \mathbb{R}^{l\times l\times l}$ is a binary operation tensor defined as:

$$\mathsf{R}^{\text{op}}_{ijk} = \begin{cases} 1 & i \,\text{op}\, j \equiv k \pmod{l} \\ 0 & otherwise \end{cases},$$

for all binary operators, including comparators $\text{op} \in \{+, -, *, /, <, >, =\}$.

This concludes all helper functions sufficient to define all the commands of ∂4. However, to simplify further exposition, we additionally define the next-on-stack (NOS) pointer as:

$$\mathbf{p}^{-1\intercal} \leftarrow \mathbf{p}^{\intercal}\mathbf{O}^{1-}.$$

The complete formalisation of continuously relaxed FORTH words from Section 3.2.2 is given in the following equations:

Data stack operations Since these carry out pushing, popping and data transformations, their continuous relaxations include compositions of reading, writing and unary/binary operation composition on the data stack $\boldsymbol{\mathcal{D}}$.

$$[\![\, \underline{\texttt{int}} \,]\!] \stackrel{\text{def}}{=} \text{push}_{\mathbf{D}}(\mathbf{1}_{\underline{\texttt{int}}})^{12} \tag{3.13}$$

$$[\![\, \texttt{DROP} \,]\!] \stackrel{\text{def}}{=} \text{pop}_{\mathbf{D}}() \tag{3.14}$$

$$[\![\, \texttt{DUP} \,]\!] \stackrel{\text{def}}{=} \text{push}_{\mathbf{D}}(\text{read}_{\mathbf{D}}(\mathbf{d})) \tag{3.15}$$

$$[\![\, \texttt{SWAP} \,]\!] \stackrel{\text{def}}{=} \begin{cases} a \leftarrow \text{pop}_{\mathbf{D}}() \\ b \leftarrow \text{pop}_{\mathbf{D}}() \\ \text{push}_{\mathbf{D}}(a) \\ \text{push}_{\mathbf{D}}(b) \end{cases} \tag{3.16}$$

$$[\![\, \texttt{OVER} \,]\!] \stackrel{\text{def}}{=} \text{push}_{\mathbf{D}}(\text{read}_{\mathbf{D}}(\mathbf{d}^{-1})) \tag{3.17}$$

$$[\![\, \texttt{1+} \,]\!] \stackrel{\text{def}}{=} \text{write}_{\mathbf{D}}(\text{inc}(\text{read}_{\mathbf{D}}(\mathbf{d})), \mathbf{d}) \tag{3.18}$$

$$[\![\, \texttt{1-} \,]\!] \stackrel{\text{def}}{=} \text{write}_{\mathbf{D}}(\text{dec}(\text{read}_{\mathbf{D}}(\mathbf{d})), \mathbf{d}) \tag{3.19}$$

$$[\![\, \underline{\texttt{op}} \,]\!] \stackrel{\text{def}}{=} \begin{cases} a \leftarrow \text{pop}_{\mathbf{D}}() \\ b \leftarrow \text{pop}_{\mathbf{D}}() \\ \text{push}_{\mathbf{D}}(\underline{\texttt{op}}(a, b))^{13} \end{cases} \tag{3.20}$$

Heap operations These operations are simply reading and writing operations executed on the heap $\mathbf{H}$.

$$[\![\texttt{@}]\!] \stackrel{\text{def}}{=} \text{push}_{\mathbf{D}}(\text{read}_{\mathbf{H}}(\text{pop}_{\boldsymbol{\mathcal{D}}}())) \tag{3.21}$$

$$[\![\texttt{!}]\!] \stackrel{\text{def}}{=} \begin{cases} \mathbf{x} \leftarrow \text{pop}_{\mathbf{D}}() \\ \mathbf{a} \leftarrow \text{pop}_{\mathbf{D}}() \\ \text{write}_{\mathbf{H}}(\mathbf{x}, \mathbf{a}) \end{cases} \tag{3.22}$$

Return stack operations Given the nature of transferral to and from the return stack, these operations are based on reading and writing operations executed between the return stack $\boldsymbol{\mathcal{R}}$ and the data stack $\boldsymbol{\mathcal{D}}$.

$$[\![\texttt{>R}]\!] \stackrel{\text{def}}{=} \text{push}_{\mathbf{R}}(\text{pop}_{\mathbf{D}}()) \tag{3.23}$$

$$[\![\texttt{R>}]\!] \stackrel{\text{def}}{=} \text{push}_{\mathbf{D}}(\text{pop}_{\mathbf{R}}()) \tag{3.24}$$

$$[\![\texttt{@R}]\!] \stackrel{\text{def}}{=} \text{push}_{\mathbf{D}}(\text{read}_{\mathbf{R}}(\mathbf{r})) \tag{3.25}$$

Control statements These statements particularly rely on the branching commands branch and branch0. They are effectively implemented through the use of label addresses. For example, $addr_{word}$ defines an address—a value of the program counter at a specific location in the code—and invoking it in a branching command returns the value of the address, effectively branching to that location. Labelling and branching together enable control statements of ∂4.

$$[\![\texttt{IF..}_1\texttt{THEN}]\!] \stackrel{\text{def}}{=} \begin{cases} \text{branch0}(\text{inc}(addr_{THEN})) \\ [\![\texttt{..}_1]\!] \end{cases} \tag{3.26}$$

$$[\![\texttt{IF..}_1\texttt{ELSE..}_2\texttt{THEN}]\!] \stackrel{\text{def}}{=} \begin{cases} \text{branch0}(addr_{\texttt{..}_2}) \\ [\![\texttt{..}_1]\!] \\ \text{branch}(\text{inc}(addr_{THEN})) \\ [\![\texttt{..}_2]\!] \end{cases} \tag{3.27}$$

[12] $\underline{\texttt{int}}$ is a literal denoting a non-negative integer, and $[\![\underline{\texttt{int}}]\!]$ denotes the value of the integer.

[13] $\underline{\texttt{op}} \in [\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{<}, \texttt{>}, \texttt{=}]$

$$[\![\ \texttt{BEGIN..}_1\texttt{WHILE..}_2\texttt{REPEAT}\ ]\!] \stackrel{\text{def}}{=} \begin{cases} [\![\ \texttt{..}_1\ ]\!] \\ \text{branch0}(\text{inc}(addr_{REPEAT})) \\ [\![\ \texttt{..}_2\ ]\!] \\ \text{branch}(addr_{\texttt{..}_1}) \end{cases} \tag{3.28}$$

$$[\![\ \texttt{DO..LOOP}\ ]\!] \stackrel{\text{def}}{=} \begin{cases} [\![\ \texttt{..}\ ]\!] \\ [\![\ \texttt{+1 =}\ ]\!] \\ \text{branch0}(addr_{\texttt{..}}) \\ [\![\ \texttt{DROP DROP}\ ]\!] \end{cases} \tag{3.29}$$

Subroutine control These commands boil down to saving calling addresses in preprocessing and then using those addresses during runtime. In the preprocessing step subroutine definition `:` $\underline{\texttt{sub}}$ effectively saves the address $addr_{\underline{sub}}$ as the address immediately after $\underline{\texttt{sub}}$ which is then used by the subroutine invocation.

$$[\![\ \underline{\texttt{sub}}\ ]\!] \stackrel{\text{def}}{=} \begin{cases} \text{push}_{\mathbf{R}}(\text{inc}(\mathbf{c})) \\ \text{branch}(addr_{\underline{sub}}) \end{cases}^{15} \tag{3.30}$$

$$[\![\ \texttt{;}\ ]\!] \stackrel{\text{def}}{=} \text{branch}(\text{pop}_{\mathbf{R}}()) \tag{3.31}$$

Variable creation Variable creation words are implemented with preprocessing. Variable names used with both `VARIABLE` and `CREATE` are simply replaced with a statically pre-allocated address on the heap and their invocations simply return the allocated addresses as literals.

Other `NOP` does nothing but simply stepping to the next word, and `MACRO` is a function implemented via inlining before interpretation.[16]

3.3.3 ∂4 Sketches

We define a FORTH sketch $\mathbf{P}_\theta$ as a sequence of continuous transition functions $\mathbf{P}_\theta = (\mathtt{w}_1, \mathtt{w}_2, \ldots, \mathtt{w}_n)$. Here, $[\![\ \mathtt{w}_i\ ]\!] : \mathbf{S} \to \mathbf{S}$ either corresponds to a neural FORTH

[15] $\underline{\texttt{sub}}$ is a literal denoting a subroutine name.

[16] Inlining replaces a function call (word) with the body of the invoked function (word definition).

word or a parameterised transition function $[\![\mathtt{w}_{\theta_i}]\!] : \mathbf{S} \times \theta \to \mathbf{S}$, i.e. an MLP in our case. We will call these trainable functions *slots*, as they correspond to underspecified *slots* in the program code that need to be filled by learned behaviour.

We allow users to define a slot `w` as an MLP by specifying the elements of the MLP, concretely the input layer, the hidden layers and the output layer.

The input layer $[\![\mathtt{w}_{\text{in}}]\!] : \mathcal{S} \to \mathbb{R}^m$ consumes a user-specified subset of the machine state $\mathbf{S}^{(t)}$ and produces a latent representation of the machine state, fed into the following layer—a hidden or the output layer. The hidden layer $[\![\mathtt{w}_{\text{h}}]\!] : \mathbb{R}^m \to \mathbb{R}^n$ consumes the representation of the previous layer (representation dimensions agree between layers) and maps it into a new latent representation. The latent representation of the input or the previous hidden layer is then transformed by the output layer $[\![\mathtt{w}_{\text{out}}]\!] : \mathbb{R}^o \to \mathcal{S}$ which maps it into the next machine state $\mathbf{S}^{(t+1)}$.

This enables us to chain these transformations like $[\![\mathtt{w}]\!] = [\![\mathtt{w}_{\text{out}}]\!] \circ [\![\mathtt{w}_{\text{h}_n}]\!] \circ \ldots \circ [\![\mathtt{w}_{\text{h}_2}]\!] \circ [\![\mathtt{w}_{\text{h}_1}]\!] \circ [\![\mathtt{w}_{\text{in}}]\!]$ like the MLP does, as in Equation (2.8). To use slots within FORTH program code, we introduce a notation that reflects this decomposition.

In particular, slots are defined by the following syntax:

$$\texttt{\{ input (-> hidden)}^* \texttt{ -> output \}},$$

where `input` specifies the input layer, `(-> hidden)*` specifies zero or more hidden layers, and `output` specifies the output layer, as described in more detail below.

Input We provide the following two options for the input layer of the MLP:

`static`	produces a static representation, independent of the actual machine state; essentially a bias-only input layer.
`observe` $e_1 \ldots e_m$	concatenates the elements $e_1 \ldots e_m$ of the machine state $\mathbf{S}$. An element can be a stack item `Di` at relative index i, a return stack item `Ri`, etc.

Hidden The hidden layers are specified by chaining the affine transformation and the activation function as in Equation (2.7), fully enabling the MLP layer:

linear N	represents the affine transformation of the hidden layer—a linear transformation that projects the representation to N dimensions.
sigmoid, tanh	represents the activation function φ of the hidden layer.

Output Users can specify the following ways how the output can influence the machine state:

choose $\mathtt{w}_1 \ldots \mathtt{w}_m$	chooses from the FORTH words $\mathtt{w}_1 \ldots \mathtt{w}_m$. Takes the output of the previous layer, transforms it into a softmaxed vector h of length m to produce a weighted combination of machine states $\sum_i^m h_i \mathtt{w}_i(\mathbf{S})$.
manipulate $e_1 \ldots e_m$	directly manipulates the machine state elements $e_1 \ldots e_m$ by taking the output of the previous layer, transforming it into a softmaxed vector and writing it directly in the specified machine state elements with $\text{write}_{\mathbf{M}}$.[17]
permute $e_1 \ldots e_m$	permutes the machine state elements $e_1 \ldots e_m$; it takes the output of the previous layer, and transforms it into a softmaxed vector that weights the $m!$ permutations of state vectors in lexicographic ordering, i.e. for 2 elements, the output is a convex combination of $e_1 e_2$ and $e_2 e_1$.

Note that parameters of these layers are shared between the RNN execution of the same MLP. We do not yet support parameter sharing between different MLPs.

For example, consider the slot defined by:

```
{ observe D0 D-1 -> choose 1+ 1- }.
```

[17] Instead of softmaxing the output, one could rely on the optimisation to directly produce the required output. However, this is substantially more difficult to train.

This slot uses the data stack TOS (`D0`) and NOS (`D-1`) elements as state representation to determine whether to execute `1+` or `1-`. This corresponds to a parametrised single-layer neural network whose input is the concatenation of TOS and NOS vectors, and the output is the state S, equal to the convex combination of states led to by the `1+` and `1-` words.

3.3.4 ∂4 Execution Loop and the Interpreter

We model the ∂4 execution loop using an execution RNN which produces the next state $\mathbf{S}^{(t+1)}$ conditioned on the previous state $\mathbf{S}^{(t)}$. A single command execution (Equation (2.23)) is achieved by passing the current state $\mathbf{S}^{(t)}$ to each word/function $[\![\mathtt{w}_i]\!]$ of the sketch $\mathbf{P}_\theta$, and weighing the obtained outputs with the program counter vector $\mathbf{c}$:

$$\iota_{\mathbf{P}_\theta}(\mathbf{S}^{(t-1)}) = \sum_{i=1}^{|\mathbf{P}_\theta|} c_i^{(t-1)} [\![\mathtt{w}_i]\!](\mathbf{S}^{(t-1)}) = \mathbf{S}^{(t)}\,. \tag{3.32}$$

Note that here the counter vector $\mathbf{c}$ weighs the state $\mathbf{S}^{(t-1)}$ by simply weighing each element of the state. Following Equation (2.10), the execution function defines the execution RNN, which defines the ∂4 interpreter $\Upsilon_{\partial 4}$ (Equation (2.27)) ran/unrolled for predetermined, user-specified t steps from a starting state $\mathbf{S}^{(0)}$ as:

$$\begin{aligned}
\Upsilon_{\partial 4}(\mathbf{P}_\theta, \mathbf{S}^{(0)}, t) &= \mathrm{eRNN}_{\mathbf{P}_\theta}(\mathbf{S}^{(0)}, t) \\
&= (\iota_{\mathbf{P}_\theta} \circ \ldots \circ \iota_{\mathbf{P}_\theta} \circ \iota_{\mathbf{P}_\theta})(\mathbf{S}^{(0)}) \\
&= \iota_{\mathbf{P}_\theta}(\ldots(\iota_{\mathbf{P}_\theta}(\iota_{\mathbf{P}_\theta}(\mathbf{S}^{(0)})))) \\
&= \mathbf{S}^{(t)}\,,
\end{aligned} \tag{3.33}$$

where t denotes the number of time steps to unroll the RNN, $\mathbf{P}_\theta$ denotes the sketch parametrised with θ, $\mathbf{S}^{(0)}$ is the initial state, often initialised to the required input, and allowing a slight abuse of notation where $\mathbf{c}$ is the program counter corresponding to the state $\mathbf{S}^{(t-1)}$. Clearly, this recursion, and its final state, are differentiable with respect to the parameters θ of the sketch $\mathbf{P}_\theta$, as well as the input state $\mathbf{S}^{(0)}$.

Furthermore, in case of a parameterless sketch consisting of only ∂4 words (i.e. no slots are present) and one-hot values, the final state $\mathbf{S}^{(t)}$ of the ∂4 program will correspond exactly to the final state of the equivalent FORTH program.

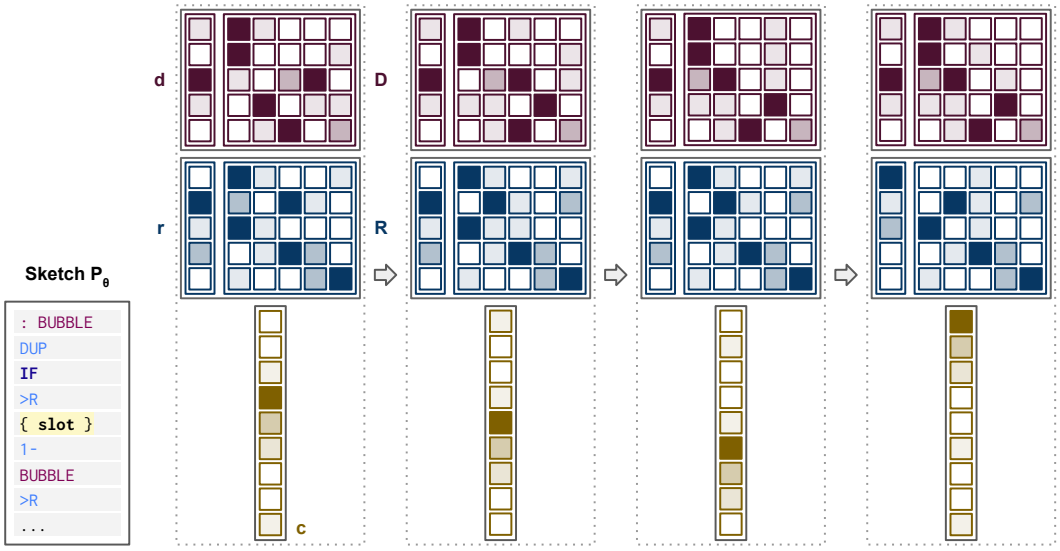


Figure 3.4: Graphical depiction of a part of the machine state during Bubble sort sketch in Listing 3.3. Heap omitted for clarity as it is not used in the sketch.

An example of a ∂4 machine state, during the Bubble Sort sketch in Listing 3.3 is depicted in Figure 3.4.

3.4 Training

Given a dataset of input-output pairs of machine start and desired end values, $\mathfrak{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$, our goal is to infer the parameters θ of the sketch $\mathbf{P}_\theta$ in a supervised fashion. Although we can determine the complete start and end state of the machine with the dataset $\mathfrak{D}$, in our experiments we use only the data stack $\boldsymbol{\mathcal{D}}$—we set the input data $\mathbf{x}_i$ on the data stack $\boldsymbol{\mathcal{D}}$ and expect the output $\mathbf{y}_i$ on the same data stack after execution. In our case, the training input $\mathbf{x}_i$ uniquely defines a starting data stack $\boldsymbol{\mathcal{D}}_i^{(0)} = (\mathbf{D}_i^{(0)}, \mathbf{d}_i^{(0)})$ and is directly encoded into the starting state $\mathbf{S}^{(0)}$.[18] Similarly, the training output $\mathbf{y}_i$ uniquely defines the desired final data stack $\boldsymbol{\mathcal{D}}_i^{(t)} = (\mathbf{D}_i^{(t)}, \mathbf{d}_i^{(t)})$.

Since the size of the outputted data stack and the desired one do not necessarily have to correspond to each other. This is because as we might be interested in only a subset of the stack, given the popping operation is non-destructive and values stay on the stack. Therefore we introduce a mask $\mathbf{K}_i$ that indicates

[18] The training input $\mathbf{x}_i$ is directly written on top of the "empty state". In an empty state, $\mathbf{D}$ and $\mathbf{R}$ are set to a matrix of one-hot zero rows, $\mathbf{d}$ and $\mathbf{r}$ are set to a one-hot value pointing to the first element of the stack, and $\mathbf{c}$ is set to point to the first command of the sketch $\mathbf{P}_\theta$.

which components of both stacks should contribute to the loss. This mask is uniquely defined by the desired output $\mathbf{y}_i$ and it effectively denotes which values of the data stack buffer we care about, and which should be ignored. We define the loss function of $\partial 4$ as follows:

$$\begin{aligned}\mathcal{L}_\theta(\mathfrak{D}) &= \sum_{i\in\mathfrak{D}} H(\mathbf{K}_i \odot \mathbf{S}_i^{(t)}, \mathbf{K}_i \odot \hat{\mathbf{S}}_i^{(t)}) \\ &\approx \sum_{i\in\mathfrak{D}} H(\mathbf{K}_i \odot \mathbf{D}_i^{(t)}, \mathbf{K}_i \odot \hat{\mathbf{D}}_i^{(t)}) + H(\mathbf{K}_i \odot \mathbf{d}_i^{(t)}, \mathbf{K}_i \odot \hat{\mathbf{d}}_i^{(t)}),\end{aligned} \tag{3.34}$$

where $H(\mathbf{x},\mathbf{y}) = \sum_i -x_i \log y_i$ is the cross-entropy loss[19] and $\hat{\mathbf{S}}_i^{(t)} = \Upsilon_{\mathfrak{d}4}(\mathbf{P}_\theta, \mathbf{S}_i^{(0)}, t))$ is the final output state, with the circumflex denoting the network output (i.e. the value estimate).

Since both the loss and the whole machine is end-to-end differentiable, we can use backpropagation and any gradient optimisation method to optimise this loss function and the parameters θ. Note that although we are optimising parameters over a possibly long timeline, it would be possible to include trace-based supervision, i.e. supervision at each intermediate state, as done by the Neural Programmer-Interpreter (NPI) [Reed and De Freitas, 2016].

3.4.1 Interpreter Optimisations

The above-defined interpreter $\Upsilon_{\mathfrak{d}4}$ requires a single RNN time step per transition. Concretely, a single time step implies the execution of every possible state transition of the sketch $\mathbf{P}_\theta$, weighing each element of the state $\mathbf{S}$ with the program counter $\mathbf{c}$ (Equation (3.33)), after which the program counter is updated by either an increment (next instruction), explicit assignment (function call, control statement operations) or a pop from the return stack (function exit). Since every time step is computationally expensive, a full RNN execution is consequently very costly and decreasing the number of RNN steps, while retaining the equivalence of the calculation would speed up both the training and the inference. To that extent, we employ two strategies, the *symbolic execution* and the *branch interpolation*.

Symbolic Execution Whenever we have a sequence of FORTH words that contains no slots and no branch entry or exit points, we can collapse said

[19] Note that the cross-entropy is applied element-wise and the result is the summation of all the elements

sequence into a single transition instead of naively interpreting words one-by-one at each time step. We symbolically execute [King, 1976] a sequence of FORTH words to calculate a new machine state. We then use the difference between the symbolically calculated machine state and the initial state to derive the transition function of the sequence. This effectively means we derive single-step transition function (concretely, which matrix/tensor operations) we need to execute to produce the same effect as a sequence of multiple transition functions corresponding to the sequence of words.

For example, starting in a symbolic state $D = (d_1, d_2, ..., d_l)$ and $R = (r_1, r_2, ..., r_l)$, the sequence `R> SWAP >R`, swaps top elements of the data and the return stacks, and results in a new symbolic state $D = (r_1, d_2, ..., d_l)$ and $R = (d_1, r_2, ..., r_l)$. Comparing the initial state and the and the resulting state, we derive a single transition (i.e. matrix/tensor operations) that only needs to swap the top elements of $\boldsymbol{\mathcal{D}}$ and $\boldsymbol{\mathcal{R}}$.

Branch Interpolation We can apply symbolic execution on any sequence of words where the transitions are independent of the machine state. However, that does not hold for code with branching points—the machine state which results from branching behaviour is contingent on the current machine state and hence cannot be resolved symbolically.

For example, the code `0 = IF SWAP THEN DUP ELSE` cannot be resolved symbolically as the transitions themselves are dependent on the machine state (concretely, if the data stack TOS is equal to 0, the resulting symbolic state would be $D = (d_2, d_1, ..., d_l)$, otherwise it would be $D = (d_1, d_1, d_2, ..., d_l)$.

However, we can still collapse `IF`-branches that involve no function calls or loops, by executing both branches in parallel and weighing their output states by the value of the condition. Doing this again effectively replaces a sequence of matrix/tensor operations with a single equivalent operation. In cases where `IF` branches contain function calls or loops, we simply fall back to the execution of all words weighted by the program counter.

3.5 Experiments

We hypothesise that the differentiable interpreter ∂4, due to its strong architectural bias, enables:

H1 arbitrarily complex inductive bias

H2 support of working both with discrete and continuous data

H3 training from a small number of input-output data

H4 strong generalisation

H5 composition with other neural models, due to its differentiable nature

In this section, we aim to experimentally verify these hypotheses by evaluating $\partial 4$ on three tasks in total. The first two of these tasks are simple *transduction tasks* on discrete inputs (H2), presented as neural programming tasks of sorting and elementary addition, introduced in Reed and De Freitas [2016]. For each of these tasks, we present two $\partial 4$ sketches that capture different degrees of prior knowledge—both sketches provide a recursive structure of the algorithms (H1), enabling learning of number comparison for the sorting task, and digit addition for the addition task. We show that, given only a small number of input-output pairs (H3), $\partial 4$ can learn to fill the sketch with the missing behaviour and that the resulting sketch generalises well to problems several orders of magnitude bigger than the training ones (H4).

The last task we apply $\partial 4$ to is the task of solving an instance of *Word Algebra Problems*—algebra problems expressed in natural language. We show that $\partial 4$ can train a basic algorithmic sketch trained jointly with an upstream LSTM (H5). This shows that $\partial 4$ can learn to read natural language narratives, composed of both continuous (textual representations) and discrete data (numbers) (H2), extract important numerical quantities and reason with them, ultimately answering corresponding mathematical questions without the need for explicit intermediate representations, as is done in previous work.

In addition to these experiments, we analyse the speed improvements of the proposed optimisations, and qualitatively analyse the learning procedure.

3.5.1 Sorting

Sorting sequences of digits is a hard task for RNNs since they fail to generalise to sequences even marginally longer from than the training sequences [Reed and De Freitas, 2016]. Inspired by this issue, we investigate two sketches of the Bubble Sort program in Listing 3.1 that enable learning to sort from only a few hundred training examples, with a suitable bias.

```
1  : BUBBLE   ( a1 .. an seq_len-1 -- a1 .. an seq_len-1 )
2      DUP IF >R
3          { observe D0 D-1 -> choose NOP SWAP }
4          R> SWAP >R 1- BUBBLE R>
5      ELSE
6          DROP
7      THEN
8  ;
9  : SORT   ( a1 .. an seq_len -- an .. a1 )
10   1- DUP 0 DO >R R@ BUBBLE R> LOOP DROP
11 ;
12 SORT
```

Listing 3.2: The COMPARE sketch for the sorting task. Note that the data is directly fed onto the data stack externally.

3.5.1.1 Sketches

By defining sufficient procedural structure from Listing 3.1, we make the resulting network invariant to the input sequence length. The procedural part of the sketch is effectively sorting the input sequence by repeatedly executing a procedure which ensures that the biggest element of each sorting pass is moved to the bottom of the sequence, while the learned part is effectively learning the comparison of digits. Note that this comparison is implicitly learned from the input-output data, without any prior information, while imposing a strong inductive bias of the sorting algorithm containing loops, conditionals, function calls and recursion.

We relax the specification of the Bubble Sort code into two sketches, the COMPARE and the PERMUTE sketch.

Compare sketch The COMPARE sketch, listed in Listing 3.2, requires learning just the comparison of the numbers (NOS and TOS) on the data stack. This behaviour is learned by the slot—a parametrised neural network in the line 3 of Listing 3.2, which, observing the top two elements on the data stack (`D0` and `D-1`), chooses to either do the swap (`SWAP`) or not (`NOP`). The rest of the sketch is the same as the original Bubble Sort code in Listing 3.1.

Permute sketch The PERMUTE sketch, listed in Listing 3.3, provides less structure and leaves more behaviour open to be learned. Concretely, in contrast to the COMPARE sketch, the PERMUTE sketch requires learning both the comparison of the two top elements on the data stack and taking care of the length of the subsequence on the return stack. This is achieved by learning to permute the NOS and TOS of the data stack and the TOS of the return stack, conditioned on the values of the top two elements on the data stack that

```
1  : BUBBLE   ( a1 .. an seq_len-1 -- a1 .. an seq_len-1 )
2      DUP IF >R
3          { observe D0 D-1 -> permute D-1 D0 R0}
4          1- BUBBLE R>
5      ELSE
6          DROP
7      THEN
8  ;
9  : SORT   ( a1 .. an seq_len -- an .. a1 )
10   1- DUP 0 DO >R R@ BUBBLE R> LOOP DROP
11 ;
12 SORT
```

Listing 3.3: The PERMUTE sketch for the sorting task.

need to be compared. The parametrised neural network in the slot in line 3 requires that both the value comparison and the permutation behaviour must be learned.

3.5.1.2 Experimental Setup

We optimised each $\partial 4$ sketch with Adam [Kingma and Ba, 2015] for a maximum of 200 epochs, with early stopping on the development set. We added noise to gradients [Neelakantan et al., 2015b] and clipped gradients [Pascanu et al., 2013] larger than 1.0. We tuned the initial learning rate (1.0), batch size (between 16 and 64), and the parameters of the gradient noise in a random search on a development set for each task. During testing, we discretise all the continuous elements of the machine making the test-time execution discrete.

Baseline We compare these sketches to the standard seq2seq [Sutskever et al., 2014] baseline. The seq2seq baseline models are single-layer networks with LSTM cells of 50 dimensions. The training procedure for these models consists of 500 epochs of Adam optimisation, with a batch size of 128, a learning rate of 0.01, and gradient clipping when the L2 norm of the model parameters exceeded 5.0.

Data All the models were trained on randomly generated data—we uniformly chose each digit of the sequence and sorted the obtained sequence to provide a target. We generated the train, development and test sets containing 256, 32 and 32 instances, respectively. The low number of development and test instances was chosen to decrease the computational cost of the evaluation.

Table 3.1: Accuracy, expressed in Hamming distance, of PERMUTE and COMPARE sketches in comparison to a seq2seq baseline on the sorting problem. Dagger † denotes values different from values reported in Bošnjak et al. [2017]. For an explanation why that is, see the footnote.

	Test Length 8			Test Length: 64		
Train Length:	2	3	4	2	3	4
seq2seq	26.2	29.2	39.1	13.3	13.6	15.9
∂4 PERMUTE	**100.0**	**100.0**	**100.0**†	**100.0**	**100.0**	**100.0**†
∂4 COMPARE	**100.0**	**100.0**	**100.0**†	**100.0**	**100.0**	**100.0**†

3.5.1.3 Testing Strong Generalisation

A quantitative comparison of our models on the Bubble sort task is provided in Table 3.1. For a given test sequence length, we vary the training set lengths to illustrate the model's ability to generalise to sequences longer than those it observed during training. We find that ∂4 quickly learns the correct sketch behaviour, and it is able to generalise perfectly to sort sequences of 64 elements after observing only sequences of length two, three and four during training. In comparison, the seq2seq baseline falters when attempting similar generalisations, and performs close to chance when tested on longer sequences. Both ∂4 sketches perform flawlessly when trained on short sequence lengths. However, they both under-perform when trained on a sequence of length 5 and beyond since the execution RNN of ∂4 Bubble Sort unrolls to a large number of steps (122), given the quadratic nature of the Bubble Sort algorithm). This, in turn, causes numerical instabilities and erroneous large gradients which cause the learning to diverge.

3.5.1.4 The Effect of the Dataset Size

When measuring the performance of the model as the number of training *instances* varies, we can observe in Figure 3.5 the benefit of additional prior knowledge to the optimisation process. We find that when stronger prior knowledge is provided (COMPARE), the model quickly maximises the training accuracy. Providing less structure (PERMUTE) results in lower testing accuracy

†NOTE: In our experiments, long unrolls led to to numerical instabilities and erroneous gradients. In Bošnjak et al. [2017] these values were lower due to these errors. Here we are using the double precision (`float64`) to circumvent this issue. However, for train lengths 5 (unrolls to 122 steps) and beyond we still experience numerical instabilities, and none of the standard fixes we tried worked for us.

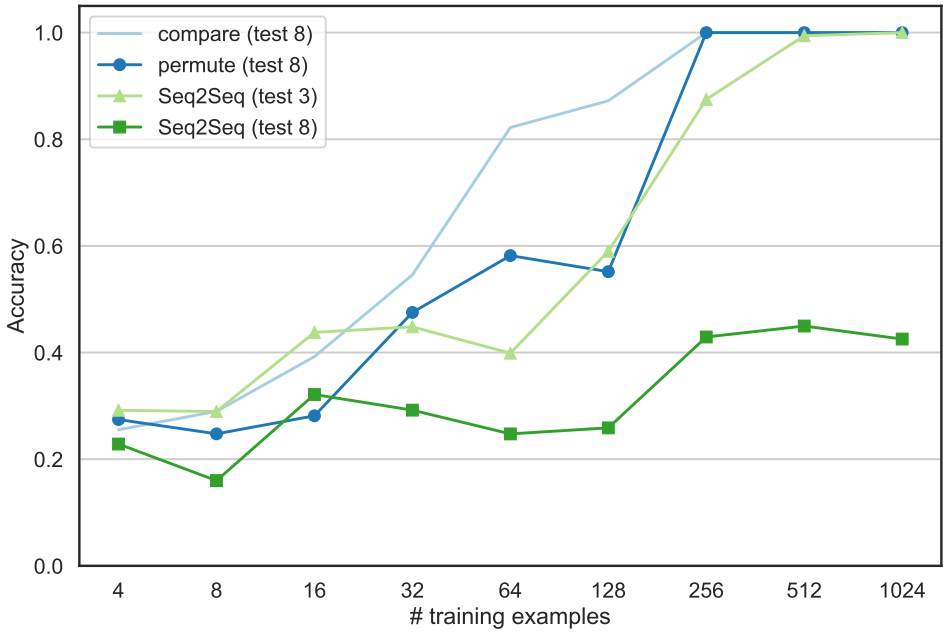


Figure 3.5: Accuracy of models for a varying number of training examples, trained on input sequence of length 3 for the Bubble sort task. COMPARE, PERMUTE, and seq2seq (test 8) were tested on sequence lengths 8, and seq2seq (test 3) was tested on sequence length 3.

initially, however, both sketches learn the correct behaviour and generalise equally well after seeing 256 training instances. Additionally, it is worth noting that the PERMUTE sketch was not always able to converge into a result of the correct length, and both sketches are not trivial to train.

In comparison, seq2seq baseline is able to generalise only to the sequence it was trained on (seq2seq trained and tested on sequence length 3). When training it on sequence length 3, and testing it on a much longer sequence length of 8, seq2seq baseline is not able to achieve more than 45% accuracy.

3.5.1.5 The Effect of the Program Code Optimisations

To quantify the usefulness of the interpreter optimisations introduced in Section 3.4.1, we run an ablation analysis over the options, analysing the number of possible state transitions the RNN can take, the number of steps the RNN takes in practice, and we take a look at the produced results.

Number of transitions The number of possible RNN transitions, produced by each of the optimisation options is given in Table 3.2. We see that the symbolic interpretation alone causes the number of transitions to halve. The

Table 3.2: The effect of the optimisations on the number of state transition functions of the FORTH implementation of Bubble sort in Listing 3.1.

Symbolic Execution	Branch Interpolation	# transitions
-	-	33
-	ON	32
ON	-	15
ON	ON	15

branch interpolation causes only a saving of one transition function when symbolic execution is not on. In this particular case, the branch interpolation does not seem meaningful, but in cases where there are multiple commands under each if and else branches, like `IF DUP SWAP ELSE SWAP DUP THEN`, branch interpolation would still be useful even if symbolic execution was turned on.

Steps Taken The number of steps taken, for each of the optimisation options is given in Figure 3.6. Again, the symbolic execution single-handedly causes a huge savings of a factor of almost 3. However, we see that the branch interpolation, albeit it does not cause a big win (as a matter of a fact, if paired with symbolic execution, it even results in a higher number of steps taken), does make the number of execution steps constant, regardless of the input.

Qualitative Analysis In Figure 3.7 we can see the concrete output of each of the optimisation options. We see the clear success of the symbolic execution, which merges a lot of commands, and the branch interpolation which is applicable just on the single case of the `IF` command.

3.5.1.6 Qualitative Analysis of Program Counter Traces

In Figure 3.8 we visualise the program counter traces. The trace follows a single example from the start, over the middle to the end of the training process. At the beginning of training, the program counter starts to deviate from the one-hot representation in the first 20 steps (not observed in the figure due to unobservable changes), and after a single iteration of `SORT`, the sketch already fails to correctly determine the next word. After a few training epochs ∂4 learns better permutations which enable the algorithm to take crisp decisions and halt in the correct state.

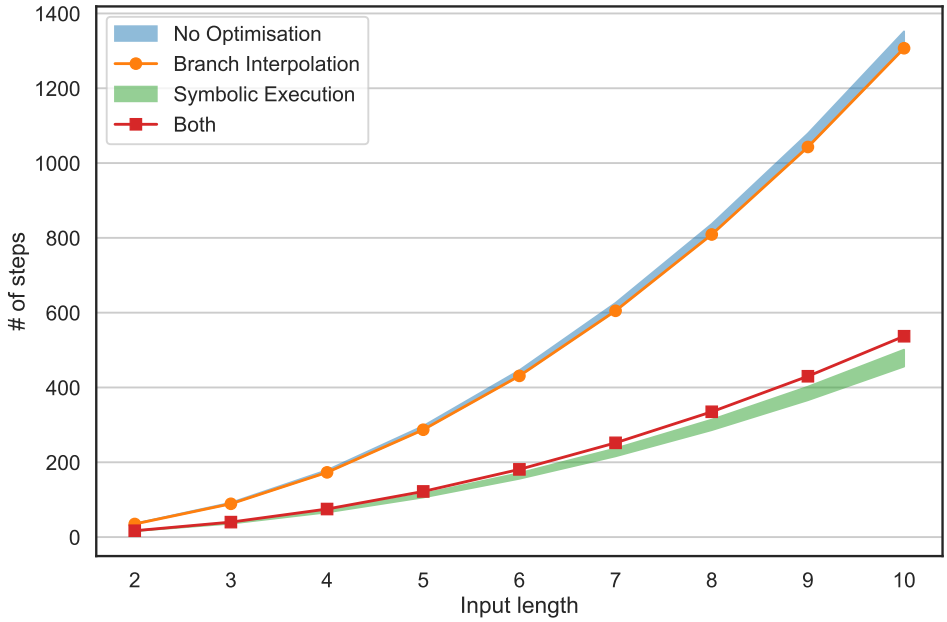


Figure 3.6: The effect of the optimisations on the number of (RNN) execution steps. Note that "No Optimisation" and "Symbolic Execution" are denoted by the filled area as the number of steps taken depends on the input.

3.5.2 Addition

Next, we applied ∂4 to the problem of learning to add two n-digit numbers. We rely on the standard elementary school addition algorithm, where the goal is to iterate over pairs of aligned digits, calculating the sum of each to yield the resulting sum. The key complication arises when two digits sum to a two-digit number, requiring that the correct extra digit—a *carry*—be carried over to the subsequent column.

We assume aligned pairs of digits as input, with a carry for the least significant digit (potentially equal to 0), and the length of the respective numbers. The sketches define the high-level operations through recursion, leaving the core addition to be learned from data.

The specified high-level behaviour includes the recursive call template and the halting condition of the recursion (no remaining digits, line 2 in Listing 3.4). The underspecified addition operation must take three digits from the previous call—the two digits to sum and a previous carry—and produce a single digit (the sum) and the resulting carry. The rest of the sketch code reduces the problem size by one and returns the solution, popping it from the return stack.

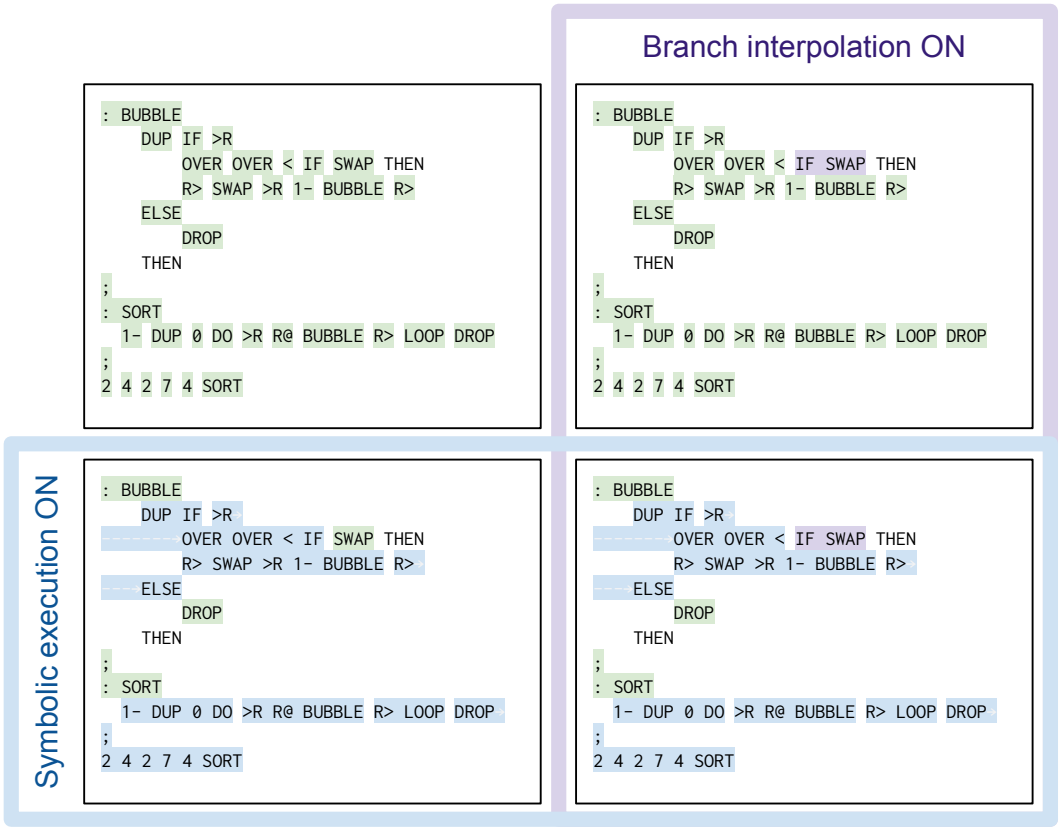
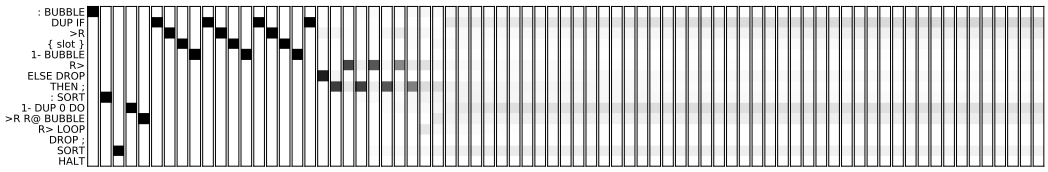


Figure 3.7: The results of different optimisation techniques applied to the Bubble sort program in Listing 3.1. Single commands highlighted in green are the result of standard execution, purple spans denote the commands collapsed by branch interpolation, and the blue spans denote the commands collapsed by symbolic execution. Arrows denote a cross-line span. We can see the drastic effect of the symbolic execution and the nuanced effect of the branch interpolation in this example.

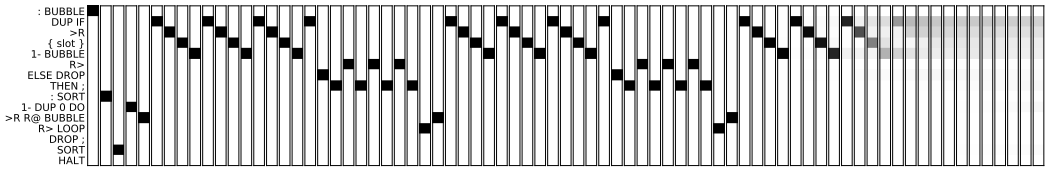
We introduce two sketches for inducing this behaviour, the Manipulate sketch and the Choose sketches.

Manipulate sketch The Manipulate sketch, listed in Listing 3.4, provides less prior knowledge as it directly manipulates the ∂4 machine state. It does so by filling in a carry and the result digits, based on the top three elements of the data stack—two digits and the carry—with a slot in line 6 of Listing 3.4. The same slot translates to a 3-layer perceptron consisting of the input layer with the tanh nonlinearity, a linear layer of 70 units and the output of 2 units.

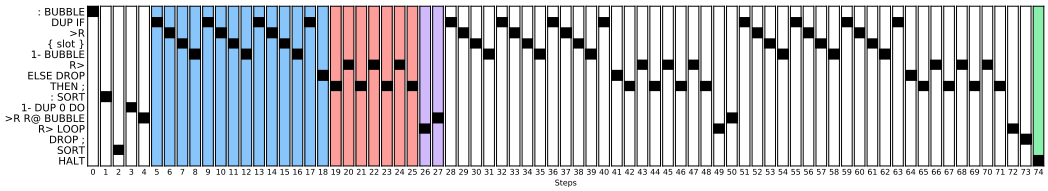
Choose sketch The Choose sketch, listed in Listing 3.5, provides more prior knowledge by exactly specifying the results of the computation. Concretely, the slot in line 6 contains a neural network that outputs the carry, and the slot



(a) PC trace at the start of training.



(b) PC trace mid-training.



(c) PC trace at the end of training.

Figure 3.8: Program Counter traces for a single example at different stages of training the Bubble sort PERMUTE sketch in Listing 3.3. Blue—successive recursion calls of `BUBBLE`; red—successive returns from the recursion; purple—calls to `SORT`; green—the halting state. The rows are labelled with words they represent after the optimisation step which groups them together.

```
1  : ADD-DIGITS ( a1 b1 a2 b2 ... an bn carry n -- r1 r2 ... r_{n+1} )
2    DUP 0 = IF
3      DROP
4    ELSE
5      >R \ put n on R
6      { observe D0 D-1 D-2 -> tanh -> linear 70 -> manipulate D-1 D-2 }
7      DROP
8      R> 1- SWAP >R  \ new_carry n-1
9      ADD-DIGITS \ call add-digits on n-1 subseq.
10     R> \ put remembered results back on the stack
11   THEN
12 ;
13 ADD-DIGITS
```

Listing 3.4: The MANIPULATE sketch for the Addition problem. Input data is used to fill data stack externally.

```
1  : ADD-DIGITS ( a1 b1 a2 b2 ... an bn carry n -- r1 r2 ... r_{n+1} )
2    DUP 0 = IF
3      DROP
4    ELSE
5      >R \ put n on R
6      { observe D0 D-1 D-2 -> tanh -> linear 10 -> choose 0 1 }
7      { observe D-1 D-2 D-3 -> tanh -> linear 50 -> choose 0 1 2 3 4 5 6 7 8 9 }
8      >R SWAP DROP SWAP DROP SWAP DROP R>
9      R> 1- SWAP >R  \ new_carry n-1
10     ADD-DIGITS \ call add-digits on n-1 subseq.
11     R> \ put remembered results back on the stack
12   THEN
13 ;
14 ADD-DIGITS
```

Listing 3.5: The CHOOSE sketch for the Addition problem. Input data is used to fill data stack externally.

Table 3.3: Accuracy (Hamming distance) of CHOOSE and MANIPULATE sketches in comparison to a seq2seq baseline on the addition problem. Note that lengths corresponds to the length of the input sequence (two times the number of digits of both numbers).

	Test Length 8			Test Length 64		
Train Length:	2	4	8	2	4	8
seq2seq	37.9	57.8	99.8	15.0	13.5	13.3
∂4 CHOOSE	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**
∂4 MANIPULATE	**98.58**	**100.0**	**100.0**	**99.49**	**100.0**	**100.0**

in line 7 specifies a neural network that needs to learn to output the resulting digit, both being conditioned on the two digits and the carry on the data stack.

Baseline As in the sorting experiment, we use seq2seq model as a baseline. The hyperparameter sweep is identical to the sweep done in the sorting experiments Section 3.5.1.2.

Training details We trained the addition CHOOSE and MANIPULATEx sketches presented in Table 3.1 on a randomly generated train, development and test sets of sizes 512, 256, and 1024 respectively. We uniformly sampled each digit of each n-digit numbers and added them to construct the target. The batch size was set to 16, and we used an initial learning rate of 0.05

3.5.2.1 Generalisation

In a set of experiments analogous to those in our evaluation on Bubble sort, we demonstrate the performance of ∂4 on the addition task by examining test set sequence lengths of 8 and 64 while varying the lengths of the train-

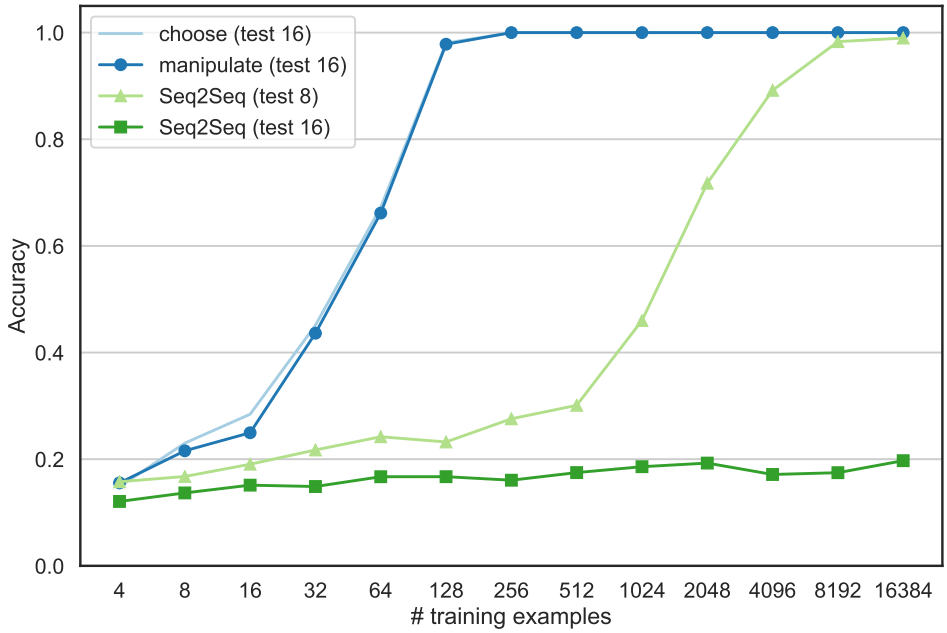


Figure 3.9: Accuracy of models for a varying number of training examples, trained on input sequence of length 8 for the addition task. MANIPULATE, CHOOSE, and seq2seq (test 16) were tested on sequence lengths 16, and seq2seq (test 8) was tested on sequence length 8.

ing set instances (Table 3.3). The seq2seq model again fails to generalise to longer sequences than those observed during training. In comparison, both the CHOOSE sketch and the less structured MANIPULATE sketch learn the correct sketch behaviour and generalise to all test sequence lengths (with an exception of MANIPULATE which required more data to train perfectly). In additional experiments, we were able to successfully train both the CHOOSE and the MANIPULATE sketches from sequences of input length 24, and we tested them up to the sequence length of 128, confirming their perfect training and generalisation capabilities.

Note that our experiments with MANIPULATE include softmaxing values written on the stack, as described in Section 3.3.3. If we remove the softmax, the sketch is able to learn only from the minimal sequence length of 2, and none other. We conjecture that this is due to difficulty of the direct state manipulation where the neural network in the slot needs to directly write the unbounded value on the data stack, and thus learn by itself the required output, as opposed to CHOOSE sketch where the user defines the output directly.

The role of softmaxing can be thought of as a neural version of typing—it

types the output to a number from 0 to l, as opposed to the more specific (categorical) type in the choose sketch.

3.5.2.2 Accuracy per number of training examples

We tested the models to train on datasets of increasing size on the addition task. The results, depicted in Figure 3.9 show that both the Choose and the Manipulate sketch are able to perfectly generalise from 256 examples, trained on sequence lengths of 8, tested on 16. In comparison, the seq2seq baseline achieves 98% when trained on 16384 examples, but only when tested on the input of the same length, 8. If we test seq2seq as we tested the sketches, it is unable to achieve more than 19.7% accuracy.

3.5.3 Word Algebra Problems

Word algebra problems are often used to assess the numerical reasoning abilities of schoolchildren. Questions are short narratives which focus on numerical quantities, culminating with a question. For example:

> *A florist had 50 roses. If she sold 15 of them and then later picked 21 more, how many roses would she have?*

Answering such questions requires both the understanding of language and of algebra—one must know which numeric operations correspond to which phrase and how to execute these operations.

Previous work cast word algebra problems as a transduction task by mapping a question to a template of a mathematical formula, thus requiring manually labelled formulas. For instance, one formula that can be used to correctly answer the question in the example above is `(50 - 15) + 21 = 56`. In previous work, local classifiers [Roy and Roth, 2015, Roy et al., 2015], hand-crafted grammars [Koncel-Kedziorski et al., 2015], and recurrent neural models [Bouchard et al., 2016] have been used to perform this task. Predicted formula templates may be marginalised during training [Kushman et al., 2014], or evaluated directly to produce an answer.

In contrast to these approaches, ∂4 is able to learn both, a soft mapping from text to algebraic operations *and their execution*, without the need for manually labelled equations and no explicit symbolic representation of a formula, by jointly training a ∂4 sketch with a neural sequence model.

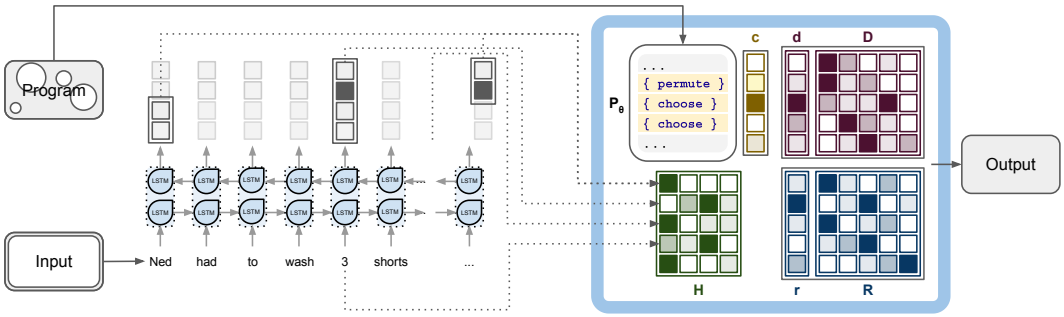


Figure 3.10: The Word Algebra Problem model: a Neural FORTH Abstract Machine (on the left) executing the Word Algebra Problem sketch, connected to the BiLSTM which reads word problems (on the right). Output vectors corresponding to a representation of the entire problem, as well as context representations of numbers and the numbers themselves are fed into **H** to solve tasks. The entire system is end-to-end differentiable.

3.5.3.1 Model Description and the Sketch

Our model is fully end-to-end differentiable, consisting of a $\partial 4$ interpreter with an appropriate sketch, and a BiLSTM reader.

The BiLSTM reader reads the text of the problem and produces a vector representation (word vectors) for each word, concatenated from the forward and the backward pass of the BiLSTM network. We use the resulting word vectors corresponding only to numbers in the text, numerical values of those numbers (encoded as one-hot vectors), and a vector representation of the whole problem (concatenation of the last and the first vector of the opposite passes) to initialise the $\partial 4$ heap **H**. This is done in an end-to-end fashion, enabling gradient propagation through the BiLSTM to the vector representations. The process is depicted in Figure 3.10.

The sketch, depicted in Listing 3.6 dictates the differentiable computation. First, it copies values from the heap **H** – word vectors to the return stack $\boldsymbol{\mathcal{R}}$, and numbers (as one-hot vectors) on the data stack $\boldsymbol{\mathcal{D}}$. Second, it contains four slots that define the space of all possible operations of four operators on three operands, all conditioned on the vector representations on the return stack. These slots are i) permutation of the elements on the data stack, ii) choosing the first operator, iii) possibly swapping the intermediate result and the last operand, and iv) the choice of the second operator. These four slots fully define the space of all possible formulas over three operands and four operators (e.g. `X+Y-Z`, `(X-Y)/Z`, etc.), however, the model needs to learn which

```
VARIABLE QUESTION                    \ address of the question on H

CREATE REPR_BUFFER 4 ALLOT           \ allotting H for representations
CREATE NUM_BUFFER 4 ALLOT            \ and numbers

VARIABLE REPR                        \ addresses of the first representation
VARIABLE NUM                         \ and number

REPR_BUFFER REPR !
NUM_BUFFER NUM !

                                     \ macro functions for:
MACRO: STEP_NUM NUM @ 1+ NUM ! ;     \ incrementing the pointer to numbers in H
MACRO: STEP_REPR REPR @ 1+ REPR ! ; \ incrementing the pointer to reps. in H
MACRO: CURRENT_NUM NUM @ @ ;         \ fetching current numbers
MACRO: CURRENT_REPR REPR @ @ ;       \ fetching current representations

CURRENT_NUM                          \ copy numbers to D
STEP_NUM
CURRENT_NUM
STEP_NUM
CURRENT_NUM

QUESTION @ >R                        \ copy question vector ..
CURRENT_REPR >R                      \ .. and representations of numbers to R
STEP_REPR
CURRENT_REPR >R
STEP_REPR
CURRENT_REPR >R

\ based on the question and number representations...
{ observe R0 R-1 R-2 R-3 -> permute D0 D-1 D-2 } \ permute stack elements
{ observe R0 R-1 R-2 R-3 -> choose + - * / }      \ choose the first operation
{ observe R0 R-1 R-2 R-3 -> choose SWAP NOP }     \ swap prev. result and 3rd num
{ observe R0 R-1 R-2 R-3 -> choose + - * / }      \ choose the second operation

R> DROP R> DROP R> DROP R> DROP                   \ empty out R
```

Listing 3.6: The Word Algebra Problem sketch

equation to induce in order to calculate the correct result. The final set of commands simply empties out the return stack $\mathcal{R}$.

3.5.3.2 Experimental Setup

We evaluate the model on the CC dataset, introduced by Roy and Roth [2015]. CC is notable for having the most diverse set of equation patterns, consisting of four operators (+, -, ×, ÷), with up to three operands.

The CC dataset is partitioned into a train, development, and test set containing 300, 100, and 200 questions, respectively. The batch size was set to 50, and we used an initial learning rate of 0.02. The BiLSTM word vectors were initialised randomly to vectors of length 75. The stack width was set to 150 and the stack size to 5.

Table 3.4: Accuracies of models on the CC dataset. The asterisk denotes results obtained from Bouchard et al. [2016]. Note that i) GeNeRe makes use of additional data, ii) the difference between our seq2seq baseline and the one in Bouchard et al. [2016] stems from the Bouchard et al. [2016] baseline being trained on train+dev sets following hyperparameter selection on the dev set.

Model	Accuracy (%)
Template Mapping	
Roy and Roth [2015]	55.5
seq2seq (our implementation)	85.0
seq2seq* Bouchard et al. [2016]	95.0
GeNeRe* Bouchard et al. [2016]	98.5
Fully End-to-End	
seq2seq	0.0
$\partial 4$	96.0

3.5.3.3 Results

We compare against three baseline systems: (1) a local classifier with handcrafted features [Roy and Roth, 2015], (2) a seq2seq baseline, and (3) the same model with a data generation component (GeNeRe) Bouchard et al. [2016]. All baselines are trained to predict the best equation, which is executed outside of the model to obtain the answer. In contrast, $\partial 4$ is trained end-to-end from input-output pairs and predicts the answer directly without the need for an intermediate symbolic representation of a formula.

Neural Models The comparison between $\partial 4$ and neural baselines is shown in Table 3.4. Our method slightly outperforms the seq2seq baseline from Bouchard et al. [2016]. However, upon reimplementation of the seq2seq baseline and a close inspection of the code by Bouchard et al. [2016], we noticed that their model was trained on the train+dev set after they selected the best hyperparameters on the dev set. As expected, we confirmed that the effect of their train set expansion has a positive effect on the final results. However, we still decided to train $\partial 4$ on train set only and use the dev set for early stopping, as is the standard practice. The reason why we decided to stick with the standard practice is that when training on train+dev, we cannot use the dev set for early stopping. This implies we have to train the model for a predetermined number of epochs, hoping that the training stops when the model generalises since at this point the dev set performance is what is being opti-

mised and cannot be used as a reliable estimate of the model's performance. In that sense, we cannot directly compare to the accuracies of the seq2seq baseline and GeNeRe from Bouchard et al. [2016].

However, our goal here is not to achieve the state-of-the-art results over GeNeRe (though it is quite possible that their performance is lower when trained on train set only) but to establish a close-to enough model that is, crucially, trained end-to-end without additional formulas. It is worthwhile to emphasise once again that, as opposed to GeNeRe and seq2seq from Bouchard et al. [2016], which are template mapping models, ∂4 is trained completely end-to-end, from the text of the problem to the final numeric result. Additionally, we ran the seq2seq baseline in the same end-to-end fashion, to quantify the difficulty of the task when solved end-to-end. The results confirm that the task is difficult—the seq2seq baseline is unable to train at all as it lacks data to even start learning arithmetic expressions and execution.

Comparison to Roy and Roth [2015] After publishing Bošnjak et al. [2017], where we compared the neural models with the model of Roy and Roth [2015], we noticed that there are significant differences in the datasets used in the experiments between Roy and Roth [2015] and Bouchard et al. [2016].

Concretely, the dataset splits in Bouchard et al. [2016] were done randomly, whereas the splits in Roy and Roth [2015] were done in a structured way which makes the task much more difficult, justifying the large difference between the performances of these two models. The issue stems from the fact that that Roy and Roth [2015] split the dataset into 6 folds, where each fold contains problems from a single equation category (e.g. addition followed by subtraction, subtraction and multiplication, etc.), making the dataset more challenging. Since they have split the dataset into train and dev only, Bouchard et al. [2016] wanted to split the dataset into train, dev and test, but they split the dataset randomly, mixing the equation categories across datasets, making the problem significantly easier to solve.

For a fair comparison, we ran our model on the original split. We split the train set into train and dev by making the dev set a random subset of the train set. The results of this experiment can be found in Table 3.5. We can observe that ∂4 performs poorly on this split as the model struggles to catch the relationships applicable over different equation templates. Interestingly,

Table 3.5: Performance of $\partial 4$ and the model from Roy and Roth [2015] on splits from Roy and Roth [2015]

Model	Accuracy (%)
Template Mapping	
Roy and Roth [2015]	45.2
Fully End-to-End	
$\partial 4$	5.0

when comparing the performance of Roy and Roth [2015] on the random split and the harder split, their model yields only 10% improvement, showing that the model struggles even though it has access to all the equation templates at training time. In contrast to that, $\partial 4$ is able to clearly capitalise on the access to all equation templates, which boosts its score drastically. It is again worth emphasising that $\partial 4$ is trained fully end-to-end, without using the formula template, thus solving a harder problem, as opposed to the Roy and Roth [2015] model.

3.6 Related Work

We present differentiable interpreters in the light of connections they draw from several related areas of research: computability of neural networks, program induction and synthesis, probabilistic programming and recent computational architectures in neural networks.

3.6.1 The Computational Power of Neural Networks

A body of prior groundwork established the theoretical background behind the mathematical notion of program expressivity and execution through the computational power and subsequent use of that power for language interpretation.

From Turing Completeness... The computational power of neural networks has been studied since the early days of neural networks. McCulloch and Pitts [1943] showed the correspondence between feed-forward neural networks with hard-thresholds and propositional logic, positing that the RNNs are even more expressive, and drawing conclusions that a NN with a tape is Turing Complete. The subsequent work of Kleene [1951] built on top of their work to relate the representational powers of hard-threshold neural networks and finite automata.

Later, Pollack [1987] showed that a particular architecture of RNNs with high-order connections is Turing complete, while Franklin and Garzon [1990] showed Turing completeness of a neural network with an unbounded number of neurons. This work lead to the seminal work of Siegelmann and Sontag [1991] and Siegelmann and Sontag [1995] who finally proved Turing completeness of finite RNNs without high-order connections.[20] This work gave a theoretical background to the representational power of recurrent architectures. Recent work by Pérez et al. [2019] corroborated these findings on modern NN architectures, which do not just interpret an algorithm but learn it as well. Though these proofs require infinite precision and computation time which computers cannot achieve, [Weiss et al., 2018] further shows that different architectures exhibit different computational powers in practice.

...to Language Interpretation and Compilation The Turing completeness of NNs entails their ability to simulate Universal Turing Machines, effectively making them interpreters for other Turing Machines. Some of the prior work focused on using these abilities of neural networks to build neural interpreters and compilers; translations of programs in a high-level language into a neural network which executes said program. Siegelmann [1994] were the first who proposed compiling a program, in their case written in the NEural Language (NETL), to a NN which computes the original program. Intended as a bridge between the symbolic and neural computation, their system compiled a FOL or any general algorithm to a RNN that simply executes it, without learning. Later, Gruau et al. [1995] presented a compiler that translates a PASCAL [Wirth, 1971] program into a neural network that executes it. Interestingly, although their neural compiler defines a function that calls a learned NN in the code, they do not support its training. Similarly, Siegelmann [1996] compile a program written in Neural Information Processing Programming Language (NIL) to a NN and enable training it with neural evolution. A series of work [Neto et al., 1998, 2003, Neto, 2006] showed that programs written in a high-level language called NETDEF can be translated to a modular RNNs, with Neto et al. [2000] adding to it the theoretical support to learn. Lastly, not all efforts were focusing on high-level programming languages, though; Neto et al. [2004] translated high-level Turing Machine programs in a form of partial

[20] On an interesting side-note, Siegelmann [1995] and Siegelmann [2012] show that neural networks with real numbers can achieve computational power beyond the Turing limit, but that work is out of the scope of this thesis.

recursive function descriptions to NNs. However, subsequent work did mostly use higher-level programming languages to facilitate use.

On a related note, the programming language community worked on the notion of smooth software and smooth interpretation. DeMillo and Lipton [1991] proved the existence of continuous functions which can capture the execution of discrete state transition functions, i.e. that software can be cast via continuous functions. Though their work did not recognise any practical applications of it, later work by Chaudhuri and Solar-Lezama [2010, 2011] used this to introduce the concept of "smooth interpretation" defined via relaxing discrete programs' states with Gaussian mixture distributions. Interestingly, they utilise it to synthesise parameters of, essentially, program sketches, but via Nelder-Mead [Nelder and Mead, 1965], a gradient-free optimisation technique. In more recent work, Inala et al. [2018] additionally relax this Gaussian smoothing akin to our approach and find parameters with a combination of SAT solving [Eén and Sörensson, 2003] and sequential quadratic programming [Gill et al., 2005]. to synthesise both the discrete and floating-point parameters of the sketch.

3.6.2 Program Synthesis

The idea of program synthesis [Gulwani et al., 2017] is as old as AI [Church, 1957], and has a long history in computer science [Manna and Waldinger, 1971]. Whereas a large body of work has focused on using genetic programming [Koza and Koza, 1992] to induce programs from the given input-output specification [Nordin, 1997], there are also various Inductive Programming approaches [Kitzelmann, 2009] aimed at inducing programs from incomplete specifications of the code to be implemented [Albarghouthi et al., 2013, Solar-Lezama et al., 2006]. Further work even tackled the problem of learning sketches from corpora of code [Murali et al., 2018]. We tackle the same problem of sketching, but in our case, we fill the sketches with neural networks able to learn the slot behaviour. It is worth noting that the very recent work on neural models pushed the state of the program synthesis research by either employing elaborate deep learning models [Parisotto et al., 2017], or by aiding a specialised synthesiser [Parisotto et al., 2017].

3.6.3 Probabilistic and Bayesian Programming

Our work is closely related to probabilistic programming languages such as Church [Goodman et al., 2012]. They allow users to inject random choice primitives into programs as a way to define generative distributions over possible execution traces. In a sense, the random choice primitives in such languages correspond to the slots in our sketches. A core difference lies in the way we train the behaviour of slots: instead of calculating their posteriors using probabilistic inference, we estimate their parameters using backpropagation and gradient descent. This is similar in-kind to `TerpreT`'s FMGD algorithm [Gaunt et al., 2016], which is employed for code induction via backpropagation. In comparison, our model which optimises slots of neural networks surrounded by continuous approximations of code enables the combination of procedural behaviour and neural networks. In addition, the underlying programming and probabilistic paradigm in these programming languages is often functional and declarative, whereas our approach focuses on a procedural and discriminative view. By using an end-to-end differentiable architecture, it is easy to seamlessly connect our sketches to further neural input and output modules, such as an LSTM that feeds into the machine heap.

3.6.4 Memory Augmented Neural Networks

Recently, there has been a surge of research in differentiable execution and program synthesis in deep learning, with increasingly elaborate deep models. Many of these models were based on differentiable versions of abstract data structures. Most well-known out of the continuous data structures are the continuous stack [Giles et al., 1990, Sun, 1991, Das et al., 1992, 1993, Sun et al., 1993, Joulin and Mikolov, 2015, Grefenstette et al., 2015], continuous queue [Grefenstette et al., 2015] and the general continuous memory, i.e. heap [Graves et al., 2014, Weston et al., 2015, Sukhbaatar et al., 2015].

Following them are the continuous abstract machines, such as the NTM [Graves et al., 2014] and their successors Differentiable Neural Computers [Graves et al., 2016], the Neural RAM [Kurach et al., 2016], and the Neural GPUs [Kaiser and Sutskever, 2016]. These continuous abstract machines are primarily used as algorithm learners that induce algorithmic behaviour from data, which already sets them apart from ∂4 which focuses on incorporating algorithmic knowledge into models. Taking a closer look at

these models, we see some similarities as well as differences between them, based on the data and the algorithm representation they are using.

The NTM and the Neural RAM, similarly to $\partial 4$, represent its data with a soft one-hot alike representation throughout the model. In principle, the same should be possible with fully dense data too, like the word embeddings used in our word algebra problem experiments. Neural GPUs, on the other hand, are using one-hot input data, but their hidden representations are completely opaque and cannot be interpreter nor easily influenced in any way.

All these models are based on different continuous abstract machines, which translates to different properties of their algorithm representation. The NTM, even though working with soft one-hot alike data, still contain an opaque model (MLP or LSTM) that represents its hard-to-interpret transition function that essentially codes up the program. Even if we were able to reliably extract an interpretable transition function, working with it is still quite hard because it requires a lot of effort to understand and program it. Neural GPUs are even worse in that regard as they are essentially a neural cellular automata which are impossible to interpret and to program. Neural RAM is better in that sense, as it generates a circuitry of commands that are interpretable and programmable. However, the language this machine implements, though easier to work with than with a Turing Machine, is less powerful than an assembly language and would require more careful planning when dealing with loops in their circuit-based execution. Admittedly, all these models can induce somewhat complex behaviour purely from data, which we do not do in this work, but we enable significantly easier incorporation of concrete programmatic constructs into models, which these models cannot.

Related to our efforts is also the Autograd [Maclaurin et al., 2015b,a], which enables automatic gradient computation in pure Python code, but does not define nor use differentiable access to its underlying abstract machine.

The work in neural approximations to abstract structures and machines naturally leads to more elaborate machinery able to induce and call code or code-like behaviour. Neelakantan et al. [2016] learned simple SQL-like behaviour—querying tables from the natural language with simple arithmetic operations. Although sharing similarities on a high level, the primary goal of our model is not induction of (fully expressive) code but its injection. Andreas et al. [2016b] compose parse tree -guided neural modules to produce the desired behaviour

for a visual QA task. Neural Programmer-Interpreters [Reed and De Freitas, 2016] learn to represent and execute programs, operating on different modes of an environment and are able to incorporate decisions better captured in a neural network than in many lines of code (e.g. using an image as an input). Users inject prior procedural knowledge by training on program traces and hence require *full* procedural knowledge. In contrast, we enable users to use their partial knowledge in sketches.

Neural approaches to language compilation have also been researched, from compiling a language into neural networks [Siegelmann, 1994, Neto et al., 1998, 2003], over building neural compilers [Gruau et al., 1995] to adaptive compilation [Bunel et al., 2016]. However, that line of research did not perceive neural interpreters and compilers as a means of injecting procedural knowledge as we did. To the best of our knowledge, ∂4 is the first working neural implementation of an abstract machine for an actual programming language, and this enables us to inject such priors in a straightforward manner.

Most recent work Quickly after we published a preprint of our work, more work on differentiable interpreters kept popping up, showing that similar ideas were developed concurrently. Notably, TerpreT's [Gaunt et al., 2016] (published as a preprint 3 months after our paper) neural network backend is an instance of a differentiable interpreter, employed for inductive program synthesis via backpropagation. Soon after, Feser et al. [2017] improved program induction via a set of structural changes modelled on functional programming constructs. Most similar to our work is the work by Gaunt et al. [2017] which successfully used differentiable interpreters for program synthesis with parametrised neural networks. Their differentiable interpreter is able to jointly induce code and train a neural network invoked in the same code. Albeit it seems that their model can fully induce code thus obviate a need for strong priors, our contribution still stands as it includes more complex programmatic structures such as unbounded loops and recursion, can be used to write longer code, and was published as a pre-print 6 months before their work. Their work opened interesting further development in the form of HOUDINI [Valkov et al., 2018], a neuro-symbolic hybrid that combines the symbolic program synthesis and differentiable function interpretation able to reuse neural networks from a library of components.

Lastly, we want to acknowledge DeepProbLog [Manhaeve et al., 2018], end-

to-end trainable integration of a probabilistic logic programming language ProbLog [De Raedt et al., 2007] with neural networks. This model successfully combines symbolic and sub-symbolic reasoning of a logic-based declarative language which is faster to execute has a smaller code footprint and produces comparable, and in some cases even better results than ∂4—it can be trained from longer input sequences on the sorting task. Even though it can provide better results in some cases, we note that with a lighter and optimised implementation, ∂4 should be able to perform much better than it does now, but we leave this for future work.

The wave of "differentiability" caught on and produced many other differentiable models, which due to differentiability, can now be treated like modules ready to be connected into a bigger model. Notable examples include differentiable physics engines [de Avila Belbute-Peres et al., 2018], used as parts of ML-based robotics models [Degrave et al., 2019], differentiable rendering [Loper and Black, 2014, Li et al., 2018, Liu et al., 2019], differentiable optimisation [Djolonga and Krause, 2017, Amos and Kolter, 2017], differentiable dynamic programming [Mensch and Blondel, 2018], differentiable arithmetic units [Trask et al., 2018], differentiable satisfiability solvers [Wang et al., 2019], and others [Ferber et al., 2019].

3.7 Conclusion and Future Work

In this chapter we presented ∂4, a differentiable interpreter for the Forth programming language. By designing ∂4 through a series of continuous approximations of the discrete interpreter execution, and using backpropagation, we are able to utilise it as a way to provide strong and complex procedural inductive biases. ∂4 complements programmers' prior knowledge by allowing them to code a strong and complex procedural inductive bias while enabling learning of unspecified behaviours. We showed experimentally that ∂4 learns to sort, add and solve word algebra problems, by using program sketches and a small number of input-output pairs, while achieving strong generalisation.

Future Work There are several directions we plan to explore for the future work of ∂4. **More user-friendly language paradigms**, like functional, logic, or in general declarative programming, as well as other host languages can ease the programmers' efforts, increase productivity, and open new venues, for example, differentiable database querying. **Scaling up learning and ex-**

ecution would open up the possibilities of using even larger and more complex programs, as well as induce or synthesise larger parts of the learned behaviour. **Sketch synthesis**, based on, for example, a hybrid of search and continuous relaxation, would enable us to synthesise larger portions of the unknown code, leaving less of coding to the user and more to the model. **Integrating sketching with non-differentiable transitions**, such as those arising in interaction with a real environment can open up quick mastery of otherwise difficult-to-learn tasks, such as computation, reasoning and optimisation, in the reinforcement learning setting. Notably, we see differentiable sketch programming as a promising approach in Hierarchical Reinforcement Learning as sketching can enable learning parts of machines in Hierarchies of Abstract Machines [Parr and Russell, 1998]. Finally, we see **Natural Language Processing (NLP) as an application domain** which can particularly benefit from sketching as tasks in machine reading, numerical reasoning and knowledge base inference are particularly amenable to the sketching approach.

Chapter 4

gNTP: Greedy Neural Theorem Provers

Automated reasoning over real-world data, such as KBs and natural language is an essential challenge for AI and NLP [Craven et al., 1998, Etzioni et al., 2006, Banko et al., 2007]. Standard symbolic reasoning approaches provide a sound way to reason with knowledge, though at a computational cost and the need to formalise knowledge symbolically [Green and Raphael, 1968, Winograd, 1972]. Learning approaches such as NNs, on the other hand, provide a way to efficiently learn from raw data, though at the expense of transparency and difficulties with systematic generalisation [Marcus, 2018, Chollet, 2019, Marcus, 2020]. Neuro-symbolic integration promises principled integration of these two approaches that enable robust learning and structured reasoning with what is learned [Garcez et al., 2019]. One such model, a differentiable Datalog interpreter NTP [Rocktaschel and Riedel, 2017], blends the strong inductive bias of symbolic reasoning with representation learning. However, the NTP relies on a computationally demanding continuous relaxation of declarative (logic) interpretation. This relaxation renders NTPs infeasible to use for but small KBs and further disheartens their application to natural language.

In order to scale NTPs to larger KBs, we analyse the computational bottlenecks of their proof generation and present a pruning strategy to cut down their time and memory complexity drastically. This gain in efficiency, in turn, drives us to consider further expanding NTPs to utilise compositional aspects of language to reason over KBs and natural language texts jointly. The resulting model, dubbed gNTP, can scale to large KBs to perform deductive-like reasoning on

learned dense representations of both KBs facts and natural language texts.

4.1 Scaling Reasoning as a Strong Inductive Bias

The benefit of neuro-symbolic systems is their ability to capitalise on the complementary strengths and weaknesses of both neural and symbolic models, inheriting the best of both worlds [Garcez et al., 2012]. Symbolic models are easily interpretable and can strongly generalise from a small number of examples, but are brittle and prone to failure in noisy and ambiguous environments, such as natural language and real-world KBs [McDermott, 1987]. Neural models, on the other hand, are robust to noise and ambiguity, but are rarely interpretable and do not generalise well outside of the training distribution. Recent neuro-symbolic systems [Garcez et al., 2019] enable learning dense representations of symbols, allowing for ambiguous and non-discrete comparison of symbols, maintaining interpretability and strong generalisation [Marcus, 2018]. A notable model in this class of systems is the Neural Theorem Prover.

NTPs [Rocktaschel and Riedel, 2017] are an end-to-end differentiable reasoning model, effectively a differentiable interpreter for a DATALOG-alike [Roussel, 1975, Ceri et al., 1989] logical language, based on a continuous relaxation of the backward chaining algorithm [Russell and Norvig, 2009]. Following the structure of the interpreter, defined in Section 2.3.2, NTPs follow the backward chaining algorithm (Section 2.2.2) and soft unification to build a full proof-tree for proving a goal G in a given KB. The backward chaining proof path construction enables NTPs their strong inductive bias of symbolic reasoning, while the continuously relaxed unification operator enables end-to-end learning of dense representations for symbols. This structure enables NTPs to learn interpretable rules from data and makes them explainable as both proof paths and induced rules are interpretable.

However, though the continuous relaxation of the underlying logic (declarative) interpreter is what enables all these desirable properties, it is also at the crux of their inability to scale to large datasets. During both training and inference, NTPs need to compute all possible proof paths needed for proving a goal, relying on the continuous unification of the goal with *all* the rules and facts in the KB, as opposed to the standard unification which unifies goals only with

compatible rules and facts. This often becomes infeasible for even medium-sized datasets, as the number of proof paths grows exponentially. This issue can be even more dramatic when considering applying NTPs on facts expressed in natural language, as one can easily collect many more facts expressed in natural language than in regular KBs. Adding facts and rules expressed in natural language can thus drastically increase the size of the KBs, making scaling somewhat of a prerequisite for applying NTPs to reasoning over natural language. Finally, NTPs are interpretable as they produce readable induced rules. However, proof paths produced are interpretable too and they are highly indicative of how the model utilises the induced rules. This utilisation of rules by proof paths is another element of models' interpretability that has not yet been analysed.

We pose the following research questions:

- Can we overcome the scalability issue of NTPs and scale them up to large datasets without sacrificing their evaluation performance?
- How can we make NTPs reason with KBs and natural language texts?
- Is the interpretability of NTPs as useful as presented by Rocktaschel and Riedel [2017]?

We answer these questions by showing that:

- NTPs scores prefer closest representations of unified atoms to goals, so we can radically reduce the computational complexity of both inference and learning by only unifying goals with the k-nearest neighbouring atoms, preferring the most promising proof paths instead of enumerating them all.
- an attention mechanism further lowers down the representational complexity of rule-learning and helps to achieve better evaluation performance.
- employing these two techniques improves both the memory and the runtime performance of NTPs while performing as well or better than NTPs on small link prediction tasks.[1]
- integrating a compositional language reader which represents natural lan-

[1] Also known as Knowledge Base Completion tasks; tasks of automatic inference of missing facts in KBs.

guage as predicates by composing word representations enables NTPs to incorporate textual knowledge in KBs and jointly learn the representations of text in the same embedding space as the representations of symbols.

- a qualitative analysis of proofs and induced rules reveals that decoding induced rules with a 1-nearest neighbour can result in erroneous interpretations.

The contributions in this chapter are the following: i) we present gNTP, an efficient NTP model which significantly reduces the time and space complexity requirements by greedily reducing the number of candidate proof paths and lowering the number of parameters for rule learning with the attention mechanism, ii) we extend gNTP with a compositional reading module which jointly embeds predicates and natural language texts in the same embedding space, iii) we experimentally show that gNTP perform on par with, or better than NTP at a fraction of the cost, and can achieve competitive link prediction results on large-scale datasets while being able to provide interpretable explanations for each prediction and iv) that decoding induced rules with the method presented in Rocktaschel and Riedel [2017] can even be misleading, and that best-ranking proof paths too should be used when interpreting the induced rules.

4.2 Background: Neural Theorem Provers

Logic program interpreters can give answers to queries when the answers can be logically deduced from the KB, relying on unification for inspecting equality between symbols compared during the deduction. For cases where symbols are not the same but might support the concept of similarity, standard logic programming fails. Rocktaschel and Riedel [2017] introduced NTPs as a way to expand logic programming with the ability to deduce over symbols based on their similarity, importantly, by learning representations of KB symbols from data.

NTPs are a class of neural network models for end-to-end differentiable deductive reasoning. They mimic the backward chaining proving strategy (Section 2.2.2) by recursively building a neural network which enumerates all possible proof paths for proving a goal over the KB, up to a specific

proof depth, making the enumeration effectively a depth-limited Breadth-First Search (BFS). To support learning representations in a KB $\mathfrak{K}$, NTPs expand a given KB with a set of parameters, $\mathfrak{K}^{\theta} = (\mathfrak{K}, \theta)$, where θ stands for a learned d-dimensional vector representation for each constant and predicate symbol in KB, indexed by the symbol, e.g. $\theta_{\texttt{locatedIn}} \in \mathbb{R}^d$.

4.2.1 Continuous Relaxation of Backward Chaining

The recursive enumeration of all proof paths by NTPs relies on three modules for building this neural network, derived from same-named modules in Section 2.2.2; the unification module `unify`, which compares sub-symbolic representations of symbols, and mutually recursive `or` and `and` modules, which jointly enumerate all the possible proof paths, before the final aggregation selects the single, highest-scoring state.[2]

We briefly overview these modules and the training procedure in the following. We follow the notation presented in the previous section while simplifying the notation with a partial function definition, treating failure as the omission of the function definition.

Unification module In backward chaining, unification [Herbrand, 1930, Robinson, 1965] is the process of finding substitutions that make two logical expressions look identical [Russell and Norvig, 2009] as a means of checking whether they can represent the same structure. Symbolic unification checks for equality between the corresponding elements of two atoms, e.g. atoms `districtIn(X,Z)` and `districtIn(BLOOMSBURY,LONDON)` unify as their predicates are equal, and the variables can be bound to symbols via the substitution $\{\text{X}/\text{BLOOMSBURY}, \text{Y}/\text{LONDON}\}$. This is a strict process that does not support the notion of possible similarity between atoms, e.g. `locatedIn`, could be similar to `situatedIn` or UNITED_KINGDOM could be similar to GREAT_BRITAIN.

In lieu of the symbolic unification in Equation (2.20), NTPs employ continuous unification, which enables comparing different symbols with similar semantics, by comparing their vector representations with a similarity function, e.g. $\texttt{sim}(\texttt{locatedIn}, \texttt{situatedIn}) = f(\theta_{\texttt{locatedIn}}, \theta_{\texttt{situatedIn}})$. This continuous

[2] Though NTPs would, as would the backward chaining, return all proof paths, the model assumes that only the highest-scoring proof path is the valid one, hence the final proof aggregation.

unification makes comparing ground atoms and applying rules possible even in cases where discrete symbols differ:

$$\texttt{unify}_{\mathfrak{K}_\theta}(H, G, S) = \begin{cases} S & \text{if } H = G = [] \\ \texttt{unify}_{\mathfrak{K}_\theta}(H', G', S') & \text{if } |H| = |G|, H = [h : H'], G = [g : G'] \end{cases}$$
$$\text{where}$$
$$S = (\psi, \rho)$$
$$S' = (\psi', \rho')$$
$$\psi' = \texttt{unify-var}(h, g, \psi)$$
$$\rho' = \min(\rho, \texttt{sim}(h, g, \mathfrak{K}_\theta)) \,. \tag{4.1}$$

Note that now, the proof state $S = (\psi, \rho)$ consists of two elements: the substitution ψ, as in backward chaining, and the proof score ρ which quantifies the agreement of two substitutions. This reflects on the `unify-var` function which does not lead to unification failures as in Equation (2.21), but allows comparison of all expressions:

$$\texttt{unify-var}(h, g, \psi) = \psi \cup \begin{cases} \{h/g\} & \text{if } h \in V \\ \{g/h\} & \text{if } h \notin V, g \in V \\ \emptyset & \text{otherwise} \end{cases} \,. \tag{4.2}$$

The continuous unification operator now calculates a proof score ρ for each proof path by relying on a similarity function `sim` defined with a Radial Basis Function (RBF) kernel [Broomhead and Lowe, 1988]:

$$\texttt{sim}(h, g, \mathfrak{K}_\theta) = \begin{cases} \exp\left(\frac{-\|\theta_h - \theta_g\|^2}{2\sigma^2}\right) & \text{if } h, g \notin V \\ 1 & \text{otherwise} \end{cases} \,, \tag{4.3}$$

where θ_g and θ_h are representations from $\mathfrak{K}_\theta$ corresponding to symbols h and g. We can see that, by design, the unification of a variable and a constant does not influence the proof score (Equation (4.2)), while the unification between two different symbols influences the score if the symbol similarity is lesser than the current proof score (Equation (4.1)).

For example, given atoms `locatedIn(`LONDON`,`UK`)` and `situatedIn(`X`,`Y`)`,

and a proof state $S = (\psi, \rho)$, the unify module calculates the similarity of embedding representations $\theta_{\texttt{locatedIn}}$ and $\theta_{\texttt{situatedIn}}$ with a RBF kernel, updates the substitution set $\psi' = \psi \cup \{\text{X}/\text{LONDON}, \text{Y}/\text{UK}\}$, and calculates the new proof score as $\rho' = \min(\rho, \texttt{sim}(\theta_{\texttt{locatedIn}}, \theta_{\texttt{situatedIn}}))$.

OR module The or module aims to prove the goal by unifying it with every fact and rule in the KB $\mathfrak{K}_\theta$.

Formally, as in Equation (2.18), or unifies the goal G with the head H of each rule $H :- B \in \mathfrak{K}$, and calls the and module with the resulting substitution to prove atoms in the body B of each rule:

$$\texttt{or}_{\mathfrak{K}_\theta}(G, d, S) = \left[S' \;\middle|\; \begin{array}{l} H \;:-\; B \in \mathfrak{K}_\theta \\ S' \in \texttt{and}_{\mathfrak{K}_\theta}(B, d, \texttt{unify}_{\mathfrak{K}_\theta}(H, G, S)) \end{array} \right]. \tag{4.4}$$

For example, given a goal $G = \texttt{situatedIn}(\text{LONDON}, \text{UK})$, and the rule $\texttt{locatedIn}(\text{X}, \text{Y}) :- \texttt{districtIn}(\text{X}, \text{Z}), \texttt{capitalOf}(\text{Z}, \text{Y})$, the model would unify the goal with the head $\texttt{locatedIn}(\text{X}, \text{Y})$ of the rule, and instantiate and modules, to prove sub-goals in the body $\texttt{districtIn}(\text{X}, \text{Z}), \texttt{capitalOf}(\text{Z}, \text{Y})$ of the rule .

AND module The and module, in turn, aims to recursively prove a list of sub-goals such as body atoms in a rule. Concretely, given a list of sub-goals, $G = [g : G']$, the and module will apply the substitution ψ to the sub-goal g, and call the or module to further unify it with the rules in the KB. The resulting state of the unification is the starting state to further recursively prove the rest of the list, the tail G' by invoking the and module on it, as in Equation (2.19):

$$\texttt{and}_{\mathfrak{K}_\theta}(G, d, S) = \begin{cases} S & \text{if } G = [\,] \\ \left[S'' \;\middle|\; \begin{array}{l} S' \in \texttt{or}_{\mathfrak{K}_\theta}(\psi\ g, d-1, S) \\ S'' \in \texttt{and}_{\mathfrak{K}_\theta}(G', d, S') \end{array} \right] & d > 0, G = [g : G'] \end{cases}. \tag{4.5}$$

For example, when invoked on the rule body B from the example before, the and module will first substitute variables with constants for the sub-goal $\texttt{districtIn}(\text{X}, \text{Z})$ and invoke the or module on it. Starting with the resulting state, and is invoked on $\texttt{capitalOf}(\text{Z}, \text{Y})$.

We depict the interplay of all these modules in the NTP model in an example of a run on the KB in Listing 2.1 on the goal $\texttt{situatedIn}(\text{BLOOMSBURY}, \text{UK})$

in Figure 4.1. Note that now, in contrast with the backward chaining example in Figure 2.3, the mutual recursion of the `and` and `or` in NTP produces a significantly larger proof space, where the only failure encountered (depicted for comparison) is the result of the loop avoidance [Van Gelder, 1987].

NTP as Differentiable Logic Interpreter Following the exposition in Section 2.3.2, per Equation (2.28), NTP is a differentiable logic interpreter:

$$\Upsilon_{ntp}(\mathfrak{K}_\theta, G, d) = \max\{\rho \mid (\psi, \rho) \in \texttt{or}_{\mathfrak{K}_\theta}(G, d, (\emptyset, 1)\}\,, \tag{4.6}$$

with the initial proof state set to $(\emptyset, 1)$, an empty substitution set and a starting proof score of 1. The execution of the `or` function results in the enumeration of all the proof paths of the goal G on a KB $\mathfrak{K}_\theta$, to a pre-specified depth d. NTPs assume that only the highest-scoring proof path is the correct one, hence the maximisation over the scores ρ.

4.2.2 Training

We can learn the parameters θ of the predicate and constant representations by optimising the binary cross-entropy loss on the final proof score. By iteratively masking facts in the KB and trying to prove them using all the other available facts and rules [Rocktaschel and Riedel, 2017] we get:

$$\begin{aligned}\mathcal{L}_{\mathfrak{K}_\theta}(\mathfrak{D}) &= \sum_{(G,y)\in\mathfrak{D}} -y\log(\Upsilon_{ntp}(\mathfrak{K}_\theta, G, d)) - (1-y)\log(1-\Upsilon_{ntp}(\mathfrak{K}_\theta, G, d)) \\ &= \sum_{(G,y)\in\mathfrak{D}} H(y, \Upsilon_{ntp}(\mathfrak{K}_\theta, G, d)) + H(1-y, 1-\Upsilon_{ntp}(\mathfrak{K}_\theta, G, d))\,,\end{aligned} \tag{4.7}$$

where $\mathfrak{D} = \{(G, y)\}$ is the dataset containing pairs of goals G and labels y, where $y = 1$ for the goals present in the KB $\mathfrak{K}_\theta$. The negative examples, in this case, are sampled from the positive ones by corrupting the entities and setting $y = 0$.

The original NTP model is computationally demanding, so Rocktaschel and Riedel [2017] implement two optimisations to speed up training—batch processing of several proofs in parallel, and limiting the number of possible unifications of free variables in rules with a differentiable K-max heuristic [Rocktaschel and Riedel, 2017].

Note that, strictly speaking, though NTPs are continuous, they are not dif-

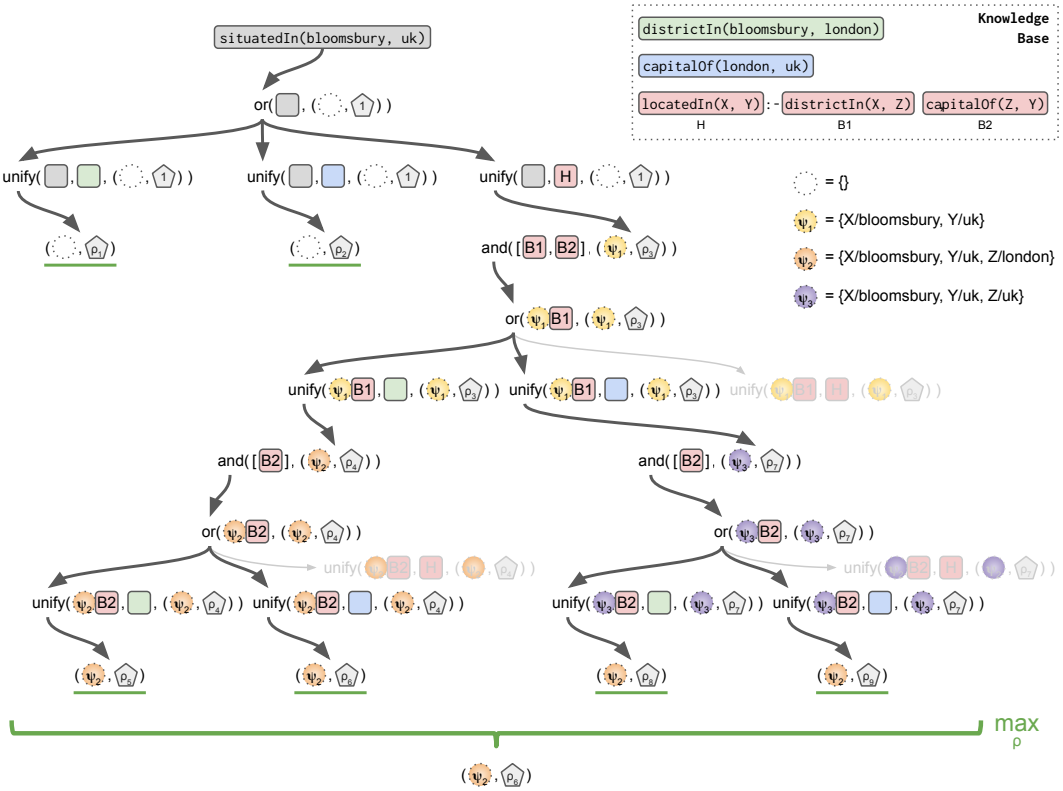


Figure 4.1: An example of the execution of NTP on a small knowledge base. The knowledge base, presented in Listing 2.1, is also noted in the top right corner. Circles signify substitutions, squares atoms (with circles next to squares signifying applying a substitution to an atom) and pentagons signify the output scores. The colour codings of the KB and the unifications follow through the example. We omit some calls to and for clarity. The calls resulting in failure are transparent to accentuate that the algorithm ignores them, but we present them for comparison to Figure 2.3. As opposed to the linearly recursive call structure in $\partial 4$, we can note the fully recursive structure of calls in NTP. Note the significantly larger number of proof paths that the algorithm maximises over, compared to Figure 2.3—this is what we aim to improve.

ferentiable since the min and max functions used are not differentiable everywhere. However, they do have subgradients that we can use to propagate the gradient information through the extrema of these functions.[3]

Logic Program Induction The ability to learn continuous representations of predicates opens the possibility of learning interpretable rules from data. Rocktaschel and Riedel [2017] show that manually specifying rule templates such as `#p(X,Y) :- #q(X,Z),#r(Z,Y)`, where `#p,#q,#r` denote learned predicates with $\theta_{\texttt{\#p}}, \theta_{\texttt{\#q}}, \theta_{\texttt{\#r}}$ as free parameters, it is possible to learn rules from data. They show that the parameters can be learned from data and simply decoded for inspection by searching the closest representation of each parameter in the space of all predicate parameters.

4.3 Greedy Neural Theorem Provers

The NTPs, as presented in Section 4.2, are end-to-end differentiable models capable of deductive reasoning, learning representations, program synthesis and they provide explainable predictions. In theory, this makes them an excellent candidate for models for theorem proving over text. In practice, however, the representation learning capacity of these models comes at a huge cost—computational intractability for all but the small KBs.

In this section, we present gNTPs, models which i) attack the issue of computational intractability by employing a heuristic to filter out proof paths deemed unnecessary, thus making the model scale to larger datasets ii) incorporate a compositional language reader making them readily applicable to text-enriched datasets, as a step towards natural language reasoning.

4.3.1 Scaling up NTPs

In general, the inability of NTPs to scale to bigger datasets stems from three interconnected causes. First, NTPs inherit the complexity of the backward chaining algorithm, usually implemented with the DFS search strategy. Second, the inherited complexity is additionally exacerbated by the continuous unification which, instead of checking for equality, needs to calculate the compatibility of all symbols. Finally, this is yet worsened with the need to utilise

[3]The gradient is passed through the path of the minimum/maximum element, respectively.

the (depth-limited) BFS strategy to enumerate all the proof paths, as a prerequisite to simplified Graphics Processing Unit (GPU) computation and efficient batching. The combination of these three issues creates a significant computational bottleneck in the model. Here, we focus on solving the issue of continuous unification as a core computational issue introduced by the model.

As opposed to a discrete theorem prover, where the unification stops non-comparable symbols to be unified and further explored, the continuous unification allows comparison of all symbols, implying that a simple equality check now expands to a similarity calculation to all elements in the KB. Concretely, for each (sub-)goal, the process of enumeration and scoring all bounded-depth proof paths requires unifying the (sub-)goal with all the representations of all rule heads and facts in the KB. Furthermore, the expansion of rules with more than one atom in the body causes an increase of the sub-goals to prove (paired with the unification increase), both because all atoms in the body need to be proven, and because rules (e.g. `locatedIn(X,Y) :- districtIn(X,Z),capitalOf(Z,Y)`) may contain newly introduced free variables which are not present in the head of the rule, and these variables additionally complicate the calculation as instead of binding them to the same value, their binding needs to be calculated by the unification. Even though Rocktaschel and Riedel [2017] addressed this issue with the differentiable k-max, the differentiable k-max still requires an on-the-spot calculation of all the scores before choosing only the top-ranking ones.

We propose implementing a heuristic approach that tackles the complexity of continuous unification, thus mitigating the computational costs of NTPs unification, making them readily applicable to large datasets, and opening up the possibility of applying them to text-enriched data. We dub these models gNTPs, given the greedy nature of the heuristic.

We implemented gNTPs in Tensorflow Eager [Agrawal et al., 2019] and made it freely available under the MIT license at `https://github.com/uclnlp/gntp`.

4.3.1.1 Greedy Unification

We analyse the issues of continuous unification on two fronts, the unification of goals with facts, and the unification of goals with atoms in rules (i.e. rule selection).

Greedy Fact Unification The number of facts in real-world KBs can be large [Paulheim, 2017], for example, Freebase [Bollacker et al., 2008] contained[4] 637 million facts [Dong et al., 2014], while the Google Knowledge Graph contains 18 billion facts [Nickel et al., 2015] . This is why NTP-style unification of sub-goals with each one of the facts is simply intractable on such huge datasets. Concretely, let us assume a simple parametrised KB $\mathfrak{K}_\theta$ composed of n facts and no rules. Given a query G over a rule-free $\mathfrak{K}_\theta$, and following Equation (4.5) and Equation (4.4), NTP reduces Equation (4.6) to the following problem:

$$\mathcal{Y}_{ntp}(\mathfrak{K}_\theta, G, 1) = \max\{\rho \mid (\psi, \rho) \in \texttt{unify}_{\mathfrak{K}_\theta}(F, G, (\emptyset, 1)), F \in \mathfrak{K}_\theta\}, \tag{4.8}$$

that is to say, to find the maximum scoring path, NTP needs to find a single fact $F \in \mathfrak{K}_\theta$, that, when unified with the goal G yields the maximum unification score. This implies that NTPs will compute the unification score between the sub-goal G and every fact $F \in \mathfrak{K}_\theta$ in order to find the one which corresponds to the maximum score. Given that this scales linearly with the number of facts in the KB for all the leaves of the proof tree, this is computationally prohibitive for large datasets.

Moreover, NTPs will only return a single highest proof score. This means that during training, NTPs update parameters only along the path of the highest proof score. During inference, NTPs provide only the highest proof score.

The insights that finding the maximum scoring path reduces to finding the fact F closest to the goal G, the notion that we need to obtain only a single proof score and the fact that the similarity used in the unification is calculated via a RBF kernel, suggests we can cast this problem as a Nearest Neighbour Search (NNS) problem [Fix, 1951, Bentley, 1975, Yianilos, 1993]. Utilising NNS is feasible since the RBF kernel, used by the NTPs is monotonically decreasing with the increasing Euclidean distance between the goal G and the fact F. Identifying the closest fact F to the goal G would permit the reduction of the number of path computations from $\mathcal{O}(|\mathfrak{K}_\theta|)$ to $\mathcal{O}(1)$, not factoring in the cost of the NNS search, thus casting the problem in Equation (4.8) to:

$$\mathcal{Y}_{ntp}(\mathfrak{K}_\theta, G, 1) = \max\{\rho \mid (\psi, \rho) \in \texttt{unify}_{\mathfrak{K}_\theta}(F, G, (\emptyset, 1)), F \in \mathcal{N}_k^{\mathfrak{K}_\theta}(G)\}, \tag{4.9}$$

that is, given the goal G we restrict the search for the closest fact F to a

[4] Freebase was supplanted by Wikidata from May 2016.

Euclidean local k-neighbourhood $\mathcal{N}_k^{\mathfrak{K}_\theta}(G)$ defined as:

$$\mathcal{N}_k^{\mathfrak{K}_\theta}(G) = \text{k-}\arg\min_{F \in \mathfrak{K}_\theta} \|\theta_F - \theta_G\| . \tag{4.10}$$

This problem in Equation (4.8) is equivalent to the problem in Equation (4.9), but we still need to be able to calculate the k-nearest neighbours $\mathcal{N}_k^{\mathfrak{K}_\theta}(G)$ efficiently to benefit from this. Here, we propose to efficiently compute the nearest neighbours between a sub-goal G and facts $F \in \mathfrak{K}_\theta$ with fast NNS algorithms. However, finding nearest neighbours is a difficult task—due to the curse of dimensionality [Bellman, 2015], finding the exact neighbourhood of a point in a Euclidean space is very costly [Indyk and Motwani, 1998]. Experiments showed that methods for identifying the exact neighbourhood can rarely outperform brute-force linear scan methods when the dimensionality is high [Weber et al., 1998], and that the high dimensionality exacerbates possible issues of the quality of the NNS results [Beyer et al., 1999, Hinneburg et al., 2000]

One practical solution is to utilise Approximate Nearest Neighbour Search (ANNS) algorithms, which focus on finding an approximate solution to the k-NNS problem. Several families of ANNS algorithms exist, such as Locality-Sensitive Hashing [Andoni et al., 2015], Product Quantisation [Jegou et al., 2011, Johnson et al., 2017] and Proximity Graphs [Malkov et al., 2014]. One of the most promising approaches is the Hierarchical Navigable Small World [Malkov and Yashunin, 2018], a graph-based incremental ANNS structure which offers significantly better logarithmic complexity scaling during neighbourhood search than other approaches [Li et al., 2019]. In our first series of experiments, we experimented with this technique as a part of the `nmslib`[5] library.

Another practical solution is to utilise a fast, exact NNS algorithm, executed on a GPU, such as Facebook AI Similarity Search (FAISS) [Johnson et al., 2017].[6] Since optimised for GPU computation, this library provides exact results in less time than the approximate methods, for large enough datasets (up to 1 million facts). However, for more than 1 million facts, it would be advisable to use ANNS algorithms, some of which are also implemented on a GPU, such as the GPU implementation of the Product Quantisation in FAISS

[5] https://github.com/nmslib/nmslib
[6] https://github.com/facebookresearch/faiss

In this work, we opt for the exact NNS with GPU support with FAISS for efficient NNS. We chose the exact one simply because the GPU-supported NNS is significantly faster than the CPU-based ANNS for the sizes of problems we deal with here. For even larger datasets, however, the exact NNS stops being a viable option.

Notice one important remark here: both the ANNS and NNS models build an indexing structure for the particular instance of the KB, and the cost of that index building is often higher than the cost of querying it. To offset this cost, we chose to re-build the index every i-th batch. This choice to act upon stale information from a stale index invalidates the equivalence of Equation (4.8) and Equation (4.9). However, we assume that small updates by stochastic gradient descent would not necessarily invalidate previous search indexes and thus support the choice of a heuristic on stale information.

Greedy Rule Selection Analogous to facts, we can extend the same procedure to select which rules to activate for proving a given goal as well. For example, given a goal `situatedIn(LONDON,UK)` and a rule head H `locatedIn(X,Y)`, it is sensible to expand said rule if there is high similarity between the embedding $\theta_{\texttt{locatedIn}}$ and $\theta_{\texttt{situatedIn}}$. Strictly speaking, this local decision of choosing a rule with a closer predicate (or predicate and one atom, in the case of sub-goals with one variable assigned), is a greedy decision which does not guarantee the highest scoring path in the end. However, although the rule chosen due to the high similarity to the goal may lead to a sub-optimal proof path, we empirically observed that unifying the goal with the closest rule heads is likely to generate high-scoring proof paths.

Specifically, in our implementation, we generate a set of NNS indexes corresponding to a partitioning $\text{Part}(\mathfrak{K}_\theta)$ of the KB where each element of the partition groups all facts and rules in $\mathfrak{K}_\theta$ sharing the same signature. For example, all facts such as `locatedIn(LONDON,UK)` and `situatedIn(LONDON,UK)` share the same index, as well as all atoms of the form `p(X,A)` and `q(Y,B)`.[7]

Having both the fact and rule unification cast as a NNS problem, and having indexes for NNS both the fact and rule selection defined through a partitioning

[7] In total, the number of these partitions equals at most 2^v, where v is the number of variables in an atom.

$\text{Part}(\mathfrak{K}_\theta)$ of the KB, allows us to redefine the `or` module as:

$$\texttt{or}_{\mathfrak{K}_\theta}(G, d, S) = \left[S' \;\middle|\; \begin{array}{l} H \;\texttt{:-}\; B \in \mathcal{N}_k^{\text{Part}(\mathfrak{K}_\theta)}(G) \\ S' \in \texttt{and}_{\mathfrak{K}_\theta}(B, d, \texttt{unify}_{\mathfrak{K}_\theta}(H, G, S)) \end{array} \right], \tag{4.11}$$

where instead of unifying a goal G with all facts and rule heads in the KB, we constrain the unification with ANNS to only facts and rule heads in the local neighbourhood of the goal $\mathcal{N}_k^{\text{Part}(\mathfrak{K}_\theta)}(\mathfrak{K}_\theta)$.

4.3.1.2 Attention

We can use NTPs to learn interpretable rules [Rocktaschel and Riedel, 2017]. However, since the number of parameters associated with predicates is the same as the number of parameters associated with constants, rule learning can be quite inefficient in cases where the number of predicates is smaller than the number of dimensions. For example, considering a rule example `#p(X,Y) :- #q(X,Z),#r(Z,Y)`, we observe that the number of parameters in it is three times the size of the embeddings d, and learning them directly can take time, given that the model needs to find a good set of parameters in the full d-dimensional space.

Given that the rules would ideally employ embeddings of existing predicates, we propose to constrain the number of parameters in predicate embeddings by using an attention mechanism [Bahdanau et al., 2015] over already known predicates in the KB.

Concretely, given $\mathcal{P}$, a set of known predicates in the KB, $\mathfrak{K}_\theta^{\mathcal{P}}$, the subset of the KB corresponding to $\mathcal{P}$, and $\mathbf{P}$, a matrix of embeddings of $\mathcal{P}$, we re-define the predicate parameters $\theta_{\mathcal{P}}$ per Equation (2.15) as:

$$\theta_{\mathcal{P}}^{\intercal} = \text{softmax}(\hat{\theta}_{\mathcal{P}})^{\intercal}\mathbf{P}\,, \tag{4.12}$$

where $\hat{\theta}_{\mathcal{P}}$ are the (new) attention parameters associated with predicates $\mathcal{P}$ (one value per predicate).

The attention has a twofold effect here. First, it can improve the parameter efficiency of the model by lowering the number of parameters for training rules in cases where the number of known predicates is lower than than the embedding size d, by introducing $c|\mathcal{P}|$ parameters for each rule rather than cd, where c is the number of trainable predicate embeddings in the rule. Second,

it has a constraining effect on the trainable embeddings as it drives the newly learned embedding to a convex hull of known predicates in the KB.

4.3.2 Joint Reasoning on Knowledge Bases and Natural Language

Natural language is characterised by compositionality, a property crucial to production and understanding of a large (seemingly infinite) number of sentences with a limited number of words [Katz and Fodor, 1963, Davidson, 1967, Grandy, 1990]. In short, the principle of compositionality dictates that meaning of a unit of text is a function of the meaning of its constituent words and the rules that combine them [Cresswell, 1973, Partee, 1995]. In the representation learning setting, this is often modelled by representing sentences as functions of representations of its elements.

NTPs do not use compositional representations; they entirely rely on monolithic representations of symbols, hence any representation of relations or entities from a piece of text would rely on representing the full text as a single symbol. Given the compositional character of language, this seems like a wasted opportunity. For example, sentences like "London is located in the UK" and "London is in the UK" obviously both testify to a relationship between entities LONDON and the UK in the KB, and share much informational content, which would not be captured by a monolithic representation.

This inspired us to expand gNTPs to support reasoning with textual knowledge using compositional language representation.

We focus on representing knowledge in the form of *textual facts*, treating the text between the entities in the KB as a *relation mention* [Mintz et al., 2009], assuming that sentences mentioning two entities express a relation between them [Hoffmann et al., 2011]. For example, we represent "Bloomsbury is a quarter of London" with a relation mention `"[#1] is a quarter of [#2]"` and entities BLOOMSBURY and UK in a textual fact `"[#1] is a quarter of [#2]"`(BLOOMSBURY,LONDON).[8] Other than using the text between these two entities, we can also use other information representations such as whole sentences [Riedel et al., 2010], or derivative information such as parts of the dependency parse [Riedel et al., 2013].

[8] `[#1]` and `[#2]` denote placeholders for the first and the second entity/argument of the textual fact.

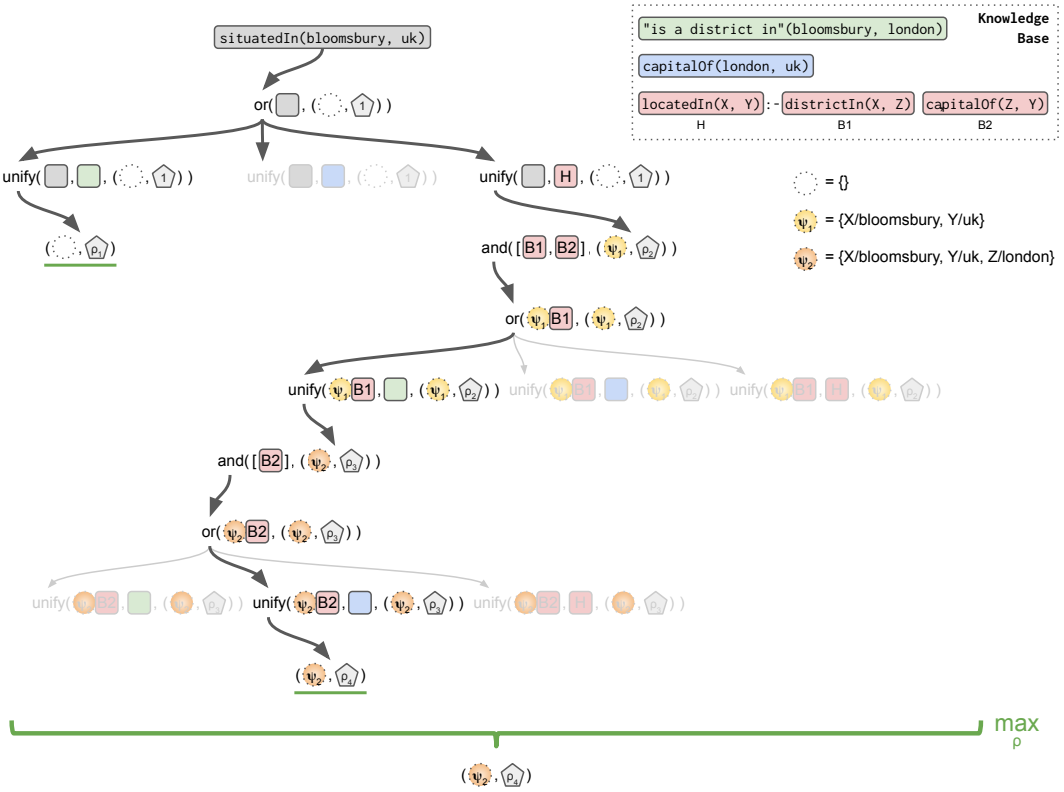


Figure 4.2: An example of the execution of gNTP on a small knowledge base, presented in the top right corner. Circles signify substitutions, squares atoms (with circles next to squares signifying applying a substitution to an atom) and pentagons signify the output scores. The colour codings of the KB and the unifications follow through the example. We omit some calls to `and` for clarity. The calls resulting in `FAIL` are transparent to accentuate that the algorithm ignores them, but we present them for comparison to Figure 4.1. Note that the model maximises over only two proof paths in this case. The difference is even more pronounced on larger KBs.

To that end, we expand the KBs with textual facts, by expanding standard predicates with relation mentions. The standard predicates are encoded by a look-up into the parameters θ, whereas relation mentions are encoded with a compositional reading module `reader` which maps a sequence of token representations into the same d-dimensional space of the original predicate. By composing token representations into a representation of the mention, we effectively model the compositional aspect of language [Mikolov et al., 2013].

Formally, given a relation mention m, and the knowledge base $\mathfrak{K}_\theta$ with parameters expanded with a set of token embeddings, the `reader` encodes the

surface pattern m as a mean of embeddings of tokens $t \in m$:

$$\texttt{reader}_{\mathfrak{K}_\theta}(m) = \frac{1}{|m|} \sum_{t \in m} \theta_t \,. \tag{4.13}$$

Although `reader` can be implemented by any parametrised differentiable architecture, such as CNNs [Kalchbrenner et al., 2014], RNNs [Mikolov et al., 2010] or a transformers [Vaswani et al., 2017], we opted for a simple parameter-free averaging model for the sake of simplicity and efficiency. It is worthwhile noting that, albeit simple, the averaging model has been shown to perform on par or even better than other, more elaborate models, thanks to a lower tendency to overfit to training data [White et al., 2015, Arora et al., 2017, Mitchell et al., 2018]. We leave more elaborate parametrised models for future work.

An example of a gNTP run on a text-enriched KB can be seen in Figure 4.2. Note the drastic reduction in complexity, when compared to NTP in Figure 4.1, resulting in a smaller proof tree.

4.4 Experiments

Given that the goal of modelling gNTPs is to deal with computational intractability of NTPs and make them applicable to textual data, we hypothesise that gNTPs:

H1 perform similarly to NTPs on smaller datasets, where they can be directly compared

H2 are more time and memory performant than NTP

H3 scale to large datasets, as a direct consequence of H2, and if they do, we aim to investigate how they compare to state-of-the-art models

H4 can utilise the newly-added ability to deal with textual data, and we aim to see whether they successfully use the added benefit of the aforementioned data

H5 can provide interpretable rules and proofs, useful for qualitative analysis

We test these hypotheses via an extensive evaluation of gNTPs models on multiple tasks, splitting them by the dataset size to small and large datasets.

We start with four small datasets of varying sizes and complexities, on which we quantify the performance of gNTPs compared to the performance of NTPs (H1). Concurrently, we compare gNTPs with other state-of-the-art link prediction models as well as with related, continuous-logic and path-based models. Afterwards, we compare the time and memory use of gNTPs and NTPs on the same small datasets, given that these datasets present the limit of usage for NTPs (H2). Next, we test the scaling capabilities of gNTPs by running them on three large datasets and comparing them to multiple state-of-the-art models and baselines (H3). We interleave small and large datasets experiments with qualitative analyses, to analyse the use of interpretable rules and proofs (H5). Finally, we conclude the experimental section with the analysis of gNTP on the text-enriched Countries datasets (H4).

4.4.1 Datasets, Evaluation and Baselines

4.4.1.1 Datasets

To compare gNTP with NTP and state-of-the-art link prediction models, we use the same datasets as Rocktaschel and Riedel [2017], namely the Countries S1, S2, S3, Nations, Kinship and UMLS datasets. Due to their humble size, we refer to these datasets as *small datasets*. **Countries** is the Nickel et al. [2016b] version of the Bouchard et al. [2015] dataset, intended as a benchmark dataset for testing long-range reasoning capabilities of link prediction models. It is a dataset of 1158 facts about 244 countries, denoting relationships of neighbourhood and affiliation (2 relations) to a hierarchy of 23 subregions and 5 regions (countries, subregions and regions being the entities) of the world. The dataset is split based on countries so that each country in the development (20 countries) and the test (20 countries) set has at least a neighbour in the train (204 countries) set. This is the basis for three progressively harder versions of this dataset. **Countries S1** is missing the region affiliation of test set countries, making it solvable by utilising the subregion information with the trainsitivity rule `locatedIn(X,Y) :- locatedIn(X,Z),locatedIn(Z,Y).` **Countries S2**, in addition to S1, is missing the subregion affiliation of test set countries, making it solvable by inferring location from neighbour information: `locatedIn(X,Y) :- neighborOf(X,Z),locatedIn(Z,Y).`[9] **Countries S3**,

[9]Neighbouring countries might not be in the same region hence this rule does not always hold

in addition to S2 is missing the region affiliation of all neighbours of all countries in the dev and test sets, making it solvable via the three-hop rule `locatedIn(X,Y) :- neighborOf(X,Z),neighborOf(Z,w),locatedIn(w,Y).`

To test the usefulness of textual capabilities of gNTP, we generated variants of the Countries datasets named **Countries with mentions** by randomly replacing a varying percentage of training set triples with a randomly chosen mention out of a set of human-generated mentions. The mentions we used are enumerated in Table B.1.

Following Countries as the benchmarking datasets, we use three datasets used in previous work [Kemp et al., 2006, Kok and Domingos, 2007] for relational learning, namely the Nations, Kinship and UMLS datasets. **Nations** [Rummel, 1976] is a political dataset, containing interactions (relations) between nations (entities).[10] **Kinship** [Denham, 1973] is a dataset containing complex kinship structures exhibited in the Australian Alyawarra tribe, containing kinship relationships (relations) between individuals (entities) of the tribe [Denham, 1973]. **UMLS** [McCray, 2003] is a biomedical ontology, presenting relationships (relations) between high-level concepts (entities).

Since gNTPs allow us to experiment on significantly larger datasets than NTPs, we use the standard large datasets, WN18, WN18RR, and FB122. **WN18** [Bordes et al., 2013] is a subset of WordNet [Miller, 1995], a lexical KB for the English language, exhibiting lexical relationships (relations) between word senses (entities). **WN18RR** [Dettmers et al., 2018] is a harder derivative of WN18, with fixed test leakage issues. **FB122** [Guo et al., 2016] is a dataset of 122 Freebase relations extracted from FB15k, coming with 47 externally induced rules, which can further be used, if the model supports using them. The rules are significant as they are the basis for the test set, which is split into two. Test-I contains triples that cannot be directly inferred by pure logical inference with the provided rules, whereas Test-II contains triples that can.

The statistics of all the datasets, both the small and the large ones are given in Table 4.1.

[10] And features of nations encoded as unary relations which have been filtered out of the dataset

Table 4.1: Dataset statistics for both the small (Countries, Nations, Kinship and UMLS) and the large datasets (WN18, WN18RR, FB15k-237).

	Dataset		# relations	# entities	# triples			
					train	dev	test	total
Small datasets	**Countries** [Bouchard et al., 2015]	**S1**	2	272	1111	24	24	1159
		S2	2	272	1063	24	24	1111
		S3	2	272	979	24	24	1027
	Nations [Rummel, 1976]		56	14	1,592	199	201	1,992
	Kinship [Denham, 1973]		26	104	8,544	1,068	1,074	10,686
	UMLS [McCray, 2003]		49	135	5,216	652	661	6,529
Large datasets	**WN18** [Bordes et al., 2013]		18	40,943	141,442	5,000	5,000	151,442
	WN18RR [Dettmers et al., 2018]		11	40,943	86,835	3,034	3,134	93,003
	FB122 [Guo et al., 2016]	**Test-I**	122	9,738	91,638	9,595	5,057	106,290
		Test-II					6,186	107,419
		Test-ALL					11243	112,476

4.4.1.2 Evaluation

We follow the standard evaluation protocols as Rocktaschel and Riedel [2017]. For Countries, this means reporting the performance in terms of the Area Under the Precision-Recall Curve (AUC-PR) [Davis and Goadrich, 2006].

For all the other datasets, we generate all possible corruptions of test fact arguments, filtering out corrupted facts that occur in the KB. We then predict the ranking of the test fact and its corruptions, and report the Mean Reciprocal Rank (MRR) [Voorhees, 2001] and the fraction of correct entities found in the top n ranked ones (HITS@n) [Bordes et al., 2013].

4.4.1.3 Baselines

NTPs We compare the performance of gNTP primarily with the performance of their predecessor, NTP. Note that the results reported in Rocktaschel and Riedel [2017] were calculated with an incorrect evaluation function—if several facts have the same score, the ranking function assigns them the same (best) rank, which artificially inflates the result. We corrected the issue and recalculated the results of their publicly available models.[11]

Neural Link Predictors Next, we compare the performance of gNTP with state-of-the-art Neural Link Predictors, DistMult [Yang et al., 2014], ComplEx [Trouillon et al., 2016] and ConvE [Dettmers et al., 2018]. These models jointly learn each entity and relation embedding with a d-dimensional vector and optimise a differentiable scoring function based on these embeddings, minimising the KB reconstruction error [Nickel et al., 2016a].

[11] https://github.com/uclmr/ntp

Neuro-symbolic models Finally, we report the results of two other neuro-symbolic reasoning systems, MINERVA [Das et al., 2017a], a reinforcement learning -based KB graph traversal model, and NeuralLP [Yang et al., 2017], a model which similarly learns first-order logical rules through a series of differentiable operations.

4.4.1.4 Experimental Setup

We followed the experimental setup of Rocktaschel and Riedel [2017] where necessary. Concretely, we provided the same rule templates, and learned their relation embeddings, but have not provided any rules in a form of templates with pre-trained embeddings. We selected the best hyperparameters through a hyperparameter sweep for each gNTP model. We use the Adam optimiser [Kingma and Ba, 2015] with the default settings for 100 epochs, on embeddings of size 100, running the models for the depth of 1. For small datasets, we swept the values of batch size in $[128, 256, 512, 1024]$, and we fixed the batch size to 1000 for the large datasets. Next, we swept the values of the learning rate in $[0.05, 0.01, 0.005]$, and values of k in $[1, 2, 5]$. On FB112 we ran the first 95 epochs passing gradients only through rule embeddings, thus pre-training rules, and then training entity embeddings and rules for the last 5 epochs. This forces gNTPs to learn good rules first.

4.4.2 Link Prediction on Small Datasets

4.4.2.1 Quantitative Analyses

We contrast the performance of gNTP with its predecessor, NTP and baselines from the category of Neural Link Predictors (DistMult, ComplEx, and ConvE), as well as the neuro-symbolic baselines MINERVA and NeuralLP, of which the latter one is a differentiable first-order rule learner. The results of the comparison are presented in Table 4.2.

There are a handful of conclusions to take from these results. First and foremost, gNTPs either have comparable performance to NTPs or outright outperform them consistently through the benchmark datasets. Other than the issue of the erroneous evaluation of NTP in the original paper, which directly hurt the performance of NTP, we hypothesise that the dominance of gNTP stems largely from it supporting a thorough hyperparameter sweep, which NTP simply cannot do due to its scaling issues.

Table 4.2: Link prediction results for small datasets. Results from Das et al. [2017a][a], Dettmers et al. [2018][b], Yang et al. [2017][c]. Globally best results in bold, best result among the graph and neural logic models underlined. Note that the NTP results form Rocktaschel and Riedel [2017] were recalculated to fix the evaluation issue described in the text.

	Countries			Nations				Kinship				UMLS			
	S1	S2	S3												
	AUC-PR			MRR	HITS			MRR	HITS			MRR	HITS		
					@1	@3	@10		@1	@3	@10		@1	@3	@10
Neural Link Prediction Models															
DistMult[a]	0.98 ± 0.00	0.69 ± 0.02	0.16 ± 0.01	–	–	–	–	**0.88**	**0.80**	**0.94**	**0.98**	**0.94**	**0.92**	**0.97**	0.99
ConvE[b]	**1.00 ± 0.00**	**0.99 ± 0.01**	0.86 ± 0.05	**0.82**	**0.72**	**0.88**	**1.00**	0.83	0.74	0.92	**0.98**	**0.94**	**0.92**	0.96	0.99
ComplEx[a]	0.99 ± 0.00	0.88 ± 0.02	0.48 ± 0.06	–	–	–	–	0.84	0.75	0.91	**0.98**	0.89	0.82	0.96	**1.00**
Graph-Based Models															
MINERVA[a]	**1.00 ± 0.00**	0.92 ± 0.02	**0.95 ± 0.01**	–	–	–	–	0.72	0.61	0.81	0.92	0.83	0.73	0.90	0.97
Neural Logic Models															
NeuralLP[a]	**1.00 ± 0.00**[c]	0.75 ± 0.00[c]	0.92 ± 0.00[c]	–	–	–	–	0.62	0.48	0.71	0.91	0.78	0.64	0.87	0.96
NTP	0.91 ± 0.15	0.87 ± 0.12	0.57 ± 0.18	0.61	0.45	0.73	0.87	0.35	0.24	0.37	0.57	0.80	0.70	0.88	0.95
gNTP	**1.00 ± 0.00**	0.88 ± 0.03	0.86 ± 0.04	0.73	0.60	0.81	0.99	0.74	0.62	0.84	0.95	0.86	0.76	0.95	0.99
gNTP (attention)	**1.00 ± 0.00**	0.91 ± 0.03	0.85 ± 0.06	0.78	0.68	0.86	**1.00**	0.76	0.64	0.85	0.96	0.86	0.76	0.95	0.99

Second, as noted in Rocktaschel and Riedel [2017] for NTPs, gNTPs too still lag behind specialised link prediction models, even more so since in the meantime there has been a number of models steadily pushing the performance on these datasets upward. gNTPs still have issues in learning subsymbolic representations, in particular since the unification score relies on applying the min operation on the representation elementwise, as opposed to neural link predictors which have a higher capacity for learning specialised representations, given that they directly optimise the score of the triple. A possible avenue for future work would be to investigate different ways to integrate score-based link predictors into the unification score, from regularisation, over forming a mixture of experts, to redesigning the unification score to directly utilise state-of-the-art triple scoring. In our experiments, we decided not to push learning auxiliary losses in the model as Rocktaschel and Riedel [2017] did as our goal here was not to break state-of-the-art but advance the original NTP model.

Third, gNTP outperforms or performs as well as MINERVA and NeuralLP on all but the Countries S2 and S3 datasets showing that gNTP still has an advantage when compared to other interpretable models.

Finally, gNTPs still keep the interpretability of NTPs as its major advantage over uninterpretable systems such as neural link predictors—one can both inspect inducted rules as a means of interpreting what the model learned, as well as reconstruct the proof path the system chooses as the highest-scoring

Table 4.3: gNTP-induced rules on the Countries dataset. The upper half of the rules are the valid ones, whereas the lower half are the invalid ones. Rules in bold are rules necessary to solve the dataset.

Rule	Score
Countries S1	
`neighborOf(X, Y) :- neighborOf(Y, X)`	0.98
`locatedIn(X, Y) :- locatedIn(X, Z), locatedIn(Z, Y)`	0.81
`locatedIn(X, Y) :- neighborOf(X, Z), neighborOf(Z, Y)`	0.67
`locatedIn(X, Y) :- locatedIn(Y, X)`	0.48
Countries S2	
`neighborOf(X, Y) :- neighborOf(Y, X)`	0.98
`locatedIn(X, Y) :- locatedIn(X, Z), locatedIn(Z, Y)`	0.95
`locatedIn(X, Y) :- neighborOf(X, Z), locatedIn(Z, Y)`	0.68
`locatedIn(X, Y) :- neighborOf(X, Z), neighborOf(Z, Y)`	0.72
Countries S3	
`neighborOf(X, Y) :- neighborOf(Y, X)`	1.00
`locatedIn(X, Y) :- locatedIn(X, Z), locatedIn(Z, Y)`	0.93
`locatedIn(X, Y) :- neighborOf(X, Z), locatedIn(Z, Y)`	0.95
`locatedIn(X, Y) :- neighborOf(X, Z), neighborOf(Z, W), locatedIn(W, Y)`	0.95
`locatedIn(X, Y) :- locatedIn(X, Z), neighborOf(Z, W), neighborOf(W, Y)`	0.70
`locatedIn(X, Y) :- locatedIn(X, Z), neighborOf(Z, W), locatedIn(W, Y)`	0.73

one. We take a look at some of the induced rule per dataset, and a few interesting proof paths for the Nations, Kinship and UMLS datasets

4.4.2.2 Qualitative Analyses

Countries As we can see in Table 4.3, gNTP induces the necessary rules to solve each of the Countries datasets. The model also induces some incorrect rules, as well as multiple instances of the correct rules (not presented in the table), albeit with varying scores. Note here that the rule and proof scores are calculated by calculating the proof score as in Equation (4.1), by comparing the similarity between the learned representations of the rule predicates and their 1-NN decoded predicates, as in Rocktaschel and Riedel [2017]. Though this does happen often in our experiments, by analysing the highest-scoring proof paths, we notice that only the highest-scoring rule is used in these proof paths, whereas the others are not. This finding tells us that we should take care when decoding the rules and that we should also take a look at the highest-scoring proof paths to analyse which of the rules are being used by the model.

Table 4.4: gNTP-induced rules and proofs on the Nations dataset.

Rules	
Rule	**Score**
`commonbloc2(X, Y) :- commonbloc2(Y, X)`	0.88
`unweightedunvote(X, Y) :- unweightedunvote(Y, X)`	0.83
`exports3(X, Y) :- relexports(X, Y)`	0.73
`tourism(X, Y) :- relstudents(Y, X)`	0.58
`unoffialacts(X, Y) :- unoffialacts(X, Z), unoffialacts(Z, Y)`	0.53
`embassy(X, Y) :- officialvisits(Y, X)`	0.31
Proofs	
Proof	**Score**
G `commonbloc2(burma, egypt)` ∴ `commonbloc2(X, Y) :- commonbloc2(Y, X)` `└ commonbloc2(egypt, burma)`	0.88
G `unweightedunvote(netherlands, poland)` ∴ `unweightedunvote(X, Y) :- unweightedunvote(Y, X)` `└ unweightedunvote(poland, netherlands)`	0.83
G `negativecomm(cuba, usa)` ∴ `commonbloc2(cuba, usa)`	0.69

Nations Next, we present some of the both high and low -scoring rules induced on the Nations dataset, as well as proofs for some goals in Table 4.4. Though Rocktaschel and Riedel [2017] did not include proof paths in their research, we find it interesting to see them as a way to understand gNTPs better. In the case of the Nations dataset, gNTPs induce and often use fairly straightforward symmetric relations, but do not necessarily use them for every proof as is the case when a proof contains similar facts, with one of the entities being different. Other induced rules often indicate a relationship between two relations which were derived from the same or similar/related data, for example, the `exports3(X, Y) :- relexports(X, Y)` rule is indicative of the strong connection between the relative value of exports and normalised principal exports between nations.

Kinship Rules and proofs for the Kinship dataset are presented in Table 4.5. Given that the dataset is anonymised, it is difficult to understand whether the rules and the proofs are correct. However, it is interesting to observe that the model induces more or less just the standard symmetric relations, but it often uses just facts for proving goals. Using just fact unification to prove a goal implies similarity between people in facts being compared, i.e. symbol embeddings for related people end up being similar.

Table 4.5: gNTP-induced rules and proofs on the Kinship dataset.

Rules	
Rule	**Score**
`term0(X, Y) :- term0(Y, X)`	0.98
`term4(X, Y) :- term4(Y, X)`	0.88
`term15(X, Y) :- term5(Y, X)`	0.78
`term9(X, Y) :- term15(X, Z), term5(Z, Y)`	0.16
`term10(X, Y) :- term5(X, Z), term15(Z, Y)`	0.07
`term24(X, Y) :- term15(X, Y)`	0.04
`term24(X, Y) :- term24(X, Z), term0(Z, Y)`	0.03
Proofs	
Proof	**Score**
G `term0(person93, person3)` ⋱ `term0(X, Y) :- term0(Y, X)` ↰ `term0(person3, person93)`	0.98
G `term17(person74, person15)` ⋱ `term17(person74, person8)`	0.95
G `term11(person1, person84)` ⋱ `term11(person1, person66)`	0.94
G `term4(person12, person49)` ⋱ `term4(X, Y) :- term4(Y, X)` ↰ `term4(person49, person12)`	0.88

UMLS We present the rules and proofs for the UMLS dataset in Table 4.6. UMLS is interesting as the rules induced include not just the symmetric relations but the transitive relations too. Interestingly, most of the transitive relations have similarly low scores and are decoded as the `isa` transitive relation.

4.4.3 Quantifying gNTP Scalability

To quantify the scalability of gNTP with respect to NTP, we empirically analyse their runtime and memory use during training as performance metrics. We focus on training only as similar performances are expected during inference. Given that the number of neighbours is an important parameter performance-wise, we quantify the performance of gNTP for different values of it.

Runtime To quantify the runtime of each model, we compare them by measuring their average number of examples processed during 10 batches of training, per model. We want to quantify the maximal runtime performance that each model can achieve, and to that extent, we approximately determine the

Table 4.6: gNTP-induced rules and proofs on the UMLS dataset. Note that the double occurrence of the `isa(X, Y) :- isa(X, Z), isa(Z, Y)` rule is a result of the rule 1-nearest neighbour interpretation. The embeddings of these rules are different, but the interpretation is the same.

Rules	
Rule	**Score**
`interacts_with(X, Y) :- interacts_with(X, Z), interacts_with(Z, Y)`	0.92
`isa(X, Y) :- isa(X, Z), isa(Z, Y)`	0.65
`isa(X, Y) :- isa(X, Y)`	0.64
`isa(X, Y) :- isa(X, Z), isa(Z, Y)`	0.59
`degree_of(X, Y) :- degree_of(X, Z), degree_of(Z, Y)`	0.41
`conceptually_related_to(X, Y) :- isa(X, Z), conceptually_related_to(Z, Y)`	0.29
Proofs	
G `performs(professional_or_occupational_group, health_care_activity)` ∴ `performs(group, health_care_activity)`	0.99
G `produces(genetic_function, hormone)` ∴ `produces(genetic_function, enzyme)`	0.98

G `interacts_with(invertebrate, fish)`
∴ `interacts_with(X, Y) :- interacts_with(X, Z), interacts_with(Z, Y)`
`interacts_with(amphibian, fish)`
`interacts_with(invertebrate, amphibian)`
	0.92

G `isa(steroid, substance)`
∴ `isa(X, Y) :- isa(X, Z), isa(Z, Y)`
`isa(chemical, substance)`
`isa(steroid, chemical)`
	0.64

maximum batch size each model can use to fit the memory of an NVIDIA GeForce GTX 1080 Ti GPU. Note that by doing this, we are quantifying the approximately maximum potential speedup gained by gNTP. This cannot be considered as the exact upper bound on the speedup of gNTP due to two reasons: i) the architecture of GPUs can often cause a bigger average number of processed examples achieved with lower batch size, and ii) the process of finding the maximum batch size was approximate.

Memory To evaluate the memory use for each model, we compare the maximum GPU memory utilisation of both models, again over 10 training batches, but importantly, comparing the models of the same batch size. This enables us to quantify the difference between the memory use of gNTP and NTP on equal grounds. Note here that the NTP often cannot utilise higher batch sizes as it rapidly reaches top memory capacity. We compared the GPU memory utilisation because both models make use of the GPU memory (the computation of all proof paths is done on the GPU and the embeddings used in them need to fit in the GPU memory), and gNTP saves the NNS index on it. In our previous experiments, we did this on CPU to ensure that we include the size

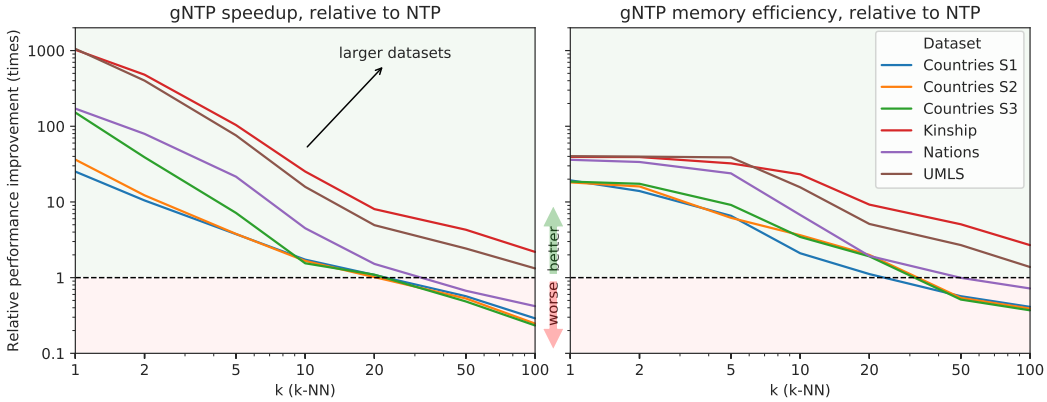


Figure 4.3: The runtime and memory performance of gNTP, relative to NTP. The performance is expressed as the models' ratio of the average number of examples processed (runtime), and the maximum GPU memory used (memory) in 10 training batches. Lightly green area signifies better performance of gNTP and the light red area signifies better performance of NTP, also denoted with the green (better) and the red (worse) arrows.

of the ANNS saved in RAM, and as a fail-safe, in case NTP did not fit into the GPU memory, as it often did not.

The results of both of the performance measures are presented in Figure 4.3. We immediately observe that gNTPs are substantially more time and memory performant.

Concretely, gNTPs yield significant speedups of an order of magnitude for the Countries S1 and S2 datasets, two orders of magnitude for Countries S3 and Nations, and even three orders of magnitude for Kinship and UMLS, when using only the top-nearest neighbour. With a higher number of neighbours, the gains fall due to the cost of NNS querying, as well due to additional engineering details around the utilisation of the NNS. This is particularly evident in the case of the Countries datasets where for k larger than 20, NTPs perform better due to the overhead of NNS. We also observe the trend of gNTPs consistently outperforming NTP with the increased size of the dataset—the larger the dataset, more the gain gNTPs exhibit. This indicates that, should it be possible to run NTP on large datasets, we would expect the same trend to continue, thus the time performance gain of gNTP would be in increasing orders of magnitude.

gNTPs are also more memory efficient, with clear possible savings above an order of magnitude for reasonable sizes of k. Similar findings from the runtime

Table 4.7: Link prediction results on small datasets of gNTP with a varied number of neighbours. Globally best results in bold, best results for each gNTP and gNTP (attention) underlined.

	Countries			Nations				Kinship				UMLS			
	S1	S2	S3												
	AUC-PR			MRR	HITS			MRR	HITS			MRR	HITS		
					@1	@3	@10		@1	@3	@10		@1	@3	@10
gNTP															
k = 1	**1.00 ± 0.00**	0.88 ± 0.03	0.86 ± 0.04	0.74	0.62	0.83	**1.00**	0.64	0.50	0.73	0.93	0.70	0.57	0.78	0.92
k = 2	**1.00 ± 0.00**	0.69 ± 0.19	0.83 ± 0.07	0.73	0.60	0.81	0.99	0.74	0.62	0.84	0.95	**0.86**	**0.76**	**0.95**	**0.99**
k = 5	0.99 ± 0.03	0.90 ± 0.00	**0.88 ± 0.04**	0.74	0.62	0.84	0.99	0.71	0.57	0.82	0.95	0.84	0.73	**0.95**	0.98
gNTP (attention)															
k = 1	**1.00 ± 0.00**	0.91 ± 0.03	0.85 ± 0.06	0.75	0.63	0.84	0.99	0.59	0.45	0.68	0.86	0.71	0.59	0.80	0.90
k = 2	**1.00 ± 0.00**	**0.92 ± 0.04**	0.85 ± 0.06	**0.78**	**0.68**	**0.86**	**1.00**	0.72	0.58	0.83	0.95	0.85	0.74	0.94	**0.99**
k = 5	**1.00 ± 0.00**	0.91 ± 0.01	0.78 ± 0.15	0.73	0.61	0.81	0.99	**0.76**	**0.64**	**0.85**	**0.96**	**0.86**	**0.76**	**0.95**	**0.99**

experiments apply; for larger values of k performance gains of gNTPs drop, for Countries, gNTPs perform worse than NTPs for $k > 20$, and the trend of gNTPs ourperforming NTPs on datasets of increasing sizes still holds.

In addition, we analysed the performance of gNTPs with and without attention as a function of the number of neighbours k. The results in Table 4.7 do not suggest a strong relationship between k and model performance as the best performances are achieved across all the tested values of $k \in \{1, 2, 5\}$. However, we do observe that for the smallest of datasets (Countries) lower values of $k \in \{1, 2\}$ achieve the best performances, and for others a larger value of $k \in \{2, 5\}$ performs the best, even though lower values of k yield only marginally worse performance than the best ones. This additionally affirms the importance of sweeping k as a hyperparameter. Note that there is discrepancy between the best values in Table 4.7 and Table 4.2 because we chose the best performing model per each k based on models' development set performances. This implies that the best performing models from Table 4.2 appear in Table 4.7, but there are even better-performing ones even though they have a globally lower dev set performance. This finding tells us that the best development set performance does not necessarily correspond to the best test set performance.

We can clearly state that the proof path pruning in gNTPs drastically increases the efficiency of learning (and the inference process as they both rely on the same model mechanisms). These findings indicate that gNTPs should be readily applicable to large datasets, which we do next.

Table 4.8: Link prediction results for the WN18 and WN18RR datasets. Results from [Das et al., 2017a][a], [Yang et al., 2017][b], [Dettmers et al., 2018][c], [Kadlec et al., 2017][d], [Trouillon et al., 2016][e]. Globally best results in bold, best results among the graph and neural logic models underlined. Note that NTPs cannot run on these datasets.

	WN18				**WN18RR**			
	MRR	HITS			MRR	HITS		
		@1	@3	@10		@1	@3	@10
Neural Link Prediction Models								
DistMult	0.797[d]	–	–	0.95	0.433[a]	0.410	0.441	0.475
ConvE	**0.943**[c]	0.935	**0.946**	**0.956**	0.438[a]	0.403	0.452	0.519
ComplEx	0.941[e]	0.936	0.945	0.947	0.415[a]	0.382	0.433	0.480
Graph-Based Models								
MINERVA	–	–	–	–	0.448[a]	**0.413**	0.456	0.513
Neural Logic Models								
NeuralLP	0.940[b]	–	–	0.945[b]	**0.463**[a]	0.376	**0.468**	**0.657**
NTP	–	–	–	–	–	–	–	–
gNTP	0.940	**0.938**	0.943	0.944	0.434	0.410	0.442	0.484

4.4.4 Link Prediction on Large Datasets

Previously, we showed the potential of gNTP to scale to large datasets. Here we evaluate their performance on large link prediction datasets, WN18, WN18RR and FB122.

4.4.4.1 Quantitative Analyses

WN18 and WN18RR We evaluate the performance of gNTPs and the baseline models on the WN18 and WN18RR datasets in Table 4.8. In terms of ranking accuracies, we observe gNTPs comparing well to ComplEx and NeuralLP, though still lagging behind ConvE on the WN18 dataset. However, on the WN18RR, gNTPs, though outperforming complex, still lag behind NeuralLP and MINERVA, as well as ConvE.

Next, we wanted to contrast one representative of neural link predictors and gNTP to see whether there are any differences in the treatment of the dataset—where does the link prediction perform better and gNTP fails and vice-versa. To that extent, we compared gNTP and ComplEx per-predicate in terms of MRR on the test set of both datasets.

The results, presented in Table 4.9 and Table 4.10 show that gNTP and ComplEx have complementary strengths and weaknesses. We observe that

Table 4.9: Per-predicate MRR comparison for ComplEx and gNTP on the WN18 dataset.

Predicate	ComplEx	gNTP
_hyponym	0.890	**0.937**
_member_holonym	0.809	**0.912**
_hypernym	0.891	**0.934**
_part_of	0.826	**0.921**
_derivationally_related_form	**0.917**	0.035
_member_of_domain_topic	**0.745**	0.722
_instance_hyponym	**0.776**	0.490
_synset_domain_topic_of	0.746	**0.771**
_synset_domain_region_of	**0.689**	0.362
_member_of_domain_region	**0.667**	0.417
_has_part	**0.839**	0.680
_also_see	0.511	**0.554**
_instance_hypernym	**0.774**	0.645
_member_meronym	**0.815**	0.614
_verb_group	0.677	**0.951**
_synset_domain_usage_of	**0.776**	0.775
_member_of_domain_usage	0.722	**0.769**
_similar_to	**1.000**	**1.000**

Table 4.10: Per-predicate MRR comparison for ComplEx and gNTP on the WN18RR dataset.

Predicate	ComplEx	gNTP
_hypernym	**0.092**	0.022
_derivationally_related_form	**0.941**	0.934
_member_meronym	**0.133**	0.055
_has_part	**0.123**	0.046
_also_see	0.522	**0.593**
_member_of_domain_region	**0.040**	0.011
_verb_group	0.825	**0.893**
_synset_domain_topic_of	**0.184**	0.042
_instance_hypernym	**0.241**	0.093
_member_of_domain_usage	**0.201**	0.030
_similar_to	**1.000**	0.764

gNTPs benefit from a clear logical structure on the WN18 dataset, which is characterised by a more logical relational structure. For instance, inducing almost crisp rules such as part_of(X,Y) :− has_part(Y,X), hyponym(X,Y) :− hypernym(Y,X), and hypernym(X,Y) :− hyponym(Y,X), aids gNTP in accurately predicting the underlying structure in WN18 by using these rules to yield more accurate results on the part_of, _hyponym and the _hypernim relations, as presented in Table 4.9.

On the other hand, we can also observe that, in some cases, logic rules and continuous unification do not suffice for some relations. For exam-

Table 4.11: Link prediction results on the FB122 dataset. Results from Minervini et al. [2017][a], Guo et al. [2016][b], Garcia-Duran and Niepert [2017][c]. Globally best results in bold, best results among the models which do not use rules underlined. KALE, *ASR* methods, and KBLR use the set of rules provided by Guo et al. [2016] while neural link predictors and gNTPs do not.

		Test-I				Test-II				Test-ALL			
		MRR	HITS			MRR	HITS			MRR	HITS		
			@3	@5	@10		@3	@5	@10		@3	@5	@10
Use Rules	KALE-Pre[b]	0.291	0.358	0.419	0.498	0.713	0.829	0.861	0.899	0.523	0.617	0.662	0.718
	KALE-Joint[b]	0.325	**0.384**	**0.447**	**0.522**	0.684	0.797	0.841	0.896	0.523	0.612	0.664	0.728
	ASR-DistMult[a]	0.330	0.363	0.403	0.449	0.948	0.980	0.990	0.992	0.675	0.707	0.731	0.752
	ASR-ComplEx[a]	**0.338**	0.373	0.410	0.459	0.984	**0.992**	**0.993**	**0.994**	0.698	0.717	0.736	0.757
	KBLR[c]	–	–	–	–	–	–	–	–	**0.702**	**0.740**	**0.770**	**0.797**
No Rules	TransE[a]	0.296	0.360	0.415	0.481	0.630	0.775	0.828	0.884	0.480	0.589	0.642	0.702
	DistMult[a]	0.313	0.360	0.403	0.453	0.874	0.923	0.938	0.947	0.628	0.674	0.702	0.729
	ComplEx[a]	0.329	0.370	0.413	0.462	0.887	0.914	0.919	0.924	0.641	0.673	0.695	0.719
	gNTPs	0.314	0.338	0.373	0.415	**0.987**	0.990	0.991	0.992	0.684	0.697	0.713	0.733

ple, gNTP is not able to learn a set of rules for accurately predicting the `_derivationally_related_form` predicate, where ComplEx simply thrives. However, ComplEx predictions are not easy to explain, since the score is a function of the embedding of the predicate and the entities involved in the prediction.

On the WN18RR dataset, ComplEx shines on relations that reflect the cluster structure of the underlying graph, such as `_also_see` and `_derivationally_related_form` as it does not need to rely on an underlying logical structure as gNTP does, it can more accurately handle the cases where such a structure is missing. Yet, ComplEx yields less accurate results on relations which can be accurately predicted by leveraging an underlying logical structure, which gNTP can learn and then leverage at test time.

Given that ComplEx and gNTP have complementary strengths and weaknesses, we believe the gap between them can be narrowed down by using ComplEx or any other link prediction algorithm as a regulariser (akin to the NTP-lambda in the original NTP paper), by proposing a mixture of experts, and possibly by adding a mixture of correctly induced rules from multiple runs of gNTP.

All in all, gNTP can learn symmetry rules, while also softly unifying related predicates and by leveraging such rules can perform better or on par with ComplEx on relations exhibiting a clear logical structure (symmetric relations), while still benefiting from the continuous unification.

FB122 The link prediction results for FB122 are presented in Table 4.11. It presents the results of the baselines we use throughout the chapter (TransE, DistMult and ComplEx) with a series of three additional models which can utilise rules in the dataset. These models are KALE [Guo et al., 2016], DistMult and ComplEx using Adversarial Sets [Minervini et al., 2017] (a method for incorporating rules in neural link predictors with adversarial training), and KBLR [Garcia-Duran and Niepert, 2017]. Note that these methods have access to the 47 rules coming with the dataset, while gNTP does not, and it needs to induce them to make accurate predictions.

Results, presented in Table 4.11 show that gNTP, though not having access to rules, can perform on-par with methods that have access to them. Concretely, gNTP lags behind specialised rule-using baselines when there is no clear logical structure in the dataset, as in the Test-I split. However, where the dataset exhibits a clear logical structure, as in the Test-II split, gNTP can induce the rules to fit the structure to its advantage, and not just use the manually provided rules, as the rule-using baselines do.

4.4.4.2 Qualitative Analyses

WN18 and WN18RR Table 4.12 displays gNTP-induced rules and proofs for the WN18 dataset, and Table 4.13 presents the same for the WN18RR dataset, giving us a glimpse of what gNTP learned. We see that gNTP mostly learned symmetrical and anti-symmetrical relations, which it strongly uses throughout the dataset. However, what is more interesting is that gNTP can find alternative, non-trivial explanations, based on the similarity between entity representations. For example, on WN18 gNTP can explain that CONGO.N.03 is a part of AFRICA.N.01 by leveraging the similarity between AFRICA.N.01 and the AFRICAN_COUNTRY.N.01, and the fact that the AFRICAN_COUNTRY.N.01 is a hyponym of CONGO.N.03. Similarly, on WN18RR, it explains that CHAPLIN.N.01 is a FILM_MAKER.N.01 by leveraging the fact that CHAPLIN.N.01 is a COMEDIAN.N.01 and the similarity between the FILM_MAKER.N.01 and COMEDIAN.N.01.

Further inspection of rules induced by gNTP on these two datasets yields interesting findings. For instance, we see that on WN18 gNTP induces crisp `_similar_to(X,Y) :- _similar_to(Y,X)`, whereas on WN18RR it does too albeit with a very low score. A deeper look into when the rule is used, we

Table 4.12: gNTP-induced rules and proofs on the WN18 dataset.

Rules	
Rule	**Score**
_part_of(X, Y) :- _has_part(Y, X)	1.00
_hypernym(X, Y) :- _hyponym(Y, X)	0.99
_hyponym(X, Y) :- _hypernym(Y, X)	0.99
_similar_to(X, Y) :- _similar_to(Y, X)	0.97
_has_part(X, Y) :- _member_meronym(Y, X)	0.97
_derivationally_related_form(X, Y) :- _hypernym(Y, X)	0.91
_hyponym(X, Y) :- _hyponym(Y, X)	0.77

Proofs	
Proof	**Score**
G _part_of(congo.n.03, africa.n.01) ∵ _part_of(X, Y) :- _has_part(Y, X) ⤷ _has_part(africa.n.01, congo.n.03)	1.00
∵ _part_of(X, Y) :- _has_part(Y, X) ⤷ _instance_hyponym(african_country.n.01, congo.n.03)	0.79
G _hyponym(extinguish.v.04, decouple.v.03) ∵ _hyponym(X, Y) :- _hypernym(Y, X) ⤷ _hypernym(decouple.v.03, extinguish.v.04)	0.99
∵ _hyponym(X, Y) :- _hypernym(Y, X) ⤷ _hypernym(snuff_out.v.01, extinguish.v.04)	0.92
G _derivationally_related_form(rewrite.v.01, rewriting.n.01) ∵ _derivationally_related_form(X, Y) :- _hypernym(Y, X) ⤷ _hypernym(revise.v.01, rewrite.v.01)	0.81
∵ _derivationally_related_form(X, Y) :- _hypernym(Y, X) ⤷ _hypernym(trench.v.05, excavate.v.04) G _derivationally_related_form(chorus.n.05, chorus.v.01) ∵ _derivationally_related_form(X, Y) :- _hypernym(Y, X) ⤷ _hypernym(hippopotamus.n.01, even-toed_ungulate.n.01)	0.21 0.40
∵ _derivationally_related_form(X, Y) :- _hypernym(Y, X) ⤷ _hypernym(sympathy.n.02, feeling.n.01)	0.03

Table 4.13: gNTP-induced rules and proofs on the WN18RR dataset.

Rules	
Rule	**Score**
_verb_group(X, Y) :- _also_see(Y, X)	0.96
_derivationally_related_form(X, Y) :- _derivationally_related_form(Y, X)	0.95
_member_meronym(X, Y) :- _hypernym(Y, X)	0.94
_synset_domain_topic_of(X, Y) :- _hypernym(Y, X)	0.90
_member_of_domain_usage(X, Y) :- _hypernym(Y, X)	0.85
_has_part(X, Y) :- _hypernym(Y, X)	0.83
_similar_to(X, Y) :- _similar_to(Y, X)	0.05
_also_see(X, Y) :- _similar_to(Y, X)	0.02

Proofs	
Proof	**Score**
G _verb_group(respire.v.02, breathe.v.01) ∵ _verb_group(X, Y) :- _also_see(Y, X) ⤷ _verb_group(breathe.v.01, respire.v.02)	0.96
G _verb_group(respire.v.02, breathe.v.01) ∵ _hypernym(hyperventilate.v.02, breathe.v.01)	0.85
G: _instance_hypernym(chaplin.n.01, film_maker.n.01)	0.81
∵ _instance_hypernym(chaplin.n.01, comedian.n.01)	
∵ _instance_hypernym(scipio.n.01, general.n.01)	0.63
G: _hypernym(krypton.n.01, noble_gas.n.01)	0.91
∵ _hypernym(krypton.n.01, chemical_element.n.01)	
∵ _hypernym(tellurium.n.01, chemical_element.n.01)	0.60
G: _has_part(tennessee.n.01, knoxville.n.01)	0.54
∵ _has_part(india.n.01, jabalpur.n.01)	
∵ _has_part(alaska.n.01, anchorage.n.03)	0.53

see that in WN18 it is often utilised, whereas on WN18RR it is never used for prediction. This is a direct consequence of the way WN18RR was created, as these particular examples are filtered out of the dev and test sets.

We observe an intriguing rule `_verb_group(X,Y) :- _also_see(Y,X)` on WN18RR. This rule is interesting, as it is often used to express multiple symmetrical relationships with predicates such as `_also_see` (indicating an alternate or equivalent version of a word sense), `_verb_group` (indicating equivalent verb sense denoting a higher abstraction level) and `_similar_to` (expressing closely related meaning) [Fialho et al., 2011]. We take a look at the final proof paths for a few examples and see that the same rule is used to prove facts with different predicates:

```
G _also_see(coherent.a.01, logical.a.01)
∵ _verb_group(X, Y) :- _also_see(Y, X)
                       └ and _also_see(logical.a.01, coherent.a.01)

G _verb_group(allow.v.03, permit.v.01)
∵ _verb_group(X, Y) :- _also_see(Y, X)
                       └ and _verb_group(permit.v.01, allow.v.03)

G _similar_to(dynamic.a.01, hold-down.n.01)
∵ _verb_group(X, Y) :- _also_see(Y, X)
                       └ and _similar_to(hold-down.n.01, dynamic.a.01)
```

This essentially tells us that the rule in question is used for representing symmetry between multiple relations and that the originally proposed decoding of the rule with a one-nearest-neighbour in Rocktaschel and Riedel [2017], though informative, should be taken with caution, as though the interpretation of the rule is crisp/discrete, the rule itself does not necessarily behave like one. This rule is likely a product of the winner-takes-all strategy employed by both the score function calculation and the fact and rule selection. A possible way to alleviate this issue is to follow the findings of de Jong and Sha [2019], who suggest propagating gradients not just through the top ranking proof paths but top-k paths.

However, one important take is that though gNTPs do not produce a concrete representation of a rule as we might wish, we can still decide whether that rule is meaningful or not, and use such insights for refining the model, improving our understanding of the domain, or providing explanations for any given prediction. Hence we suggest looking at the final proof paths when assessing

```
timeZone(X, Y) :- containedBy(X, Z), timeZone(Z, Y)
nearbyAirports(X, Y) :- containedBy(X, Z), contains(Z, Y)
nationality(X, Y) :- placeOfBirth(Y, X)
children(X, Y) :- parents(Y, X)
spouse(X, Y) :- spouse(Y, X)
```

Listing 4.1: An excerpt of the gNTP-induced rules on the FB122 datset.

these rules as they are highly informative of the use of each of these rules.

FB112 We briefly take a look at the rules gNTPs induce on the FB112 dataset. From the excerpt of the rules induced, presented in Listing 4.1, we can see they indeed are meaningful.

It is worth noting that FB122 was derived from the problematic [Dettmers et al., 2018] FB15k dataset, with Test-I and Test-II splits exemplifying the datasets issues. Test-I triples cannot be predicted with the help of rules and were designed with a similar intention to the FB15k-237 dataset, whereas the Test-II triples can since they are a direct subset of FB15k.

The finding that gNTP performs well when there is a clear logical structure in the dataset is similar to the finding on the WN18(RR) datasets, corroborating the complementary strengths of the link prediction models (better representation fit) and gNTP (better use of the logical structure).

4.4.5 Experiments with Text

To evaluate the effect of the compositional reader for integrating textual information, we created a modified version of the Countries datasets. We replaced an increasing percent of training set triples from each of the Countries S1, S2 and S3 datasets, with human-generated textual mentions. Each relation in the replaced triple was randomly sampled from a set of 30 textual mentions, per each relation, catalogued in Table B.1. For example, the fact `neighbourOf(UK,IRELAND)` can be replaced by the mention `"[#1] is positioned closest to [#2]"(UK,IRELAND)`.

Then, we evaluate two ways of integrating textual mentions, by either i) treating them as a monolithic predicate, or by ii) parsing the mentions through an encoder, as described earlier. It is important to note that both of these cases are trained on the same (decreasing) amount of training data, with mentions

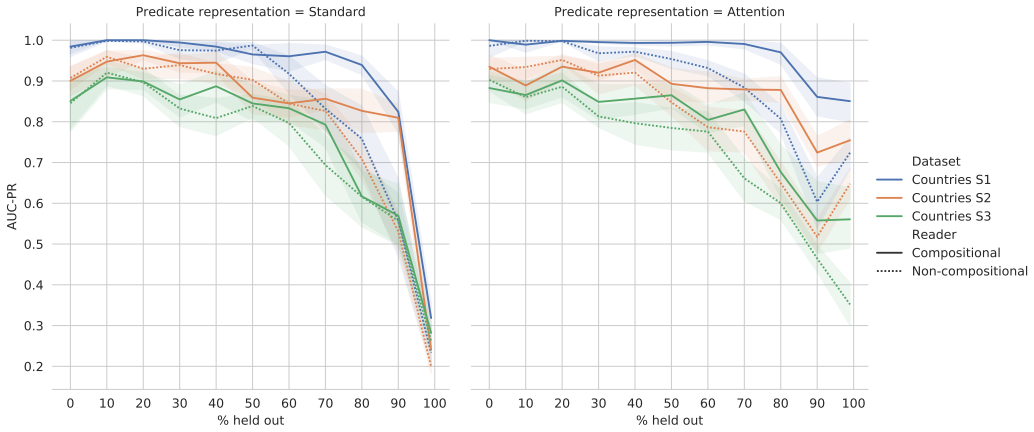


Figure 4.4: The performance of gNTPs on Countries with mentions dataset. We replaced a varying number of training triples with human-generated mentions and integrated them as facts in the KBs in two ways i) by encoding the mentions as standalone predicates, and ii) by encoding them with a compositional reader. We conducted the experiments using the standard NTP representation (left) and the attentional representation of gNTPs (right). Each experiment was run on 10 different random seeds.

not being a part of the training set—the model uses them as supporting facts in the proof process but does not directly optimise them as goals. This effectively decreases the amount of information the model uses with the increasing percent of data held out (converted to text), but still enables the model to optimise their representations.

The results, presented in Figure 4.4, present two findings. First, the proposed compositional encoding reader yields consistent improvements of the ranking accuracy, especially if attention is used. In case of using the attention for predicate representation, the compositional reader performs better from 40% of held-out data. When standard predicate representation is used, the compositional reader performs better on Countries S1 and S2 from 70% of held-out data, but performs roughly the same as the non-compositional reader for Countries S3. Second, we see that the attentional predicate representation performs significantly better than the standard representation, as is expected given that the standard representation needs to learn a much higher-dimensional representation of the predicates.

Since during training the KB is expanded with the textual mentions, rules learned by gNTPs can include both logic atoms and textual mentions. We see

```
neighborOf(X, Y) :- neighborOf(Y, X)
neighborOf(X, Y) :- "[#1] was a neighbor of [#2]"(Y, X)
neighborOf(X, Y) :- "[#1] is a neighboring state to [#2]"(Y, X)
locatedIn(X, Y) :- "[#1] was a neighboring state to [#2]"(X, Z),
                   "[#1] was located in [#2]"(Z, Y)
locatedIn(X, Y) :- "[#1] can be found in [#2]"(X, Z),
                   "[#1] is located in [#2]"(Z, Y)
```

Listing 4.2: An excerpt of rules extracted by gNTP on the Countries dataset with text.

exactly this happening when we take a look at the learned rules, as listed in Listing 4.2—gNTPs induce meaningful rules which include textual information too.

4.5 Related Work

We contrast gNTPs to related models across three central areas: neural network architectures, relational learning and Machine Learning -powered scaling.

4.5.1 Neural Network Architectures

Memory Augmented Neural Networks Recent advances in memory-enabled neural architectures aim to deal with the issues of generalisation [Graves et al., 2014, Joulin and Mikolov, 2015, Grefenstette et al., 2015, Kaiser and Sutskever, 2016], reasoning abilities [Weston et al., 2015, Sukhbaatar et al., 2015] and one-shot learning [Santoro et al., 2016] that neural networks often exhibit. Memory Augmented Neural Networks take the approach of enriching neural networks with a differentiable *external memory*, enabling these models to learn to represent and manipulate dense representations on long time scales via reading and writing to the external memory. However, even though there is a pressure to disentangle algorithm learning from learning the input representations, there is no guarantee that algorithm learning and learning input representations still do not conflate. In contrast to that, gNTPs do not learn the algorithm but fix it in the form of (differentiable) backward chaining, and then learn input representations given the fixed algorithm. Another way of improving the generalisation and extrapolation abilities of neural networks consists of designing architectures capable of learning general, reusable programs—atomic primitives that can be reused across a variety of environments and tasks [Reed and De Freitas, 2016, Neelakantan et al.,

2016, Parisotto et al., 2017]. We argue that gNTPs are such an architecture where one can easily reuse the induced set of rules by incorporating it into other KBs.

Differentiable Interpreters Differentiable interpreters enable the translation of declarative or procedural knowledge into a neural network architecture exhibiting strong inductive biases of said knowledge. In Chapter 3 we propose $\partial 4$, a differentiable abstract machine for the Forth programming language, enabling the construction of neural networks with a strong inductive bias of a procedurally written program. Rocktaschel and Riedel [2017] propose a differentiable implementation of the backward chaining algorithm, while Evans and Grefenstette [2018] propose a differentiable forward chaining algorithm, both effectively differentiable Datalog interpreters. Providing a way to encode strong inductive biases into models by partially defining the program structure used to construct the network comes with a significant drawback—their computational complexity makes them unusable except for small learning problems, and training them with Stochastic Gradient Descent (SGD) has also been shown to be a challenge [Gaunt et al., 2016]. Our work shows that there is a feasible way to push forward that limit and scale to larger problems.

Neural Module Networks Andreas et al. [2016b] introduced Neural Module Networks, an end-to-end differentiable composition of jointly trained neural modules. The allure of Neural Model Networks comes from the ability to define and train differentiable composable models and interpret and execute their compositions as simple programs. This modularity is particularly useful when dealing with reasoning tasks from visual and natural language inputs, such as question answering [Andreas et al., 2016a], visual question answering [Andreas et al., 2016b] and reasoning over text with arithmetic modules [Gupta et al., 2019].

We recognise NTPs as a recursive differentiable composition of `or` and `and` modules, following the backward-chaining reasoning algorithm, jointly trained on downstream reasoning tasks. Interestingly, though in previous work the structure of the composition is statically drawn from the data, and in our work, it is statically drawn from the data and the model parameters, other approaches are trying to learn the module composition [Hu et al., 2017, Jiang and Bansal, 2019]

4.5.2 Relational Learning

Inductive Logic Programming The paradigm of Inductive Logic Programming (ILP) uses (usually first-order) logic as a description language for a KB and addresses induction of rules from facts and background knowledge to answer queries [Muggleton, 1991, Muggleton and De Raedt, 1994]. Systems such as MARVIN [Sammut and Banerji, 1986], FOIL [Quinlan, 1990], Progol [Muggleton, 1995], ALEPH [Srinivasan, 2001] and Metagol [Muggleton et al., 2015, Cropper and Muggleton, 2015, 2016] are symbolic systems that search over discrete space of rules/logic programs, and can even invent new predicates and induce recursive rules [Cropper and Muggleton, 2016]. Though both ILP and gNTP can induce rules from data, ILP systems do not learn relation and atom representations. Besides, similarly to gNTPs, the ILP community is actively searching for heuristics to speed up the induction process [Muggleton and Feng, 1990, Giordana et al., 1994, Srinivasan, 2000, Železnỳ et al., 2002, DiMaio and Shavlik, 2004], consequently enabling mining rules from large KBs [Galárraga et al., 2013, Chen et al., 2016]. Heuristics too push the ability of these systems to extract rules from textual data; SHERLOCK [Schoenmackers et al., 2010] is another ILP system in-kind related to gNTPs, given its ability to extract rules from web texts, though gNTPs are aiming towards reasoning over KBs while using textual data, as opposed to extracting rules. Finally, it is worthwhile mentioning the probabilistic formulation of ILP [De Raedt and Kersting, 2008] as related work, and even though our work does not include a probabilistic approach, it would be an exciting possibility for future work.

Knowledge Graph Embedding By embedding KB facts into a continuous vector space, we can simplify manipulation of facts, while preserving the structure of the KBs [Wang et al., 2017b] and use scoring functions established on triples or paths in the KBs to predict relationships (reason) between entities. Out of this fairly busy field, we present two main categories, the score-based and the path-based models.

Score-based models use either a distance or semantic similarity -based score to increase the score of facts in a KB. Distance-based score models [Bordes et al., 2013, Wang et al., 2014, Lin et al., 2015] push entities and relations in the same space, with a distance metric ensuring translational functional dependency between the entities in a fact. On the other hand, semantic similarity -based models encode a similarity function, ranging from more straightforward

functions [Nickel et al., 2011, Yang et al., 2014, Nickel et al., 2016b] to more elaborate ones based on convolutions [Dettmers et al., 2018] or even different number systems [Trouillon et al., 2016, Zhang et al., 2019].

Path-based models come in a few flavours. Random walk models [Lao and Cohen, 2010, Lao et al., 2011, Gardner et al., 2013, 2014, Gardner and Mitchell, 2015, Wang et al., 2016] aggregate paths from random walks to predict a target relation. Path-encoding models [Neelakantan et al., 2015a, Das et al., 2017b] often use RNNs to encode multi-hop paths between entities in the KB and use their aggregation for relation prediction. Similarly, some models utilise Reinforcement Learning (RL) to learn to walk over KBs [Xiong et al., 2017, Das et al., 2017a, Shen et al., 2018] and find predictive paths. All the models above cannot produce useful rules, as gNTPs can.

Joint Text and KB representation By embedding the KBs and text corpora, entities, relations and natural language can be represented in the same vector space, enabling meaningful and useful comparisons between them. This idea of jointly embedding KBs and texts has been explored by both the knowledge graph and the NLP communities.

The knowledge graph community's interest in this joint embedding stems from using the capabilities of KBs to reason, by using textual information to both expand KBs and reason with new relational facts [Wang et al., 2017b]. This first started with initialising entity representations with textual descriptions [Socher et al., 2013], but later moved towards models with entity [Wang et al., 2014, Zhong et al., 2015, Xu et al., 2016] and relation [Toutanova et al., 2015] representations enriched with textual information.

In NLP, this joint embedding has been explored by relation extraction systems—systems for extraction of relational facts from natural language texts [Mooney, 1999]. Notable approaches in relation extraction leveraged KBs as a form of distant supervision—utilising KB facts as supervision to the relation extraction process, assuming that sentences mentioning two entities signify their relationship [Bunescu and Mooney, 2007, Mintz et al., 2009, Riedel et al., 2010]. This form of assumption naturally led to many models modelling texts and KBs in the shared embedding space: scoring-function based models [Weston et al., 2013], matrix factorization [Riedel et al., 2013], tensor decomposition [Chang et al., 2014], as well as exploring elaborate mention encoders [Verga et al., 2016], and path-encoding models [Das et al., 2017b].

Neuro-symbolic Models Aiming to bring together the best of the neural world and the symbolic world, neuro-symbolic approaches [Smolensky, 1988, Garcez et al., 2015] combine neural networks learning symbolic reasoning. Interestingly, the history of neuro-symbolic systems coincides with the beginnings of connectionism, given the aims of first artificial neurons to express logical calculus [McCulloch and Pitts, 1943]. The idea of constructing neural networks able to emulate boolean algebra and symbolic reasoning weakly took on [Martin and Talavage, 1963, Chan et al., 1989] until the 1990s. Recently, research into neuro-symbolic systems took on resulted in logic-inspired neural architecture construction, learning and induction of rules in several types of logic, from boolean formulas of propositional logic [Towell et al., 1990, Towell and Shavlik, 1994, Shavlik and Towell, 1991, Garcez and Zaverucha, 1999, Steinbach and Kohut, 2002], over first-order logic [Shastri, 1992, Hölldobler et al., 1999, França et al., 2014] to other non-classical logics [Garcez et al., 2007, 2008, 2014]. Interestingly, the idea of implementing neural networks simulating deduction [Komendantskaya, 2007], unification [Komendantskaya, 2011] and even a neural inference engine for Prolog have been explored before [Chan et al., 1993, Ding, 1995, Ding et al., 1996]. However, these directions did not train any of the models in any way but used them for simulation.

Recently, there has been a surge of interest in neuro-symbolic models, usually based on continuous approximations of the semantics of logic [Grefenstette, 2013, Serafini and Garcez, 2016] applied to reasoning and rule induction.

DeepProbLog [Manhaeve et al., 2018] uses continuous relaxation to imbue ProbLog with neural predicates which can be applied on raw inputs, enabling reasoning on raw inputs. Logic Tensor Networks [Serafini and Garcez, 2016, Donadello et al., 2017] ground FOL terms, atoms and clauses in continuous functions, allowing reasoning with knowledge-based constraints and Relational Neural Machines [Marra et al., 2020] further generalise their approach.

Deep Relational Machine [Lodhi, 2013], on the other hand, induce rules and use Restricted Boltzmann Machines to capture relational information in them, but do not scale. Neural Logic Inductive Learning [Yang and Song, 2020] differentiably end-to-end learns FOL rules hierarchically with a transformer, to explain patterns in visual data. Similarly, DRUM [Sadeghian et al., 2019] learns probabilistic logical rules for inductive and interpretable link prediction, without requiring representations of entities in the KB.

Other models enable both continuous reasoning and rule induction, as gNTPs do. NeuralLP [Yang et al., 2017] uses a neural controller to learn to compose TensorLog [Cohen, 2016] differentiable operators and learn rules over these compositions. This method scales well and can integrate natural language texts, as opposed to the original NTPs [Rocktaschel and Riedel, 2017]. When compared to gNTPs, NeuralLP underperforms on all datasets we tested on except WN18RR. Difflog [Raghothaman et al., 2019] extends Datalog to the continuous domain, assigning numeric weights to rules and optimising rule (program) synthesis. It does, however, leave constants entirely symbolic, and does not perform as well on Countries S1 and S3 as gNTPs do. Neural Logic Machines [Dong et al., 2019] use tensors to represent logic predicates and perform sequential logic deductions on them. It does rely on symbolic inputs, though, are challenging to train, and do not scale to large datasets. ∂ILP [Evans and Grefenstette, 2018] is a differentiable ILP solver that constructs a network by following the forward-chaining algorithm, similarly to the model of Campero et al. [2018], who use forward-chaining to induce theories/rules and core facts which can then infer the rest of the data. As forward-chaining algorithms, though enabling predicate invention and induction of recursive rules, both of these approaches cannot scale to non-toy datasets.

Some of these models can softly reason over texts, with NeuralLP being an already mentioned one. Next, there is NLProlog [Weber et al., 2019] which uses an external Prolog prover that relies on a similarity function and a pre-trained sentence encoder to reason over texts. It is a RL model capable of learning rules over natural language and enabling reasoning over texts, akin to our fully differentiable gNTPs. Compared to gNTPs, NLProlog uses Prolog for proof path search and evolution strategies to estimate gradients through the non-differentiable search executed by Prolog. NLProlog also heavily relies on thresholding the similarity function used, which is crucial to the scaling capability they claim. However, their model has been applied on comparatively small datasets compared to ours, so it is not clear at all whether their model would even be able to scale to moderately sized datasets, let alone the large datasets on which we evaluated. Finally, [Clark et al., 2020] show that a transformer can learn to softly reason over natural language texts without the need to represent knowledge formally, but just by relying on the text as an expression of rules.

4.5.3 ML-powered Scaling

The application of ML models has recently resulted in significant advances in many hard domains [Silver et al., 2016] where hand-coded heuristics were once heavily dominating [Silver et al., 2018]. These hard domains include combinatorial optimisation problems [Bengio et al., 2018] such as Traveling Salesman Problem [Bello et al., 2016, Khalil et al., 2017], (Mixed) Integer Programming [Tang et al., 2019, Gasse et al., 2019], Boolean Satisfiability [Selsam and Bjørner, 2019] and computational graph optimisation [Paliwal et al., 2019], among others, program synthesis [Balog et al., 2017, Kalyan et al., 2018, Lee et al., 2018] and, of course, theorem proving.

Albeit gNTP is not a direct competitor to symbolic theorem provers such as ENIGMA [Jakubův and Urban, 2017], Vampire [Kovács and Voronkov, 2013] or E [Schulz, 2013], it is still noteworthy to emphasise that ML models have pushed the limits of theorem proving too. Notable examples include parametrised learning models for proof guidance learning [Loos et al., 2017, Kaliszyk et al., 2018] and premise selection [Kaliszyk and Urban, 2015, Irving et al., 2016, Wang et al., 2017a], and interestingly, even non-parametric models such as NNS found their use, in fact, selection [Blanchette et al., 2016].

(Approximate) Nearest Neighbour Search Efficient NNSs and ANNSs methods have successfully improved, scaled and speed up various machine learning tasks, including classification [Zaklouta et al., 2011], regression [Shen et al., 2006], clustering [Moore, 1999, Liu et al., 2007], retrieval [Xia et al., 2014], zero-shot learning [Palatucci et al., 2009], planning [Atramentov and LaValle, 2002], and reinforcement learning [Dulac-Arnold et al., 2015].

Most similar to our work is the recent work of Rae et al. [2016] who use ANNS to sparsify read and write operations in a memory-augmented network, as a means of achieving magnitudes of time and memory savings. They apply ANNSs to query an external memory, which is akin to our KB for the k closest words. As opposed to them, gNTP build sequences of proof paths using a chain of NNSs decisions. Similarly, Kaiser et al. [2017] use ANNS for scaling memory-augmented model to large memory sizes, using it not just for hard retrieval but incorporating it in the loss and ensuring that the ANNS result affects the key formation.

Maximum Inner Product Search Related to ANNS—a metric space -based search—is the Maximum Inner Product Search (MIPS)—an inner product space -based search. As opposed to finding the nearest vector in a metric space, MIPS aims at finding a vector that results in the highest inner product with a query vector. MIPS too has been used to scale and speed up ML models. Chandar et al. [2016] present a hierarchical memory network that exploits the k-MIPS for the attention-based reader, making the model scale to large memories, at a small cost to the accuracy. Spring and Shrivastava [2017] use hashing-based MIPS during learning to reduce the computation load to every layer of the model.

4.6 Conclusion and Future Work

The strong inductive bias of the backward chaining algorithm in NTPs enables them to combine the strengths of theorem proving and neural networks and deliver trainable reasoning over KBs. However, until now, they were not applicable to large KBs nor KBs enriched with natural language due to their prohibitive computational cost.

In this chapter, we propose gNTPs, a model that overcomes these limitations by greedily considering only a subset of all the proof paths NTPs would otherwise consider. We achieve this by limiting the proof paths to facts and rules containing k-nearest atoms in the embedding space. This pruning results in drastic speedups and memory efficiency, while, somewhat surprising, retaining the same performance or outperforming NTPs. In turn, the ability of gNTPs to scale opens up their application to the combination of structured and unstructured data. By embedding logical atoms and textual mentions in the same embedding space, gNTPs can successfully operate on natural language -enriched KBs, unlocking the possibilities of further research in this area that was not possible before.

Albeit the results are in general lower than those yielded by state-of-the-art Neural Link Predictors on both the small and large datasets, they are still very competitive and with the added benefits that gNTPs retain the induction of interpretable rules and can provide human-readable proof paths as explanations of its reasoning, at scale.

Future Work Even though gNTPs drastically prune the proof path enumeration, they still prune it equally at each depth, meaning they retain the exponential growth of the full enumeration. We can imagine **further improving the scaling capabilities** of these models in several ways. First, we can envision a dynamic beam search-alike enumeration strategy which enumerates only a pre-specified number of proof paths, bringing the memory complexity of the model further down into the linear region. Second, an MCTS-style approach to enumeration can bring the computational complexity of the model down, while enabling better exploration of the proof state space. Third, and somewhat orthogonal to the previous two, instead of the k-nearest neighbours, we can employ a learned model which locally decides when to greedily expand which rule at which depth. Such a model would enable us to side-step the requirements on the embeddings and enable the model to work with different scoring functions (e.g. ComplEx).

Logic-based reasoning suffers from the inability to represent model uncertainty, which can be of crucial importance in reasoning tasks. We want to **extend gNTPs to a probabilistic generative framework** that would enable us to deal with uncertainty and utilise bidirectional/non-logical inferences through the Bayes rule, enabling us to make predictions about unobserved relations and facts.

Finally, we showed how gNTPs could deal with natural language texts by jointly embedding text excerpts with KB triples into the shared embedding space. In future work, we want to push this further and be able to meaningfully and interpretably extract and embed facts and rules from full sentences, through a combination of model-guided parsing and joint embedding. Such an approach would enable us to use gNTPs as provable and interpretable **fact-checking** models applicable to text.

Chapter 5

Conclusions and Future Work

This thesis introduces differentiable interpreters—continuously relaxed analogues of traditional program interpreters—as an effective way to exploit programs as background knowledge and utilise them as strong inductive biases of neural networks. Differentiable interpreters are a single framework that enables continuous execution of programs on inputs, but crucially, enabling the use of gradient-based optimisation for inducing missing elements of the program or learning input representation, given data. In this thesis we present two differentiable interpreters, $\partial 4$ and gNTPs.

5.1 Contribution Summary

$\partial 4$: A Differentiable Forth Interpreter We introduce $\partial 4$, a fully differentiable interpreter for an imperative programming language Forth in Chapter 3. We introduce the notion of incomplete programs for $\partial 4$, differentiable sketches, as a means of providing program-driven strong inductive bias. Sketching provides a strong bias of known parts of the Forth program, enabling the application of gradient-based optimisation for learning the unknown parts. We apply sketching on learning parts of algorithms from data, applying them on learning to sort and learning to add tasks, as well for solving world algebra problems. The sketching approach enables training these models on a small number of input-output pairs yet achieving strong generalisation on these tasks. $\partial 4$ requires the execution of all commands of the program at each time step, in addition to following the full control flow of the program, making it computationally intensive. We present optimisations to reduce the load of the control flow, as its computational burden plagues longer programs.

gNTP: Greedy Neural Theorem Provers The need to run all commands at every time step, on the other hand, is a principal issue of the differentiable Datalog interpreter NTP. In order to learn representations of semantically similar symbols, NTPs compare all facts to all sub-goals during the backward-chaining proof path construction, making it impossible to scale to bigger KBs. To curb this issue, we present gNTP in Chapter 4, a model which greedily considers only paths involving representations of symbols closest to the current sub-goal symbols, drastically improving model efficiency and scaling as a result. Moreover, we widen the applicability of gNTP to textually-enriched KBs by giving it a compositional reading module, enabling it to jointly embed predicates and natural language texts in the same representation space. We experimentally quantify that gNTP perform as well as NTP at a fraction of the time and memory cost. The empirical demonstration of scaling enabled us to apply gNTP to large datasets, yielding performance comparable to related neuro-symbolic models, yet still lacking behind specialised link predictors. Finally, we demonstrated via qualitative analyses that the rules induced by the model should not be judged by the nearest-neighbour decoding process of Rocktaschel and Riedel [2017], but should be interpreted in conjunction with best-ranking proof paths.

5.2 Discussion and Future Work

Continuous relaxation of discrete machinery showed itself as a fruitful approach in many algorithm-learning tasks. Our contribution to this area showed that we can capitalise upon explicit algorithmic knowledge and directly "bake it in" the architecture of the model, making the model follow the program. We showed that we can do this for complex programs, as well as making this scale to large logic programs. However, the framework presented still exhibits several limits, notably issues with learning efficiency and scaling.

Training $\partial 4$ was not particularly straightforward and in most of our experiments standard vanilla optimisation was not enough, so we had to resort to optimisation tricks such as gradient noise and clipping. The model design had to incorporate particular design choices, for example, we found it impossible to train $\partial 4$ models which would directly predict an element of the model state without constraining it with a softmax. When learning the word algebra problems, we found that we had to sweep over a number of seeds in order to get

the best performing model. In addition, even though we did not focus on the inductive program synthesis task as a related differentiable interpreter TerpreT, we ran $\partial 4$ on a very small program synthesis problem and experienced the same problems as TerpreT. Concretely, we observed that it is quite difficult to synthesise even small programs due to the model getting easily trapped into local optima. We did not pursue this line of work further due to the conclusions made by TerpreT, showing that gradient optimisation as a local optimiser simply cannot cope with the synthesis task as exhausting solvers can. Similarly, we noticed that gNTP exhibits similar issues with learning; in some experiments, we noticed it was necessary to pre-train rules in order to achieve better performance. This showed us that there is still room for optimisation improvement which could be a basis for future work.

On the other hand, $\partial 4$ simply does not scale for long executions as it shares standard RNN issues with diminishing gradients over long timescales. Similarly, gNTP still does not scale for larger proof depths, due to its exponential growth. Though some of these issues could be alleviated with more elaborate models and mechanisms, these would still have a limited range of success.

Still, we think there are a few useful takeaways from our research.

Language Choice We see language choice is crucial in modelling differentiable interpreters. Not only does the language dictate the details of the continuous approximation, but it also dictates the size of programs the interpreter can handle and the magnitude of the control flow that it executes. We strongly recommend that the choice of the *host language should be task-oriented*.

We initially intended to use $\partial 4$ to construct a knowledge base theorem prover by injecting the recursive structure of theorem proving, leaving either (or both) the unification or the input representations to be learned. However, we quickly realised that the code, and even more importantly the control flow, would explode in size for anything other than a trivial unit test. In this case, a differentiable interpreter for a language fit for the logical reasoning task seemed like a better fit, bringing our attention to NTP. Though still computationally intensive, a differentiable interpreter for Datalog is still a far better performing alternative to an equivalent $\partial 4$ sketch, not just because a logic language offers a succinct representation for KBs, but importantly the execution of such programs is inherently parallel—$\partial 4$ is sequentially recurrent whereas NTP is fully recursive. This is a great fit for differentiable interpretation on a GPU,

up to a point, which we successfully extended with gNTPs.

Likewise, Manhaeve et al. [2018] showed that the programs we presented in Chapter 3 are in general smaller when expressed as a logic program, and also produce a shorter control flow resulting in shorter execution time.

Similarly, should the task require (efficient) graph traversals, such as multi-hop question answering over KBs, a differentiable interpreter for a graph traversal Domain Specific Language (DSL) should be an excellent choice [Cohen et al., 2019, 2020].

Fitness for a Task Task-wise, we see that $\partial 4$ holds ground on strongly biasing the model with an algorithm, as well as gNTP works reasonably well for the KB completion tasks, yet work done by Gaunt et al. [2016] and Feser et al. [2017] shows that differentiable interpreters are simply not up to the task of inductive program synthesis.

We still think, however, that there are promising tasks and promising new languages on the horizon. Cohen et al. [2019] present Neural Query Language, essentially a differentiable graph traversal interpreter, showing exciting results on traversing huge graphs and learning relationships between nodes in the graph. We think a similar approach can be expanded with representation learning to enable large-scale associative memory recall.

On the other hand, even though the conclusions of Gaunt et al. [2016] indispose differentiable interpreters for program synthesis, Gaunt et al. [2017] show that there is still use for differentiable interpreters when synthesising (small) programs involving continuous inputs. We hypothesise that this, or similar approach integrating non-differentiable actions, can be used to synthesise programs, i.e. structure of quantum neural networks [Ostaszewski et al., 2019].

5.3 The Outlook

Having in mind both the benefits and the difficulties brought by differentiable interpreters, we are still optimistic that the elements of this approach, concretely the continuous relaxation of computational primitives, will still find use in modern architectures, and we are hoping that further research down the line will shine a light on how to effectively use them.

We hope $\partial 4$ and the paradigm it helped establish will find its use cases in

domains where tasks dictate dense representations, with high-level processing difficult to learn, but possibly easier to, even if partially, specify libraries of sketches. These libraries could enable faster learning of components for quick master and learning of difficult-to-learn tasks such as computation, reasoning, optimisation and making differentiable counterparts of non-differentiable losses.

As for gNTP, we hope to see it as a step towards scalable and interpretable theorem proving applicable on natural language.

Appendix A

Appendix to ∂4

A.1 Forth Instruction Set

In this section, we describe in detail the FORTH instruction set briefly presented in Section 3.2.2.

Description Format In order to explain the operational effect of each word, in lieu of mathematical notation or the language of operational semantics [Lucas, 1978], we use a a more pragmatic technical approach FORTH literature uses to describe words. This format focuses on explaining the effect of the word on the data stack or the return stack, while describing other behaviour internal to the interpreter in a free-form text. The effect of each word is described with the following template:

WORD `D: ( x -- y )` `R: ( x -- y )`

This is the free-form description of the `WORD` word. `( x -- y )` denotes the before (`x`) and after (`y`) state of a top of the data stack (`D:`) or the return stack (`R:`) that is necessary to understand the command, obviating the need for presenting the full state of stacks. For example, `( x -- x*2 )` denotes that the top of the data stack doubled in value, `( x -- )` denotes the removal of the top of the stack, and `( -- x )` denotes insertion of a new value on the stack.

Data stack operations

x D: (-- x)

Pushes the integer literal x to the data stack.

DROP D: (x --)

Pops the data stack TOS (non-destructive).[1]

DUP D: (x -- x x)

Duplicates the data stack TOS.

SWAP D: (x y -- y x)

Swaps the TOS and NOS on the data stack.

OVER D: (x y -- x y x)

Pushes a copy of NOS as the new TOS.

1+ D: (x -- x+1)

Increments the data stack TOS.

1- D: (x -- x-1)

Decrements the data stack TOS.

+ D: (x y -- x+y)

Addition operation—drops the data stack NOS and TOS and pushes NOS+TOS to the data stack.

- D: (x y -- x-y)

Subtraction operation—drops the data stack NOS and TOS and pushes NOS-TOS to the data stack.

***** D: (x y -- x*y)

Multiplication operation—drops the data stack NOS and TOS and pushes NOS*TOS to the data stack.

/ D: (x y -- x/y)

Division operation—drops the data stack NOS and TOS and pushes NOS/TOS to the data stack.

[1] The TOS value is still in memory, but the stack pointer is pointing to a value below it.

> D: (x y -- c)

Greater than comparator—drops the data stack NOS and TOS and pushes literal `c` to TOS. `c` is set to 1 if $x > y$ and 0 otherwise.[2]

< D: (x y -- c)

Less than comparator—drops the data stack NOS and TOS and pushes literal `c` to TOS. `c` is set to 1 if $x < y$ and 0 otherwise.

= D: (x y -- c)

Equality comparator—drops the data stack NOS and TOS and pushes literal `c` to TOS. `c` is set to 1 if $x = y$ and 0 otherwise.

Return stack operations

>R D: (x --)
R: (-- x)

Pops the data stack TOS and pushes it to the return stack.

R> D: (-- x)
R: (x --)

Pops the return stack TOS and pushes it to the data stack.

@R D: (-- x)
R: (x -- x)

Pushes a copy of the return stack TOS to the data stack.

Heap operations

@ D: (addr -- x)

Takes the data stack TOS value as the heap address from which it fetches the value `x` which it then pushes on the stack.

! D: (x addr --)

Takes the data stack NOS value as the value, and TOS as the address. It then stores the value onto the heap under that address.

Control statements

Please note that each token (keyword) of these commands carries its own separate semantics but we explain them together as their combination easily relates to usual control statements in other languages `..x` denotes

[2] In general, FORTH interprets any non-zero value as TRUE and zero as FALSE.

a sequence of words, and `?` denotes effect contingent on the condition at hand and the executed sequence of words.

`IF ..1 THEN` `D: ( flag -- ? )`

Pops the `flag` value off the data stack. If its value is TRUE (non-zero value), it executes `..1`.

`IF ..1 ELSE ..2 THEN` `D: ( flag -- ? )`

Same as above, but if `flag` is FALSE (zero) it executes `..2`.

`BEGIN ..1 WHILE ..2 REPEAT` `D: ( -- ? )`

Executes `..1`. If the (resulting) data stack TOS is TRUE, it executes `..2`, upon which it goes back to `..1`. However, if the TOS value is FALSE, it does not execute `..2` at all and resumes with following words.

`DO ..1 LOOP` `D: ( limit index -- ? )`

Pops the data stack NOS and TOS as the limit and index, and executes `..1`. Upon completion, increments the index and checks whether the index equals to the limit. If it does, continues with following words, otherwise it pushes the limit and the incremented index back to the data stack, and redirects control back to the beginning of the loop (after DO).

Subroutine control

`: SUB ..1 ;`

Defines a new word named `SUB`, as the sequence of words `..1`. It effectively creates an entry in the interpreter dictionary which points to the word following it. `:` denotes the start of the subroutine/word `SUB` as a sequence of words `..1`, and `;` signifies the end of the subroutine.

`SUB`

`R: ( -- pc-addr )`

Subroutine/word invocation—the word `SUB` is invoked by pushing the program counter c on the return stack, and assigning the address of the word following the definition of `SUB` to c as the new program counter.

Variable creation

`VARIABLE VAR`

Creates a variable named `VAR` by reserving a single address on the heap. It then binds the value of that address to the word `VAR`.

`CREATE VAR ALLOT NUM`

Creates an array variable named `VAR` by reserving `NUM` sequential addresses on the heap. It then binds the value of the first of the addresses to the word `VAR`. This effectively reserves an uninterrupted portion of the heap for the variable.

`VAR` `D: ( -- var-addr )`

Invoking a variable `VAR` by its name just returns the address of the variable.

Other

`NOP` `D: ( -- )`

No-operation. Nothing is executed.

`MACRO: SUB .. ;`

Treats the word `SUB` as a macro—akin to a preprocessing directive, it replaces the `SUB` token with `..` in the source code, before interpretation.

Appendix B

Appendix to gNTP

Table B.1: The set of textual mentions replacing the variable number of training triples in Section 4.4.5 for the `locatedIn` and the `neighborOf` predicates.

`locatedIn(X, Y)`		
[#1] can be found in [#2]	[#1] is based in [#2]	[#1] is contained in [#2]
[#1] is currently in [#2]	[#1] is found in [#2]	[#1] is in [#2]
[#1] is localized in [#2]	[#1] is located in [#2]	[#1] is placed in [#2]
[#1] is positioned in [#2]	[#1] is present in [#2]	[#1] is set in [#2]
[#1] is sited in [#2]	[#1] is situated in [#2]	[#1] is still in [#2]
[#1] used to be found in [#2]	[#1] was based in [#2]	[#1] was contained in [#2]
[#1] was currently in [#2]	[#1] was found in [#2]	[#1] was in [#2]
[#1] was localized in [#2]	[#1] was located in [#2]	[#1] was placed in [#2]
[#1] was positioned in [#2]	[#1] was present in [#2]	[#1] was set in [#2]
[#1] was sited in [#2]	[#1] was situated in [#2]	[#1] was still in [#2]

`neighborOf(X, Y)`		
[#1] borders with [#2]	[#1] borders [#2]	[#1] is a neighbor of [#2]
[#1] is a neighboring country of [#2]	[#1] is a neighboring state to [#2]	[#1] is adjacent to [#2]
[#1] is bordering [#2]	[#1] is butted against [#2]	[#1] is closest to [#2]
[#1] is nearest country to [#2]	[#1] is nearest to [#2]	[#1] is next to [#2]
[#1] is positioned closest to [#2]	[#1] is positioned next to [#2]	[#1] is right next to [#2]
[#1] neighbours with [#2]	[#1] neighbours [#2]	[#1] was a neighbor of [#2]
[#1] was a neighboring country of [#2]	[#1] was a neighboring state to [#2]	[#1] was adjacent to [#2]
[#1] was bordering [#2]	[#1] was butted against [#2]	[#1] was closest to [#2]
[#1] was nearest country to [#2]	[#1] was nearest to [#2]	[#1] was next to [#2]
[#1] was positioned closest to [#2]	[#1] was positioned next to [#2]	[#1] was right next to [#2]

Bibliography

Martín Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. TensorFlow Eager: A Multi-Stage, Python-Embedded DSL for Machine Learning. In *Proceedings of Systems for Machine Learning (SysML)*, 2019.

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers: Principles, Techniques. *Addison Wesley*, 7(8):9, 1986.

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive Program Synthesis. In *Proceedings of International Conference on Computer Aided Verification*, pages 934–950. Springer, 2013.

Brandon Amos and J Zico Kolter. OptNet: Differentiable Optimization as a Layer in Neural Networks. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 136–145. JMLR.org, 2017.

Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and Optimal LSH for Angular Distance. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 1225–1233, 2015.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to Compose Neural Networks for Question Answering. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1545–1554, 2016a.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural

Module Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 39–48, 2016b.

ANSI. ANSI X3.215-1994: Programming Languages—Forth. Standard, American National Standards Institute, Inc., 1994.

Sanjeev Arora, Yingyu Liang, and Tengyu Ma. A Simple but Tough-to-Beat Baseline for Sentence Embeddings. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

Anna Atramentov and Steven M LaValle. Efficient nearest neighbor searching for motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 1, pages 632–637, 2002.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2015.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to Write Programs. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

Michele Banko, Michael J Cafarella, Stephen Soderland, Matthew Broadhead, and Oren Etzioni. Open information extraction from the web. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2007.

Peter W. Battaglia et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Atılım Günes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

Richard E Bellman. *Adaptive Control Processes: A Guided Tour*, volume 2045. Princeton university press, 2015.

Richard Bellmann. Dynamic Programming. *Princeton, NJ*, 1957.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning Long-Term De-

pendencies with Gradient Descent is Difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *arXiv preprint arXiv:1811.06128*, 2018.

Jon Louis Bentley. A survey of techniques for fixed radius near neighbor searching. Technical report, Stanford University, 1975.

Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When Is âĂIJNearest NeighborâĂİ Meaningful? In *International conference on database theory*, pages 217–235. Springer, 1999.

Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *Journal of Automated Reasoning*, 57(3):219–244, 2016.

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 1247–1250, 2008.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-relational Data. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 2787–2795, 2013.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. A Neural Forth Abstract Machine. In *Neural Abstract Machines & Program Induction (NAMPI) Workshop @ NIPS*, 2016.

Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. Programming with a Differentiable Forth Interpreter. In *Proceedings of International Conference on Machine Learning (ICML)*, volume 70, pages 547–556, 2017.

Matko Bošnjak*, Pasquale Minervini*, Andres Campero, Tim Rocktaschel, Edward Grefenstette, and Sebastian Riedel. Neural Theorem Proving on Natural Language . In *The International Conference on Probabilistic Programming*, 2018.

Guillaume Bouchard, Sameer Singh, and Theo Trouillon. On Approximate Reasoning Capabilities of Low-Rank Vector Spaces. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2015.

Guillaume Bouchard, Pontus Stenetorp, and Sebastian Riedel. Learning to Generate Textual Data. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1608–1616, 2016.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Leo Brodie. *Thinking Forth*. Punchy Publishing, 2004. ISBN 0976458705.

Leo Brodie and CORPORATE FORTH Inc. *Starting FORTH, 2nd Ed.* Prentice-Hall, Inc., 1987.

David S Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, Royal Signals and Radar Establishment Malvern (United Kingdom), 1988.

Rudy R Bunel, Alban Desmaison, Pawan K Mudigonda, Pushmeet Kohli, and Philip Torr. Adaptive Neural Compilation. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 1444–1452, 2016.

Razvan Bunescu and Raymond Mooney. Learning to Extract Relations from the Web using Minimal Supervision. In *Proceedings of Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 576–583, 2007.

Andres Campero, Aldo Pareja, Tim Klinger, Josh Tenenbaum, and Sebastian Riedel. Logical Rule Induction and Theory Learning Using Neural Theorem Proving. *arXiv preprint arXiv:1809.02193*, 2018.

Augustin Cauchy. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.

Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE transactions on knowledge and data engineering*, 1(1):146–166, 1989.

SC Chan, LS Hsu, S Brody, and HH Teh. Neural three-valued-logic networks.

In *International 1989 Joint Conference on Neural Networks*, pages 594–vol. IEEE, 1989.

SC Chan, LS Hsu, KF Loe, and HH Teh. Neural Logic Networks. *Progress in Neural Networks*, 2, 1993.

Sarath Chandar, Sungjin Ahn, Hugo Larochelle, Pascal Vincent, Gerald Tesauro, and Yoshua Bengio. Hierarchical Memory Networks. *arXiv preprint arXiv:1605.07427*, 2016.

Kai-Wei Chang, Wen-tau Yih, Bishan Yang, and Christopher Meek. Typed Tensor Decomposition of Knowledge Bases for Relation Extraction. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1568–1579, 2014.

Swarat Chaudhuri and Armando Solar-Lezama. Smooth Interpretation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 279–291, 2010.

Swarat Chaudhuri and Armando Solar-Lezama. Smoothing a Program Soundly and Robustly. In *Proceedings of Computer Aided Verification (CAV)*, pages 277–292, 2011.

Yang Chen, Sean Goldberg, Daisy Zhe Wang, and Soumitra Siddharth Johri. Ontological Pathfinding. In *Proceedings of the 2016 International Conference on Management of Data*, pages 835–846, 2016.

François Chollet. The Measure of Intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Alonzo Church. Application of Recursive Arithmetic to the Problem of Circuit Synthesis. *Institute for Symbolic Logic, Cornell University*, 1957.

Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition. *Neural computation*, 22(12):3207–3220, 2010.

Peter Clark, Oyvind Tafjord, and Kyle Richardson. Transformers as Soft Reasoners over Language. *arXiv preprint arXiv:2002.05867*, 2020.

Nadav Cohen and Amnon Shashua. Inductive Bias of Deep Convolutional Networks through Pooling Geometry. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

Nadav Cohen, Or Sharir, Yoav Levine, Ronen Tamari, David Yakira, and Amnon Shashua. Analysis and Design of Convolutional Networks via Hierarchical Tensor Decompositions. *Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI) Special Issue on Deep Learning Theory*, 2017.

William W Cohen. TensorLog: A Differentiable Deductive Database. *arXiv preprint arXiv:1605.06523*, 2016.

William W Cohen, Matthew Siegler, and Alex Hofer. Neural Query Language: A Knowledge Base Query Language for Tensorflow. *arXiv preprint arXiv:1905.06209*, 2019.

William W Cohen, Haitian Sun, R Alex Hofer, and Matthew Siegler. Scalable Neural Methods for Reasoning With a Symbolic Knowledge Base. *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

Mark Craven, Andrew McCallum, Dan PiPasquo, Tom Mitchell, and Dayne Freitag. Learning to Extract Symbolic Knowledge from the World Wide Web. Technical report, Carnegie Mellon University, 1998.

Maxwell John Cresswell. *Logics and Languages*. Routledge, 1973.

Andrew Cropper and Stephen H Muggleton. Logical minimisation of meta-rules within Meta-Interpretive Learning. In *Inductive Logic Programming*, pages 62–75. Springer, 2015.

Andrew Cropper and Stephen H Muggleton. Learning Higher-Order Logic Programs through Abstraction and Invention. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1418–1424, 2016.

Rajarshi Das, Shehzaad Dhuliawala, Manzil Zaheer, Luke Vilnis, Ishan Durugkar, Akshay Krishnamurthy, Alex Smola, and Andrew McCallum. Go for a Walk and Arrive at the Answer: Reasoning Over Paths in Knowledge Bases using Reinforcement Learning. *arXiv preprint arXiv:1711.05851*, 2017a.

Rajarshi Das, Arvind Neelakantan, David Belanger, and Andrew McCallum. Chains of Reasoning over Entities, Relations, and Text using Recurrent Neural Networks. *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 132–141, 2017b.

Sreerupa Das, C Lee Giles, and Guo-Zheng Sun. Learning Context-free Grammars: Capabilities and Limitations of a Recurrent Neural Network with an External Stack Memory. In *Proceedings of The Fourteenth Annual Conference of Cognitive Science Society. Indiana University*, page 14, 1992.

Sreerupa Das, C Lee Giles, and Guo-Zheng Sun. Using Prior Knowledge in a NNPDA to Learn Context-Free Languages . In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 65–72, 1993.

Donald Davidson. Truth and Meaning. In *Philosophy, Language, and Artificial Intelligence*, pages 93–111. Springer, 1967.

Jesse Davis and Mark Goadrich. The relationship between Precision-Recall and ROC curves. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 233–240. ACM, 2006.

Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-End Differentiable Physics for Learning and Control. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 7178–7189, 2018.

Michiel de Jong and Fei Sha. Neural Theorem Provers Do Not Learn Rules Without Exploration. *arXiv preprint arXiv:1906.06805*, 2019.

Guillaume François Antoine Marquis de l'Hôpital. *Analyse des infiniment petits pour l'intelligence des lignes courbes*. François Montalant, Quay des Augustins, 1696.

Luc De Raedt and Kristian Kersting. Probabilistic Inductive Logic Programming. In *Probabilistic Inductive Logic Programming*, pages 1–27. Springer, 2008.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, volume 7, pages 2462–2467. Hyderabad, 2007.

Jonas Degrave, Michiel Hermans, and Joni Dambre. A Differentiable Physics Engine for Deep Learning in Robotics. *Frontiers in neurorobotics*, 13, 2019.

Olivier Delalleau and Yoshua Bengio. Shallow vs. Deep Sum-Product Net-

works. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 666–674, 2011.

Richard A. DeMillo and Richard J. Lipton. Defining Software by Continuous, Smooth Functions. *IEEE transactions on software engineering*, 17(4):383–384, 1991.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Woodrow W Denham. *The Detection of Patterns in Alyawarra Nonverbal Behavior* . PhD thesis, University of Washington, Seattle., 1973.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2D Knowledge Graph Embeddings. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2018.

Frank DiMaio and Jude Shavlik. Learning an Approximation to Inductive Logic Programming Clause Evaluation. In *International Conference on Inductive Logic Programming*, pages 80–97. Springer, 2004.

Liya Ding. Neural prolog-the concepts, construction and mechanism. In *Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on*, volume 4, pages 3603–3608. IEEE, 1995.

Liya Ding, Hoon Heng Teh, Peizhuang Wang, and Ho Chung Lui. A Prolog-like inference system based on neural logic âĂŤ An attempt towards fuzzy neural logic programming. *Fuzzy Sets and Systems*, 82(2):235–251, 1996.

Peter Dinges and Gul Agha. Targeted Test Input Generation using Symbolic-Concrete Backward Execution. In *Proceedings of the ACM/IEEE international conference on Automated software engineering*, pages 31–36, 2014.

Josip Djolonga and Andreas Krause. Differentiable Learning of Submodular Models. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 1013–1023, 2017.

Ivan Donadello, Luciano Serafini, and Artur D'Avila Garcez. Logic Tensor Networks for Semantic Image Interpretation. *arXiv preprint arXiv:1705.08968*, 2017.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural Logic Machines. *arXiv preprint arXiv:1904.11694*, 2019.

Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 601–610. ACM, 2014.

Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep Reinforcement Learning in Large Discrete Action Spaces. *arXiv preprint arXiv:1512.07679*, 2015.

Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

Ronen Eldan and Ohad Shamir. The Power of Depth for Feedforward Neural Networks. In *Conference on learning theory*, pages 907–940, 2016.

Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. Unsupervised Learning by Program Synthesis. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 973–981, 2015.

Jeffrey L Elman. Finding Structure in Time. *Cognitive science*, 14(2):179–211, 1990.

Oren Etzioni, Michele Banko, and Michael J Cafarella. Machine Reading. In *Proceedings of AAAI Conference on Artificial Intelligence*, volume 6, pages 1517–1519, 2006.

Richard Evans and Edward Grefenstette. Learning Explanatory Rules from Noisy Data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

Richard Evans, Jose Hernandez-Orallo, Johannes Welbl, Pushmeet Kohli, and Marek Sergot. Making sense of sensory input. *arXiv preprint arXiv:1910.02227*, 2019.

Aaron Ferber, Bryan Wilder, Bistra Dilina, and Milind Tambe. MIPaaL: Mixed Integer Program as a Layer. *arXiv preprint arXiv:1907.05912*, 2019.

A Ferrari, A Batson, M Lack, A Jones, D Evans, and Q Carbonneaux. x86

Assembly Guide. *Yale University,[En línea]. Available: http://flint. cs. yale. edu/cs421/papers/x86-asm/asm. html*, 2006.

John K Feser, Marc Brockschmidt, Alexander L Gaunt, and Daniel Tarlow. Neural Functional Programming. In *Proceedings of International Conference on Learning Representations (ICLR) (Workshop track)*, 2017.

Pedro Fialho, Sérgio Curto, Ana Cristina Mendes, and Luisa Coheur. Wordnet framework improvements for nlp: Defining abstraction and scalability layers. Technical report, Technical report, Technical Report 8113, Spoken Language Systems Lab âĂę, 2011.

Evelyn Fix. *Discriminatory analysis: nonparametric discrimination, consistency properties*. USAF school of Aviation Medicine, 1951.

Manoel VM França, Gerson Zaverucha, and Artur S dâĂŹAvila Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine Learning*, 94(1):81–104, 2014.

Stan Franklin and Max Garzon. Neural Computability. *Progress in neural networks*, 1(128,144), 1990.

Kunihiko Fukushima. Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. *Biological cybernetics*, 36(4):193–202, 1980.

Jasmine S. Furter and Peter C. Hauser. Interactive control of purpose built analytical instruments with Forth on microcontrollers - A tutorial. *Analytica Chimica Acta*, 2018.

Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. AMIE: Association Rule Mining under Incomplete Evidence in Ontological Knowledge Bases. In *Proceedings of the 22nd international conference on World Wide Web*, pages 413–422, 2013.

Artur d'Avila Garcez, Tarek R Besold, Luc De Raedt, Peter Földiak, Pascal Hitzler, Thomas Icard, Kai-Uwe Kühnberger, Luis C Lamb, Risto Miikkulainen, and Daniel L Silver. Neural-Symbolic Learning and Reasoning: Contributions and Challenges. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2015.

Artur d'Avila Garcez, Marco Gori, Luis C Lamb, Luciano Serafini, Michael

Spranger, and Son N Tran. Neural-Symbolic Computing: An Effective Methodology for Principled Integration of Machine Learning and Reasoning. *arXiv preprint arXiv:1905.06088*, 2019.

Artur S Avila Garcez and Gerson Zaverucha. The Connectionist Inductive Learning and Logic Programming System. *Applied Intelligence*, 11(1):59–77, 1999.

Artur S d'Avila Garcez, Krysia B Broda, and Dov M Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Springer Science & Business Media, 2012.

Artur S d'Avila Garcez, Dov M Gabbay, and Luis C Lamb. A neural cognitive model of argumentation with application to legal inference and decision making. *Journal of Applied Logic*, 12(2):109–127, 2014.

Artur S dâĂŹAvila Garcez, Dov M Gabbay, Oliver Ray, and John Woods. Abductive Reasoning in Neural-Symbolic Systems. *Topoi*, 26(1):37–49, 2007.

Artur SD'Avila Garcez, Luis C Lamb, and Dov M Gabbay. *Neural-Symbolic Cognitive Reasoning*. Springer Science & Business Media, 2008.

Alberto Garcia-Duran and Mathias Niepert. KBLRN : End-to-End Learning of Knowledge Base Representations with Latent, Relational, and Numerical Features. *arXiv preprint arXiv:1709.04676*, 2017.

Matt Gardner and Tom Mitchell. Efficient and Expressive Knowledge Base Completion Using Subgraph Feature Extraction. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1488–1498, 2015.

Matt Gardner, Partha Talukdar, Bryan Kisiel, and Tom Mitchell. Improving Learning and Inference in a Large Knowledge-base using Latent Syntactic Cues. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 833–838, 2013.

Matt Gardner, Partha Talukdar, Jayant Krishnamurthy, and Tom Mitchell. Incorporating Vector Space Similarity in Random Walk Inference over Knowledge Bases. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 397–406, 2014.

Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea

Lodi. Exact Combinatorial Optimization with Graph Convolutional Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, pages 15554–15566, 2019.

Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. TerpreT: A Probabilistic Programming Language for Program Induction. *arXiv preprint arXiv:1608.04428*, 2016.

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable Programs with Neural Libraries. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 1213–1222, 2017.

C. Lee Giles, Guo-Zheng Sun, Hsing-Hen Chen, Yee-Chun Lee, and Dong Chen. Higher Order Recurrent Networks and Grammatical Inference. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 380–387, 1990.

Philip E Gill, Walter Murray, and Michael A Saunders. SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization. *SIAM review*, 47(1): 99–131, 2005.

Attilio Giordana, Lorenza Saitta, and Floriano Zini. Learning Disjunctive Concepts by Means of Genetic Algorithms. In *Machine Learning Proceedings 1994*, pages 96–104. Elsevier, 1994.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 2, pages 729–734. IEEE, 2005.

Richard E Grandy. Understanding and the Principle of Compositionality. *Philosophical Perspectives*, 4:557–572, 1990.

Alex Graves, Santiago Fernández, and Jürgen Schmidhuber. Bidirectional

LSTM Networks for Improved Phoneme Classification and Recognition. In *International Conference on Artificial Neural Networks*, pages 799–804. Springer, 2005.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

C Cordell Green and Bertram Raphael. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 1968 23rd ACM national conference*, pages 169–181. ACM, 1968.

Edward Grefenstette. Towards a Formal Distributional Semantics: Simulating Logical Calculi with Tensors. *arXiv preprint arXiv:1304.5823*, 2013.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to Transduce with Unbounded Memory. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 1828–1836, 2015.

Andreas Griewank. Who Invented the Reverse Mode of Differentiation. *Documenta Mathematica, Extra Volume ISMP*, pages 389–400, 2012.

Frédéric Gruau, Jean-Yves Ratajszczak, and Gilles Wiber. A Neural Compiler. *Theoretical Computer Science*, 141(1):1 – 52, 1995.

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program Synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Shu Guo, Quan Wang, Lihong Wang, Bin Wang, and Li Guo. Jointly Embedding Knowledge Graphs and Logical Rules. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 192–202, 2016.

Nitish Gupta, Kevin Lin, Dan Roth, Sameer Singh, and Matt Gardner. Neural

Module Networks for Reasoning over Text. *arXiv preprint arXiv:1912.04971*, 2019.

Jacques Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Université de Paris, 1930.

Alexander Hinneburg, Charu C Aggarwal, and Daniel A Keim. What is the nearest neighbor in high dimensional spaces? In *26th Internat. Conference on Very Large Databases*, pages 506–515, 2000.

Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, 2006.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Raphael Hoffmann, Congle Zhang, Xiao Ling, Luke Zettlemoyer, and Daniel S Weld. Knowledge-Based Weak Supervision for Information Extraction of Overlapping Relations. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 541–550, 2011.

Steffen Hölldobler, Yvonne Kalinke, and Hans-Peter Störr. Approximating the Semantics of Logic Programs by Recurrent Neural. *Applied Intelligence*, 11 (1):45–58, 1999.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer Feedforward Networks are Universal Approximators. *Neural networks*, 2(5):359–366, 1989.

Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to Reason: End-To-End Module Networks for Visual Question Answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 804–813, 2017.

Jeevana Priya Inala, Sicun Gao, Soonho Kong, and Armando Solar-Lezama. REAS: Combining Numerical Optimization with SAT Solving. *arXiv preprint arXiv:1802.04408*, 2018.

Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. DeepMath - Deep Sequence Models for Premise Selection. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 2235–2243, 2016.

Aleksei GrigorÊźevich Ivakhnenko and Valentin Grigorévich Lapa. Cybernetic Predicting Devices. Technical report, PURDUE UNIV LAFAYETTE IND SCHOOL OF ELECTRICAL ENGINEERING, 1966.

Jan Jakubův and Josef Urban. ENIGMA: Efficient Learning-Based Inference Guiding Machine. In *International Conference on Intelligent Computer Mathematics*, pages 292–302. Springer, 2017.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

Yichen Jiang and Mohit Bansal. Self-Assembling Modular Networks for Interpretable Multi-Hop Reasoning. *arXiv preprint arXiv:1909.05803*, 2019.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734*, 2017.

Michael I Jordan. Serial Order: A Parallel Distributed Processing Approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier, 1997.

Armand Joulin and Tomas Mikolov. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 190–198, 2015.

Rudolf Kadlec, Ondrej Bajgar, and Jan Kleindienst. Knowledge Base Completion: Baselines Strike Back. *arXiv preprint arXiv:1705.10744*, 2017.

Łukasz Kaiser and Ilya Sutskever. Neural GPUs Learn Algorithms. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

Łukasz Kaiser, Ofir Nachum, Aurko Roy, and Samy Bengio. Learning to Remember Rare Events. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2017.

Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A Convolutional

Neural Network for Modelling Sentences. *arXiv preprint arXiv:1404.2188*, 2014.

Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *Journal of symbolic computation*, 69:109–128, 2015.

Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement Learning of Theorem Proving. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 8822–8833, 2018.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. *arXiv preprint arXiv:1804.01186*, 2018.

Jerrold J Katz and Jerry A Fodor. The Structure of a Semantic Theory. *language*, 39(2):170–210, 1963.

Charles Kemp, Joshua B Tenenbaum, Thomas L Griffiths, Takeshi Yamada, and Naonori Ueda. Learning Systems of Concepts with an Infinite Relational Model. In *Proceedings of AAAI Conference on Artificial Intelligence*, volume 3, page 5, 2006.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning Combinatorial Optimization Algorithms over Graphs. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2017.

James C King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2015.

Emanuel Kitzelmann. Inductive Programming: A Survey of Program Synthesis Techniques. In *International workshop on approaches and applications of inductive programming*, pages 50–73. Springer, 2009.

Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

Peter J Knaggs. *Practical and Theoretical Aspects of Forth Software Development*. PhD thesis, University of Teesside, 1993.

Stanley Kok and Pedro Domingos. Statistical predicate invention. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 433–440. ACM, 2007.

Andrei Nikolaevich Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. In *Doklady Akademii Nauk*, volume 114, pages 953–956. Russian Academy of Sciences, 1957.

Ekaterina Komendantskaya. First-order Deduction in Neural Networks. In *LATA*, pages 307–318, 2007.

Ekaterina Komendantskaya. Unification neural networks: unification by error-correction learning. *Logic Journal of the IGPL*, 19(6):821–847, 2011.

Rik Koncel-Kedziorski, Hannaneh Hajishirzi, Ashish Sabharwal, Oren Etzioni, and Siena Dumas Ang. Parsing Algebraic Word Problems into Equations. *Transactions of the Association for Computational Linguistics (TACL)*, 3: 585–597, 2015.

Philip J. Koopman, Jr. A Brief Introduction to Forth. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 357–358. ACM, 1993.

Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

John R Koza and John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, volume 1. MIT press, 1992.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural Random Access Machines. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

Nate Kushman, Yoav Artzi, Luke Zettlemoyer, and Regina Barzilay. Learning to Automatically Solve Algebra Word Problems. In *Proceedings of Annual*

Meeting of the Association for Computational Linguistics (ACL), pages 271–281, 2014.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building Machines That Learn and Think Like People. *Behavioral and brain sciences*, 40, 2017.

Ni Lao and William W Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine Learning*, 81(1):53–67, 2010.

Ni Lao, Tom Mitchell, and William W Cohen. Random Walk Inference and Learning in A Large Scale Knowledge Base. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 529–539, 2011.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural computation*, 1(4):541–551, 1989.

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449, 2018.

Gottfried Wilhelm Leibniz. Memoir using the chain rule. *Cited in TMME*, 7: 321–332, 1676.

Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable Monte Carlo ray tracing through edge sampling. In *SIGGRAPH Asia 2018 Technical Papers*, page 222. ACM, 2018.

Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate Nearest Neighbor Search on High Dimensional Data âĂŤ Experiments, Analyses, and Improvement. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning Entity and Relation Embeddings for Knowledge Graph Completion. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2015.

Seppo Linnainmaa. The representation of the cumulative rounding error of

an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

Ting Liu, Charles Rosenberg, and Henry A Rowley. Clustering Billions of Images with Large Scale Nearest Neighbor Search. In *2007 IEEE Workshop on Applications of Computer Vision (WACV'07)*, pages 28–28. IEEE, 2007.

Huma Lodhi. Deep Relational Machines. In *International Conference on Neural Information Processing*, pages 212–219. Springer, 2013.

Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep Network Guided Proof Search. *arXiv preprint arXiv:1701.06972*, 2017.

Matthew M Loper and Michael J Black. OpenDR: An Approximate Differentiable Renderer. In *European Conference on Computer Vision*, pages 154–169. Springer, 2014.

Peter Lucas. On the formalization of programming languages: Early history and main approaches. In *The Vienna Development Method: The Meta-Language*, pages 1–23. Springer, 1978.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective Approaches to Attention-based Neural Machine Translation. *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1412–1421, 2015.

Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based Hyperparameter Optimization through Reversible Learning. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 2113–2122, 2015a.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless Gradients in Numpy. In *ICML 2015 AutoML Workshop*, volume 238, 2015b.

Ronald Mak and Tom Copeland. *Writing compilers and interpreters*. Wiley New York, 1996.

Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir

Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

Yury A Malkov and Dmitry A Yashunin. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. DeepProbLog: Neural Probabilistic Logic Programming. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 3749–3759, 2018.

Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

Gary Marcus. Deep Learning: A Critical Appraisal. *arXiv preprint arXiv:1801.00631*, 2018.

Gary Marcus. The Next Decade in AI: Four Steps Towards Robust Artificial Intelligence, 2020.

Giuseppe Marra, Michelangelo Diligenti, Francesco Giannini, Marco Gori, and Marco Maggini. Relational Neural Machines. *arXiv preprint arXiv:2002.02193*, 2020.

TB Martin and JJ Talavage. Application of Neural Logic to Speech Analysis and Recognition. *IEEE Transactions on Military Electronics*, MIL-7(2 & 3): 189–196, 1963.

Alexa T McCray. An upper-level ontology for the biomedical domain. *International Journal of Genomics*, 4(1):80–84, 2003.

Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

Drew McDermott. A critique of pure reason. *Computational intelligence*, 3(1): 151–160, 1987.

Arthur Mensch and Mathieu Blondel. Differentiable Dynamic Programming for Structured Prediction and Attention. *arXiv preprint arXiv:1802.03676*, 2018.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 3111–3119, 2013.

George A Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

Pasquale Minervini, Thomas Demeester, Tim Rocktäschel, and Sebastian Riedel. Adversarial Sets for Regularising Neural Link Predictors. *arXiv preprint arXiv:1707.07596*, 2017.

Pasquale Minervini*, Matko Bošnjak*, Tim Rocktäschel, and Sebastian Riedel. Towards Neural Theorem Proving at Scale. In *Neural Abstract Machines & Program Induction v2 (NAMPI_v2) @ ICML*, 2018.

Pasquale Minervini*, Matko Bošnjak*, Tim Rocktaschel, Sebastian Riedel, and Edward Grefenstette. Differentiable Reasoning on Large Knowledge Bases and Natural Language. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2020.

Marvin Minsky and Seymour A Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT press, 1969.

Mike Mintz, Steven Bills, Rion Snow, and Dan Jurafsky. Distant supervision for relation extraction without labeled data. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 1003–1011, 2009.

Matthew Mirman, Dimitar Dimitrov, Pavle Djordjevic, Timon Gehr, and Martin Vechev. Training Neural Machines with Trace-Based Supervision. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 3566–3574, 2018.

Jeff Mitchell, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Extrapolation in NLP. *arXiv preprint arXiv:1805.06648*, 2018.

Tom M Mitchell. Machine Learning, 1997.

R Mooney. Relational Learning of Pattern-Match Rules for Information Extraction. In *Proceedings of AAAI Conference on Artificial Intelligence*, volume 334, 1999.

Andrew W Moore. Very Fast EM-based Mixture Model Clustering using Multiresolution kd-trees. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 543–549, 1999.

Charles H Moore and Geoffrey C Leach. FORTH - A Language for Interactive Computing. *Amsterdam: Mohasco Industries Inc*, 1970.

Stephen Muggleton. Inductive Logic Programming. *New generation computing*, 8(4):295–318, 1991.

Stephen Muggleton. Inverse entailment and Progol. *New generation computing*, 13(3-4):245–286, 1995.

Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

Stephen Muggleton and Cao Feng. *Efficient induction of logic programs*. Citeseer, 1990.

Stephen H Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural Sketch Learning for Conditional Program Generation. *arXiv preprint arXiv:1703.05698*, 2018.

Arvind Neelakantan, Benjamin Roth, and Andrew McCallum. Compositional Vector Space Models for Knowledge Base Completion. *arXiv preprint arXiv:1504.06662*, 2015a.

Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding Gradient Noise Improves Learning for Very Deep Networks. *arXiv preprint arXiv:1511.06807*, 2015b.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural Programmer:

Inducing Latent Programs with Gradient Descent. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

John A Nelder and Roger Mead. A Simplex Method for Function Minimization. *The computer journal*, 7(4):308–313, 1965.

J Pedro Neto, H Siegelmann, and J Félix Costa. On the Implementation of Programming Languages with Neural Nets. In *First International Conference on Computing Anticipatory Systems*, volume 1, pages 201–208. CHAOS, 1998.

João Pedro Neto. A Virtual Machine for Neural Computers. In *Artificial Neural Networks – ICANN*, pages 525–534, 2006.

João Pedro Neto, J.Félix Costa, Ademar Ferreira, and Luciano Gualberto. Merging Sub-symbolic and Symbolic Computation. In *Proceedings of the International ICSC Symposium on Neural Computation (NC)*, 2000.

João Pedro Neto, Hava T Siegelmann, and J Félix Costa. Symbolic processing in neural networks. *Journal of the Brazilian Computer Society*, 8(3):58–70, 2003.

João Pedro Neto, José Félix Costa, Paulo Carreira, and Miguel Rosa. A Compiler and Simulator for Partial Recursive Functions over Neural Networks. In *Applications and Science in Soft Computing*, pages 39–46. Springer, 2004.

Isaac Newton and Derek Thomas Whiteside. The Mathematical Papers of Isaac Newton, volume 1 of The Mathematical Papers of Sir Isaac Newton, 2008.

M. Nickel, K. Murphy, V. Tresp, and E. Gabrilovich. A Review of Relational Machine Learning for Knowledge Graphs. *Proceedings of the IEEE*, 104(1): 11–33, Jan 2016a. ISSN 0018-9219. doi: 10.1109/JPROC.2015.2483592.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A Three-Way Model for Collective Learning on Multi-Relational Data. In *Proceedings of International Conference on Machine Learning (ICML)*, volume 11, pages 809–816, 2011.

Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A Review of Relational Machine Learning for Knowledge Graphs. *Proceedings of the IEEE*, 104(1):11–33, 2015.

Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. Holographic Embeddings of Knowledge Graphs. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2016b.

Peter Nordin. *Evolutionary program induction of binary machine code and its applications*. Krehl, 1997.

Chris Olah and Shan Carter. Attention and Augmented Recurrent Neural Networks. *Distill*, 2016. URL http://distill.pub/2016/augmented-rnns.

Christopher Olah. Understanding LSTM Networks, 2015.

Mateusz Ostaszewski, Edward Grant, and Marcello Benedetti. Quantum circuit structure learning. *arXiv preprint arXiv:1905.09692*, 2019.

Mark Palatucci, Dean Pomerleau, Geoffrey E Hinton, and Tom M Mitchell. Zero-shot Learning with Semantic Output Codes. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 1410–1418, 2009.

Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Regal: Transfer learning for fast optimization of computation graphs. *arXiv preprint arXiv:1905.02494*, 2019.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-Symbolic Program Synthesis. *arXiv preprint arXiv:1611.01855*, 2017.

Ronald Parr and Stuart J Russell. Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*, pages 1043–1049, 1998.

Barbara Partee. Lexical Semantics and Compositionality. *An invitation to cognitive science: Language*, 1:311–360, 1995.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 1310–1318, 2013.

Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508, 2017.

Jorge Pérez, Javier Marinković, and Pablo Barceló. On the Turing

Completeness of Modern Neural Network Architectures. *arXiv preprint arXiv:1901.03429*, 2019.

Jordan Bruce Pollack. On Connectionist Models of Natural Language Processing. *PhD dissertation. University of Illinois*, 1987.

Oleksandr Polozov and Sumit Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

J. Ross Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266, 1990.

Jack Rae, Jonathan J Hunt, Ivo Danihelka, Timothy Harley, Andrew W Senior, Gregory Wayne, Alex Graves, and Timothy Lillicrap. Scaling Memory-Augmented Neural Networks with Sparse Reads and Writes. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 3621–3629, 2016.

Mukund Raghothaman, Xujie Si, Kihong Heo, and Mayur Naik. Difflog: Learning Datalog Programs by Continuous Optimization. *arXiv*, 2019.

Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale Deep Unsupervised Learning using Graphics Processors. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 873–880, 2009.

Scott Reed and Nando De Freitas. Neural Programmer-Interpreters. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2016.

Sebastian Riedel, Limin Yao, and Andrew McCallum. Modeling Relations and Their Mentions without Labeled Text. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 148–163. Springer, 2010.

Sebastian Riedel, Limin Yao, Andrew McCallum, and Benjamin M Marlin. Relation Extraction with Matrix Factorization and Universal Schemas. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 74–84, 2013.

Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The annals of mathematical statistics*, pages 400–407, 1951.

John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

Tim Rocktaschel and Sebastian Riedel. End-to-End Differentiable Proving. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 3788–3800, 2017.

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Philippe Roussel. *Prolog Manual de Reference et d'Utilisation.* Groupe d'Intelligence Artificielle der Marseille Lumimy, 1975.

Subhro Roy and Dan Roth. Solving General Arithmetic Word Problems. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1743–1752, 2015.

Subhro Roy, Tim Vieira, and Dan Roth. Reasoning about Quantities in Natural Language. *Transactions of the Association for Computational Linguistics (TACL)*, 3:1–13, 2015.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Rudolph J Rummel. *The dimensionality of nations project: attributes of nations and behavior of nations dyads, 1950-1965.* Inter-university Consortium for Political Research, 1976. doi: 10.3886/ICPSR05409.v1.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, 3rd edition, 2009.

Ali Sadeghian, Mohammadreza Armandpour, Patrick Ding, and Daisy Zhe Wang. DRUM: End-To-End Differentiable Rule Mining On Knowledge Graphs. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, pages 15321–15331, 2019.

Claude Sammut and Ranan B Banerji. Learning concepts by asking questions. *Machine learning: An artificial intelligence approach*, 2:167–192, 1986.

Arthur L Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-Learning with Memory-Augmented Neural Networks. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 1842–1850, 2016.

Jürgen Schmidhuber. Optimal Ordered Problem Solver. *Machine Learning*, 54(3):211–254, 2004.

Jürgen Schmidhuber. Ultimate Cognition à la Gödel. *Cognitive Computation*, 1(2):177–193, 2009.

Jürgen Schmidhuber. Deep Learning in Neural Networks: An Overview. *Neural networks*, 61:85–117, 2015.

Stefan Schoenmackers, Oren Etzioni, Daniel S Weld, and Jesse Davis. Learning First-Order Horn Clauses from Web Text. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1088–1098, 2010.

Stephan Schulz. System description: E 1.8. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 735–743. Springer, 2013.

Mike Schuster and Kuldip K Paliwal. Bidirectional Recurrent Neural Networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group Oxford, 1971.

Daniel Selsam and Nikolaj Bjørner. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 336–353. Springer, 2019.

Luciano Serafini and Artur d'Avila Garcez. Logic Tensor Networks: Deep Learning and Logical Reasoning from Data and Knowledge. *arXiv preprint arXiv:1606.04422*, 2016.

Lokendra Shastri. Neurally motivated constraints on the working memory capacity of a production system for parallel processing: Implications of a

connectionist model based on temporal synchrony. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society: July*, volume 29, page 159, 1992.

Jude W Shavlik and Geoffrey G Towell. An approach to combining explanation-based and neural learning algorithms. In *Applications Of Learning And Planning Methods*, pages 71–98. World Scientific, 1991.

Yelong Shen, Jianshu Chen, Po-Sen Huang, Yuqing Guo, and Jianfeng Gao. M-Walk: Learning to Walk over Graphs using Monte Carlo Tree Search. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 6786–6797, 2018.

Yirong Shen, Matthias Seeger, and Andrew Y Ng. Fast Gaussian Process Regression using KD-Trees. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 1225–1232, 2006.

Hava T. Siegelmann. Neural Programming Language. In *Proceedings of AAAI Conference on Artificial Intelligence*, pages 877–882, 1994.

Hava T. Siegelmann. Computation Beyond the Turing Limit. *Science*, 268 (5210):545–548, 1995.

Hava T Siegelmann. On nil: The software constructor of neural networks. *Parallel Processing Letters*, 6(04):575–582, 1996.

Hava T. Siegelmann. *Neural Networks and Analog Computation: Beyond the Turing Limit*. Springer Science & Business Media, 2012.

Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77 – 80, 1991.

Hava T. Siegelmann and Eduardo D. Sontag. On the Computational Power of Neural Nets. *Journal of Computer and System Sciences*, 50(1):132 – 150, 1995.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering

the game of Go with deep neural networks and tree search. *nature*, 529 (7587):484, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

Paul Smolensky. On the proper treatment of connectionism. *Behavioral and brain sciences*, 11(1):1–23, 1988.

Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning With Neural Tensor Networks for Knowledge Base Completion. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 926–934, 2013.

Armando Solar-Lezama and Rastislav Bodik. *Program Synthesis by Sketching*. PhD thesis, U.C. Berkeley, 2008.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.

Ryan Spring and Anshumali Shrivastava. Scalable and Sustainable Deep Learning via Randomized Hashing. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 445–454. ACM, 2017.

Ashwin Srinivasan. A study of two probabilistic methods for searching large spaces with ilp. Technical Report PRG-TR-16-00, Oxford University Computing Laboratory (2000), 2000.

Ashwin Srinivasan. *The Aleph Manual*, 2001.

Bernd Steinbach and Roman Kohut. Neural Networks âĂŞ A Model of Boolean Functions. In *Boolean Problems, Proceedings of the 5th International Workshop on Boolean Problems*, pages 223–240, 2002.

Sainbayar Sukhbaatar, Jason Weston, and Rob Fergus. End-to-End Memory Networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 2440–2448, 2015.

Guo-Zheng Sun, C. Lee Giles, Hsing-Hen Chen, and Yee-Chun Lee. The Neural Network Pushdown Automaton: Model, Stack and Learning Simulations. *ArXiv*, abs/1711.05738, 1993.

GZ Sun. Neural networks with external memeory stack that learn context-free grammars from examples. In *Proceedings of the Conference on Information Science and Systems, 1991*, pages 649–653. Princeton University, 1991.

Patrick Suppes. *Introduction to logic*. Courier Corporation, 1999.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 3104–3112, 2014.

Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement Learning for Integer Programming: Learning to Cut. *arXiv preprint arXiv:1906.04859*, 2019.

Joshua B Tenenbaum, Charles Kemp, Thomas L Griffiths, and Noah D Goodman. How to Grow a Mind: Statistics, Structure, and Abstraction. *science*, 331(6022):1279–1285, 2011.

Robert D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, 1976.

Reuben Thomas. The Beetle Forth Virtual Machine, 2018. URL https://github.com/rrthomas/beetle/blob/master/doc/beetle.pdf.

Ilaria Tiddi, Freddy Lécué, and Pascal Hitzler. *Knowledge Graphs for eXplainable Artificial Intelligence: Foundations, Applications and Challenges*, volume 47 of *Studies on the Semantic Web*. IOS Press, 2020. ISBN 978-1-64368-080-4.

Kristina Toutanova, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon. Representing Text for Joint Embedding of Text and Knowledge Bases. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1499–1509, 2015.

Geoffrey G Towell and Jude W Shavlik. Knowledge-Based Artificial Neural Networks. *Artificial intelligence*, 70(1-2):119–165, 1994.

Geoffrey G Towell, Jude W Shavlik, and Michiel O Noordewier. Refinement

of Approximate Domain Theories by Knowledge-Based Neural Networks. In *Proceedings of AAAI Conference on Artificial Intelligence*, 1990.

Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. Neural Arithmetic Logic Units. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 8035–8044, 2018.

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex Embeddings for Simple Link Prediction. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 2071–2080, 2016.

Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. HOUDINI: Lifelong Learning as Program Synthesis. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 8687–8698, 2018.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An Introduction to Probabilistic Programming. *arXiv preprint arXiv:1809.10756*, 2018.

Allen Van Gelder. Efficient loop detection in Prolog using the tortoise-and-hare technique. *The Journal of Logic Programming*, 4(1):23–31, 1987.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 5998–6008, 2017.

Patrick Verga, Arvind Neelakantan, and Andrew McCallum. Generalizing to Unseen Entities and Entity Pairs with Row-less Universal Schema. *arXiv preprint arXiv:1606.05804*, 2016.

Ellen M Voorhees. Overview of the TREC 2001 question answering track. In *Proceedings of the 10th Text REtrieval Conference (TREC)*, pages 42–51, 2001.

Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise Selection for Theorem Proving by Deep Graph Embedding. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 2786–2796, 2017a.

Po-Wei Wang, Priya L Donti, Bryan Wilder, and Zico Kolter. SATNet: Bridg-

ing deep learning and logical reasoning using a differentiable satisfiability solver. *arXiv preprint arXiv:1905.12149*, 2019.

Quan Wang, Jing Liu, Yuanfei Luo, Bin Wang, and Chin-Yew Lin. Knowledge Base Completion via Coupled Path Ranking. In *Proceedings of Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1308–1318, 2016.

Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge Graph Embedding: A Survey of Approaches and Applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017b.

Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge Graph and Text Jointly Embedding. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1591–1601, 2014.

David HD Warren. An abstract Prolog instruction set. *Technical note 309*, 1983.

Leon Weber, Pasquale Minervini, Jannes Münchmeyer, Ulf Leser, and Tim Rocktäschel. NLProlog: Reasoning with Weak Unification for Question Answering in Natural Language. In *Proceedings of Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 6151–6161, 2019.

Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, volume 98, pages 194–205, 1998.

Gail Weiss, Yoav Goldberg, and Eran Yahav. On the Practical Computational Power of Finite Precision RNNs for Language Recognition. *arXiv preprint arXiv:1805.04908*, 2018.

Paul J Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

Jason Weston, Antoine Bordes, Oksana Yakhnenko, and Nicolas Usunier. Connecting Language and Knowledge Bases with Embedding Models for Relation Extraction. *arXiv preprint arXiv:1307.7973*, 2013.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory Networks. *arXiv preprint arXiv:1410.3916*, 2015.

Lyndon White, Roberto Togneri, Wei Liu, and Mohammed Bennamoun. How

Well Sentence Embeddings Capture Meaning. In *Proceedings of the 20th Australasian Document Computing Symposium*, page 9. ACM, 2015.

Terry Winograd. Understanding natural language. *Cognitive psychology*, 3(1): 1–191, 1972.

Niklaus Wirth. The programming language Pascal. *Acta informatica*, 1(1): 35–63, 1971.

Rongkai Xia, Yan Pan, Hanjiang Lai, Cong Liu, and Shuicheng Yan. Supervised Hashing for Image Retrieval via Image Representation Learning. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2014.

Wenhan Xiong, Thien Hoang, and William Yang Wang. DeepPath: A Reinforcement Learning Method for Knowledge Graph Reasoning. *arXiv preprint arXiv:1707.06690*, 2017.

Jiacheng Xu, Kan Chen, Xipeng Qiu, and Xuanjing Huang. Knowledge Graph Representation with Jointly Structural and Textual Encoding. *arXiv preprint arXiv:1611.08661*, 2016.

Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. *arXiv preprint arXiv:1412.6575*, 2014.

Fan Yang, Zhilin Yang, and William W Cohen. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, pages 2319–2328, 2017.

Yuan Yang and Le Song. Learn to Explain Efficiently via Neural Logic Inductive Learning. In *Proceedings of International Conference on Learning Representations (ICLR)*, 2020.

Peter N Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *Soda*, pages 311–21, 1993.

Fatin Zaklouta, Bogdan Stanciulescu, and Omar Hamdoun. Traffic sign classification using K-d trees and Random Forests. In *The 2011 International Joint Conference on Neural Networks*, pages 2151–2155. IEEE, 2011.

Wojciech Zaremba and Ilya Sutskever. Reinforcement Learning Neural Turing Machines - Revised. *arXiv preprint arXiv:1505.00521*, 2015.

Filip Železnỳ, Ashwin Srinivasan, and David Page. Lattice-Search Runtime Distributions May Be Heavy-Tailed. In *International Conference on Inductive Logic Programming*, pages 333–345. Springer, 2002.

Shuai Zhang, Yi Tay, Lina Yao, and Qi Liu. Quaternion Knowledge Graph Embeddings. In *Proceedings of Advances in Neural Information Processing Systems (NeurIPS)*, pages 2731–2741, 2019.

Huaping Zhong, Jianwen Zhang, Zhen Wang, Hai Wan, and Zheng Chen. Aligning Knowledge and Text Embeddings by Entity Descriptions. In *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 267–272, 2015.