Reinforcement Learning for Rule Selection in End to End Differentiable Proving

Candidate Number: 1047946

University of Oxford

A thesis submitted for the degree of
Master of Science in Computer Science

Trinity Term 2021

Abstract

Neural Theorem Provers (NTPs) are neuro-symbolic models that combine deep learning with a system of logic. They can learn representations for data, induce rules, are naturally interpretable, come with built-in explanations for conclusions, and demonstrate the capacity for systematic generalization. However, since they consider all possible rules for proving a goal, they suffer from high computational complexity, and are thus unsuitable for use on large or complex datasets. Conditional Theorem Provers (CTPs) are proposed as an extension to address this issue. Nonetheless, CTPs suffer from similar computational constraints, as they still consider multiple proof paths while reasoning. We propose RL-CTPs, where CTPs are augmented with reinforcement learning to select the proof paths most likely to succeed. This allows the model designer to specify the number of proof paths to consider, to conform to the computational constraints of their use case, while retaining all of the benefits of CTPs. Using the CLUTRR dataset to perform evaluations, we provide evidence for the computational issues in existing CTP models, show that RL-CTPs alleviate these issues, and demonstrate that, in certain scenarios, the accuracy achieved by RL-CTPs is higher than CTPs with equivalent computational complexity.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Knowledge Bases and Reasoning

Recent efforts in automated knowledge base (KB) completion use neural link prediction models to learn representations of symbols in a vector space, also referred to as subsymbolic representations (Nickel et al., 2012; Riedel et al., 2013; Socher et al., 2013; Chang et al., 2014; Yang et al., 2015; Toutanova et al., 2015; Trouillon et al., 2016). These representations allow models to infer new knowledge by encoding similarities. For example, if the subsymbolic representation of the predicate `grandfatherOf` is similar to that of `grandpaOf`, they likely express a similar relation and will hold for the same sets of constants. Likewise, if constants `Alice` and `Bob` have similar representations, then similar relations likely hold for both; for example, `livesInLondon`(`Alice`) and `livesInLondon`(`Bob`). This is useful, as many state-of-the-art knowledge bases have missing entries, which can be predicted using the learnt representations. For example, over 70% of people included in Freebase have no known place of birth (West et al., 2014).

While this is effective for automatically completing KBs, it is often more important to infer more complicated rules that hold in the data; for example, that the father of the father of X is the grandfather of X. However, this kind of reasoning is difficult for neural link prediction models to capture, as they only learn to score facts in isolation. In contrast, formal symbolic theorem provers like Prolog (Gallaire & Minker, 1978) are designed specifically to do this kind of multi-hop reasoning. However, symbolic provers do not learn subsymbolic representations, which means that they are unable to do link prediction and will often fail in situations where similar but not identical symbols are used (e.g. `grandpaOf` and `grandfatherOf`).

Promising work has been done around integrating neural models and symbolic reasoning, as their complementary strengths and weaknesses make for powerful models when combined (Garcez et al., 2015; Yang et al., 2017; Evans & Grefenstette, 2018; Sadeghian et al., 2019; Minervini et al., 2020a). In this thesis, we consider such a technique: Neural Theorem Provers (Rocktäschel & Riedel, 2017).

1.2 Neuro-symbolic Reasoning

The approach of Rocktäschel & Riedel (2017) is to keep variable binding symbolic, but compare predicates and constants using their subsymbolic representations. They introduce Neural Theorem Provers (NTPs): end-to-end differentiable provers for theorems formulated as queries to a KB. Prolog's backward chaining algorithm (Gallaire & Minker, 1978) is used as a blueprint for constructing neural networks in a recursive manner, which can prove queries to a KB using vector representations of symbols. These proofs are given success scores, which are differentiable with respect to the subsymbolic representations, allowing the model to learn representations that maximize the proof scores. Using the same process, rules of predefined structures are also learnt.

NTPs have many advantages. Primarily, they learn representations of symbols in a KB like neural link prediction models, but also learn rules which hold in the KB. It is trivial to incorporate already known rules into the reasoning of NTPs, as one simply needs to include them in the knowledge base. NTPs are also naturally interpretable, since they induce subsymbolic rules that can be decoded to human-readable symbolic rules. This makes them preferable to black box natural language models such as BERT (Devlin et al., 2019), which cannot give explanations for their answers. Finally, Minervini et al. (2020b) demonstrate that NTPs have the ability to perform systematic generalization, learning how to evaluate using complex reasoning patterns while only being trained on simple examples. In contrast, many natural language understanding systems, where a deep neural network is used for question answering and inference, appear not to generalize robustly (Johnson et al., 2017; Bahdanau et al., 2019; Lake & Baroni, 2018; Sinha et al., 2019).

However, NTPs suffer from significant computational issues, as they consider all possible rules for proving a goal or sub-goal. This means they cannot be applied in situations that require a large number of rules or reasoning steps. Minervini et al. (2020b) introduce Conditional Theorem Provers (CTPs) as a solution to this: an extension of NTPs that learns to select subsets of rules to consider at each expansion step of the reasoning algorithm. CTPs do this by including a module that learns how to take a goal and return only the rules that could be used to prove it. It consists of multiple neural networks, referred to as *reformulators*, each of which can represent multiple rules in a knowledge base. This new module is end-to-end differentiable in the same way that NTPs are and thus can be trained jointly with the other processes in NTPs.

Despite being introduced as a solution to the problem, CTPs can end up suffering from computational issues in a similar fashion to NTPs, since they still need to consider multiple proof paths when reasoning. For complex datasets in which there are many ways to prove a given goal, more proof paths need to be checked. This, in conjunction with the high reasoning depth often required for such datasets, causes CTPs to become infeasibly slow. This is particularly problematic when a CTP model applied in a scenario where the evaluation time is important, such as answering a question posed by a user who needs a prompt solution. Even outside of this, the computational time complexity of CTPs is high enough that one could envision datasets on which they will take too long to evaluate for any use case.

1.3 Objectives

In this thesis, we aim to address the computational issues that CTPs suffer from by extending them to RL-CTPs. In this new model, CTPs are augmented with reinforcement learning, where an agent is trained to select the proof paths that are most likely to succeed. In this way, the number of proof paths considered during evaluation can be scaled down, at the discretion of the model designer. This allows the designer of the model to bring down the evaluation time to suit the use case and enables CTPs to evaluate on datasets that they otherwise would not be able to, due to their computational time complexity.

An alternative to addressing the computational issues with a CTP model is to simply reduce the number of reformulators trained, leading to fewer proof paths being considered. However, this makes the model less expressive, meaning it will likely be unable to capture all of the rules in a knowledge base. Thus, for RL-CTPs to be useful, an RL-CTP model expanding only k proof paths should achieve higher accuracy than a CTP model with only k reformulators.

With that in mind, the objectives of this thesis are as follows:

1. Motivate for CTP models sometimes requiring a large number of reformulators and reasoning depth.

2. Concretely establish the computational issues that CTPs suffer from, using both empirical results and a theoretical analysis.
3. Define a framework for RL-CTPs using policy gradient descent.
4. Develop and discuss various implementations of RL-CTPs.
5. Demonstrate that RL-CTPs are an improvement upon CTPs by running experiments, displaying and analyzing the results. We aim to test both their respective accuracies and evaluation times.

1.4 Thesis Structure

This thesis consists of 7 chapters. Subsequent to this introductory chapter, Chapter 2 provides all the background knowledge needed to understand CTPs, as well as placing them in their context by reviewing related work. Chapter 3 motivates for using more reformulators and higher reasoning depths in CTP models and analyzes in detail the computational issues that CTPs suffer from as a result. In Chapter 4, we introduce our framework for RL-CTPs and discuss the design of our experiments.

Chapters 5 and 6 report on the main body of implementation work that we performed, as well as showing and analyzing the results for each of the methods we attempted. In Chapter 5, we discuss the methods that failed to yield satisfactory results, as well as giving some insights into the inner workings of CTPs. In Chapter 6, we give a detailed explanation of our successful implementation, analyze the results, and discuss further improvements that could be made to the model. We conclude the thesis in Chapter 7 by summarizing the project outcomes, impact, and potential directions of future work. Appendices are also included, specifying the model hyperparameters used for evaluations.

Chapter 2

Background

In this chapter, we introduce all of the background knowledge needed for the purposes of this thesis. We start by formally defining Prolog (Gallaire & Minker, 1978), a logic programming language. Next, we introduce the main dataset of concern in this work, CLUTRR (Sinha et al., 2019), so that all future examples and discussions may be grounded in this dataset. The backward chaining algorithm is then introduced, before following on to the continuous relaxation of it into the technique of Neural Theorem Provers (Rocktäschel & Riedel, 2017). Finally, Conditional Theorem Provers (Minervini et al., 2020b) are introduced, as the technique upon which this work is based. We conclude the chapter by reviewing related work that uses similar techniques, so that this thesis may be understood in its context.

2.1 Prolog

Prolog (Gallaire & Minker, 1978) is a logic programming language that finds use in contemporary work, such as by Seipel et al. (2018); Calegari et al. (2020). Moreover, its core syntax and semantics are also found in other logic languages, such as Datalog (Ceri et al., 1989). It has been used for a variety of tasks, including automated theorem proving (Stickel, 1988), expert systems (Merritt, 2012), and natural language processing (Lally & Fodor, 2011). A Prolog knowledge base (KB) consists of *rules* and *facts*. Queries are passed to the KB, with Prolog returning whether or not the queries are entailed (proven to be true) by the KB.

For ease of reading and consistency, we formally define the syntax and semantics of Prolog to be used for the remainder of this thesis. An *atom* consists of a predicate and a list of terms. Each term in an atom can be a *variable*, denoted by a capital letter such as X, or a *constant*, denoted by a capitalized word such as `Isaac`. We denote predicates with lowercase words such as `fatherOf`. So for example, the atom

$$\texttt{fatherOf}(\texttt{Isaac}, \texttt{Jacob})$$

states that Isaac is the father of Jacob. One can also construct terms with variables, such as $\texttt{fatherOf}(X, \texttt{Jacob})$.

A Prolog rule has the form $H \leftarrow B$, where the body B is a conjunction of zero or more atoms and the head H is an atom. The variables in any rule are universally quantified. If a rule has no variables, it is said to be *grounded*. A grounded rule with an empty body constitutes a fact and is simply written out as the singular head atom. By example:

- $\texttt{fatherOf}(\texttt{Isaac}, \texttt{Jacob})$ is a fact.
- $\texttt{parentOf}(\texttt{Isaac}, \texttt{Jacob}) \leftarrow \texttt{sonOf}(\texttt{Jacob}, \texttt{Isaac})$ is a ground rule.
- $\texttt{grandfatherOf}(X, Z) \leftarrow \texttt{fatherOf}(X, Y) \wedge \texttt{fatherOf}(Y, Z)$ is a rule stating that for all X, Y, Z, if X is the father of Y and Y the father of Z, then X is the grandfather of Z.

A *substitution set* $\psi = \{X_1/t_1, ..., X_N/t_N\}$ defines a replacement of each variable X_i by a term t_i. Applying a substitution ψ to an atom or rule replaces all occurrences of variables X_i by their corresponding terms t_i.

So, given a knowledge base consisting of the rule

$$\texttt{grandfatherOf}(X, Z) \leftarrow \texttt{fatherOf}(X, Y) \wedge \texttt{fatherOf}(Y, Z)$$

and the facts $\{\texttt{fatherOf}(\texttt{Isaac}, \texttt{Jacob}), \texttt{fatherOf}(\texttt{Jacob}, \texttt{Joseph})\}$, we can apply the substitution set $\{X/\texttt{Isaac}, Y/\texttt{Jacob}, Z/\texttt{Joseph}\}$ to the rule to get that

$$\begin{aligned}\texttt{grandfatherOf}(\texttt{Isaac}, \texttt{Joseph}) \leftarrow \texttt{fatherOf}(\texttt{Isaac}, \texttt{Jacob}) \\ \wedge\, \texttt{fatherOf}(\texttt{Jacob}, \texttt{Joseph})\end{aligned}$$

and thus conclude that $\texttt{grandfatherOf}(\texttt{Isaac}, \texttt{Joseph})$ is true, since both atoms in the body of the ground rule appear as facts in the KB.

2.2 Datasets

2.2.1 Datasets from Related Work

In the original paper proposing Neural Theorem Provers, Rocktäschel & Riedel (2017) use the datasets of **Countries** (Bouchard et al., 2015), **Kinship**, **Nations**, and **UMLS** (Kemp et al., 2006) to evaluate their model in terms of its performance on neural link prediction tasks. **Countries** contains facts about regions, countries, and sub-regions, and is designed to test the ability of the model to learn the hierarchical structure of the regions, with respect to which of them are located within others. **Nations** specifies properties of nation states, **Kinship** contains specific kinship relations, and **UMLS** captures biomedical concepts, including treatments and diagnoses.

The wide array of datasets used demonstrates the applicability of this technique to many different fields, with both Neural Theorem Provers and Conditional Theorem Provers outperforming state-of-the-art neural link prediction models in most of these datasets. In the paper proposing Conditional Theorem Provers, Minervini et al. (2020b) introduce another dataset for evaluation, CLUTRR, which is the dataset we use for evaluation in this thesis.

2.2.2 CLUTRR

CLUTRR - Compositional Language Understanding and Text-based Relational Reasoning (Sinha et al., 2019) - is a system for constructing artificial datasets modelling family relationships. It contains a large number of parameters for controlling generation, including the number of examples to generate, the complexity of the examples, and the number of facts in each example. The complexity ranges from basic family relationships that are free from noise, up to family relationships with disconnected and irrelevant facts.

Given a set of family relations, the task is to infer the relationship between two family members whose relationship is not explicit in the set. To solve this, an agent ought to be able to induce the logical rules that govern family relationships, and use those rules to infer the relationship of the query members from the given relations. In particular, CLUTRR allows for testing an agent's ability to perform *systematic generalization.*

Systematic generalization is defined as the capacity to understand and produce a potentially infinite number of novel combinations from known components (Chomsky, 1957). In the case of CLUTRR, this means testing on knowledge bases that require combinations of rule applications that were not provided during training, as well as testing on datasets that require a higher number of reasoning steps than during training (Sinha et al., 2019; Gontier et al., 2020). If the model has the ability to perform systematic generalization, then it should be able to apply learnt rules in new combinations to solve the problem, as well as reasoning on larger and more complex examples than seen before. This is evaluated by training the model to infer relationships while traversing only a small number of family relations, and then evaluating on a dataset where it has to traverse a larger number.

2.2.3 Specific CLUTRR Instances

Sinha et al. (2019) published several generated dataset groups alongside the CLUTRR system. Minervini et al. (2020b) make use of two of them, under the original identifiers `089907f8` and `db9b8f04`, which they refer to as $\text{CLUTRR}_{\text{G}}(k = 2, 3)$ and $\text{CLUTRR}_{\text{G}}(k = 2, 3, 4)$ respectively. The test and train sets of both are generated as "clean" stories, designed to test generalization. $\text{CLUTRR}_{\text{G}}(k = 2, 3, 4)$ contains a training dataset with training clauses of length $k = 2, 3, 4$ and nine testing datasets, each with a different testing clause length $k = 2, 3, ..., 10$. This is the dataset group used for training and evaluation in our thesis. We refer to the training dataset as *1.2,1.3,1.4_train* and the testing datasets as *1.2_test*, *1.3_test*, ..., *1.10_test*.

It is also important to note that Minervini et al. (2020b) do some pre-processing on the datasets to turn all gendered predicates into their corresponding "ungendered" versions. For example, the fact `brother`(`Benjamin`, `Joseph`) is converted to `sibling`(`Benjamin`, `Joseph`). In this thesis, we make use of the unedited version of $\text{CLUTRR}_{\text{G}}(k = 2, 3, 4)$, as it is more complex than the ungendered version and directly corresponds to the dataset published by Sinha et al. (2019). This means that instead of learning rules such as

$$\texttt{grand}(X, Z) \leftarrow \texttt{SO}(Y, Z) \wedge \texttt{grand}(X, Y)$$

as demonstrated by Minervini et al. (2020b), we learn rules such as

$$\texttt{grandmother}(X, Z) \leftarrow \texttt{wife}(Y, Z) \wedge \texttt{grandfather}(X, Y)$$

The set of all gendered predicates that can appear in CLUTRR is: $P = \{$ `aunt`, `brother`, `brother-in-law`, `daughter`, `daughter-in-law`, `father`, `father-in-law`, `granddaughter`, `grandfather`, `grandmother`, `grandson`, `husband`, `mother`, `mother-in-law`, `nephew`, `niece`, `sister`, `sister-in-law`, `son`, `son-in-law`, `uncle`, `wife` $\}$.

Story	[Joshua] got his son, [Don], a car for his birthday. [Don] loves talking to his grandfather [James] on the phone. [James] took his daughter, [Cindy], to a baseball game.
Query	(‘Joshua’, ‘Cindy’)
Target	sister
Proof State	[(‘Joshua’, ‘sister’, ‘Cindy’): [(‘Joshua’, ‘father’, ‘James’), (‘James’, ‘daughter’, ‘Cindy’)], (‘Joshua’, ‘father’, ‘James’): [(‘Joshua’, ‘son’, ‘Don’), (‘Don’, ‘grandfather’, ‘James’)]]

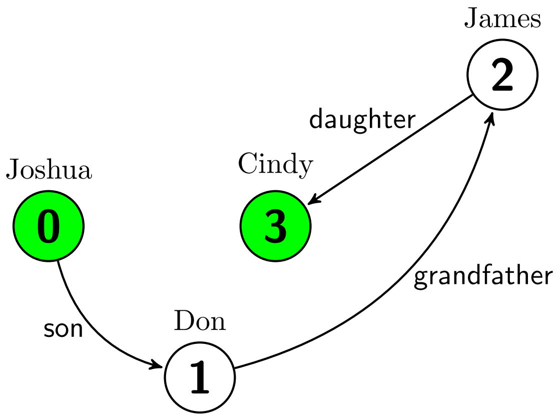
Story Edges	[(0, 1), (1, 2), (2, 3)]
Edge Types	[‘son’, ‘grandfather’, ‘daughter’]
Query Edge	(0, 3)

Figure 2.1: Example task from CLUTRR dataset *1.3_test*

Every task within a $\text{CLUTRR}_\text{G}(k = 2, 3, 4)$ dataset contains a story in textual form, a query specifying two individuals to infer the relationship between, the answer to the query, a “proof state” that demonstrates how the answer to the query could be proved, instructions for how to formally represent the story as a knowledge graph, and other less relevant info such as the gender of each individual. An extract from *1.3_test* can be seen in Figure 2.1, and the knowledge graph it represents in Figure 2.2.

2.3 Backward Chaining Algorithm

The restrictive syntax of Prolog, especially when compared to other logics such as first-order logic (Smullyan, 1995), allows one to answer queries using Prolog’s *backward chaining algorithm* (Gallaire & Minker, 1978). Given a goal, such as $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$, which is constructed from a query, Prolog tries to find substitutions for the goal by using the rules in the knowledge base. The process of checking if the head of a rule matches a goal is called *unification*. If unification succeeds, then the goal is replaced with the atoms from the body of the rule, giving a new set of sub-goals. The same process is then applied to each of these sub-goals, continuing recursively. If at any point, the set of sub-goals all exist as facts in the knowledge base, then the algorithm confirms that the query is true. If there are no more rules to apply, taking care to detect and cease applying cycles of rules, the algorithm states that the query cannot be proven from the knowledge base.



An arrow represents a possessive relationship. For example, the arrow from `Joshua` to `Don` shows that Joshua has a son called Don. The green nodes represent the query entities.

Figure 2.2: Graph representing the task from figure 2.1

Using the example from Figure 2.2, consider the following goal to be proven: $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$. We make the example semantically consistent by assuming that the atom $\texttt{grandfather}(X, Z)$ means that X has a *paternal* grandfather Z. This is necessary, since CLUTRR makes simplifying assumptions about family relationships in the datasets it generates. Assume that the rules in the knowledge base are:

$$\begin{aligned} (1)\ \ & \texttt{father}(Y, Z) \leftarrow \texttt{son}(Y, X) \wedge \texttt{grandfather}(X, Z) \\ (2)\ \ & \texttt{sister}(X, Y) \leftarrow \texttt{father}(X, Z) \wedge \texttt{daughter}(Z, Y) \end{aligned}$$

This gives an initial set of sub-goals: $\{\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})\}$. Prolog then tries to unify this sub-goal with the rules in the knowledge base. Unification with rule (1) fails, since $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$ does not unify with the head of the rule $\texttt{father}(X, Y)$ due to mismatching predicates. Unification with $\texttt{sister}(X, Y)$ succeeds under the substitution set $\{X/\texttt{Joshua}, Y/\texttt{Cindy}\}$, so the set of sub-goals is updated using rule (2) to $\{\texttt{father}(\texttt{Joshua}, Z), \texttt{daughter}(Z, \texttt{Cindy})\}$.

Now consider the sub-goal $\texttt{father}(\texttt{Joshua}, Z)$, which unifies with the head of rule (1) under the substitution set $\{Y/\texttt{Joshua}, Z/A\}$ (substituting Z as it has already been used). This gives two new sub-goals, updating our set of sub-goals to $\{\texttt{daughter}(Z, \texttt{Cindy}), \texttt{son}(\texttt{Joshua}, X), \texttt{grandfather}(X, A)\}$. But under the substitution set $\{Z/\texttt{James}, A/\texttt{James}, X/\texttt{Don}\}$, the ground atoms

$$\{\texttt{daughter}(\texttt{James}, \texttt{Cindy}), \texttt{son}(\texttt{Joshua}, \texttt{Don}), \texttt{grandfather}(\texttt{Don}, \texttt{James})\}$$

all appear as facts in the knowledge base. Thus, the algorithm confirms that the goal $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$ is true, giving an answer of 'sister' to the query ('Joshua', 'Cindy').

Note that the algorithm will often branch out into multiple potential proofs, since there can be several rules whose heads unify with a sub-goal. For example, to prove the goal $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$, one could use either of the following rules to derive sub-goals:

$$\texttt{sister}(X, Y) \leftarrow \texttt{father}(X, Z) \land \texttt{daughter}(Z, Y)$$
$$\texttt{sister}(X, Y) \leftarrow \texttt{sister}(X, Z) \land \texttt{sister}(Z, Y)$$

In Algorithm 1, based on the pseudocode provided by Rocktäschel & Riedel (2017), we define a recursive algorithm for performing backward chaining. The *or* function considers all the possible rules whose heads can be unified with a given goal, as any of them would suffice for a proof. The *and* function captures having to prove every atom in the body of a rule to prove the head true. As this method expands out backwards from the goal, considering all possible proof paths, it will find a valid proof if one does exist.

However, Prolog, and thus the backward chaining algorithm, do suffer from certain limitations. In particular, they are unable to learn subsymbolic representations for symbols in a knowledge base, and thus cannot generalize to queries containing similar predicates or constants that are represented by different symbols (for example, `grandpa` and `grandfather`). This is because even though the predicates represent the same concept, they are symbolically different, meaning that unification between them will fail.

2.4 Neural Theorem Provers

Neural Theorem Provers (NTPs), proposed by Rocktäschel & Riedel (2017), are a continuous relaxation of the backward chaining algorithm. They allow one to calculate the gradient of proof successes with respect to vector representations of symbols, and are defined in terms of modules, drawing inspiration from dynamic neural module

Algorithm 1: Backward chaining

In the code, K is the knowledge base containing the rules and facts, sets are denoted with curly brackets, lists are denoted with square brackets, an underscore matches any argument, G refers to a goal, $\hat{G}$ to a set of sub-goals, S to a substitution set, B to the body of a rule, and H to the head of a rule. To check if a goal G_1 holds true, one needs to get the output of $\mathrm{or}(G_1, [])$. If the output contains a substitution set, then the query is true, otherwise it will only contain the value `FAIL` and the query cannot be proven.

1. $\mathrm{or}(G, S) = \{S' \mid S' \in \mathrm{and}(B, \mathrm{unify}(H, G, S)) \text{ for each } H \leftarrow B \in K\}$
2. $\mathrm{and}(\_, \mathtt{FAIL}) = \mathtt{FAIL}$
3. $\mathrm{and}([], S) = S$
4. $\mathrm{and}(G : \hat{G}, S) = \{S'' \mid S'' \in \mathrm{and}(\hat{G}, S'')\ \forall S' \in \mathrm{or}(\mathrm{substitute}(G, S), S)\}$
5. $\mathrm{unify}(\_, \_, \mathtt{FAIL}) = \mathtt{FAIL}$
6. $\mathrm{unify}([], [], S) = S$
7. $\mathrm{unify}([], \_, \_) = \mathtt{FAIL}$
8. $\mathrm{unify}(\_, [], \_) = \mathtt{FAIL}$
9. $$\mathrm{unify}(h : H, g : G, S) = \mathrm{unify}\left(H, G \left\{ \begin{array}{cc} S \cup \{h/g\} & \text{if } h \in V \\ S \cup \{g/h\} & \text{if } g \in V, h \notin V \\ S & \text{if } g = h \\ \mathtt{FAIL} & \text{otherwise} \end{array} \right\}\right)$$
10. $\mathrm{substitute}([], \_) = []$
11. $$\mathrm{substitute}(g : G, S) = \left\{ \begin{array}{cc} x & \text{if } g/x \in S \\ g & \text{otherwise} \end{array} \right\}$$

networks (Andreas et al., 2016). These modules are based upon the functions from the recursive definition of the backward chaining algorithm in Algorithm 1.

The recursive expansion upon the goal is kept track of in *proof states*, which each contain a neural network that outputs the success score of the proof so far, and the substitution set. The neural network is recursively built upon, with new nodes being added as rules are applied. Every different proof path will have a different proof state associated with it. The modules and training procedures are described in detail below.

2.4.1 Unification

The conventional backward chaining algorithm does unification symbolically; for example, checking if the symbol `sister` in a goal $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$ matches the symbol `sister` in a rule $\texttt{sister}(X, Y) \leftarrow \texttt{father}(X, Z) \wedge \texttt{daughter}(Z, Y)$. NTPs replace this by instead having a dense vector representation for each predicate and constant in the knowledge base. So the predicates `sister`, `father` and the constants `Joshua`, `Cindy` would become their representations $\theta_{\texttt{sister}}, \theta_{\texttt{father}}, \theta_{\texttt{Joshua}}, \theta_{\texttt{Cindy}} \in \mathbb{R}^k$. These representations are randomized initially and trained over time.

Using these representations, NTPs compare symbols for unification using a soft matching between their vector representations. Any differentiable similarity measure $K : \mathbb{R}^k \times \mathbb{R}^k \to [0, 1]$, such as a Gaussian kernel, can be used for this purpose. Rocktäschel & Riedel (2017) choose to utilize a Radial Basis Function (RBF) kernel (Broomhead & Lowe, 1988), with $\mu = \frac{1}{\sqrt{2}}$. This means that rules can be applied even when the symbols in the goal and the head are not exactly the same, but are similar in meaning (for example, `grandpa` and `grandfather`).

To unify two atoms, the `unify` module iterates through the list of terms of each atom, with unification failing if the predicates have different arities (number of arguments). If one of the symbols is a variable, a substitution is added to the current substitution set, with pairs of constants being assigned a similarity score using the described RBF kernel. This means that unification can only fail when there are mismatching arities in the predicates, as even vastly different constants will simply mean a very low similarity score. If unification does fail, then the neural network expansion is cancelled for this particular proof branch.

2.4.2 Expansion

The `or` module attempts to apply rules in the KB to a goal. The module simply unifies the goal with the head of every rule in the KB, and then instantiates an `and` module to prove the sub-goals of every rule for which unification succeeded. The similarity score from unification gives an upper bound on the scores of the proofs expanded further from this point, capturing the notion that a proof is only as accurate as the least accurate expansion performed in it.

The `and` module tries to prove every sub-goal by instantiating an `or` module for each of them, but stops expanding if the pre-defined reasoning depth limit has already been reached. The overall success score $\texttt{ntp}_\theta^\mathcal{K}$ of proving a goal G using a KB $\mathcal{K}$, representations θ, and a reasoning depth of d, is the maximum score of any proof state originating from the goal up to depth d. A full overview of the neural backward chaining algorithm used by NTPs can be found in Algorithm 2.

2.4.3 Inductive Logic Programming

Inductive Logic Programming (ILP) tries to learn logical rules from a knowledge base and use them for reasoning (Muggleton, 1991). NTPs can achieve this by using gradient descent, instead of older methods such as a combinatorial search over the space of all possible rules (Quinlan, 1990). They apply the concept of *learning from entailment* by learning rules that maximize the proof scores for known ground atoms and minimize the proof scores for random samples of unknown ground atoms. NTPs introduce *parameterized rules* to achieve this, which are rules of a predefined structure but unknown predicates; for example, $r(X, Y) \leftarrow s(X, Z) \wedge t(Z, Y)$. During training, the representations of these parameterized predicates are optimized alongside all the other representations. This allows the model to learn optimal representations of the predicates, which often correspond to rules that hold in the knowledge base.

Rule templates are used to define the structure of multiple rules by specifying rule structures, as well as the number of parameterized rules that should be created for a given structure. For example, for the Kinship, Nations, and UMLS datasets, the following parameterized rules were used:

- 20 rules of form $r(X, Y) \leftarrow s(X, Y)$
- 20 rules of form $r(X, Y) \leftarrow s(Y, X)$
- 20 rules of form $r(X, Y) \leftarrow s(X, Z) \wedge t(Z, Y)$

Algorithm 2: Neural backward chaining

The code is based on the summary by Minervini et al. (2020b). G is a goal, d is the reasoning depth, H is the head of a rule, B is the body, $\mathcal{K}$ is a knowledge base containing rules and facts, K is the RBF kernel, S is a proof state, S_ψ is a substitution set, S_ρ is a proof score, and V is a set of variables.

```
def or($G, d, S$)
    for $H \leftarrow B \in \mathcal{K}$ do                    /* Try use any rule in KB to prove */
        for $S \in and(B, d, unify(H, G, S))$ do
            yield $S$
        end
    end
end

def and($B, d, S$)
    if $B = []$ or $d = 0$ then                  /* Empty body or reasoning depth reached */
        yield $S$
    else
        for $S' \in or(sub(B_0, S_\psi), d-1, S)$ do  /* Apply substitution to body, then try to prove each atom within */
            for $S'' \in and(B_{1:}, d, S')$ do
                yield $S''$
            end
        end
    end
end

def unify($H, G, S = (S_\psi, S_\rho)$)
    $T_i = \left\{ \begin{array}{cc} \{H_i/G_i\} & \text{if } H_i \in V \\ \{G_i/H_i\} & \text{if } G_i \in V, H_i \notin V \\ \emptyset & \text{otherwise} \end{array} \right\}$
    $S'_\psi = S_\psi \bigcup T_i$                          /* Extend the substitution set */

    $S'_\rho = \min\{S_\rho\} \bigcup_{H_i, G_i \notin V} \{K(\theta_{H_i}, \theta_{G_i})\}\}$   /* Similarity value is the minimum similarity across all representations */

    return $S' = (S'_\psi, S'_\rho)$
end
```

After training, a parameterized rule can be decoded by finding the closest predicate representation to each predicate in the rule.

2.4.4 Learning

Since the neural network in a proof state is constructed using only the operations of *min*, *max*, and the RBF kernel, the final score of the proof (the output of the neural network) is differentiable with respect to the embeddings of the predicates and constants. One issue is that known facts that are being proved can be trivially unified with themselves in the knowledge base, giving no updates during training. This is resolved by removing the current fact being proved from the knowledge base before training, then adding it back again afterwards.

Since in general, and particularly for the datasets used by Rocktäschel & Riedel (2017), negative facts are not provided, sampled corrupted ground atoms are used instead. This is an approach that has found success in previous work, such as that by Bordes et al. (2013). For every fact $s(i, j) \in \mathcal{K}$, corrupted atoms $s(\hat{i}, j), s(i, \hat{j}), s(\hat{k}, \hat{l}) \notin \mathcal{K}$ are found by sampling $\hat{i}, \hat{j}, \hat{k}, \hat{l}$ from the set of constants. The corrupted atoms are resampled in every training iteration.

Using the negative log-likelihood of the proof success score as a loss function, NTPs are trained with target scores of 1 for known ground atoms and 0 for the corrupted atoms. Letting T denote the set of known and corrupted atoms, $s(i, j)$ a training ground atom, y its target proof success score, θ the predicate and constant embeddings, d the reasoning depth, and $\mathcal{K}$ the knowledge base; the loss function is:

$$L_{\mathtt{ntp}_\theta^{\mathcal{K}}} = \sum_{(s(i,j),y)\in T} -y \log(\mathtt{ntp}_\theta^{\mathcal{K}}(s(i,j), d) - (1-y)\log(1 - \mathtt{ntp}_\theta^{\mathcal{K}}(s(i,j), d)$$

2.5 Conditional Theorem Provers

2.5.1 Issues with NTPs

As noted by Rocktäschel & Riedel (2017), NTPs suffer from severe computational limitations. In standard backward chaining, a proof path can be aborted when unification with a rule fails, but this happens far less in neural backward chaining, since unification only fails when predicates do not have matching arities. Thus, almost every rule in the KB needs to be considered when proving a goal or sub-goal, rather than certain rules being used for a particular goal. When one considers the rule templates used by Rocktäschel & Riedel (2017) for their 4 datasets, the problem becomes even more noticeable, as all of them have the same predicate arity in the head of the rule: two. This means that every rule has to be considered to prove every goal and sub-goal, which is 60 rules for each of the Kinship, Nations, and UMLS datasets.

They try to solve these issues by proposing two optimizations to NTPs. They use batch processing (Abdelfattah et al., 2016; Agarwal, 2019) to process many proofs in parallel on the GPU, updating the unification module accordingly. They also exploit the sparse gradients created by the *min* and *max* operations in the algorithm to perform truncated forward and backward passes, thus lowering the number of proofs needed for calculating gradients. However, as Minervini et al. (2020a,b); Bošnjak (2021) note, NTPs still do not scale to large datasets.

2.5.2 Conditional Rule Selection

Minervini et al. (2020b) introduce Conditional Theorem Provers (CTPs) as a solution to this problem, proposing that the rules used to prove a goal should be conditioned upon the goal. For example, given a goal such as $\texttt{sister}(\texttt{Joshua}, \texttt{Cindy})$, the prover should only consider rules such as $\texttt{sister}(X, Y) \leftarrow \texttt{father}(X, Z) \wedge \texttt{daughter}(Z, Y)$ to prove the goal, and not rules such as $\texttt{father}(Y, Z) \leftarrow \texttt{son}(Y, X) \wedge \texttt{grandfather}(X, Z)$. Since in NTPs, predicates and constants are represented by their embedding vectors, the useful above rule can be represented by a mapping $\theta_{\texttt{sister:}} \longmapsto [\theta_{\texttt{father:}}, \theta_{\texttt{daughter:}}]$.

Thus, Minervini et al. (2020b) propose introducing a new module into the system, `select`, to reduce the number of rules being considered when expanding upon a goal. In the `or` module, instead of considering every $H \leftarrow B \in \mathcal{K}$, they only consider each $H \leftarrow B \in \texttt{select}_\theta(G)$. See Algorithm 3 for the adapted `or` module. The selection module can be implemented by a sequence of differentiable parameterized functions $\texttt{select}_\theta^1(G), \texttt{select}_\theta^2(G), ..., \texttt{select}_\theta^n(G)$ that each, given a goal, produces a set of sub-goals. We refer to each of these of these functions as a *reformulator*.

2.5.3 Selection Module

The selection module is thus a function $\texttt{select}_\theta : A \to [A \leftarrow A^*]$, with $\texttt{select}_\theta(G) = [\texttt{select}_\theta^1(G), ..., \texttt{select}_\theta^n(G)]$. In the above, V is a set of variables, $A \in \mathbb{R}^k \times (\mathbb{R}^k \cup V) \times (\mathbb{R}^k \cup V)$ the embedding of an atom such as $\texttt{sister}(X, \texttt{Cindy})$, A^* means any number of atoms, and $A \leftarrow A^*$ thus represents a rule. Note that this means that we have

Algorithm 3: Conditional theorem proving: inclusion of selection module

```
The rules considered for expansion are conditioned upon the goal $G$.

def $or(G, d, S)$
    for $H \leftarrow B \in select_\theta(G)$ do
        for $S \in and(B, d, unify(H, G, S))$ do
            yield $S$
        end
    end
end
```

limited the system to use only binary predicates, which is fine for CLUTRR datasets, as all of the relationships expressed in the knowledge base are binary. However, the theory presented in CTPs can easily be extended to work with predicates of a higher or lower arity by hard coding it into the structure of the reformulators. For the purposes of this thesis, we will only consider CTPs operating on binary predicates.

This means that each reformulator $\texttt{select}_\theta^i$ is a function $\texttt{select}_\theta^i : A \rightarrow A \times A \times ... \times A$, where the number of A's in the Cartesian product within the co-domain is equivalent to the number of atoms in the body of the rules it represents. For example, a reformulator $\texttt{select}_\theta^1 : A \rightarrow A \times A$ could capture the rule $\texttt{sister}(X, Y) \leftarrow \texttt{father}(X, Z) \wedge \texttt{daughter}(Z, Y)$ by mapping the goal $G = [\theta_{\texttt{sister}}, X, Y]$ to the subgoals $[[\theta_{\texttt{father}}, X, Z], [\theta_{\texttt{daughter}}, Z, Y]]$. It could also capture other rules, as long as the head of each rule is distinct. Provided that the positions of the variables in the corresponding rule structure are fixed, then each reformulator, and hence the entire `select` module, is end-to-end differentiable with respect to the model parameters θ.

CTPs are trained in a similar manner to NTPs, with positive examples being assigned a target proof score of 1 and negative examples 0. Negative examples are generated from each CLUTRR task, say with a target of $p(c_1, c_2)$, by including $p'(c_1, c_2)$ with a target proof score of 0 for every $p \neq p' \in P$, where P is the set of all predicates available in CLUTRR. To get the answer to a query (c_1, c_2) during evaluation, the CTP model computes the score of $p(c_1, c_2)$ for every $p \in P$ and returns the predicate p with the highest score as an answer.

2.5.4 Reformulator Architectures

Minervini et al. (2020b) introduce three different neural network architectures for implementing reformulators: linear, attentive, and memory-based goal reformulation.

As the most basic implementation, one can define a reformulator linearly by:

$$\texttt{select}_\theta^i(G) = F_H(G) \leftarrow F_{B_1}(G) \wedge F_{B_2}(G)$$

where the head of the rule is $F_H(G) = [f_H(\theta_{G_1}), X, Y]$, and the body of the rule consists of $F_{B_1}(G) = [f_{B_1}(\theta_{G_1}), X, Z]$ and $F_{B_2}(G) = [f_{B_2}(\theta_{G_1}), Z, Y]$. Note that the reformulator has taken on a particular structure of fixed variable positions in the rule it represents, as well as a fixed number of atoms in the body. In the above, each $f_j : \mathbb{R}^k \to \mathbb{R}^k$ can be implemented by a linear projection $f_j(x) = W_j x + b$, where $W_j \in \mathbb{R}^{k \times k}$ and $b \in \mathbb{R}^k$. This is referred to as a *linear* reformulator. Notice that a single reformulator can thus capture any number of rules, provided that the structure of each rule corresponds to the positions of the variables and number of atoms in the reformulator, and that each rule has a distinct head predicate.

This basic reformulator architecture can be improved upon by incorporating a prior from the setup of the module. Specifically, predicate symbols used in the rules that the reformulator represents already exist in the KB, within the available predicates P. This can be exploited to perform *attentive* goal reformulation, by using the goal G to generate an attention distribution over the predicate in P:

$$f_j(x) = \alpha E_P$$

where $\alpha = \text{softmax}(W_j x) \in \triangle^{|P|-1}$ is an attention distribution over P, $E_P \in \mathbb{R}^{|P| \times k}$ is the predicate embedding matrix, $W_j \in \mathbb{R}^{k \times |P|}$, and $\triangle^n$ is the standard n-simplex $\triangle^n = \{(\alpha_0, ..., \alpha_n) \in \mathbb{R}^{n+1} \mid \sum_{i=0}^n \alpha_i = 1 \text{ and } \forall i : \alpha_i \geq 0\}$.

Finally, Minervini et al. (2020b) also introduce memory-based goal reformulation, drawing inspiration from the work of Miller et al. (2016). This allows one to inspect the rules by analyzing the model parameters, as the rules are stored in differentiable memory. Attentive and memory-based reformulators seem to perform far better than linear reformulators and are comparable in accuracy to each other. Thus, for the purposes of this thesis, we will use attentive reformulators, as they converge slightly faster than memory-based reformulators. This means that fewer training epochs will be needed when evaluating the model.

2.5.5 Summary

Since we have just presented a lot of information defining CTPs as an extension of NTPs, we provide a brief summary of how CTPs operate when applied specifically to CLUTRR. The summary departs slightly from the preceding theoretical construction, as it corresponds more directly with how CTPs are implemented in code. However, it is still equivalent to the theoretical description.

A CTP model has a number of reformulators n, each of which can represent multiple rules. It also contains representations for predicates and constants. These are all initialized randomly and optimized during training. The CTP model starts with a goal $p(c_1, c_2)$, and initializes $G = \{p(c_1, c_2)\}$ as the set of sub-goals. Then, recursively up to the reasoning depth d, the model applies each reformulator to every sub-goal in G, generating a new set of sub-goals from the output of the reformulators with each recursive step. Every time the model steps down recursively, it branches out into a new proof path for each reformulator. This means n branches exist after the first step, n^2 after the second, and n^d branches after the reasoning depth has been reached. The model maximizes scores over proof paths. Once the reasoning depth is hit, the model unifies every sub-goal from the proof path with the knowledge base of facts, given in the CLUTRR task. These similarity values are propagated up, with the score of the atom in the head of a reformulator being set to the minimum similarity of all the atoms in its body.

2.6 Related Work

We identify the three main functions of CTPs to be: being able to perform systematic generalization, inducing sensible representations for predicates and constants used in a knowledge base, and performing inductive logic programming. Note that they do also have other attractive qualities, such as built-in explainability for conclusions, since any answer to a query comes with the set of reformulators and fact embeddings used to answer the query, representing the rules and facts used. By contrast, other modern natural language processing methods such as transformer architectures (Devlin et al., 2019; Wolf et al., 2020) are black boxes that do not provide explanations for their answers. Attempts have been made to make such deep learning models explainable, such as by computing the sensitivity of certain predictions with respect to changes in the input variables (Samek et al., 2018). However, these attempts do not yield anywhere near the same kind of explainability that a model with built-in explanations for conclusions, such as CTPs, does.

In the following subsections, we compare CTPs to other related models that provide one or more of these three functions, and conclude by discussing some other neuro-symbolic models.

2.6.1 Systematic Generalization

Memory-enabled neural architectures (Hochreiter & Schmidhuber, 1997) have been introduced as a potential solution to the issues that neural networks often have with systematic generalization. Memory Augmented Neural Networks introduce differentiable memory into the standard neural network architecture, allowing models to learn to represent and manipulate representations by reading from and writing to the external memory. This technique was used by Sukhbaatar et al. (2015) to perform multi-hop reasoning over text, by Santoro et al. (2016) to learn swiftly from new data, and by Graves et al. (2014); Joulin & Mikolov (2015); Grefenstette et al. (2015); Kaiser & Sutskever (2016) to induce algorithmic behaviours. CTPs separate representations and calculations in a similar manner, thus improving their generalization and reasoning abilities.

2.6.2 Knowledge Graph Embedding

Knowledge bases (which are also sometimes referred to as *knowledge graphs*) can have their facts embedded in a continuous vector space, such as how CTPs learn embeddings for the predicates and constants. This can simplify the manipulation of facts in a KB, while preserving the structure of the KB (Wang et al., 2017). These embeddings can thus be used to predicate relationships between given constants, allowing for *knowledge graph completion* (Lin et al., 2015). This is especially useful for datasets that have missing facts. While CTPs reason over knowledge bases to induce representations, other models use a variety of techniques.

Score-based models use distance or semantic similarity-based methods to score facts in a KB. Distance-based score models (Bordes et al., 2013; Wang et al., 2014; Lin et al., 2015) embed constants and predicates in the same vector space, with a distance metric providing functional dependency between the constants using translations. Semantic similarity-based models encode a similarity function between symbols, from simple functions (Yang et al., 2015; Nickel et al., 2016) to more complex ones involving different number systems such as the complex numbers or quaternions (Trouillon et al., 2016; Zhang et al., 2019). Finally, there are some methods that use reinforcement learning to learn to walk in reasoning steps over knowledge bases, and thus find paths that can predict relationships between constants (Xiong et al., 2017; Das et al., 2018; Shen et al., 2018).

2.6.3 Inductive Logic Programming

Inductive Logic Programming (ILP) tries to induce rules from facts in a knowledge base, which can then be used to answer queries (Muggleton, 1991). Works by Sammut & Banerji (1986); Quinlan (1990); Muggleton (1995); Srinivasan (2001); Muggleton et al. (2015) use symbolic techniques to implement ILP, by searching over the discrete space of logical rules to find promising rules for the KB. More recent works, such as by Cropper & Muggleton (2016) can even construct definitions for new predicates, as well as learning recursive rules. This is in contrast to CTPs, which can currently only learn rules of pre-defined structures that have to conform to the syntax of Prolog, instead of more expressive first-order logic rules. However, unlike CTPs, these symbolic ILP systems do not learn representations for symbols in the KB, and they often suffer from serious computational constraints due to the large space of possible rules.

2.6.4 Neuro-symbolic Models

Neuro-symbolic approaches to reasoning (Smolensky, 1988; Garcez et al., 2015) combine neural networks with classical symbolic reasoning. There is a plethora of work exploring the construction of neural network architectures that draw inspiration from the structure of logic languages, such as from propositional logic (Towell et al., 1990; Towell & Shavlik, 1994; Garcez & Zaverucha, 1999; Steinbach & Kohut, 2002), first-order logic (Shastri, 1992; Hölldobler et al., 1999; França et al., 2014; Donadello et al., 2017), and even some non-classical logics (Garcez et al., 2007, 2014). More recently, there has been interest in exploring neuro-symbolic models further, usually with methods based upon continuous approximations of the semantics of logic (Grefenstette, 2013; Serafini & Garcez, 2016). This has been applied to rule induction and reasoning.

In contrast to Rocktäschel & Riedel (2017), who propose NTPs as a differentiable implementation of the backward chaining algorithm, Evans & Grefenstette (2018) provide a differentiable forward chaining reasoning algorithm. Using yet another approach, Yang et al. (2017); Sadeghian et al. (2019) introduce a method for learning function-free Datalog clauses (Ceri et al., 1989) from knowledge bases, utilizing a differentiable graph traversal operator. Many of these approaches suffer from serious computational constraints, due to their high computational complexity. This means that they are unsuitable for use on larger-scale datasets with more complex examples. Minervini et al. (2020a) introduce Greedy Neural Theorem Provers (GNTPs), an extension of NTPs, as a potential solution to this problem. In GNTPs, only the top k facts and rules are used during the reasoning process.

Chapter 3

Explanation of the Problem

In this chapter, we establish the issue with Conditional Theorem Provers that we aim to address in this thesis. Minervini et al. (2020b) propose CTPs as a solution to the issues that NTPs have with computational complexity. However, as we will note, CTPs still suffer from similar computational issues, due to the number of the proof paths that need to be considered in backward chaining. We first establish the need for a potentially large number of reformulators, then motivate for why higher reasoning depths are desirable in such models. This is justified theoretically and experimentally. Then, we demonstrate how these requirements lead to CTPs suffering from computational issues, as well as outlining the proposed solution.

3.1 Number of Reformulators Needed

3.1.1 Expressivity of CTPs

The expressivity of a relational learning model is an important consideration for its viability (Natarajan et al., 2012; Trouillon et al., 2016). A more expressive model can capture more types of data and relations than a less expressive one, meaning it can be applied to more complex datasets. Due to the restrictive nature of the syntax of Prolog, CTPs are already quite limited when it comes to the structure of expressions upon which they can reason. However, these limitations go further, as the rules that a CTP model can capture depend on the structure and number of reformulators in the model.

As we have already noted, a single reformulator can capture any number of rules, provided that for each rule, the positions of the variables in the rule correspond to the positions of their representations in the reformulator. In addition, given an atom for the head of a rule, a reformulator can only capture one rule with the given head.

For example, a single reformulator would be unable to capture both of the following rules, even though the positions of the variables are the same in both:

$$\texttt{sister}(X, Y) \leftarrow \texttt{father}(X, Z) \wedge \texttt{daughter}(Z, Y)$$
$$\texttt{sister}(X, Y) \leftarrow \texttt{sister}(X, Z) \wedge \texttt{sister}(Z, Y)$$

This is because a reformulator is a function that maps from $A \in \mathbb{R}^k \times (\mathbb{R}^k \cup V) \times (\mathbb{R}^k \cup V)$, meaning that each A has to map to a unique output. As such, to fully capture all of the rules in a knowledge base, we need as many reformulators as there are the maximum number of rules with the same head atom. We refer to this number as the *minimal full expressivity bound*.

Sinha et al. (2019) do not provide a list of the rules used in CLUTRR datasets. However, we were able to find their code[1] and see the exact list of possible rules. Minervini et al. (2020b) only focus on capturing the rules with a compositional variable structure (like the ones above), so we will do the same in this work. With this in mind, the highest number of rules with the same head atom is five. For example, there are five rules that can be applied to the goal $\texttt{grandfather}(X, Y)$:

$$\texttt{grandfather}(X, Y) \leftarrow \texttt{father}(X, Z) \wedge \texttt{father}(Z, Y)$$
$$\texttt{grandfather}(X, Y) \leftarrow \texttt{mother}(X, Z) \wedge \texttt{father}(Z, Y)$$
$$\texttt{grandfather}(X, Y) \leftarrow \texttt{grandmother}(X, Z) \wedge \texttt{husband}(Z, Y)$$
$$\texttt{grandfather}(X, Y) \leftarrow \texttt{brother}(X, Z) \wedge \texttt{grandfather}(Z, Y)$$
$$\texttt{grandfather}(X, Y) \leftarrow \texttt{sister}(X, Z) \wedge \texttt{grandfather}(Z, Y)$$

So to have a fully expressive CTP model for CLUTRR, one needs at least 5 reformulators. However, CLUTRR is a relatively simple dataset with a limited number of predicates and ways of proving them. For more complex datasets, more reformulators would likely be required.

Even if a model is fully expressive, that does not guarantee that it will fully learn all of the rules in the knowledge base, merely its potential to do so. Reformulators are implemented by neural networks. Thus, during training, loss that arose from the incorrect application of one rule being backpropagated through the network of a reformulator could also update the weights that affect the other rules the reformulator represents. Depending on the random initialization of the neural network and the

---

[1] https://github.com/facebookresearch/clutrr

embeddings of the predicates and constants, there could end up being a set of weights that affect the applications of two particular rules very strongly. If these two rules require very different weights to be represented properly, then the model could be unable to learn to represent both of them.

The issue of individual reformulators lacking expressive power could be addressed by making the reformulator architectures themselves more expressive (increasing the *capacity* of the model). This could be done, for example, by increasing the number of parameters in the neural networks. However, the higher the capacity of the model, the higher the likelihood that it will overfit on the training data (Caruana et al., 2001; Salman & Liu, 2019). Thus, we will not expand upon the architectures proposed by Minervini et al. (2020b). Another potential solution to this is to use more reformulators than the *minimal full expressivity bound*, so that if a reformulator is unable to learn every possible rule with a different head, then another reformulator can learn the ones that it was unable to.

3.1.2 Experimental Results

To see the effect of increasing the number of reformulators, we fix the train and test reasoning depths, and evaluate the test accuracy of CTPs using a number of reformulators varying from 1 to 8. The evaluation is done on *1.4_test* and *1.10_test*, to test the effect this has on datasets of varying complexity. Furthermore, we present both the maximum accuracy achieved across all random seeds for each number of reformulators, as well as the average accuracy. Maximum accuracy can be used as a gauge for the expressivity of the model, as a model cannot perform better than its maximum expressivity. The average accuracy indicates how well we can expect such models to operate in general. Both sets of results can be seen in Figure 3.1 and full hyperparameter details for this evaluation can be found in Appendix A.2.

These results motivate for the benefit of using more reformulators, as doing so almost always increases the average accuracy of the model. The maximum average accuracy for both *1.4_test* and *1.10_test* is seen when 8 reformulators are used. We expect that at some point, the number of reformulators used becomes high enough that the model would start to overfit on the training data, but this is evidently still not so for 8 reformulators.

Despite the *minimal full expressivity bound* for CLUTRR being 5, the maximum accuracy in Figure 3.1 seems to increase sharply as the number of reformulators ranges from 1 to 3, and then only increases slowly after that. This seems to indicate that despite CLUTRR having the potential to use 5 different rules with the same head,

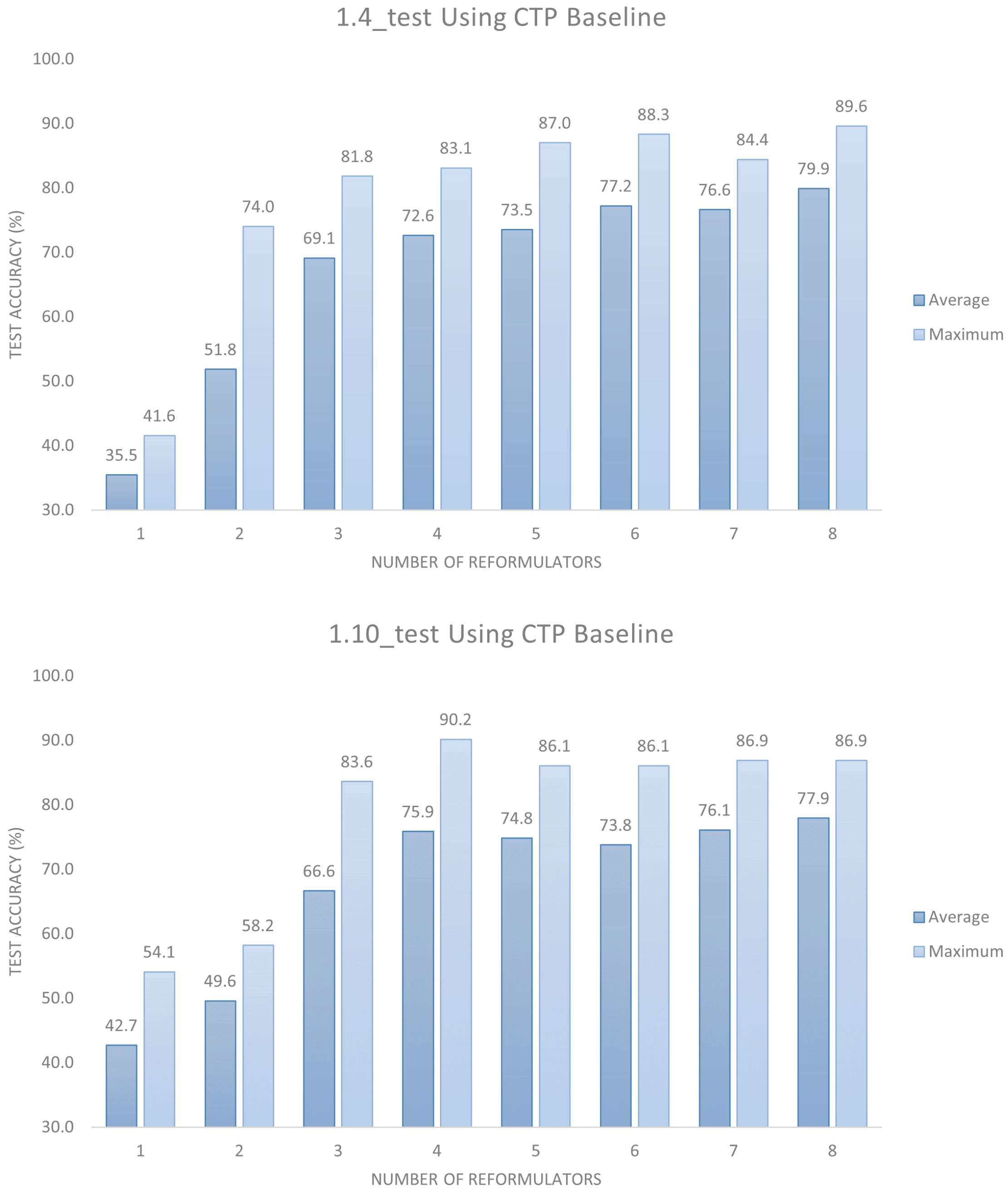


Figure 3.1: CTP average and maximum test accuracy on datasets *1.4_test* and *1.10_test* for a varying number of reformulators

it rarely uses more than 3. Another possible explanation is that the goals are able to be proved using other proof paths that do not require the use of the rules that were not captured. The comparatively high maximum accuracy achieved using 4 reformulators on *1.10_test* is an outlier, as no other runs with the same parameters achieved even close to the same accuracy. Finally, there is a big jump in accuracy when moving from 1 to 2 reformulators on *1.4_test*, whereas this jump only occurs on *1.10_test* when moving from 2 to 3 reformulators. This is due to *1.10_test* being a more complex dataset that uses more rules in its proofs.

3.2 Reasoning Depth

3.2.1 Required Reasoning Depths

Unlike the backward chaining algorithm (Gallaire & Minker, 1978), which relies upon detecting cycles of rule application to stop expanding upon a goal, CTPs reason up to a pre-defined reasoning depth and then unify the existing sub-goals with the facts in the knowledge base. This means that, even if the CTP model has perfectly learned to represent all of the rules in the knowledge base with reformulators, it still needs a sufficient reasoning depth to prove the goal. The required reasoning depth for a valid proof path is the minimum number of recursive steps down, such that every sub-goal appears in the knowledge base of facts. The reasoning depth needed to solve a task is thus the minimum required reasoning depth across all possible proof paths.

Notice that the reasoning depth of a task and the number of rule applications to solve the task are distinct. This is because with every recursive expansion upon the sub-goals up to the reasoning depth, multiple rules are being applied, each to a different sub-goal. This can be seen in the CTP `and` module. Minervini et al. (2020b) use a reasoning depth of 2 during training and a test reasoning depth of 4, as the training dataset only requires reasoning up to a depth of 2 to solve the tasks. This demonstrates the model's ability for systematic generalization, as it is trained to solve simple tasks and generalizes to more complex ones. However, higher test reasoning depths than 4 appear to be required for some of the CLUTRR datasets, as we demonstrate by example in Section 3.2.2 and experimentally in Section 3.2.3. More complex datasets than CLUTRR will likely require even higher test reasoning depths.

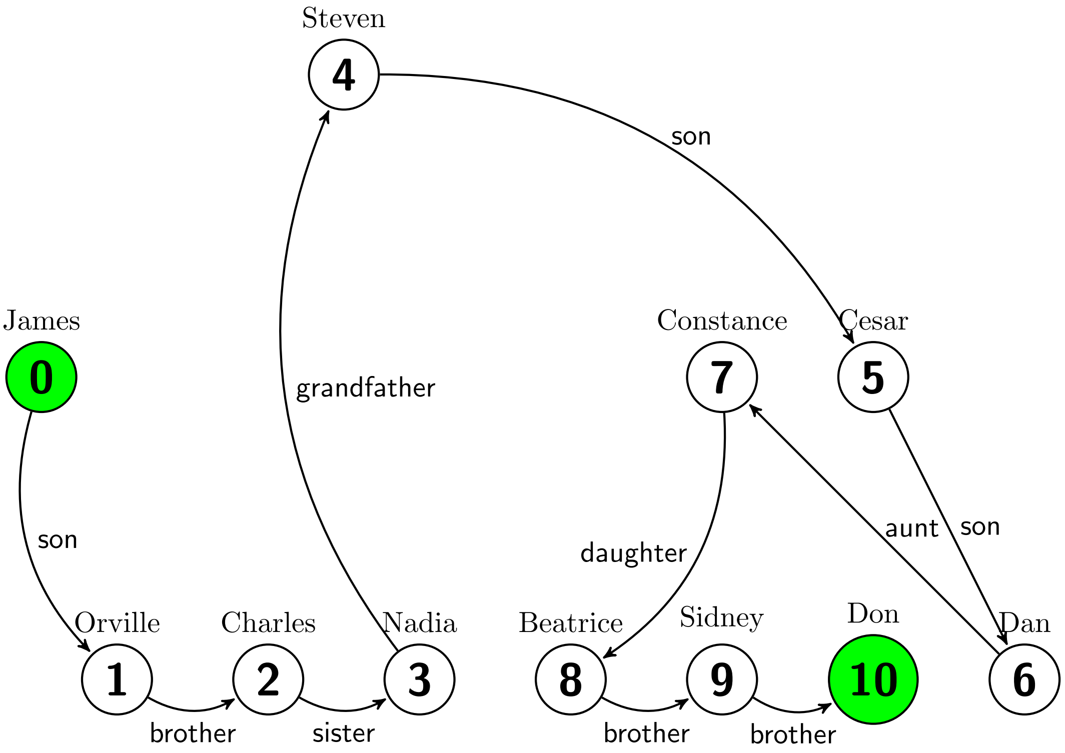


Figure 3.2: Graph representing the provided facts from an instance of the dataset *1.10_test*

3.2.2 Example

To demonstrate these ideas by means of example, we consider a task from *1.10_test*. It contains a knowledge base of facts $\mathcal{K}$ = { `daughter(Constance, Beatrice)`, `grandfather(Nadia, Steven)`, `aunt(Dan, Constance)`, `brother(Sidney, Don)`, `brother(Beatrice, Sidney)`, `son(Steven, Cesar)`, `son(James, Orville)`, `son(Cesar, Dan)`, `brother(Orville, Charles)`, `sister(Charles, Nadia)` }, which can be represented by a graph as in Figure 3.2.

The dataset entry provides the following reasoning steps to solve the query, which is the relationship between `James` and `Don`:

1. $\texttt{sister}(\texttt{Orville}, \texttt{Nadia}) \leftarrow \texttt{brother}(\texttt{Orville}, \texttt{Charles}) \land \texttt{sister}(\texttt{Charles}, \texttt{Nadia})$
2. $\texttt{sister}(\texttt{Cesar}, \texttt{Constance}) \leftarrow \texttt{son}(\texttt{Cesar}, \texttt{Dan}) \land \texttt{aunt}(\texttt{Dan}, \texttt{Constance})$
3. $\texttt{daughter}(\texttt{James}, \texttt{Nadia}) \leftarrow \texttt{son}(\texttt{James}, \texttt{Orville}) \land \texttt{sister}(\texttt{Orville}, \texttt{Nadia})$
4. $\texttt{daughter}(\texttt{Steven}, \texttt{Constance}) \leftarrow \texttt{son}(\texttt{Steven}, \texttt{Cesar}) \land \texttt{sister}(\texttt{Cesar}, \texttt{Constance})$
5. $\texttt{father}(\texttt{James}, \texttt{Steven}) \leftarrow \texttt{daughter}(\texttt{James}, \texttt{Nadia}) \land \texttt{grandfather}(\texttt{Nadia}, \texttt{Steven})$
6. $\texttt{brother}(\texttt{Beatrice}, \texttt{Don}) \leftarrow \texttt{brother}(\texttt{Beatrice}, \texttt{Sidney}) \land \texttt{brother}(\texttt{Sidney}, \texttt{Don})$

7. $\texttt{son(Constance,Don)} \leftarrow \texttt{daughter(Constance,Beatrice)} \land \texttt{brother(Beatrice,Don)}$

8. $\texttt{sister(James,Constance)} \leftarrow \texttt{father(James,Steven)} \land \texttt{daughter(Steven,Constance)}$

9. $\texttt{nephew(James,Don)} \leftarrow \texttt{sister(James,Constance)} \land \texttt{son(Constance,Don)}$

However, these reasoning steps do not all have to be applied in sequence; some of them can be done in parallel. For example, if proving in the forward direction, steps 1, 2, and 6 can be done in parallel immediately, as neither of depend on facts from anywhere but the knowledge base. There is some dependency between the applications of other reasoning steps, which we represent with a graph in Figure 3.3. So, to prove the query using backward chaining, CTPs could apply rule 9 to the goal, then rule 8 to the one sub-goal, then rule 5 to one of the sub-goals of that rule, then rules 3, 4, and 7 to the sub-goals, then rules 1, 2, and 6 to the sub-goals. After this, the remaining sub-goals can all be unified with the facts in the knowledge base. Thus, the required reasoning depth to solve this task is 5.

Note that even though it is notated as branching out from the goal, the entire graph in Figure 3.3 represents a single proof path. This is because for each sub-goal, only one reformulator was applied to generate new sub-goals. Multiple proof paths arise when multiple reformulators are considered for expanding upon each sub-goal.

3.2.3 Experimental Results

To see the effect of using different test reasoning depths, we fix the train reasoning depth and the number of reformulators. We range from a test reasoning depth of 1 to 5. The evaluation is again done on *1.4_test* and *1.10_test*. We provide the maximum accuracy across all runs for each test reasoning depth in Figure 3.4 and for the sake of brevity, the average accuracy in Figure 3.5. We also discuss the results for the other test datasets without explicitly displaying them. Full hyperparameter details for this evaluation can be found in Appendix A.3.

As seen in Figure 3.4, the maximum accuracy on *1.4_test* is achieved for a test reasoning depth of only 3. This is because none of the tasks in *1.4_test* require a reasoning depth greater than that to be solved. However, the accuracy does not decrease as the test reasoning depth continues to increase, meaning that if one is in doubt as to what to set the reasoning depth to for maximizing accuracy, one can always make it higher than needed. The non-zero accuracy that the model achieved with test reasoning depths less than 3 can be attributed to: tasks in the test set that truly did not need a reasoning depth of 3, the model learning rules that allow it to bypass the reasoning steps laid out in the CLUTRR task and solve the problem with

Each node represents the application of a rule during backward chaining, with an edge $(x) \rightarrow (y)$ meaning (x) creates a sub-goal which appears in the head of (y).

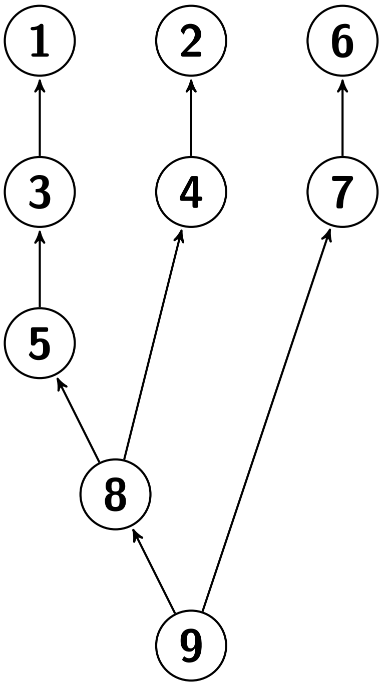


Figure 3.3: Graph representing the dependency of reasoning steps in Figure 3.2

fewer steps, or the model getting lucky and finding the correct answer with an invalid proof path. The latter is not terribly unlikely, as CLUTRR only has 22 possible predicates, meaning that even an answer given completely at random has a $\frac{1}{22}$ chance of being correct.

The accuracy on *1.10_test* continues to increase all the way up to using a test reasoning depth of 5. Tasks such as that shown in Figure 3.2 demonstrate why this is the case: some tasks in *1.10_test* need a reasoning depth of greater than 4 to be solved. The datasets of *1.9_test*, *1.8_test*, and *1.7_test* also see an increase in accuracy from using a test reasoning depth of 5 instead of 4, with the effect becoming less pronounced as the complexity of the dataset decreases. For every dataset, it is never the case that increasing the test reasoning depth lowers the accuracy.

While we have demonstrated that it is never harmful and often beneficial to increase the test reasoning depth for greater accuracy, doing so can lead to some serious issues with evaluation time.

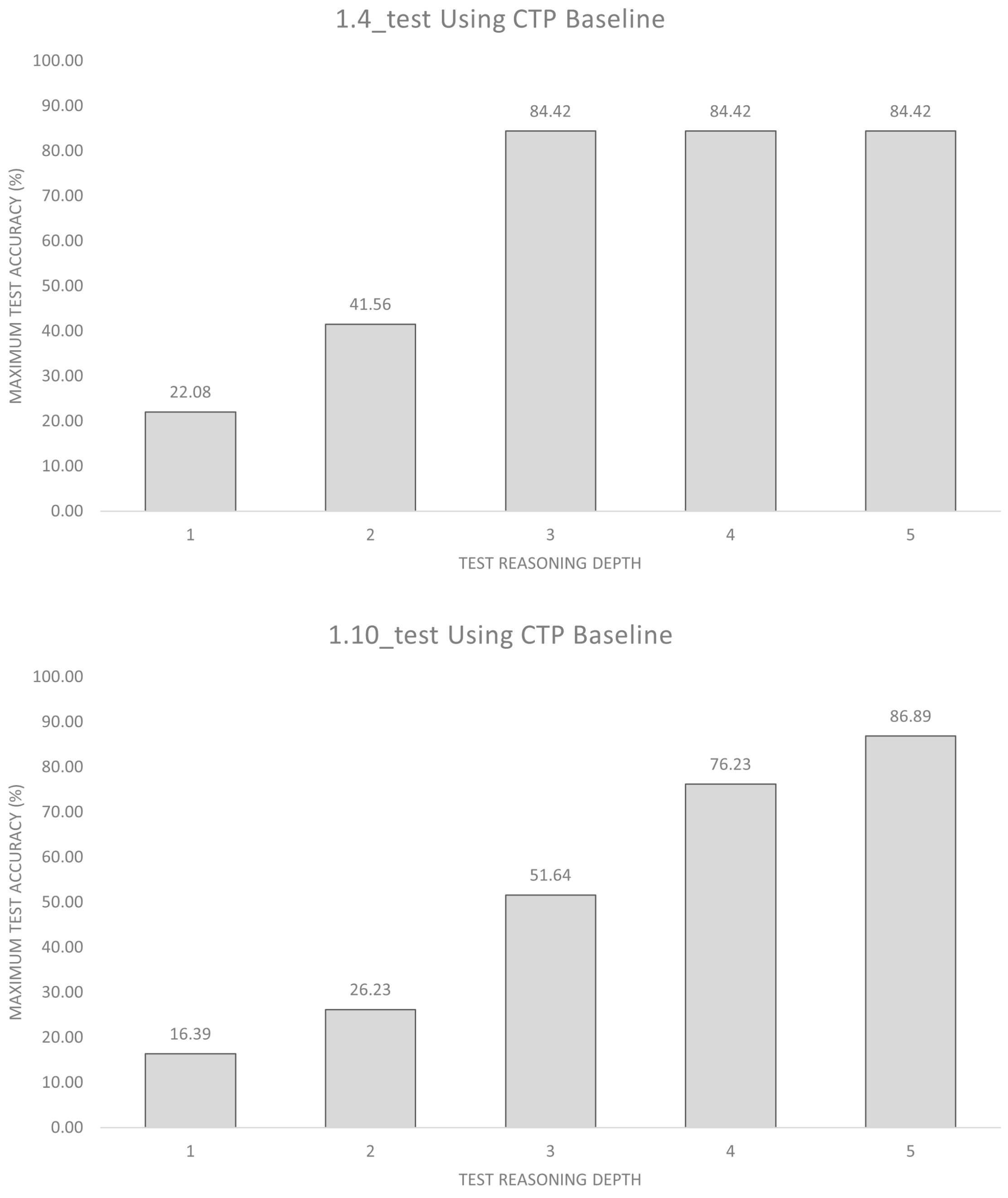


Figure 3.4: CTP maximum test accuracy on datasets *1.4_test* and *1.10_test* for a varying test reasoning depth

Dataset	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 5$
1.4_test	18.61 ± 4.01	31.17 ± 7.65	71.00 ± 9.62	71.00 ± 9.62	71.86 ± 8.89
1.10_test	15.57 ± 1.16	22.95 ± 2.68	45.90 ± 4.39	68.31 ± 7.52	75.96 ± 7.84

Figure 3.5: CTP average test accuracy and standard deviation on datasets *1.4_test* and *1.10_test* for a varying test reasoning depth

3.3 Computational Issues

3.3.1 Time Complexity Analysis

We start by presenting a theoretical analysis of the time complexity of CTPs. We consider two base operations that we wish to count: the number of reformulator applications (data being passed through a neural network) and the number of sub-goals that need unifying with the knowledge base after the reasoning depth is reached (comparisons with all fact embeddings using the RBF kernel). The remainder of the operations in CTPs are either tied into one of these two or take constant time. We count these operations with respect to the number of reformulators used n, the reasoning depth d, and the maximum number of atoms in the body of any reformulator used m.

Let s_i be the number of sub-goals after the model has applied i recursive expansions upon the goal. With each recursive step down, n reformulators are applied to every sub-goal, with each reformulator generating $\mathcal{O}(m)$ new sub-goals to be proved. This means $\mathcal{O}(nm)$ new sub-goals for each existing sub-goal, so $s_{i+1} = \mathcal{O}(s_i \times nm)$. Noting that $s_0 = 1$, representing the query passed to the model, we see that the number of sub-goals after the test reasoning depth has been reached is:

$$\begin{aligned} s_d &= s_0 \times \mathcal{O}((nm)^d) \\ &= \mathcal{O}((nm)^d) \end{aligned}$$

Furthermore, at depth i with s_i sub-goals, there are ns_i reformulator applications. Thus, the total number of reformulator applications until the reasoning depth has been reached is:

$$\begin{aligned}
r_d &= ns_0 + ns_1 + ns_2 + ... + ns_{d-1} \\
&= \mathcal{O}(n) + \mathcal{O}((nm)^1 n) + \mathcal{O}((nm)^2 n) + ... + \mathcal{O}((nm)^{d-1} n) \\
&= \mathcal{O}((nm)^{d-1} nd) \\
&= \mathcal{O}(n^d m^{d-1} d) \\
&= \mathcal{O}((nm)^d d)
\end{aligned}$$

Then since $n > 1$ and $m > 1$ for any CTP model of non-trivial complexity, $r_d = \mathcal{O}((nm)^d)$. Thus, as $s_d = r_d$, we state that the time complexity for CTPs as a whole is $\mathcal{O}((nm)^d)$. Note that this is the time complexity for a single query and that to evaluate or train on a dataset, these operations will need to be performed for every task in it. However, the number of tasks is simply linear in the size of the dataset, so this is not a concern.

The main issue with this time complexity is the raising to the power of d. Exponential time complexities get unreasonably large very quickly and can be completely untenable to work with. For a fixed depth d, the time complexity is a d-degree polynomial in n and m, which can also cause difficulties for a suitably large value of d. We have already provided motivation for using a test reasoning depth of 5 on CLUTRR, meaning that this would become degree 5 polynomial.

3.3.2 Wall-Clock Time

As CTPs have demonstrated the ability to learn using only small reasoning depths, generalizing to larger reasoning depths, the main point of concern with respect to time taken is the time taken is during evaluation. This is because CTPs can always be trained on simple examples, with the computational issues presenting themselves when the test reasoning depth has to be made high enough to make the model fully expressive. As such, we compute the average evaluation time across all CLUTRR datasets, noting that the results are not entirely stable due to them being evaluated and aggregated across a variety of machines. We show this with respect to the number of reformulators in Figure 3.6, and with respect to the test reasoning depth in Figure 3.7. The hyperparameters for these experiments can be found in Appendix A.2 and Appendix A.3 respectively.

For reference, the longest evaluation time was for a test reasoning depth of 5, with 5 reformulators, on the *1.10_test* dataset. It took 16.7 hours to evaluate.

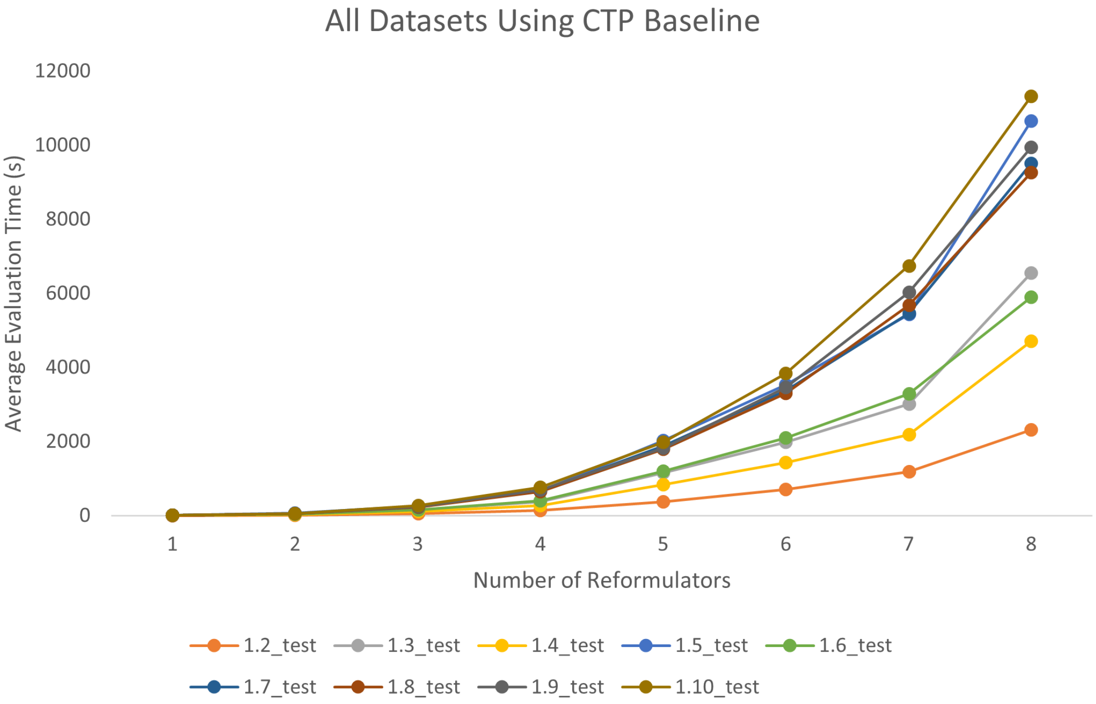


Figure 3.6: CTP average evaluation time across all datasets for a varying number of reformulators

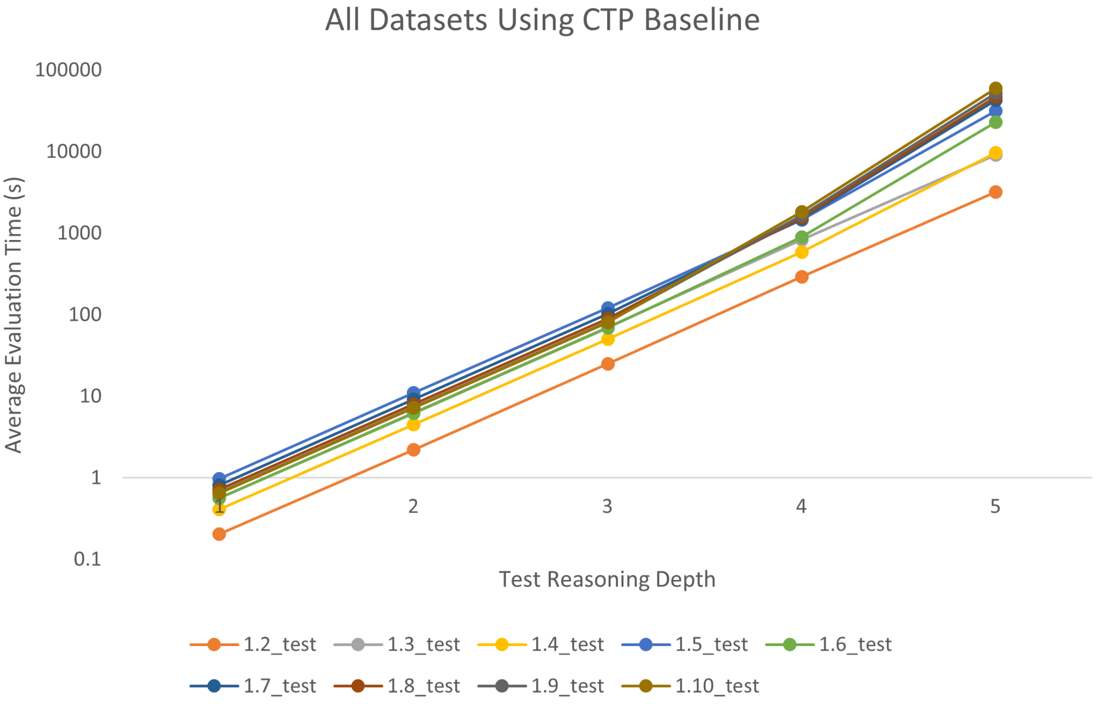


Figure 3.7: CTP average evaluation time across all datasets for a varying test reasoning depth, using a logarithmic scale

3.3.3 Outline of Solution

Since the time complexity of CTPs is $\mathcal{O}((nm)^d)$, optimizing CTP evaluation time consists of trying to keep each of these variables as low as possible. The maximum number of atoms in the body of any reformulator used m is almost impossible to reduce, as it is completely determined by the rules the model is trying to capture in the knowledge base. A possible technique for lowering it could be to use multiple rules with fewer atoms in the body to represent a single rule with more atoms in the body. For example, the rule $h \leftarrow b_1 \wedge b_2 \wedge b_3 \wedge b_4$ could instead be captured by a combination of the rules $h_1 \leftarrow b_1 \wedge b_2$, $h_2 \leftarrow b_3 \wedge b_4$, and $h \leftarrow h_1 \wedge h_2$. However, this means increasing the number of rules in the knowledge base (thus potentially increasing the number of reformulators n) and increasing the reasoning depth needed for any path containing the original rule by 1. This would likely just increase the evaluation time. However, for CLUTRR, this is not an issue, as $m = 2$ is fixed for all of the rules concerned.

The depth d is also challenging to reduce, as a certain depth is required for full expressivity on the test datasets. However, a dataset may contain tasks that do not require the maximum test depth to be solved. This is not often the case for CLUTRR, as the tasks are grouped into datasets by their difficulty. However, it does occur intermittently, and one could envision datasets where the required reasoning depth is radically different across tasks. Thus, to optimize the model, one could try adding the capacity for it to realize when it has arrived at the required reasoning depth and stop recursively expanding upon the goal. A potential way to do this would be to attempt unification with the knowledge base at every depth and stop expanding when the similarity values are all higher than a given threshold. However, we choose not to focus on such a technique in this thesis, leaving it for future work.

Our approach is an attempt to keep the number of reformulators n used at each expansion step as low as possible, minimizing the number of proof paths that need to be considered. Notice that every CLUTRR task has a valid proof path, as it is provided alongside each task. Thus, if the reformulators have perfectly captured the rules in the knowledge base and the model can identify which reformulator to apply at each reasoning step to construct the valid proof path, then only one reformulator is needed for each sub-goal expansion to achieve full expressivity. Hence, the problem to be solved is: when considering a sub-goal to be expanded, predict which reformulator is the one most likely to maximize the proof score. In the following chapter, we elaborate on our proposed method for doing so.

Chapter 4

Method

In this chapter, we introduce in general the approach that we use to solve the problem of choosing the optimal reformulator for expansion. More specific details about the various ways in which we attempted this are discussed in Chapter 5 and Chapter 6. We first provide motivation as to why this appears to be a solvable problem, then discuss how we expect this to speed up the model. Next, we introduce the machine learning paradigm that we adopt to solve the problem: reinforcement learning. We formally define the specific reinforcement learning algorithm that we use, our implementation of it, and discuss several issues encountered with the architecture. Finally, we discuss the experiment design in this thesis: what hyperparameters are used in CTPs, the training procedure we adopted, and the baseline that needs to be outperformed to call this work a success.

4.1 Optimising Proof Paths

4.1.1 Motivation for the Existence of a Solution

When mathematicians attempt to prove something, they do not simply apply every rule they know to the problem again and again, stopping when they have finally arrived at a solution. Rather, they have an intuition, built up from experience, as to which rules are going to be the most promising for constructing a valid proof. The rules needed will depend on the problem, but this is something that mathematicians have an understanding of: an understanding that continues to get better as they deal with similar problems. They will then try to use these rules, expanding upon the proof until it either succeeds or they realize that it has failed. In the latter case, they will go back and try another rule that they initially thought to be less promising. The ability of mathematicians to learn this skill indicates that we also ought to be able to train a model to select the rules for expansion that are most likely to lead to a successful proof.

Aside from this somewhat philosophical motivation, other works have already explored models that learn to traverse a knowledge base. Das et al. (2018); Xiong et al. (2017) use reinforcement learning to learn inference paths in large knowledge bases. Both of these works are based upon the path ranking algorithm (Lao et al., 2011), which uses random walks with restarts to perform several upper-bounded depth-first searches to find paths along relations. When combined with elastic-net based learning, the algorithm can learn to choose paths which are more likely to complete the inference.

Das et al. (2018) propose MINERVA, a method for searching a knowledge graph for answer-providing paths using reinforcement learning, conditioned on the query. Given a knowledge graph, it attempts to learn a policy which, given a query of the form $\texttt{predicate}(\texttt{constant}_1, X)$, starts from $\texttt{constant}_1$ and walks over relations (edges in the knowledge graph), choosing a relation at each step, conditioned upon the query predicate and the walk so far. This is done with reinforcement learning by trying to maximize the reward: reaching the correct answer constant. The model achieves state of the art accuracies on the **Countries**, **UMLS**, and **Kinship** datasets.

Xiong et al. (2017) adopt a similar but slightly simpler approach with DeepPath, which also uses reinforcement learning to find paths between pairs of constants. However, in contrast to Das et al. (2018), they also condition upon the answer constant while traversing the graph. This is because they aim to solve the simpler task of predicting whether or not a fact is true, rather than the query-answering tasks that MINERVA can be applied to. They manage to achieve results that outperform the path ranking algorithm in both the **FB15K-237** (Bordes et al., 2013) and **NELL-995** (Carlson et al., 2010) datasets.

While our proposed method is not exactly what MINERVA and DeepPath do, their existence and success at least indicate that the problem of learning which reasoning steps to take in a knowledge base is a solvable one. Finally, we also draw inspiration for our method from the work of Li et al. (2020), who provide motivation for training large transformer models and then heavily compressing them before testing. In a similar manner, our method aims to utilize a large number of reformulators when training, and then only use the selected ones during evaluation. This is particularly seen in one of our initial attempts, outlined in Section 5.4.

4.1.2 Choosing Reformulators

Given that the time complexity of CTPs is $\mathcal{O}((nm)^d)$, having n reformulators and only selecting k at each expansion step would lead to a time complexity of $\mathcal{O}((km)^d)$ instead, which is just $\mathcal{O}(m^d)$ if k is fixed. Ideally, we would just be able to use $k = 1$ and learn to select the perfect reformulator at every expansion step. However, initial experiments demonstrated this to be an almost impossible task. This makes sense, as even well-trained mathematicians rarely try only one proof path to solve their problems.

A neat solution would be to explore one proof path and then back-track to try another if it appears to fail, continuing on in this way until the problem has been solved. However, this method does not apply well to CTPs for two reasons. Firstly, the architecture of CTPs does not lend itself well to back-tracking up from a certain depth, as queries are batched together and all sent through the reformulators at the same time. The only data propagated back up is the proof scores, which are then maximized across the reformulators: corresponding to maximizing across the proof paths. Secondly, since CTPs perform unification using an RBF kernel, there is no definitive way to say that a proof path has "failed"; it will simply have a low proof score. One could potentially add a hyperparameter to the model that gives an upper bound on "successful" proof scores, with proof paths being abandoned if they do not score higher than the bound. However, we choose not to adopt this approach, as it fails to integrate into the aforementioned process of the CTP architecture. Furthermore, proof scores vary between tasks, and fixing a bound that is high enough to cut off failed proof paths but low enough to ensure that there is always at least one successful proof path is nigh impossible.

Rather, we fix some value of k as a hyperparameter of the model, to be tuned for maximizing accuracy and satisfying the computational constraints of the test datasets. This means that instead of n reformulators being used at every expansion step in the reasoning, only k are selected and used. We refer to the module making these selection decisions as the *selection module*. This is distinct from the selection module discussed in the architecture of CTPs. Henceforth, all instances of the term *selection module* refer to the module deciding which k of the n reformulators to use.

4.1.3 Wall-Clock Time Speedup

We see that if the above method is used, there is at least a theoretical speedup from the baseline of CTPs. Ideally this would translate into a speedup in wall-clock time as well, but this is not guaranteed for all datasets and values of m and d. The larger the values of m and d, the greater the effect of choosing k from n reformulators will have; this follows directly from the computational complexity $\mathcal{O}((km)^d)$. Furthermore, larger and more complex datasets will also see this effect being more pronounced, as they contain more facts that need to be unified with the $\mathcal{O}((km)^d)$ sub-goals once the test depth is reached.

Counteracting this is the overhead that comes from having to do the selection at each reasoning step, instead of just applying every reformulator. If the overhead is high enough, then better wall-clock times might not present themselves for the evaluations we do on CLUTRR. However, even if this is not the case, we argue that the theoretical speedup of this method proves its usefulness regardless: eventually the dataset complexity and reasoning depths will be high enough that the resulting theoretical speedup overcomes the overhead that comes with the method. Serious computational concerns for CTPs will occur more often for complex datasets and high reasoning depths anyway, which is exactly when the theoretical speedup becomes an advantage.

Also, note that we are mainly concerned about the wall-clock evaluation times, and not the training times. This is because when models are being used to solve tasks that require an answer to be given within certain time constraints, it is only the evaluation time that matters. How long the model took to train is independent of its usefulness in the time-constrained scenario. Furthermore, for CTPs in particular, being able to train at low reasoning depths means that computation time problems will rarely occur when training. However, high training times mean that the model takes longer to optimize, so we will also try to keep them as low as possible.

Finally, in selecting k from n reformulators, we do expect the accuracy to drop somewhat, as fewer proof paths are being considered. It will not decrease if the model is able to perfectly learn which reformulators to choose, but we expect this to not be the case.

4.2 REINFORCE

4.2.1 Reinforcement Learning

Reinforcement learning (Sutton & Barto, 2018) is a powerful machine learning technique where the model learns to map situations to actions, so as to maximize some kind of reward. The model is not told which action it ought to take, but rather learns over time which actions are good in different scenarios. It does this by trying different actions and learning to prefer the ones that lead to the greatest reward, where the environment and the reward function are defined by the creator of the model.

More formally, reinforcement learning for our purposes is performed on a *deterministic Markov decision process* (S, A, δ, R). S (the state space) is the set of states that the model can be in, A (the action set) specifies the set of actions the model can take when in a state, δ (the transition function) defines which state the model ends up in after performing an action in a state, and R (the reward function) defines the reward the model receives for ending up in a given state.

The task of the model is to learn a policy function $\pi_\theta : S \rightarrow A$ such that, given some initial state s_0, π_θ produces a sequence of states $s_0, s_1, ..., s_d$ by $s_{i+1} = \pi_\theta(s_i)$ such that the reward $R(s_d)$ is maximized after d actions. In this case, θ represents the parameters of the policy function. For our purposes, this policy function will be the agent that decides which reformulators to use to expand upon a sub-goal.

4.2.2 Policy Gradient Descent

In an environment with a finite state space, one could implement the policy π_θ as a 2-dimensional array that maps every state directly to an action, learning which actions to use over time. However, in cases such as ours where the state space is infinite and continuous, such an approach will not work. We have to turn to a Q-value approach such as Deep Q-Networks (Mnih et al., 2015) or policy gradient methods such as REINFORCE (Williams, 1992). Deep Q-Networks attempt to learn a function $Q : S \times A \rightarrow \mathbb{R}$ that approximates the reward that will result from a given state-action pair. The function Q is implemented by a deep neural network, hence why the method works for a continuous state space.

On the other hand, policy gradient methods learn a policy function $\pi_\theta : S \rightarrow [0, 1]^{|A|}$ which, given a state, outputs a probability distribution over actions that represents the model's confidence of each action maximizing the reward. This policy function is again implemented by a neural network, with a softmax function after the output layer to convert the outputs into a probability distribution. The softmax function is applied to a k-dimensional vector and is defined by:

$$\text{softmax}(x)_i = \frac{\mathrm{e}^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

We chose to adopt REINFORCE into our model as the solution for selecting reformulators. It is used by Liang et al. (2017); Das et al. (2018); Xiong et al. (2017)

to solve tasks related to reasoning and shows great promise for solving our particular problem. The basic procedure for REINFORCE can be seen in Algorithm 4.

Algorithm 4: REINFORCE

In this algorithm: θ represents the model parameters, N is the number of episodes, K is the number of episodes per batch, T is the number of steps in an episode, γ is the *discount factor* used to make further away rewards worth less, $\pi_\theta(s)_a$ gives the probability of action a from the distribution produced by π_θ applied to s, and α is the learning rate.

```
n = 0
while $n < N$ do
    for $K$ episodes in batch do
        Generate episode $s_0, a_0, R_0, ..., s_T, a_T, r_T$ using the policy $\pi_\theta$ to output probability distributions which are then sampled from to get actions
        for $t \in 1, ..., T$ do
            Calculate discounted rewards from each state: $G_t := \sum_{i=t}^{T} \gamma^i R_i$
        end
        $n := n + 1$
    end
    Calculate policy loss for entire batch: $L(\theta) := -\frac{1}{K} \sum_{t=1}^{K} \ln(G_t \pi_\theta(s)_{a_t})$
    Update policy: $\theta := \theta + \alpha \nabla L(\theta)$
end
```

4.2.3 Implementation

In this section, we introduce the specifics of our particular implementation of REINFORCE. Let us first define the deterministic Markov decision process (S, A, δ, R).

- **States**. A state in the model represents which sub-goal we are currently considering for expansion in the proof. It is thus an atom A where $A \in \mathbb{R}^k \times (\mathbb{R}^k \cup V) \times (\mathbb{R}^k \cup V)$. In order that the policy estimator may be implemented by a neural network, we define $S := \mathbb{R}^k \times \mathbb{R}^k \times \mathbb{R}^k = \mathbb{R}^{3k}$.
- **Actions**. The set of possible actions from any state is the reformulators that could be used to expand upon the sub-goal. With n reformulators, we thus have $A = \{1, ..., n\}$. Note that as we are choosing k reformulators for expansion, multiple actions are chosen for a given state by sampling without repetition from the probability distribution $\pi_\theta(s)$.

- **Transition Function**. After some subset of reformulators is chosen, the CTP model proceeds as normal, expanding out into a different proof path for each reformulator. Thus, we have a deterministic transition function δ defined by this process.
- **Rewards**. The proof score is an obvious choice for reward signal, as higher proof scores correspond to the model performing better on positive tasks. However, rather than discounting future rewards, we can use the fact that proof scores are propagated back up through the CTP model to have access to the exact score that choosing a reformulator leads to. The reward given for choosing a reformulator is thus the maximum proof score across all proof paths originating from the reformulator applied to the current sub-goal. This can be extracted directly from the CTP architecture.

The policy network is implemented by a neural network with a single hidden layer, containing 30 hidden nodes. We use a Rectified Linear Unit (ReLU) activation function before the hidden layer: $\text{ReLU}(x) := max(0, x)$. The output of the policy network is thus

$$\pi_\theta(s) := \text{softmax}(W_2 \times \text{ReLU}(W_1 \times s))$$

and the loss is given by

$$L(\theta) = -\frac{1}{B} \sum_{b=1}^{B} R_b \times \ln(\pi_\theta(s_b)_{a_b})$$

where B is the batch size, a_b is the action chosen in a particular task in the batch, R_b is the proof score that resulted from the action, and $\pi_\theta(s_b)_{a_b}$ is the probability that action a_b has in the distribution $\pi_\theta(s_b)$. The loss is applied separately for each of the k actions (reformulators) chosen. Rather than having episodes, we simply execute the CTP model as usual, applying the policy and calculating the loss at every expansion step in the reasoning. For disambiguation, we refer to this model as *RL-CTP* and the original baseline CTP model as *CTP*.

We also considered an alternative to this architecture but chose to rather implement the above instead. In this alternative, instead of there being one policy network that generates distributions over the reformulators, each reformulator would have its own neural network that, given the current sub-goal, should determine whether or not to use the reformulator for expansion. The reward given to each neural network would again be the proof score, but regularized to prevent the model from always using the reformulator (as always applying every reformulator is the best way to maximize the proof score). Getting the regularization fine-tuned enough to ensure that multiple reformulators are being chosen, but that not all of them are, appeared to be a very difficult problem. As such, we decided to rather focus on the above-described implementation, as it allows one to specify an exact number of reformulators to use with each expansion.

4.2.4 Issues Encountered with the Architecture

There were several issues in the REINFORCE architecture that we experienced during development. These are issues particular to how the reinforcement learning is implemented and not to how it is integrated into the existing CTP code. We addressed all of them and will outline the solutions briefly in this section. Further improvements that could be made to the architecture are discussed in Section 6.4.

The first issue encountered was that the model exhibited numerical instability during training. The gradients of the model would frequently explode, yielding unreasonably large weights in the neural network. These large weights lead to very high values being passed through to the softmax function, giving undefined probabilities as outputs. Around 78% of runs failed due to this issue. We addressed it by replacing the softmax function with a log_softmax function, which is known to be numerically more stable and less likely to lead to undefined probabilities[1]. log_softmax(x) is defined as log(softmax(x)), but it is not computed that way. Instead, it is computed as:

$$\begin{aligned}
\text{log\_softmax}(x) &= \log(\text{softmax}(x)) \\
&= \log(\frac{\mathrm{e}^{x}}{\sum_{j=1}^{k} e^{x_j}}) \\
&= \log(\frac{\mathrm{e}^{x-b}\mathrm{e}^{b}}{\sum_{j=1}^{k} e^{x_j-b}e^{b}}) \\
&= \log(\frac{\mathrm{e}^{x-b}}{\sum_{j=1}^{k} e^{x_j-b}}) \\
&= (x-b) - \log(\sum_{j=1}^{k} e^{x_j-b})
\end{aligned}$$

---

[1] https://discuss.pytorch.org/t/how-to-avoid-nan-in-softmax/1676

Setting $b := \max_{i \leq k}(x_i)$, the final line of the equation, which is used to compute the value of $\text{log\_softmax}(x)$, has stability against overflow and underflow[2]. We use log_softmax instead of softmax for all computation, only calculating the softmax probabilities when needed for sampling by exponentiating: $\text{softmax}(x) = \mathrm{e}^{\text{log\_softmax}(x)}$.

Another issue we had not accounted for in the original design is that the probability distribution $\pi_\theta(s)$ may contain fewer non-zero elements than k. This means that when sampling without replacement, it is not possible to get k different choices for reformulators to use. However, the architecture outputting such a distribution is not an inherently bad phenomenon; it can be a good sign that the model is very confident as to which reformulators will maximize the proof score. Thus, if such predictions arise, we only expand k' reformulators, where k' is the number of non-zero entries in $\pi_\theta(s)$.

A third problem, which only occurred to us after many experiments showing promising results had been run, is that the reward signal being used was the proof score across all queries, not just the positive tasks. This is an issue, as the target proof score of a negative task is zero, so the model should be acting to minimize the proof score for negative tasks. However, this did not seem to disrupt the training of the model, so we did not address the perceived issue. Since the reformulators are pre-trained before the selection module is trained (discussed in Section 4.3.2), most negative tasks have low proof scores for all proof paths, so it is almost impossible for the selection module to choose reformulators that maximize that proof score. On the other hand, positive tasks have particular proof paths that, when used, will maximize the proof score. Thus, the selection module only really learned to maximize proof scores when positive examples were given to it. Nevertheless, a potential way to address this issue is discussed in Section 6.4.

The final issue encountered was how to deal with variables in the selection module. The policy estimator is a function from $\mathbb{R}^{3k}$, so real values have to be assigned to variables to use the policy estimator for expanding upon a sub-goal with at least one variable. We simply fixed the value of a variable to be $\{0\}^k$. This is not ideal, as the embedding $\{0\}^k$ already has some kind of meaning in the context of the knowledge base embeddings. However, this solution yielded satisfactory results, so we left it as is. Again, potentially better ways to address this are discussed in Section 6.4.

---

[2] https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick

4.3 Experiment Design

4.3.1 Model Hyperparameters

CTPs have many hyperparameters that can be used to tune the model. They are shown and explained in Figure 4.1. To save on training time, we assume that the fine-tuning done by Minervini et al. (2020b) was done optimally and adopt their hyperparameters to be used in our CTP models. Full details of these fixed hyperparameters can be found in Appendix A.1. In particular, we fixed the training depth to 2 and the testing depth to 4. Even though the benefits of using RL-CTPs will manifest more clearly when using a test depth of 5 and such a test depth is needed for full expressivity, we choose to stick to the test depth of 4, as this allows us to use the optimized hyperparameters of Minervini et al. (2020b).

To extend CTPs to RL-CTPs, we added 3 new hyperparameters. The *rl-learning-rate* specifies the learning rate of the policy estimator, *rl-actions-selected* specifies how many reformulators k should be chosen at each expansion step, and *rl-epochs* specifies how many epochs the selection module should be trained for. We build upon the existing implementation[3] of CTPs in PyTorch (Paszke et al., 2019), augmenting it with our proposed selection module.

4.3.2 Training Procedure

Our initial approach to training the selection module was to train it in conjunction with the reformulators. However, this yielded extremely poor results, which we can attribute to three issues we perceive in the training approach. The first is that, since the reformulators take a reasonable amount of training to begin to represent actual rules that hold in the knowledge base, the optimal proof paths being learned by the selection module up until that point are meaningless. In such a scenario, the module is initially being trained to learn reasoning patterns that will not hold true when the reformulators have converged to actual rules.

We also noticed that during training, the selection module very quickly learned to always prefer some reformulators over others. This is a problem that compounds upon itself, as the same set of reformulators being selected more often means that they will be trained more often, leading to them better representing rules in the KB, and the selection module being more likely to pick them going forward. The RL-CTP model then ends up only having some k' reformulators which are always chosen, with

---

[3] https://github.com/uclnlp/ctp

Name	Type	Explanation
batch-size	int	The number of elements in each training batch
embedding-size	int	The size k of the embedding space $\mathbb{R}^k$
epochs	int	The number of training epochs
evaluate-every	int	How often (in terms of number of epochs) to evaluate on the provided test sets
hops	string	A list of integers $r_1, r_2, ..., r_n \in \mathbb{Z}$, with each r_i specifying to include a reformulator i of compositional variable structure, with r_i atoms in the body. For CLUTRR, $r_i = 2\ \forall i$
init	string	How to initialize the embeddings (e.g. "uniform" meaning to sample from a uniform distribution)
init-size	float	How much to scale up the predicate embeddings by after initialization
k-max	int	How many of the top scores to retain when expanding
learning-rate	float	The learning rate for the entire model
max-depth	int	The reasoning depth during training
nb-rules	int	The number of rules to store when using a memory reformulator
optimizer	string	Which PyTorch optimizer to use
ref-init	string	How to initialize the reformulators
reformulator	string	Which reformulator architecture to use from the options of linear, attentive, and memory
seed	int	The seed used to initialize all random processes in the model
slope	float	The slope of the RBF kernel
test	string	The file path to the test datasets to use for evaluation
test-batch-size	int	The batch size during testing
test-max-depth	int	The reasoning depth during testing
tnorm	string	How to combine proof scores together from the subgoals of a reformulator. We always use minimization, but the architecture also makes multiplication and averaging available
train	string	The file path to the dataset to use for training

Figure 4.1: CTP Hyperparameters

the other $n - k'$ reformulators never chosen and barely even trained. This completely defeats the point of using RL-CTPs over CTPs, as one may as well instead use a CTP model with k' reformulators.

The final issue is that having such a training approach makes it impossible to tune how many epochs the reformulators are trained for without also affecting how long the selection module is trained for. This becomes particularly problematic if one needs to stop training either the reformulators or the selection module at some cut-off point to prevent over-fitting.

Instead of this, we adopt the procedure of training the reformulators first, in the normal CTP way. Then, once the reformulators are trained, we train only the selection module, no longer applying the loss to the reformulators. This alleviates all of the above-described issues, as it means the selection module is learning to select proof paths over actual rules in the knowledge base, all reformulators are properly trained, and we can independently control how long the reformulators and selection module are trained for. This does mean extra training time, but it only multiplies it by a constant factor of 2, so the time complexity of training does not change.

4.3.3 Outperforming the Baseline

As already stated, to prove that RL-CTPs can be an improvement to CTPs, we do not need to demonstrate a speedup in wall-clock time, as the theoretical speedup suffices. However, we require that, at the very least, the accuracy obtained when using n reformulators and choosing k is better than just using k reformulators. If it is not, then there is no point in using RL-CTPs, as using k reformulators in CTP models has the same time complexity as having n and choosing k in RL-CTP models.

Note that while we do care about the average performance of the models across all seeds, it is more important to consider the best performing random seed, as the seed can also be optimized as a hyperparameter. This is especially pertinent for deep reinforcement learning architectures, such as that used in RL-CTPs, where the model initialization can play a large part in determining the accuracy of the model (Colas et al., 2018; Henderson et al., 2018; Clary et al., 2018). Thus, for a fair comparison, we apply the same hyperparameter tuning to all of our models. To get the accuracy of a model, we run it across 6 different seeds and select the model that displays the highest accuracy on the evaluation dataset. We then report the model accuracy on all of the test datasets. This accuracy prediction is made more confident by repeating the preceding process 5 times and reporting the mean across all 5 of the optimization procedures.

For optimizing hyperparameters, we adopt the same approach to evaluation and test sets as Minervini et al. (2020b). We perform two different optimizations: the first has hyperparameters tuned on an evaluation set of *1.3_test* and is tested on all other datasets, and the second is tuned on an evaluation set of *1.9_test*. The full model training procedure is described in Algorithm 5.

In this chapter, we argued for the existence of a solution to the problem of choosing the optimal reformulators for expansion, as well as demonstrated how this allows us to solve the computational issues faced by CTPs. Using policy gradient descent, we formally defined a model architecture for RL-CTPs: our proposed extension of CTPs. Lastly, we also noted what is required for RL-CTPs to be an improvement upon CTPs. In the following two chapters, we discuss our concrete implementations of RL-CTPs.

Algorithm 5: Model training procedure

```
def get_best_model(model, train_set, hyperparameter_grid, evaluation_set)
    best_model = None
    best_accuracy = -1
    for hyperparameter_set ∈ hyperparameter_grid do
        train_model(model, train_set)
        accuracy_value = accuracy(model, evaluation_set)
        if accuracy_value > best_accuracy then
            best_accuracy = accuracy_value
            best_model = model
        end
    end
    return best_model
end

def get_best_seeded_model(model, train_set, hyperparameter_grid,
  evaluation_set)
    pick 6 seeds $S = \{s_0, ..., s_5\}$ at random
    include seeds $S$ in hyperparameter_grid
    return get_best_model(model, train_set, hyperparameter_grid,
      evaluation_set)
end

def get_model_accuracy(model, train_set, evaluation_set, test_sets)
    initialize hyperparameter_grid
    total_accuracy = 0
    for 5 iterations do
        best_model = get_best_seeded_model(model, train_set,
          hyperparameter_grid, evaluation_set)
        test_accuracies = accuracy(best_model, test_sets)
        total_accuracy += test_accuracies
    end
    return total_accuracy / 5
end

get_model_accuracy(model, train_set, 1.3_test, test_sets)
get_model_accuracy(model, train_set, 1.9_test, test_sets)
```

Chapter 5

Initial Attempts

In this chapter, we cover in more detail the various approaches we adopted to implement the architecture described in Chapter 4. More specifically, we outline each of our failed attempts to solve the problem before we arrived at the solution described in Chapter 6. We do this to report upon the work that we did, show the thought process that led to the working solution, give insight as to why various methods failed, and provide more information about the inner workings of CTPs. Since the methods did not yield satisfactory results, we refrain from providing technical details around their implementations, and rather save this for the working solution. We begin by discussing our first attempt, a recursive solution, following on to our attempted improvement upon it by performing the same process iteratively. Next, we discuss the performance of CTPs when only certain subsets of reformulators are utilized during evaluation, using this as motivation for why our first batch element solution might be viable. Finally, we present and analyze the results of our first batch element solution.

5.1 Recursive Method

5.1.1 Core Implementation Issue with the Architecture

While the methodology has been almost fully described in Chapter 4, there is a core problem with integrating the selection module into the existing CTP architecture. It is this which led to the sequence of failed attempts to implement RL-CTPs. The issue arises around batching; when proposing NTPs, Rocktäschel & Riedel (2017) batch process many proofs together, so that they may be expanded upon in parallel on GPUs. The same approach is taken by Minervini et al. (2020b) for CTPs.

Sub-goals, represented by *tensors*, are passed through in batches to the methods (computer programming methods) responsible for expansion, with every reformulator then being applied to every sub-goal in the batch. This process is summarized in Algorithm 6. However, in RL-CTPs, for each sub-goal in the batch, only k specific reformulators ought to be used to expand upon it. This means that the tensors representing the sub-goals can no longer be passed as one cohesive batch through the neural networks of the reformulators. Instead, each sub-goal in the batch needs to be passed through a specific subset of k neural networks. Figuring out how to do this efficiently was at the heart of the problems we encountered.

Algorithm 6: Conditional theorem proving: batch expansion

```
$S$ is a batch of sub-goals.

def expand(S)
    for each reformulator $r_i$ do
        $S_{\text{new}} := r_i(S)$
        Then carry on with new sub-goals $S_{\text{new}}$
    end
end
```

5.1.2 Naive Solution

Our initial approach was to utilize the existing CTP methods for expansion, by simply adding extra recursion to them. If a method receives a batch with more than 1 element, it will iterate through all of the elements of the batch, calling recursively to itself with the same arguments, but with only a single element in the batch. The results of all these recursive calls are then collected back into a batch and returned. If a method receives a batch with only 1 element, then it calls to the selection module to get a list of reformulators to use for the sub-goal in the batch, skipping all reformulators that were not included in the list. The method otherwise proceeds as normal, with the resulting proof scores being used as rewards to update the selection module. The method is illustrated in Algorithm 7.

While relatively simple to understand and to implement, this method yielded immediately disastrous results. A single training epoch took around 41 hours to run and evaluation on *1.10_test* took around 17.5 hours. Despite this still being faster than CTPs in a theoretical sense, we judged that the training times were far too long for this method to be feasible and decided to abandon it in favour of an iterative attempt to achieve the same result. We suspected that much of the increase

Algorithm 7: Recursive RL-CTPs

```
S is a batch of sub-goals.

def expand(S)
    if $|S| > 1$ then
        results = []
        for $j \in \{0, 1, ..., |S| - 1\}$ do
            results[$j$] = expand($S[j]$)
            return results
        end
    end

    $s := S[0]$                              /* The only sub-goal in $S$ */
    selected_reformulators = selection_module($s$)

    for each reformulator $r_i \in$ selected_reformulators do
        $S_{\text{new}} := r_i(S)$
        Then carry on with new sub-goals $S_{\text{new}}$
    end
end
```

in computation time was coming from all the recursive calls being made, as there is significant overhead in doing so[1]. Thus, we judged that if the same solution could be implemented iteratively, without the need for so many recursive calls, it might solve the computational issues.

5.2 Iterative Method

5.2.1 Mismatching Batch Sizes

Like the recursive attempt, this solution integrates into the existing CTP methods that are used for expanding upon sub-goals. It begins by addressing a problem that was not even considered in the recursive solution, as it was already far too slow to investigate further. The issue is that the aforementioned CTP methods also have the fact and constant embeddings passed along as arguments, to be used when the reasoning depth is reached and the sub-goals are unified with the knowledge base. When a sub-goal is expanded upon into m new sub-goals (recall that this is always $m = 2$ for CLUTRR), the next recursive step down has both of the sub-goals as

---

[1] https://tech.marksblogg.com/faster-python.html

batch elements. Thus, since this is done for every batch element, the number of batch elements is being multiplied by the constant factor m with each recursive step.

However, CTPs save on expensive copying operations by not copying the batches of fact and constant embeddings as well. Instead, they use the fact that for each reformulator, if there are b current batch elements and m sub-goals being expanded to, then there are bm sub-goals in the new batch. Due to how sub-goals are positioned in the batch, a new batch sub-goal with index $= i$ corresponds to the fact and constant embeddings with indices $= i \bmod \frac{b}{m}$. This does not work for RL-CTPs, as each reformulator is only expanding upon some subset of the batch elements, meaning that the number of sub-goals in the new batch will no longer be exactly bm, but will depend on how many sub-goals the reformulator was selected for.

5.2.2 Outline of Solution

To solve the above-described issue, the method begins by copying batch elements of the fact and constant embeddings to ensure they match up to the corresponding sub-goals. Then, the method calls to the selection module to get a list of reformulators to use for each sub-goal in the batch. For each reformulator i, it creates a new batch of sub-goals, fact embeddings, and constant embeddings. It does this by including every batch element for which the selection module returned a list of reformulators containing i. This is implemented by iterating over every element of the full batch, including the selected elements in the new constructed batch. The sub-goals of the new batch are then passed through the reformulator to expand them into new sub-goals, with the rest of the CTP operations proceeding as normal. A summary of this process is presented in Algorithm 8.

5.2.3 Speed

While an improvement upon the naive recursive solution, this method was still unreasonably slow. Each training epoch took around 8 hours and it took 3.5 hours to evaluate on *1.10_test*. This is partially due to the expensive copying operations that needed to be performed but has more to do with how the new batches for each reformulator were constructed. Along with most modern machine learning libraries, PyTorch (Paszke et al., 2019) is optimized to execute tensor operations, such as multiplication, in parallel on powerful GPUs. Iterating through a tensor and performing operations on each element is not a process that GPUs excel at, and the loss in speed we experienced is largely attributed to it.

Algorithm 8: Iterative RL-CTPs

```
S is a batch of sub-goals, F is a batch of fact embeddings, and C is a batch
 of constant embeddings. copy(F, m) produces a sequence of m copies of F.

def expand(S, F, C)
    if $|S| > |F|$ then
                                    /* Ensure matching batch elements */
        $m := \frac{|S|}{|F|}$
        $F := copy(F, m)$
        $C := copy(C, m)$
    end

    selected_reformulators = selection_module(S)        /* Selected
     reformulators for each sub-goal in the batch */

    for each reformulator $r_i$ do
        $S_i, F_i, C_i := []$  /* Include the batch elements of the arguments
         for which this reformulator was selected */

        for $j \in \{0, 1, ..., |S| - 1\}$ do
            if $r_i \in selected\_reformulators[j]$ then
                Include $S[j]$ in $S_i$
                Include $F[j]$ in $F_i$
                Include $C[j]$ in $C_i$
            end
        end

        $S_{\text{new}} := r_i(S_i)$
        Then carry on with new sub-goals $S_{\text{new}}$
    end
end
```

We attempted to move the entire model off the GPU and onto the CPU, but doing so did not result in any significant speedup. This is because machine learning algorithms are so well suited to being trained in batches on graphics cards, that it is almost impossible to achieve comparable speeds using a CPU, even if the code is well optimized. This is a well-known phenomenon that is noted in many works, including by Steinkraus et al. (2005); Raina et al. (2009); Baldini et al. (2014); Schlegel (2015); Efthymiou et al. (2019). Thus, any feasible solution will have to be implemented using tensor operations that can be performed efficiently on the GPU.

5.3 Reformulator Subsets

This section does not outline an actual solution we attempted. Rather, it serves to provide motivation as to why the solution described in Section 5.4 might produce promising results, as well as give insight into the inner workings of CTPs. We present an analysis of the strengths of various subsets of reformulators after they have been trained in a CTP model.

5.3.1 Relative Strengths of Reformulator Subsets

Since all reformulators have different random initializations, they all converge to different local optima. Thus, it is clear that some reformulators will capture more rules than others, and that some will learn a particular rule better than the others. Moreover, certain subsets of reformulators are likely to contribute more to proofs than others, with subsets of the reformulators that better cover the range of rules in the knowledge base giving higher accuracies when used for evaluation. We demonstrate this empirically in Figure 5.1, where we train 5 reformulators across 10 different seeds and then use 4 random subsets of 3 reformulators each for evaluation. The accuracies of the highest and lowest scoring subsets are shown. This is demonstrated similarly for 8 reformulators.

We see that there is consistently a large discrepancy between the performance of different subsets, indicating that when it comes to maximizing accuracy during evaluation, there are reformulators whose inclusion in the model is far more important than that of other reformulators.

5.3.2 Average Reformulator Strength

We also present the following hypothesis: as the number of reformulators increases, individual reformulators become weaker. To put this more formally: as we use more

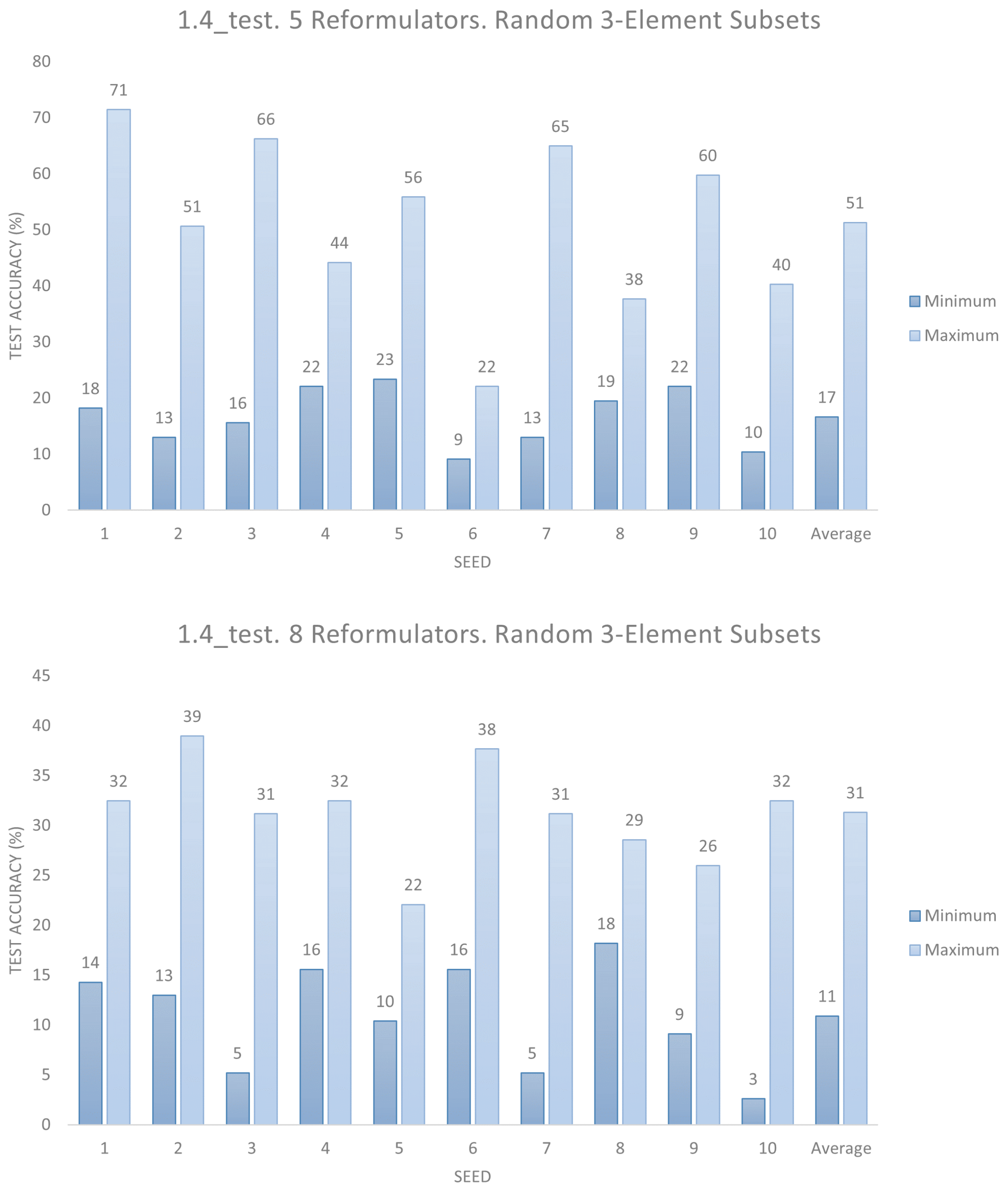


Figure 5.1: CTP maximum and minimum accuracy on *1.4_test* when 5 or 8 reformulators are trained, evaluating using 4 random subsets of 3 reformulators each. Results shown across 10 seeds

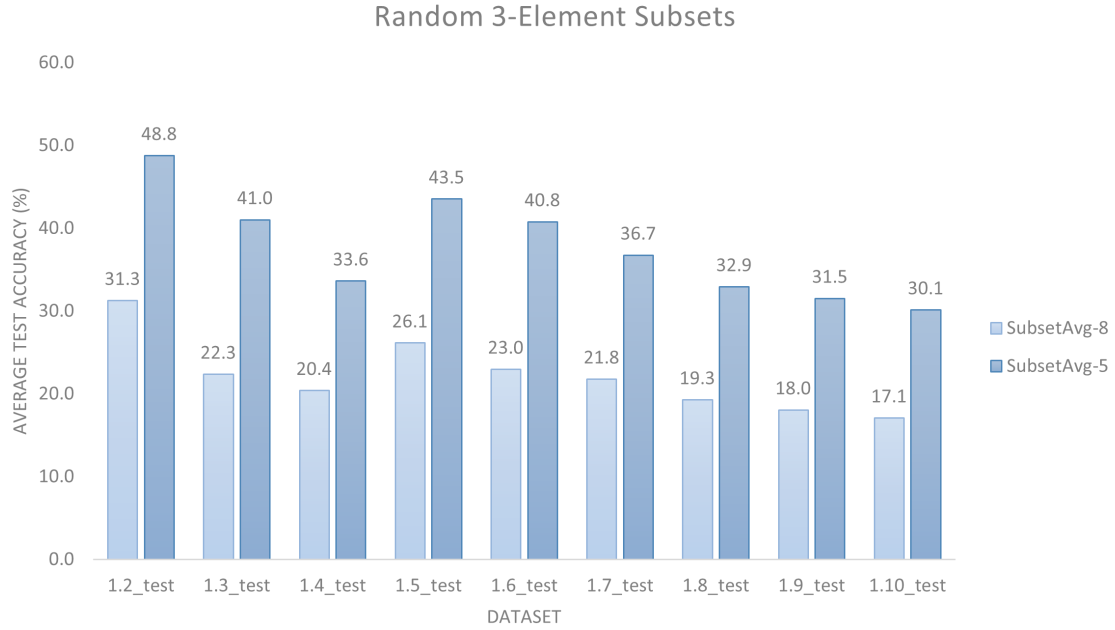


Figure 5.2: CTP accuracy across all datasets, using 5 and 8 reformulators during training, evaluating using 4 random subsets of 3 reformulators each

reformulators during training, the expected accuracy when using a fixed-size subset of the reformulators during evaluation decreases. This is an important hypothesis to note and prove, as it means that the task of selecting the best reformulators for a proof becomes harder as the total number of reformulators increases.

This hypothesis is supported empirically by the results in Figure 5.2. We trained CTP models with 5 and 8 reformulators respectively, reporting the average accuracy when 4 random subsets of 3 reformulators were used for evaluation. The results clearly show that for every dataset, expected accuracy when using subsets of 3 reformulators is higher when fewer reformulators are trained. This holds despite the fact that, as demonstrated previously in Section 3.1.2, using more reformulators tends to increase the accuracy of the model. We attribute this to the idea that having more reformulators leads to them relying on each other while learning, meaning that an individual reformulator may not need to learn a rule that it is able to, as another reformulator has already learnt to capture that rule. This makes individual reformulators weaker.

5.3.3 Maximizing Subset Performance

To get an indication for how well subsets of reformulators can perform when the optimal subsets are selected, we propose the following method. Following the training

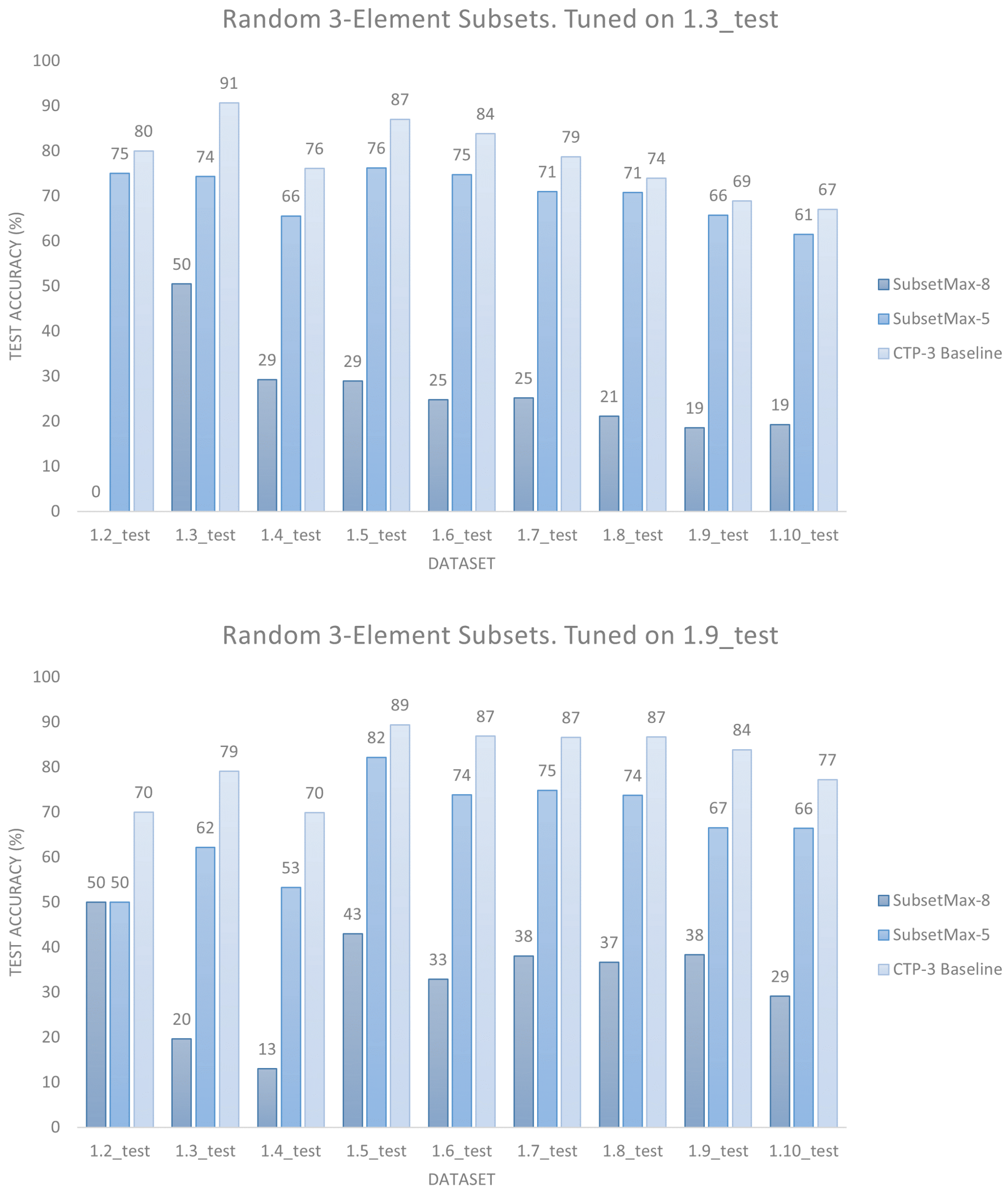


Figure 5.3: Comparison across all datasets of accuracy achieved using SubsetMax with 5 reformulators, SubsetMax with 8 reformulators, and CTPs with 3 reformulators. Hyperparameters tuned on *1.3_test* and *1.9_test*

procedure described in Algorithm 5, we use 4 random subsets of 3 reformulators, selecting the subset that maximizes the accuracy on the evaluation dataset. We refer to this method as SubsetMax-n, where n is the number of reformulators used during training. In Figure 5.3, we compare SubsetMax-5 and SubsetMax-8 to CTPs in which 3 reformulators were used, which we refer to as CTP-3. The CTP model was optimized in the same way: using the process described in Algorithm 5. While not an exhaustive search over all possible reformulators subsets, this method at least gives an indicator for the accuracy that the best performing subset of reformulators can achieve.

We see that while CTP-3 outperformed both SubsetMax-5 and SubsetMax-8 across all datasets, SubsetMax-5 achieved at least competitive accuracies in some cases. As expected from our analysis in the preceding section, SubsetMax-5 outperformed SubsetMax-8 across all datasets, due to individual reformulators being weaker when more reformulators are used. The results indicate that the best performing subset of reformulators may have comparable performance to baseline CTPs. Thus, instead of trying every possible subset of n reformulators (of which there are 2^n), we propose a method whereby we use RL-CTPs to learn which reformulators perform the best. This method draws inspiration from the work of Li et al. (2020), in that our CTP model is “trained large” by training more reformulators than needed, and then “compressed” by trying to learn the best performing subset of reformulators, using them for evaluation.

5.4 First Batch Element Approximation

5.4.1 Outline of Solution

To learn which reformulators perform best overall, we implement RL-CTPs by conditioning them on only the first sub-goal of the batch. The decision as to which reformulators to use is then applied to every element in the batch, with the resulting proof scores being used to update the policy estimator. This means that the selection module performs optimally when it learns to select reformulators that maximize the proof scores across all batch elements. Being conditioned on the first sub-goal of the batch allows this method to scale well to datasets and situations where similar examples are all batched together, since the reformulators selected for expanding one sub-goal will likely perform well on the others as well. However, this is not the case for the CLUTRR datasets used in this thesis. The method is illustrated in Algorithm 9.

Algorithm 9: FirstBatch RL-CTPs

```
S is a batch of sub-goals.

def expand(S)
    selected_reformulators = selection_module($S[0]$)    /* Condition the selection module on only the first sub-goal in the batch */

    for each reformulator $r_i \in$ selected_reformulators do
        $S_{\text{new}} := r_i(S)$
        Then carry on with new sub-goals $S_{\text{new}}$
    end
end
```

This method solves the issue of individual batch elements needing different reformulators applied to them, as the same set of reformulators is applied to every sub-goal in the batch. This means that there is no need for expensive tensor copying operations or iteration over the batch elements, which solves the computational issues encountered in the iterative and recursive methods. We refer to the method as FirstBatch-nCk, where n reformulators are trained and k are selected for expansion. The full list of hyperparameters used to evaluate the method can be found in Appendix A.4.

5.4.2 Speedup

As shown in Figure 5.4, FirstBatch-8C3 models are significantly faster than CTPs with 8 reformulators, as they only use 3 reformulators for evaluation. They are however slightly slower than CTPs with 3 reformulators, due to the overhead that comes from the selection module. These results show promise for any implementation of RL-CTPs that does not have too much overhead when it comes to integrating the selection module.

5.4.3 Results

Accuracy results across all datasets are shown for FirstBatch-8C3 in Figure 5.5 and for FirstBatch-8C2 in Figure 5.6. We compare FirstBatch-8C3 to both the average accuracy when a subset of 3 reformulators is used and the CTP-3 baseline, whereas we only compare FirstBatch-8C2 to the CTP-2 baseline. The fact that FirstBatch-8C3 consistently and significantly outperforms the average accuracy when using a

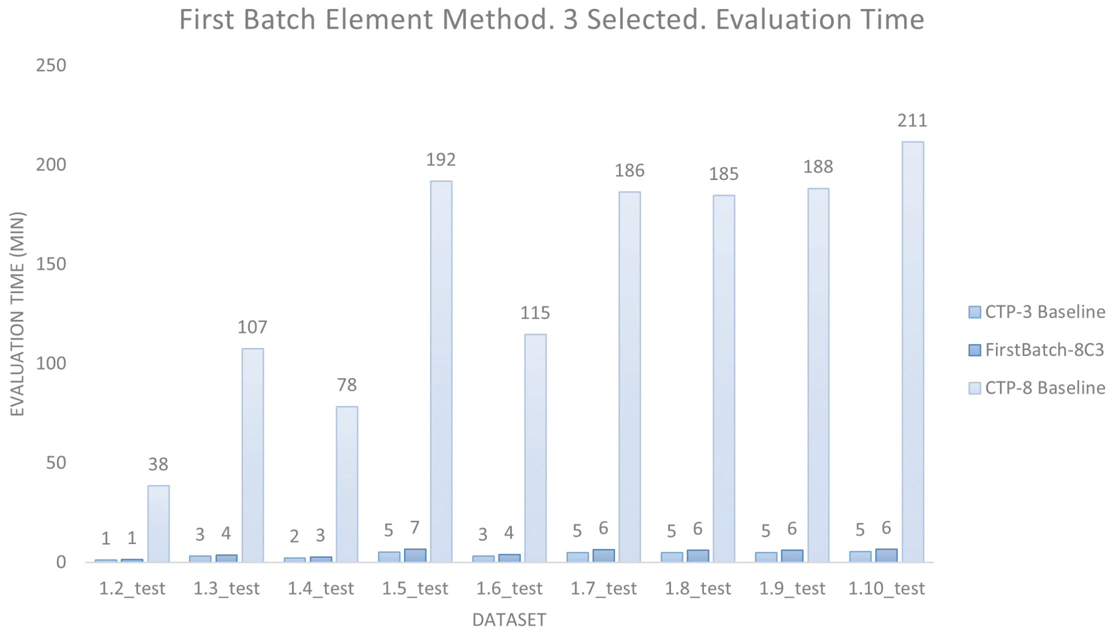


Figure 5.4: Comparison across all datasets of evaluation time when using CTPs with 3 reformulators, CTPs with 8 reformulators, and the FirstBatch method with 8 choosing 3 reformulators

subset of 3 reformulators indicates that, at the very least, FirstBatch is learning which reformulators are better than others.

Overall, the baseline of CTP-3 outperforms FirstBatch in every scenario, except for when tuning hyperparameters on *1.9_test*, where FirstBatch-8C3 achieves a slightly higher accuracy on *1.3_test*. This is technically a success for the model, but taking into account how much better the baseline performed in every other scenario, we conclude that the FirstBatch method has failed to improve upon the baseline, and that we can expect it to often perform significantly worse. However, this does not completely invalidate the usefulness of the model in all scenarios, since as we have already noted, we can expect it to perform better in cases where similar examples have all been batched together. Further improvements upon the architecture of the selection module itself could also help lessen the gap in accuracy between the baseline and FirstBatch.

Comparing between FirstBatch-8C3 and FirstBatch-8C2, FirstBatch-8C3 achieves significantly higher accuracy in every single case. This is to be expected though, as including more reformulators during evaluation will never bring down the accuracy of the model. More interesting to note is that the discrepancy in performance between CTPs and FirstBatch is far larger when 2 reformulators are used than when 3

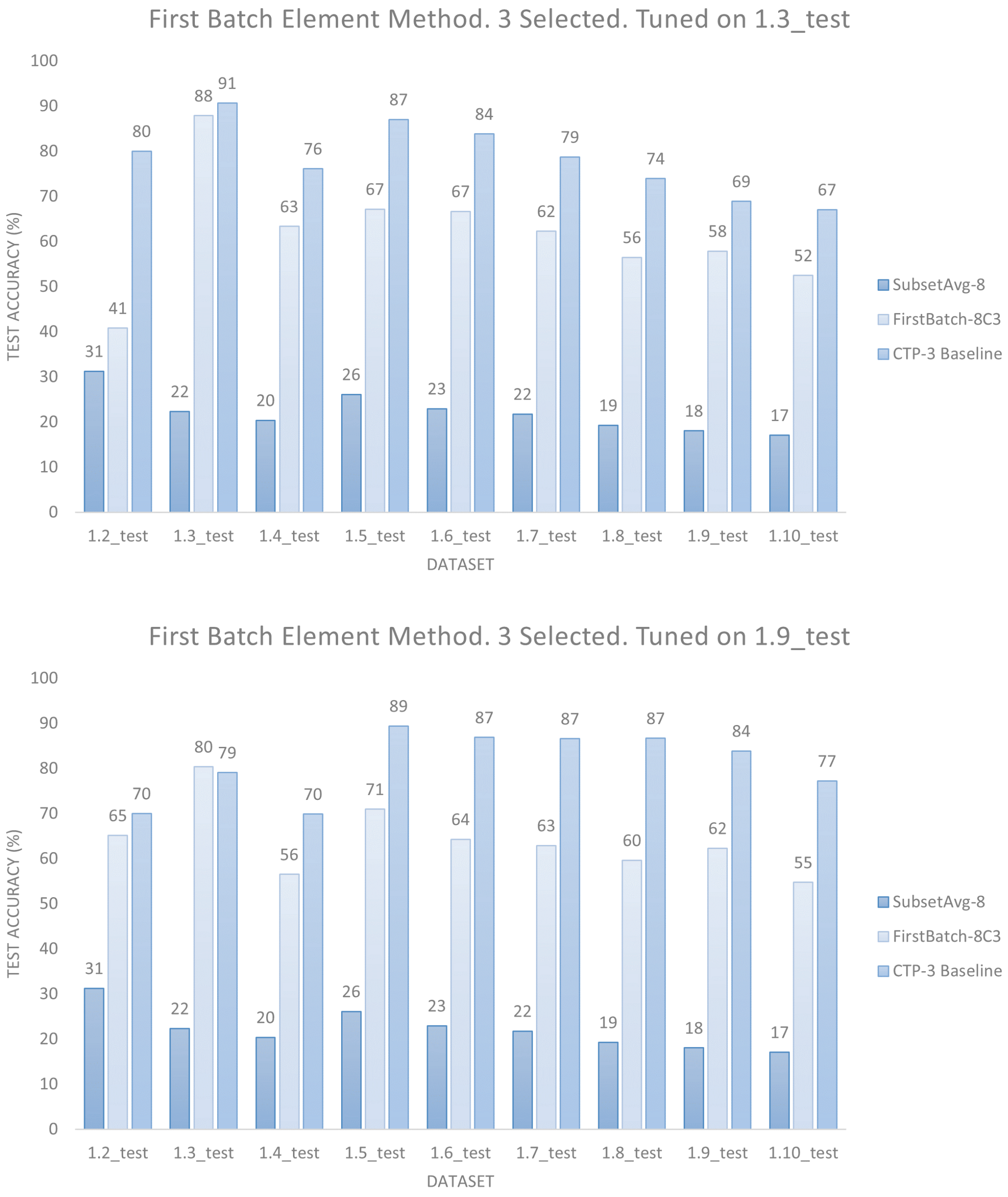
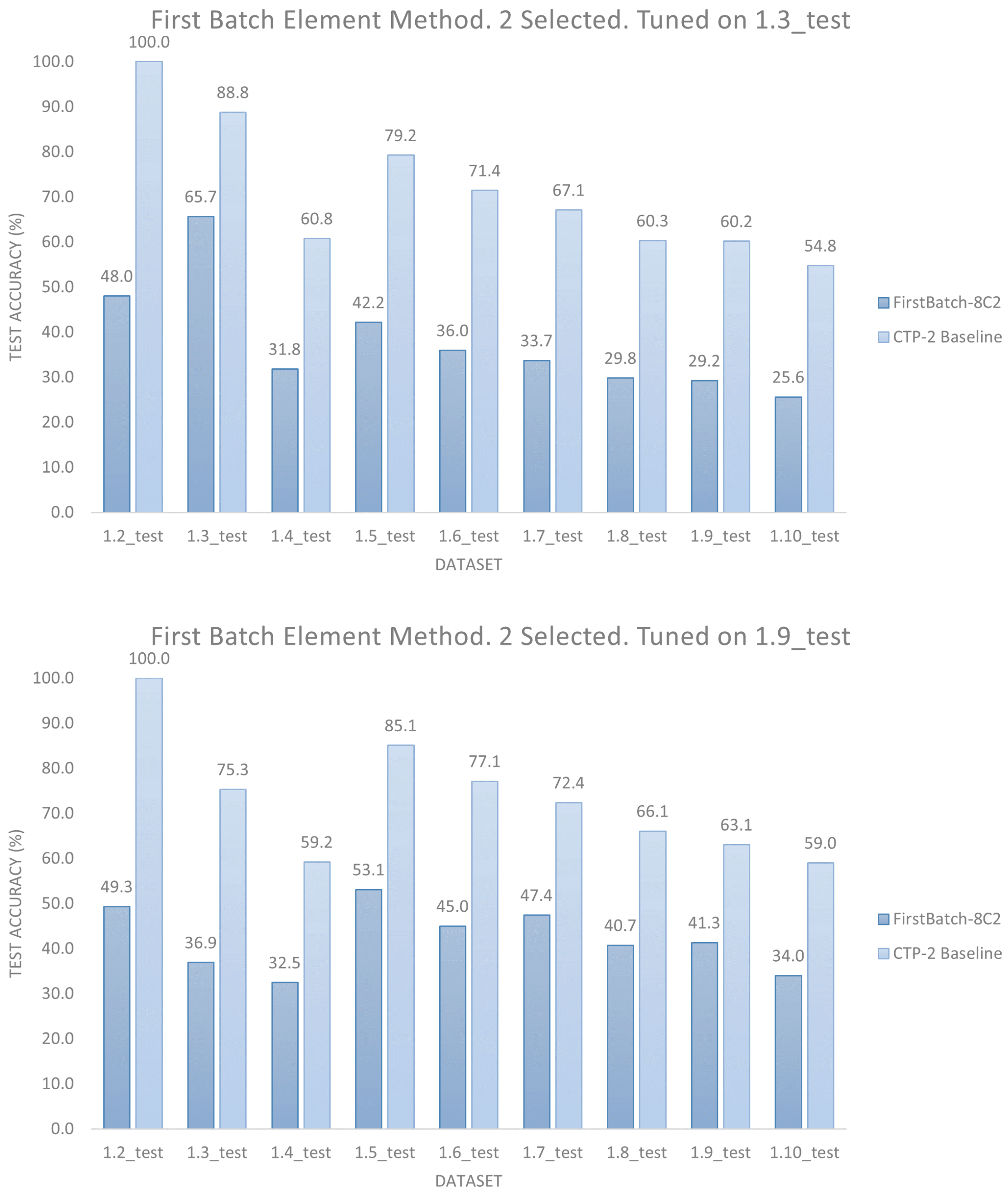


Figure 5.5: Comparison across all datasets of accuracy achieved using random subsets of size 3 of 8 reformulators, the FirstBatch method with 8 choosing 3 reformulators, and CTPs with 3 reformulators. Hyperparameters tuned on *1.3_test* and *1.9_test*


First Batch Element Method. 2 Selected. Tuned on 1.3_test

First Batch Element Method. 2 Selected. Tuned on 1.9_test

Figure 5.6: Comparison across all datasets of accuracy achieved using the FirstBatch method with 8 choosing 2 reformulators, and CTPs with 2 reformulators. Hyperparameters tuned on *1.3_test* and *1.9_test*

reformulators are used. For example, when considering evaluating on *1.9_test* after tuning hyperparameters on *1.3_test*, FirstBatch-8C3 gives 11% less accuracy (a ratio of 0.84:1) than the baseline, whereas FirstBatch-8C2 gives 31% less accuracy (a ratio of 0.49:1). This seems to imply that the rate at which the selection task becomes harder as fewer reformulators are selected is greater than the rate at which CTP performance drops from using fewer reformulators. However, we conjecture that for higher values of n and k, this performance gap will shrink and may no longer even present itself.

As a final observation, we see that overall, the performance of FirstBatch models appears less sensitive to which evaluation set is used than that of CTP models. This can be interpreted as both a bad and a good phenomenon. CTPs perform better for more complex test datasets when they are tuned on a more complex evaluation dataset, since they learn rules that will be more useful when applied to more difficult tasks. FirstBatch not demonstrating this to the same extent could indicate that it is not learning to prefer such rules, even when tuned on the datasets that require them. However, it could also be a good sign that FirstBatch is generalizing more, as its performance does not deviate much when tuned on different evaluation datasets.

Chapter 6

Tensor Operations Solution

In this chapter, we describe in detail the final implementation of RL-CTPs that we attempted. This implementation is the culmination of all the work discussed in Chapter 5. We begin by providing a technical description of how the implementation was done, outlining where in the CTP code the method was integrated, how the selection module was implemented, our solution for creating subsets of the batch to be operated on by reformulators, and how rewards were collected for updating the policy estimator. We then briefly describe how the model was optimized during the development process and during evaluation. Next, we present all the results pertinent to the model, comparing it to the baseline of CTPs while considering different values of n and k: the number of reformulators trained and number used during evaluation respectively. We also provide a test of statistical significance for the most promising results and show the wall-clock speedup that the model exhibits. We conclude the chapter by discussing aspects of the model that are still lacking and how they could be improved upon.

6.1 Technicalities of Solution

We provide a technical description of how RL-CTPs were implemented in code, using our final implementation. We refer to such models as TensorOp-nCk, in a similar manner to how FirstBatch models were expressed. While the underlying principles of how the code works are the same as the theoretical description, we have to adapt our language slightly to give a rigorous description of the implementation. In particular, all representations of predicates and constants are stored in PyTorch *tensors*, which are manipulated by the neural networks that represent reformulators. Furthermore, instead of having two functions `and` and `or` that define the expansion of CTPs upon a goal, the code has two methods: `depth_r_score` and `depth_r_forward`. Each of these encompasses the logic of both the `and` and `or` functions, with the former being used specifically for ground sub-goals, and the latter for sub-goals containing variables. They are recursively applied up to the reasoning depth, at which point the model calls to an entirely separate module to unify the sub-goals with the knowledge base.

The `depth_r_score` method begins by checking if it has reached the reasoning depth. If it has not, it then iterates through the reformulators. Each reformulator is applied to all the sub-goals in the batch, generating a batch of lists of new sub-goals. For each list of sub-goals in the batch, proof scores are recursively calculated, with the minimum scores from each list being stored as the proof scores for the sub-goals that were expanded upon. The model maximizes these scores across all reformulators. They are then returned as the proof scores for each sub-goal in the batch originally passed to the `depth_r_score` method. The `depth_r_forward` method operates in the exact same manner, but also handles substitutions by manipulating the positions of variables in the sub-goals.

6.1.1 Points of Integration

The first place where new code was needed was in the main training routine of CTPs. Instead of just training the reformulators and then evaluating, the reformulators are trained for a given number of epochs, then the selection module is activated and training continues for another given number of epochs. Training is done on the same dataset and with the same batch size. During this second round of training, the reformulators and predicate/constant embeddings are no longer updated. Evaluation happens after this round of training is complete, with the selection module still active.

Next, logic was added to the CTP model to use the selection module if it is active, and to otherwise proceed with the normal CTP operations. We describe only how the `depth_r_score` method needs modifications, as the changes required to `depth_r_forward` are very similar. After the depth limit check in `depth_r_score`, the model now needs to match the batch sizes of the arguments. Specifically, the batch elements of predicate/constant embeddings and lists of facts need to be copied, a fixed scaling factor number of times, to match up with their corresponding batch elements of sub-goals. Recall that this happens due to how the new m sub-goals for each batch element are arranged in the new batch that is created by a reformulator application. Thus, the scaling factor is calculated dynamically, to allow for the model to work for different values of m.

Following this scaling, the batch of sub-goals is passed through to the selection module to get a list of reformulators to use for each sub-goal. This is returned as a batch of lists of indices, where i being included in the list of the jth element of the batch means that reformulator i should be applied to batch sub-goal j. A sub-goal is represented by concatenating the representations of the predicate and two constants. If either of the constants are variables instead, they are represented by the zero tensor when passed to the selection module. Also returned by the selection module is a count of how many times each reformulator was selected. The method then continues by iterating over each reformulator.

A reformulator is skipped if the count of the number of times it was selected is zero. Otherwise, new tensors are constructed for the batches of sub-goals, predicate/constant embeddings, and lists of facts. The elements of these tensors are selected from the original tensors of the arguments passed to the `depth_r_score` method, using the reformulator lists returned by the selection module. This means we now have tensors containing batch elements of each of the arguments of the method, but only the batch elements that this particular reformulator ought to be applied to. The process used to achieve this is described in Section 6.1.3.

The method otherwise proceeds as normal, albeit that every instance where the method has a reference to the batch size is replaced with a reference to the count of how many times the current reformulator being considered was selected. This is because the latter is the actual number of batch elements the reformulator is now being applied to. At the end of the method, proof scores are collected to be used for calculating rewards, as described in Section 6.1.4. For maximizing the proof score across reformulators, proof scores of zero are added in for each reformulator when it was not selected for a particular batch element. The adaptions to the method are summarized in Algorithm 11.

6.1.2 Selection Module

The selection module is implemented by a class, an instance of which is instantiated while the rest of the CTP model is also being set up. It allows the model designer to specify the number of reformulators n, the number of reformulators to be selected k, the embedding size of the model, and the learning rate of the optimizer. These parameters are used to initialize the policy estimator, which contains a single hidden layer of 30 neurons, a ReLU activation function, and a log_softmax function after the output layer. Two main methods control its operation: `get_actions` and `apply_reward`.

The `get_actions` method is given a batch of states as an argument, which is passed to the policy estimator, the output of which is exponentiated to get a batch of probability distributions over the reformulators. This is then moved to the CPU

Algorithm 10: RL-CTP selection module

S is a batch of states, A is a batch of actions, and R a batch of rewards. π denotes the policy estimator and k is the number of reformulators the module ought to select. *sample_with_replacement*(S, p, k) draws k samples from S with replacement, using the probability distribution p.

```
def get_actions(S)
    $P := e^{\pi(S)}$                    /* A batch of probability distributions */
    selected_reformulators = []
    reformulator_counts = []

    for each probability distribution $p \in P$ do
        indices = []
        if n_positive_entries(p) $< k$ then
            indices = indices of the positive entries in $p$
        else
            indices = sample_with_replacement($\{0, ..., n-1\}, p, k$)
        end

        selected_reformulators.append(indices)
        update reformulator_counts using indices
    end

    return selected_reformulators, reformulator_counts
end

def apply_reward(S, A, R)
    $L$ := get_loss($S, A, R$)
    optimizer.apply_loss($L$, retain_graph = True)
end
```

so that the batch may be iterated over and have its distributions sampled from. For each probability distribution in the batch, k samples are drawn without replacement, creating a list of reformulator indices to use for each state in the batch. If the number of positive entries in a distribution is some $k' < k$, then all of the k' reformulator indices are deterministically chosen. At the same time, the method constructs a count of the number of times each reformulator was selected. This and the list of reformulators chosen for each batch element are returned from the method.

The `apply_reward` method is used to update the policy estimator with the reward signal received from taking the actions. It is passed a batch of states (sub-goal representations), a batch of actions (reformulator indices), and a batch of rewards (calculated from proof scores). These are used to calculate the loss, as described in Section 4.2.3. The loss is then applied to update the model, using the PyTorch Adam optimizer. When backpropagating the loss, we had to set the *retain_graph* flag[1] to *True*. Normally, once a tensor is used to calculate loss, the computation graph used to compute the loss will be freed. However, this cannot be done in our case, because the reward tensors are being taken from the proof scores, some of which will be used again for calculating loss further up in the model. A summary of the module is provided in Algorithm 10.

6.1.3 Tensor Masking

We now describe how subsets of batch elements were formed into new tensors, so that the selected reformulators could be applied to them. This was attempted in our Iterative method, as described in Section 5.2, by iterating over the elements of the batch. For this implementation, to avoid the slow-down that came from iterating over tensors stored on the GPU, we used PyTorch tensor operations.

First, the reinforce module is used to get a list of reformulators to use for each batch element, represented by a tensor of the shape $[B, k]$, where B is the number of elements in the batch. Each number in the tensor is a reformulator index. Note that sub-tensors along dimension 1 may contain fewer than k elements if the policy estimator produced fewer than k positive entries in the corresponding probability distributions. Then, for each reformulator i, this tensor is used to construct a *selection mask*. This is best shown by example; consider the following tensor returned by the selection module:

---

[1] https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$$

This represents that for the first batch element, reformulators 0 and 1 were selected, and that for the second batch element, reformulators 2 and 1 were selected. Let us assume that we are currently constructing new batches for reformulator $i := 1$ to be applied to. We then compare A to the index of the current reformulator: $A == 1$, giving a new tensor consisting of True/False values. In this case, it produces:

$$(A == i) = \begin{bmatrix} \text{False} & \text{True} \\ \text{False} & \text{True} \end{bmatrix}$$

We then calculate the maximum values across dimension 1 of the tensor (shown above as left to right), understanding that *True* is assigned a value of 1 and *False* a value of 0 in Python. This maximization represents including a reformulator if it was listed at all. It yields a new tensor:

$$\text{max}_{\text{dim=1}}(A == i) = \begin{bmatrix} \text{True} & \text{True} \end{bmatrix}$$

This final tensor, the selection mask, specifies whether or not reformulator i should be used for each element of the batch. In this particular case, it states that reformulator 1 should be used for both batch elements. PyTorch allows such tensors to be used for constructing a new tensor from another, by using indexing. The way in which the new batch is computed for reformulator i to be applied to is thus:

$$\text{new\_batch} := \text{batch}[\text{max}_{\text{dim=1}}(A == i)]$$

6.1.4 Collecting Rewards

At the end of the application of a reformulator i, rewards need to be collected and applied to the selection module through the `apply_reward` method. The selected batch of states (sub-goals) is constructed by applying the selection mask to the full batch of states. The batch of actions is constructed by initializing a tensor consisting only of the number i, repeated the number of times for which the reformulator was chosen. The batch of rewards is exactly the proof scores that arose from the application of the reformulator to each element in the selected batch. The full summarized `depth_r_score` method can be seen in Algorithm 11.

Since each reformulator is applied in sequence, the selection module will be trained on batches that consist only of a single action applied to different sub-goals. This

Algorithm 11: TensorOp RL-CTPs

S is a batch of sub-goals, F is a batch of fact embeddings, and C is a batch of constant embeddings. *copy(F, m)* produces a sequence of m copies of F.

```
def depth_r_score(S, F, C)
    if $|S| > |F|$ then
                                    /* Ensure matching batch elements */
        $m := \frac{|S|}{|F|}$
        $F := copy(F, m)$
        $C := copy(C, m)$
    end

    $(A, \text{reformulator\_counts}) := \text{selection\_module.get\_actions}(S)$
     /* Selected reformulators for each sub-goal in the batch */

    for each reformulator $r_i$ do
        $\text{mask} := \max_{\text{dim}=1}(A == i)$
        $S_i, F_i, C_i := S[\text{mask}], F[\text{mask}], C[\text{mask}]$        /* Include the batch
         elements for which this reformulator was selected */

        $S_{\text{new}} := r_i(S_i)$
        Then carry on with new sub-goals $S_{\text{new}}$
        ...

        $R_i :=$ proof scores from $S_{\text{new}}$
        $A_i := [i].\text{repeat}(\text{reformulator\_counts}[i])$
        $\text{selection\_module.apply\_reward}(S_i, A_i, R_i)$
    end
end
```

is different from the standard implementation of REINFORCE, where the batch of actions is determined from how the model ended up choosing them in an episode and will likely contain several different actions. However, in theory, the way in which we implemented it should still work, as REINFORCE still trains even if the model happens to select the same action many times in a row. Our one concern was that the model would tend to prefer reformulators with lower indices, as they are always the ones that apply rewards first. However, we hypothesize that this effect is too small to make any kind of significant difference and note that we did not see such preferences manifesting themselves during experiments. Therefore, we chose not to address this with a solution such as collecting the rewards from all reformulators and then scrambling the order before applying them, as doing so would add needless computation time.

6.2 Model Optimization

The selection module was implemented and optimized initially during development. This was done to avoid having to do a hyperparameter grid search on too many different parameters, as there are exponentially more configurations to try as the number of parameters increases. Thus, the only hyperparameters of RL-CTPs that we allowed to vary were the learning rate, the number of reformulators used, the number of reformulators selected, and the seed. This meant that the rest of the selection module architecture needed to be fixed.

During the development process, we started with 16 neurons in the hidden layer of the policy estimator, increasing it to 30 once we realized that 16 may not provide enough model capacity to solve the task. We conjecture that the optimal number may be even higher, but more computing power is needed to test and optimize this. We also switched from using a sigmoid activation function to ReLU, since ReLU appears to be more prevalent in similar works and is also known to be faster (Si et al., 2018). Finally, we started by training the selection module over 10 epochs, but lowered it to 3 after we noticed that the model performance increased initially over the first few epochs, but then dropped significantly as the model became overconfident in its predictions, often only choosing a single reformulator despite it being allowed to choose several.

We did initial evaluations on larger ranges of learning rates, from 0.01 to 0.2, narrowing it down to 2 learning rates each for our chosen classes of RL-CTP models: those trained with 5 reformulators and those trained with 8. These learning rates are

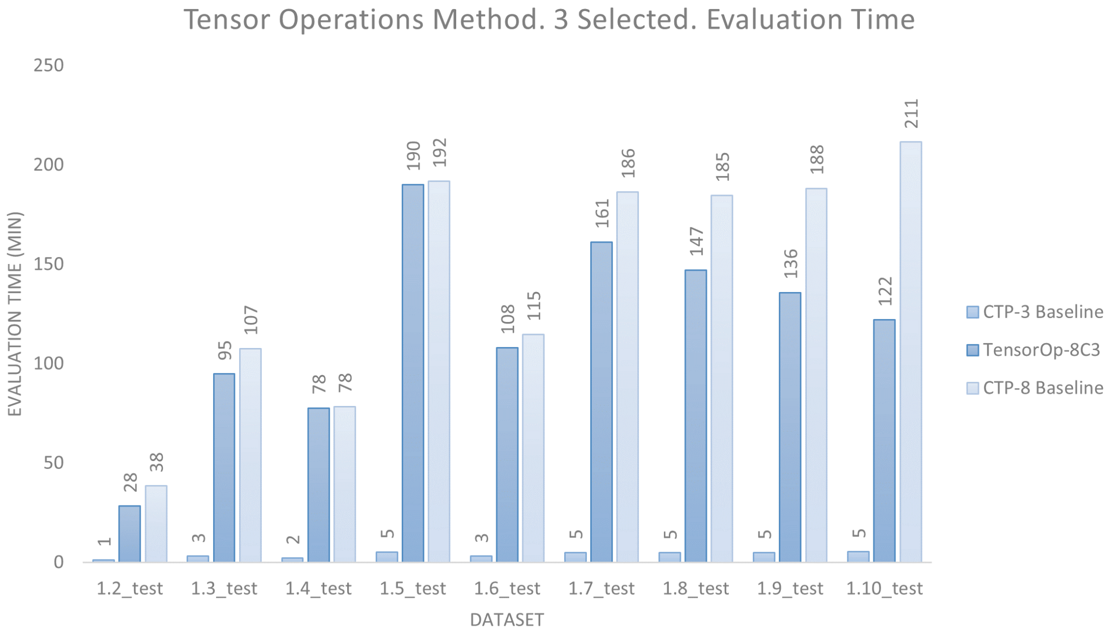


Figure 6.1: Comparison across all datasets of evaluation time when using CTPs with 3 reformulators, CTPs with 8 reformulators, and the TensorOp method with 8 choosing 3 reformulators

$\{0.001, 0.01\}$ and $\{0.005, 0.01\}$ respectively. The full list of hyperparameters used for evaluating the TensorOp method can be seen in Appendix A.6. We chose to evaluate RL-CTPs with $n \in \{5, 8\}$ to measure their effectiveness when more (and individually weaker) reformulators are used and with $k \in \{2, 3\}$ to measure the effectiveness of RL-CTPs when they are allowed to expand fewer proof paths. This yielded 4 different scenarios for evaluation.

6.3 Results

6.3.1 Speedup

In Figure 6.1, we compare the evaluation times of RL-CTPs and baseline CTPs, with RL-CTPs implemented by the TensorOp method and the baseline operating on 3 and 8 reformulators respectively. As expected, the overhead caused by the copying, masking, and other operations needed in the TensorOp method led to it taking significantly longer to evaluate than CTP-3. However, the overhead was low enough for TensorOp-8C3 to take less time to evaluate than CTP-8 across all datasets. The effect becomes more pronounced as the complexity of the dataset increases, since the number of facts in the knowledge base to unify the sub-goals with increases, which has a significant effect on the computational complexity of the model.

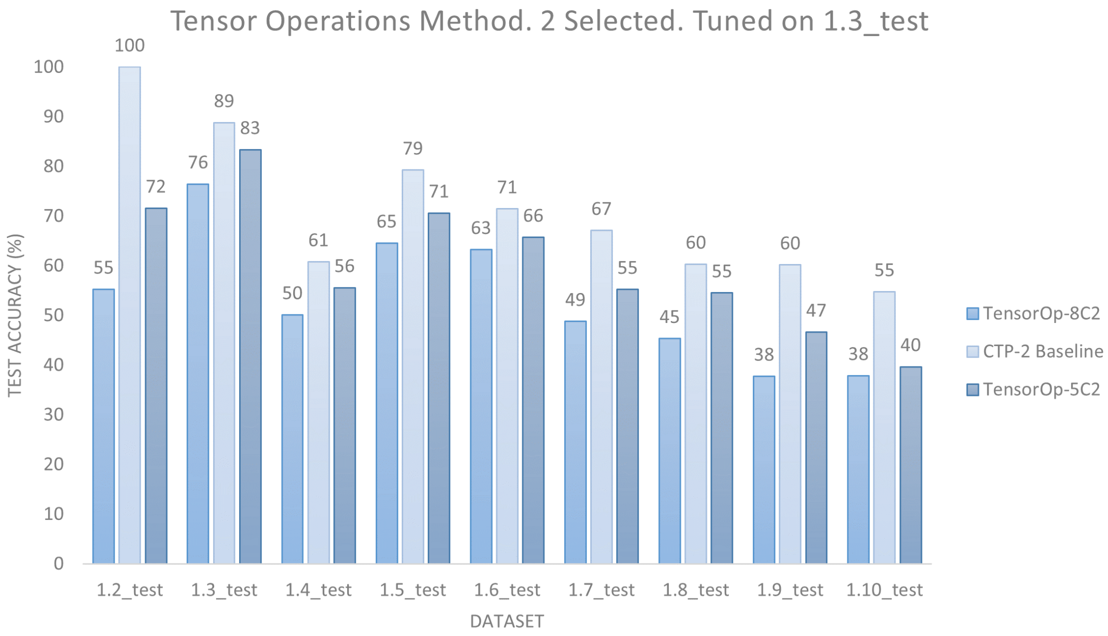
As the dataset complexity continues to increase. the overhead will become more negligible, leading the evaluation time of TensorOp-nCk models to tend to those of CTP-k models. The increasing gap between the evaluation time of TensorOp-8C3 and CTP-8 in Figure 6.1 is a clear visual illustration of this trend. Datasets always taking longer to evaluate on than others is explained by the size of the datasets. For example, despite *1.6_test* being a more complex dataset than *1.5_test*, it only contains 104 tasks, compared to the 184 of *1.5_test*.

6.3.2 Comparison to Baseline

We evaluate the TensorOp method when 5 and 8 reformulators have been trained, with 2 and 3 reformulators being chosen. Recalling that TensorOp-nCk denotes a TensorOp model trained with n reformulators, where k are chosen by the selection module, we demonstrate the performance of TensorOp8C2 and TensorOp5C2 in Figure 6.2. We also show the performance of TensorOp8C3 and TensorOp5C3 in Figure 6.3. These models are evaluated by comparing them to their respective baselines: CTP-2 and CTP-3. This yields 4 different scenarios in which RL-CTPs are evaluated: 8 choosing 3, 8 choosing 2, 5 choosing 3, and 5 choosing 2 reformulators.

Let us first consider the results in Figure 6.3, which pertain to RL-CTPs that choose 3 reformulators. We see that, when tuned on *1.3_test*, TensorOp5C3 outperforms the baseline on every dataset except *1.2_test*. These results demonstrate that, at the very least, there are some scenarios in which RL-CTP models achieve higher accuracies than computationally equivalent CTP models. We calculate the statistical significance of these particular results in Section 6.3.3. When tuning the same models on *1.9_test*, RL-CTPs only outperform the baseline on the simpler datasets. We also note that when tuned on *1.3_test*, TensorOp8C3 outperforms the baseline on one of the test datasets, but performs comparatively poorly on the rest. Out of the 4 scenarios in which we evaluate RL-CTPs, they only consistently outperform the baseline in one.

As seen in Section 5.4, RL-CTPs implemented by the FirstBatch method perform significantly worse when only 2 reformulators are chosen instead of 3. The same effect is seen for TensorOp RL-CTPs in Figure 6.2: TensorOp models that only choose 2 reformulators are outperformed by the baseline across every dataset. We hypothesize that, for CLUTRR, the task of learning which two reformulators are the most promising is one that is just too difficult for the model to find a solution to.



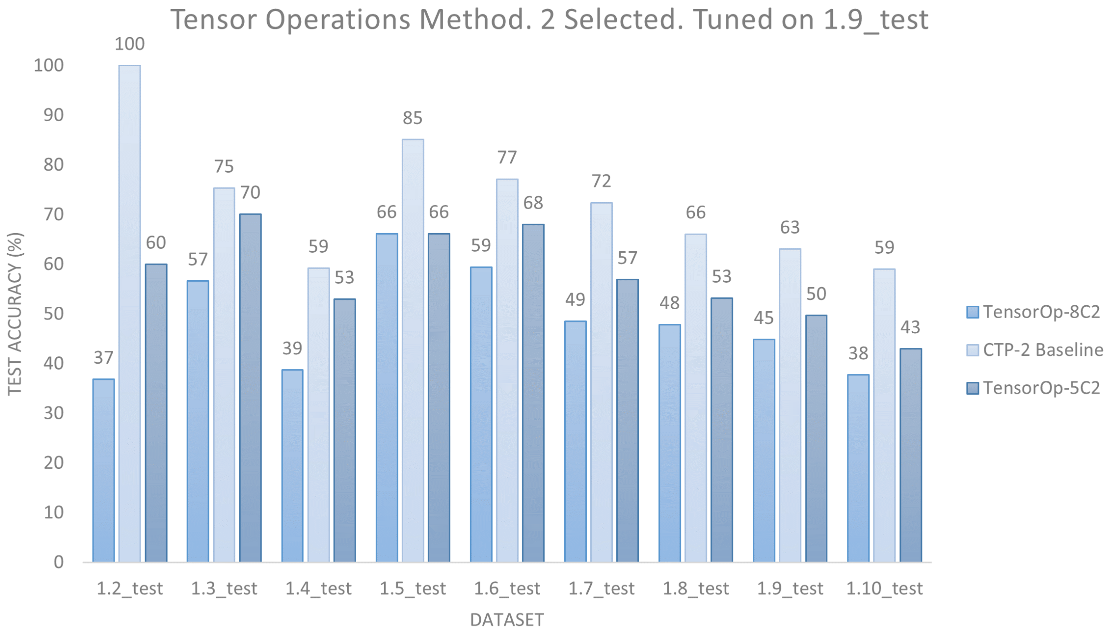


Figure 6.2: Comparison across all datasets of accuracy achieved using TensorOp with 8 choosing 2 reformulators, TensorOp with 5 choosing 2 reformulators, and CTPs with 2 reformulators. Hyperparameters tuned on *1.3_test* and *1.9_test*

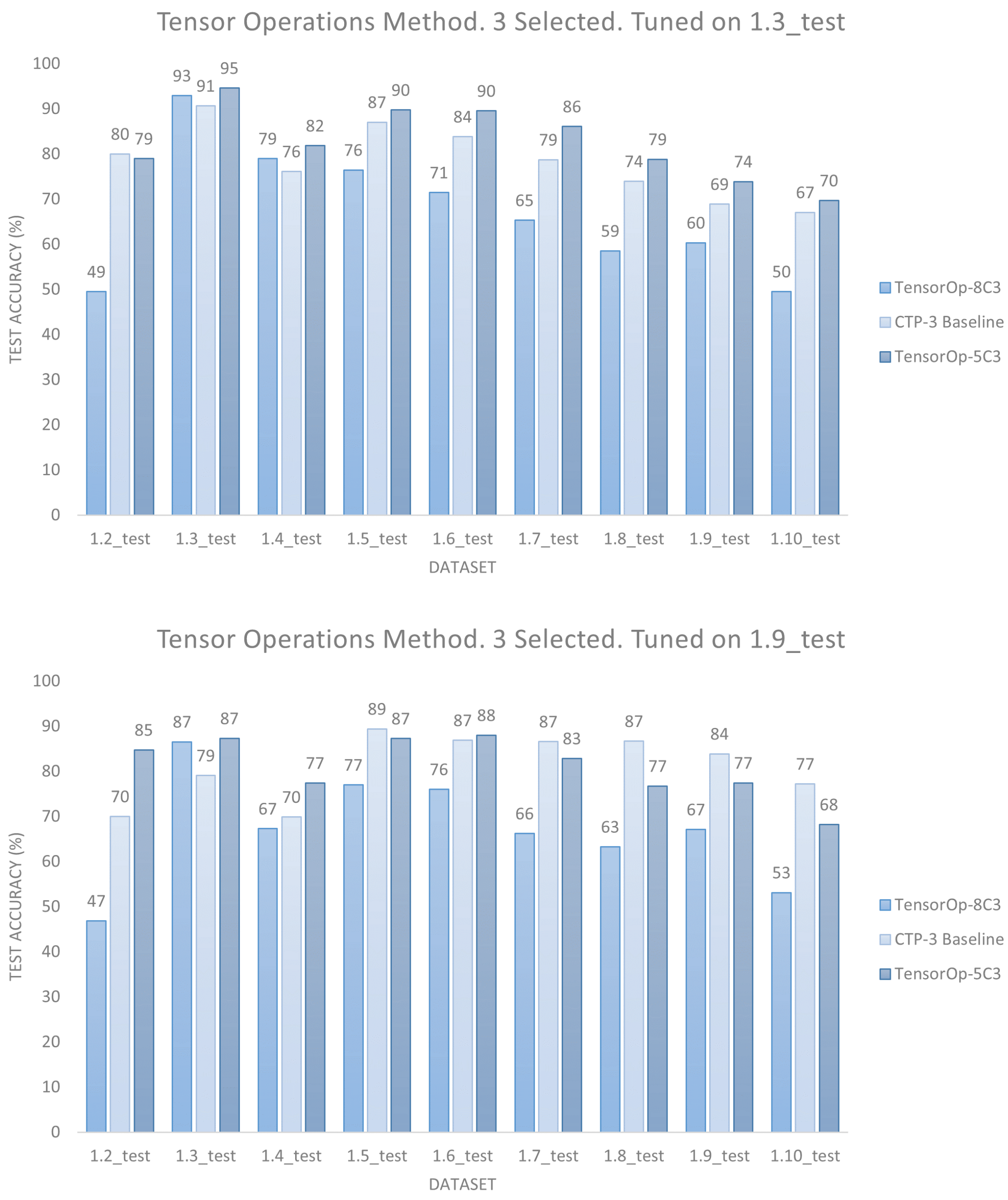


Figure 6.3: Comparison across all datasets of accuracy achieved using TensorOp with 8 choosing 3 reformulators, TensorOp with 5 choosing 3 reformulators, and CTPs with 3 reformulators. Hyperparameters tuned on *1.3_test* and *1.9_test*

As shown in Section 5.3.2: as the number of reformulators increases, individual reformulators become weaker. Thus, the task of choosing the optimal reformulators for expansion becomes more difficult as the number of reformulators increases. This means that, all else being constant, the accuracy of RL-CTP models will drop as more reformulators are trained. This effect is offset by the increasing expressivity of the model as more reformulators are used. Hence, as expected, RL-CTPs implemented by the TensorOp method consistently perform worse when more reformulators are trained. This is seen is both Figure 6.2 and Figure 6.3, where TensorOp5Ck models outperform TensorOp8Ck models in every scenario and across every dataset.

Finally, we note that tuning hyperparameters on *1.9_test* instead of *1.3_test* causes baseline CTP performance to decrease for the simpler datasets but increase for the more complex datasets. For RL-CTPs however, this effect appears far less pronounced, with the accuracy of TensorOp5C3 models even dropping slightly for the more complex test datasets, when tuning on *1.9_test* instead of *1.3_test*. This indicates that RL-CTPs are not learning the reasoning patterns needed for the more complex datasets, even when tuned on such a dataset. It is also possible that RL-CTPs could be overfitting to the evaluation set when tuned on *1.9_test*. However, since the accuracy of RL-CTPs on *1.9_test* does not even increase that much when tuning on the dataset, we find the former explanation to be more likely.

6.3.3 Statistical Significance

We present a test of statistical significance for the results obtained in the one scenario where RL-CTPs outperformed CTPs. The test compares CTPs and RL-CTPs, where RL-CTPs are implemented by TensorOp5C3, CTPs use 3 reformulators, and hyperparameters are tuned on *1.3_test*. A separate one-tailed unpaired t-test (Mann, 2007) is performed between the two methods for each dataset. The respective means, standard deviations, and resulting p-values are shown in Figure 6.4. Each population has a sample size of 5, since that is the number of tests that were run for each method, as specified in Algorithm 5. The null hypothesis of each test is that CTPs and RL-CTPs achieve the same accuracy, and the alternative hypothesis is that RL-CTPs achieve a higher accuracy.

Adopting a standard significance level of 0.05 (Fisher, 1992), we reject the null hypothesis on the datasets of *1.4_test*, *1.6_test*, and *1.7_test*. Significant results were not achieved for the other datasets, as despite RL-CTPs outperforming CTPs on every dataset except *1.2_test*, relatively high variance was experienced in the results. This is attributed to the limited computational resources we had access to for this thesis,

	CTP		**RL-CTP**		
	μ ±	σ	μ ±	σ	**One-Tailed P-Value**
1.2_test	80.00	24.49	78.95	25.84	0.525
1.3_test	90.65	7.24	94.58	1.09	0.147
1.4_test	76.10	4.97	81.82	4.57	0.048
1.5_test	87.03	3.31	89.73	5.34	0.185
1.6_test	83.81	3.41	89.52	4.82	0.033
1.7_test	78.71	1.82	86.06	5.01	0.014
1.8_test	73.93	5.20	78.82	6.50	0.114
1.9_test	68.87	7.58	73.87	5.94	0.140
1.10_test	67.05	4.96	69.67	6.43	0.246

Figure 6.4: One-tailed unpaired t-test between the baseline of CTPs with 3 reformulators and RL-CTPs implemented by TensorOp with 5 choosing 3 reformulators. Hyperparameters tuned on *1.3_test*

as the higher the number of times the evaluation procedure is run, the more confident the results become. The variance experienced on *1.2_test* is an outlier among the datasets; all methods that operated on the dataset demonstrated a high variance in their results.

As a final point, we note that RL-CTPs do not appear to exhibit higher instability in their accuracies than CTPs, with the models showing comparable levels of variance. This is a good sign for RL-CTPs, as models that have higher variance in their performance are riskier to use, even if they achieve higher accuracies on average.

6.4 Further Improvements

In this section, we discuss some of the issues with RL-CTPs, their possible solutions, and other ways one could improve upon the model. These improvements were considered but ultimately not implemented due to them being beyond the scope of this thesis. We leave them for future work.

6.4.1 Negative Examples

As already noted in Section 4.2.4, we train the selection module of RL-CTPs to maximize the proof score across all tasks, not just positive tasks. This is problematic, since the target proof score of a negative task is zero. This did not appear to cause significant issues with model performance, but ideally this problem should be dealt with anyway. We present two possible solutions. The first is to only use positive tasks during the training of the selection module. The second is to assign the negative of the proof score as a reward for the model, whenever it is trained on a negative task. However, the latter suggestion leads to some strange concepts arising around the behaviour of the selection module. We are essentially telling the model that when the goal does not hold true, it should try select the proof path that does the worst job at proving it, rather than trusting that there are no proof paths that will yield a satisfactory proof score.

As such, we suggest the first solution as an improvement to be made in future work, which can be implemented by doing pre-processing on the dataset to remove all negative examples before the selection module is trained.

6.4.2 Variables

The policy estimator in RL-CTPs needs tensors to operate upon, so there remains the open question of how to deal with variables in a sub-goal. The approach we took was to fix the representation of any variable to be the tensor consisting of all zeroes. However, there are perhaps more sophisticated ways of addressing this problem. The first is to condition only upon the predicate of the sub-goal, eliminating the need to consider the variables and constants at all. However, reformulators already operate on only the predicate, meaning that in theory, they should already capture all the different ways the predicate could be expanded upon. Without the extra context of the constants and variables, the selection module cannot possibly know which proof path is the most promising, and simply has to learn which reformulators are generally better than others. Thus, this is a poor solution.

Another option is to add an extra 4 inputs to the policy estimator, which can be used for one-hot encodings, to tell the selection model whether it has received a sub-goal of the form $p(c_1, c_2)$, $p(X, c_2)$, $p(c_1, X)$, or $p(X, Y)$. Alternatively, there could be a separate selection module used for each of these cases. Finally, one could also try learn a representation for variables, instead of fixing it to be the zero tensor. This representation could be optimized jointly with the rest of the selection module.

6.4.3 Entropy Regularization

RL-CTPs have to be trained for very few epochs to prevent them from becoming overconfident as they train. To solve this, we can encourage the model to choose diverse distributions of reformulators during training using entropy regularization, a method proposed by Mnih et al. (2016). With entropy regularization, the diversity of the model predictions is used when calculating the loss. The following term is added to the loss function, where β is a hyperparameter used to scale the regularization:

$$H(X) = -\beta \sum_{x} \pi(x) \log(\pi(x))$$

This technique has been used by Neu et al. (2017); Nachum et al. (2017); Xiong et al. (2017); Das et al. (2018); Haarnoja et al. (2018) to great effect, and we suspect it to be the change to the model that, if implemented, would yield the greatest increase in performance. This is because we see that over time, even the best-performing models tend to towards choosing specific subsets of reformulators, meaning that training needs to be stopped before they overly fixate on them. With entropy regularization, the model could be trained for longer and across more diverse selections, learning patterns that it would otherwise not be able to.

6.4.4 REINFORCE with Baseline

One of the issues with policy gradient methods is the high variance they experience due to the diverse rewards earned during exploration. A common way to address this is to use *Policy Gradient with Baseline* (Evans & Swartz, 2000; Fishman, 2013; Hammersley, 2013). A baseline scoring function is introduced into the model, to be used as a proxy for the expected actual return. Before the reward is applied, the baseline is subtracted from it. The value function itself can be used for the baseline, or a *learned baseline* could be introduced, where another separate model is used to approximate the expected return.

6.4.5 Training Modules in Parallel

In the initial stages of developing RL-CTPs, we trained the reformulators and the selection module in parallel. However, we abandoned this training routine due to the poor performance we experienced: the selection module would choose only a few reformulators that continued to improve as they were trained, leading to the selection module choosing them more frequently. This feedback loop led to only a small subset of reformulators being properly trained, with the selection module always choosing them. We have already noted that this seems to defeat the point of using RL-CTPs. However, there is a potential use case for this.

One of the current open problems in the CTP framework is how to induce the rule structures. The structures of the reformulators define the types of rules they can capture, so this prior knowledge about the data is needed to learn the rules. With this in mind, we propose the following process for learning rule structures. Initialize many different reformulators, several for each type of rule structure that one believes could hold in the knowledge base. Use a selection module similar to the FirstBatch implementation, but not conditioned upon the sub-goals, so that it purely tries to learn which reformulator subsets are the best. Include entropy regularization in the model and increase the value of the hyperparameter that controls the regularization for the early training epochs, to encourage even more diversity during initial training. Fix some value of k for the selection module which is as high as the evaluation time constraints allow for. Then, the model should tend towards learning which reformulators, and thus which rule structures, best capture the rules in the knowledge base.

The problem of the feedback loop between the selection module and reformulator performance can be solved by a suitable tuning of the entropy regularization, and the issue of some reformulators being chosen more than others during training becomes a feature of the model. This is because reformulators that better capture the rules in the knowledge base will yield higher rewards over time and be preferred by the selection module.

6.4.6 Conditioning Upon More

In the current model of RL-CTPs, we condition the policy estimator upon only the current sub-goal being considered for expansion. However, the model may benefit from having more information about the current state of the proof path, such as how many reasoning steps are left before unification, the main goal to be proved, the reformulators used for expansion to reach this point in the proof path, and the other existing sub-goals. Providing the selection module with this extra context could allow it to make predictions about the success of different proof paths that it could not before. Das et al. (2018) parameterize their policy estimator in MINERVA with the entire history of reasoning steps taken. They use a long short-term memory network (Hochreiter & Schmidhuber, 1997) to implement this. A similar approach could be used to allow RL-CTPs to condition upon the other parameters described above.

Chapter 7

Conclusion

In this thesis, we provided motivation for cases in which CTP models would be required to have a large number of reformulators and a high reasoning depth, as well as demonstrating how this leads to computational complexity concerns both theoretically and empirically. We defined a framework for RL-CTPs as an extension to CTPs, in which reinforcement learning is used to learn to select optimal reformulators for expansion during a proof. This allows the model designer to scale down the number of selected reformulators, such that the computational constraints of the use case may be met. Specifically, we defined a model architecture based on REINFORCE, a policy gradient method. We outlined our various failed attempts to implement this architecture, giving insight as to why they failed, as well as the inner workings of CTPs. Most importantly, we noted that certain subsets of reformulators perform significantly better than others, and that individual reformulators tend to become weaker as the number of reformulators used in a CTP model increases. This means that the task of selecting reformulators becomes more difficult as the number of reformulators increases.

We also gave an outline of our successful implementation of RL-CTPs, which we refer to as the TensorOp method. We evaluated the model in 4 separate scenarios, which vary in regard to the number of reformulators trained and the number of reformulators selected. In 1 of these 4 scenarios, we found that RL-CTPs outperformed CTPs on 8 out of 9 test datasets. However, due to high variance in accuracies from limited testing (as we lacked computational power), statistically significant results were only achieved for 3 of the datasets. In the remainder of the scenarios, RL-CTPs failed to outperform the baseline CTP models. The results demonstrate the usefulness of RL-CTPs over CTPs in certain situations, but also highlight their failing to be a categorical improvement upon CTPs. Overall, we identify that the framework has shown its potential, but is not especially useful in its current form.

This research opens several promising avenues for future work. The remaining problems we identified in the RL-CTP architecture provide a particularly fruitful place to start. In Section 6.4, we discussed in detail several issues we perceived in the model and outlined our proposed solutions to these problems. Specifically, we suggested how the model could handle negative examples, deal better with variables appearing in sub-goals, avoid overconfidence during training by incorporating entropy regularization into the loss function, decrease the variance by using Policy Gradient with Baseline, train the reformulators and selection module in parallel, and condition the selection module upon more data. From among these, we identify entropy regularization as the improvement that would yield the greatest immediate increase in the performance of the model.

Overall, but independent to the aims of this thesis, the greatest remaining weakness of CTPs is their inability to induce the structures of the rules that they learn. The rule structures have to be specified before training and are built into the structures of the reformulators. The need for this prior knowledge greatly inhibits the application of CTPs to datasets where the structure of rules is not known up front. To address this, CTPs should be able to induce the structures of the rules during training. Besides providing an outline in Section 6.4.5 as to how one might approach this using an implementation of RL-CTPs, we leave this for future work.

Appendix A

Model Hyperparameters

In this appendix, we provide the hyperparameters used for each of our evaluations.

A.1 Fixed CTP Hyperparameters

Name	**Value**
batch-size	16
embedding-size	50
epochs	20
evaluate-every	100
init	random
init-size	1
k-max	5
learning-rate	0.1
max-depth	2
nb-rules	512
optimizer	adagrad
ref-init	random
reformulator	attentive
scoring-type	concat
slope	1
test	*ALL TEST DATASETS*
test-batch-size	1
tnorm	min
train	data/clutrr-emnlp/data_db9b8f04/1.2,1.3,1.4_train.csv

A.2 Number of Reformulators

Name	Values
hops	1-8 reformulators, each with 2 atoms in the body. For example, 4 reformulators is denoted by: “2 2 2 2”
seed	1-30
test-max-depth	4

A.3 Reasoning Depth

Name	Values
hops	5 reformulators, each with 2 atoms in the body, denoted by: “2 2 2 2 2”
seed	1-3
test-max-depth	1-5

A.4 First Batch Element Approximation

Name	Values
hops	8 reformulators, each with 2 atoms in the body, denoted by: “2 2 2 2 2 2 2 2”
seed	1-20
test-max-depth	4
rl-actions-selected	2, 3
rl-epochs	3
rl-learning-rate	0.005, 0.01

A.5 Reformulators Subsets

Name	Values
hops	5, 8 reformulators, each with 2 atoms in the body, denoted by: “2 2 2 2 2” and “2 2 2 2 2 2 2 2” respectively
seed	1-10
test-max-depth	4
subset	Use reformulators with the following indices for evaluation: { [0 1 2], [2 3 4], [0 3 4], [1 2 4] }

A.6 RL with Tensor Operations

A.6.1 5 Reformulators

Name	Values
hops	5 reformulators, each with 2 atoms in the body, denoted by: “2 2 2 2 2”
seed	1-30
test-max-depth	4
rl-actions-selected	2, 3
rl-epochs	3
rl-learning-rate	0.001, 0.01

A.6.2 8 Reformulators

Name	Values
hops	8 reformulators, each with 2 atoms in the body, denoted by: “2 2 2 2 2 2 2 2”
seed	1-30
test-max-depth	4
rl-actions-selected	2, 3
rl-epochs	3
rl-learning-rate	0.005, 0.01

Bibliography

Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. Performance, design, and autotuning of batched gemm for gpus. In *International Conference on High Performance Computing*, pp. 21–38. Springer, 2016.

Ashish Agarwal. Static automatic batching in tensorflow. In *International Conference on Machine Learning*, pp. 92–101. PMLR, 2019.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Learning to compose neural networks for question answering. *NAACL*, 2016.

Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron Courville. Systematic generalization: what is required and can it be learned? *ICLR*, 2019.

Ioana Baldini, Stephen J Fink, and Erik Altman. Predicting gpu performance from cpu runs using machine learning. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pp. 254–261. IEEE, 2014.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, 26, 2013.

Matko Bošnjak. *On Differentiable Interpreters*. PhD thesis, UCL (University College London), 2021.

Guillaume Bouchard, Sameer Singh, and Theo Trouillon. On approximate reasoning capabilities of low-rank vector spaces. In *AAAI Spring Symposia*. AAAI Press, 2015.

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.

Roberta Calegari, Giovanni Ciatto, Enrico Denti, and Andrea Omicini. Logic-based technologies for intelligent systems: State of the art and perspectives. *Information*, 11(3):167, 2020.

Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka, and Tom M Mitchell. Toward an architecture for never-ending language learning. In *Twenty-Fourth AAAI conference on artificial intelligence*, 2010.

Rich Caruana, Steve Lawrence, and Lee Giles. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. *Advances in neural information processing systems*, pp. 402–408, 2001.

Stefano Ceri, Georg Gottlob, Letizia Tanca, et al. 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.

Kai-Wei Chang, Wen-tau Yih, Bishan Yang, and Christopher Meek. Typed tensor decomposition of knowledge bases for relation extraction. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1568–1579, 2014.

Noam Chomsky. Logical structures in language. *American Documentation (pre-1986)*, 8(4):284, 1957.

Kaleigh Clary, Emma Tosch, John Foley, and David Jensen. Let's play again: Variability of deep reinforcement learning agents in atari environments. *NeurIPS Critiquing and Correcting Trends Workshop*, 2018.

Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. *arXiv preprint arXiv:1806.08295*, 2018.

Andrew Cropper and Stephen H Muggleton. Learning higher-order logic programs through abstraction and invention. In *IJCAI*, pp. 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. *Proceedings of the 7th International Conference on Learning Representations*, 2018.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

Ivan Donadello, Luciano Serafini, and Artur D'Avila Garcez. Logic tensor networks for semantic image interpretation. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pp. 1596–1602, 2017.

Stavros Efthymiou, Jack Hidary, and Stefan Leichenauer. Tensornetwork for machine learning. *arXiv preprint arXiv:1906.06329*, 2019.

Michael Evans and Timothy Swartz. *Approximating integrals via Monte Carlo and deterministic methods*, volume 20. OUP Oxford, 2000.

Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

Ronald Aylmer Fisher. Statistical methods for research workers. In *Breakthroughs in statistics*, pp. 66–70. Springer, 1992.

George Fishman. *Monte Carlo: concepts, algorithms, and applications.* Springer Science & Business Media, 2013.

Manoel VM França, Gerson Zaverucha, and Artur S d'Avila Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine learning*, 94(1):81–104, 2014.

Hervé Gallaire and Jack Minker. Logic and data bases, symposium on logic and data bases, centre d'études et de recherches de toulouse, 1977. *Advances in Data Base Theory*, 1978.

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 *2015 AAAI Spring Symposium Series*, 2015.

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, 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 d'Avila Garcez, Dov M Gabbay, Oliver Ray, and John Woods. Abductive reasoning in neural-symbolic systems. *Topoi*, 26(1):37–49, 2007.

Nicolas Gontier, Koustuv Sinha, Siva Reddy, and Christopher Pal. Measuring systematic generalization in neural proof generation with transformers. *NeurIPS'20*, 2020.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Edward Grefenstette. Towards a formal distributional semantics: Simulating logical calculi with tensors. *SEM*, 2013.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. *Advances in neural information processing systems*, 28:1828–1836, 2015.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.

John Hammersley. *Monte carlo methods*. Springer Science & Business Media, 2013.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Steffen Hölldobler, Yvonne Kalinke, and Hans-Peter Störr. Approximating the semantics of logic programs by recurrent neural networks. *Applied Intelligence*, 11 (1):45–58, 1999.

Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2901–2910, 2017.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in neural information processing systems*, 28: 190–198, 2015.

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *International Conference on Learning Representations (ICLR)*, 2016.

Charles Kemp, Joshua B Tenenbaum, Thomas L Griffiths, Takeshi Yamada, and Naonori Ueda. Learning systems of concepts with an infinite relational model. In *AAAI*, volume 3, pp. 5, 2006.

Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pp. 2873–2882. PMLR, 2018.

Adam Lally and Paul Fodor. Natural language processing with prolog in the ibm watson system. *The Association for Logic Programming (ALP) Newsletter*, 9, 2011.

Ni Lao, Tom Mitchell, and William Cohen. Random walk inference and learning in a large scale knowledge base. In *Proceedings of the 2011 conference on empirical methods in natural language processing*, pp. 529–539, 2011.

Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joseph E Gonzalez. Train large, then compress: Rethinking model size for efficient training and inference of transformers. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, pp. 5958–5968, 2020.

Chen Liang, Jonathan Berant, Quoc Le, Kenneth D Forbus, and Ni Lao. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *the Association for Computational Linguistics annual meeting (ACL)*, 2017.

Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning entity and relation embeddings for knowledge graph completion. In *Twenty-ninth AAAI conference on artificial intelligence*, pp. 2181–2187. AAAI Press, 2015.

Prem S Mann. *Introductory statistics.* John Wiley & Sons, 2007.

Dennis Merritt. *Building expert systems in Prolog.* Springer Science & Business Media, 2012.

Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. Key-value memory networks for directly reading documents. *Proc. of EMNLP*, 2016.

Pasquale Minervini, Matko Bošnjak, Tim Rocktäschel, Sebastian Riedel, and Edward Grefenstette. Differentiable reasoning on large knowledge bases and natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 5182–5190, 2020a.

Pasquale Minervini, Sebastian Riedel, Pontus Stenetorp, Edward Grefenstette, and Tim Rocktäschel. Learning reasoning strategies in end-to-end differentiable proving. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pp. 6938–6949. PMLR, 2020b.

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

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4): 295–318, 1991.

Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13 (3):245–286, 1995.

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.

Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 2772–2782, 2017.

Sriraam Natarajan, Tushar Khot, Kristian Kersting, Bernd Gutmann, and Jude Shavlik. Gradient-based boosting for statistical relational learning: The relational dependency network case. *Machine Learning*, 86(1):25–56, 2012.

Gergely Neu, Anders Jonsson, and Vicenç Gómez. A unified view of entropy-regularized markov decision processes. *arXiv preprint arXiv:1705.07798*, 2017.

Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. Factorizing yago: scalable machine learning for linked data. In *Proceedings of the 21st international conference on World Wide Web*, pp. 271–280, 2012.

Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. Holographic embeddings of knowledge graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5: 239–266, 1990. doi: 10.1007/BF00117105.

Rajat Raina, Anand Madhavan, and Andrew Y Ng. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pp. 873–880, 2009.

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*, pp. 74–84, 2013.

Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *NIPS*, pp. 3788—-3800, 2017.

Ali Sadeghian, Mohammadreza Armandpour, Patrick Ding, and Daisy Zhe Wang. Drum: End-to-end differentiable rule mining on knowledge graphs. *Proc. of NIPS*, 2019.

Shaeke Salman and Xiuwen Liu. Overfitting mechanism and avoidance in deep neural networks. *arXiv preprint arXiv:1901.06566*, 2019.

Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. In *ITU J.*, volume 1 of *ICT Discoveries*, pp. 39–48, 2018.

Claude Sammut and Ranan B Banerji. Learning concepts by asking questions. *Machine learning: An artificial intelligence approach*, 2:167–192, 1986.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, pp. 1842–1850. PMLR, 2016.

Daniel Schlegel. Deep machine learning on gpu. *University of Heidelber-Ziti*, 12, 2015.

Dietmar Seipel, Falco Nogatz, and Salvador Abreu. Domain-specific languages in prolog for declarative expert knowledge in rules and ontologies. *Computer Languages, Systems & Structures*, 51:102–117, 2018.

Luciano Serafini and Artur d'Avila Garcez. Logic tensor networks: Deep learning and logical reasoning from data and knowledge. In *Proceedings of the 11th International Workshop on Neural-Symbolic Learning and Reasoning (NeSy'16) co-located with the Joint Multi-Conference on Human-Level Artificial Intelligence (HLAI 2016), New York City, NY, USA, July 16-17, 2016.*, 2016.

Lokendra Shastri. Neurally motivated constraints on the working memory capacity of a production system for parallel processing: Implications of. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society: July 29 to August 1, 1992, Cognitive Science Program, Indiana University, Bloomington*, volume 14, pp. 159. Psychology Press, 1992.

Yelong Shen, Jianshu Chen, Po-Sen Huang, Yuqing Guo, and Jianfeng Gao. M-walk: Learning to walk over graphs using monte carlo tree search. *Advances in Neural Information Processing Systems*, 2018.

Jiong Si, Sarah L Harris, and Evangelos Yfantis. A dynamic relu on neural network. In *2018 IEEE 13th Dallas Circuits and Systems Conference (DCAS)*, pp. 1–6. IEEE, 2018.

Koustuv Sinha, Shagun Sodhani, Jin Dong, Joelle Pineau, and William L. Hamilton. Clutrr: A diagnostic benchmark for inductive reasoning from text. *Empirical Methods of Natural Language Processing (EMNLP)*, 2019.

Paul Smolensky. On the proper treatment of connectionism. *Behavioral and brain sciences*, 11(1):1–23, 1988.

Raymond M Smullyan. *First-order logic.* Dover Publications, Inc., New York, 1995.

Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pp. 926–934. Citeseer, 2013.

Ashwin Srinivasan. The aleph manual, 2001.

Bernd Steinbach and Roman Kohut. Neural networks: a model of boolean functions. In *Proceedings of the 5th International Workshop on Boolean Problems*, pp. 223–240, 2002.

Dave Steinkraus, Ian Buck, and PY Simard. Using gpus for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*, pp. 1115–1120. IEEE, 2005.

Mark E Stickel. A prolog technology theorem prover: Implementation by an extended prolog compiler. *Journal of Automated reasoning*, 4(4):353–380, 1988.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Advances in Neural Information Processing Systems*, volume 28, pp. 2440–2448. Curran Associates Inc., 2015.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

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 the 2015 conference on empirical methods in natural language processing*, pp. 1499–1509, 2015.

Geoffrey G Towell and Jude W Shavlik. Knowledge-based artificial neural networks. *Artificial intelligence*, 70(1-2):119–165, 1994.

Geofrey G Towell, Jude W Shavlik, Michiel O Noordewier, et al. Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the eighth National conference on Artificial intelligence*, volume 861866. Boston: AAAI Press, 1990.

Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. Complex embeddings for simple link prediction. In *International Conference on Machine Learning*, pp. 2071–2080. PMLR, 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, 2017.

Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge graph and text jointly embedding. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1591–1601, 2014.

Robert West, Evgeniy Gabrilovich, Kevin Murphy, Shaohua Sun, Rahul Gupta, and Dekang Lin. Knowledge base completion via search-based question answering. In *Proceedings of the 23rd international conference on World wide web*, pp. 515–526, 2014.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, 2020.

Wenhan Xiong, Thien Hoang, and William Yang Wang. Deeppath: A reinforcement learning method for knowledge graph reasoning. *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2017.

Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding entities and relations for learning and inference in knowledge bases. *International Conference on Learning Representations*, 2015.

Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base reasoning. *Advances in Neural Information Processing Systems 30*, pp. 2316–2325, 2017.

Shuai Zhang, Yi Tay, Lina Yao, and Qi Liu. Quaternion knowledge graph embeddings. *Proc. NeurIPS*, pp. 2731–2741, 2019.