Clause Representation for Proof Guidance using Neural Networks

University of Miami

Supervised by Dr. Geoff Sutcliffe

John McKeown

October 19, 2021

Abstract

Automated reasoning is the study of logic and reasoning from a computational perspective. While machine learning has enjoyed most of the AI hype lately, many people have either never heard of automated reasoning, or think that automated reasoning is an obsolete relic from a time before machine learning. The truth is that formal logic systems are uniquely well suited for encoding many unsolved real world problems, and there still remains exciting work to be done in order to make automated reasoning more efficient.

Automated Theorem Proving (ATP) is a subfield of automated reasoning wherein a conjecture is formally proved from axioms using computational methods. ATP benefits greatly from heuristics that estimate which inferences, during a proof attempt, are most relevant for finding a proof. Machine learning has enjoyed great success in recent years, and is well suited for mining heuristics from data. Having good representations of logical formulae is a very important prerequisite to using machine learning for finding useful heuristics for ATP.

This thesis explores some approaches to representing logical formulae as rooted ordered labeled directed acyclic graphs, as well as finding embeddings of these logical formulae graphs as vectors. Vector embeddings serve as an ideal representation of logical formulae for processing in machine learning models such as neural networks.

Dedication

To my family, friends, colleagues and acquaintances.

Declaration

I hereby declare that the following thesis is my own work. It is being submitted for the degree of Master of Science in Computer Science from the University of Miami. It has not been submitted for any other degree or examination at any other university.

Acknowledgements

I want to thank my advisor Geoff Sutcliffe for his input regarding the soundness of various ideas related to this research as well as his patience. Qinghua Liu was also an indispensible resource in my research leading me in the right direction via sharing references to relevant past work, as well as her own advice. I would also like to thank Stephan Schulz for his excellent work on the E theorem prover [1] and his help in interfacing with E.

Contents

Chapter 1

Background

1.1 Automated Reasoning

Automated reasoning is the study of logic and reasoning from a computational perspective. Automated reasoning is a broad topic. It includes concrete subtopics like automated theorem proving and proof verification, as well as abstract topics such as the discovery of new formal logic systems and the determination of which systems are useful for solving which problems. Automated theorem proving (ATP) is the subfield of automated reasoning that deals with computational methods for proving a conjecture from a set of axioms. The conjecture and axioms are presented to an ATP system as logical statements called (*well-formed*) *formulae*, which need to conform to a particular standard. Different formal logic systems define this standard in their own way.

1.2 Formal Logic Systems

1.2.1 Propositional Logic

In propositional logic, well-formed formulae are formed from *propositions* (constant expressions that are interpreted as either $true$ or $false$.) These propositions can be combined using *logical connectives* such as *or* (|), *not* ($\sim$), *and* (&), *implication*

($\Rightarrow$), and *equivalence* ($\Leftrightarrow$). In the following, a and b are examples of propositions:

- $a \mid b$ is *true* if and only if either a is true or b is *true* (or both).
- $\sim a$ is *true* if and only if a is $false$.
- $a \,\&\, b$ is *true* if and only if a is *true* and b is *true*.
- $a \Rightarrow b$ is equivalent to $(\sim a) \mid b$.
- $a \Leftrightarrow b$ is equivalent to $(a \Rightarrow b) \,\&\, (b \Rightarrow a)$.

Although propositional logic is a very simple system of logic, theorem proving in propositional logic is an NP-complete problem. In fact, it was the first problem ever to be proven to be NP-complete [2]! Perhaps unsurprisingly, more complicated formal logic systems lead to even worse complexity for theorem proving.

1.2.2 First-Order Logic

First-order logic (also known as predicate logic) introduces the concepts of *constants*, *functions*, *predicates*, and *variables*. Constants represent objects in the world (e.g., *apple*). Functions represent transformations between objects (e.g., *color* such that perhaps $color(apple) = red$). Predicates are truth-valued function, e.g., *isRed* such that $isRed(apple)$ means that the apple is red. Variables can be existentially ($\exists$) or universally ($\forall$) quantified, and represent speculative objects in the real world. Variables are typically stylized in uppercase. Existential quantification allows statements such as, "There exists an object, X, such that X is red." $(\exists X(isRed(X)))$. Universal quantification allows statements such as, "For all objects, X, if X is an apple, then X is red or green or yellow" $(\forall X(isApple(X) \implies (isRed(X)|isGreen(X)|isYellow(X))))$. Statements in first-order logic (and propositional logic) can be converted into Clause Normal Form (CNF). Each formulae is converted into a set of *clauses* which are implicitly combined with *and*. A clause is a disjunction of *literals*. A literal is an application of a predicate to terms ($isRed(apple)$) or the negation of such a predicate

$$\forall X \exists Y \sim ((p(X) \;\&\; q(Y)) | p(Y))$$
$$\downarrow$$
$$\underline{(\sim p(X) \mid \sim q(sk(X)))} \;\&\; \underline{\sim p(sk(X))}$$

Figure 1.1: Example Conversion to CNF

application ($\sim isRed(apple)$). In propositional logic, the resulting clauses are logically equivalent to the original statements. This is not true in first-order logic because the conversion introduces extra functions called *Skolem functions* in order to remove existential quantification. With existential quantification eliminated, all variables are implicitly universally quantified. Nonetheless, a set of clauses is satisfiable if and only if the first-order formulae they were derived from are satisfiable. Figure 1.1 shows an example conversion of a first-order formula to clauses. In Figure 1.1, the underlined sections are clauses, and sk is a skolem function introduced to eliminate the existentially quantified variable Y. The skolem function can be thought of as picking a particular value for Y since one must exist for each value of X. The additional concepts provided by first-order logic allow for a more natural representation of real world problems than is possible using propositional logic. One simple thing we take for granted in math is the notion of equality. Equality is a predicate of arity two, but some axioms of equality require quantification over functions or predicates (e.g., $\forall F \forall X \forall Y((X = Y) \implies (F(X) = F(Y)))$). Since quantification over functions and predicates is not allowed in first-order logic, equality is problematic. In practice, equality is so common and important that it is supported by many first-order theorem provers nonetheless. This can be done by adding first-order axioms of equality for each function and predicate (e.g., $\forall X, Y : (X = Y) \implies (f(X) = f(Y))$). A more efficient way of supporting equality is to use additional inference rules such as superposition [3].

1.2.3 Typed First-Order Logic

First-order logic can be adorned with *types* to facilitate encoding of problems. For instance, it is more intuitive for most people to say "all people are mortal"

$(\forall(X : person)\ mortal(X))$ than to say "for all things, if that thing is a person, then it is mortal" $(\forall X(person(X) \implies mortal(X)))$. Type errors can be also be detected in typed first-order logic like they are in a statically typed programming language. This makes encoding problems in typed first-order logic less error prone. Type information can also help theorem provers search for proofs more efficiently.

1.2.4 Higher-Order Logic

Formal systems of higher-order logic exist wherein variables can represent not only objects in the world, but also functions and more. For example, the following statement is well-formed in a higher-order logic, but is not well-formed in first-order logic: $\forall F(p(F(a)) \implies p(F(b)))$. In first-order logic, it is simply disallowed to say "For all functions, F, ..." In order to implement higher-order logical systems without running into Russell's Paradox [4], objects *must* have types. (Although types in first-order logic are optional, they are mandatory in higher-order logic.)

1.3 Saturation Based Theorem Proving

The formal logic systems described above are the most common, but many other formal systems exist. Although first-order logic is not the most expressive of these logical systems, many interesting problems can be represented using first-order logic. These problems range from very practical verification of software and hardware, to problems in the sciences, to theoretical problems in mathematics. Theorem proving in higher-order logic becomes extremely difficult, so first-order logic serves as a sort of middle ground in which a large number of problems can be both represented and solved. Another advantage that first-order logic has over higher-order logic is that it is *complete*, which means that there is a proof of every theorem of a given set of axioms.

1.3.1 Saturation

Although first-order ATP systems such as E [1] and Vampire [5] each have their own methods, strengths, and weaknesses, a method common to many such ATP systems is *saturation*. In a saturation-based ATP system, inference rules are used to grow a set of known statements by adding statements that are logical consequences of previously known statements. Usually this is done in order to find a contradiction between the negated conjecture and the axioms of a problem, thereby proving the conjecture.

When an ATP system first receives an input problem, formulae are often converted into CNF in order to enable the application of common inference rules. A set of first-order formulae are not necessarily logically equivalent to their CNF form, but a set of first-order formulae is satisfiable if and only if, the set of clauses in their CNF form is satisfiable. This means that a contradiction found amongst the CNF form of some formulae implies a contradiction in the original formulae.

Once the axioms and negated conjecture are converted into clauses, saturation can proceed. Resolution and superposition are two rules that ATP systems use to infer new clauses. If a contradiction exists between the negated conjecture and the axioms, then these rules are guaranteed to eventually generate the empty clause ($false$), thereby exhibiting a contradiction. An ATP system is called *refutation-complete* if it will eventually find such a contradiction whenever one exists. This is a weaker version of the notion of completeness defined above.

1.3.2 Unprocessed and Processed Sets

Saturation based theorem provers typically maintain two sets of clauses: a *processed* set and an *unprocessed* set. The axiom clauses and the negated conjecture clauses are split between the unprocessed and processed sets. Usually the processed set is initially empty, but certain proof strategies may dictate that certain clauses start off in the processed set. During its proof search, a prover chooses a *given* clause from the unprocessed set to bring into the processed set. The given

clause is then used in combination with clauses in the processed set to infer new clauses. A proof has been found if the empty clause is inferred. If an empty clause is not inferred, the inferred clauses are simplified in various ways and checked against all clauses in the processed and unprocessed sets for redundancy with previously known clauses. If an inferred clause is not redundant, it gets inserted into the unprocessed set. If an inferred clause makes other known clauses redundant, the less general clauses are replaced by the more general inferred clause.

After processing the inferences from a given clause, the prover once again selects a given clause from the unprocessed set to bring into the processed set and the process continues. The reason for maintaining separate processed and unprocessed sets is efficiency. By choosing clauses one at a time from the unprocessed set, the prover avoids having to consider inferences between all pairs of clauses, and instead only performs inferences involving the given clause from the unprocessed set and clauses from the processed set.

1.4 Given Clause Selection

As one might expect, the choice of given clause in a saturation system is very important. The unprocessed set grows much more quickly than the processed set. Because of this, finding a clause in the unprocessed set that is useful for a proof soon becomes like finding a needle in a haystack. Even sophisticated theorem provers often fail to find proofs, selecting irrelevant given clauses and failing to select clauses that are needed for a proof. On a practical note, if a theorem prover is going to find a proof, it will usually do so very quickly. The likelihood of finding a proof in 10 hours is often not much higher than the likelihood of finding a proof in 10 seconds. Because the size of the unprocessed set grows superexponentially, proof search is easily derailed by poor given clause selection. This is why having good heuristics can make such a big difference.

1.5 Axiom Selection

The search for a proof is made more difficult when a problem is initially stated with many extra axioms that are not needed for a proof of the conjecture. Therefore, when given a large number of axioms as input, ATP systems often try to use only the most relevant axioms to increase the probability of finding a proof in a reasonable amount of time. This selection of only the most relevant axioms is referred to as *axiom selection* or *premise selection*. It is important to note that the cost of misclassifying an *unimportant* axiom as *important* is much lower than the cost of misclassifying an *important* axiom as *unimportant*. Failing to select a single important axiom will mean that no proof is found. On the other hand, selecting a single unimportant axiom can make the search for a proof less efficient, but is less likely to single-handedly thwart the proof search. The work described in this thesis focuses primarily on given clause selection and general representation of clauses, but the techniques also apply in axiom selection.

1.6 Research Goals

This thesis is primarily focused on the development of *clause embeddings* on top of which a simple neural network can learn given clause selection. A *clause embedding* refers to a fixed-length vector of floating-point numbers acting as features of a clause. In other words, the goal is automatic feature engineering for given clause selection. Chapter 2 describes previous work in this area. Chapter 3 describes methodology. Chapter 4 describes supervised approaches to the development of clause embeddings. Chapter 5 describes unsupervised approaches to the development of clause embeddings. Chapter 6 describes implementation details. The approaches described in Chapters 4 and 5 are evaluated in Chapter 7. A conclusion and directions for future work can be found in Chapter 8

1.7 Representing Clauses as Graphs

As a first step towards a clause embedding, consider a clause as an ordered tree. The tree has the *or* symbol ("|") as its root, representing a disjunction of literals. A negative literal is represented by a node for the negation, whose only child is the subtree representing the atom being negated. A positive literal induces no such additional node and is therefore simply represented as a subtree representing its atom. Constants and variables are leaf nodes. The arguments of predicate and function symbols are the ordered children of the corresponding nodes in the tree. The order of the children of a node in a tree matters. Nodes also have associated symbols and types where the type of a node is one of the following: "|", "$\sim$", "predicate", "function", "constant", or "variable." Figure 1.2 shows a fairly general example of this.

Many machine learning models benefit from merging equivalent subtrees to form a Directed Acyclic Graph (DAG), as in Figure 1.3. Consequently, the preferred representation of a clause could most precisely be called a "rooted ordered typed labeled DAG," but for brevity I will simply refer to one as a "DAG."

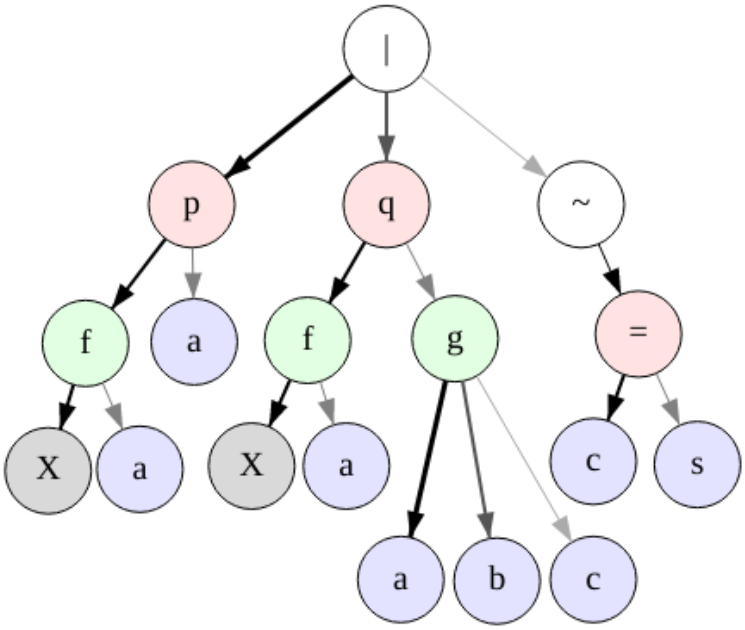

Figure 1.2: Tree for **`p(f(X,a),a) | q(f(X,a),g(a,b,c)) | c!=s`**
p and q are predicates, f and g are functions, a,b,c,r, and s are constants, and X is a variable. This information is also depicted using node colors with white, red, green, blue, and grey being used for logical connectives, predicates, functions, constants, and variables respectively. Arrow thickness depicts *children ordering*, where thicker lines denote earlier children. (In general, an order-respecting DAG layout may be impossible.)

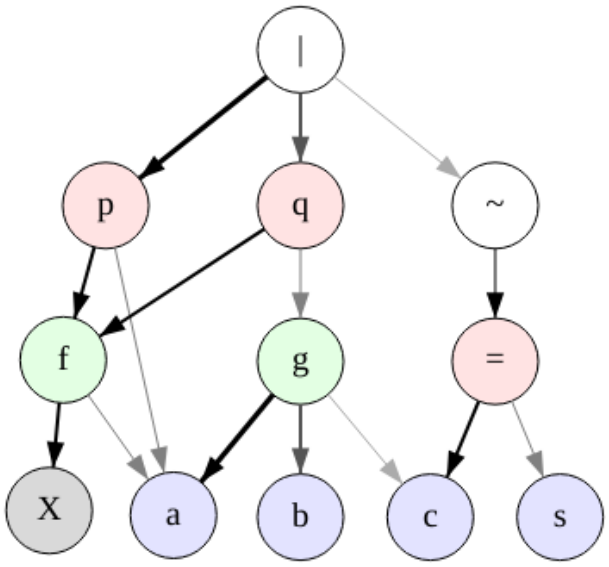

Figure 1.3: DAG for **`p(f(X,a),a) | q(f(X,a),g(a,b,c)) | c!=s`**
(Compare with the tree version in Figure 1.2)

Chapter 2

Previous Work

2.1 Graph Neural Networks

The Graph Neural Network (GNN) was born out of a need to process graph structured data using machine learning. Although many ideas related to GNNs are timeless, the term "Graph Neural Network" was coined in 2005 by Gori et al.[6]. The short but illuminating paper describes GNNs in the following language:

> The state [of graph node n], x_n, is defined as the solution of the system of equations:
>
> $$x_n = f_w(l_n, x_{ne[n]}, l_{ne[n]}), n \in \mathbb{N}$$
>
> where l_n, $x_{ne[n]}$, $l_{ne[n]}$ are the label of n, and the states and the labels of the nodes in the neighborhood of n, respectively.

An exact understanding of the above formalism is not necessary for understanding the rest of this thesis and its formulation of GNNs, but is helpful for understanding the history of GNNs. The original approach was to find a solution to this system of equations where the form of f_w is a neural network, and where the sets $x_{ne[n]}$ and $l_{ne[n]}$ are each separately summed before input into f_w so that f_w has fixed arity. The solution was found by updating node states, x until convergence. This system of equations has a unique solution when f_w is a contraction mapping (with resepct to x) due to the Banach fixed-point theorem [7]. The restriction of

f_w to a contraction mapping, however, limits the form of f_w in such a way that the influence that one node can have on another decays exponentially with the length of the shortest path between them.

Modern GNNs often perform only a small fixed number of state update steps rather than updating states until convergence. Modern GNNs also use more sophisticated methods for propagating information around the graph, which are more selective about which information to keep or throw away. For instance, GNNs that use LSTM-like gating to control state updates were introduced by Ruiz et al. [8].

Parallel to some of this earliest research on GNNs, Convolutional Neural Networks (CNNs) began to experience great success. CNNs operate on grids of data, where a grid is a special type of graph in which every node has the same degree. This led researchers to consider how convolution could be generalized to work on generic graphs. Early work in this area focused on so-called *spectral* approaches [9, 10]. These approaches rely on the eigendecomposition of the graph Laplacian matrix that is computed from the adjacency matrix, and as such are applicable only when each input to the network shares the same graph structure (like traditional CNNs). In 2017, Kipf et al. [11] introduced Graph Convolutional Networks (GCNs) that, although derived from the spectral graph convolution, do not require each input graph to have the same number of nodes and same connectivity. This model is a GNN wherein neighbor aggregation takes the degree (number of incident edges on a node) of neighboring nodes into account. In a GCN, nodes that have a large degree are weighted less during aggregation. Intuitively, this means that nodes with high degree don't get to "dominate the conversation."

A large number of popular neural networks on graphs are special cases of "Message Passing Neural Networks (MPNN)." The MPNN name was coined by Gilmer et al. [12]. MPNN allow edge labels, e_{vw}, in addition to the common node labels, x_v. The edge labels provide a mechanism for encoding different types of relationships. In particular, they provide a way to include child ordering information, which is useful for graphs of logical formulae and/or clauses.

In the language of Gilmer et al.:

> The message passing phase runs for T time steps and is defined in terms of message functions M_t and vertex update functions U_t.
>
> During the message passing phase, hidden states h_v^t at each node in the graph are updated based on messages m_v^{t+1} according to
>
> $$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \quad (2.1)$$
> $$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}) \quad (2.2)$$
>
> where in the sum, $N(v)$ denotes the neighbors of v in graph G.
>
> The readout phase computes a feature vector for the whole graph using some readout function R according
>
> $$\hat{y} = R(\{h_v^T | v \in G\})$$

2.2 Other Models

Two other interesting models that are suitable for processing formulae are the Tree LSTM/DAG LSTM and Recursive Neural Networks. The Tree LSTM model generalizes the LSTM (Long Short Term Memory) gating mechanism to tree structured data. Because Tree LSTMs and their generalization to DAGs, DAG LSTMs, operate on only acyclic graphs (unlike GNNs), they don't require multiple message passing phases. Instead, each node state is updated only once after the node's children have been updated. This can easily be implemented using a reverse topological sort on the nodes of the DAG. Bidirectional DAG LSTMs have been used very successfully both for proof step classification (on the HolStep [13] dataset) and for axiom selection (on the Mizar40 [14] dataset) by Crouse et al. [15].

Recursive Neural Networks (not to be confused with Recurrent Neural Networks) are used by Chvalovsky et al. for given clause selection in ENIGMA-NG [16]. They consider each function, predicate, and logical connective to be a neural network, and treat constants and variables as learnable constant vectors. Although this approach seems very reasonable, and very nicely mirrors the reality that these

graphs come from logical formulae, the data requirements are much larger than for a message passing network because there is no weight sharing between individual function/predicate/connective submodules. These recursive networks failed to surpass the performance of the earlier ENIGMA systems that use gradient boosted tree ensembles (XGBoost [17]) trained on handcrafted features. These handcrafted features are the following, in their words:

> **Vertical Features** are (top-down-)oriented term-tree walks of length 3. For example, the unit clause $P(f(a,b))$ contains only features (P,f,a) and (P,f,b).
>
> **Horizontal Features** are horizontal cuts of a term tree. For every term $f(t_1,\ldots,t_n)$ in the clause, we introduce the feature $f(s_1,\ldots,s_n)$ where s_i is the top-level symbol of t_i.
>
> **Symbol Features** are various statistics about clause symbols, namely, the number of occurrences and the maximal depth for each symbol.
>
> **Length Features** count the clause length and the numbers of positive and negative literals.
>
> **Conjecture Features** embed information about the conjecture being proved into the feature vector. In this way, ENIGMA can provide conjecture-dependent predictions.

The vast number of vertical and horizontal features leads to poor accuracy of machine learning models. This is the so-called "curse-of-dimensionality." The authors thus use feature-hashing to reduce the number of features. They do this by representing each feature as a string and using a generic string hashing function to map features to an index in a fixed sized array. They then simply sum feature counts when two features happen to collide.[1] The conjecture features are roughly the same as the features for given clauses, and are simply concatenated with the given clause features before evaluation in XGBoost. The authors report true-positive rate (accuracy in classifying positively labeled clauses) and true-negative rate (accuracy in classifying negatively labeled clauses). Although their recursive network approach didn't quite perform as well as XGBoost on these handcrafted features (82% vs. 83.35% true-positive rate on their class-rebalanced testing data),

[1] If they had replaced the XGBoost with a (non-graph) neural network, then they could also replace this arbitrary feature hashing function with a learnable feature hashing function. Perhaps this is an interesting idea for future work.

it did outperform a linear classifier trained on the same features (80.54%). Interestingly, the true-negative rate of the recursive network was actually higher than that of the XGBoost model (76.88% vs. 72.60%) with the linear model once again underperforming (62.28%). The XGBoost model gave the best results when tested on the MPTP2078 dataset solving 1256 problems compared with the 1197 problems solved by the recursive network model, the 1210 problems solved by the linear model, and the 1086 problems solved by the E baseline guidance.

Chapter 3

Methodology

3.1 Clause Selection as Classification

Given clause selection can be thought of as a binary classification problem, with the goal being to classify whether a clause is part of a refutation. Given clause selection presented as a classification problem throughout this thesis. Downsides to this approach and alternatives to be explored in future work are discussed in Section 8.4.2.

3.2 Selection Context

Heuristic evaluation of a clause for given clause selection should not be done in isolation. Selecting a particular clause as the given clause may be a good choice when solving a certain problem, but a bad choice when solving a different problem. A clause is good to select if it will lead the theorem prover towards a refutation. This is heavily dependant upon the original problem clauses. Therefore, for the task of given clause selection, classification of a given clause as good or bad should be performed relative to its *context*. The context is represented here as a set of clauses.

In a saturation based system, the most general context would be all of the clauses in both the processed and unprocessed sets, since they represent the cur-

rent state of the theorem prover. Although this approach captures the most information, it has a very serious downside – the unprocessed set grows very quickly. Since evaluating a clause against its context takes more time when the context is larger, evaluation of clauses would get slower and slower during the proof search.

One might consider using only the processed set as context since it grows much more slowly than the unprocessed set. Even so, a changing context means that a clause that was previously considered bad may become good, or vice-versa. Clauses are normally evaluated upon entry into the unprocessed set. With a changing context, every clause in the unprocessed set must be re-evaluated every time the context changes.

Because of these difficulties, previous work has used a static context [16]. Although one could use all clauses from the input problem as context, it has been common to use just the original negated conjecture clauses from the input problem as the heuristic evaluation context [16]. This is a helpful bias as it makes the proof search *goal-oriented*. Still, the full problem context has strictly more information, so it would be surprising if it provided no useful information when processed well.

3.3 The E Theorem Prover

The E theorem prover [1] is a saturation based theorem prover developed by Dr. Stephan Schulz. E uses resolution and superposition, and has proven its flexibility and quality in the CADE ATP System Competition (CASC) [18]. E has been in active development since 1998, and is frequently used as a base for theorem proving experiments [18, 16]. E's popularity is partially due to its great performance (both in terms of numbers of problems solved and speed in solving them). E also has excellent documentation and the code is easy to install and run. E is ideal for research because it supports custom heuristics and many command line flags that customize behavior.

Although E has many good heuristics for given clause selection, strict adherence

to some heuristics can break the refutational completeness of the theorem prover. E maintains refutational completeness by using multiple heuristics, only some of which are refutationally complete when used alone. Each heuristic takes turns picking a predefined number of given clauses. For example, a heuristic schedule[1] can be specified as follows: "for every 5 times heuristic A is used, heuristic B is used 3 times and heuristic C is used 2 times before coming back to heuristic A."

3.4 The MPTPTP2078 Dataset

The Mizar Project [19] is dedicated to formalizing a large number of mathematical concepts and theorems in first-order logic. This is very important work for benchmarking theorem provers. 2078 problems from Mizar, in TPTP [20] format, make up the MPTP2078 dataset [21]. This dataset is based on an earlier MPTP Challenge dataset that consists of 252 problems extracted by the MPTP system from 33 articles leading to a proof of the Bolzano-Weierstrass theorem. The MPTP2078 problems are extracted by a newer version of the MPTP system from the same 33 Mizar articles. It has many more problems because all problems that can be extracted are used instead of just the ones related to the Bolzano-Weierstrass theorem. The problems in this dataset exist in *bushy* and *chainy* variants. The bushy variants include only the axioms necessary for a proof, whereas the chainy variants use all previously derived (within Mizar) formulae as axioms. The determination of which axioms are necessary is done in a greedy manner by removing axioms until Mizar can no longer automatically verify a proof. The chainy variants are harder to solve because they include axioms that are unnecessary for a proof. The chainy variants more realistically emulate the scenario of proving new conjectures over a large mathematical library. The data used in this thesis comes from the bushy problems in a cleaned version of this dataset, called MPTPTP2078. The bushy variants are used for the following reasons:

[1] E documentation and code simply call this heuristic schedule a "Heuristic" and use the phrase "Clause Evaluation Functions" to refer to the individual heuristics.

1. When using a smaller set of axioms, a larger proportion of potential given clauses will be related to the negated conjecture context clauses. This means less noise in the training data.

2. Since bushy variants are easier for E to solve, more training data can be extracted from successful E proofs.

For more information about the MPTP2078 and MPTPTP2078 datasets, visit the following URLs:
`https://github.com/JUrban/MPTP2078`
`https://github.com/TPTPWorld/MPTPTP2078`.

3.5 PyTorch and PyTorch Geometric

PyTorch is a very popular deep learning framework. All models in this research were trained using PyTorch. PyTorch Geometric (PyG) is a popular framework for GNNs built atop PyTorch. PyG will be used more heavily in future research because it is more efficient and more easily parallelized on GPUs than the more naive PyTorch code.

3.6 Creation of Training and Testing Data

This section describes the creation of the training and testing data that is used throughout the thesis. The MPTPTP2078 bushy problems are fed as input to E in order to extract the negated conjecture context clauses. The original first-order problems are then fed to E to be solved. The set of symbols used in all problems and solutions (called the *signature*) is extracted. This includes the names of predicates, functions, constants, variables, and logical connectives. For each generated proof, the given clauses that were used in the proof are labeled *positive* and the given clauses that were not used in the proof are labeled *negative*. The labelled given clauses are then split into training (80%) and testing (20%) sets. For the sake of

fair evaluation on the testing data, labelled given clauses from a particular problem are either all in the training set or all in the testing set.

Details of this process can be found in Chapter 6.

Chapter 4

Supervised Embedding for Clauses

Various strategies have been employed for representing a clause as a vector of features. For example, recall from Section 2.2 that ENIGMA[16] uses handcrafted features along with feature hashing. This work considers various Graph Neural Net (GNN) architectures.

Both of the models described in this chapter operate on clause DAGs of the form described in Section 1.7 in order to embed DAGs as vectors. The approaches presented in this chapter are referred to as *supervised* because these embedding models are trained by feeding embedding model outputs into a classifier to predict the labels described in Section 1.7. Since both the clause embedding models and the classifier are differentiable, all parameters can be trained at once. This is in contrast with the unsupervised clause embedding models described in Chapter 5, where labels are not used for training.

Each symbol used in the MPTPTP2078 dataset is assigned its own learnable vector. These symbol vectors can be thought of as part of the model and therefore are the same for every problem. When processing a clause DAG, each node has an associated vector. These node vectors start off as the learnable vector embeddings corresponding to the node symbols. The node vectors are then updated as the

various layers of the GNN model dictate[1]. Once the given clause and context DAGs have been processed, each DAG is summarized by a single vector. The details of how node vectors are updated and how DAGs are summarized from their updated node vectors lie in the definition of each clause embedding model such as those described in Section 4.1 and Section 4.2.

Once each DAG is embedded, the context DAG embeddings are further summarized into a single vector by $f(\sum g(x_i))$ where f and g are simple 2-layer neural net modules and the x_i's are the context clause embeddings. The given clause embedding is concatenated with the context summary vector and fed into a classifier. The classifier used is a simple 2 layer neural network module with a ReLU activation function after the first layer. The output has no activation, but the loss applied is a "binary_cross_entropy_loss_with_logits" in PyTorch, which makes this model output suitable for binary classification.

4.1 The Custom Model

My first model was created using only PyTorch and custom modules, because I feared that GNN libraries wouldn't be flexible enough to encode this model. Because of this, I refer to it throughout this thesis as my *custom model*. This model is atypical because of the following features that I believed would be helpful:

- Asynchronous processing of nodes via a reverse topological sorting.
- Nodes "attending" to their children, similar to [22].
- The whole DAG being summarized by the root node, not via pooling.

While traditional GNN models update all nodes' feature vectors at one time (synchronously), this model updates the nodes' feature vectors iteratively from

[1]While CNN and fully connected neural networks layers are usually described as transforming inputs into new outputs, since GNNs typically don't change anything about the shape of the graph, GNN layers are sometimes described as mutating inputs in place.

leaves to roots (asynchronously). Processing nodes asynchronously allows information to propagate from the leaves of a DAG to the root much more quickly than in traditional GNN models (without parallelization). Each node vector is updated only once as a function of the node vectors of that node's children. This gives the model a strong bias towards defining terms by their subterms and not by their siblings or parents.

In most GNNs, a node vector is updated as a simple function of the sum of all neighboring nodes' vectors. However, not all subterms of a term are equally important. Attention captures this idea, and has been very useful in natural language tasks [22, 23]. Although Graph Attention Networks (GATs)[24] are an adaptation of this idea to GNNs, the model presented here uses a form of attention that even more closely mimics that of Vaswani et al. [22]. More specifically, learnable *query*, *key* and *value* matrices are used. Intuitively, the *query* matrix can be thought of as a mechanism for a parent node to "ask a question" of its children. The *key* matrix can then be thought of as a way for the children to assess how relevant the question is to them. The *value* matrix can then be thought of as the answers of the children back to the parent. In reality, the parent node is also included in the "children" for this operation so that information from the parent can be retained after the update. The resulting responses from the "children" are then summed and this sum goes through 1 linear layer with a ReLU activation to yield the new parent node vector.

Summarizing a rooted DAG as the root node vector is essentially what is done by TreeRNN/DagRNN models, and seemed reasonable since all of the node updates send information towards the root. However, sum pooling easily incorporates information from nodes far from the root, whereas a root node vector summary struggles in this regard. One explanation for the relative success of sum pooling is that the derivative of a sum with respect to any of the summands is 1. Consequently, the effect of any node vector on the entire DAG representation is simple. In contrast, the derivative of the root node vector with respect to a leaf node vector

is complicated.

Although the ideas of asynchronous processing and attention are promising (these ideas of asynchronous processing and attention were both used very successfully by Crouse et al. [15]), there has also been a great deal of success with traditional GNN models. The custom PyTorch model described in this section is very slow, and failed to match the accuracy (on the dataset described in Section 6.1.3) of very simple models that use only symbol counts as input. These ideas are worth revisiting in the future, but custom code led to both slow model execution and slow development time.

4.2 The Simple PyTorch Geometric Model

Because of these drawbacks to the custom model, I chose to create a very simple PyTorch Geometric (PyG) model using two standard graph convolutional network (GCN) layers. GCN layers compute nodes as a weighted sum of neighboring nodes. The weights are chosen intelligently so that nodes with high degree have less effect on their neighbors. Intuitively this prevents certain "loud" nodes from "dominating the conversation." This is a good middle ground between a simple sum and something even more complex like attention.

4.3 Performance

The custom model achieved a very pitiful 77% accuracy (training and testing) on the given clause selection classification task. The simple PyG model, however, achieved a more reasonable 94% accuracy on training data and 88% accuracy on testing data.

Although both the custom model and the more simple PyG model yielded poor performance for the task of given clause selection, it is unclear whether the simple classifier described at the beginning of this chapter is bad, the formulation of given clause selection as a classification problem is bad, or the DAG embedding models

themselves are bad. This ambiguity led to the investigation of DAG embedding models in isolation (not specifically in the context of given clause selection). This investigation into DAG embedding models in isolation is the subject of Chapter 5.

Chapter 5

Unsupervised Embedding for Clauses

This chapter explores ways of training a clause embedding model in a way that is decoupled from given clause selection.

5.1 Autoencoders

An autoencoder is a type of neural network consisting of two subnetworks: an encoder and a decoder. The goal of an autoencoder is to learn better representations of data, where a "better representation" is defined by the architecture of the encoder and decoder as well as the loss function used during training. During the training of a normal feed-forward neural network, an input, x, is passed through the network, $f_\theta : X \to Y$, to obtain $f_\theta(x)$. The parameters of the network, θ, are then updated so that $f_\theta(x)$ approaches the desired output, y. When training an autoencoder, $y = x$. Thus $\theta = (\theta_1, \theta_2)$ is updated during training to satisfy $f_\theta(x) = decoder_{\theta_2}(encoder_{\theta_1}(x)) \approx x$.

It may be hard, at first glance, to see the motivation behind training a neural network to learn the identity function. Optimizing this "reconstruction" objective guarantees that $encoder(x)$ retains all of the useful information about x for any downstream task. Indeed, any function, $f : X \to Y$, could be transformed into

a function, $g : Z \to Y$, via $g(z) = f(decoder(z))$, where $z = encoder(x)$ and $x \approx decoder(z)$. This means that, for future learning problems, x can be replaced by $encoder(x)$ as the input.

It may also be hard, at first glance, to see why replacing x with $encoder(x)$ would be desirable. In general, it isn't, but the fact that $encoder(x)$ retains the important information in x means that the encoder can be freely designed in order to fulfill other constraints. The most common constraint for the encoder is that its output should have significantly smaller dimension than its input. If the autoencoder is still able to train effectively, then the encoder becomes an effective method for non-linear dimensionality reduction.

One very successful application of autoencoders is the automatic extraction of high level features from image data. Convolutional neural networks are very useful for processing image data. Convolutional autoencoders use convolutional layers to encode an image as a much smaller dimensional vector, which is then decoded into a full image again using deconvolution layers. This enables downstream classifiers to operate directly on a small dimensional vector of high-level features, $encoder(x)$, rather than on the original image, x. Another benefit of autoencoders is that they allow for "semisupervised learning", wherein a large amount of unlabelled data is used to train an autoencoder, before the decoder is replaced with a classifier that is then trained using a smaller amount of labelled data.

Variational autoencoders aim to ensure convexity in the space of encoded vectors. They accomplish this by having the encoder output mean and standard-deviation vectors that represent a distribution that is then sampled in order to get the vector which is finally sent to the decoder.

5.2 Graph Autoencoders

In 2016, Kipf & Welling introduced the graph autoencoder (GAE) as well as its variational counterpart (VGAE) [25] (these are the same folks that introduced

GCN layers explained in Section 2.1.). This model is well suited to incorporating graph connectivity information into node feature vectors, and is remarkably simple and elegant. Let $G = (V, E)$ be a graph with adjacency matrix A (dimension $|V| \times |V|$) and node feature vectors X (dimension $|V| \times d$ where d is the dimension of a node feature vector). In the case of the normal GAE, the encoding is simply $Z = GCN(A, X)$, where $GCN(A, X)$ is a simple 2 layer graph convolutional network.

The decoded adjacency matrix is simply $\hat{A} = \sigma(ZZ^\top)$, with σ being the logistic sigmoid function. The sigmoid function maps $ZZ^\top$ to $(0, 1)$ so that $\hat{A}$ is like an adjacency matrix. Note that the network is not explicitly trained to recall the original node features, but only to recall the connectivity of the graph. In order to accommodate logical formulae, a graph autoencoder needs to be aware of edge directions as well as additional information such as edge ordering. This information cannot be encoded in an adjacency matrix and therefore GAE models are ill-suited to clause embeddings.

5.3 Clause Autoencoder

Purgal et al. [26] found a way to make autoencoders work for DAGs representing logical formulae. They use a variety of sequence based models (CNNs, RNNs, WaveNet, Transformers) to encode logical terms and subterms as vectors. These models serve as various forms of encoders. Their decoders have multiple parts: a *symbol extractor network* and multiple *child extractor networks*. The symbol extractor network is trained to extract the top symbol from a subterm, and the i^{th} child extractor network is trained to output the encoding of the subterm rooted at this node's i^{th} child (or all zeros if this node has no i^{th} child). The main idea here is that the encoded vector representation of a logical term should be enough to recover its own symbol and the encoded vector representations of its subterms. If this is achieved, then logical formulae are fully recoverable from only the encoded

vector representation of their root nodes.

Purgal et al. investigated two modes of training, which they call "difference training" and "recursive training." In difference training the ith child extractor network is explicitly trained to reproduce the encoding of the ith child node. In recursive training, direct comparison of encodings is avoided by passing the child extractor output into the symbol extractor network. Here, the loss is based on the difference between the actual and decoded symbols. In other words, it recursively decodes children, verifying them by their decoded symbols.

Purgal et al. evaluated their autoencoder jointly on two datasets: the Mizar40 dataset and their own custom "logical properties" dataset. Their models, although able to obtain an impressive 91% recursive formula decoding accuracy on the smaller and less diverse logical properties dataset, obtained only 6% recursive decoding accuracy on the larger Mizar40 dataset. For a more complete description of their results see [26].

5.4 The Ordered DAG Autoencoders

Purgal et al. considered various encoders, but they were all sequential models that don't take graph structure into account. The work in this section explores my attempts to use a GNN as an encoder instead. The GNN used as an encoder here is a Message Passing Neural Network (MPNN) (recall from Section 2.1). MPNNs are capable of taking into account edge ordering information because they support edge features that are used during message passing (unlike both models from Section 4).

5.4.1 The Variational Ordered DAG Autoencoders

An MPNN encoder was trained so that any node vector in an encoded DAG could both reconstruct its symbol (via a call to the symbol extractor network) and reconstruct the encoded node vector of each of its children (the ith child is

reconstructed via a call to the ith child extractor network). If trained to perfection, this model would be able to decode the entire DAG recursively from the root (duplication of shared subterms aside).

Initial experiments with this MPNN encoder showed that recursive decoding of a DAG from the root fails despite low loss during training. The symbol extractor network trains well to reproduce symbols from encoded node vectors. The child extractor networks are able to faithfully extract the first generation of children and their symbols, but errors in initial extraction of children are amplified as these extractions are used as the input for extracting grandchildren, etc.

"Recursive training" is one possible solution to this failure of recursive decoding, but is quite slow because it is much harder to parallelize. Another possible solution is to mitigate compounding errors in child extraction. My first approach to mitigating compounding errors was to add some gaussian noise to the encoded node embeddings before applying the child extractor networks during training. The hope here was that the child and symbol extractor networks would be forced to be robust to a small amount of input noise, making recursive decoding possible after training. This did not work very well at all. One problem with this approach was that the network could adapt to the input noise by simply making the encoded node embeddings larger (so that the noise was relatively small).

This idea of adding gaussian noise led me to create a "variational" variant of the DAG autoencoder. In this model, the encoder outputs are multivariate gaussian distributions with diagonal covariance matrices for each node in the graph. This directly mirrors traditional VAE models. The decoder modules (symbol extractor and child extractors) are the same as before, but they take in a sample from the encoder output distribution instead of taking the encoder output as input directly.

Kullback–Leibler (KL) divergence is an asymmetric measure of how one probability distribution differs from another. It is closely related to cross-entropy. As in a traditional VAE, encoder output distributions are pushed toward a standard gaussian by a KL divergence term. The child extractor portion of the loss also uses

a KL divergence loss instead of the MSE loss used in the normal DAG autoencoder.

5.4.2 Ordered DAG Autoencoder Evaluation

Finding an appropriate way to evaluate the performance of an autoencoder can be difficult. One way is to test how well the encoded features can be utilized for downstream tasks such as given clause selection or axiom selection. Although this is the truest indication of usefulness, the evaluation is limited to that application as a logical DAG encoding that is well suited for some task might be *uniquely* suited for that task and will not work well in general. Although unlikely, it is more principled to evaluate embeddings in isolation.

Another way to evaluate the performance of an autoencoder is based on the loss during training. This method of evaluation misses the point of recursive decoding. If the downstream application of classification is done using all of the node encodings as input (perhaps using an attention mechanism to select out relevant information from certain nodes), then the children and symbol extraction loss could be an accurate portrayal of applicability. If, however, downstream classification is done by using the root vector as the sole representation of the logical DAG, then the usefulness of an encoding is not accurately portrayed by the loss of the children and symbol extractions during training. Along the same lines, the performance could be evaluated by looking at other metrics available during training, such as the accuracy of symbol decoding and the accuracy of arity prediction (the arity of a symbol is implicitly predicted as one can apply child decoders until some nth child decoder indicates that the symbol has no nth child).

Recursive decoding provides the most wholistic approach for evaluation. Although recursive decoding is too expensive during training, it is important to verify that the root node embedding is able to encode significant information about the DAG as a whole. One approach to this is to decode a DAG of the exact same shape as the input DAG by recursively calling the child extractors. These nodes can then all go through the symbol extractor network and a loss can be computed

based on only the symbols.

Both the non-variational and variational variants of ordered DAG autoencoder failed to train well enough for recursive decoding to be reliable. Both models were able to decode symbols (99% accuracy in symbol extraction), but this accuracy quickly dropped with recursive decoding (65% accuracy for extracting symbols from extracted children vectors).

Chapter 6

Implementation

This chapter describes how training data is created and how PyTorch models can be used inside of E for given clause selection.

6.1 Creating Training Data

6.1.1 Extracting Context Clauses and Solving Problems

The MPTPTP2078 bushy problems are processed by first running E with `--cnf` in order to extract the negated conjecture context clauses. After extracting the context, the original first-order problems are fed to E to be solved. This call to E uses the following flags:

```
--free-numbers --prefer-initial-clauses -R -s
--print-statistics --tstp-format -p
--training-examples=3 --soft-cpu-limit=290
--cpu-limit=300 --definitional-cnf=24
--split-aggressive --simul-paramod
--forward-context-sr --destructive-er-aggressive
--destructive-er -tKBO6 -winvfreqrank -c1 -Ginvfreq -F1
--delete-bad-limit=150000000 -WSelectMaxLComplexAvoidPosPred
```

Most of these flags were copied from Chvalovsky et al. [16]. One of the most

important flags is `--training-examples=3`. This flag makes E output (in addition to its normal proof output) all given clauses annotated with `#trainpos` or `#trainneg` indicating whether or not they were used in the proof.

6.1.2 Inferring the Signature

The E output clauses (from both CNF conversion and proofs) from Section 6.1 are used to infer the signature of MPTPTP2078. The signature includes predicate, function, variable, and constant names as well as logical connectives. Extracting the signature from E output clauses makes TPTP parsing easier and provides access to any symbols added by E. These symbols from the extracted signature are each given a unique integer to represent them. These integers are used as indices into the list of learnable feature vectors in the models described at the beginning of Chapter 4.

6.1.3 Final Training/Testing Example Labeling

For each proof generated in section 6.1, each selected clause, with its associated context and label, becomes one training example. If a selected clause was in a proof, then it gets a label of 1, otherwise it gets a label of 0. The clauses themselves are parsed by a simple TPTP CNF-only parser.[1] The common subterms of a parse tree are merged (making the trees into DAGs), and symbol strings are replaced with the integers assigned in the signature. The ordering information in these DAGs is important. This results in the data structure representing a clause being deceptively complicated. The edges are represented as an ordered list of ordered pairs of node indices.[2] Although this representation sufficiently encodes the ordering information, edge ordering is redundantly encoded as an array of *edge-types* (integers) for each edge.[3] This encoding is a standard encoding for graphs in PyTorch Geometric. A triplet of $(givenClause, contextClauses, label)$ represents

[1] The same simple parser is used for inferring the signature.
[2] PyG documentation refers to this common attribute as `edge_index`.
[3] PyG documentation refers to this common attribute as `edge_attr`.

one training example.

6.2 Interfacing with E

The E theorem prover is written in C. Since C++ is directly descended from C, C programs can interface with C++ with relative ease. PyTorch is primarily a Python framework, but it has a C++ interface called *libtorch*. After training a PyTorch model, it can be just-in-time compiled into *torchscript* format, saved, and loaded from C++ using libtorch. I wrote a custom C++ library for calling libtorch models for given clause selection. This simple library was linked into a custom Clause Evaluation Function (CEF) written into E. Once this CEF was defined, it could be invoked from the E command line with the `-H` flag.

Although libtorch is a very efficient way to invoke a PyTorch model from E, only a limited subset of PyTorch models can be compiled with torchscript. In addition to this, all PyTorch models must be adorned with Python type annotations. PyTorch Geometric has only limited support for torchscript (although support is getting better). Because of these limitations, future research will work toward a more generic named-pipe or shared memory interface with E.

Chapter 7

Model Comparisons and Evaluations

Recall the four clause embedding models considered in this thesis:

- The Custom Model (Section 4.1)
- The Simple 2-layer PyG GCN model (Section 4.2)
- The Ordered DAG Autoencoder (Section 5.4)
- The Variational Ordered DAG Autoencoder (Subsection 5.4.1)

These models were evaluated and compared in the context of given clause selection as a classification task, as well as in the context of a few other relevant objectives. The custom model is omitted in some of the following discussion because it was prohibitively slow to train.

7.1 Explicit Clause Objectives

The goal of this section is to evaluate clause embeddings independently of the given clause selection task. The custom model, the simple 2-layer GCN model, and the two autoencoder models are evaluated for objectives other than given

clause selection. The following features of a clause were chosen as being important for a DAG embedding to preserve.

- **Horn** - Is a clause a Horn clause (at most one positive literal)?
- **NumNodes** - How many nodes does a clause graph contain?
- **NumDistinctVars** - How many distinct variables occur in the clause?
- **NumEqualities** - How many "=" symbols occur in the clause?
- **NumNegations** - How many negative literals are in the clause?
- **MaxDepth** - What is the maximum nesting of subterms in the clause?

The effect of pretraining (whether for given clause selection or autoencoding) is evaluated by training classifiers atop both pretrained and simply initialized embedding models. This is denoted in the plots below by the presence or absence of the model suffix "_base".

The effect of refining each embedding model while training the classifier is evaluated by deciding whether the gradient descent optimizer would optimize only the parameters from the classifier, or also the parameters in the embedding model. This is denoted in the plots below by the presence or absence of the word "refine".

An important feature of these objectives is whether they can be learned via global pooling information alone or if more intricate processing is necessary. For objectives such as *NumNegations, NumEqualities*, and to a slightly less extent *NumDistinctVars, Horn, and NumNodes*, it is plausible that a classifier could learn a lot from a simple sum of random symbol embeddings (shown in the below plots as *SimpleEmbedding*). For the *MaxDepth* objective, however, a simple sum will not suffice and slightly more complicated logic is required. A fully random embedding (*RandomEmbedding*) of clauses is also included as a blind baseline.

The figures below show the accuracy only on the test data (the given clauses from the 20% of problems not used for training) in an effort to improve visibility

in the figures. The models in the figure legends are sorted from most accurate to least accurate at the end of training (ties sometimes occur).

7.1.1 Horn

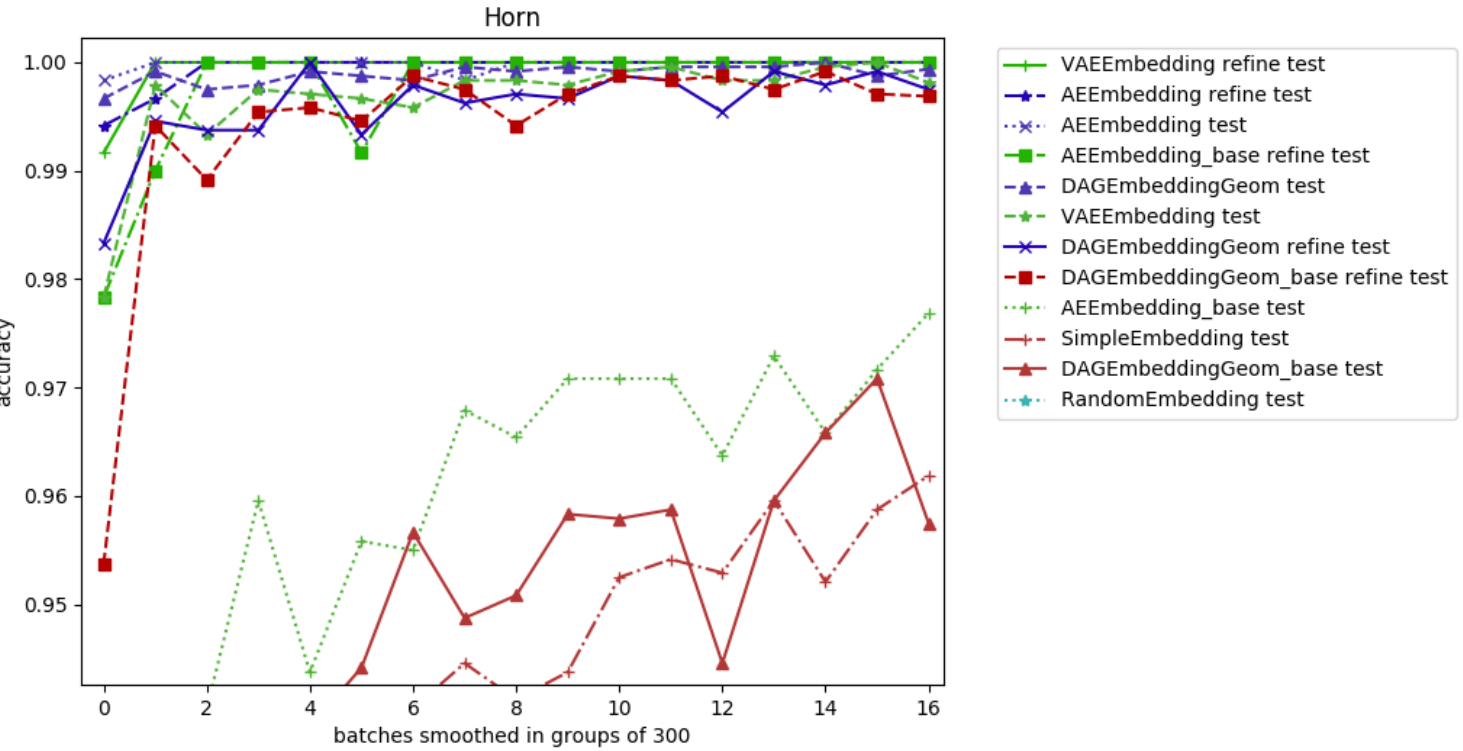

Figure 7.1: Accuracy for *Horn* explicit clause objective

Figure 7.1 shows the accuracy of classifiers trained atop each embedding for the *Horn* objective. The four best performing models overlap and are all tied for 100% accuracy with the next four models all obtaining accuracy above 99.5%. Overall, there are eight models that finish training with over 95% accuracy. The three visible but distinctly poorly-performing embedding models (still 96%-97%) were all randomly initialized and were not allowed to update during training. *SimpleEmbedding* and *RandomEmbedding* have no trainable parameters. The *RandomEmbedding* model is below the visible portion of this plot at approximately 50% accuracy.

7.1.2 NumNodes

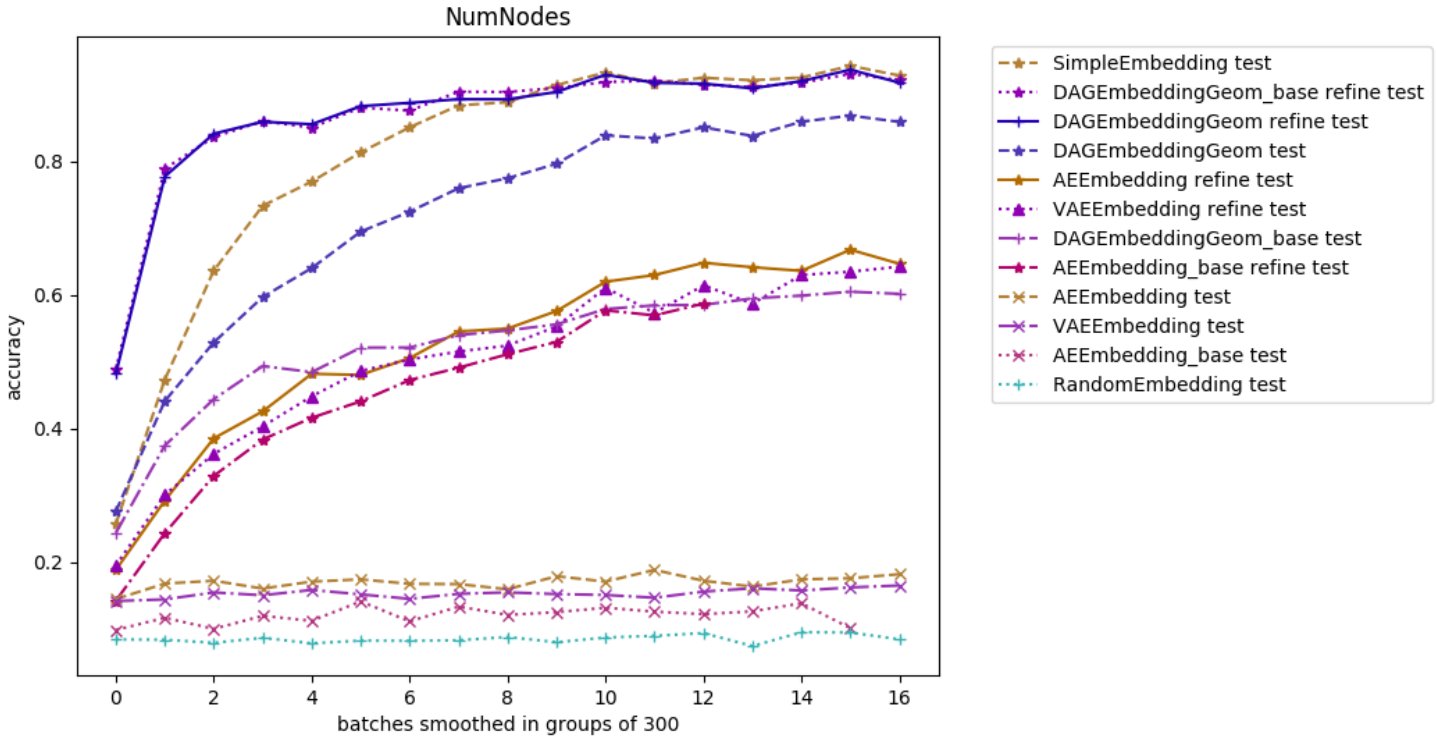

Figure 7.2: Accuracy for *NumNodes* explicit clause objective

Figure 7.2 shows the accuracy of classifiers trained atop each embedding for the *NumNodes* objective. The *SimpleEmbedding* is roughly tied with the both the pretrained and "_base" refined *DAGEmbeddingGeom* models at 93%. The main insight from this plot (and many of these other plots) is that while a simple symbol counting embedding can encode the total number of nodes in a DAG, embeddings (like the autoencoder embeddings) that obtain their final clause representation from the root node of a DAG fail to capture this information.

7.1.3 NumDistinctVars

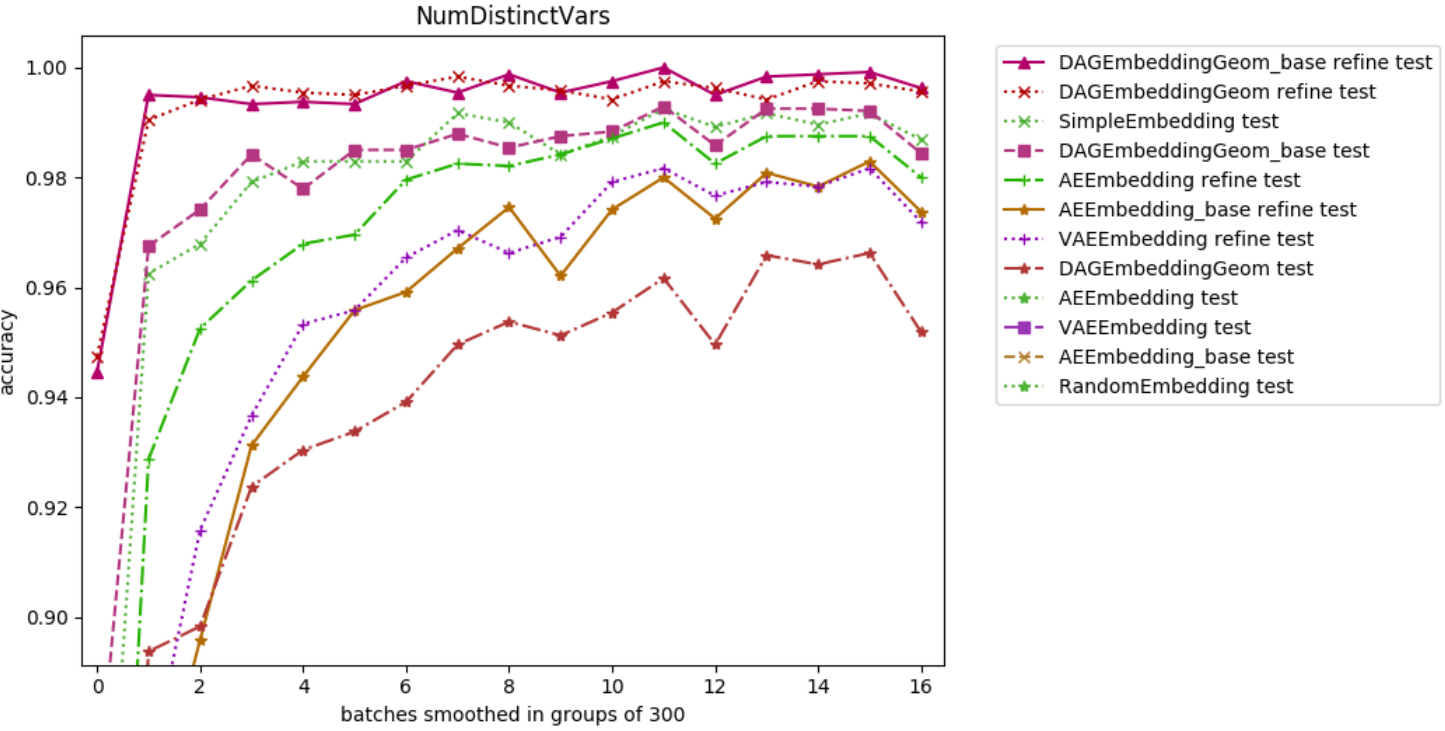

Figure 7.3: Accuracy for *NumDistinctVars* explicit clause objective

Figure 7.3 shows the accuracy of classifiers trained atop each embedding for the *NumDistinctVars* objective. The four best models for this objective are all *DAGEmbeddingGeom and SimpleEmbedding* models. Below the visible portion of this plot is *RandomEmbedding* at approximately 25% accuracy, and *VAEEmbedding, AEEmbedding_base, and AEEmbedding* at approximately 40% accuracy. This suggests that for this objective aggregation is again much more important than hierarchical processing.

7.1.4 NumEqualities

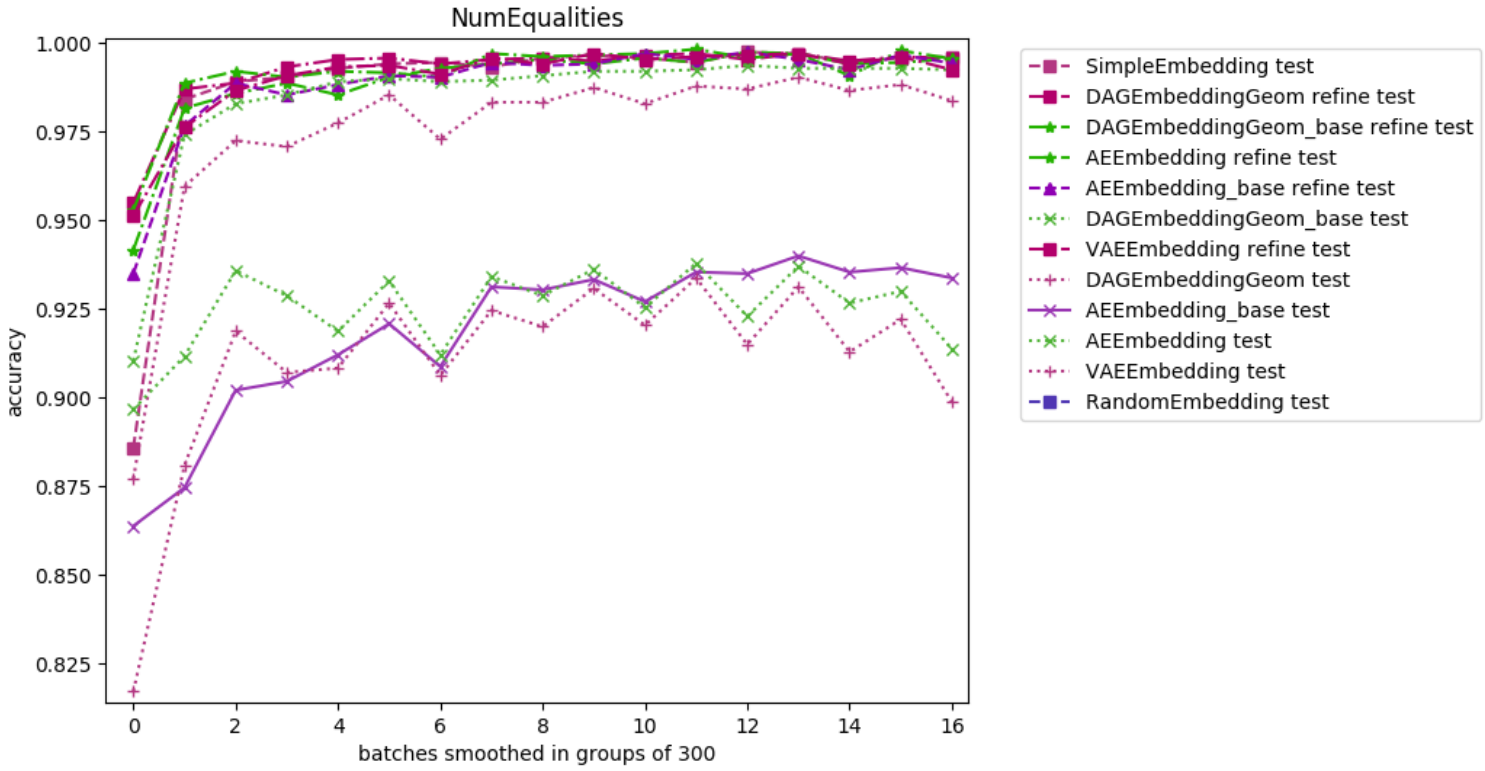

Figure 7.4: Accuracy for *NumEqualities* explicit clause objective

Figure 7.4 shows the accuracy of classifiers trained atop each embedding for the *NumEqualities* objective. The *SimpleEmbedding* does quite well at this task, probably because the sum of all random symbol vectors includes the sum of all "=" symbol vectors. The classifier can then learn to extract this objective via a simple dot product of the *SimpleEmbedding* output with the symbol vector for "=". The unrefined autoencoding models fail because information about "=" is always either at depth two or three in the DAG, and that information can easily be thrown away by a random initialization. Below the visible portion of this plot is the *RandomEmbedding* at approximately 59% accuracy.

7.1.5 NumNegations

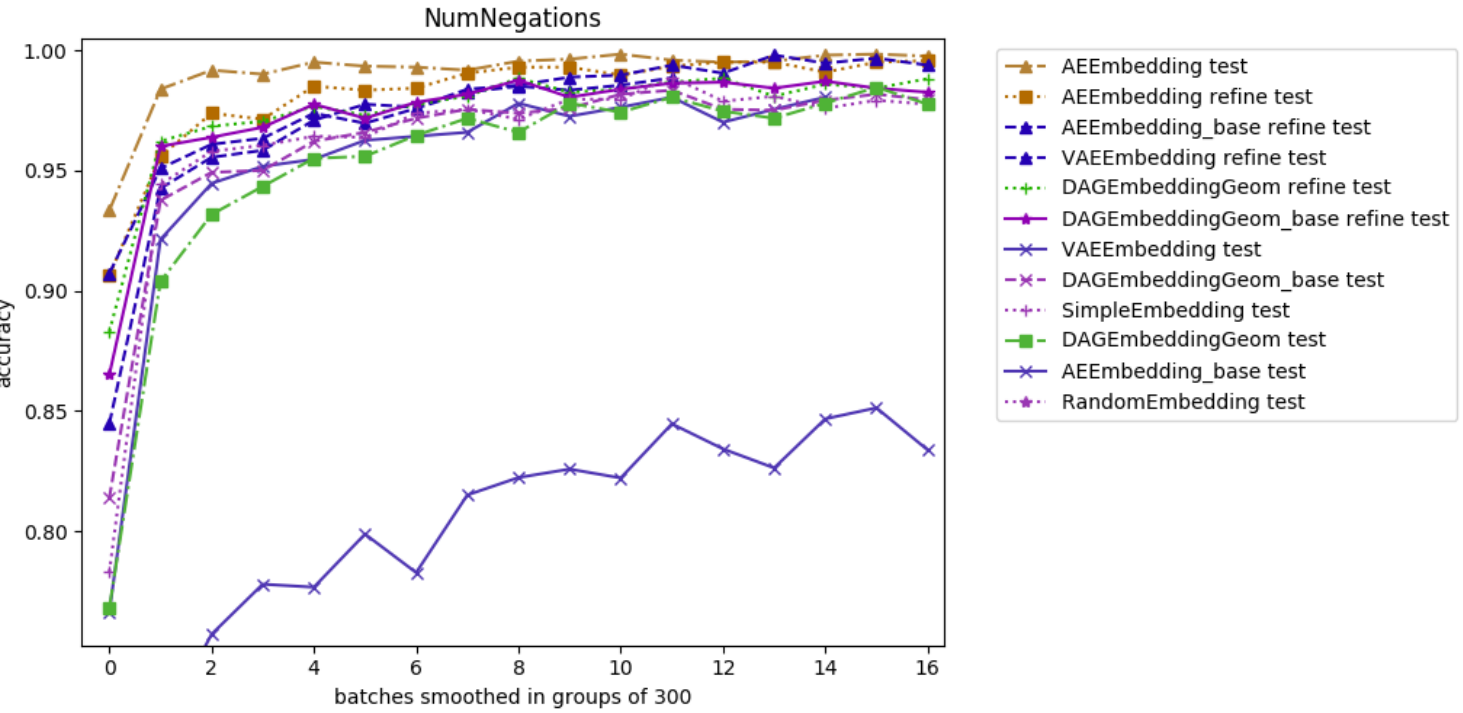

Figure 7.5: Accuracy for *NumNegations* explicit clause objective

Figure 7.5 shows the accuracy of classifiers trained atop each embedding for the *NumNegations* objective. Although *NumNegations* should be just as easy to learn as *NumEqualities* for *SimpleEmbedding*, *SimpleEmbedding* fails to outperform the other models. Perhaps this is because the other models are all easily able to incorporate the necessary information into their embeddings (all "∼" symbols occur as immediate children of the root), and because their additional parameters enable them to eliminate noise from other symbols. Below the visible portion of this plot is the *RandomEmbedding* at approximately 25% accuracy.

7.1.6 MaxDepth

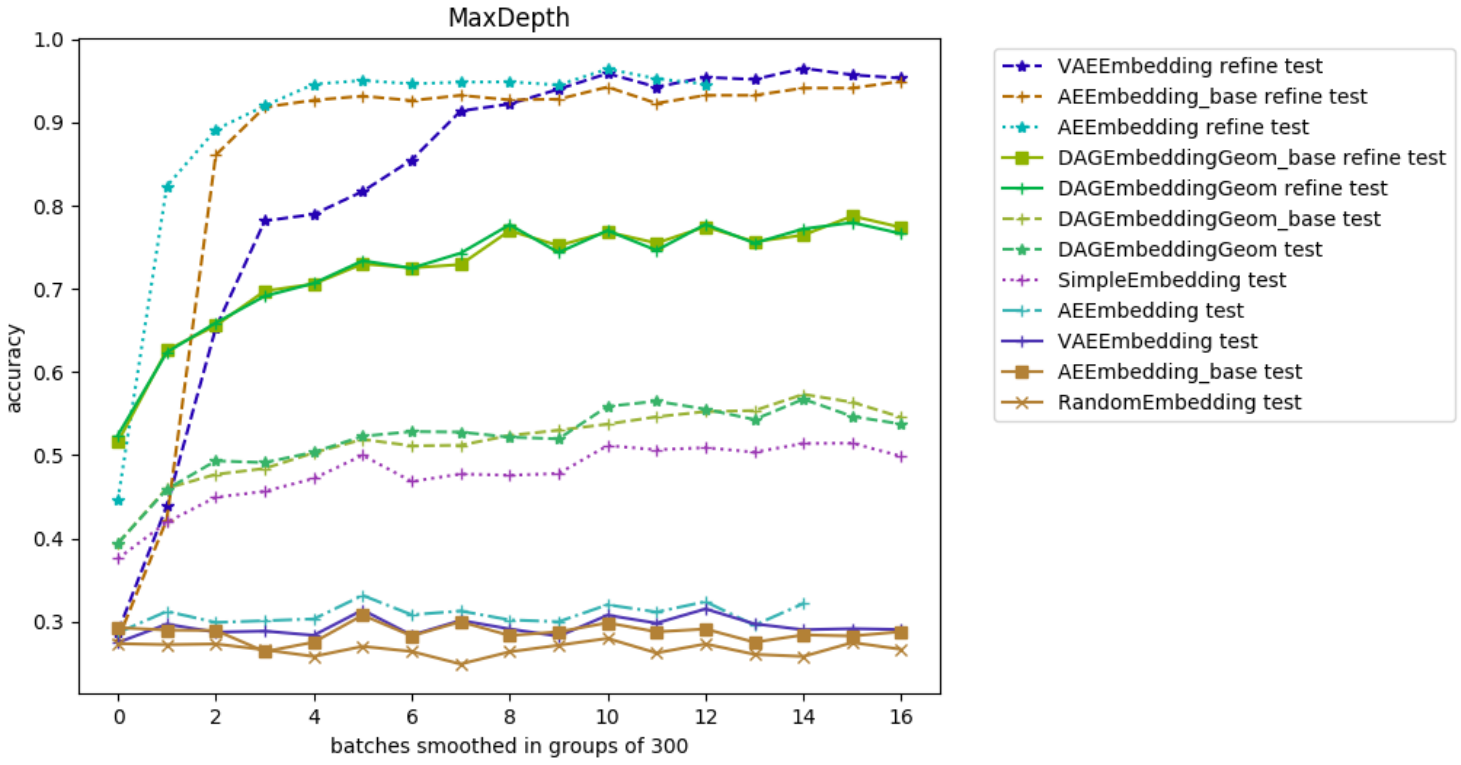

Figure 7.6: Accuracy for *MaxDepth* explicit clause objective

Figure 7.6 shows the accuracy of classifiers trained atop each embedding for the *MaxDepth* objective. Recall that the autoencoding models use the root node of the DAG as the representation of the DAG, as opposed to the max pooling used by the other models. In this plot, we see that the *MaxDepth* objective is best learned by the refined autoencoding models, and that the unrefined autoencoding models perform worst (on par with random embeddings at approximately 30%).

7.2 The Given Clause Selection Task

The four clause embedding models summarized at the beginning of this chapter were also evaluated for given clause selection. Figures 7.7 and 7.8 show the accuracy on the testing data described in Section 6.1.3. The plots are smoothed to show the rough outline of how training went for each model. Recall from Section 7.1 that the *SimpleEmbedding* model simply represents a DAG as a vector of symbol counts, and has no learnable parameters. It is provided here as a baseline. Recall that "refinement" refers to whether or not the embedding parameters were updated while training the classifier. Refinement for the "Custom Model" and "PyTorch Geometric GCN" models should have less of an effect because they were trained for given clause selection initially, whereas the autoencoder models are pretrained for reconstruction loss.

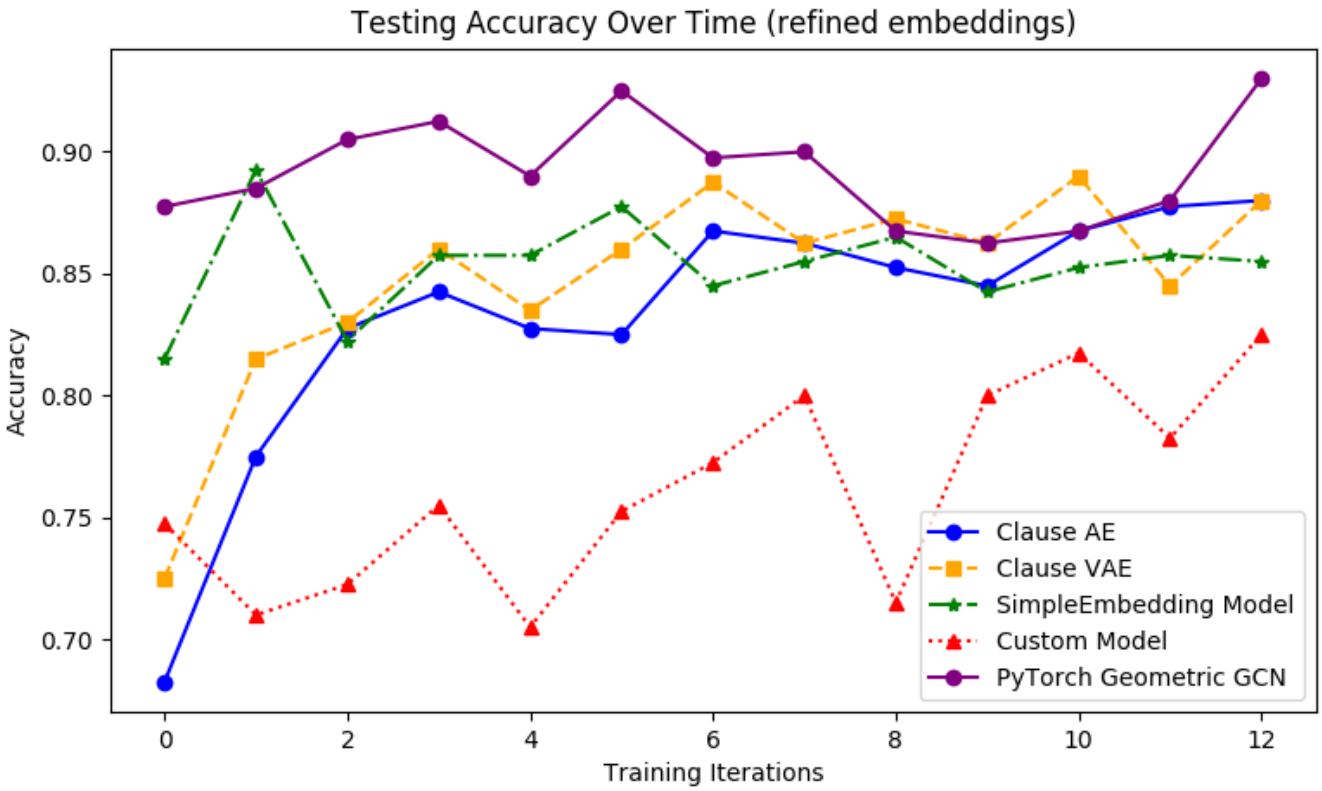

Figure 7.7: Accuracy of given clause selection classifiers when clause embeddings are refined during training.

Figure 7.7 shows the testing accuracies for the refined models. The PyTorch

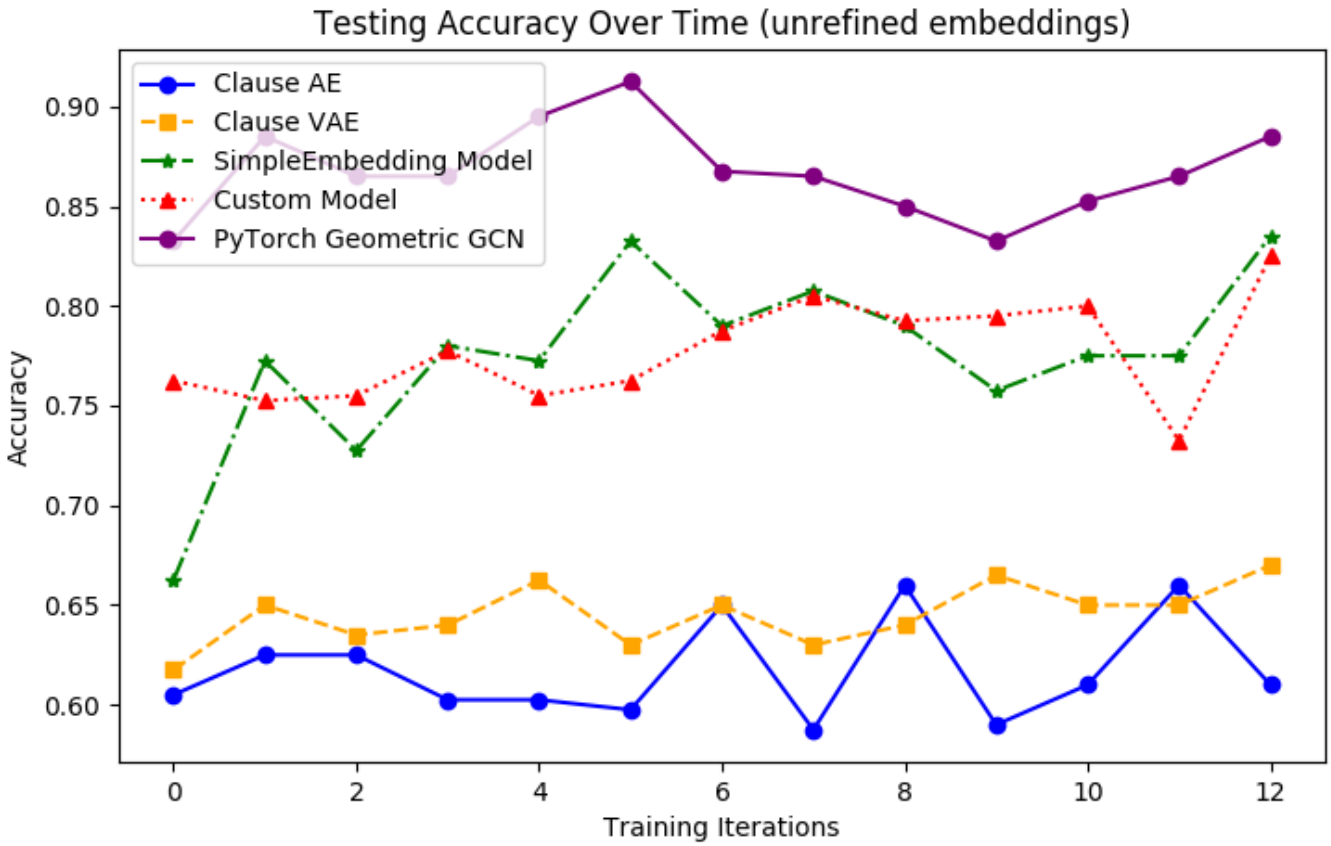

Figure 7.8: Accuracy of given clause selection classifiers when clause embeddings are not refined during training.

Geometric GCN model achieved the highest testing accuracy at roughly 92% accuracy at the end of training. The autoencoder models both achieved around 87% testing accuracy at the end of training with the variational autoencoder performing better for the majority of training.

Figure 7.8 shows the testing accuracies for the unrefined models. The PyTorch Geometric GCN model once again achieved the highest testing accuracy at roughly 86% at the end of training. The autoencoder embedding models clearly failed to generalize to the task of given clause selection.

Chapter 8

Conclusion & Future Work

Various attempts to represent clauses using neural networks have been shown in this work. Although these representations have not directly led to good heuristics for given clause selection, they have led to the following insights.

8.1 Simplicity is Key

The failure of more complex models suggests that perhaps simple models are already able to capture a large portion of the important information about a clause. Models that take into account only symbol occurrence frequencies seem to perform just as well as much more complex GNN models. Although it is unlikely that children ordering information is useless, it seems that the incorporation of such information is not easy for the models described and evaluated in this thesis.

8.2 Problem Formulation is Very Important

Even if a neural network model is well suited to processing the more nuanced features of clause DAGs, perhaps presenting given clause selection as a classification problem is too restricting. There are many questions that need to be answered when formulating given clause selection for machine learning. The answers to these questions could very well have a larger impact on the application to theorem

proving than the neural network architecture itself. In this work, many of these questions have been answered by default due to practical limitations, but they deserve a more thorough investigation. Some such questions are:

- Which clauses should the context include?
- Should the context clauses be static or change during training?
- Should (or how should) context clauses be combined into a single context vector?
- How should context affect the given clause selection?
- How should given clauses be labelled ($\{0,1\}$ labels only or continuous labels based on some measure of clause usefulness)?
- What forms of regularization of machine learning models are useful in this domain?

8.3 Clause Autoencoders

The failure of clause autoencoders to recursively encode and decode entire clauses, while disheartening, hints at a fundamental difficulty of the explored GNN models to assimilate information from far down in a DAG. For certain applications of GNNs, the short-range processing performed by typical GNN models is fine because a final pooling operation is enough to assimilate information from the entire graph. With logical formulae, however, the meanings of two subterms may drastically affect each other's meaning despite being far away from each other. Most importantly, they might affect each other in a way that is not easily learned by a global pooling operation. Despite the failure of clause autoencoders thus far, they may yet serve as a useful test for expressiveness of future GNN architectures.

8.4 Future Work

The above insights/hypotheses and others will be tested more thoroughly in the future work described below.

8.4.1 Iterated Axiom Selection

Neural network approaches for axiom selection benefit from parallel evaluation of all relevant clauses at once. This is an advantage of the axiom selection task over given clause selection (although, internally, E batches clauses as much as possible before evaluation).

One possible direction for future research is to modify E to only allow inferences among original problem clauses (no inferences involving previously inferred clauses). This version of E would fail to prove almost anything on its own. However, clauses inferred by E during one invocation would be used (along with the original problem clauses) as initial problem clauses for another invocation of E. Each time E is invoked, custom selection models would be used.

A normal run of E never forgets clauses that it has inferred unless they are subsumed by newer inferred clauses. The proposed version of E would forget those clauses inferred by earlier invocations of E that it deems unimportant via selection at each new invocation. Although this forgetting makes the theorem prover incomplete, it may be beneficial for finding proofs in practice.

8.4.2 Robust Training

Treating given clause selection as a classification task is perhaps questionable. In reality, given clause selection should be thought of as ordering given clauses relative to the "promise" they show (for being part of a proof) relative to the context clauses. Recall from Chapter 6 that the training examples were taken from successful runs of E, where the negative examples were clauses that didn't end up in the proof, and the positive examples were clauses that did end up in the

proof.

If a clause ends up in the proof, then it was probably a good choice. However, maybe there was another clause that could have been selected instead that would have also led to a proof. Likewise, clauses which don't end up in the proof are not necessarily bad choices, as they might have ended up in an alternative proof. Because of this, it seems that making neural nets for given clause selection robust to label noise could be quite beneficial.

When viewing given clause selection as the task of assessing the "promise" of a given clause, clauses should be given labels from zero *to* one instead of simply zero *or* one. One possible way of doing this is to come up with multiple distinct proofs for each problem, *prob*, and label each given clause, c, with the label $\frac{proof(c, prob)}{selected(c, prob)}$, where $proof(c, prob)$ is the number of proofs of problem *prob* that clause c was used in and $selected(c, prob)$ is the number of proofs of problem *prob* where clause c was selected by some E heuristic. In order to do this, one must be able to tell if two clauses are semantically equivalent. The appendix contains an "Aside on DAG hashing" which is useful for this. It has been generally useful for maintaining sets of clause DAGs.

8.4.3 Improved Tools

Iterating on ideas quickly is very important for research. Thus far, this research has been quite slow for a number of reasons. Interfacing with E requires writing C code for a custom heuristic that can be compiled along with E. As nice as this interface is in retrospect, learning it still took a bit of time and effort which could have been focused on research ideas. Many machine learning libraries are available only (or most convenient to use) in Python. This leads to more difficulties in having a fast and reliable connection between C and Python. Because of this, future research includes a fast and language-agnostic protocol for interfacing heuristic code with E.

Appendix A

Appendix

A.1 Aside on DAG Hashing

It is sometimes useful to consider sets of clauses or to consider hash maps with clauses as keys. Towards this goal, it is desirable to have a hash function on clauses which, barring hash collisions, guarantees that two clauses hash to the same value if and only if they are logically equivalent.

All possible representations of a clause can be generated by sets of permutations. Any permutations of a clause's literals is logically equivalent. Also, the swapping of an equality literal's terms does not change the meaning of that equality literal. Variable names also don't matter. Ignoring variable names is tricky, however, because although it is unimportant that two variables, X and Y, are specifically named "X" and "Y", it is important that they are different. For instance, it is important to retain the ability do distinguish between $p(X, Y, Z)$ and $p(X, Y, X)$.

Although I have no proof of correctness for the following DAG hashing procedure, it has been tested on many randomly sampled clauses and has remained consistent across all tests. Python provides basic hashing functions on sets (frozenset), tuples, and integers. These are utilized throughout the algorithm wherever hashing is mentioned as a subroutine.

Throughout the algorithm, an integer value is maintained and updated for each node. These values are collectively called the "hash cache." At the end of the algorithm, the hash of the DAG is given by the hash cache value of the root node. The algorithm has two phases: the "bottom-up" and "top-down" phases. Intuitively, during the bottom-up phase, each node summarizes its descendants and during the top-down phase, each node summarizes its ancestors. The bottom-up phase proceeds first, followed by the top-down phase, followed by one more bottom-up phase. As the name suggests, the bottom-up phase visits nodes from the bottom to the top of the DAG. More formally, it proceeds in a reversed topological sort of the DAG nodes so that all descendants of a node are visited before the node itself. The top-down phase then proceeds in the reverse order so that all ancestors of node will be visited before the node itself.

When a leaf is visited during the **bottom-up** phase, its hash cache value is set to the hash of its symbol number if the leaf node represents a constant and the hash of zero if it represents a variable.

When a non-commutative internal node is visited during the **bottom-up** phase, its hash cache value is set to the following: $hash((symbol, childHashes))$ where $symbol$ is the symbol of the node and $childHashes$ is an ordered tuple of the hash cache values for a node's children in the DAG. Since $childHashes$ is an ordered tuple, Python's hashing ensures that if the children of this node were in a different order, the node would receive a different hash cache value.

When a commutative internal node is visited during the **bottom-up** phase, its hash cache value is set to the following: $hash((symbol, frozenset(childHashes)))$. Since Python enables hashing of sets in this way, the hash cache value of this node will be the same regardless of the ordering of its children.

A single bottom-up phase would be enough to produce a DAG hash if it wasn't necessary to distinguish between pairs such as $p(X, Y, Z)$ and $p(X, X, Y)$. Since the hash cache values of variables are all equal, they can only be distinguished by

their context, which is encorporated via the top-down phase.

When any node is visited during the **top-down** phase, its hash cache value is set to the hash of a set of tuples: one tuple for each of the node's parents. These tuples are of the form (n, ph, h), where ph is the hash cache value for the parent node, h is the hash cache value for the node itself, and n indicates ordering information. For instance, $n = 2$ for instance would mean that the node is the 3rd (0-indexed) child of that particular parent. If the parent node is a commutative-node, then $n = -1$.

Now that variables have been given distinct hash cache values based on their context, that information must be propogated back upward to the root to give us the final DAG hash. This is accomplished via a final bottom-up phase. This proceeds in the same manner as before, except that leaf hash cache values are left untouched instead of being set as before.

As mentioned before, the final hash value of the DAG is the hash cache value of the root node after this second bottom-up phase.

Bibliography

[1] S. Schulz, S. Cruanes, and P. Vukmirovic. Faster, Higher, Stronger: E 2.3. In *Proceedings of the 27th International Conference on Automated Deduction*, number 11716 in Lecture Notes in Computer Science, pages 495–507. Springer-Verlag, 2019.

[2] S. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158. Association for Computing Machinery, 1971.

[3] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic Computation*, 4(3):217–247, 1994.

[4] B. Russell. Mathematical Logic as Based on the Theory of Types. *American Journal of Mathematics*, 30:222.

[5] L. Kovacs and A. Voronkov. First-Order Theorem Proving and Vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification*, number 8044 in Lecture Notes in Artificial Intelligence, pages 1–35. Springer-Verlag, 2013.

[6] M. Gori, G. Monfardini, and F. Scarselli. A New Model for Learning in Graph Domains. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, pages 729–734 vol. 2, 2005.

[7] S. Banach. Sur Les Opérations Dans Les Ensembles Abstraits et Leur Application aux Equations Intégrales. *Fundamenta Mathematicae*, 3(1):133–181, 1922.

[8] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated Graph Sequence Neural Networks, 2017.

[9] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral Networks and Locally Connected Networks on Graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[10] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. *CoRR*, abs/1606.09375, 2016.

[11] T. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks, 2017.

[12] J. Gilmer, S. Schoenholz, P. Riley, O. Vinyals, and G. Dahl. Neural Message Passing for Quantum Chemistry. *CoRR*, abs/1704.01212, 2017.

[13] C. Kaliszyk, F. Chollet, and C. Szegedy. HolStep: A Machine Learning Dataset for Higher-order Logic Theorem Proving. *CoRR*, abs/1703.00426, 2017.

[14] C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *Journal of Automated Reasoning*, 55, 10 2013.

[15] M. Crouse, I. Abdelaziz, C. Cornelio, V. Thost, L. Wu, K. Forbus, and A. Fokoue. Improving Graph Neural Network Representations of Logical Formulae with Subgraph Pooling, 2020.

[16] K. Chvalovsky, J. Jakubuv, M. Suda, and J. Urban. ENIGMA-NG: Efficient Neural and Gradient-Boosted Inference Guidance for E. In *Proceedings of*

the 27th International Conference on Automated Deduction, number 11716 in Lecture Notes in Computer Science, pages 197–215. Springer-Verlag, 2019.

[17] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[18] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[19] G. Bancerek, C. Bylinski, A. Grabowski, A. Kornilowicz, R. Matuszewski, A. Naumowicz, K. Pak, and J. Urban. Mizar: State-of-the-art and Beyond. In *Intelligent Computer Mathematics - International Conference, CICM July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 261–279. Springer, 2015.

[20] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[21] J. Alama, D. Kühlwein, E. Tsivtsivadze, J. Urban, and T. Heskes. Premise Selection for Mathematics by Corpus Analysis and Kernel Methods. *CoRR*, abs/1108.3446, 2011.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[23] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, et al. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[24] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks, 2018.

[25] T. Kipf and M. Welling. Variational Graph Auto-Encoders, 2016.

[26] S. PurgaŁ, J. Parsert, and C. Kaliszyk. A Study of Continuous Vector Representations for Theorem Proving. *Journal of Logic and Computation*, Feb 2021.