Saarland University
Faculty of Mathematics and Computer Science

Master's Thesis

NeuRes: A Neural Resolution Prover of Unsatisfiability

Author
Mohamed Abdelhamid Ghanem

Supervisor
Prof. Bernd Finkbeiner, PhD

Advisors
Frederik Schmitt
Julian Siber

Reviewers
Prof. Bernd Finkbeiner, PhD
Dr. Christopher Hahn

Submitted: 24th January 2024

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 24th January, 2024

Erklärung

Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken, 24th January, 2024

Abstract

We introduce NeuRes, a neuro-symbolic generative model for proving Boolean unsatisfiability using resolution. A resolution proof is a sequence of case distinctions ending in the empty clause (falsum). Similar propositional logic tasks have proven fertile grounds for neuro-symbolic methods such as NeuroSAT. However, these methods often lack easily verifiable certificates for unsatisfiability to support their predictions, whereby verifying their output effectively requires solving the satisfiability problem again. In contrast, resolution proofs produced by NeuRes provide an easily checkable certificate for unsatisfiability. We introduce a general architecture that adapts elements from Graph Neural Networks and Pointer Networks to autoregressively select pairs of nodes from a dynamic graph structure. Our model is trained and evaluated on a dataset of expert proofs that we compiled with the same formula distribution used by NeuroSAT. Compared to previous methods, we demonstrate NeuRes to be more data efficient, requiring only 8K training formulas to concisely prove unsatisfiability for 92.84% of unseen test formulas. In addition to the high success rate of our model, we further demonstrate its ability to largely shorten teacher proofs in a bootstrapped fashion with no extra guidance.

Acknowledgements

I would like to express my gratitude to Frederik Schmitt and Julian Siber for their insightful discussions and remarks over the course of this thesis. Also, I would like to give special thanks to Frederik for his amazing support on this project on technical and formal logistics. My thanks also go to Dr. Christopher Hahn for reviewing my thesis. Last but not least, I would like to thank Prof. Finkbeiner for giving me the opportunity to do my thesis in the Reactive Systems group, and for showing great support and interest in this project which I found deeply motivating.

Contents

Abstract **v**

1 Introduction **1**

2 Related Work **4**

3 Preliminaries **6**
3.1 SAT 6
3.2 Resolution 8
3.3 Graph Neural Networks 9

4 Models **11**
4.1 General Architecture 11
4.2 Message-Passing Embedder 12
4.3 Encoder-Decoder Network 14
4.4 Selector Networks 16
4.4.1 Cascaded Pointer-Attention (Casc-Attn) 16
4.4.2 Full Self-Attention (Full-Attn) 17
4.4.3 Anchored Self-Attention (Anch-Attn) 18

5 Training and Hyperparameters **21**
5.1 Dataset 21
5.2 Loss Function 21
5.3 Proof-Reduction Bootstrapping 22
5.4 Hyperparameters 22

6 Experiments **24**
6.1 Attention Variants 25
6.2 Out-Of-Distribution Performance 26
6.3 Number of Message-Passing Rounds 27
6.4 Shortening Teacher Proofs 28

6.5 Bootstrapped Training 30

7 Conclusion 33

A Appendix 35
A.1 Constant vs. Linearly Annealed Learning Rate 35
A.2 Network Parameters 36

Bibliography 39

Chapter 1

Introduction

Boolean satisfiability (SAT) is a fundamental problem in computer science. For theory, this stems from SAT being the first problem proven to be NP-complete [8]. For practice, this is due to many highly optimized SAT solvers used as flexible reasoning engines in many practical tasks such as hardware verification [5]. Recently, SAT has also served as a litmus test for assessing the symbolic reasoning capabilities of neural models, with neuro-symbolic models such as NeuroSAT [27] generalizing to the semantics of propositional logic, and its predictions even leading to considerable speed-ups in industrial-strength solvers [26]. Since SAT solvers are often employed where correctness is critical and are complex systems with a documented history of bugs [6, 17], a key problem has long been providing certificates that verify their outputs. This is even more crucial for neural solvers because, unlike algorithmic solvers, we do not currently possess the means to prove their correctness. For neural models, certification[1] provides an opportunity to build a sound-by-design solver: Combining neural prediction with a cheap algorithmic check of the produced solution has, for instance, been proposed for circuit synthesis [24]. Figure 1.1 outlines the ideal neural solver in the sense that it follows its SAT verdict with a certificate. In this work, we focus on designing a neural model capable of generating such certificates for unsatisfiable formulas of propositional logic.

For a satisfiable formula, the witnessing variable assignment is an easily verifiable certificate that classical solvers and even NeuroSAT can produce. For an unsatisfiable formula, however, both classical solvers and NeuroSAT struggle with providing a comparably useful certificate. We refer to the annual SAT competitions [4] for a comprehensive overview on the ever-evolving landscape of SAT solvers, benchmarks, and proof checkers. Notably, certificates for proofs of unsatisfiability have been partially required in this competition since 2013 [3]. The first approaches

[1]We use the term ***certification*** here in the context of outputting witnesses for SAT predictions, not in the sense of machine learning certification, where the models themselves, not the outputs, are certified.

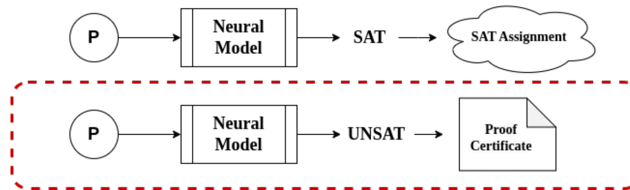


Figure 1.1: Ideal Neural Solver

aimed at certification of unsatisfiability were based on resolution [33, 13]. The main appeal to resolution proofs lies in their simplicity and low verification cost [10]. However, it is non-trivial to output resolution proofs from modern solvers which are almost universally based on the paradigm of conflict-driven clause learning [21] and employ complex preprocessing and inprocessing techniques that cannot always be mapped neatly to resolution steps. Moreover, resolution proofs can get quite large to the point where they require rather huge amounts of disk space.

The solution for the SAT-solving community has been to resort to the more relaxed notion of clausal proofs, i.e., sequences of formulas where a formula at a given index has to be implied by the original formula unified with all formulas at preceding indices. The most common format to represent clausal proofs is arguably the DRAT format [31]. This format is able to capture most of the optimizations employed by modern SAT solvers and is particularly storage-efficient. However, this comes at a cost in complexity of verifying a DRAT certificate, which can take longer than proof discovery. This complexity has practical ramifications on its human interpretability and interestingly, the standard tool for DRAT checking, drat-trim, has been shown to have bugs before [19].

Towards proving unsatisfiability, the neural model NeuroSAT only provides an unsatisfiable core, i.e., a subset of formulas that are already unsatisfiable. Checking the correctness of both clausal proofs and unsatisfiable cores requires again solving SAT problems and may often take longer than the original proof discovery [16].

Given that NeuroSAT was only trained on predicting satisfiability and neural models are not as fundamentally barred from reproducing resolution proofs as industrial-strength solvers, we study the following question in this thesis: **Can neural models learn to produce resolution proofs as easily checkable certificates of unsatisfiability?** We answer this question affirmatively and present a novel architecture drawing inspiration from the Graph Neural Network approach of NeuroSAT and the selection mechanism of pointer networks. After only being trained on a relatively small number of formulas and expert proofs, the resulting model, NeuRes, is able to produce resolution proofs of comparable size for 92.84% of unseen test formulas.

Our contributions. In this thesis, we make the following main contributions:

1. We introduce three novel neural architectures for generating resolution proofs of unsatisfiability for CNF formulas. We then evaluate them in terms of success rate and optimality to determine the most effective variant.

2. We show our model's ability to largely shorten expert proofs given by an advanced solver with no external guidance. In essence, we show that by simply training on solver proofs, our model is able to learn novel insights into the redundancy involved in those proofs and ultimately shortcut them.

3. Having established such reduction capabilities, we devise a bootstrapped training procedure where our model progressively becomes its own teacher which further boosts its success rate and optimality beyond those obtained by regular teacher-forcing. We further showcase the potency of this procedure to shorten a dataset of resolution proofs by simply training on it – a feature that stands great potential value in alleviating the storage constraints of resolution proofs.

Chapter 2

Related Work

Supervised Neural SAT Solvers. NeuroSAT [27] was the first study of the Boolean satisfiability problem as an end-to-end learning problem. Building upon the NeuroSAT architecture, [26] trained a simplified version to predict unsatisfiable cores and successfully integrated it in a state-of-the-art SAT solver. [7] showed that both the NeuroSAT architecture and a newly introduced deep exchangeable architecture can outperform SAT solvers on instances of 3-SAT problems and be trained to additionally provide a satisfying assignment. The NeuroSAT architecture has also been applied on special classes of crypto-analysis problems [28].

Unsupervised Neural SAT Solvers. In addition to supervised learning, unsupervised methods have been proposed for solving propositional satisfiability problems with deep learning. For Circuit-SAT a deep-gated DAG recursive neural architecture has been proposed together with a differentiable training objective to optimize towards solving the Circuit-SAT problem and finding a satisfying assignment [2]. For Boolean satisfiability, a differentiable training objective has been proposed together with a query mechanism that allows for recurrent solution trials [22].

Deep learning has been further used in the deep generative neural network G2SAT for generating SAT formulas [32].

Certificates in Neuro-symbolic Computing. Below are some examples of prior work on training neural networks on formally verifiable certificates:

- **Symbolic integration in mathematics**: [20] define a syntax to represent symbolic integration and differential equation problems in mathematics and train a sequence-to-sequence neural network to produce step-by-step solutions for these tasks.

- **Proof generation/repair**: [23] explored the use of transformer-based large language models (LLMs) for automated theorem proving where they managed to produce new concise proofs that were accepted into the main Meta-

math library. [12] utilizes LLMs to generate whole proofs at once (not step by step) and then use a fine-tuned repair model to fix the generated proofs.

Chapter 3

Preliminaries

3.1 SAT

The satisfiability problem, often abbreviated as SAT, is a classical problem in computer science and mathematical logic. The problem is to determine whether a given logical formula, expressed in Boolean logic, can ever be true (satisfied) by some assignment of truth values to its variables. In the following, we cover the relevant concepts around SAT.

A Boolean formula is an expression constructed using Boolean variables joined by logical connectives. For example, given Boolean variables A, B, C, we can construct a Boolean formula as: $(A \lor B) \land (B \lor \neg C)$.

Syntax of Boolean Formulas

1. **Boolean variables** are symbolic tokens that can take the truth value of either be true (1, $\top$) or false (0, $\bot$). A Boolean *literal* is a variable occurrence either positively (e.g., A) or negatively (e.g., $\neg A$).
2. **Connectives**: A set of logical operators, typically including conjunction ($\land$), disjunction ($\lor$), and negation ($\neg$).
3. **Formation rules**: A Boolean formula is defined recursively as follows:
    - The empty formula with no variables represents falsehood ($\bot$).
    - A unit Boolean formula only contains a single Boolean variable.
    - If F and G are Boolean formulas, then $(F \land G)$, $(F \lor G)$, $\neg F$, and $\neg G$ are Boolean formulas.

A truth assignment (also known as an interpretation) is a function $I : V \to \{\bot, \top\}$ where V is the set of Boolean variables.

The truth value of a boolean formula is recursively defined:

- The truth value of a variable $v_i \in V$ is $I(v_i)$.eit

- $I(F \wedge G) = \top$ if $I(F) = I(G) = \top$, and $\bot$ otherwise.
- $I(F \vee G) = \bot$ if $I(F) = I(G) = \bot$, and $\top$ otherwise.
- $I(\neg F) = \bot$ if $I(F) = \top$, and $\top$ otherwise.

There are multiple standard forms, termed *normal forms*, to express Boolean formulas, and they are all pairwise convertible. The two most common normal forms are arguably the Conjunctive Normal Form (CNF) and the Disjunctive Normal Form (DNF). CNF formulas are expressed as a conjunction of disjunctions (commonly referred to as ***clauses***) as follows:

$$\bigwedge_{i=1}^{k} \left(\bigvee_{j=1}^{m_i} L_{ij} \right)$$

where L_{ij} is the j-th literal in the i-th clause, whose length is m_i.

DNF formulas are a disjunction of conjunctions compactly expressed as:

$$\bigvee_{i=1}^{k} \left(\bigwedge_{j=1}^{m_i} L_{ij} \right)$$

As such, to satisfy a CNF formula, a variable assignment should satisfy every single clause in the formula while for a DNF formula, it suffices to satisfy one conjunctive clause. Any Boolean formula can be converted to an *equivalent* formula in either formats [11]. However, the conversion can result in a formula size that is exponential in the size of the original formula. That being said, for the purposes of determining satisfiability, we need not worry about *equivalence* but rather *equisatisfiability*. Two formulas F and G are equisatisfiable when F is satisfiable if and only if G is satisfiable. Fortunately, any Boolean formula can be converted to an equisatisfiable CNF formula in polynomial time using the Tseytin transformation [29]. No similar polynomial-time reduction has been found for DNF formulas. Therefore, we operate on CNF formulas.

3.2 Resolution

Resolution reasons about propositional formulas in **clausal normal form**, e.g., $(x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_3)$. As previously explained, such a formula is called **satisfiable** if there is an assignment to the Boolean variables that renders it **true**, and it is called **unsatisfiable** otherwise. We will use a simplified notation of such formulas as sets of sets, e.g., $\{\{1, 2, -4\}, \{-1, 3\}\}$. An inner set such as $\{-1, 3\}$, called a **clause**, corresponds to a disjunction of **literals**. Literals are variables, such as 3 (for x_3) or their negation, such as -1 (for $\neg x_1$). The resolution rule picks clauses with two complementary literals and performs the following inference:

$$\frac{C_1 \cup \{x\} \qquad C_2 \cup \{\bar{x}\}}{C_1 \cup C_2} \; \mathit{Res}$$

Resolution effectively performs a case distinction on the value of variable x: Either it is assigned to *false*, then C_1 has to evaluate to *true*, or it is assigned to *true*, then C_2 has to evaluate to *true*. Hence, we may infer the clause $C_1 \cup C_2$. A **resolution proof of unsatisfiability** for a formula F is a sequence of applications of the resolution rule **Res** to pairs of clauses that either appear in F or were inferred in previous applications of **Res**, and that ends with the empty clause. Proof systems based only on propositional resolution are already complete [15]: applying the resolution rule until no longer possible implies satisfiability if the empty clause is not inferred, and unsatisfiability otherwise. They are also evidently sound based on the soundness of the resolution rule itself.

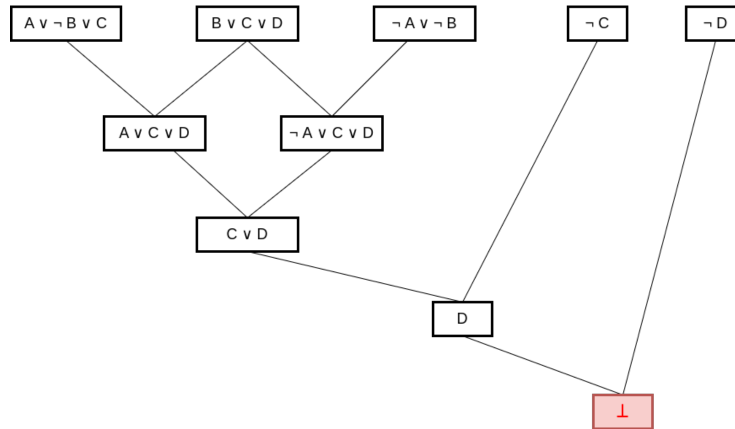


Figure 3.1: Resolution Tree Example

Figure 3.1 shows an example of a resolution proof tree. For the scope of this thesis, we only use binary resolution although our architecture can, in principle, be extended to cover generalized resolution (involving more than two clauses) as well.

3.3 Graph Neural Networks

Graph Neural Networks (GNNs) are a family of neural networks designed to handle graph-structured data. In theory, data entities are featurized by GNNs similarly to regular feed-forward networks; it is the arbitrary nature of the relationships between those entities that prompts the use of GNNs.

Graph representation

In graph structures, data entities are referred to as nodes, and the relationships between nodes are represented by edges. For instance, nodes can represent individuals in a social network where edges represent personal connections between them.

One challenge that graphs pose for neural models is that they are of arbitrary size, and they cannot be standardized into data structures of fixed dimensions like images or matrices[1]. To address this challenge, the key concept of *message passing* protocols in GNNs is devised. These protocols allow the network to capture neighbourhood information for each node into a local feature vector, often referred to as the *node embedding*. Figure 3.2 shows an example of one message passing round in action. During the round k, neighbouring nodes exchange their current embedding state vectors h_i^k, then each nodes aggregates the incoming embedding vectors into its own to produce its updated embedding vector to be used in the next round.

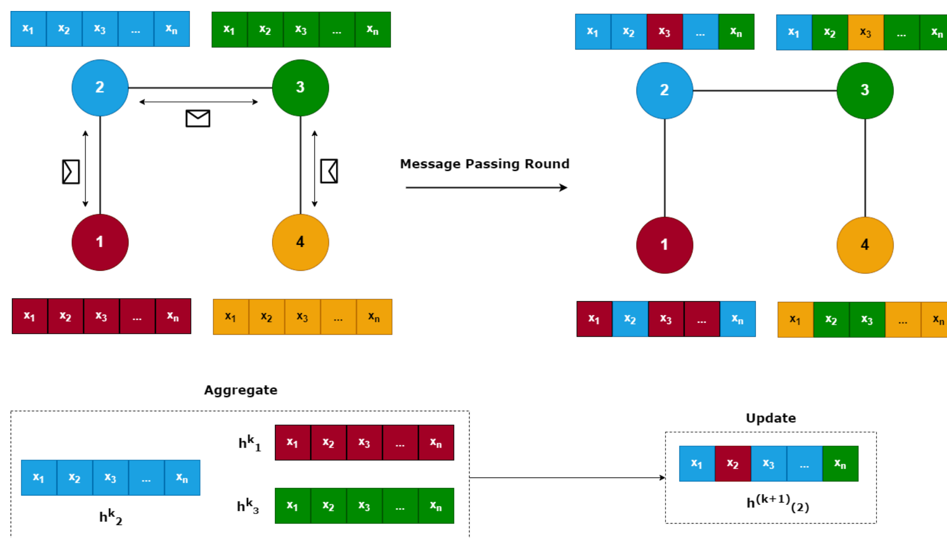


Figure 3.2: Message Passing Embedding

[1]Even adjacency matrices will be of variable dimensions based on the number of nodes.

The aggregation function is typically a neural network that gets trained as part of the larger GNN model, and can be formally expressed in the update rule as:

$$h_v^{(k+1)} = \mathcal{F}\left(\{h_u^{(k)} \mid u \in \mathcal{N}(v)\}\right) \tag{3.1}$$

where:

- $h_v^{kl)}$ is the representation of node v at round k,
- $\mathcal{N}(v)$ is the set of neighbors of node v,
- $\mathcal{F}$ is the aggregation function, which can be a sum, mean, max, or a neural operation.

There are several message passing protocols in the literature such as *Structure2Vec* [9] and *GraphSAGE* [14]. These embedding techniques provide a neural representation for input graphs that can be operated on by the later stages of the network. After the embedding phase, GNN architectures start to differentiate themselves based on their target objective/task. These tasks include, but are not limited to:

- **Node classification**: e.g., labelling malicious individuals on a social network
- **Graph classification**: e.g., whether a given graph is a clique
- **Graph clustering**: e.g., partition a social network into friend circles
- A combination of the above

In this thesis, we employ a GNN to embed CNF formulas in form of a set of clause embeddings, which we then use to make selections over clauses or pairs of clauses for resolution. The details of the formula graph construction and embedding will be discussed in Section 4.2.

Chapter 4

Models

Our approach is inspired mostly by two works in the neuro-symbolic literature, namely NeuroSAT [27] for Boolean formula embedding and Pointer Networks (PtrNet) [30] for clause selection. However, our setting and objective are fundamentally different from both models. Firstly, while NeuroSAT predicts satisfiability with no certificate, our model proves unsatisfiability by generating a certificate. As such, NeuroSAT is a classifier while NeuRes is a generative network. Secondly, PtrNet can only select from the set of input tokens while NeuRes can select between pairs (and possibly tuples) of input tokens. We present several variations of NeuRes in our study, which we describe in this section. We start by outlining the overall architecture shared by all variants in Section 4.1. The main difference between our variants is the selection mechanism for choosing a pair of input tokens. We present these variations in Section 4.4.

4.1 General Architecture

NeuRes is a neural network that takes an unsatisfiable Boolean CNF formula as a set of clauses and outputs an unsatisfiability proof as a sequence of resolution steps

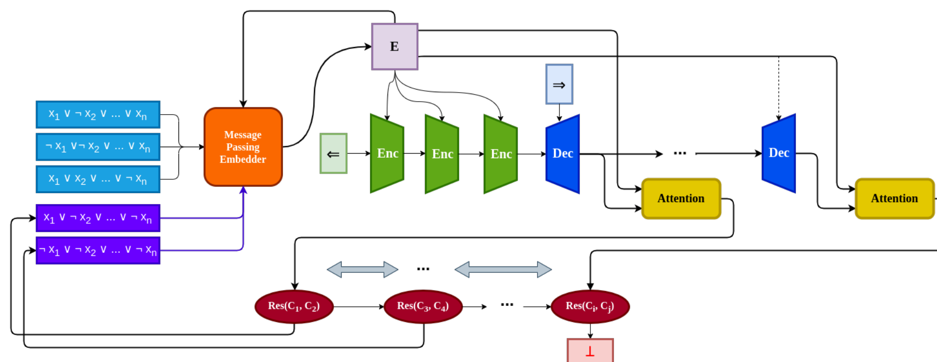


Figure 4.1: Overall NeuRes architecture

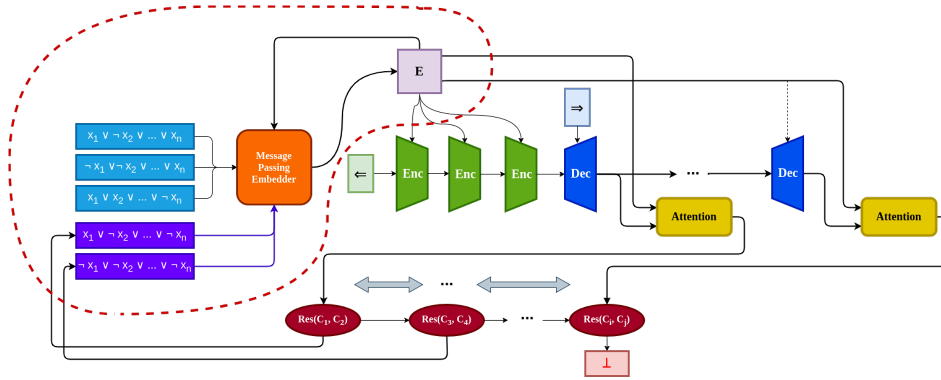
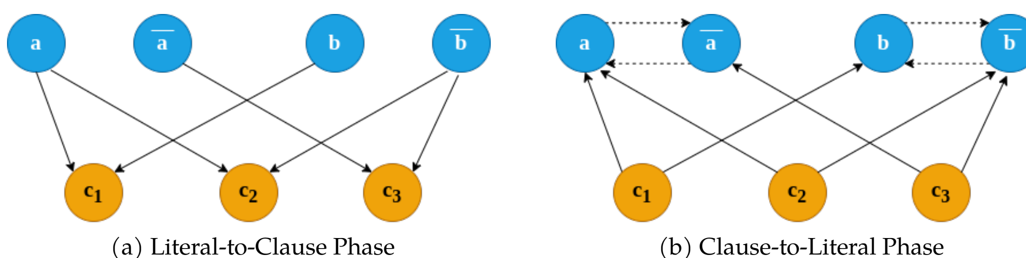


Figure 4.2: Embedding Sub-network

(pairs of clauses). As such, our model comprises a formula embedder followed by an encoder-decoder LSTM-based network connected to an attention sub-network responsible for selecting clause pairs. See Figure 4.1 for an overview of the NeuRes architecture. After obtaining the initial clause embeddings (representing the input formula), they are passed through the encoder to get the initial hidden state of the decoder, which marks the proof phase. Each time step, the model selects a clause pair which gets resolved into a new clause to append to the current formula graph. The model keeps deriving new clauses until either the empty clause is derived (marking proof completion) or the limit on proof length is reached (marking timeout).

4.2 Message-Passing Embedder

Similar to NeuroSAT, we use a message-passing GNN to obtain clause and literal embeddings by performing a predetermined number of rounds. Our formula graph is constructed in a similar fashion to NeuroSAT graphs as shown in Figure 4.3. For a formula F in n^* clauses and m variables, the ultimate output of this GNN is two matrices: $E^L \in \mathbb{R}^{m \times d}$ for literal embeddings and $E^C \in \mathbb{R}^{n \times d}$ for clause embedding, where $n \geqslant n^*$ is the current number of clauses (input+derived) and $d \in \mathbb{N}^+$ is the embedding vector width. Here we have two key differences from NeuroSAT. Firstly, NeuroSAT uses these embeddings as voters to predict satisfiability through a classification MLP. In our case, we use these embeddings as attention tokens for clause pair selection. Secondly, since our model derives new clauses with every application of the resolution rule **Res**, we need to embed these new clauses, as well as update existing embeddings to reflect their relation to the newly inferred clauses. Consequently, we need to introduce a new phase to the message-passing of NeuroSAT, for which we explore two approaches: **static embeddings** and **dynamic embeddings**.



(a) Literal-to-Clause Phase (b) Clause-to-Literal Phase

Figure 4.3: NeuroSAT Graph 2-Phase Message-Passing Round

Static Embeddings. In this approach, we do not change the embeddings of old clauses upon inferring a new clause. Instead, we exchange local messages between the node corresponding to the new clause and its literal nodes, in both directions. We do this for a predetermined number of times. The main advantage of this approach is its low cost. A major drawback is that old clauses never learn information about their relation to newly inferred clauses, which might not be critical for resolution steps involving an old clause and a derived one since the derived clause is aware of the old one. It would be problematic, however, when we consider resolution between two input clauses since neither of them would know about any derived clauses which makes it harder for our attention-based selector network to properly assign a score to that step as these two clauses are missing information about other clauses in the network.

Dynamic Embeddings. In this approach, we do not only generate a new clause and its embedding, we further update the embeddings of all other clauses. This accounts for the fact that the utility of an old clause may change with the introduction of a new clause. We do this by performing one message-passing round on the mature graph for every newly derived clause, which produces the new clause embedding and updates other clauses. Since message-passing rounds are parallel across clauses, a single update to the whole embedding matrix is reasonably efficient. An alternative is to repeat the entire message-passing protocol with every new clause, but this is vastly more expensive. Further, it is mostly redundant: Our new clause embeddings mature over time as they will get involved in the message passing rounds of subsequent inferences.

An upside for dynamic embeddings is that they encode the problem state along with the decoder hidden state as these two are the only changing components during the derivation phase. As such, having static embeddings places a heavier burden on the decoder hidden state to capture all changes to the problem state after each step[1]. From a decision-making perspective, the model often needs to select

[1]One caveat to that statement is that with static embeddings, newly generated embeddings also

largely different/distant pairs of clauses at consecutive steps, which means that our token selection scores should vary significantly from one step to the next. Updates to decoder hidden state induce this variance locally whereas updates to clause embeddings induce it more globally. Last but not least, performing one message-passing round after each step potentially mitigates overfitting on a fixed number of rounds.

Message-passing Rounds. For message passing, we need a way to determine the number of rounds to perform for each formula F. In our evaluation, we experiment with two settings for the number of rounds: a fixed number (16, 32, 64), and $|V_F| + 1$ where V_F is the set of variables of formula F. One could make the argument that unlike classical graph embedding tasks that place an emphasis on exploring node connectivity, our NeuroSAT-style graphs for UNSAT formulas have an unsatisfiable core that is very dense (with short diameters) since clauses in these cores are highly interdependent. Consequently, effectively embedding these formulas boils down to embedding these dense UNSAT cores, whose connectivity does not require many message-passing rounds to capture. This means that only a small portion of message-passing rounds are used to discover node connectivity leaving the larger remaining portion for learning useful relations between clauses other than mere reachability. Nevertheless, the model does not know this UNSAT core beforehand, so it needs to fully capture the formula graph to be able to recognize clauses that are part of this UNSAT subgraph from others that are not. Therefore, our choice of message-passing rounds should guarantee full exploration of graph connections. That is essentially the rationale behind the adaptive ($|V_F| + 1$)-round configuration as it is the maximum number of rounds needed to explore the whole graph. Note that this is a guaranteed upper bound by construction of the formula graph where clause nodes are anchored to their constituent literal nodes.

4.3 Encoder-Decoder Network

Following the embedding stage, we use a bidirectional LSTM encoder to summarize the formula using the clause embeddings as a token sequence. The last encoder hidden state is then used as the initial state for the decoder, which initially takes the start token as input. Subsequently, the input to the decoder becomes the embeddings for newly derived clauses. In our design, the decoder state encodes the solver decision at each step while the subsequent attention network interprets this decision into a concrete resolution step. It is noteworthy that the way information flows into the decoder encourages a stateful approach towards resolution proofs. That is, each step the decoder receives the previous hidden state containing formula summary as state and the last clause embedding as input. As such, the decoder answers the question: *Given the current formula and the fact that you just*

capture some state change, albeit somewhat minor.

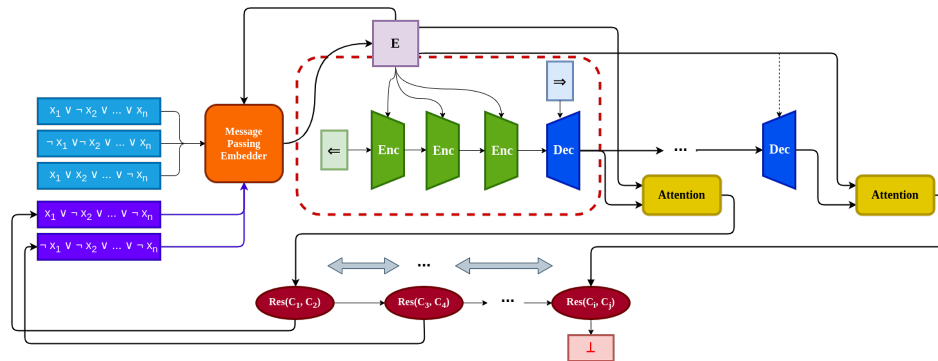


Figure 4.4: Encoder-Decoder Sub-network

derived this clause, what is the next step? as opposed to *Given the current formula, what is the next step?* Though both queries entail the same information about the current formula, the former explicitly prompts the model to build on its previous decisions.

4.4 Selector Networks

After producing clause and literal embeddings followed by the encoder phase, our model enters the derivation stage highlighted in Figure 4.5. For each step, our model needs to select two input clauses to resolve, produce the resultant clause, and add it to the current formula. We tokenize clauses by their embeddings as opposed to their encoder outputs as in PtrNet. The reason for that is that our embeddings are dynamic, and so the encoder outputs need to be dynamic, too. While embeddings can be updated in parallel, encoder outputs are generated in sequence, which is rather costly.

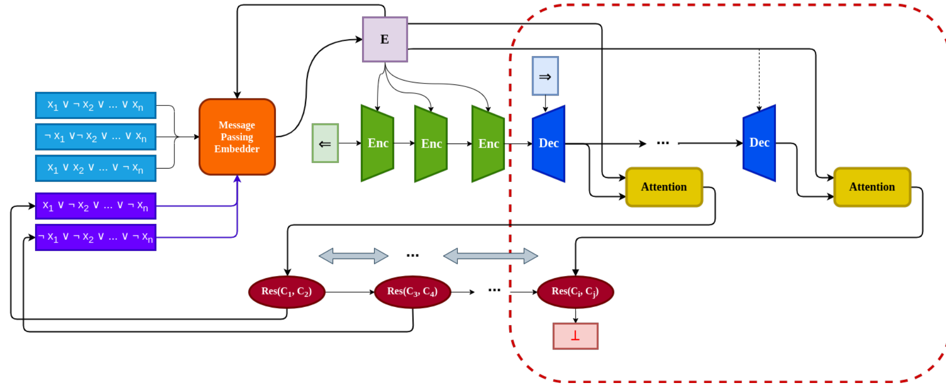


Figure 4.5: Selector Sub-network

To realize our clause-pair selection mechanism, we employ three attention-based designs. With the decoder as the main driver for our model, at each step t we use the decoder hidden state vector h_t for querying our attention networks over the set of tokens (clauses or literals).

4.4.1 Cascaded Pointer-Attention (Casc-Attn)

This is arguably the most naive approach. In this design, pairs are selected by making two consecutive attention queries on the clause pool. This is, however, not ideal because the two clauses are interdependent, which makes the selection more about the pair than each clause independently. This can be mitigated by conditioning the second attention query on the outcome (i.e., the clause) of the first query. Figure 4.6 shows this scheme where we perform the first query using h_t concatenated with a zero token vector while performing the second query using h_t concatenated with the embedding vector E_i^C of the clause selected in the first query.

Formally, Casc-Attn selects a clause index pair (c_1, c_2) as follows:

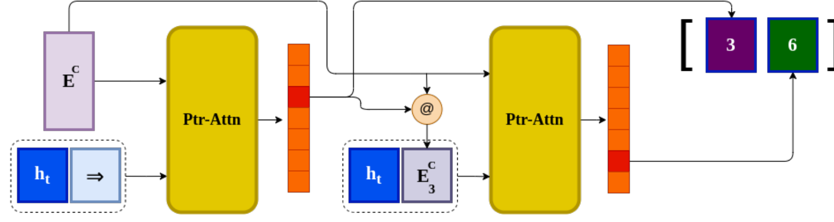


Figure 4.6: Cascaded Attention

$$Q_r = \begin{cases} h_t \parallel \mathbf{0} & \text{if } r = 1 \\ h_t \parallel E^C_{c_1} & \text{if } r = 2 \end{cases} \tag{4.1}$$

$$c_r = \underset{i}{\mathbf{argmax}} \left[u^T \tanh(W_1 Q_r + W_2 E^C_i)\right] \tag{4.2}$$

where $W_1 \in \mathbb{R}^{2d \times d}, W_2 \in \mathbb{R}^{d \times d}, u \in \mathbb{R}^d$ are trainable network parameters.

One upside of this design is that it is not limited to pair selection, as it can be used to select a tuple of arbitrary length. Nevertheless, this design still chooses each clause independently from the subsequent ones, which is undesirable.

4.4.2 Full Self-Attention (Full-Attn)

To address the downside of independent clause selection, this variant performs self-attention between all clauses to obtain a matrix $S \in \mathbb{R}^{n \times n}$ where $S_{i,j}$ represents the attention score of the clause pair (C_i, C_j) as shown in Figure 4.7. Thus, the model selects clause pairs in tandem by choosing the cell with the maximal score. In this attention scheme, clause embeddings are used as keys while queries are constructed by concatenating the decoder hidden state h_t with all clause embeddings. In one experimental variant, we compute a value matrix for clause embeddings to build a context vector from the weighted softmax average of clause values. We then blend this context vector with the decoder hidden state using a GRU cell with the aim of refreshing the problem state. Doing so, however, did not empirically result in any notable improvements.

Formally, Full-Attn selects a clause index pair (c_1, c_2) as follows:

$$Q_r = W_Q(h_t \parallel E^C_r); \qquad K_r = W_K E^C_r \tag{4.3}$$

$$S = \frac{QK^T}{\sqrt{d}} \tag{4.4}$$

$$(c_1, c_2) = \underset{(i,j)}{\mathbf{argmax}}\ S_{i,j} \tag{4.5}$$

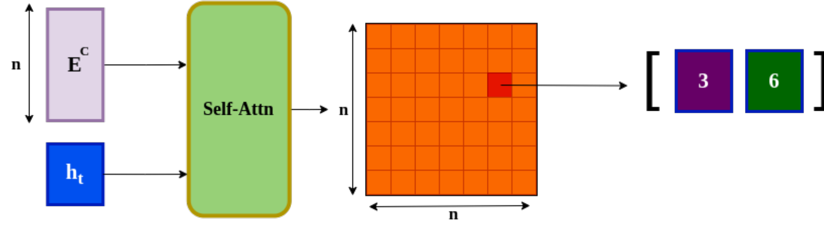


Figure 4.7: Full Self-Attention

where $W_Q \in \mathbb{R}^{2d \times d}$, $W_K \in \mathbb{R}^{d \times d}$ are trainable network parameters. Since S contains many cells that correspond to invalid resolution steps (i.e., two clauses that cannot be resolved), we mask out the invalid cells from the attention grid in ensure the network selection is valid at every step.

In Section 4.2, we stated that under static embeddings for a derived clause, as the embedder creates its embedding, it only updates the representations of the variables involved in it – leaving other clause embeddings intact. This might present a problem for Full-Attn where the attention grid contains all clauses including disconnected[2] pairs. An example of such a pair would be two derived clauses that do not share a variable. This could potentially lower the efficacy of the attention mechanism as it tries to match clauses that are unaware of each other.

4.4.3 Anchored Self-Attention (Anch-Attn)

Effective as full self-attention is, the attention grid grows quadratically with the number of clauses, which is arguably expensive. In this variant, we relax this cost by exploiting a property of binary resolution where each step targets a single variable in the two resolvent clauses. This allows us to narrow down candidate clause pairs by first selecting a variable as an anchor on which our clauses should be resolved. As such, we do not need to consider the full clause set at once, only the clauses containing the chosen variable **v**. We further compress the attention grid by lining clauses containing the literal **v** on rows while lining clauses containing the literal $\bar{\mathbf{v}}$ on columns. This reduces the redundancy of the attention grid since clauses containing the variable **v** with the same parity cannot be resolved on **v**, so there is no point in matching them. In the worst case, this relaxed grid is of size $\frac{n}{2} \times \frac{n}{2} = \frac{n^2}{4}$ instead of n^2. In this scheme, we have two attention modules: one pointer-attention network to choose an anchor variable followed by a self-attention network to produce the anchored score grid.

In light of Figure 4.8, this approach combines structural elements from Casc-Attn

[2]We use the terms ***connected*** and ***disconnected*** here to refer to the fact of whether two nodes have exchanged messages (in either direction) or not, respectively.

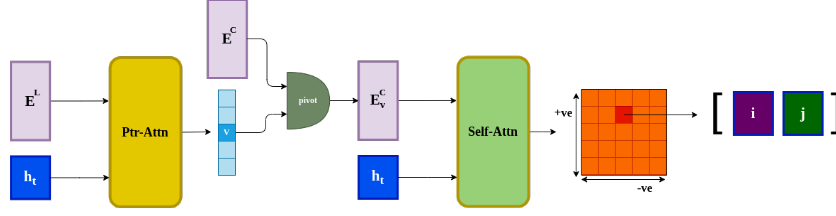


Figure 4.8: Anchored Self-Attention

(pointer attention) Full-Attn (self-attention); however, both elements are used differently in Anch-Attn. Firstly, pointer-attention in Casc-Attn is used to select clauses whereas in Anch-Attn, it is used to select variables. Secondly, self-attention in Full-Attn matches any pair of clauses (c_i, c_j) in both directions as the row and column dimensions in the attention score grid reflect the same clauses (all clauses). By contrast, Anch-Attn only computes self-attention scores for clause pairs in only one order (positive instance to negative instance).

Formally, Anch-Attn selects an anchor variable **v** as follows:

$$\mathbf{v} = \underset{i}{\mathbf{argmax}} \left[u^T \tanh(W_1 h_t + W_2(E_i^{L^+} + E_i^{L^-})) \right] \tag{4.6}$$

where $W_1 \in \mathbb{R}^{d \times d}, W_2 \in \mathbb{R}^{d \times d}, u \in \mathbb{R}^d$ are trainable network parameters. The clause index pair (c_1, c_2) is then selected according to the same equations of Full-Attn (Eq. 4.5) using the **v**-anchored set of clause embeddings. Note that, unlike Full-Attn, the indices on the self-attention grid in Anch-Attn need to be mapped back to the original indices.

Despite being a relaxation on Full-Attn, Anch-Attn has a distinct edge over Full-Attn under static embeddings in form of the following property:

Lemma 4.1 *Clauses in the variable-anchored attention grid of Anch-Attn are guaranteed to be connected under both static and dynamic embeddings.*

Proof Let **v** be a variable in the input formula, and the set of clauses of a **v**-anchored attention grid be A. We show that we always have at least one clause $A_i \in A$ that reaches all other clauses in A on the formula graph. We make two case distinctions:

Case 1: All clauses in A are input clauses (in the original formula). Here, the lemma follows trivially since all these clause were connected during the input-phase message-passing protocol as they share at least one variable **v**.

Case 2: A contains derived clauses. Let A_i be the most recently derived clause in A. Since A_i shares variable **v** with all other clauses in A, then A_i would be connected to them all during the derivation-phase message-passing protocol immediately after

A_i was derived. This is because A_i receives a message from $\mathbf{v}$ (under both static and dynamic embeddings) containing information about all other clauses containing $\mathbf{v}$, which is precisely $A \setminus \{A_i\}$. Therefore, the lemma holds. $\square$

Chapter 5

Training and Hyperparameters

5.1 Dataset

For our training and testing data, we adopt the same formula generation method as NeuroSAT, namely **SR**(n) where n is the number of variables in the formula. This method was designed to generate a generalized formula distribution that is not limited to a particular domain of SAT problems. The generation starts with an empty formula then each turn, it adds a random clause and checks the SAT status of the resulting formula using a classical SAT solver. Clauses are generated by sampling k variables (with a preset mean of $\bar{k} = 4$) where a variable is negated with a 50% probability. Once an added clause makes the formula unsatisfiable, the generation stops and the resulting unsatisfiable formula F is added to the dataset along with its satisfiable conjugate $F^{'}$ obtained by negating one literal in the last clause. For the purposes of NeuRes, we are only interested in the unsatisfiable formulas, so $F^{'}$ is discarded. To control our data distributions, we vary the range on the number of Boolean variables involved in each formula. For our training data, we use formulas in **SR**$(U(10, 40))$ where $U(10, 40)$ denotes the uniform distribution on integers between 10 and 40 (inclusive). For our out-of-distribution dataset, we generate formulas in **SR**$(U(40, 80))$. As for our teacher resolution proofs, we use a ***BooleForce***[1] solver on the formulas generated on the **SR** distribution.

5.2 Loss Function

We train our model in a supervised fashion using teacher forcing on expert proofs. During training, expert actions (clause pairs) are imposed throughout a formula run (i.e., episode). The length of the teacher proof dictates the length of the respective episode, denoted as T. Model parameters θ are trained to maximize the likelihood of expert choices y_t as shown in Eq 5.1. These likelihoods are weighted by a time-horizon discounting factor $\gamma < 1.0$ in order to assign higher weights to later stages of the run as the model gets closer to deriving the empty clause $(t \to T)$. In addition, discounting step-wise losses gives higher mean episodic weights to

shorter proofs, which is beneficial since shorter proofs have less room for redundancy and variance – making them clearer learning signals. This discounting was also empirically observed to yield better results than using unweighted likelihoods.

$$\max_{\theta} \quad \frac{1}{T}\sum_{t}^{T} \log(p(y_t;\theta)) \cdot \gamma^{(T-t)} \tag{5.1}$$

5.3 Proof-Reduction Bootstrapping

Advanced as industrial resolution solvers are, they are still algorithmically suboptimal. One important implication of that is their proofs having a significant margin of redundancy which varies from case to case. That is, the expert behaves on some problem instances more optimally than others. Since our model tries to maximize expert-action likelihoods on average, that results in the model learning strategies that are less optimal than most-optimal expert proofs as well as ones more optimal than least-optimal expert proofs. We exploit the latter fact in our training by pre-rolling each input problem using model actions only, and whenever the model proof is shorter than the expert's, this shorter proof replaces the expert's in the dataset. In other words, we maximize the likelihood of the shorter proof (between model and expert). In doing so recursively, the model progressively becomes its own teacher by exploiting redundancies in the expert algorithm. Naturally, this introduces a bias towards the model's own actions while reducing variance by having a more concise learning signal in form of shorter proofs. It is, of course, uncertain whether training on shorter proofs necessarily leads to an improved performance, but it is a hypothesis we explore in our experiments.

To mitigate the introduced bias effect, we train several model generations, each trained on the reduced data pool of its predecessor. In doing so, each fresh model gets trained on proofs shortened by a different model, which allows the later generations to both exploit the existing reduction insights while having room for finding new ones. In the case of a single generation, the latter effect gets throttled, as the model matures, due to that self-bias, which gets reset in each new generation.

5.4 Hyperparameters

Our neural solver has four main hyperparameters that influence network size, depth, and loss weighting.

Latent Vector Widths. Embedding and hidden state vectors have a fixed width of 512, which was experimentally found to yield the best trade-off between memory and learning capacity.

Training hyperparameters. We train our model with a batch size of 1 and an Adam optimizer [18] for 50 epochs. We linearly anneal the learning rate from

5×10^{-5} to zero over the training episodes. This was shown to yield notably better results than using a constant learning rate (cf. Appendix A.1). We use a time discounting factor $\lambda = 0.99$ for the episodic loss.

Chapter 6

Experiments

There are two main obstacles towards establishing a fair comparison between NeuRes and other neural methods in the literature. On one hand, there are currently no standard UNSAT benchmark datasets used across the neuro-symbolic literature, so comparing baselines on different datasets would compromise the fairness and reliability of the evaluation. On the other hand, to the best of our knowledge, existing neural methods either do not produce formal certificates to support their predictions or produce ones that are expensive to check (most commonly UNSAT cores), which further undermines the fairness of comparing against them. As such, in this section, we compare the test performance of our different NeuRes variants in terms of success rate (i.e., problems solved before timeout) and proof length relative to expert, denoted as p-Len $= \frac{|\mathcal{P}_{\text{NeuRes}}|}{|\mathcal{P}_{\text{expert}}|}$. Although we tend to care more about the success rate than optimality, p-Len is still a crucial metric to report, and it becomes particularly relevant in the context of our model's ground-truth reduction capabilities. There are several ways to set a timeout on proof length as a function of the input formula. To get a more precise comparison to expert performance, we set our maximum proof length as $4 \cdot |\mathcal{P}_{\text{expert}}|$. Note that we measure p-Len only for solved formulas to avoid diluting the average with resolution trails that timed out. For our evaluation, overall problem difficulty is quantified by the number of variables involved in the formula. To further demonstrate the reliability of our model learning, we test our models on 10K test formulas, which is more than the number of formulas they were trained on (8K). Experiments in this chapter are ordered in such a way that only the best performer of each experiment carries over to the subsequent ones in order to have a more focused evaluation.

6.1 Attention Variants

To assess the basic performance of NeuRes, we test it on a test set comprising 10K formulas belonging to the same distribution as the training dataset. We evaluate each attention variant using both static and dynamic embeddings to measure the effect of embedding dynamicity on each of them.

Variant	**Static Embedding**		**Dynamic Embedding**	
	Solved (%)	**p-Len**	**Solved (%)**	**p-Len**
Casc-Attn	13.62	1.8	38.98	1.97
Full-Attn	23.68	**1.59**	**87.19**	1.74
Anch-Attn	**26.28**	1.99	48.63	**1.6**

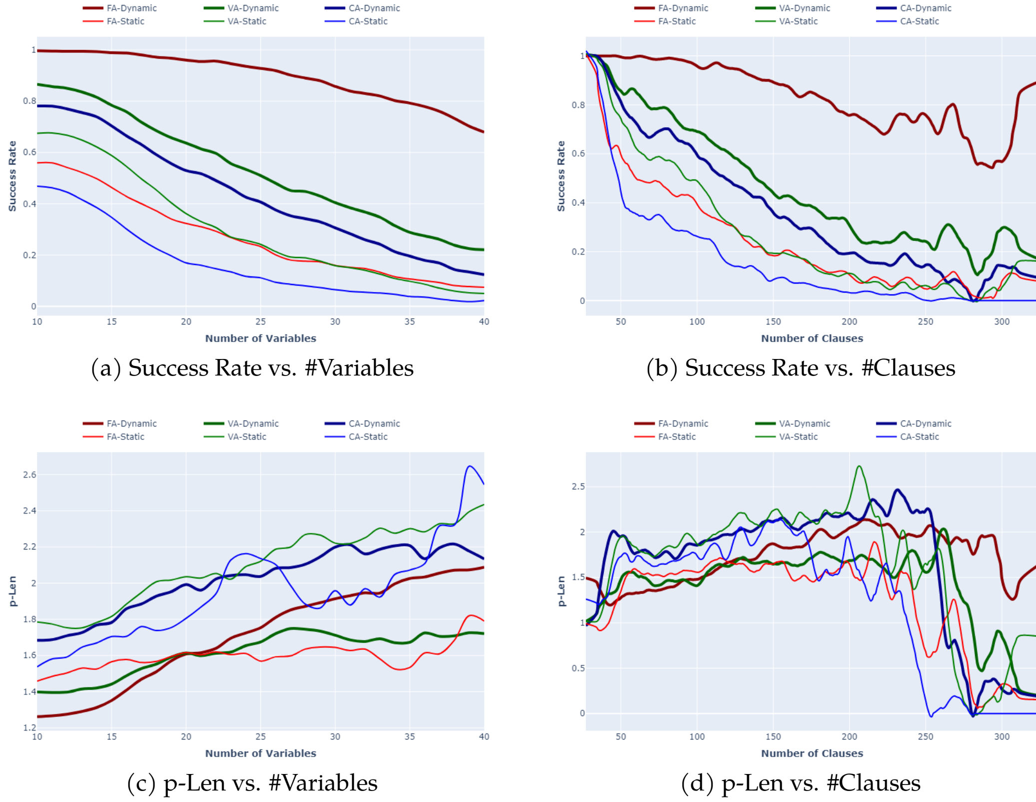
Table 6.1: In-Distribution Performance

As shown in Table 6.1, dynamic embedding is decisively superior for all three attention variants, thereby confirming its conceptual merit. While Anchored-Attention leads over other variants under static embeddings, the shift to dynamic embeddings unlocks the potential of Full-Attention where it has a much higher success rate than the other two variants, albeit at the cost of somewhat longer proofs on average.

In Section 4.4.3, we presented Anch-Attn as a compromise between Casc-Attn and Full-Attn with the expectation that it would perform somewhere higher than Casc-Attn and lower than Full-Attn. This appears to be the case under dynamic embeddings; however, under static embeddings, Anch-Attn performs better than Full-Attn. Numerically, the difference between their success rates in the static setting might not be large enough to warrant an explanation. Nonetheless, a potential way to explain it lies in Lemma 4.1 stating the guaranteed connectivity of all clauses in the self-attention grids of Anch-Attn – a property that clearly does not hold for Full-Attn under static embeddings.

One orthogonal takeaway from Figure 6.1 is that the number of variables is a more truthful indicator of problem difficulty than the number of clauses as increasing the former leads to a nearly monotonic decline in success rates while the latter does not present the same consistent inverse proportionality with success rates.

Having demonstrated ***dynamic-embedding*** **Full-Attn** to be the best-performing configuration (highest success rate by far) over in-distribution test settings, the remaining evaluation experiments will be exclusively demonstrated on this variant.



(a) Success Rate vs. #Variables

(b) Success Rate vs. #Clauses

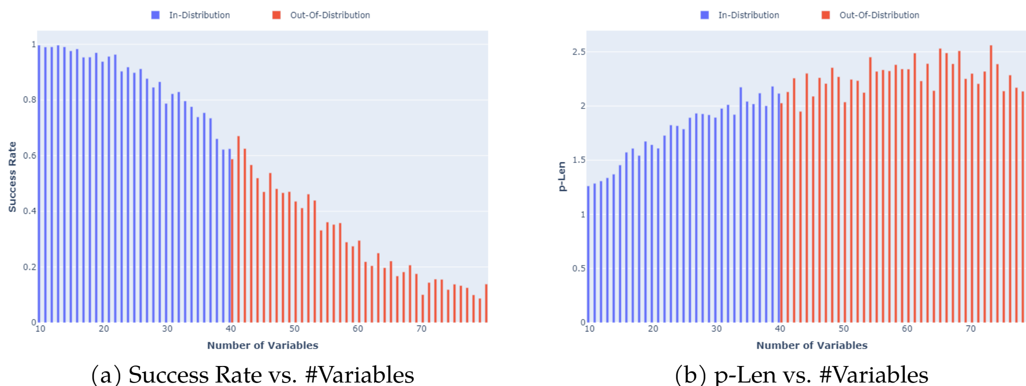
(c) p-Len vs. #Variables

(d) p-Len vs. #Clauses

Figure 6.1: In-Distribution Performance Breakdown

6.2 Out-Of-Distribution Performance

To further test the generalizability of NeuRes to unseen formulas, we evaluate our best model on a test set following a different distribution from that of the training data. Our out-of-distribution (OOD) test set contains 10K formulas in $\mathbf{SR}(\mathrm{U}(40, 80))$ while training formulas were in $\mathbf{SR}(\mathrm{U}(10, 40))$. Thus, our OOD test formulas are not only of a different distribution, but they are also larger in size than our training formulas. In doing so, we evaluate our model's ability to generalize to larger problem instances. OOD learning is a prominent problem in machine learning, so generally speaking, machine learning models should not be expected to perform equally well on unseen input distributions. As shown in Figure 6.2, although NeuRes still manages to perform reasonably well on formulas twice the sizes it was trained on, its performance still drops substantially from its in-distribution mark. This finding suggests that NeuRes learns a generally effective resolution strategy that applies to arbitrary formulas.



(a) Success Rate vs. #Variables

(b) p-Len vs. #Variables

Figure 6.2: Out-Of-Distribution Performance Breakdown (Dynamic Full-Attn)

Data	Solved (%)	p-Len
In-Distribution	87.19	1.74
Out-of-Distribution	30.58	2.25

Table 6.2: Overall Out-Of-Distribution Performance (Dynamic Full-Attn)

6.3 Number of Message-Passing Rounds

As we discussed in Section 4.2, the number of message-passing rounds is an important hyperparameter for our model. We experiment with 4 different configurations for the length of the message-passing protocol. In light of Table 6.3, we find that performing as many message-passing rounds as the maximum formula graph diameter (i.e., $|V_F| + 1$) yields better results in terms of both success rate and optimality.

#Rounds	Solved (%)	p-Len
16	82.12	1.83
32	87.19	1.74
64	52.11	1.77
$\|V_F\| + 1$	**89.43**	**1.71**

Table 6.3: Performance On Different Numbers of Message-Passing Rounds

Aside from boosting our previous best performer, this finding is also a practical relief because it removes the need to search for an ideal number of rounds for a given dataset distribution, as it already adapts to the problem sizes contained therein.

6.4 Shortening Teacher Proofs

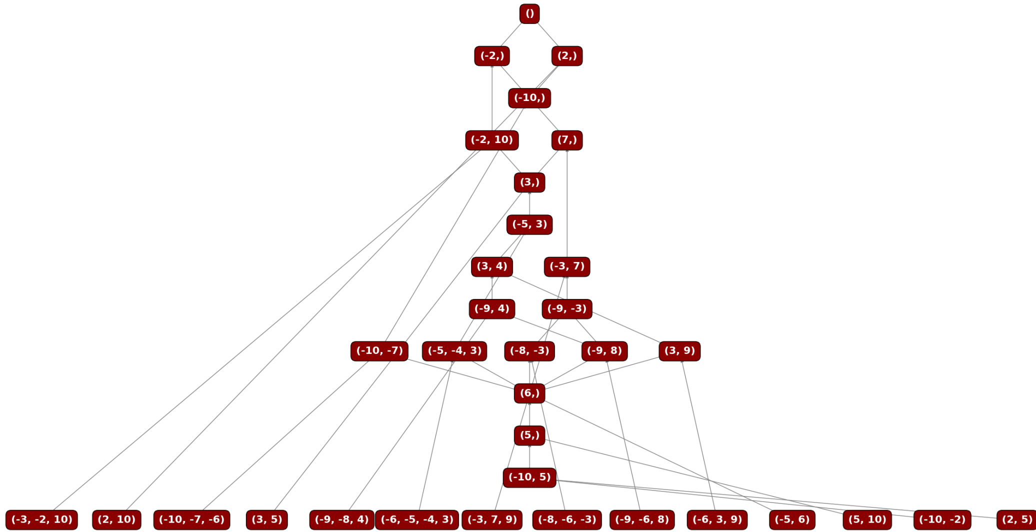
As NeuRes is solving a much harder problem than NeuroSAT, it also requires more supervision since it is trained with expert proofs. While these proofs are cheap to obtain using industrial solvers, it is still of interest to know how much NeuRes can absorb from a small pool of expert proofs. Although the model should not be expected to outperform the teacher *on average*, it would be somewhat uninspiring if all NeuRes did was encode the teacher strategy as is, only to play it back less optimally without any novelty. To investigate this, we check how often (if ever) NeuRes produces proofs shorter than the expert proof (i.e., p-Len < 1.0) on both training and unseen test problems. We perform this analysis on our previous best performer trained with regular teacher-forcing. We find that, on average, NeuRes manages to shorten $\sim 18\%$ of expert proofs by a notable factor – sometimes solving problems in less than half as many steps as the expert. This shows that NeuRes is capable of learning novel strategies to find shortcuts unseen by its teacher. To further quantify this effect, we report the maximum and average reduction ratios relative to teach proof length, on the pool of reduced proofs. Finally, we report the total reduction made to the dataset size in terms of total proof steps.

(%)	**Train**	**Test**
Proofs Reduced	17.82	18.29
Max. Reduction	86.11	76.4
Avg. Reduction	23.55	23.65
Total Reduction	3.07	3.15

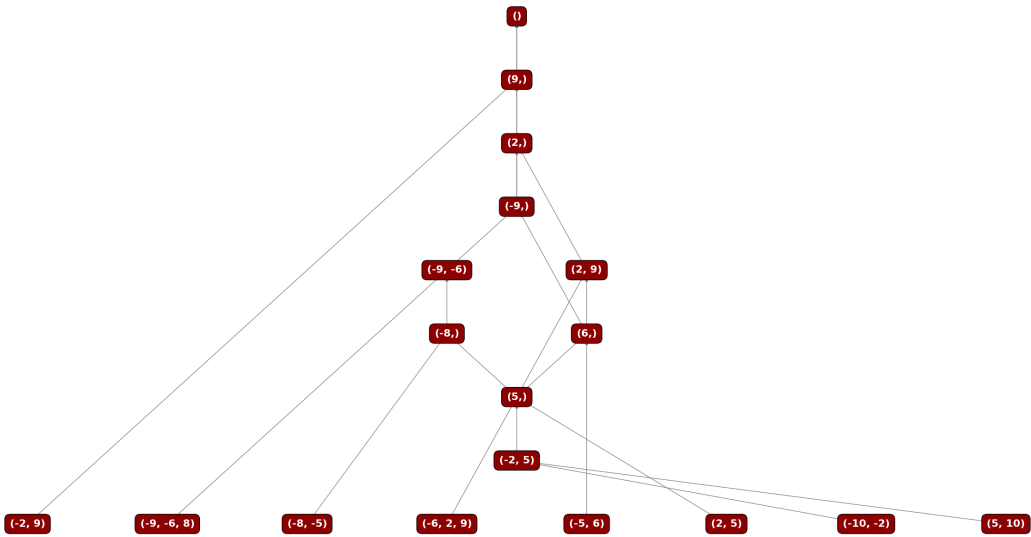
Table 6.4: Teacher Proof Reduction Statistics

Note that all rows in Table 6.4, except for Total Reduction, are computed over the *reduced portion* of the dataset, i.e., the proofs that were successfully shortened by NeuRes. Figure 6.3 shows an example for a teacher proof that NeuRes reduced by half (from 20 steps to 10 steps). There are examples of major reductions on much larger proofs (e.g., from ~ 800 to 400 steps), but due to size constraints, we only include this small example.

On rather interesting observation on Table 6.4 is that the model appears to be marginally better at producing shorter proofs for unseen (test) formulas than for training formulas. While we would normally expect the opposite, a fair speculation would be that the trained model was teacher-forced to match teacher proofs during training over multiple epochs while the same does not hold for unseen formulas where the bias towards teacher behavior is significantly lower. To definitively confirm this would require a more in-depth investigation.



(a) Teacher Proof



(b) NeuRes Proof

Figure 6.3: Teacher Proof Reduction Example

This is, in fact, the founding rationale behind bootstrapped training (introduced in Section 5.3), that is, that strict teacher-forcing throttles the model's exploration, which can allow it to learn better strategies while using teacher trajectories as a guide as opposed to a golden standard.

6.5 Bootstrapped Training

Having demonstrated the model's ability to produce shorter proofs than those of the expert, it is only natural to explore the impact of using the model as its own teacher during training on the success rate and optimality of the solver. The rationale here is that a combination of the model and expert is on average better than the expert alone. As such, training proofs can be progressively shortened by immediately putting intermediate model updates to use. Table 6.5 shows the performance comparison between the base non-bootstrapped Full-Attn model to its bootstrapped counterparts. In Section 5.3, we hypothesize a decline in the overall model performance as a result of the self-bias introduced by bootstrapping. Our experiments do indeed confirm this decline in the first and second bootstrapped generations.

Variant	**Solved (%)**	**p-Len**
Non-bootstrapped Base	89.43	1.71
Bootstrapped Gen-1	76.87	1.505
Bootstrapped Gen-2	88.27	1.367
Bootstrapped Gen-3	**92.84**	**1.23**

Table 6.5: Bootstrapped Models Performance (on in-distribution test formulas)

To put this in perspective, bootstrapping encourages the model to specialize in a subset of the training formulas, improving the model performance on this subset at the expense of limiting the model's generality. From a numerical standpoint, this effect can be seen as a byproduct of time-discounting our mean-based teacher forcing loss (Eq. 5.1) by a factor $\gamma < 1.0$, which results in shorter episodes having a higher mean weight $\mathbb{E}_t[\gamma^t]$. As such, shortening the guide proof for a particular formula increases its contribution to the loss function, placing higher emphasis on it, hence incentivizing the model to further optimize these proofs since they are easier to optimize because the intermediate model strategy already works on them. While this enhances the model's depth (p-Len), it limits its breadth over the problem space[1]. Therefore, to alleviate this, we train three bootstrapped generations, each trained on the dataset reduced by its predecessor with a fresh model to reset this bias. Table 6.5 shows that the performance loss caused by bootstrapping

[1]Breadth represents *how often* the model solves a given formula while depth represents *how well* the model solves a formula (**given that it solves it**).

in Gen-1 is gradually overcome through Gen-2 and Gen-3 into beating the non-bootstrapped baseline in terms of both success rate and optimality with a notable margin. Particularly, the sharp decline patterns in test p-Len show that the proof shortcutting insights, learned by the model on training formulas, are transferable to unseen test formulas as opposed to overfitting to training formulas.

In addition to success rate and p-Len, we inspect the reduction performance of our bootstrapped generations in light of the same metrics in Table 6.4. However, since during training, Gen-x models perform multiple reduction scans over the training dataset, we add a new metric of reduction depth computed as the number of progressive[2] reductions made to a proof.

	Gen-1	**Gen-2**	**Gen-3**	***Overall***
Max. Depth	8	7	6	11
Avg. Depth	2.1	1.76	1.46	2.24
Proofs Reduced (%)	45.19	45.29	35.94	70.53
Max. Reduction (%)	86.11	63.68	55.0	86.11
Avg. Reduction (%)	27.20	15.95	11.19	30.8
Total Reduction (%)	9.46	7.28	4.18	19.56

Table 6.6: Bootstrapped Training Dataset Reduction Statistics. Gen-2 and Gen-3 metrics are computed relative to their predecessor-shrunk dataset. Gen-1 and *Overall* are evaluated relative to the original training dataset.

Naturally, we should expect reductions to yield diminishing returns, which can be seen across all metrics in Table 6.6 with each new generation. Nonetheless, observing the overall reduction statistics, we find that this bootstrapped reduction process manages to reduce 70.53% of our training proofs by nearly one third on average. In terms of raw size, NeuRes effectively reduces the whole training proofs dataset by nearly 20%. Figure 6.4 shows the progression of the total reduction made by each generation to their given training dataset. We observe that Gen-1 has the highest slope as it operated on the original unreduced dataset whereas Gen-2 and Gen-3 operated on already-reduced datasets where there is less room for further reductions. Indeed, we already see that effect in miniature for all three progressions as their slope tends to attenuate near the end.

[2]Remember that each reduced proof replaces its predecessor on the fly, so these reductions are recursive.

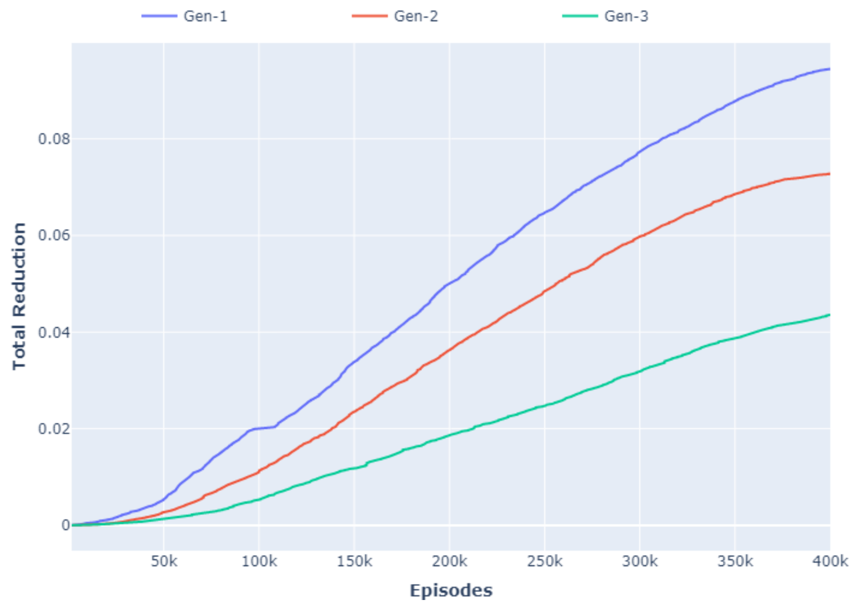


Figure 6.4: Train Dataset Reduction Over Number of Training Episodes

It would be interesting to find a cut-off on the number of generations where we no longer get any significant reductions with newer generations. However, owing to the sequential nature of this experiment[3], we defer this to a future work.

[3]Each generation can only start training after its predecessor finishes.

Chapter 7

Conclusion

In this thesis, we have presented NeuRes, the first approach to utilize neural networks for generating resolution proofs of unsatisfiability. Structurally, NeuRes is a fusion between PtrNet and NeuroSAT that advances their functional capabilities. On the one hand, it extends PtrNet by not only generating solutions composed of input tokens but also combinations of a growing pool thereof. On the other hand, it goes beyond NeuroSAT by generating sound-by-design resolution certificates and realizing a more exploratory mode of operation. Furthermore, being a resolution prover, NeuRes is sound and complete as it will always eventually solve the input formula given enough time.

Despite its promising benchmark performance, NeuRes still cannot solely outperform highly engineered industrial solvers. This is generally the case for neuro-symbolic methods as standalone tools. Nevertheless, neural models were shown to improve symbolic solvers when used as an auxiliary component. For instance, [25] modifies existing SAT solvers by replacing variable activity scores with NeuroSAT's predictions, which resulted in solving notably more problems within standard timeout. Similarly, NeuRes attention scores can be used as a branching guide for a SAT-solving algorithm such as CDCL.

In addition to its value as a resolution prover, NeuRes can also be used as a proof reducer. We have established this capability of NeuRes models trained with both regular teacher-forcing and bootstrapped training – with the latter being largely more powerful. This is indeed a feature we find of great practical and conceptual import as it points to an inherent ability to find and reduce redundancies in teacher proofs by simply training on them with no extra guidance.

For future extensions, NeuRes – though trained in a supervised fashion – can also operate as a reinforcement learning agent given the action-based nature of its predictions along with having a singular well-defined goal: deriving the empty clause from an input formula. In its current form, NeuRes can be simply trained with a

policy-gradient algorithm such as REINFORCE or Actor-Critic. Nonetheless, carefully crafted reward functions might be necessary since a binary outcome-based reward, on whether or not the agent derives the empty clause, is arguably too sparse given the huge action space. This is a curious path to explore with our method as it stands the potential of finding more optimal strategies than those of a human-crafted algorithm.

Implementation Source Code: We provide the full source code of our implementation including the experiment scripts on the following GitHub repository: `https://github.com/Oschart/NeuRes`

Appendix A

Appendix

A.1 Constant vs. Linearly Annealed Learning Rate

Here we show the effect of linearly annealing the Adam optimizer base learning rate to zero over the number of training episodes on the validation success rate throughout training. As shown in Figure A.1, linear annealing brings considerable gains to the model performance. Consequently, we use a linearly annealed learning rate for all our main experiments.

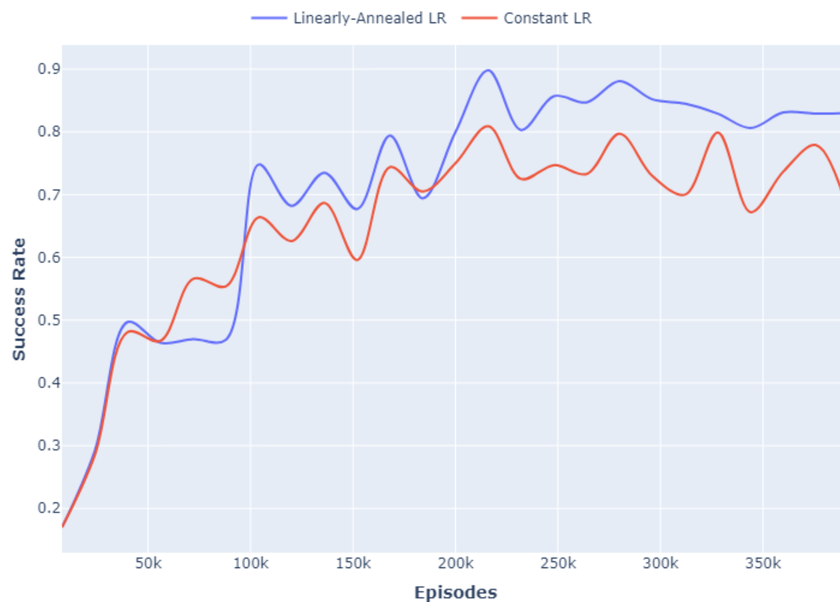


Figure A.1: Validation Success Rate With Different LR Schedules

A.2 Network Parameters

Modules	Shape	#Parameters
embedder.$L_{init}.weight$	512x1	512
embedder.$L_{init}.bias$	512	512
embedder.$C_{init}.weight$	512x1	512
embedder.$C_{init}.bias$	512	512
embedder.$L_{msg}.l1.weight$	512x512	262,144
embedder.$L_{msg}.l1.bias$	512	512
embedder.$L_{msg}.l2.weight$	512x512	262,144
embedder.$L_{msg}.l2.bias$	512	512
embedder.$L_{msg}.l3.weight$	512x512	262,144
embedder.$L_{msg}.l3.bias$	512	512
embedder.$C_{msg}.l1.weight$	512x512	262,144
embedder.$C_{msg}.l1.bias$	512	512
embedder.$C_{msg}.l2.weight$	512x512	262,144
embedder.$C_{msg}.l2.bias$	512	512
embedder.$C_{msg}.l3.weight$	512x512	262,144
embedder.$C_{msg}.l3.bias$	512	512
embedder.$L_{update}.weight_{ih-l0}$	2048x1024	2,097,152
embedder.$L_{update}.weight_{hh-l0}$	2048x512	1,048,576
embedder.$L_{update}.bias_{ih-l0}$	2048	2,048
embedder.$L_{update}.bias_{hh-l0}$	2048	2,048
embedder.$C_{update}.weight_{ih-l0}$	2048x512	1,048,576
embedder.$C_{update}.weight_{hh-l0}$	2048x512	1,048,576
embedder.$C_{update}.bias_{ih-l0}$	2048	2,048
embedder.$C_{update}.bias_{hh-l0}$	2048	2,048
encoder.weight$_{ih-l0}$	2048x512	1,048,576
encoder.weight$_{hh-l0}$	2048x512	1,048,576
encoder.bias$_{ih-l0}$	2048	2,048
encoder.bias$_{hh-l0}$	2048	2,048
encoder.weight$_{ih-l0-rev}$	2048x512	1,048,576
encoder.weight$_{hh-l0-rev}$	2048x512	1,048,576
encoder.bias$_{ih-l0-rev}$	2048	2,048
encoder.bias$_{hh-l0-rev}$	2048	2,048
decoder.weight$_{ih}$	2048x512	1,048,576
decoder.weight$_{hh}$	2048x512	1,048,576
decoder.bias$_{ih}$	2048	2,048
decoder.bias$_{hh}$	2048	2,048
$C_{selector}.W_Q.weight$	512x1024	524,288
$C_{selector}.W_K.weight$	512x512	262,144
Total		13,919,232

Table A.1: Full Self-Attention Network Parameters

Modules	**Shape**	**#Parameters**
embedder.$L_{init}.weight$	512x1	512
embedder.$L_{init}.bias$	512	512
embedder.$C_{init}.weight$	512x1	512
embedder.$C_{init}.bias$	512	512
embedder.$L_{msg}.l1.weight$	512x512	262,144
embedder.$L_{msg}.l1.bias$	512	512
embedder.$L_{msg}.l2.weight$	512x512	262,144
embedder.$L_{msg}.l2.bias$	512	512
embedder.$L_{msg}.l3.weight$	512x512	262,144
embedder.$L_{msg}.l3.bias$	512	512
embedder.$C_{msg}.l1.weight$	512x512	262,144
embedder.$C_{msg}.l1.bias$	512	512
embedder.$C_{msg}.l2.weight$	512x512	262,144
embedder.$C_{msg}.l2.bias$	512	512
embedder.$C_{msg}.l3.weight$	512x512	262,144
embedder.$C_{msg}.l3.bias$	512	512
embedder.$L_{update}.weight_{ih-l0}$	2048x1024	2,097,152
embedder.$L_{update}.weight_{hh-l0}$	2048x512	1,048,576
embedder.$L_{update}.bias_{ih-l0}$	2048	2,048
embedder.$L_{update}.bias_{hh-l0}$	2048	2,048
embedder.$C_{update}.weight_{ih-l0}$	2048x512	1,048,576
embedder.$C_{update}.weight_{hh-l0}$	2048x512	1,048,576
embedder.$C_{update}.bias_{ih-l0}$	2048	2,048
embedder.$C_{update}.bias_{hh-l0}$	2048	2,048
encoder.weight$_{ih-l0}$	2048x512	1,048,576
encoder.weight$_{hh-l0}$	2048x512	1,048,576
encoder.bias$_{ih-l0}$	2048	2,048
encoder.bias$_{hh-l0}$	2048	2,048
encoder.weight$_{ih-l0-rev}$	2048x512	1,048,576
encoder.weight$_{hh-l0-rev}$	2048x512	1,048,576
encoder.bias$_{ih-l0-rev}$	2048	2,048
encoder.bias$_{hh-l0-rev}$	2048	2,048
decoder.weight$_{ih}$	2048x512	1,048,576
decoder.weight$_{hh}$	2048x512	1,048,576
decoder.bias$_{ih}$	2048	2,048
decoder.bias$_{hh}$	2048	2,048
$C_{selector}.var_Q.weight$	512x512	262,144
$C_{selector}.var_K.weight$	512x512	262,144
$C_{selector}.var_{attn}.weight$	1x512	512
$C_{selector}.W_Q.weight$	512x1024	524,288
$C_{selector}.W_K.weight$	512x512	262,144
Total		14,444,032

Table A.2: Anchored Self-Attention Network Parameters

Modules	Shape	#Parameters
embedder.$L_{init}.weight$	512x1	512
embedder.$L_{init}.bias$	512	512
embedder.$C_{init}.weight$	512x1	512
embedder.$C_{init}.bias$	512	512
embedder.$L_{msg}.l1.weight$	512x512	262,144
embedder.$L_{msg}.l1.bias$	512	512
embedder.$L_{msg}.l2.weight$	512x512	262,144
embedder.$L_{msg}.l2.bias$	512	512
embedder.$L_{msg}.l3.weight$	512x512	262,144
embedder.$L_{msg}.l3.bias$	512	512
embedder.$C_{msg}.l1.weight$	512x512	262,144
embedder.$C_{msg}.l1.bias$	512	512
embedder.$C_{msg}.l2.weight$	512x512	262,144
embedder.$C_{msg}.l2.bias$	512	512
embedder.$C_{msg}.l3.weight$	512x512	262,144
embedder.$C_{msg}.l3.bias$	512	512
embedder.$L_{update}.weight_{ih-l0}$	2048x1024	2,097,152
embedder.$L_{update}.weight_{hh-l0}$	2048x512	1,048,576
embedder.$L_{update}.bias_{ih-l0}$	2048	2,048
embedder.$L_{update}.bias_{hh-l0}$	2048	2,048
embedder.$C_{update}.weight_{ih-l0}$	2048x512	1,048,576
embedder.$C_{update}.weight_{hh-l0}$	2048x512	1,048,576
embedder.$C_{update}.bias_{ih-l0}$	2048	2,048
embedder.$C_{update}.bias_{hh-l0}$	2048	2,048
encoder.weight$_{ih-l0}$	2048x512	1,048,576
encoder.weight$_{hh-l0}$	2048x512	1,048,576
encoder.bias$_{ih-l0}$	2048	2,048
encoder.bias$_{hh-l0}$	2048	2,048
encoder.weight$_{ih-l0-rev}$	2048x512	1,048,576
encoder.weight$_{hh-l0-rev}$	2048x512	1,048,576
encoder.bias$_{ih-l0-rev}$	2048	2,048
encoder.bias$_{hh-l0-rev}$	2048	2,048
decoder.weight$_{ih}$	2048x512	1,048,576
decoder.weight$_{hh}$	2048x512	1,048,576
decoder.bias$_{ih}$	2048	2,048
decoder.bias$_{hh}$	2048	2,048
$C_{selector}.W1.weight$	512x512	262,144
$C_{selector}.W2.weight$	512x1024	524,288
$C_{selector}.vt.weight$	1x512	512
Total		13,919,744

Table A.3: Cascaded Pointer-Attention Network Parameters

Bibliography

[1] Booleforce sat solver. https://fmv.jku.at/booleforce/.

[2] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. Learning to solve circuit-sat: An unsupervised differentiable approach. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=BJxgz2R9t7.

[3] Adrian Balint, Anton Belov, Marijn J.H. Heule, and Matti Järvisalo, editors. *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, Finland, 2013.

[4] Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2023.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003. doi: 10.1016/S0065-2458(03)58003-2. URL https://doi.org/10.1016/S0065-2458(03)58003-2.

[6] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing - SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010. doi: 10.1007/978-3-642-14186-7_6. URL https://doi.org/10.1007/978-3-642-14186-7_6.

[7] Chris Cameron, Rex Chen, Jason S. Hartford, and Kevin Leyton-Brown. Predicting propositional satisfiability via end-to-end learning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second*

Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, pages 3324–3331. AAAI Press, 2020. doi: 10.1609/aaai.v34i04.5733. URL `https://doi.org/10.1609/aaai.v34i04.5733`.

[8] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971. doi: 10.1145/800157.805047. URL `https://doi.org/10.1145/800157.805047`.

[9] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711. PMLR, 2016.

[10] Ashish Darbari, Bernd Fischer, and João Marques-Silva. Industrial-strength certified SAT solving through verified SAT proof checking. In Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel, and Jim Woodcock, editors, *Theoretical Aspects of Computing - ICTAC 2010, 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings*, volume 6255 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2010. doi: 10.1007/978-3-642-14808-8_18. URL `https://doi.org/10.1007/978-3-642-14808-8_18`.

[11] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[12] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. *CoRR*, abs/2303.04910, 2023. doi: 10.48550/arXiv.2303.04910. URL `https://doi.org/10.48550/arXiv.2303.04910`.

[13] Evguenii I. Goldberg and Yakov Novikov. Verification of proofs of unsatisfiability for CNF formulas. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10886–10891. IEEE Computer Society, 2003. doi: 10.1109/DATE.2003.10008. URL `https://doi.ieeecomputersociety.org/10.1109/DATE.2003.10008`.

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[15] Leon Henkin. The completeness of the first-order functional calculus. *The journal of symbolic logic*, 14(3):159–166, 1949.

[16] Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. Bridging the gap between easy generation and efficient verification of unsatisfiability proofs. *Softw. Test. Verification Reliab.*, 24(8):593–607, 2014. doi: 10.1002/stvr.1549. URL `https://doi.org/10.1002/stvr.1549`.

[17] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2012. doi: 10.1007/978-3-642-31365-3_28. URL `https://doi.org/10.1007/978-3-642-31365-3_28`.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Peter Lammich. Efficient verified (UN)SAT certificate checking. *J. Autom. Reason.*, 64(3):513–532, 2020. doi: 10.1007/s10817-019-09525-z. URL `https://doi.org/10.1007/s10817-019-09525-z`.

[20] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL `https://openreview.net/forum?id=S1eZYeHFDS`.

[21] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi: 10.3233/FAIA200987. URL `https://doi.org/10.3233/FAIA200987`.

[22] Emils Ozolins, Karlis Freivalds, Andis Draguns, Eliza Gaile, Ronalds Zakovskis, and Sergejs Kozlovics. Goal-aware neural SAT solver. In *International Joint Conference on Neural Networks, IJCNN 2022, Padua, Italy, July 18-23, 2022*, pages 1–8. IEEE, 2022. doi: 10.1109/IJCNN55064.2022.9892733. URL `https://doi.org/10.1109/IJCNN55064.2022.9892733`.

[23] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *CoRR*, abs/2009.03393, 2020. URL `https://arxiv.org/abs/2009.03393`.

[24] Frederik Schmitt, Christopher Hahn, Markus N. Rabe, and Bernd Finkbeiner. Neural circuit synthesis from specification patterns. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Process-*

ing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual, pages 15408–15420, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/8230bea7d54bcdf99cdfe85cb07313d5-Abstract.html.

[25] Daniel Selsam and Nikolaj Bjørner. Guiding high-performance sat solvers with unsat-core predictions. In *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*, pages 336–353. Springer, 2019.

[26] Daniel Selsam and Nikolaj S. Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 336–353. Springer, 2019. doi: 10.1007/978-3-030-24258-9_24. URL https://doi.org/10.1007/978-3-030-24258-9_24.

[27] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=HJMC_iA5tm.

[28] Ling Sun, David Gérault, Adrien Benamira, and Thomas Peyrin. Neurogift: Using a machine learning based sat solver for cryptanalysis. In Shlomi Dolev, Vladimir Kolesnikov, Sachin Lodha, and Gera Weiss, editors, *Cyber Security Cryptography and Machine Learning - Fourth International Symposium, CSCML 2020, Be'er Sheva, Israel, July 2-3, 2020, Proceedings*, volume 12161 of *Lecture Notes in Computer Science*, pages 62–84. Springer, 2020. doi: 10.1007/978-3-030-49785-9_5. URL https://doi.org/10.1007/978-3-030-49785-9_5.

[29] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

[30] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[31] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Sum-*

mer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, volume 8561 of *Lecture Notes in Computer Science*, pages 422–429. Springer, 2014. doi: 10.1007/978-3-319-09284-3_31. URL https://doi.org/10.1007/978-3-319-09284-3_31.

[32] Jiaxuan You, Haoze Wu, Clark W. Barrett, Raghuram Ramanujan, and Jure Leskovec. G2SAT: learning to generate SAT formulas. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 10552–10563, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/daea32adcae6abcb548134fa98f139f9-Abstract.html.

[33] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, pages 10880–10885. IEEE Computer Society, 2003. doi: 10.1109/DATE.2003.10014. URL https://doi.ieeecomputersociety.org/10.1109/DATE.2003.10014.