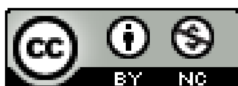
NEURAL NETWORKS AND THE SATISFIABILITY PROBLEM

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Daniel Selsam
June 2019

© 2019 by Daniel Selsam. All Rights Reserved.

Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/us/

This dissertation is online at: http://purl.stanford.edu/jt562cf4590

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Percy Liang, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Dill, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Chris Re

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Neural networks and satisfiability (SAT) solvers are two of the crowning achievements of computer science, and have each brought vital improvements to diverse real-world problems. In the past few years, researchers have begun to apply increasingly sophisticated neural network architectures to increasingly challenging problems, with many encouraging results. In the SAT field, on the other hand, after two decades of consistent, staggering improvements in the performance of SAT solvers, the rate of improvement has declined significantly. Together these observations raise two critical scientific questions. First, what are the fundamental capabilities of neural networks? Second, can neural networks be leveraged to improve high-performance SAT solvers?

We consider these two questions and make the following two contributions. First, we demonstrate a surprising capability of neural networks. We show that a simple neural network architecture trained in a certain way can learn to solve SAT problems on its own without the help of hard-coded search procedures, even after only end-to-end training with minimal supervision. Thus we establish that neural networks are capable of learning to perform discrete search. Second, we show that neural networks can indeed be leveraged to improve high-performance SAT solvers. We use the same neural network architecture to provide heuristic guidance to several state-of-the-art SAT solvers, and find that each enhanced solver solves substantially more problems than the original on a benchmark of challenging and diverse real-world SAT problems.

Acknowledgments

I found the PhD program to be an incredibly rewarding experience, enriched beyond measure by the many people I got to learn from and collaborate with.

I am grateful to my two advisors, Percy Liang and David L. Dill, who encouraged me to do my research as if I had nothing to lose. Their guidance, support and general wisdom have been invaluable resources throughout my journey. They were also both crucial to designing the experiments that led to understanding the phenomena of §4, which had remained puzzling anomalies for several months.

I am also grateful to my two additional mentors and long-time collaborators from Microsoft Research: Leonardo de Moura and Nikolaj Bjørner. Leonardo taught me the craft of theorem proving. The craft cannot yet be fully grasped by reading books, papers, or even code—much of it is still passed down in person, from master to apprentice. The two years I spent working closely with Leonardo on the Lean Theorem Prover were formative, and he and Lean both remain sources of inspiration. My collaboration with Nikolaj (which led to the results of §5) was among the most exciting stretches. Even though Nikolaj warned me that SAT was a "grad-student graveyard" and that "nothing ever works", he still spent months in the trenches with me, trying wild ideas and looking for an angle. Our impassioned high-five after first seeing the results of Figure 5.5 stands out as a highlight of my PhD experience.

I had two other influential mentors early on in my research career: Vikash Mansinghka at MIT and Christopher Ré at Stanford. Vikash taught me the value of grand, unifying ideas. He also taught me that it takes courage to champion an idea—even a great idea—while it is still in its early stages. Working with him on Venture was my first experience of the thrill of research, and much of my work is still motivated by

challenges we faced together. Chris taught me that good engineering requires thinking like a scientist and ablating one's methods until revealing their essence. I also gleaned a lot about how to be an effective human being by watching him operate.

My friends at Stanford provided ongoing stimulation and contributed to my research in many ways. I am especially grateful to Alexander Ratner, Cristina White, Nathaniel Thomas, and Jacob Steinhardt, for workshopping many of my ideas, papers, and talks, even when the subjects were far removed from their areas of interest. I am also grateful to my childhood friends—Josh Morgenthau, Josh Koenigsberg, Brent Katz, Jacob Luce, Simon Rich, and Christopher Quirk—for continuing to welcome me into their lives all these years. My research is still in the shadow of the schemes we pulled off together back in the day.

None of my quixotic research would have been possible without the financial support of my funders. I was lucky to be awarded a Stanford Graduate Fellowship as the Herbert Kunzel Fellow, as well as a research grant from the Future of Life Institute. I thank Professor Aiken for helping me with the grant application, and of course, I thank all the donors themselves.

Most of all, I am grateful to my family. My parents, Robert and Priscilla, have provided me with unwavering love and support throughout my entire life. Somehow they always believed that I would eventually find my way, even after an aberrant amount of meandering. Upon completing my Bachelor's Degree in History without having taken a single computer science course, I announced to them that I was going to become an AI researcher. Whereas many parents would question such a seemingly unreasonable declaration, they simply shrugged, nodded, and committed their support. Since then, they have patiently discussed many aspects of my research with me, despite admitting recently that they "have not understood a single word he said for many years". My wonderful sister, Margaret, has been a pillar in my life for as long as I can remember. My beloved niece, Olivia, has been a source of joy since the moment I first saw her.

Contents

Abstract **iv**

Acknowledgments **v**

1 Introduction **1**

2 Neural Networks **4**

2.1 Multilayer Perceptrons (MLPs) 5

2.2 Recurrent neural networks (RNNs) 5

2.3 Graph neural networks (GNNs) 6

3 The Satisfiability Problem **9**

3.1 Problem formulation 9

3.2 Conflict-Driven Clause Learning (CDCL) 11

3.3 Other Algorithms for SAT 13

3.3.1 Look-ahead solvers 13

3.3.2 Cube and conquer 14

3.3.3 Survey propagation (SP) 15

3.4 The International SAT Competition 15

4 Learning a SAT Solver From Single-Bit Supervision **17**

4.1 Overview 18

4.2 The Prediction Task 19

4.3 A Neural Network Architecture for SAT 20

4.4 Training data 23
4.5 Predicting satisfiability 24
4.6 Decoding satisfying assignments 25
4.7 Extrapolating 29
4.7.1 More iterations 29
4.7.2 Bigger problems 30
4.7.3 Different problems 32
4.8 Finding unsat cores 34
4.9 Related work 36
4.10 Discussion 38

5 Guiding CDCL with Unsat Core Predictions 39

5.1 Overview 39
5.2 Data generation 41
5.3 Neural Network Architecture 43
5.4 Hybrid Solving: CDCL and NeuroCore 46
5.5 Solver Experiments 48
5.6 Related Work 53
5.7 The Vast Design Space 54

6 Conclusion 57

List of Tables

4.1 NeuroSAT results on **SR**(40) 28

List of Figures

4.1 NeuroSAT overview 18
4.2 NeuroSAT solving a problem 26
4.3 NeuroSAT's literals clustering according to solution 28
4.4 NeuroSAT running on a satisfiable problem from **SR**(40) that it fails to solve 29
4.5 NeuroSAT solving a problem after 150 iterations 29
4.6 NeuroSAT on bigger problems 30
4.7 NeuroSAT on bigger problems 31
4.8 Example graph from the Forest-Fire distribution 31
4.9 Visualizing the diversity 33
4.10 NeuroUNSAT running on a pair of problems 36
4.11 NeuroUNSAT literals in core forming their own cluster 37

5.1 An overview of the NeuroCore architecture 45
5.2 Cactus plot of *neuro-minisat* on SATCOMP-2018 49
5.3 Scatter plot of *neuro-minisat* on SATCOMP-2018 49
5.4 Scatter plot of *neuro-glucose* on SATCOMP-2018 51
5.5 Cactus plot of *neuro-glucose* on hard scheduling problems 53

Chapter 1

Introduction

This dissertation presents two principal findings: first, that neural networks can learn to solve satisfiability (SAT) problems on their own without the help of hard-coded search procedures after only end-to-end training with minimal supervision, and second, that neural networks can be leveraged to improve high-performance SAT solvers on challenging and diverse real-world problems.

Neural networks and SAT solvers are two of the crowning achievements of computer science. Both approaches have emerged as widely applicable tools and have brought vital improvements to many real-world problems. Neural networks are routinely used to recognize objects in images [51, 35], convert spoken word to written text [39, 23, 33], translate between natural languages [15, 6, 82, 90], control robotic limbs [54, 58, 3, 55], and solve countless other problems in computer vision, natural language processing, and robotics. SAT solvers are routinely used to verify many different types of correctness properties of both hardware and software [16, 26, 52], manage network security protocols [42], solve hard planning [73] and scheduling problems [17, 31] arising from a variety of domains, and even prove long-standing conjectures in pure mathematics [28].

In the past few years, researchers have experimented with applying neural networks to increasingly challenging and diverse problems, including open-domain question answering [13], dialogue generation [57], visual question answering [44, 91], social network analysis [71], predicting properties of molecules [27], learning heuristics for

board games [83, 78, 79] and computer games [61, 93], learning high-performance indexing data structures [50], and countless others, with many promising results. Researchers have even experimented with applying neural networks to discrete search problems, such as program synthesis [67, 1], the traveling salesman problem [8, 21], first-order theorem proving [45, 59] and higher-order theorem proving [89, 87, 46], though in these domains the results have not yet been as convincing.

As a consequence of this exploration, the notion of neural network has continually expanded to include increasingly sophisticated architectures that bear little resemblance to the classic neural networks of previous decades. One recent trend has been to devise differentiable analogues of traditional data structures, such as differentiable stacks [30], queues [30], arrays [29], and key-value stores [85]. Many researchers in the field have taken to referring to neural networks as *differentiable programs*, to stress that neural networks are not one particular off-the-shelf tool but rather embody an open-ended modeling idiom that can be customized and extended in countless ways.

The increasing scope and ambition of the neural network community has raised critical scientific questions. What are the fundamental capabilities of neural networks? What kinds of problems are reasonable to try to solve with them? Can they solve problems that require search or reasoning?

In the SAT community, the past few years have been very different. Starting around 1992 and continuing until around 2015 was a veritable golden age for the SAT community. During this time there were major conceptual advances in algorithms, data structures, and heuristics, and the modern SAT algorithm (called CDCL, explained in §3) emerged and solidified. There were consistent, staggering improvements in the solvers themselves, and over these two decades SAT solvers went from an academic curiosity to an indispensable tool in industry. Yet since around 2015, the rate of performance improvements of SAT solvers has declined significantly [65]. This decline, along with the substantial successes and ever broadening scope of the neural network community has raised the critical question: can neural networks somehow be leveraged to improve high-performance SAT solvers?

In this work, we consider these questions in depth and make the following two contributions. First, in §4, we demonstrate a surprising capability of neural networks.

We show that a simple neural network architecture can learn to perform discrete search on its own without the help of hard-coded search procedures, even after only end-to-end training with minimal supervision. More specifically, we introduce a neural network architecture *NeuroSAT*, and show that when it is trained to predict the satisfiability of a particular kind of synthetic SAT problem—with only a single bit of supervision per problem—it learns to find satisfying assignments on its own. The same trained network can even solve SAT problems that are substantially larger and from entirely different domains than the problems it was trained on. The satisfiability problem is widely considered to be the canonical discrete search problem, and as it is **NP**-complete, searching for any kind of efficiently-checkable certificate in any context can be efficiently reduced to solving a SAT problem. Thus solving SAT problems is a proxy for solving all discrete search problems, and we conclude that in appropriate contexts, neural networks can learn to perform discrete search.

Second, in §5 we show how to leverage the same NeuroSAT architecture to improve high-performance SAT solvers. The neural network trained in §4 is remarkable from a scientific perspective, but on its own, it is nowhere near competitive with state-of-the-art solvers. We adopt a hybrid approach and use NeuroSAT to guide a high-performance SAT solver. First, we mine subproblems from existing benchmarks of SAT problems in order to generate hundreds of thousands of distinct problems. Second, we solve these problems using existing SAT solvers and in the process emit detailed logs of the search histories. Third, we analyze these logs to determine which variables would have been good to branch on in hindsight, and train a simplified NeuroSAT architecture to map SAT problems to the variables deemed good to branch on. Finally, we modify several state-of-the-art SAT solvers to prioritize branching on variables that NeuroSAT suggests. We find that each modified solver solves substantially more problems than the original on a benchmark of challenging and diverse real-world SAT problems.

Chapter 2

Neural Networks

We use the term *neural network* to refer to a computer program that is differentiable with respect to a set of real-valued, unknown parameters. There may be thousands, millions, or even billions of such parameters, and it would be impossible to specify them by hand. Instead, the practitioner specifies a second differentiable program called the *loss function*, which takes a collection of input/output pairs (*i.e.* training data), runs the neural network on the inputs, and computes a scalar score that measures how much the neural network's outputs disagree with the true outputs. Numerical optimization—usually stochastic gradient descent (SGD)—is then used to find values of the unknown parameters that make the loss function as small as possible. SGD can be applied automatically to such differentiable programs using any number of off-the-shelf software tools [18, 9, 2, 14, 75]. We use TensorFlow [2] to perform the optimization on all neural networks trained as part of this dissertation. Note that the adjective "neural" in the phrase "neural network" is purely historical, and does not indicate any concrete connection to the neurons of biological organisms.

Notation. Throughout this work, we signify the application of a neural network using function-call notation, where the different arguments to the network are implicitly concatenated. For example, if $N : \mathbb{R}^{d_1+d_2} \to \mathbb{R}^{d_{out}}$ is a neural network and $x_1 \in \mathbb{R}^{d_1}, x_2 \in \mathbb{R}^{d_2}$ are vectors, we write $N(x_1, x_2) \in \mathbb{R}^{d_{out}}$ to denote the result of

applying N to the concatenation of x_1 and x_2. For performance reasons, one almost never applies a neural network to an individual vector, and instead applies it to a *batch* of vectors of the same dimension, concatenated into a matrix. Thus if $X_1 \in \mathbb{R}^{k \times d_1}, X_2 \in \mathbb{R}^{k \times d_2}$, we write $N(X_1, X_2) \in \mathbb{R}^{k \times d_{out}}$ to denote the result of first concatenating X_1 and X_2 into a $\mathbb{R}^{k \times (d_1 + d_2)}$ matrix, applying N to the each of the k rows separately and then concatenating the k results back into a matrix.

2.1 Multilayer Perceptrons (MLPs)

The quintessential non-trivial (*i.e.* non-convex) neural network is the multilayer perceptron (MLP), also called a feed-forward network or a fully-connected network. An MLP takes as input a vector $x \in \mathbb{R}^{d_{in}}$ for a fixed d_{in}, and outputs a vector $y \in \mathbb{R}^{d_{out}}$ for a fixed d_{out}. It computes y from x by applying a sequence of (parameterized) affine transformations, each but the last followed by a component-wise nonlinear function called an *activation function*. The most common activation function is the rectified linear unit (ReLU), which is the identity function on positive numbers and sets all negative numbers to zero.

Even MLPs with only a single hidden layer have been shown to be universal function approximators [20, 43]. However, MLPs have a major limitation: a given MLP can only be applied to input vectors of a prespecified length. We are ultimately interested in applying neural networks to problems in Boolean satisfiability, which have different sizes and shapes, and so MLPs on their own are insufficient.

2.2 Recurrent neural networks (RNNs)

Recurrent neural networks (RNNs) are neural networks that operate on arbitrarily long sequences of inputs that all have the same size. An RNN takes a finite sequence of input vectors $\{x_i \in \mathbb{R}^{d_{in}}\}$ (for a fixed d_{in}) and outputs a vector $y \in \mathbb{R}^{d_{out}}$ (for a fixed d_{out}) that is intended to capture information about the entire input sequence. We refer to the output y as the *embedding* of the input sequence.

An RNN is parameterized by a neural network $N \in \mathbb{R}^{d_{out}+d_{in}} \to \mathbb{R}^{d_{out}}$. It computes

y from the input sequence $\{x_i\}$ as follows. First, it initializes a hidden-state vector $h \in \mathbb{R}_{d_{out}}$, usually to all zeros. Then it traverses the inputs x_i in sequence, each time updating h:

$$h \leftarrow N(h, x_i)$$

After traversing the input sequence, it returns the final value of its hidden-state h.

In the simplest case, N is simply an MLP. However, such "vanilla" RNNs are rarely used in practice due to the vanishing gradient problem [41]. Instead, N is usually a slightly more sophisticated neural network, designed to allow effectively ignoring certain inputs x_i and "remembering" previous values of h. The most notable RNN variant is called Long Short-Term Memory (LSTM) [40]. We make use of the LSTM in §4, but we omit the technical details of how they work since they are not relevant to our work.

Any digital information of any kind could in principle be embedded into a vector by an RNN, *e.g.* by representing the information as a sequence of its individual bits, with 0 represented as the vector $[1.0, 0.0]$ and 1 as the vector $[0.0, 1.0]$. However, this naïve approach may not yield useful embeddings for two reasons. First, the unfolded neural network will have depth at least as large as the length of the input, which would be enormous for many inputs of interest and would pose major challenges for SGD. Second, there may be many equivalent ways of encoding a particular type of structured information as a sequence of bits, and RNNs may make radically different predictions on different encodings of the same information.

2.3 Graph neural networks (GNNs)

Graphs are a good example of a class of inputs that are poorly represented by RNNs. Large graphs induce long sequences that pose major challenges for SGD. Moreover, the nodes and edges of a graph are semantically invariant to permutation, and yet the naïve sequence representation fails to encode this property.

With the differentiable programming mindset, it is easy to design a neural network that embeds arbitrarily large graphs, such that the depth of the unrolled network

need not scale with the size of the graphs, and such that by construction the network is invariant to permuting the nodes and edges of the graph. The standard neural network architecture for doing this is called the graph neural network (GNN), which we now describe. In contrast to the MLP, which takes fixed-sized vectors as input, and the RNN, which takes arbitrarily long sequences of fixed-size vectors as input, a GNN takes arbitrarily large graphs as inputs. The GNN represents each node in a graph—no matter how big the graph is—by a *fixed*-sized vector (called the *embedding* of the node). The GNN is parameterized by a simple neural network (*e.g.* an MLP) that specifies how a node updates its embedding given some (fixed-dimensional) aggregation of the embeddings of its neighbors in the graph.

More specifically, a GNN is parameterized by a neural network $N \in \mathbb{R}^{2d} \to \mathbb{R}^d$ (usually an MLP) where d is a fixed hyperparameter. It takes a graph $G = (V, E)$ as input, and computes an embedding vector in $\mathbb{R}^d$ for each node in the graph as follows. First, it allocates initial d-dimensional vectors x_i for each node in the graph, and either initializes them based on precomputed features of the respective nodes, or else arbitrarily (*e.g.* to all ones). Then, it iteratively refines these embeddings by a series of *message passing updates*. Each message passing update works as follows. For each node in the graph, an incoming *message* is computed by summing together the embeddings of the node's neighbors in the graph; then, the GNN's internal neural network N is applied to the node's current embedding and the incoming message in order to compute the node's new embedding:

$$x_i \leftarrow N\left(x_i, \sum_{j:(i,j)\in E} x_j\right)$$

After T such iterations (where T is another hyperparameter), the GNN returns the final embeddings. In practice, the final embeddings are used in different ways depending on the goal. For example, if the goal is to make a prediction about each node individually, then a downstream classifier can be applied to each node individually. On the other hand, if the goal is to make a prediction about the entire graph, the final embeddings can be aggregated into a single vector by reducing them with

any differentiable, associative and commutative operator (*e.g.* mean), and only then passed to a downstream classifier.

For best performance, the algorithm does not explicitly loop over the nodes during a given iteration. Instead, one concatenates all the embeddings $x_i \in \mathbb{R}^d$ together into a matrix $X \in \mathbb{R}^{|V| \times d}$, and performs all the updates in one high-level operation using the adjacency matrix A of the graph:

$$X \leftarrow N(X, AX)$$

Customizing architectures. We have provided descriptions of prototypical MLPs, RNNs, and GNNs, but we remark that the details can be easily customized in any number of ways. For example, in the GNN, the internal network N could be an MLP, or it could itself be a recurrent neural network (RNN) that maintains some state distinct from the embedding vectors it receives as input at each iteration. The embeddings could be averaged instead of summed when computing the messages, or their max or min could be taken. The number of iterations T could be a function of the graph, and could be scaled in proportion to the graph's diameter so that message passing can always propagate information between every pair of nodes. There could also be an embedding maintained for the entire graph, that sends a message to every node and that receives as a message the sum of the embeddings of all other nodes. Or, inspired by the data structures used in backtracking search, there could be a differentiable stack for the entire graph, that each node can pop from jointly, or perhaps pop from individually. Although none of these variations may be significant enough to merit names, they may have substantial impact on the performance of a neural network in some contexts. Thanks to the software packages that automate the computation of gradients, there is very little friction to experimenting with different variants, or even with entirely different kinds of architectures.

Chapter 3

The Satisfiability Problem

The propositional satisfiability problem (SAT) is one of the most fundamental problems of computer science. In [19], Stephen Cook showed that the problem is **NP**-complete, which means that searching for any kind of efficiently-checkable certificate in any context can be reduced to finding a satisfying assignment of a propositional formula. Although there are many problem formalisms that are **NP**-complete, SAT stands out for its extreme simplicity, and is widely considered to be the canonical **NP**-complete problem. In practice, search problems arising from a wide range of domains such as hardware and software verification, test pattern generation, planning, scheduling, and combinatorics are all routinely solved by constructing an appropriate SAT problem and then calling a SAT solver [28]. Modern SAT solvers based on backtracking search are extremely well-engineered and have been able to solve problems of practical interest with millions of variables [11].

3.1 Problem formulation

A formula of propositional logic is a Boolean expression built using the constants true (1) and false (0), variables, negations, conjunctions, and disjunctions. A *model* for a formula is an assignment of Boolean values to its variables such that the formula evaluates to 1. For example, for the formula $(x_1 \vee x_2 \vee x_3) \wedge \neg(x_1 \wedge x_2 \wedge x_3)$, any assignment that does not map x_1, x_2 and x_3 to the same value is a model. A model

is also called a *satisfying assignment*. We say a formula is *satisfiable* provided it has a satisfying assignment.

For every formula, there exists an equisatisfiable formula in *conjunctive normal form* (CNF), expressed as a conjunction of disjunctions of (possibly negated) variables. This transformation can be done in linear time such that the size of the resulting formula has only grown linearly with respect to the original formula [84]. Each conjunct of a formula in CNF is called a *clause*, and each (possibly negated) variable within a clause is called a *literal*. The formula above is equivalent to the CNF formula $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$, where $\overline{x}$ is shorthard for $\neg x$. SAT solvers only operate on formulae in CNF, which significantly simplifies their internal data structures. A formula in CNF has a satisfying assignment if and only if it has an assignment such that every clause has at least one literal mapped to 1. Note: throughout this dissertation, we use n to denote the number of variables in a CNF formula and m to denote the number of clauses.

A formula that is not satisfiable is called *unsatisfiable*. When a formula is satisfiable, we are usually interested in finding one or more satisfying assignments, each of which takes at most n bits of space and can be verified in linear time with respect to the size of the original problem. On the other hand, when a formula is unsatisfiable, we often want to find a *proof* that the formula is indeed unsatisfiable. There are many sound and complete proof systems for propositional logic. The *resolution* proof system is the standard choice for formulae in CNF, and has only a single inference rule:

$$\frac{C_1 \vee x \quad \overline{x} \vee C_2}{C_1 \vee C_2}$$

where C_1 and C_2 are disjunctions of literals and x is a variable. If a set of clauses is unsatisfiable, then the empty clause can be derived from them by resolution; however, such a proof may be exponentially large compared to the size of the original problem. As of this writing it is widely believed that $\mathbf{NP} \neq \mathbf{co{-}NP}$, in which case there could not exist any proof system for propositional logic that avoids a worst-case exponential blowup. There are several standardized proof formats for various extensions of resolution, and in §4 we make use of the Deletion Resolution Asymmetric Tautology

(DRAT) format [88], though the details are not relevant for our purposes.

If a formula in CNF is unsatisfiable, a proof of unsatisfiability may only refer to a small subset of the original clauses. For example, the formula

$$(x_{1,371} \vee \cdots \vee \overline{x_{23,345}}) \wedge \cdots \wedge x_1 \wedge \overline{x_1}$$

can be proven unsatisfiable in one resolution step by resolving the final two clauses, without even looking at any of the other clauses. A subset of clauses of a formula that are unsatisfiable is called an *unsatisfiable core*.

A *SAT problem* is a formula in CNF, where the goal is to determine if the formula is satisfiable, and if so, to produce a satisfying assignment, and if not, to produce a proof of unsatisfiability. We say that a method *solves* a particular (satisfiable) SAT problem if it finds a satisfying assignment for it, even if the method has no way of producing proofs and is not even complete.

3.2 Conflict-Driven Clause Learning (CDCL)

There have been many different approaches to solving SAT problems. Some solvers search for models by global search, whereas other solvers search for models by local search. Still others do not search for models at all and instead search for proofs. There have also been many more exotic approaches to solving throughout the years. Some perform variants of belief propagation [69, 70] on probabilistic interpretations of the SAT problem [12], while others perform optimization on numerical relaxations of the SAT problem [72].

Despite the diversity of approaches, modern high-performance SAT solvers are almost all based on the same approach: the conflict-driven clause learning (CDCL) algorithm. The basic idea is to search for models and proofs simultaneously. The search for models heuristically guides the search for (resolution) proof steps, while the inferred clauses help prune the search for models, all in a virtuous cycle.

Specifically, a CDCL solver maintains a database of clauses that includes the original clauses as well as clauses derived from them by resolution, and it also maintains

a *trail* of literals tentatively assigned to 1 that do not falsify any of the clauses. If the database ever includes the empty clause, then the solver has discovered a proof that the original problem is unsatisfiable. On the other hand, if the trail ever includes all of the variables in the problem, then the trail is a model and the problem is satisfiable.

Whenever a literal is added to the trail, the solver checks if any clause has all but one of its literals set to 0 under the trail. If it finds such a clause, it adds the remaining literal to the trail and repeats. This process is called *unit propagation* and is performed efficiently using clever data structures [63]. If unit propagation completes without finding a clause with all literals set to 0, a new *decision literal* is chosen (using some heuristic) and added to the trail, and the process continues.

If unit propagation does find a clause with all literals set to 0, then the trail must be inconsistent. Whereas a traditional backtracking algorithm might simply pop every literal up to and including the most recent decision literal and then continue the search, a CDCL solver does something more sophisticated: it analyzes the directed acyclic graph of propagations since the most recent decision literal, and performs a sequence of resolution steps to derive either the empty clause or a *learned* clause for which all but one literal was set to false at a previous decision level. If it derives the empty clause, then the problem is unsatisfiable. Otherwise, it adds the learned clause to the database, pops literals off the trail until the first decision level at which the learned clause would have propagated, and then propagates the learned clause. Only then does it resume the search for models. The name "conflict-driven clause learning" refers to this process of analyzing conflicts to infer new clauses by resolution.

Modern solvers have many additional features in addition to the main loop that we have just described. They periodically perform various forms of simplification, for example removing subsumed clauses and eliminating variables. They also periodically prune the learned clauses in the database, since this set grows very quickly and most of the clauses are unlikely to be useful. They also periodically *restart*, *i.e.* clear the trail entirely. Restarting for CDCL is not as drastic an action as it would be for a vanilla backtracking solver since the learned clauses are preserved, as are many statistics from the search history that influence various heuristics.

There are many crucial heuristic decisions that a CDCL solver must make, such

as which variable to branch on next, what polarity to set it to, which learned clauses to prune and when, when to simplify and in what way, and also when to restart. In §5, we focus on the first one: which variable to branch on next. The decision of which variable to branch on next has been the subject of intense study for decades and many approaches have been proposed. See [10] for a comprehensive overview. The most widely-used heuristic is a variant of the Variable State-Independent Decaying Sum (VSIDS) heuristic (first introduced in [63]) called Exponential VSIDS (EVSIDS). The EVSIDS score of a variable x after the tth conflict is defined by:

$$\mathbf{InConflict}(x,i) = \begin{cases} 1 & x \text{ was involved in the } i\text{th conflict} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathbf{EVSIDS}(x,t) = \sum_i \mathbf{InConflict}(x,i)\rho^{t-i}$$

Intuitively, it measures how many conflicts the variable has been involved in, with more recent conflicts weighted much more than past conflicts. Most other heuristics that have been proposed in the literature are based on the same intuition that it is good to branch on variables that have been involved in recent conflicts.

3.3 Other Algorithms for SAT

As mentioned above, many different approaches to solving SAT problems have been proposed throughout the years. We briefly describe three non-CDCL approaches that provide valuable context for our work.

3.3.1 Look-ahead solvers

One of the biggest meta-questions in all of discrete search is how much effort to put into each branching decision. CDCL solvers make variable branching decisions using cheap heuristics (*e.g.* EVSIDS) and so put very little effort into each individual branching decision. They make up for potentially poor variable branching decisions

by learning conflict clauses that prevent similar mistakes in the future, and of course by being able to try more decisions in the same amount of time than they could if they took longer for each decision.

There is a different family of SAT solvers called look-ahead solvers [36] that take the opposite approach, and put substantial effort into each variable branching decision. Look-ahead solvers do what their name suggests: for each variable branching decision, they consider a set of candidate variables, and for each one they "look ahead" to see how much progress setting that variable seems to make. Specifically, they set the variable to each of the Boolean values, and for each value they perform unit propagation and compute various metrics of the reduced problem. These metrics are transformed into scalar scores for each variable by (often very complicated) heuristics, and the look-ahead solver chooses the variable with the best score to branch on. The most widely-used look-ahead heuristic is called March [60]. We make use of March in §5, and refer to it in various places in this work, but the technical details of how it works are not relevant for our purposes.

Look-ahead solvers are not competitive with CDCL on industrial problems, but they are considered state-of-the-art on many classes of hard synthetic problems, and are an important building-block in cube and conquer solvers, which we now discuss.

3.3.2 Cube and conquer

As we discussed in §3.3.1, there are many classes of hard synthetic problems for which look-ahead solvers reliably beat CDCL solvers, and yet on most industrial problems of interest, look-ahead solvers are far inferior to CDCL solvers. Cube and conquer [37] is an increasingly popular paradigm for integrating the two approaches in the hopes of realizing the best of both worlds. The intuition behind cube and conquer is that the early branching decisions are disproportionately important and merit the extra work of expensive, global analysis, whereas the cheap, local heuristics of CDCL are preferable once enough literals have been set that the problem has become sufficiently easy. Specifically, cube and conquer solvers perform expensive branching decisions using look-ahead heuristics at the beginning of search—producing

so-called *cubes*—before trying to “conquer” the subproblems using CDCL. Cube and conquer is especially useful in a multicore or distributed setting, since the generated subproblems can be solved in parallel, but it has also been shown to improve serial solving time in many cases as well. Note that in the serial case, the CDCL solver may choose to keep conflict clauses across invocations, so the hybrid solver can effectively backjump (*i.e.* backtrack multiple steps at once) past the look-ahead/CDCL interface.

3.3.3 Survey propagation (SP)

There is an alternative family of approaches to solving SAT problems that involves performing variants of belief propagation [69, 70] on probabilistic interpretations of the SAT problem. Although these approaches are not used to solve real-world problems, one variant called survey propagation [12] has proved extremely effective at finding satisfying assignments for a particular distribution of random SAT problems. We briefly describe this family of approaches here, since it was an inspiration for the neural network architecture for SAT problems we present in §4.

The main idea is to maintain scalars for every literal, and to iteratively refine these scalars by passing scalar messages back and forth between the literals and the clauses they appear in. After the message passing, the literal scalars can be interpreted as defining marginal distributions over the variables, which can be used in different ways to guide a search procedure. The reader may see similarities to the GNN of §2.3. We stress that in contrast to neural networks, the survey propagation algorithm involves no learned parameters, and is instead derived from mathematical assumptions.

3.4 The International SAT Competition

The standard way of evaluating SAT solvers is the annual International SAT Competition (SATCOMP) [74]. Every year, hundreds of new benchmark SAT problems are submitted from a wide range of different domains. For example, the 2017 benchmarks include problems from bounded model checking (both hardware and software), railway system safety verification, prime factorization, SHA-1 preimage attacks, Rubik's cube

puzzles, and crafted combinatorial and graph coloring problems, among many others [7]. The 2018 benchmarks include problems arising from proving theorems about bit-vectors, reversing Cellular Automata, verifying floating-point computations, finding efficient polynomial multiplication circuits, mining Bitcoins, allocating time slots to students with preferences, and finding Hamiltonian cycles as part of a puzzle game, among many others [38].

Every year many solvers are submitted to the competition. In the main track of the competition, each solver runs on each (previously unseen) problem for up to 5,000 seconds on a single CPU core. The solvers are officially evaluated by a scalar metric called PAR-2, which is the sum of all its runtimes for solved instances plus twice the timeout for all unsolved instances. However, the PAR-2 metric is hard for humans to interpret, and so it is common to compare solvers based on the number of solved instances and to consider more fine-grained metrics as appropriate to the context. Note that since the problems are new and from highly varying domains each year, the relative scores of different fixed solvers may vary from year to year as well.

Chapter 4

Learning a SAT Solver From Single-Bit Supervision

In §1, we discussed how the increasing scope and ambition of the neural network community has raised a critical scientific question: *what are the fundamental capabilities of neural networks*? This question is broad and open-ended, and so in this work we consider one potential capability in particular: the ability to search. Since SAT is the canonical search problem (and furthermore, is **NP**-complete), we consider the concrete proxy question: *can a neural network learn to solve SAT problems without being explicitly supervised to do so?*

In this chapter, we answer the question in the affirmative, and show that a simple neural network architecture can indeed learn to perform discrete search on its own without the help of hard-coded search procedures after only end-to-end training with minimal supervision.

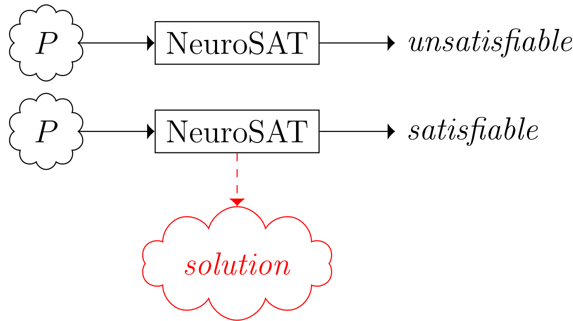
The work in this chapter is the result of a collaboration with Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill, and benefited from discussions with Steve Mussmann, Alexander Ratner, Nathaniel Thomas, Vatsal Sharan, Cristina White, William Hamilton, Geoffrey Irving and Arun Chaganty. A paper describing it was published in the Seventh International Conference on Learning Representations (ICLR-2019) [77].

4.1 Overview

We develop a novel GNN, *NeuroSAT*, and train it as a classifier to predict satisfiability on a dataset of random SAT problems. We provide NeuroSAT with only a single bit of supervision for each SAT problem that indicates whether or not the problem is satisfiable. When making a prediction about a new SAT problem, we find that NeuroSAT guesses *unsatisfiable* with low confidence until it finds a solution, at which point it converges and guesses *satisfiable* with very high confidence. The solution itself can almost always be automatically decoded from the network's activations, making NeuroSAT an end-to-end SAT solver. See Figure 4.1 for an illustration of the train and test regimes.

$$\left\{ \begin{array}{ll} \text{Input:} & \text{SAT problem P} \\ \text{Output:} & \mathbb{1}\left\{P \text{ is satisfiable}\right\} \end{array} \right\}$$

(a) Train



(b) Test

Figure 4.1: We train NeuroSAT to predict whether SAT problems are satisfiable, providing only a single bit of supervision for each problem. At test time, when NeuroSAT predicts *satisfiable*, we can almost always extract a satisfying assignment from the network's activations. The problems at test time can also be substantially larger, more difficult, and even from entirely different domains than the problems seen during training.

Although it is not competitive with state-of-the-art SAT solvers, NeuroSAT can

solve SAT problems that are substantially larger and more difficult than it ever saw during training by performing more iterations of message passing. Despite only running for a few dozen iterations during training, at test time NeuroSAT continues to find solutions to harder problems after hundreds and even thousands of iterations. The learning process has yielded not a traditional classifier but rather a procedure that can be run indefinitely to search for solutions to problems of varying difficulty.

Moreover, NeuroSAT generalizes to entirely new domains. Since NeuroSAT operates on SAT problems and since SAT is **NP**-complete, NeuroSAT can be queried on SAT problems encoding any kind of search problem for which certificates can be checked efficiently. Although we train it using only problems from a single random problem generator, at test time it can solve SAT problems encoding graph coloring, clique detection, dominating set, and vertex cover problems, all on a range of distributions over small random graphs.

The same neural network architecture can also be used to help construct proofs for unsatisfiable problems. When we train it on a different dataset in which every unsatisfiable problem contains a small unsat core (call this trained model *NeuroUNSAT*), it learns to detect these unsat cores instead of searching for satisfying assignments. Just as we can extract solutions from NeuroSAT's activations, we can extract the variables involved in the unsat core from NeuroUNSAT's activations. When the number of variables involved in the unsat core is small relative to the total number of variables, knowing which variables are involved in the unsat core can enable constructing a resolution proof more efficiently.

4.2 The Prediction Task

For a SAT problem P, we define $\phi(P)$ to be true if and only if P is satisfiable. Our first goal is to learn a classifier that approximates ϕ. Given a distribution Ψ over SAT problems, we can construct datasets $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ with examples of the form $(P, \phi(P))$ by sampling problems $P \sim \Psi$ and computing $\phi(P)$ using an existing SAT solver. At test time, we get only the problem P and the goal is to predict $\phi(P)$, *i.e.* to determine if P is satisfiable. We ultimately care about the *solving task*, which also

includes finding solutions to satisfiable problems.

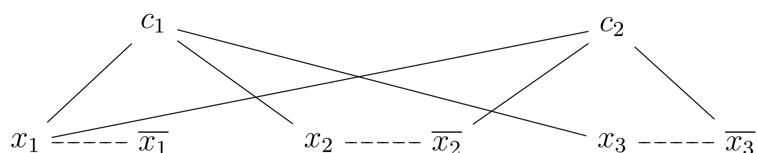
4.3 A Neural Network Architecture for SAT

A SAT problem has a simple syntactic structure and therefore could be encoded into a vector space using standard methods such as an RNN. However, the semantics of propositional logic induce rich invariances that such a syntactic method would ignore, such as permutation invariance and negation invariance. Specifically, the satisfiability of a formula is not affected by permuting the variables (*e.g.* swapping x_1 and x_2 throughout the formula), by permuting the clauses (*e.g.* swapping the first clause with the second clause), or by permuting the literals within a clause (*e.g.* replacing the clause $(x_1 \vee \overline{x_2})$ with $(\overline{x_2} \vee x_1)$. The satisfiability of a formula is also not affected by negating every literal corresponding to a given variable (*e.g.* negating all occurrences of x_1 in the SAT problem $(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3})$ to yield the new problem $(\overline{x_1} \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_3})$.

We now describe our neural network architecture, NeuroSAT, that enforces both permutation invariance and negation invariance. We represent a Boolean formula in CNF by an undirected graph with nodes for every literal and clause, with two different types of edges: *occurrence* edges between literals and the clauses they appear in, and *flip* edges between literals and their negations. For example, we represent the formula

$$\underbrace{(x_1 \vee x_2 \vee x_3)}_{c_1} \wedge \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{c_2}$$

by the following graph:



where the solid lines denote occurrence edges and the dashed lines denote flip edges. Let n and m be the number of variables and clauses in the formula respectively. To

pass as input to our neural network, we encode the graph of a Boolean formula as an $m \times 2n$ sparse adjacency matrix $\mathcal{G}$, where (for reasons we will discuss shortly) the first n columns represent the positive literals and the last n columns represent the negative literals. For example, we represent the graph above as the following 2×6 (sparse) matrix:

$\mathcal{G}$:

	x_1	x_2	x_3	$\overline{x_1}$	$\overline{x_2}$	$\overline{x_3}$
c_1	1	1	1	0	0	0
c_2	1	0	0	0	1	1

Note that the flip edges are only implicit in this representation.

To a first approximation, our model is simply a GNN (see §2.3) on this graph representation of the SAT problem. It iteratively refines a vector space embedding for each node in the graph by passing "messages" back and forth along the edges. At the beginning of every iteration, we have an embedding for every literal and every clause. An iteration consists of two stages. First, each clause receives messages from its neighboring literals and updates its embedding accordingly. Next, each literal receives messages from its neighboring clauses as well as from its complement and then updates its embedding accordingly.

More formally, our model is parameterized by two vectors ($\mathbf{C}_{\text{init}}$, $\mathbf{L}_{\text{init}}$), three multilayer perceptrons ($\mathbf{L}_{\text{msg}}$, $\mathbf{C}_{\text{msg}}$, $\mathbf{L}_{\text{vote}}$) and two layer-norm LSTMs [5, 40] ($\mathbf{C}_{\text{update}}$, $\mathbf{L}_{\text{update}}$). The network computes forward as follows. First, it initializes two matrices $C^{(0)} \in \mathbb{R}^{m \times d}$ and $L^{(0)} \in \mathbb{R}^{2n \times d}$ by tiling $\mathbf{C}_{\text{init}}$ and $\mathbf{L}_{\text{init}}$ respectively. Each row of C corresponds to a clause, while each row of L corresponds to a literal:

$$C = \begin{bmatrix} — & c_1 & — \\ & \vdots & \\ — & c_m & — \end{bmatrix}, \qquad L = \begin{bmatrix} — & x_1 & — \\ & \vdots & \\ — & x_n & — \\ — & \overline{x_1} & — \\ & \vdots & \\ — & \overline{x_n} & — \end{bmatrix}$$

We refer to the row corresponding to a clause c or a literal ℓ as the *embedding* of

that clause or literal. Define the operation Flip to swap the first half of the rows of a matrix with the second half, so that in Flip(L), each literal's row is swapped with its negation's:

$$\text{Flip}(L) = \begin{bmatrix} - & \overline{x_1} & - \\ & \vdots & \\ - & \overline{x_n} & - \\ - & x_1 & - \\ & \vdots & \\ - & x_n & - \end{bmatrix}$$

Next, after initializing C and L, the network performs T iterations of message passing, where a single iteration consists of the following two steps. First, each clause updates its embedding based on the current embeddings of the literals it contains: $\forall c, c \leftarrow \mathbf{C}_{\text{update}}\left(c, \sum_{\ell \in c} \mathbf{L}_{\text{msg}}(\ell)\right)$. Next, each literal updates its embedding based on the current embeddings of the clauses it occurs in, as well as the current embedding of its negation: $\forall \ell, \ell \leftarrow \mathbf{L}_{\text{update}}\left(\ell, \sum_{c | \ell \in c} \mathbf{C}_{\text{msg}}(c), \overline{\ell}\right)$. Note that we can compute both updates efficiently using our adjacency matrix $\mathcal{G}$ as follows:

$$\begin{aligned} C^{(t+1)} &\leftarrow \mathbf{C}_{\text{update}}\left(C^{(t)}, \mathcal{G}\mathbf{L}_{\text{msg}}(L^{(t)})\right) \\ L^{(t+1)} &\leftarrow \mathbf{L}_{\text{update}}\left(L^{(t)}, \mathcal{G}^{\top}\mathbf{C}_{\text{msg}}(C^{(t+1)}), \text{Flip}(L^{(t)})\right) \end{aligned}$$

After T iterations, we compute $L_*^{(T)} \leftarrow \mathbf{L}_{\text{vote}}(L^{(T)}) \in \mathbb{R}^{2n}$, which consists of a single scalar for each literal (the literal's *vote*), and then we compute the average of the literal votes $y^{(T)} \leftarrow \text{mean}(L_*^{(T)}) \in \mathbb{R}$. We train the network to minimize the sigmoid cross-entropy loss between the true label $\phi(P)$ and the logit $y^{(T)}$.

Our architecture enforces permutation invariance by always embedding each node's neighborhood into a fixed dimensional space with an associative and commutative operator (*i.e.* sum), and never operating on nodes or edges in an order-dependent way. Likewise, it enforces negation invariance by treating all literals the same no matter whether they originated as a positive or negative occurrence of a variable. However,

there are some invariances afforded by the semantics of satisfiability that our architecture not only fails to enforce but makes impossible to learn. For example, since our model does not allow any communication between disconnected components, and since we reduce the votes with mean at the end instead of min (which we do only because min is harder to train), our model has no way to learn that a single disconnected component that is *unsat* must necessarily override all other votes for *sat*. However, disconnected components can be easily preprocessed away in linear time. It is also impossible for our architecture to learn in full generality that duplicating literals, duplicating clauses, and adding clauses with complementary literals (*e.g.* $(x_1 \vee x_5 \vee \overline{x_1})$) all have no effect. Duplicate literals and clauses would be ignored if we reduced both the votes and the messages with min or max. We see no easy way to ensure that clauses with complementary literals are ignored, though removing such clauses is a common (linear-time) preprocessing step in many SAT solvers.

We stress that none of the learned parameters depend on the size of the SAT problem and that a single model can be trained and tested on problems of arbitrary and varying sizes. At both train and test time, the input to the model is the adjacency matrix $\mathcal{G}$ that represents an arbitrary SAT problem over any number of literals and clauses. The learned parameters only determine how each individual literal and clause behaves in terms of its neighbors in the graph. Variation in problem size is handled by the aggregation operators: we sum the outgoing messages of each of a node's neighbors to form the incoming message, and we take the mean of the literal votes at the end of message passing to form the logit $y^{(T)}$.

4.4 Training data

One can easily construct distributions over SAT problems for which it would be possible to predict satisfiability with perfect accuracy based only on crude statistics; however, a neural network trained on such a distribution would be unlikely to generalize to problems from other domains. To force our network to learn something substantive, we create a distribution $\mathbf{SR}(n)$ over pairs of random SAT problems on n variables with the following property: one element of the pair is satisfiable, the other

is unsatisfiable, and the two differ by negating only a single literal occurrence in a single clause. To sample a pair of problems from $\mathbf{SR}(n)$, we start by adding random clauses one by one to an initially empty SAT problem. We sample each clause by first sampling a small integer k (with mean a little over 4)[1], then sampling k variables out of $\{x_1, \ldots, x_n\}$ uniformly at random without replacement, and finally negating each one with independent probability 50%. We continue to sample clauses c_i in this fashion, adding each one to the SAT problem and re-querying a traditional SAT solver each time (we used MiniSat [24]), until adding the clause c_m finally makes the problem unsatisfiable. Since $(c_1 \wedge \cdots \wedge c_{m-1})$ had a satisfying assignment, negating a single literal in c_m to yield $c_{m'}$ creates a satisfiable problem $(c_1 \wedge \cdots \wedge c_{m-1} \wedge c_{m'})$. The pair $(c_1 \wedge \cdots \wedge c_{m-1} \wedge c_m)$ and $(c_1 \wedge \cdots \wedge c_{m-1} \wedge c_{m'})$ is a sample from $\mathbf{SR}(n)$.

4.5 Predicting satisfiability

Although our ultimate goal is to solve SAT problems arising from a variety of domains, we begin by training NeuroSAT as a classifier to predict satisfiability on $\mathbf{SR}(40)$. Problems in $\mathbf{SR}(40)$ are small enough to be solved efficiently by modern SAT solvers—a fact we rely on to generate the problems—but the classification problem is highly non-trivial from a machine learning perspective. Each problem has 40 variables and over 200 clauses on average, and the positive and negative examples differ by negating only a single literal occurrence out of a thousand. We were unable to train an LSTM on a many-hot encoding of clauses (specialized to problems with 40 variables) to predict with >50% accuracy on its training set. Even the popular SAT solver MiniSat [24] needs to backjump almost ten times on average, and needs to perform over a hundred primitive logical inferences (*i.e.* unit propagations) to solve each problem.

We instantiated the NeuroSAT architecture described in §4.3 with d=128 dimensions for the literal embeddings, the clause embeddings, and all the hidden units; 3

[1] We use $1 + \mathbf{Bernoulli}(0.7) + \mathbf{Geo}(0.4)$ so that we generate clauses of varying size but with only a small number of clauses of length 2, since too many random clauses of length 2 make the problems too easy on average.

hidden layers and a linear output layer for each of the MLPs $\mathbf{L}_{\text{msg}}$, $\mathbf{C}_{\text{msg}}$, and $\mathbf{L}_{\text{vote}}$; and rectified linear units for all non-linearities. We regularized by the ℓ_2 norm of the parameters scaled by 10^{-10}, and performed $T{=}26$ iterations of message passing on every problem. We trained our model using the Adam optimizer [47] with a learning rate of 2×10^{-5}, clipping the gradients by global norm with clipping ratio 0.65 [68]. We batched multiple problems together, with each batch containing up to 12,000 nodes (*i.e.* literals plus clauses). To accelerate the learning, we sampled the number of variables n uniformly from between 10 and 40 during training (*i.e.* we trained on $\mathbf{SR}(\mathbf{U}(10, 40))$). We trained on millions of problems.

After training, NeuroSAT is able to classify the $\mathbf{SR}(40)$ test set correctly with 85% accuracy. In the next section, we examine how NeuroSAT manages to do so and show how we can decode solutions to satisfiable problems from its activations. Note: for the rest of this chapter, *NeuroSAT* refers to the specific trained model that has only been trained on $\mathbf{SR}(\mathbf{U}(10, 40))$.

4.6 Decoding satisfying assignments

Let us try to understand what NeuroSAT (trained on $\mathbf{SR}(\mathbf{U}(10, 40))$) is computing as it runs on new problems at test time. For a given run, we can compute and visualize the vector of literal votes $L_*^{(t)} \in \mathbb{R}^{2n} \leftarrow \mathbf{L}_{\text{vote}}(L^{(t)})$ at every iteration t. Figure 4.2 illustrates the sequence of literal votes $L_*^{(1)}$ to $L_*^{(24)}$ as NeuroSAT runs on a satisfiable problem from $\mathbf{SR}(20)$. For clarity, we reshape each $L_*^{(t)}$ to be an $\mathbb{R}^{n \times 2}$ matrix so that each literal is paired with its complement; specifically, the ith row contains the scalar votes for x_i and $\overline{x_i}$. Here white represents zero, blue negative and red positive. For several iterations, almost every literal is voting *unsat* with low confidence (light blue). Then a few scattered literals start voting *sat* for the next few iterations, but not enough to affect the mean vote. Suddenly, there is a phase transition and all the literals (and hence the network as a whole) start to vote *sat* with very high confidence (dark red). After the phase transition, the literal votes converge and the network stops evolving.

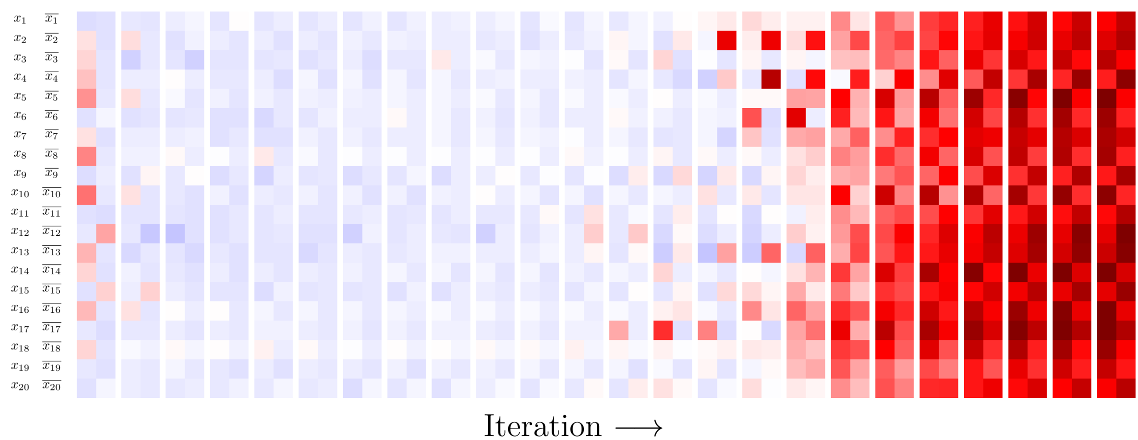


Figure 4.2: The sequence of literal votes $L_*^{(1)}$ to $L_*^{(24)}$ as NeuroSAT runs on a satisfiable problem from $\mathbf{SR}(20)$. For clarity, we reshape each $L_*^{(t)}$ to be an $\mathbb{R}^{n\times 2}$ matrix so that each literal is paired with its complement; specifically, the ith row contains the scalar votes for x_i and $\overline{x_i}$. Here white represents zero, blue negative and red positive. For several iterations, almost every literal is voting *unsat* with low confidence (light blue). Then a few scattered literals start voting *sat* for the next few iterations, but not enough to affect the mean vote. Suddenly there is a phase transition and all the literals (and hence the network as a whole) start to vote *sat* with very high confidence (dark red). After the phase transition, the literal votes converge and the network stops evolving.

NeuroSAT seems to exhibit qualitatively similar behavior on every satisfiable problem that it predicts correctly. NeuroSAT's behavior on the other problems is similar except without the phase change: NeuroSAT continues to guess *unsat* with low confidence for as many iterations as it runs for. NeuroSAT never becomes highly confident that a problem is *unsat*, and it almost never guesses *sat* on an *unsat* problem. These results suggest that NeuroSAT searches for a certificate of satisfiability, and that it only guesses *sat* once it has found one.

Let us look more carefully at the literal votes $L_*^{(24)}$ from Figure 4.2 after convergence. Note that most of the variables have one literal vote distinctly darker than the other. Moreover, most of the darker votes are approximately equal to each other, and most of the lighter votes are approximately equal to each other as well. Thus the votes seem to encode one bit for each variable. It turns out that these bits encode a satisfying assignment in this case, but they do not do so reliably in general.

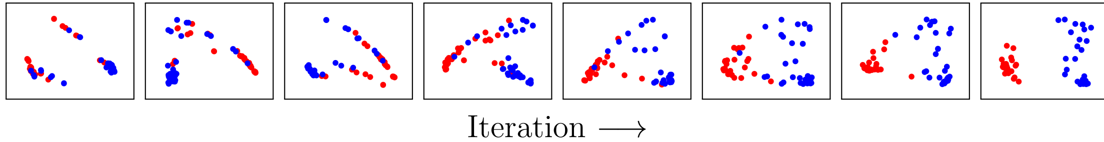
There is another way to visualize NeuroSAT's internal representations that sheds more light on this phenomenon. Recall from §4.3 that NeuroSAT projects the higher dimensional literal embeddings $L^{(T)} \in \mathbb{R}^{2n \times d}$ to the literal votes $L_*^{(T)}$ using the MLP $\mathbf{L}_{\text{vote}}$. Figure 4.3 illustrates the two-dimensional PCA embeddings for $L^{(12)}$ to $L^{(26)}$ (skipping every other time step) as NeuroSAT runs on a satisfiable problem from $\mathbf{SR}(40)$. Blue and red dots indicate literals that are set to 0 and 1 in the satisfying assignment that it eventually finds, respectively. The blue and red dots cannot be linearly separated until the phase transition at the end, at which point they form two distinct clusters according to the satisfying assignment. We observe a similar clustering almost every time the network guesses *sat*. The literal votes $L_*^{(T)}$ only ever encode the satisfying assignment by chance when the projection $\mathbf{L}_{\text{vote}}$ happens to preserve this clustering.

Our analysis suggests a more reliable way to decode solutions from NeuroSAT's internal activations: 2-cluster $L^{(T)}$ to get cluster centers Δ_1 and Δ_2, partition the variables according to the predicate $\|x_i - \Delta_1\|^2 + \|\overline{x_i} - \Delta_2\|^2 < \|x_i - \Delta_2\|^2 + \|\overline{x_i} - \Delta_1\|^2$, and then try both candidate assignments that result from mapping the partitions to truth values. This decoding procedure (using k-means to find the two cluster centers) successfully decodes a satisfying assignment for over 70% of the satisfiable problems in

Trained on:	$\mathbf{SR}(\mathbf{U}(10, 40))$
Trained with:	26 iterations
Tested on:	$\mathbf{SR}(40)$
Tested with:	26 iterations
Overall test accuracy:	85%
Accuracy on *unsat* problems:	96%
Accuracy on *sat* problems:	73%
Percent of *sat* problems solved:	**70%**

Table 4.1: NeuroSAT's performance at test time on $\mathbf{SR}(40)$ after training on $\mathbf{SR}(\mathbf{U}(10, 40))$. It almost never guesses *sat* on unsatisfiable problems. On satisfiable problems, it correctly guesses *sat* 73% of the time, and we can decode a satisfying assignment for 70% of the satisfiable problems by clustering the literal embeddings $L^{(T)}$ as described in §4.6.

the $\mathbf{SR}(40)$ test set. Table 4.1 summarizes the results when training on $\mathbf{SR}(\mathbf{U}(10, 40))$ and testing on $\mathbf{SR}(40)$.



Iteration $\longrightarrow$

Figure 4.3: PCA projections for the high-dimensional literal embeddings $L^{(12)}$ to $L^{(26)}$ (skipping every other time step) as NeuroSAT runs on a satisfiable problem from $\mathbf{SR}(40)$. Blue and red dots indicate literals that are set to 0 and 1 in the satisfying assignment that it eventually finds, respectively. The blue and red dots cannot be linearly separated until the phase transition at the end, at which point they form two distinct clusters according to the satisfying assignment.

Recall that at training time, NeuroSAT is only given *a single bit* of supervision for each SAT problem. Moreover, the positive and negative examples in the dataset differ only by the placement of a single edge. NeuroSAT has learned to search for satisfying assignments as a way of explaining that single bit of supervision.

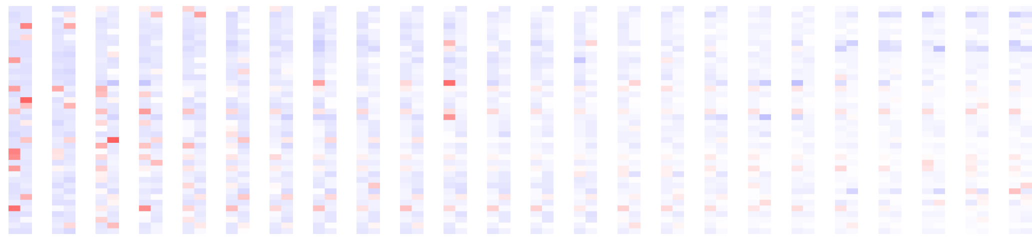


Figure 4.4: NeuroSAT running on a satisfiable problem from **SR**(40) that it fails to solve.

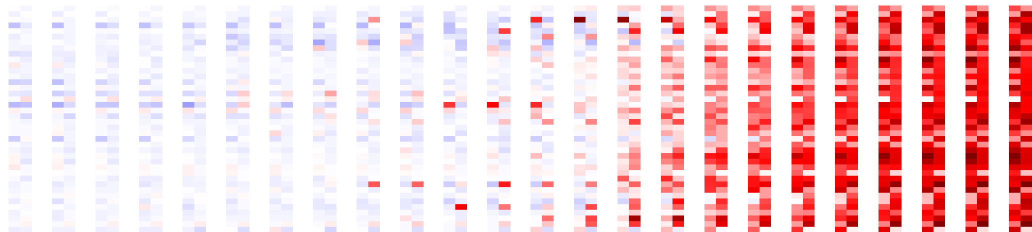


Figure 4.5: NeuroSAT running on the same satisfiable problem as in Figure 4.4, beginning with iteration 137. NeuroSAT eventually solves the problem after roughly 150 iterations.

4.7 Extrapolating

4.7.1 More iterations

What about the 30% of satisfiable problems in **SR**(40) that NeuroSAT fails to solve? Figure 4.4 shows NeuroSAT running on a satisfiable problem from **SR**(40) that it fails to solve after 26 iterations. It turns out that for this problem and many others, if we simply continue running NeuroSAT for more iterations, NeuroSAT eventually solves the problem. Figure 4.5 shows NeuroSAT running on the same problem as in Figure 4.4, starting at iteration 137, and undergoing the familiar phase transition around iteration 150. Figure 4.6 shows the percentage of problems solved in **SR**(40) as a function of the number of iterations that NeuroSAT runs for. We see that when it runs for 1,000 iterations, NeuroSAT can solve almost all (satisfiable) problems in **SR**(40).

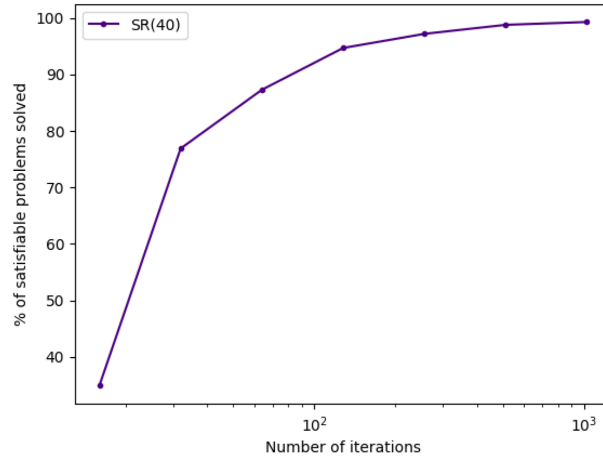


Figure 4.6: NeuroSAT's success rate on $\mathbf{SR}(40)$ as a function of the number of iterations T. Even though we only train NeuroSAT with T=26 iterations, it continues to search for solutions productively for at least 50 times as many iterations at test time, and is able to solve almost all satisfiable problems in $\mathbf{SR}(40)$.

4.7.2 Bigger problems

Even though we only train NeuroSAT on $\mathbf{SR}(\mathbf{U}(10, 40))$, it is able to solve SAT problems sampled from $\mathbf{SR}(n)$ for n much larger than 40 by simply running for more iterations of message passing. Figure 4.7 shows NeuroSAT's success rate on $\mathbf{SR}(n)$ for a range of n as a function of the number of iterations T. For $n = 200$, there are 2^{160} times more possible assignments to the variables than any problem it saw during training, and yet it can solve 25% of the satisfiable problems in $\mathbf{SR}(200)$ by running for four times more iterations than it performed during training. On the other hand, when restricted to the number of iterations it was trained with, it solves under 10% of them. Thus we see that its ability to solve bigger and harder problems depends on the fact that the dynamical system it has learned encodes generic procedural knowledge that can operate effectively over a wide range of time frames.

Although we found it surprising that it could solve any problems at all for large n (*i.e.* n=200), it is still natural to wonder why the curves seem to plateau well below 100%. We explored this question as follows. We ran NeuroSAT on a collection of problems from $\mathbf{SR}(40)$, and after every time step that NeuroSAT did not predict

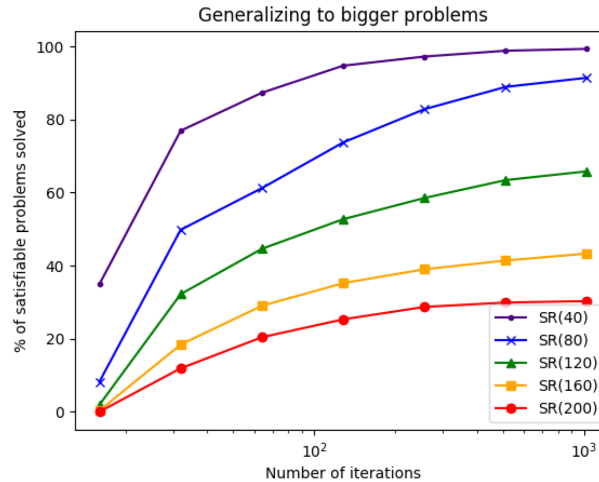


Figure 4.7: NeuroSAT's success rate on $\mathbf{SR}(n)$ for a range of n as a function of the number of iterations T. Even though we only train NeuroSAT on $\mathbf{SR}(40)$ and below, it is able to solve SAT problems sampled from $\mathbf{SR}(n)$ for n much larger than 40 by simply running for more iterations.

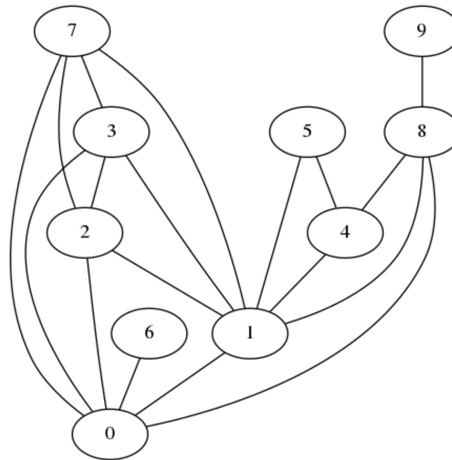


Figure 4.8: Example graph from the Forest-Fire distribution. The graph has a coloring for $k \geq 5$, a clique for $k \leq 3$, a dominating set for $k \geq 3$, and a vertex cover for $k \geq 6$. However, these properties are not perceptually obvious and require deliberate computation to determine.

sat we nonetheless attempted to decode a candidate assignment using the process described in §4.6. We then looked at the number of variables flipped and the percentage of clauses satisfied at each round by the candidate assignment. We found that the number of variables flipped almost always decreased monotonically, while the percentage of satisfied clauses almost always increased monotonically. These findings suggest that NeuroSAT has learned an analogue of simulated annealing, and that it effectively becomes more cautious as the number of iterations increases and as the candidate assignment satisfies a greater percentage of clauses. Thus it seems that NeuroSAT (at least when trained on $\mathbf{SR}(\mathbf{U}(10, 40))$) is susceptible to getting stuck in local optima. It also suggests a potential way to improve the architecture: give it access to a source of randomness, so that it can learn to inject noise into its embeddings as a way of breaking out of local optima and re-increasing its effective temperature.

4.7.3 Different problems

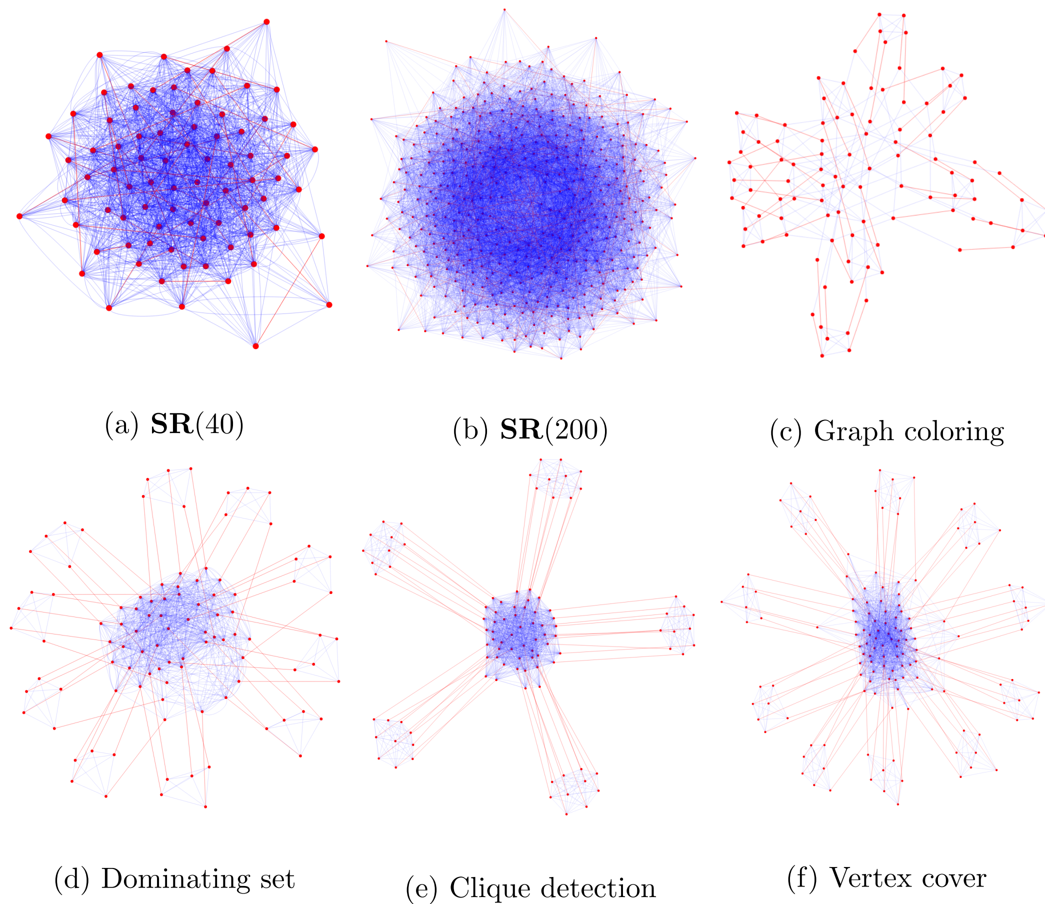
Every problem in **NP** can be reduced to SAT in polynomial time, and SAT problems arising from different domains may have radically different structural and statistical properties. Even though NeuroSAT has learned to search for satisfying assignments on problems from $\mathbf{SR}(n)$, we may still find that the dynamical system it has learned only works properly on problems similar to those it was trained on.

To assess NeuroSAT's ability to extrapolate to different classes of problems, we generated problems in several other domains and then encoded them all into SAT problems (using standard encodings). In particular, we started by generating one hundred graphs from each of six different random graph distributions (Barabasi, Erdös-Renyi, Forest-Fire, Random-k-Regular, Random-Static-Power-Law, and Random-Geometric).[2] We found parameters for the random graph generators such that each graph has ten nodes and seventeen edges on average. For each graph in each collection, we generated graph coloring problems ($3 \leq k \leq 5$), dominating-set problems ($2 \leq k \leq 4$)), clique-detection problems ($3 \leq k \leq 5$), and vertex cover problems ($4 \leq k \leq 6$).[3] We chose the range of k for each problem to include the threshold

[2] See [64] for an overview of random graph distributions.

[3] See [56] for an overview of these problems as well as the standard encodings.

for most of the graphs while avoiding trivial problems such as 2-clique. As before, we used MiniSat [24] to determine satisfiability. Figure 4.8 shows an example graph from the distribution. Note that the trained network does not know anything a priori about these tasks; the generated SAT problems need to encode not only the graphs themselves but also formal descriptions of the tasks to be solved. Figure 4.9 shows visualizations of the various graph distributions that highlight their diversity.



(a) **SR**(40) (b) **SR**(200) (c) Graph coloring

(d) Dominating set (e) Clique detection (f) Vertex cover

Figure 4.9: Visualizations of the various distributions of SAT problems discussed in this chapter. Note the distinct difference between the tangled, structureless **SR**(n) problems and the highly structured problems arising from the other domains.

Out of the 7,200 generated problems, we kept only the 4,888 satisfiable problems. On average these problems contained over two and a half times as many clauses as

the problems in $\mathbf{SR}(40)$. We ran NeuroSAT for 512 iterations on each of them and found that we could successfully decode solutions for 85% of them. In contrast, survey propagation (see §3.3.3), the standard (learning-free) message passing algorithm for satisfiability, does not on its own converge to a satisfying assignment for a single problem in the set.[4] This suggests that NeuroSAT has not simply found a way to approximate SP, but rather has synthesized a qualitatively different algorithm.

Extrapolation is possible. One of the most notorious weaknesses of neural networks has always been their failure to extrapolate in reasonable ways on test data that is insufficiently similar to the training data. The extrapolation abilities of NeuroSAT presented in this section show that poor generalization is not an inherent limitation of neural networks but rather is a property of specific neural network architectures and specific training regimes. The ability to extrapolate can be cultivated in a neural network by building better inductive biases into the architecture itself, and by exerting careful control over the training data to ensure that large classes of undesirable hypotheses are not able to achieve relatively small training loss.

4.8 Finding unsat cores

NeuroSAT (trained on $\mathbf{SR}(\mathbf{U}(10, 40))$) can find satisfying assignments but is not helpful in constructing proofs of unsatisfiability. When it runs on an unsatisfiable problem, it keeps searching for a satisfying assignment indefinitely and non-systematically. However, when we train the same architecture on a dataset in which each unsatisfiable problem has a small unsat core, it learns to detect these unsat cores instead of searching for satisfying assignments. The literals involved in the unsat core can almost always be decoded from its internal activations. When the number of literals involved in the unsat core is small relative to the total number of literals, knowing the literals involved in the unsat core can enable constructing a resolution proof more efficiently.

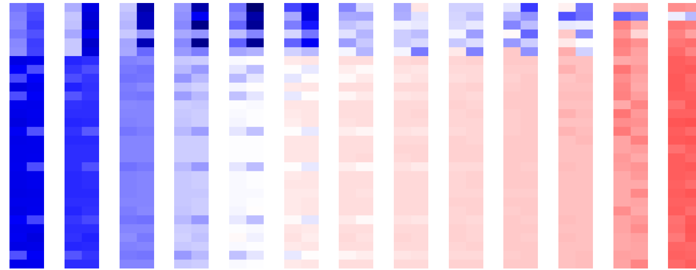
[4] We implemented the version with reinforcement messages described in [49], along with the numerical trick explained in Exercise 359.

We created a new distribution $\mathbf{SRC}(n, u)$ that is similar to $\mathbf{SR}(n)$ except that every unsatisfiable problem contains a small unsat core. Here n is the number of variables as before, and u is an unsat core over $x_1, \ldots, x_k$ $(k < n)$ that can be made into a satisfiable set of clauses u' by negating a single literal. We sample a pair from $\mathbf{SRC}(n, u)$ as follows. We first sample a literal to flip in u to yield u', initialize a problem with u', and then sample clauses (over x_1 to x_n) just as we did for $\mathbf{SR}(n)$ until the problem becomes unsatisfiable. We then negate a literal in the final clause to get a satisfiable problem, and finally we can swap u' for u to yield an unsatisfiable problem that differs from the satisfiable one by negating a single literal in a single clause, and that by construction contains the unsat core u. We created train and test datasets from $\mathbf{SRC}(40, u)$ with u sampled at random for each problem from a collection of three unsat cores ranging from three clauses to nine clauses: the unsat core R from [49], and the two unsat cores resulting from encoding the pigeonhole principles $\mathbf{PP}(2, 1)$ and $\mathbf{PP}(3, 2)$.[5] We trained our architecture on this dataset, and we refer to the trained model as *NeuroUNSAT*.

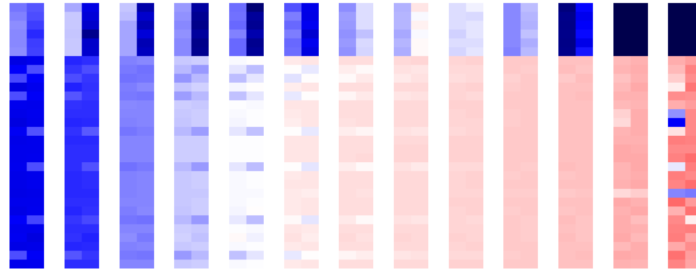
NeuroUNSAT is able to predict satisfiability on the test set with 100% accuracy. Upon inspection, it seems to do so by learning to recognize the unsat cores. Figure 4.10 shows NeuroUNSAT running on a pair of problems from $\mathbf{SRC}(30, \mathbf{PP}(3, 2))$. In both cases, the literals in the first six rows are involved in the unsat core. In Figure 4.10a, NeuroUNSAT inspects the modified core u' of the satisfiable problem but concludes that it does not match the pattern exactly. In Figure 4.10b, NeuroUNSAT finds the unsat core u and votes *unsat* with high confidence (dark blue). As in §4.6, the literals involved in the unsat core can sometimes be decoded from the literal votes $L_*^{(T)}$, but it is more reliable to 2-cluster the higher-dimensional literal embeddings $L^{(T)}$. Figure 4.11 shows the two-dimensional PCA embeddings for $L^{(T)}$ as NeuroUNSAT runs on a much larger problem from $\mathbf{SRC}(100, \mathbf{PP}(3, 2))$. Although only 6% of the literals are involved in the unsat core, they can easily be separated from the other 94%. On the test set, the small number of literals involved in the unsat core end up in their own cluster 98% of the time.

Note that we do not expect NeuroUNSAT to generalize to arbitrary unsat cores:

[5] The pigeonhole principle and the standard SAT encoding are described in [49].



(a) NeuroUNSAT running on a satisfiable problem from $\mathbf{SRC}(30, \mathbf{PP}(3, 2))$.



(b) NeuroUNSAT running on an unsatisfiable problem from $\mathbf{SRC}(30, \mathbf{PP}(3, 2))$.

Figure 4.10: The sequence of literal votes $L_*^{(t)}$ as NeuroUNSAT runs on a pair of problems from $\mathbf{SRC}(30, \mathbf{PP}(3, 2))$. In both cases, the literals in the first six rows are involved in the unsat core. In 4.10a, NeuroUNSAT inspects the modified core u' of the satisfiable problem but concludes that it does not match the pattern. In 4.10b, NeuroUNSAT finds the unsat core u and votes *unsat* with high confidence (dark blue).

as far as we know it is simply memorizing a collection of specific subgraphs, and there is no evidence it has learned a generic procedure to prove *unsat*.

4.9 Related work

The contemporaneous [66] showed that a GNN can be trained to predict the unique solutions of Sudoku puzzles. Although their model was trained with node-level supervision and was specific to Sudoku, we believe their network's success is an instance of the phenomenon we study in this chapter, namely that GNNs can synthesize local

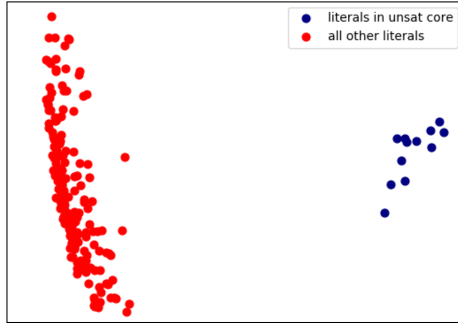


Figure 4.11: PCA projections of the high-dimensional literal embeddings $L^{(T)}$ as NeuroUNSAT detects an unsat core on an unsatisfiable problem from $\mathbf{SRC}(100, \mathbf{PP}(3, 2))$. Blue dots indicate literals that are involved in the unsat core and all remaining literals are red. Although only 6% of the literals are involved in the unsat core, they can easily be separated from the other 94%.

search algorithms for constraint satisfaction problems. The contemporaneous [25] present a neural network architecture that can learn to predict whether one propositional formula entails another by randomly sampling and evaluating candidate assignments. Unlike NeuroSAT, their network does not perform heuristic search and can only work on simple problems for which random guessing is tractable. There have also been several recent papers showing that various neural network architectures can learn heuristics for **NP**-hard combinatorial optimization problems [86, 8, 21]; however, finding low-cost solutions to optimization problems requires less precise reasoning than finding satisfying assignments, and none of these papers present evidence that the networks have learned to search as opposed to simply detecting useful patterns.

Survey propagation (SP) (see §3.3.3) is similar to NeuroSAT in that continuous-valued messages are iteratively passed between literals and the clauses they occur in [12]. The most pertinent difference between SP and NeuroSAT is that in SP, the update function is fixed, whereas in NeuroSAT, it is learned. There are many minor differences as well: the SP messages have a different structure (*e.g.* a clause sends different messages to each of its literals), the SP algorithm involves division and often fails due to division-by-zero, and SP is designed to approximate marginals rather than to predict satisfiability. As we discussed above, SP on its own does not converge to a

satisfying assignment for any of the problems considered in §4.7.3, whereas NeuroSAT solves most of them. Thus NeuroSAT has not just learned to imitate SP but rather has learned something qualitatively different.

4.10 Discussion

Throughout this chapter, our motivation has been scientific: to better understand the extent to which neural networks are capable of precise reasoning. Our work has definitively established that neural networks can learn to perform discrete search on their own without the help of hard-coded search procedures, even after only end-to-end training with minimal supervision. We found this result surprising and think it constitutes an important contribution to the community's evolving understanding of the capabilities of neural networks.

Yet as we stressed above, as an end-to-end SAT solver the trained NeuroSAT system discussed in this chapter is vastly less reliable than the state-of-the-art. We concede that we see no path to beating existing CDCL solvers with such a radically *de novo* approach. In §5 we show how we can leverage the same NeuroSAT architecture to improve the state-of-the-art by deeply integrating it with existing high-performance CDCL solvers.

Chapter 5

Guiding CDCL with Unsat Core Predictions

In §1, we discussed how the declining rate of progress in the SAT community and the substantial, diverse successes of the neural network community have raised the question: *can neural networks somehow be leveraged to improve high-performance SAT solvers*? In this chapter, we answer this question in the affirmative.

The work in this chapter is the result of a collaboration with Nikolaj Bjørner, and benefited from discussions with Percy Liang, David L. Dill, and Marijn J. H. Heule. A paper describing it will be published in the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT-2019), and a preprint has been published to arXiv [76].

5.1 Overview

In this work, we make use of the NeuroSAT architecture presented in §4, but whereas there we used it as an end-to-end solver on toy problems, here we use it to help inform variable branching decisions within high-performance SAT solvers on real problems. Given this goal, the main design decision becomes how to produce data to train the network. There are many possible choices. For example, one could create a dataset of satisfiable problems and their solutions, and train the network to predict the latter

from the former. Or, one could use reinforcement learning, with [80] as a prominent example, and try to learn a policy that minimizes the running time of a hybrid solver on a collection of problems. Our approach is inspired by the *fail first* strategy articulated in [34] that advocates branching on variables most likely to cause conflicts. We approximate this strategy by training NeuroSAT to predict which variables will be involved in unsatisfiable cores. Note that perfect predictions would not always yield a useful variable branching heuristic; for some problems, the smallest core may include every variable, and of course for satisfiable problems, there are no cores at all. Thus, our approach is pragmatic; we rely on NeuroSAT predicting *imperfectly*, and hope that the probability NeuroSAT assigns to a given variable being in a core correlates well with that variable being good to branch on.

The next biggest design decision is how to make use of the predictions inside a SAT solver. Even if we wanted to query NeuroSAT for every variable branching decision, doing so would have severe performance implications, particularly for large problems. A SAT solver makes tens of thousands of assignments every second, whereas even with an on-device GPU, querying NeuroSAT on an industrial-sized problem may take hundreds or even thousands of milliseconds. We settle for complementing—rather than trying to replace—the efficient variable branching heuristics used by existing solvers. All three solvers we extend—MiniSat [24], Glucose [4], and Z3 [22]—use the Exponential Variable State-Independent Decaying Sum (EVSIDS) heuristic (see §3.2), which involves maintaining activity scores for every variable and branching on the free variable with the highest score. The only change we make is that we periodically query NeuroSAT on the *entire* problem (i.e. not conditioning on the current trail), and set all variable activity scores at once in proportion to how likely NeuroSAT thinks the variable is to be involved in an unsat core. We refer to our integration strategy as *periodic refocusing*. We remark that the base heuristics are already strong, and they may only need an occasional, globally-informed reprioritization to yield substantial improvements.

We summarize our pipeline:

1. Generate many unsatisfiable problems by decimating existing problems.

2. For each such problem, generate a DRAT proof (see §3.1), and extract the variables that appear in the unsat core.

3. Train NeuroSAT (henceforth NeuroCore) to map unsatisfiable problems to the variables in the core.

4. Instrument state-of-the-art solvers (MiniSat, Glucose, Z3) to query NeuroCore periodically (using the original and the learnt clauses), and to reset their variable activity scores according to NeuroCore's predictions.

As a result of these modifications, the MiniSat solver solves 10% more problems on SATCOMP-2018 within the standard 5,000 second timeout. The modified Glucose 4.1 solves 11% more problems than the original, while the modified Z3 solves 6% more. The gains are even greater when the training is specialized for a specific distribution of problems; our training set included (easy) subproblems of a collection of hard scheduling problems, and on that collection of hard problems the modified Glucose solves 20% more problems than the original does within a one hour timeout. Our results demonstrate that NeuroSAT (and in particular, NeuroCore) can be leveraged to improve high-performance SAT solvers on real problems.

5.2 Data generation

As discussed in §5.1, we want to train our neural network architecture to predict which variables will be involved in unsat cores. Unfortunately, there are only a few thousand unsatisfiable problems across all SATCOMP competitions, and a network trained on such few examples would be unlikely to generalize well to unseen problems. We overcome this limitation and generate a dataset containing over 150,000 different problems with labeled cores by considering unsatisfiable *subproblems* of existing problems.

Specifically, we generate training data as follows. We use the distributed execution framework ray [62] to coordinate one driver and hundreds of workers distributed over several machines. The driver maintains a queue of (sub)problems, and begins by

enqueuing all problems from SATCOMP (through 2017 only) as well as a few hundred hard scheduling problems. It might help to initialize with even more problems, but we did not find it necessary to do so. Whenever a worker becomes free, the driver dequeues a problem and passes it to the worker. The worker tries to solve it using Z3 with a fixed timeout (we used 60 seconds). If Z3 returns *sat*, it does nothing, but if Z3 returns *unsat*, it passes the generated DRAT proof to DRAT-trim [88] to determine which of the original clauses were used in the proof. It then computes the variables in the core by traversing the clauses in the core, and finally generates a single datapoint in a format suitable for NeuroSAT. If Z3 returns *unknown*, the worker uses a relatively expensive, hand-engineered variable branching heuristic—specifically, Z3's implementation of the March heuristic (see §3.3.1)—and returns the two subproblems to the driver to be added to the queue.

This process generates one datapoint roughly every 60 seconds per worker. Some of the original problems are very difficult, and so the process may not terminate in a reasonable amount of time; thus we stopped it once we had generated 150,000 datapoints.

As a minor extension, each worker generates even more data for each subproblem found to be *unsat* by randomly removing a small percentage of the clauses and re-solving; if it is still *unsat*, it computes the core, generates a datapoint, and repeats, up to a maximum of 10 times.

Note that our data generation process is not guaranteed to generate diverse cores. To the extent that March is successful in selecting variables to branch on that are in the core, the cores of the two subproblems will be different; if it fails to do this, then the cores of the two subproblems may be the same (though the non-core clauses will still be different). We remark that there are many other ways one might augment the dataset, for example by including additional problems from synthetic distributions, or by directly perturbing the signs of the literals in the existing problems. However, our simple approach proved sufficient.

We stress that predicting the (binary) presence of variables in the core is simplistic. As mentioned in §5.1, for some problems, the smallest core may include every variable, in which case the datapoint for that problem would contain no information.

Even if only a small fraction of variables are in the core, it may still be that only a small fraction of those core variable would make good branching decisions. A more sophisticated approach would analyze the full DRAT proof and calculate a more nuanced score for each variable that reflects its importance in the proof. However, as we will see in §5.5, our simplistic approach of predicting the variables in the core proved sufficient to achieve compelling results.

5.3 Neural Network Architecture

The neural network architecture we use here is a simplified version of the one presented in §4.3. For the convenience of the reader, we now describe our simplified version in detail. Readers already comfortable with NeuroSAT may choose to skip to the summary of key differences at the end of this section.

Our neural network represents a SAT problem as its clause-literal adjacency matrix $\mathcal{G}$, as in §4.3. The neural network itself is made up of three standard multilayer perceptrons:

$$\begin{aligned}
\mathbf{C}_{\text{update}} &: \mathbb{R}^{2d} \rightarrow \mathbb{R}^{d} \\
\mathbf{L}_{\text{update}} &: \mathbb{R}^{3d} \rightarrow \mathbb{R}^{d} \\
\mathbf{V}_{\text{proj}} &: \mathbb{R}^{2d} \rightarrow \mathbb{R}
\end{aligned}$$

The network computes forward as follows. First, it initializes two matrices $C \in \mathbb{R}^{m \times d}$ and $L \in \mathbb{R}^{2n \times d}$ to all ones. Each row of C corresponds to a clause, while each row of L corresponds to a literal:

$$C = \begin{bmatrix} — & c_1 & — \\ & \vdots & \\ — & c_m & — \end{bmatrix}, \qquad L = \begin{bmatrix} — & x_1 & — \\ & \vdots & \\ — & x_n & — \\ — & \overline{x_1} & — \\ & \vdots & \\ — & \overline{x_n} & — \end{bmatrix}$$

We refer to the row corresponding to a clause c or a literal ℓ as the *embedding* of

that clause or literal. Define the operation Flip to swap the first half of the rows of a matrix with the second half, so that in Flip(L), each literal's row is swapped with its negation's:

$$\text{Flip}(L) = \begin{bmatrix} — & \overline{x_1} & — \\ & \vdots & \\ — & \overline{x_n} & — \\ — & x_1 & — \\ & \vdots & \\ — & x_n & — \end{bmatrix}$$

Next, the network applies the following updates T times (we used $T = 4$):

$$C \leftarrow \mathbf{C}_{\text{update}}\left(C, \mathcal{G}L\right) \tag{5.1}$$

$$L \leftarrow \mathbf{L}_{\text{update}}\left(L, \mathcal{G}^\top C, \text{Flip}(L)\right) \tag{5.2}$$

Note that passing Flip(L) as input to the update for L manifests the implicit flip edge in the problem graph. After each such update, the network also normalizes each column of C and L to have mean zero and variance one.

Define the operation Flop to concatenate the first half of the rows of a matrix with the second half along the second axis, so that in Flop(L), the two vectors corresponding to the same variable are concatenated:

$$\text{Flop}(L) = \begin{bmatrix} — & x_1 & — & — & \overline{x_1} & — \\ & & & \vdots & & \\ — & x_n & — & — & \overline{x_n} & — \end{bmatrix} \in \mathbb{R}^{n \times 2d}$$

After T iterations, the network flops L to produce the matrix $V \in \mathbb{R}^{n \times 2d}$. Note that flopping breaks negation invariance. Although we could easily maintain negation invariance in any number of ways, we do not do so here because in real-world problems (as opposed to random problems), the sign of a literal may actually contain useful

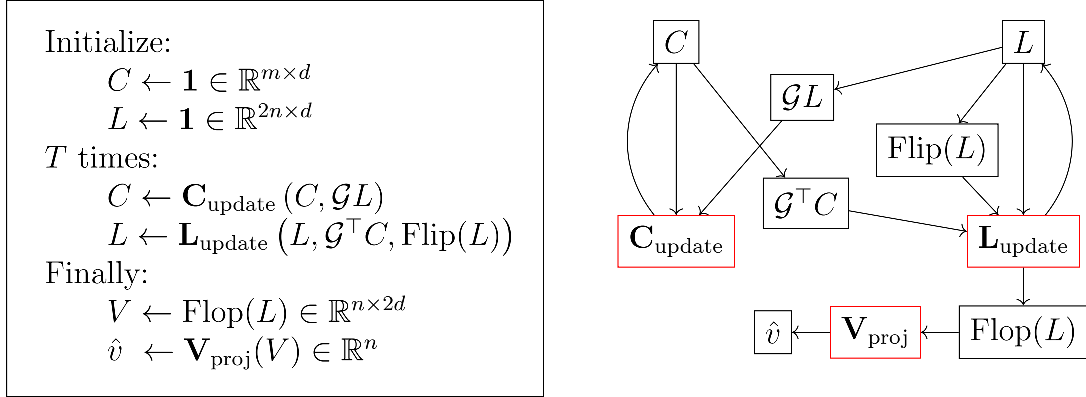


Figure 5.1: An overview of the NeuroCore architecture

(heuristic) information.

Finally, the network projects V into an n-dimensional vector $\hat{v}$ using the third MLP, $\mathbf{V}_{\text{proj}}$:

$$\hat{v} \leftarrow \mathbf{V}_{\text{proj}}(V) \in \mathbb{R}^n$$

The vector $\hat{v}$ is the output of NeuroCore, and consists of a numerical score for each variable, which can be passed to the softmax function to define a probability distribution $\hat{p}$ over the variables. During training, we turn each labeled bitmask over variables into a probability distribution p^* by assigning uniform probability to each variable in the core and zero probability to the others. We optimize the three MLPs all at once to minimize the Kullback-Leibler divergence [53]:

$$\mathbf{D}_{KL}(p^* \parallel \hat{p}) = \sum_{i=1}^{n} p_i^* \log\left(p_i^* / \hat{p}_i\right)$$

Figure 5.1 summarizes the architecture.

Comparison to the original NeuroSAT. While the original NeuroSAT architecture was designed to solve small problems end-to-end, the version in this chapter is designed to provide cheap, heuristic guidance on (potentially) large problems. Accordingly, our network differs from the original in a few key ways. First, ours only runs for 4 iterations at both train and test time, whereas the original was trained with

26 iterations and ran for upwards of a thousand iterations at test time. Second, ours simply initializes all elements of all embeddings to one, whereas the original learned two initialization vectors, one for the literal embeddings and one for the clause embeddings. Third, our update networks are simple feed-forward networks, whereas the original used layer-norm LSTMS [5, 40], as well as separate "message" networks to transform the neighbor embeddings before aggregating them. Finally, as discussed above, ours is trained with supervision at every variable and outputs a vector $\hat{v} \in \mathbb{R}^n$, whereas the original is trained with only a single bit of supervision and accordingly only outputs a single scalar.

Training NeuroCore. As we discussed in §5.1, our goal is not to learn a perfect core predictor, but rather only to learn a coarse heuristic that broadly assigns higher score to more important variables. Thus, fine-tuning the network is relatively unimportant, and we only ever trained with a single set of hyperparameters. We used the Adam optimizer [47] with a constant learning rate of 10^{-4}, and trained asynchronously with 20 GPUs for under an hour, using distributed TensorFlow [2].

5.4 Hybrid Solving: CDCL and NeuroCore

As discussed in §5.1, it is too expensive to query NeuroCore for every variable branching decision, and so we settle for querying periodically on the entire problem (*i.e.* not conditioning on the trail) and replacing the variable activity scores with NeuroCore's prediction. We now describe this process in detail.

When we query NeuroCore, we build the sparse clause-literal adjacency matrix $\mathcal{G}$ (see §4.3) as follows. First, we collect all variables in the problem that have not been eliminated during simplification and that do not appear in any unit clauses (*i.e.* clauses of length one). These are the only variables that we tell NeuroCore about. Second, we collect all the clauses that we plan to tell NeuroCore about. We would like to tell NeuroCore about all the (non-unit) clauses, both original and learnt, but the size of the problem can get extremely large as the solver accumulates learnt clauses. At some point the problem would no longer fit in GPU memory, and it might be

undesirably expensive even before that point. After collecting the original clauses, we traverse the learned clauses in ascending size order, collecting clauses until the number of literals plus the number of clauses plus the number of cells (*i.e.* literal occurrences in clauses) exceeds a fixed cutoff (we used 10 million). If a problem is so big that the original clauses already exceed this cutoff, then for simplicity we do not query NeuroCore at all, although we could have still queried it on random subsets of the clauses. Finally, we traverse the chosen clauses to construct $\mathcal{G}$. Note that because of the learned clauses (unit and otherwise) and the eliminated variables, NeuroCore is shown a substantially different graph on each query even though we do not condition on the trail.

NeuroCore then returns a vector $\hat{v} \in \mathbb{R}^n$, where a higher score for a variable indicates that NeuroCore thinks the corresponding variable is more likely to be in the core. We turn $\hat{v}$ into a probability distribution by dividing it by a scalar temperature parameter τ (we used 0.25) and taking the softmax, and then we scale the resulting vector by the number of variables in the problem, and additionally by a fixed constant κ (we used 10^4). Finally, we replace all the EVSIDS scores at once:[1]

$$\forall i, \mathbf{EVSIDS}(x_i, t) \leftarrow \text{Softmax}(\hat{v}/\tau)_i n\kappa$$

Note that the decay factor ρ is often rather small (MiniSat uses $\rho = 0.95$), and to a first approximation solvers average ten thousand conflicts per second, so these scores decay to 0 in only a fraction of a second. However, such an intervention can still have a powerful effect by refocusing EVSIDS on a more important part of the search space. We refer to our integration strategy as *periodic refocusing* to stress that we are only refocusing EVSIDS rather than trying to replace it. Our hybrid solver based on MiniSat only queries NeuroCore once every 100 seconds.

[1] In MiniSat, this involves setting the activity vector to these values, resetting the variable increment to 1.0, and rebuilding the order-heap.

5.5 Solver Experiments

We evaluate the hybrid solver *neuro-minisat* (which operates as described in §5.4) and the original MiniSat solver *minisat* on the 400 problems from the main track of SATCOMP-2018, with the same 5,000 second timeout used in the competition. For each solver, we solved the 400 problems in 400 different processes in parallel, spread out over 8 identical 64-core machines, with no other compute-intensive processes running on any of the machines. In addition, the hybrid solver also had network access to 5 machines each with 4 GPUs, with the 20 GPUs split evenly and randomly across the 400 processes. We calculate the running time of a solver by adding together its process time with the sum of the wall-clock times of each of the TensorFlow queries it requests on the GPU servers. We ignore the network transmission times since in practice one would often use an on-device hardware accelerator.

Note that although we did not train NeuroCore on any (sub)problems from the SATCOMP-2018 benchmarks, we did use a small number of runs of *neuro-minisat* on problems from SATCOMP-2018 while performing extremely coarse tuning of the parameter κ, which a priori could have reasonably been set to any double-precision floating point value.[2]

Results. The main result, alluded to in §5.1, is that *neuro-minisat* solves 205 problems within the 5,000 second timeout whereas *minisat* only solves 187. This corresponds to an increase of 10%. Most of the improvement comes from solving more satisfiable problems: *neuro-minisat* solve 125 satisfiable problems compared to *minisat*'s 109, which is a 15% increase. On the other hand, *neuro-minisat* only solved 3% more unsatisfiable problems (80 vs 78). Figure 5.2 shows a cactus plot of the two solvers, which shows that *neuro-minisat* gains a substantial lead within the first few minutes and maintains the lead until the end. Figure 5.3 shows a scatter plot of the same data, which shows there are quite a few problems that *neuro-minisat* solves within a few minutes that *minisat* times out on. It also shows that there are very few problems on which *neuro-minisat* is substantially worse than *minisat*.

[2]In hindsight we regret not using alternate problems for this, but we strongly suspect that we would have found a similar ballpark by only tuning on problems from other sources.

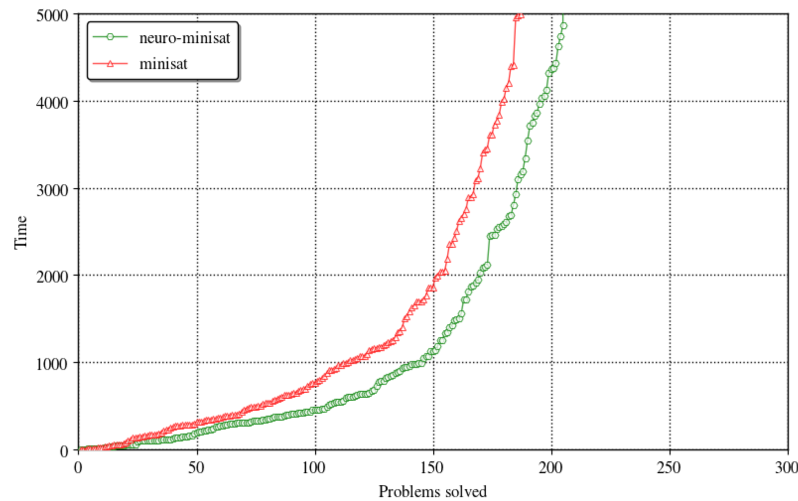


Figure 5.2: Cactus plot comparing NeuroCore-assisted MiniSat (*neuro-minisat*) with the original (*minisat*) on SATCOMP-2018. It shows that *neuro-minisat* gains a substantial lead within the first few minutes and maintains the lead until the end.

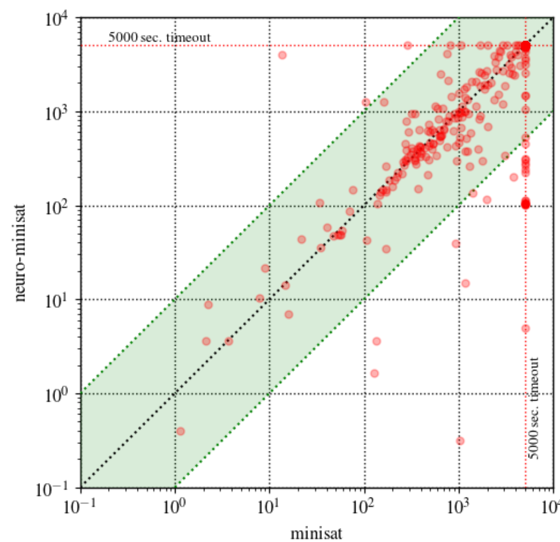


Figure 5.3: Scatter plot comparing NeuroCore-assisted MiniSat (*neuro-minisat*) with the original (*minisat*) on SATCOMP-2018. It shows that there are quite a few problems that *neuro-minisat* solves within a few minutes that *minisat* times out on, and that there are very few problems on which *neuro-minisat* is substantially worse than *minisat*.

Ablations. The results show that our hybrid approach is effective, but do not tell us much about why it is effective. We do not have a satisfying answer to this question yet. As a consolation, we report a few ablations that shed some light on why it may work. For these ablations, we periodically refocus every 10 seconds instead of every 100 seconds to make the effect of the quality of the scores more pronounced. When we query NeuroCore every 10 seconds instead of every 100 seconds, *neuro-minisat* still solves 205 problems within the timeout.

First, we investigated whether using EVSIDS between queries was necessary, or whether NeuroCore's predictions were sufficient on their own. Simply increasing κ from 10^4 to 10^{40} (which only prevents EVSIDS from taking over for approximately 200ms following each query) already had an substantial negative effect: it solved less than a third of the problems that *minisat* solved. Thus NeuroCore is not a replacement to EVSIDS but only a complement to it. Second, we investigated whether NeuroCore's predictions even mattered at all, or if the solver would benefit equally from just periodically setting the EVSIDS scores to random values. When we would otherwise call NeuroCore, we substituted $\hat{v}$ with scores sampled uniformly between $(-1, 1)$, and transformed them to EVSIDS scores using the original τ and κ values. This change had a even more harmful effect: the resulting solver only solved a tiny handful of problems out of 400. Third, we considered that perhaps NeuroCore's predictions are mostly irrelevant, and that the important part is that they are roughly the same at every query. We tried the same experiment with random scores but with the scores sampled uniformly once at the beginning of search and reused at every query. This did a little better than when the scores changed each time, but not by much. These experiments do not rule out the possibility that there is a simple, hardcodeable heuristic that could do just as well as NeuroCore, but they do suggest that there is substantial signal in NeuroCore's predictions.

Glucose. As a follow-up experiment and sanity check, we made the same modifications to Glucose 4.1 and evaluated in the same way on SATCOMP-2018. To provide further assurance that our findings are robust, we altered the NeuroCore schedule, changing from fixed pauses (100 seconds) to exponential backoff (5 seconds at first

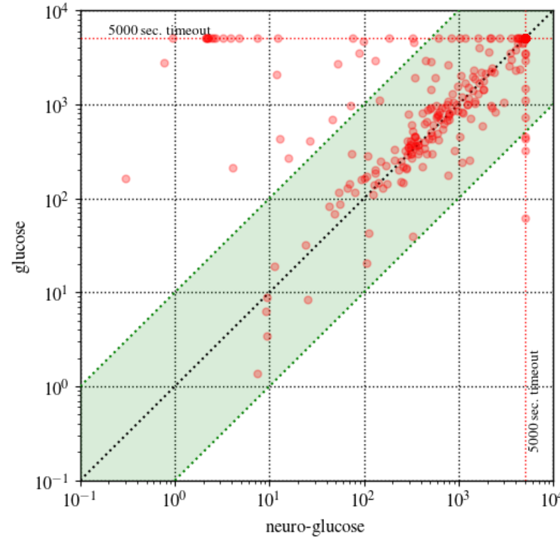


Figure 5.4: Scatter plot comparing NeuroCore-assisted Glucose (*neuro-glucose*) with the original (*glucose*) on SATCOMP-2018. It shows that there are quite a few problems that *neuro-glucose* solves within a few seconds that *glucose* times out on, and that there are very few problems on which *neuro-glucose* is substantially worse than *glucose*.

with multiplier γ=1.2). The results of the experiment are very similar to the results from the MiniSat experiment described above. The number of problems solved within the timeout jumps 11% from 186 to 206. Figure 5.4 show the scatter plot comparing *neuro-glucose* to *glucose*. This comparison is even more favorable to the NeuroCore-assisted solver than Figure 5.3, as it shows that there are many problems *neuro-glucose* solves within seconds that *glucose* times out on. We believe this is due to the exponential backoff query schedule, which makes more frequent NeuroCore queries in the beginning of search. The cactus plot for the Glucose experiment is almost identical to the one in Figure 5.2 and so is not shown.

Z3. Lastly, we made the same modifications to Z3, except we once again altered the NeuroCore schedule, this time from exponential backoff in terms of user time to geometric backoff in terms of the number of conflicts. Specifically, we first query NeuroCore after 50,000 conflicts, and then each time wait 50,000 more conflicts than

the previous time before querying NeuroCore again. The modified Z3 solves 170 problems within the timeout, up from 161 problems, which is a 6% increase.

Note that for the Z3 experiment, to save on computational costs, we evaluated both solvers simultaneously instead of sequentially. To ensure fairness, we ordered the task queue by problem rather than by solver. The lower absolute scores compared to MiniSat and Glucose are partly the result of the increased contention.

A more favorable regime. It is worth remarking that SATCOMP-2018 is an extremely unfavorable regime for machine learning methods. As discussed in §3.4, all problems are arbitrarily out of distribution. The 2018 benchmarks include problems arising from a dizzyingly diverse set of domains: proving theorems about bit-vectors, reversing Cellular Automata, verifying floating-point computations, finding efficient polynomial multiplication circuits, mining Bitcoins, allocating time slots to students with preferences, and finding Hamiltonian cycles as part of a puzzle game, among many others [38].

In practice, one often wants to solve many problems arising from a common source over an extended period of time, in which case it could be worth training a neural network specifically for the problem distribution in question. We approximate this regime by evaluating the same trained network discussed above on the set of 303 (non-public) hard scheduling problems that were included in the data generation process along with SATCOMP-2013 to SATCOMP-2017. Note that although NeuroCore may have seen unsat cores of *subproblems* of these problems during training, most of the problems are so hard that many variables need to be set before Z3 can solve them in under a minute. Also, at deployment time we are passing the learned clauses to NeuroCore as well, which may vastly outnumber the original clauses. Thus, although it clearly cannot hurt to train on subproblems of the test problems, NeuroCore is still being queried on problems that are substantially different than those it saw during training.

For this experiment, we compared *glucose* to *neuro-glucose* on the 303 scheduling problems, using a one hour timeout and the same setting of κ as for the SATCOMP-2018 experiment above. As one might expect, the results are even better than in the

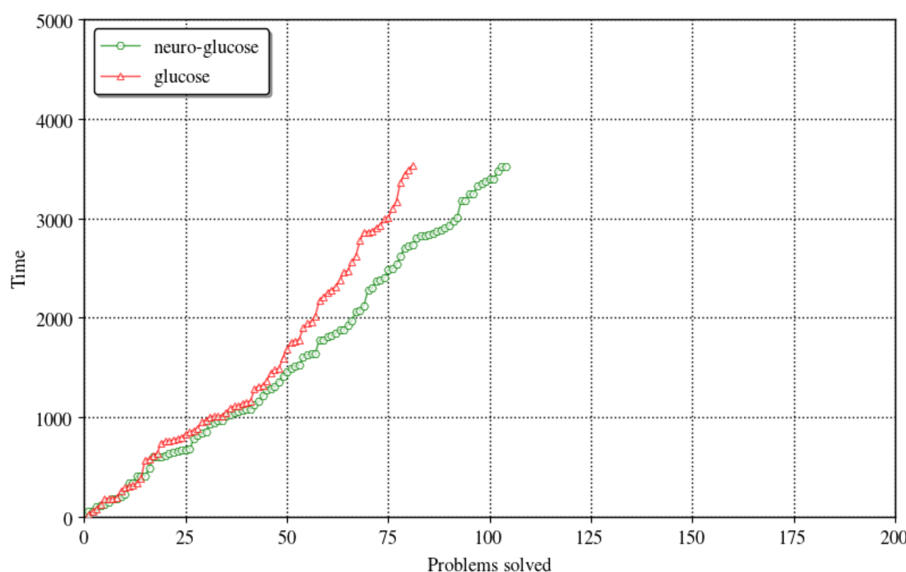


Figure 5.5: Cactus plot comparing NeuroCore-assisted Glucose (*neuro-glucose*) with the original (*glucose*) on a benchmark of 303 (non-public) challenging scheduling problems, for which some subproblems were included in the training set. In contrast to Figure 5.2, which showed that on SATCOMP-2018 *neuro-minisat* got off to an early lead and maintained it throughout, here we see that the solvers are roughly tied for the first thirty minutes, at which point *neuro-glucose* begins to pull away, and continues to add to its lead until the one hour timeout.

SATCOMP regime. The hybrid *neuro-glucose* solver solves 20% more problems than *glucose* within the timeout. Figure 5.5 shows a cactus plot comparing the two solvers. In contrast to Figure 5.2, which showed that on SATCOMP-2018 *neuro-minisat* got off to an early lead and maintained it throughout, here we see that the solvers are roughly tied for the first thirty minutes, at which point *neuro-glucose* begins to pull away, and continues to add to its lead until the one hour timeout.

5.6 Related Work

The use of machine learning techniques for variable branching decisions in SAT is to our knowledge relatively unexplored. However, there have been many attempts over the years to apply statistical learning to other aspects of the SAT problem, including

to restart strategies [32], parameter tuning [81], and solver selection [92]. In contrast to our work, none of these approaches use neural networks, and instead make use of both generic graph features and features extracted from the runs of SAT solvers. In most cases, the statistical models have a few dozen trainable parameters at most, are called once per problem, and output only a few bits either to set a small number of parameters or to indicate a choice among curated high-level strategies. Such lightweight approaches can be very effective, though we believe that our approach of parameter-rich neural networks, semantically meaningful training objectives (*e.g.* predicting cores), and deep integration with CDCL solvers ultimately has much greater potential impact on the field.

As far we know, our approach of deeply integrating global analysis with the cheap, adaptive heuristics of CDCL solvers—which we have called *periodic refocusing*—is novel, and can be viewed as an alternative paradigm to cube and conquer (see §3.3.2) for trading off between cheap and expensive branching decisions. In our case the global analysis is performed by a neural network, but the paradigm is compatible with any kind of global analysis. Indeed, we speculate that some of the benefits of our approach could be realized without neural networks, by *e.g.* using the scores computed by March to periodically refocus the CDCL solver. However, March is a clever and complicated heuristic that is difficult to implement while potentially being very far from optimal; indeed, simplifying and automating this kind of heuristic was one of the original motivations for our work with NeuroSAT.

5.7 The Vast Design Space

There is a vast design space for how to train NeuroSAT and how to use it to guide SAT solvers. This work has focused on only one tiny point in that design space. We now briefly discuss other approaches we considered.

For our first experiment predicting unsatisfiable cores, we trained NeuroSAT on a synthetic graph coloring distribution that happened to have tiny cores. NeuroSAT was able to predict these cores so accurately that we could get almost arbitrarily big speedups by only giving Z3 the tiny fraction of clauses that NeuroSAT thought most

likely to be in the core (and doubling the number of clauses given as necessary until they included the core). Unfortunately, it is much harder to learn a general-purpose core predictor than one on a particular synthetic distribution for which instances may all have similar cores. Real problems also rarely have such tiny cores, so even a perfect core predictor might not be such a silver bullet. However, we do think that core predictions may nonetheless be useful in guiding clause pruning. Our first efforts here were hampered by the fact that we were rarely able to fit the majority of conflict clauses in GPU memory given our relatively large network architecture (*i.e.* d=80). Simply retraining with a smaller d would address this problem, and we plan to pursue this in the future.

We also experimented with training NeuroSAT to imitate the decisions of the March cubing heuristic. Based on preliminary experiments in a challenging scheduling domain, we found that NeuroSAT trained only to imitate March may actually produce better cubes than March itself, though it remains to be seen if this result holds up to greater scrutiny. In contrast, using the unsat core predictions to make cubing decisions seemed to perform consistently worse than the March baseline, though still respectably well. We also tried using NeuroSAT's March predictions to refocus the EVSIDS scores, and found this to perform worse than its unsat core predictions. However, we note that the March predictions were much peakier than the unsat core predictions, and so the inferior performance may have been the result of using such a low value for the temperature parameter τ.

We also briefly experimented with predicting models directly. Specifically, we used existing solvers to find models of satisfiable problems, and then trained NeuroSAT to predict the phases of each of the variables individually. Then, we instrumented MiniSat to choose the phase of each decision variable in proportion to NeuroSAT's prediction. Note that this approach is very simplistic, since a single problem may have many models; for example, if it suffices to assign only $\epsilon\%$ of the variables to satisfy all the clauses, then $(1-\epsilon)\%$ of the phases will be arbitrary. Nonetheless, we still found some preliminary evidence that even this simplistic approach may help in some cases, particularly on unsatisfiable problems, though the preliminary results were insufficiently promising for us to pursue further at this stage. An alternative

approach to learning a phase heuristic may be to only predict the phases of variables for which one literal has been proven to be entailed. It is trivial to generate a huge amount of data for this task, since every learned conflict clause provides one datapoint.

Lastly, inspired by the success of [80], we experimented with various forms of Monte Carlo tree search and reinforcement learning, though so far the only competitive heuristic we have been learn de novo is a cubing strategy for uniform random problems. There are two main challenges for learning variable branching heuristics by exploration alone: problems may have a huge number of variables, and it may take substantial time to solve the (sub)problems in order to get feedback about a given branching decision. The former challenge can be mitigated by beginning with imitation learning (e.g. by imitating March). We tried to mitigate the latter by pretraining a value function based on data collected from solving a collection of benchmarks, and then using the value function estimates to make cheap importance sampling estimates of the size of the search tree under different policies as described in [48]. We found that even in the supervised context, training the value function is difficult; without taking logs it is numerically difficult, and with taking logs, one can get very low loss while ignoring the relatively few hard subproblems towards the roots that make the most difference. Ultimately, we think that the satisfiability problem offers such great opportunities for post facto analysis and principled credit assignment that there is simply no need to resort to generic reinforcement learning methods.

We have only scratched the surface of this design space. We hope that our promising initial results with NeuroCore inspire others to try leveraging NeuroSAT in other, creative ways.

Chapter 6

Conclusion

We think that one of the most important takeaways from the success of NeuroSAT (§4) is that it is reasonable to expect that an end-to-end neural network approach can be made to perform well even in domains that may require some form of search. This insight is especially promising for domains where a precise representation of a search problem is not a priori available, for which it would not be feasible to supervise the network to reason directly nor to implement a hard-coded search procedure.

On the other hand, in domains where it *is* relatively straightforward to apply hard-coded search procedures, we think end-to-end neural approaches are unlikely to rival state-of-the-art search methods except in certain niche cases. However, the success of NeuroCore (presented in §5) tells us that neural networks can nonetheless provide substantial improvements on top of existing, highly engineered search procedures. Although many details of our approach in §5 were specific to SAT (*e.g.* replacing the EVSIDS scores), at a high level our approach is more broadly applicable to other problems in discrete search, such as satisfiability modulo theories (SMT), program synthesis, and higher-order theorem proving. The keys steps of our methodology are as follows.

Step 1 (design network architecture). The first step is to design a custom neural network architecture for the desired formalism that builds in useful invariances. In the NeuroSAT case, we built in permutation and negation invariance, but richer formalisms may have more sophisticated invariances that may be worth

building in. For example, in higher-order logic, the expression $(\lambda x.\lambda y.x \wedge y)$ is trivially (α-)equivalent to $(\lambda w.\lambda z.w \wedge z)$, and yet a naïve embedding of the syntax tree would not know this a priori. However, as discussed in §4.3, not every invariance is worth building into the model. For example, in higher-order logic, the expression $(\lambda x.\neg x)((\lambda y.\lambda z.z)\,0\,1)$ is β-equivalent to $\neg 1$, and yet we see no simple way to build β-equivalence into a neural network architecture. We stress that designing good architectures for different formalisms may require cleverness and experimentation.

Step 2 (collect problems). The second step is to collect diverse problems. As in §5, there may not be enough problems easily accessible to train a neural network, and so we advocate extensive problem transformations to augment the dataset. One approach is to mine subproblems of the original problems by fixing the values of subsets of the variables. Another approach is to transform problems syntactically, for example by randomly negating variable occurrences, by discarding clauses, or by combining components of different problems. It may also be desirable to write synthetic problem generators, or even to train neural networks to generate problems that are similar to the original problems.

Step 3 (perform post facto analysis). The third step is to run state-of-the-art solvers on the collected problems, outputting a history of the search in the process. This history can then be analyzed post facto to determine which decisions were good and which were bad. In some cases, the history can be analyzed to determine even better decisions than what the solver even considered. This analysis can then be used to create a labeled dataset mapping problems to scores for each decision being considered.

Step 4 (deploy with periodic refocusing). Except in rare cases, we consider it unlikely that the trained network will be fast and reliable enough to replace the underlying solver. Thus, the final step is to integrate the trained network with an existing solver through some form of periodic refocusing, as we did in §5.4. The exact mechanism will depend on the formalism and the solver, but many solvers have cheap built-in heuristics that rely on numerical scores that

are dynamically adjusted throughout the solver run, as in EVSIDS. For such solvers, periodic refocusing can proceed analogously to §5.4. For other solvers, the appropriate way to use the output of the neural network to refocus the search may require cleverness and experimentation.

Although we stress that each new domain will likely require new insights, we are cautiously optimistic that similar approaches will lead to substantial gains in many other areas of discrete search in addition to SAT.

Lastly, we think the results of §4 help illuminate important limitations of GNNs. As we discuss in §4.7.2, NeuroSAT seems to perform an analogue of simulated annealing, and has trouble escaping from local optima. We think this weakness may plague GNNs in other contexts too, and architectural extensions that address this weakness (perhaps by providing a source of randomness) may prove useful in other domains. A related weakness that we highlight in §4.8 is that NeuroSAT seems to only perform *incomplete* search—unless we plant simple contradictions during training, it never becomes confident in unsatisfiability. This weakness is not surprising, since both exhaustive backtracking search and proof systems such as resolution seem to require more sophisticated data structures, *e.g.* stacks for the former and maps for the latter. We experimented with extending NeuroSAT with a single global neural stack [30], but found that, at least in our $\mathbf{SR}(\mathbf{U}(10, 40))$ training regime, it learned to ignore the stack completely. It is an open question if there exists an architectural extension that lets NeuroSAT become confident in unsatisfiability.

At a higher level, studying GNNs in the SAT domain helps clarify what may be their most fundamental weakness: they are essentially propositional automata. As such, we doubt that they would be capable of competent reasoning in more abstract logics for which neither proofs nor models can be represented as bitmasks on the syntax of the original problem. Although GNNs are unlikely to be sufficient, we do think the methodology we adopted in §4 might prove fruitful for more abstract logics. Can we design a neural network architecture that learns to prove (or find models for) higher-order propositions? We think such an architecture may prove useful in domains that require abstract reasoning, such as natural language understanding.

Bibliography

[1] RobustFill: Neural program learning under noisy I/O.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation OSDI 16*, pages 265–283, 2016.

[3] Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, et al. Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*, 2018.

[4] Gilles Audemard and Laurent Simon. GLUCOSE: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[5] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[6] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[7] Proceedings of SAT competition 2017; solver and benchmark descriptions. 2017.

[8] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[9] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *Python for Scientific Computing Conference*, 2010.

[10] Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 405–422. Springer, 2015.

[11] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[12] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.

[13] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. Reading Wikipedia to answer open-domain questions. *arXiv preprint arXiv:1704.00051*, 2017.

[14] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[15] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[16] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

[17] José Coelho and Mario Vanhoucke. Multi-mode resource-constrained project scheduling using RCPSP and SAT solvers. *European Journal of Operational Research*, 213(1):73–82, 2011.

[18] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[19] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[20] George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989.

[21] Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

[22] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[23] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at Microsoft. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8604–8608. IEEE, 2013.

[24] Niklas Een. MiniSat: A SAT solver with conflict-clause minimization. In *Proc. SAT-05: 8th Int. Conf. on Theory and Applications of Satisfiability Testing*, pages 502–518, 2005.

[25] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? *arXiv preprint arXiv:1802.08535*, 2018.

[26] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 340–354. Springer, 2007.

[27] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

[28] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[29] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *arXiv preprint arXiv:1410.5401*, 2014.

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

[31] Peter Großmann, Steffen Hölldobler, Norbert Manthey, Karl Nachtigall, Jens Opitz, and Peter Steinke. Solving periodic event scheduling problems with SAT. In *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pages 166–175. Springer, 2012.

[32] Shai Haim and Toby Walsh. Restart strategy selection using machine learning techniques. *CoRR*, abs/0907.5032, 2009.

[33] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. Deep Speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[34] Robert M Haralick and Gordon L Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[35] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[36] Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. *Handbook of satisfiability*, 185:155–184, 2009.

[37] Marijn JH Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer, 2011.

[38] Proceedings of SAT competition 2018; solver and benchmark descriptions. 2018.

[39] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

[40] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[41] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[42] John Homer and Xinming Ou. SAT-solving approaches to context-aware enterprise network security management. *IEEE Journal on selected areas in communications*, 27(3):315–322, 2009.

[43] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[44] Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 804–813, 2017.

[45] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Een, François Chollet, and Josef Urban. DeepMath: Deep sequence models for premise selection. In *Advances in Neural Information Processing Systems*, pages 2235–2243, 2016.

[46] Cezary Kaliszyk, François Chollet, and Christian Szegedy. HolStep: A machine learning dataset for higher-order logic theorem proving. *arXiv preprint arXiv:1703.00426*, 2017.

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

[48] Donald E Knuth. Estimating the efficiency of backtrack programs. *Mathematics of computation*, 29(129):122–136, 1975.

[49] Donald E Knuth. The art of computer programming, volume 4, fascicle 6: Satisfiability, 2015.

[50] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[51] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[52] Mirco Kuhlmann, Lars Hamann, and Martin Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 290–306. Springer, 2011.

[53] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[54] Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps. *The International Journal of Robotics Research*, 34(4-5):705–724, 2015.

[55] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.

[56] Harry R Lewis. Computers and intractability. a guide to the theory of NP-completeness, 1983.

[57] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*, 2016.

[58] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[59] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017.

[60] Sid Mijnders, Boris de Wilde, and Marijn Heule. Symbiosis of search and heuristics for random 3-SAT. *CoRR*, abs/1402.4455, 2010.

[61] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[62] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation OSDI 18)*, pages 561–577, 2018.

[63] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[64] Mark Newman. *Networks: an introduction*. Oxford university press, 2010.

[65] Chanseok Oh. *Improving SAT solvers by exploiting empirical characteristics of CDCL*. PhD thesis, New York University, 2016.

[66] Rasmus Berg Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks for complex relational reasoning. *arXiv preprint arXiv:1711.08028*, 2017.

[67] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[68] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2012.

[69] Judea Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science, 1982.

[70] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier, 2014.

[71] B. Perozzi, R. Al-Rfou, and S. Skiena. DeepWalk: Online learning of social representations. In *International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 701–710, 2014.

[72] Gadi Pinkas. Energy minimization and the satisfiability of propositional logic. In *Connectionist Models*, pages 23–31. Elsevier, 1991.

[73] Jussi Rintanen. Madagascar: Scalable planning with SAT. *Proceedings of the 8th International Planning Competition (IPC-2014)*, 21, 2014.

[74] The International SAT Competitions web page. `https://www.satcompetition.org/`. Accessed: 2019-04-01.

[75] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

[76] Daniel Selsam and Nikolaj Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. *arXiv preprint arXiv:1903.04671*, 2019.

[77] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

[78] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[79] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[80] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[81] Rishabh Singh, Joseph P Near, Vijay Ganesh, and Martin Rinard. AvatarSAT: An auto-tuning Boolean SAT solver. 2009.

[82] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3104–3112, 2014.

[83] Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[84] G Tseitin. On the complexity ofderivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[85] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[86] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2674–2682, 2015.

[87] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, pages 2786–2796, 2017.

[88] Nathan Wetzler, Marijn JH Heule, and Warren A Hunt. DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429. Springer, 2014.

[89] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *arXiv preprint arXiv:1608.02644*, 2016.

[90] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[91] Caiming Xiong, Stephen Merity, and Richard Socher. Dynamic memory networks for visual and textual question answering. In *International conference on machine learning*, pages 2397–2406, 2016.

[92] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, 32:565–606, 2008.

[93] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, et al. Deep reinforcement learning with relational inductive biases. 2018.