UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Neural Guidance in Constraint Solvers

Permalink
https://escholarship.org/uc/item/6g54j8x2

Author
Lederman, Gil

Publication Date
2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Neural Guidance in Constraint Solvers

by

Gil Lederman

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Co-chair
Professor Sanjit A. Seshia, Co-chair
Assistant Professor William Fithian

Spring 2021

Neural Guidance in Constraint Solvers

Copyright 2021
by
Gil Lederman

Abstract

Neural Guidance in Constraint Solvers

by

Gil Lederman

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Edward A. Lee, Co-chair

Professor Sanjit A. Seshia, Co-chair

Boolean Constraint Satisfaction Problems naturally arise in a variety of fields in Formal Methods and Artificial Intelligence. Constraint Solvers, the specialized software tools that solve them, are therefore a core enabling technology in industry and research. They are normally used as black-box components, applied to practical problems such as hardware and software verification, test generation, planning, synthesis and more. Based on classical algorithms that have been optimized over decades by researchers, there is a noticeable gap between Constraint Solvers and the technology of Deep Learning, which over the last decade found its way into countless domains, outperforming established domain-specific algorithms.

This thesis aims to narrow this gap, and by using Deep Neural Networks, teach classical Constraint Solvers to "learn from experience". The research I present in this thesis starts by addressing the challenge of representation, using a Graph Neural Networks based architecture to process propositional formulas as graphs. I will then show how to automatically learn a solver's branching heuristic by mapping it to a Markov Decision Process and training it using Deep Reinforcement Learning. I will present an implementation based on different two competitive solvers, and experiments showing automatically learned heuristics to outperform the state of the art in the two domains.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

This work was supported in part by the TerraSwarm Research Center, one of six centers administered by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Later work in this thesis was supported in part by the National Science Foundation (NSF) award #CNS-1836601 (Reconciling Safety with the Internet), The Berkeley Deep Drive, and the iCyPhy (Industrial Cyber-Physical Systems) Research Center supported by Avast, Camozzi Industries, DENSO International America, Inc., Ford, Siemens, and Toyota.

Chapter 1

Introduction

There was a joke running around as the previous millennium was coming to a close. It went something like this: Bill Gates recently said at a conference that "If GM had kept up with the technology like the computer industry has, we would all be driving $25.00 cars that got 1,000 miles to the gallon.". In reply, GM issued a press release[1] saying that if GM had developed technology like Microsoft, we would be driving cars that spontaneously crash twice a day, require an engine re-installation after the road had been repainted, and occasionally lock us out of the car and refuse to let us in until we simultaneously lift the door handle, turn the key, and grab the radio antenna.

Almost two and half decades later, this joke still packs a bit of a punch. The gap between Computers and Cars has rapidly narrowed since then, as did the gap between computers and, well, practically everything. For the most part, we survived. People don't get locked out of their car twice a day, even though most modern vehicles are literally mobile computing platforms. However, there is a deeper truth to this joke, beyond the quip at Microsoft's expense. Complex hardware and especially software systems are notorious for having bugs, in a way that products of other engineering disciplines don't. We don't expect bridges or mines to have unexpected edge cases. A combustion engine may degrade due to wear and tear and fail over time, but it is uncommon that an unknown "vulnerability" would be detected in its design after two years of use. Structural engineers are able to give guarantees about the structure they build, in a way programmers often cannot. At the most abstract level, this problem could be attributed to a lack of some level of mathematical rigour in day-to-day programming, that is present in other engineering disciplines.

Good programmers are aware of this deficiency of software development, and tend to compensate for it through processes, methodologies, and tools of software development. Testing, code review processes, continuous integration and development are used to detect errors in the software. Using proven design patterns, strongly typed programming languages, or a uniform coding style are techniques that provide the programmer with "freedom from choice," and aim to reduce the probability of bugs occurring in the first place, sometimes

[1] In reality He didn't, and they didn't. It would be funnier if it were true, but it's made up.

entirely eliminating certain types of bugs (For example, eliminating some types of memory bugs in moving from C to Java and dropping pointers). Moreover, the infinite flexibility of software tends to help, since a bug found today would just be fixed in tomorrow's patch. And so in many circumstances, this lack of rigour does not pose a great problem. Many pieces of software are not critical, and as long as they work mostly as expected, we've grown accustomed to occasional small glitches in our smart devices, cryptic error messages on the web, or having to reboot our PCs from time to time for good measure.

For critical systems, where the cost of failure is high, and could be sometimes measured in human lives, we often look for a stronger degree of guarantees. *Formal Methods* is the discipline within computer science that allows us to derive some precise guarantees regarding software and hardware systems, and formally prove things about them (or rather their models). It is a collection of methods based on defining formal mathematical structures that describe the specification, operation and various properties of computations. These mathematical structures can take many forms, such as Finite State Machines (FSM), Petri Nets, Actor model, or an abstract machine. They come in different styles, matching those of program semantics, denotational, operational, and axiomatic. But invariably, many of the techniques that verify properties or relationships between those structures end up reducing them to Boolean functions and recasting the search for a solution or proof as a Boolean *Constraint Satisfaction Problem* (CSP).

1.1 Boolean Constraint Satisfaction Problems

The most well-known CSP is the Boolean Satisfiability Problem, called SAT [85]. It is, given a Boolean propositional formula ϕ such as $(\neg x \vee y) \wedge (\neg y \vee (\neg x \wedge y))$, the problem of deciding whether there is an assignment to the variables which satisfies ϕ. It was the first problem to be proved NP-complete. All CSP are NP-hard, and in fact SAT is the easiest, which is also why it is the most well-studied and widely used.

The reason SAT and other CSP are important (from a practical point of view), is that many useful problems can be reduced to them. For example, throughout the processes of producing microprocessors, logic circuits, which implement Boolean functions, undergo complex transformations. To verify that the transformed circuit still implements the same function, we have to show that the circuits are functionally equivalent. If we have two circuits on the same inputs, $C_1(x), C_2(x)$, we can reduce the question of their equivalence to the (un)satisfiability of the circuit $C_3(x) = C_1(x) \oplus C_2(x)$. Another important example is Transition Systems. If the set of states is S, and $|S| = 2^n$, we can express the transition relation over the states $\delta : S \times S \mapsto \{0, 1\}$ as a Boolean function over $2n$ inputs (the variables for current and next state), $F_\delta(x, y)$. This encoding of transition systems allows for example for Bounded Model Checking - checking whether a state is reachable within T steps. We define T copies of the n state variables, say, $x_1 \ldots, x_T$. Now, if $F_I(x)$ is a formula that defines the initial states and $F_s(x)$ defines the target set, the formula $F_I(x_1) \wedge F_\delta(x_1, x_2) \wedge \ldots, \wedge F_\delta(x_{T-1}, x_T) \wedge F_s(x_T)$

represents the reachability problem. Closely related is the reduction of AI planning problems to SAT [68, 44].

Another type of CSP is Quantified Boolean Formula (QBF), which is, given a propositional formula with quantifiers such as $\forall x \exists y (\neg x \vee y) \wedge (\neg y \vee (\neg x \wedge y))$, whether it is true, where $\forall x$ is understood as "For all x" and $\exists y$ as "There exists y". It is harder than SAT, and is in fact the canonical PSPACE-complete problem. Its interchanging quantifiers make it suitable for modeling 2-person games, or for example planning in presence of non-deterministic environment. We can ask whether there exists some policy x such that a safety property holds no matter what y the environment throws at us. Similarly, synthesis problems can be reduced to QBF, where we ask to find an implementation x such that a property $Q(x, y)$ holds for all possible inputs y (see for example in Solar-Lezama et al. [132]).

For a given Boolean formula, we can also ask not just whether it is satisfiable, but how many solutions it has. This is called the #SAT problem, and it is the canonical #P-complete problem. It naturally lends itself to questions of probabilities. Problems of probabilistic inference can be reduced to model counting through weighted model counting [115], or sometimes directly through unweighted counting [149]. *Satisfiability Modulo Theory* (SMT) [15] expands SAT to transcend propositional logic and include other background theories. It allows us to replace variables in the propositional formula with predicates that are interpreted over another theory, for example $3x + 2y \geq 5$ over Linear Real Arithmetic, or theories of arrays and bitvectors which help model software.

Despite the fact that these CSP are NP-hard, there exist algorithms and tools that (sometimes) solve them in practice. They are called *Constraint Solvers* (CS), or just solvers. State of the art SAT solvers are routinely used to solve industrial problems, many times through SMT (as in, SMT relies on SAT). More complex CS such as QBF and Model Counting solvers are not as common in industry yet, but are used and researched in academia (see Biere, Heule, and Maaren [23]).

1.2 Learning from Experience

It so happened that the developer of one such CS, A QBF solver called CADET, is a colleague of mine. He showed me a graph produced by Cadet as a certificate for a solution of a problem, which looked something like Fig. 1.1, and I immediately wondered about what seemed like repeating patterns in it. Following the discussion that ensued I learned that repeating patterns would not be unlikely given that problems are encoded from logical circuits. It turned out though that Cadet, like other CS, are completely oblivious to this fact. It can solve a 1000 circuit problems with the same repeating common components, yet this "experience" doesn't register, nor does it effect a single decision cadet will make when it solves the next problem.

This stands in stark contrast to how humans solve problems - we tend to adapt our techniques to the problems we see. Many times we can acquire valuable experience from solving small "toy problems," experience that serves us well when we face harder similar problems. This obliviousness to experience is also quite peculiar from the perspective of

modern techniques in *Artificial Intelligence* (AI). A few decades ago, classic AI was more about tools like cadet and others, which performed logic inference and reasoning according to fixed rules, early AI researcher's idea of what intelligence is. Since then however, AI based on symbolic manipulation had receded, and over the last decade Machine Learning (ML) and specifically "Deep Learning" [47] have taken the stage. Over the last decade, Deep Neural Networks (DNN, or just NN) have achieved super-human performance in a number of important tasks in various fields such as vision, speech, natural language, robotics and games. In all cases, the success of the NN architectures is based to a large part on learning from, or "experiencing" the data. Without any prior knowledge they managed to outperform algorithms crafted by domain experts. A classic example is AlphaZero [127], a (partly) NN architecture that, by playing against itself, managed to go from knowing nothing about games such as Chess or Go to defeating world champions.

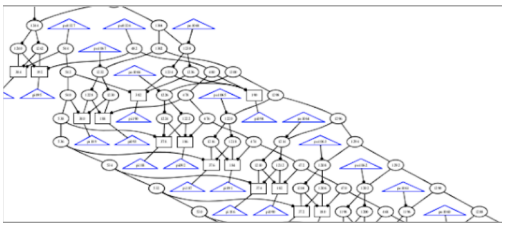


Figure 1.1: Satisfiability Certificate from Cadet.

The purpose of this thesis is to try and bridge that gap between NN and symbolic reasoning. Specifically, to combine the techniques of modern Deep Learning in a state of the art CS, in a way that allows it to learn from experience and become better than the manually-designed heuristics. It is part of a long line of research that focuses on leveraging ML towards improving CS from many different angles. There have been works using classical ML methods to choose between encodings for SAT by classifying formulas [124], approaches that automatically tune the many parameters of SAT solvers [57], portfolio methods [158] that classify formulas in order to choose the solving algorithm, and approaches that model SAT as a multi-armed bandit problem [80]. While this thesis focuses on QBF and model counting, a lot of work recently had been directed towards improving SMT through ML by learning to choose tactics [13] or solvers [118].

1.3 Overview

We turn first to the Deep Learning side. Chapter 2 starts with necessary background on NN, and reviews the previous attempts to apply them to logical reasoning. It presents the representation challenges that arise when trying to apply NN to logical formulas, and proposes a solution that addresses those challenges based on a Graph NN architecture. It then presents some empirical results that validate it, and a conjecture trying to explain the architecture's success in deciding satisfiability. Once we have a suitable representation of logical formulas, we turn to integrating it within Cadet in Chapter 3. It starts with background on the DPLL algorithm and CDCL-based solvers, then discusses the details of bridging the two different worlds, of NN and CS. It then describes Cadet, and shows how to map its branching heuristic to the Reinforcement Learning settings. Finally, it describes

experimental results of a Cadet-based implementation, outperforming the state of the art by an order of magnitude (10x) on some challenging problem families. Chapter 4 shows how this method can be adapted to `SharpSAT`, a modern competitive exact model solver. It starts with some background on Model Counting, and then presents `Neuro#`, A model counter augmented with a learning component for its branching heuristic. It discusses the idea of semantic features and how they may be implemented in problems of modeling dynamical systems. It shows experimental results of `Neuro#` improving on the state of the art, and offers some interpretation of the learned heuristic. Chapter 5 describes attempts to apply these ideas to the important domain of SAT. It covers the challenges and several approaches attempted, which so far have achieved very partial success or none at all. Chapter 6 contains a discussion of several potentially fruitful directions for future work, and some concluding remarks.

Collaborators

I am indebted to quite a few colleagues and mentors for their involvement and contribution to the research presented in this thesis. The pronoun 'We' refers to myself and all respective Co-Authors of relevant material, as listed below:

- Chapter 2 is based on unpublished material, and the ideas in it are based on discussions with Markus Rabe, Edward Lee, and Sanjit Seshia.
- Chapter 3 is adapted from the Paper "Learning Heuristics for Quantified Boolean Formulas through Reinforcement Learning" [78], Co-authored with Markus Rabe, Edward Lee, and Sanjit Seshia.
- Chapter 4 is adapted from the Paper "Learning Branching Heuristics for Propositional Model Counting" [146], Co-authored with Pashootan Vaezipoor, Yuhuai Wu, Faheim Bacchus, Sanjit Seshia, Roger Grosse, and Chris Maddison.
- Chapter 5 is based on unpublished work done with Pashootan Vaezipoor, and discussions wih Faheim Bacchus and Roger Grosse

Chapter 2

Representation

The first challenge facing us when trying to incorporate neural networks into constraint solvers is one of *representation*. Specifically, while there are many moving parts in modern state of the art logical constraint solvers, which we discuss in detail throughout the next chapters, the main entity common to all of them is the propositional Boolean logical formula, for example:

$$(a \vee b) \wedge \neg c \rightarrow \neg a \wedge (c \vee a) \tag{2.1}$$

In this chapter we develop a representation of such Boolean formulas, and the corresponding Neural Network architecture which can process it.

We will start with some background on the concepts of representation, embedding and induced bias in the NN literature, review previous attempts at representing Boolean formulas growing out of the Natural Language Processing (NLP) literature, and build upon them to come up with a new representation and architecture that better match the structure of propositional logic. We will then present experiments and empirical results to establish some properties of the new representation, and discuss its remarkable success in the specific task of deciding satisfiability.

2.1 Neural Networks Preliminaries

Terminology and Notation

Neural Networks, as the name implies, were originally roughly motivated by the connectivity structure of biological Neurons in the brains of living creatures. However, from an abstract mathematical point of view, they can be considered as a specific type of parametrized function. Indeed, they are referred to as *function approximators*. We will often represent them schematically as a computation graph. For example, the most well-known basic example is the *Perceptron*, represented graphically in Fig 2.1 [112]:

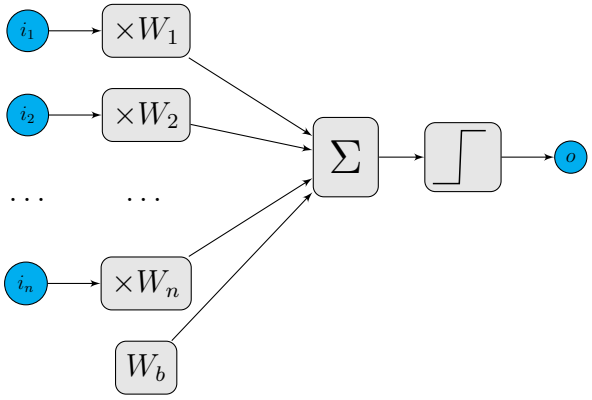


Figure 2.1: Computation Graph of a Perceptron

Equivalently, we can define the same operation in a functional notation. The Perceptron is then viewed as a function P parameterized by $W_1, \ldots, W_n, W_b$. Parameters in Neural Networks are also called *weights*, and when clear from context we will abuse notation and denote W as an entire vector or matrix of parameters, hence the parameterized function defining the perceptron is denoted P_W:

$$o = P_W(i_1, \ldots, i_n) = \mathcal{H}\left(\sum_{k=1}^{k=n} W_k i_k + W_b\right) \tag{2.2}$$

$$\mathcal{H}(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases} \tag{2.3}$$

Where $\mathcal{H}(x)$ is the Heaviside step function. In the context of neural networks, $\mathcal{H}$ is called, quite intuitively, an *activation* function. It controls whether the perceptron is "activated". The single weight W_b is called the *bias* of the perceptron (not to be confused with the concept of bias in Neural Networks, to be introduced shortly). Mathematically, it is clear that the $n+1$ parameters define an Affine Hyperplane in $\mathbb{R}^n$, and P_W defines the function which outputs 0 or 1, depending on which half-space the point $(i_1, \ldots, i_n)$ resides in.

A generalized version of the Perceptron is called a *Neuron*, the basic unit of a Neural Network. The only difference is that we relax the definition of the activation function[1], and allow it to be some univariate function other than the Heaviside function. Common activation functions used in practice are $\tanh$, $ReLU(x) = \max(0, x)$, $Sigmoid(x) = \frac{1}{1+e^{-x}}$.

Since the neuron is the basic building bloc of neural networks, its scheme of "apply an activation function to a shifted weighted sum of the inputs" is so common that it is often omitted from graphical representations of larger NN. The next simplest construction, called

[1]The Perceptron was inspired by the biological neuron, and so the Heaviside function represented the neuron "firing" or not.

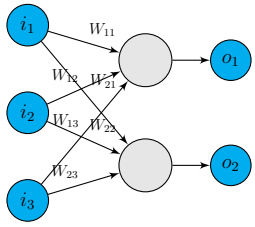


Figure 2.2: A single feed-forward layer with 3 inputs and 2 neurons

a *feed-forward layer*, is an arrangement of multiple neurons in parallel. As seen in Fig 2.2, it is often represented schematically in the literature, such that the sum, the activation, and the bias weights are implicit. Since the sum is assumed as part of the Neuron, the weights are associated with the incoming edges. For larger networks, weights are also omitted and specified separately.

Functionally, a single layer can be represented using matrix arithmetic in vector notation.

$$O = f(WI + W_b) \tag{2.4}$$

Where $f \in \mathbb{R} \mapsto \mathbb{R}$ is any activation function, lifted pointwise to $\mathbb{R}^m \mapsto \mathbb{R}^m$. $I \in \mathbb{R}^n$ and $O \in \mathbb{R}^m$ are vectors in Euclidean Spaces, and W, W_b are the network weights (W_b is usually implicit in graphical representation). In the example of Fig 2.2 we would have $I \in \mathbb{R}^3, O, W_b \in \mathbb{R}^2, W \in \mathbb{R}^{2\times 3}$.

The next construction generalizes the single layer, and is called an n-layer *feed forward network*, or *multi-layer perceptron* (MLP), and is simply a functional composition of such layers. Note, the architecture of a MLP is specified entirely by layers' width, which induces the dimensions of its parameter matrices. It could be thought as the "signature", or "type" of the MLP component. For example, Fig 2.3 depicts the component $MLP(2, 3, 1)$. The single layer in Fig 2.2 is $MLP(3, 2)$. MLP components can be sequentially composed with each other as long as their input/output dimensions match. Multi-layered NN are sometimes called "Deep Neural Networks".

Training Neural Networks

Definitions

Before NN can be usefully employed, they have to be trained to become good at their task. In *Supervised Learning*, the goal is to learn a function based on input-output examples. More formally, Let us denote a network as f_W (where W stands for all of its parameters, regardless of partition or dimensions), its input domain as X and output domain as Y. We train networks on A *Training Set*, which is a set of example tuples $\{(x_i, y_i)\}_{i=1}^n$, such that for input x_i, the correct answer is y_i. We define a *Loss function* $\mathcal{L} : Y \times Y \longmapsto \mathbb{R}$. Intuitively,

it is a measure of distance[2] in the space Y, which lets us measure "how far" the network's output is from the correct answer for a given input. The network loss for a tuple (x, y) is then $\mathcal{L}(f_W(x), y)$, and the entire *Training loss* is defined to be:

$$\mathcal{L}_{training}(W) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}_i(W) \tag{2.5}$$

$$\mathcal{L}_i(W) := \mathcal{L}(f_W(x_i), y_i) \tag{2.6}$$

Finally, training a NN means optimizing the parameters W with the objective of minimizing the training loss.

Gradient Descent Optimization and Batching

The actual optimization of the NN is usually done with a variant of the *Gradient Descent* (GD) algorithm. Intuitively, GD is a simple first-order hill-climbing (or rather descending) algorithm which can be used to iteratively find a local optimum of a differentiable scalar function $f : \mathbb{R}^n \longmapsto \mathbb{R}$. It starts with some guess $x_0 \in \mathbb{R}^n$, and at every round moves a small step in the direction of the steepest incline (decline) of f, i.e, in the direction of its gradient at x. In the context of NN, fixing the training set and the network architecture, the training loss in Eq. 2.5 can be seen as a scalar function on the space of network weights W. The only thing we now need to apply GD to optimize the network's weights is to be able to efficiently compute the derivative $\frac{d\mathcal{L}_{training}}{dW}$, which, since derivative is a linear operator, is reduced to computing $\frac{d\mathcal{L}_i}{dW}$. The GD update rule with step size η is then:

$$w := w - \eta\nabla\mathcal{L}_{training}(w) = w - \frac{\eta}{n}\sum_{i=1}^{n}\nabla\mathcal{L}_i(w) \tag{2.7}$$

In practice though, NN are not optimized with "vanilla" GD, mainly due to two complications:

- It is highly unlikely for any non-trivial problem to have a loss function that is convex over the parameter space. In the common non-convex case, GD will get "stuck" at a sub optimal local optimum, and in practice would depend entirely on initialization.
- As seen in Eq 2.7, the loss is a sum over the entire training set. In practice, n could be tens of millions of examples. To compute the gradient of the sum we must compute the gradient of $\mathcal{L}_i$, the loss for each example, and this for every single gradient step.

The *Stochastic Gradient Descent* (SGD) algorithm solves both of these problems by introducing randomness. Intuitively, instead of calculating the exact loss gradient over n examples, we calculate a cheap approximation to the gradient by considering only one random

[2]While similar to distance, loss functions are generally not in fact *metrics* in the formal topological sense

sample from the training set. More formally, given a random permutation $\{\sigma(1), \ldots, \sigma(n)\}$, the update rule of SGD on the *ith* step is:

$$w := w - \eta \nabla \mathcal{L}_{\sigma(i)}(w) \tag{2.8}$$

SGD only has to compute gradient at one sample point for each step, and while it is still not guaranteed to converge to global optimum (unless the function is convex, a highly unusual case in NN), with appropriately decreasing step size it is guaranteed to converge to a local optimum, and its noisiness makes it less likely to get stuck at a **bad** local minimum in practice. SGD however has its own inefficiencies:

- Convergence in practice can be slow and noisy. A single example gives a poor approximation of the real gradient.
- Gradients are computed using fast hardware (GPUs) and software libraries that support vectorization. Computing gradients one at a time is highly inefficient.

And so in practice we use a hybrid of SGD and GD (which in this context is called *batched* Gradient Descent). Instead of using either one sample or all of them per gradient step, we set a *Batch Size* hyperparameter, and on every step we sample an entire *mini-batch* (or just batch if clear from context) and compute the gradient on them. This both stabilizes the convergence, and efficiently utilizes modern frameworks implementation. To a large extent, NN became popular as modeling components because they facilitate the efficient computation of those derivatives using the well-known *backpropagation* algorithm [113], allowing for first-order optimization. Moreover, the advent of popular *auto-diff* frameworks such as TensorFlow and PyTorch [101] (see at App. A.4) makes computing the gradient of networks with respect to weights a straightforward matter.

Classification and Regression

We will be using NN for ultimately one of two tasks - regression or classification. Regression means the network's output is in $\mathbb{R}^n$. The standard loss function used in training for such tasks is the *Mean Squared Error* (MSE), which is simply scaled squared l_2 norm. For $x, y \in \mathbb{R}^n$ it is:

$$MSE := \frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2 = \frac{1}{n} \|x - y\|_2^2 \tag{2.9}$$

The classification scenario is slightly more involved, since NN are by their nature a smooth construction. Suppose our training set is made of tuples (x, y) such that $x \in \mathbb{R}^n, y \in \{1, \ldots, m\}$. We use a NN to classify its inputs into m discrete categories by computing m outputs (and so there are m neurons in the last layer). Those outputs are called *logits*, and let us denote them $l_i, 1 \leq i \leq m$. On top of this layer, we apply a *Softmax* function, from $\mathbb{R}^m$ to $\mathbb{R}^m$:

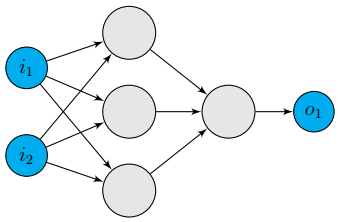


Figure 2.3: Multi-layer Perceptron with dimensions $2, 3, 1$.

$$p_i = Softmax(l)_i := \frac{e^{l_i}}{\sum_{j=1}^{m} e^{l_j}} \tag{2.10}$$

The softmax ensures that the final m outputs are all positive, and sum to one. We treat them as parameters for a multinomial probability distribution over the m categories. We next proceed to represent y in *one-hot representation*. That means mapping $y \in \{1, \ldots, m\}$ to $(y_1, \ldots, y_m) \in \{0, 1\}^m$:

$$onehot(y) := (y_1, \ldots, y_m) \tag{2.11}$$

$$y_i = \begin{cases} 1 & i = y \\ 0 & i \neq y \end{cases} \tag{2.12}$$

And finally define the *Cross-Entropy Loss* function as:

$$CE := -\sum_{i=1}^{m} y_i \log p_i = C(l) - l_{i=y}, C(l) = \log \sum_{i=1}^{m} e^{l_i} \tag{2.13}$$

This loss term is slightly less intuitive, but it is in fact the cross-entropy between the distribution concentrated on the single correct answer and the distribution over the categories produced by the network, $(p_1, \ldots, p_m)$. Cross Entropy between two distributions p, q, also called relative entropy, is defined to be $H(p, q) = -\mathbb{E}_p \log q = -\sum_{i=1}^{m} p_i \log q_i$. In our case, the loss is $H(y, p)$, where y is the point mass distribution on the correct answer. The cross entropy, up to a constant, is the *Kullback-Leibler Divergence* (KL) between the distributions. Specifically, $KL(p\|q) = H(p, q) - H(p)$ While the KL divergence is not exactly a metric in the space of distributions[3] , we can still think of it intuitively as "distance between distributions". It grows smaller as distributions become similar, and $KL(p\|q) = 0 \iff p = q$.

[3]It is asymmetric, but converges to be symmetric as q gets infinitesimally close to p.

2.2 Representation and Architecture in Neural Networks

One of the main strengths of NN is their flexibility with respect to the modalities of data they can process. Beginning with Krizhevsky, Sutskever, and Hinton [73], they have been repeatedly used to achieve state of the art results in different domains - Images & Video [152], Natural Language [97], Robotics [140], Speech [99], Combinatorial Games [127] and mode, often surpassing established domain-specific algorithms. Many factors have contributed to the rise of deep neural networks in those various domains. One of the main ones, which is used to incorporate domain knowledge into a network architecture and will become important in our context, is referred to in the literature as *Induced Bias*.

Expressiveness

As reviewed in section 2.1, a NN can be seen as a parametrized function $f_W : \mathbb{R}^n \longmapsto \mathbb{R}^m$. Fixing an architecture implicitly induces a mapping from $\mathbb{R}^{|W|}$ to $\mathbb{R}^n \mapsto \mathbb{R}^m$. A natural question that arises is what functions are in the range of this mapping, or equivalently, what functions the NN could possibly approximate. This notion is informally called 'expressiveness'. It seems intuitive that the more parameters, the more "expressive" the network is. And indeed, the *Universal Approximation Theorem* [] formalizes this intuition. It shows that under mild conditions, any function in $f : \mathbb{R}^n \longmapsto \mathbb{R}^m$ can be approximated arbitrarily close by $MLP(n, k, m)$ as $k \to \infty$. That is, for every "reasonable" $f \in \{\mathbb{R}^n \mapsto \mathbb{R}^m\}$ and $\varepsilon > 0$ there exists k and appropriate weights W such that $\|MLP_W(n, k, m) - f\| < \varepsilon$.

And so, just about **any** function can be approximated arbitrarily well by a simple network architecture - just a 2-layer feed-forward network. Indeed, with k large enough, it can be as expressive as we'd like. Visually, it is easy to see that feed-forward connectivity between layers is the "most general", every neuron in a layer is connected to every neuron in the following layer.[4] However, naive expressiveness like that doesn't come for free - as k goes up, so does the number of weights in the network. More weights increase the dimension of the optimization space, and accordingly the amount of time, computing resources, and most importantly, data, required for training. And so a fundamental trade off in NN design is between expressiveness and size. We address it by leveraging domain knowledge.

Induced Bias

How can a network be both expressive and efficient with respect to number of weights? The key observation is that a specific NN is never required to approximate "any arbitrary function", but rather a much more constrained function, where the constraints depend on the expected properties of the data distribution, and embedded either implicitly or explicitly in the structure of the data. Reflecting and leveraging those constraints in the architecture is

[4] More formally, the weights are the full cross-product of the two layers

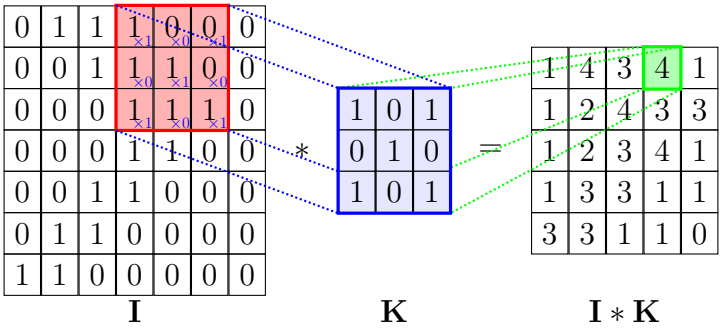


Figure 2.4: The Convolution operation of the image I with the 3×3 filter K computes at every (2D) coordinate the inner product of K with the local neighborhood of that coordinate in the input (in red).

referred to as *Inducing Bias* on the network. We will describe two classic network architectures as examples of inducing bias:

Processing Images with Convolutional NN

Consider the task of processing and acting on visual data. Suppose our input data are 100×100 black and white images we are tasked with classifying (for example, detecting whether some object is in the image). A naive implementation could simply use a $MLP(10^4, k, 2)$ for some k. We "flatten" the 100×100 input matrix into a 10^4 dimensional input vector, and send it through the MLP.

However, this naive solution ignores some important structure that is common to all images in the real world. The most notable is the spatial locality bias. The meaning of a single pixel (or a small rectangle of pixels) is far more dependent on its surrounding pixels than on pixels far away. While our input space is, in theory, 2^{10000}, in practice the distribution of inputs the network will process is biased towards only those inputs that have this spatial locality property. Furthermore, this meaning derivation is (mostly) invariant to translations. A triangle is a triangle, whether its on the left side or the right side of the image. The MLP cannot capture those constraints. Think for example on $MLP(10^4, 10^4, 2)$, where each unit in the second layer can be intuitively thought of as computing the "meaning" of the respective pixel. Because we flattened the input and the connections between consecutive layers is their cross product, Every pixel by definition depends on **all** other pixels, and has to learn the locality principle on its own.

A *Convolutional Neural Network* (CNN) [] addresses these issues by learning a set of small, local filters, and *convolving* them along the 2D image, as seen in Fig 2.4. Each such application of filters is called a *Convolutional Layer*, and a network processing visual data is usually composed of several of these layers composed on top of each other. For our purposes,

it is enough to note that by matching the structure of the data, the architecture has several advantages compared to the naive solution with MLP:

- By construction, each output of a convolutional layer only depends on its local neighborhood in the input.
- By construction, since the same K is convolved along the entire input, the output is invariant to translations.
- The size of filters (hence number of weights to optimize over) are orders of magnitude less than what we would get by a cross-product connectivity like in MLP.

By using pre-existing domain knowledge to constrain the network's expressiveness, we make it more efficient. By making it respect by construction the invariants of the data, we save weights (because the network doesn't have to re-learn the same translated property over and over. The same small kernels are applied to all locations). This is a theme to keep in mind when thinking of processing logical formulas, one that repeats itself throughout NN architecture and design. The good architecture follows the data.

Processing Sequential Data with Recurrent NN

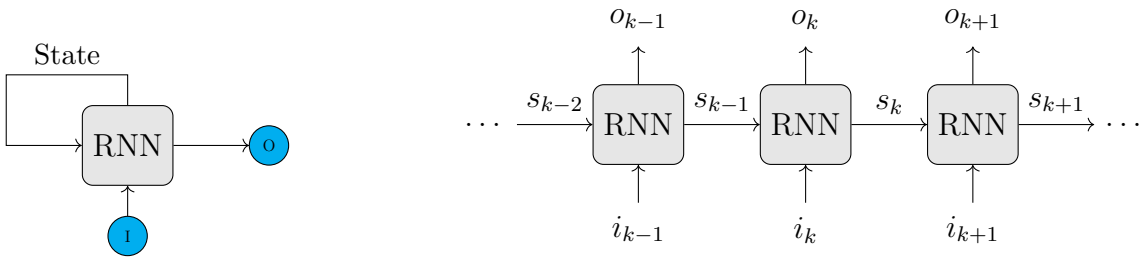
A second example of an induced bias is the use of *Recurrent* NN (RNN) for the processing of sequential data. It is often applied to processing a sequence of words in the field of Neural Language Processing (NLP)[]. The difference compared to MLP or CNN mirrors the difference between image and text input domains. When processing still images (unlike video), there is no temporal dependency between the inputs. How the network processes a given image is completely independent on the previous image and will have no effect on the next. With language, the meaning of a word in a sentence clearly depends on the previous ones.[5] RNNs (Fig 2.5a) match this structure by incorporating feedback and state variables (sometimes called hidden state), making their output depend on both the input and the previous state. The diagram in Fig 2.5b is showing the RNN *unrolled through time*. Functionally, the network approximates the function $(i_k, s_{k-1}) \longmapsto (o_k, s_k)$.

Low Dimensional Embeddings

NN often deal with extremely high-dimensional input domains. For example, If a language has 100k words, every word can be represented as a one-hot vector of dimension 100k. Every sentence is a sequence of such vectors. In the image domain described in section 2.2 every image is a 10k dimensional vector. However, the actual input distribution in the input domain is sparse. In the language example, though the input domain is $\mathbb{R}^{10^5}$, there are only 100k different inputs, each, one-hot encoded, along just a single dimension.[6] Generally, in other

[5] Sometimes also on future words, but this is a less important detail in our exposition.

[6] In general, NN treat all numbers as floating point. Even if it is a black-and-white image or a one-hot encoded vector composed only of $\{0, 1\}$



(a) An abstract computation tree of an RNN. (b) Unrolled RNN.

Figure 2.5: Recurrent NN

domains, it is less clear how **precisely** the input is distributed, but it is assumed to be distributed along a low-dimensional sub-manifold of the input domain. For example, in the image domain, we do not expect to encounter any input that looks like random noise, because it contradicts the "spatial locality" principle. Mathematically, a low-dimensional manifold is said to be embedded in $\mathbb{R}^n$.

Informally, part of what NN do is to "invert" this embedding. They take input in a high dimensional space, and apply a sequence of non-linear transformations to produce a low dimensional representation. For example, $MLP(n, \ldots, k)$ takes input in $\mathbb{R}^n$, transforms it through a sequence of layers and produces output in $\mathbb{R}^k$ where $k << n$. Through a slight abuse of (mathematical) notation, these are referred to in the literature as *Low Dimensional Embeddings*, Dense Embeddings, or just Embeddings.

Word Embeddings

The special case of words in a language encoded as one-hot vectors is worth describing in more detail, as it will become the starting point of our investigation into representing logical formulas. As described in Section 2.2, a sentence is processed by an RNN as a sequence, one word at a time.[7] However, NN process inputs in $\mathbb{R}^m$ (in practice, m floating point numbers), so we must encode the words somehow as m dimensional vectors, called word embeddings.

One hot encoding (see Eq. 2.11) is used to represent categorical information, where the input domain is a discrete space, which generally has no natural ordering.[8] And so, as seen in Fig 2.6, we look up each word in a dictionary, and encode its index as a one-hot vector. Note that a m-dimensional one-hot vector processed by a feed-forward layer $MLP(m, n)$ (we assume the Identity function as activation. That is, no activation) is equivalent to multiplying the vector by an $m \times n$ weight matrix. If we encode the integer k, it is equivalent to choosing the kth row of the matrix, which is therefore known as an *embedding matrix*.

[7] In practice modern NLP models use more elaborate techniques and process sentences also from the end towards the beginning, or all at once [147]. But this simple approach outlines the motivation and basic idea

[8] A note able exception is Integers, which are often also modeled as categorical data in one-hot, even though they are ordered

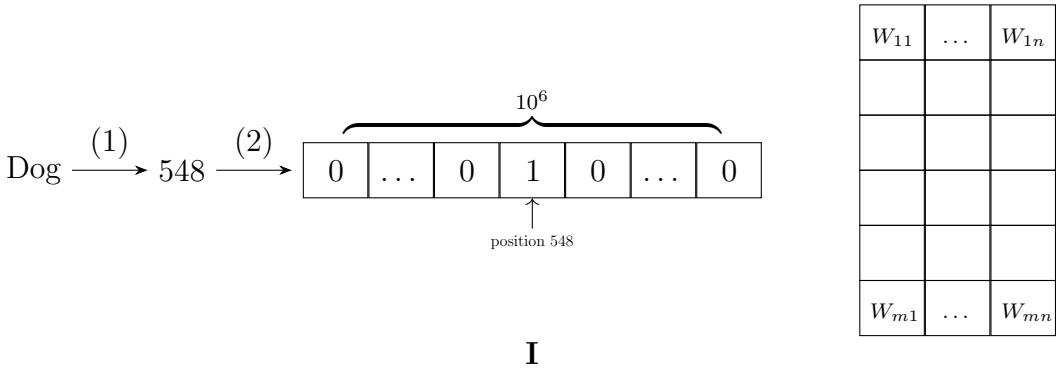


Figure 2.6: Encoding a word into an n-dimensional vector. (1) Lookup A word's index in a dictionary. (2) Encode the index as a one-hot vector. (3) Multiply by an "Embeddings Matrix" to effectively choose a raw as an embedding for the index.

Learning Embeddings

Embeddings in practice are learned in one of two ways. If we have enough labeled data and computing power, the straightforward approach is to learn directly during supervised training from what is referred to as the *Downstream Task*. In an NLP task, this effectively means treating the embedding matrix as normal parameters. We initialize them randomly, train the entire model, and the resulting word vectors (the rows of the embedding matrix) at the end of training are the embeddings. The intuition is that during the training process, words are automatically mapped to the $\mathbb{R}^k$ embedding space in such a way that is of the most benefit towards performing the specific task. This simple approach produces the best embeddings towards the specific task, but for a few caveats. NN effectiveness is known to grow with the size of the model. Training large models from scratch in a supervised setting requires large amounts of two expensive resources - labeled data, and computing power. The size of both must grow with the number of model parameters in order to effectively train. Additionally, there is the problem of *transfer*. The embeddings learned are generally task-specific. Embeddings learned from training on a sentiment analysis task (which tries to judge the "positiveness" of a sentence) aren't necessarily effective for translation from English to French.

A method that gets around these difficulties is *Transfer Learning* (TL). In the context of NN, TL means training on some foundational (semi-)supervised task with enough labeled data and computing power, and using parts of the trained model for other downstream tasks. The canonical example is image classification task on ImageNet [37], a dataset of millions of images labeled into thousands of categories. Large reference architectures [49, 128, 138] are trained at a considerable cost, and are made publicly available.[9] Users download these pre-trained models, remove the last layer (which performs the actual classification), and use

[9]https://pytorch.org/docs/stable/torchvision/models.html

what's left as a general-purpose feature extractor, upon which they build and train their own architectures for other downstream tasks, such as image captioning, object manipulation, etc'. The intuition behind this approach is that because the ImageNet task is foundational and difficult, the learned features are transferable, and useful for many other vision tasks. A similar technique is used in NLP, with the main difference being that models are trained in an unsupervised (arguably semi-supervised) manner, so there is no lack of data. This is because A text corpus comes with its own labels, so to speak, the structure of sentences in a language. And so semi-supervised tasks used towards this purpose are invariably a variation of a 'cloze' task̃footnoteThe cloze task was used in tests given to humans long before it found use in NLP - predicting some masked words based on their context. Increasingly sophisticated and large[10] architectures [91, 108, 102, 38] are used to learn pre-trained word embeddings. We will review one class of them in more detail shortly, the RNN-based *Language Models*.

2.3 Representing Logical Formulas

Every advancement in science has a history, and context. When it comes to representing logical formulas for processing by NN, this history is firmly rooted in the NLP literature. Unsurprisingly so, perhaps, because logical formulas (and mathematical objects in general) are usually represented as text to us humans, ML researchers included. We will start with describing how these early attempts grew, the tools they used, and their shortcomings when applied to the domain of logical formulas solved by modern CDCL based constraint solvers. We will then see how to address these shortcomings by our proposed graph-based architecture.

Structured Text as... Text

The early attempts to process logical formulas by NN were based on a simple premise - from an abstract point of view, logical and algorithmic tasks are nothing but structured, arguably unnatural, natural language tasks. They can be seen in the context of a still ongoing research effort, aiming to apply the standard toolkit of NLP to tasks involving text that is more structured than natural language, such as predicting output of code snippets [165], mathematical identities [164], symbolic integration [77], Boolean formulas [3]. We will start by describing two of the main tools used throughout these attempts, Language models and the Sequence-to-Sequence framework, and how they were applied.

Language Models

RNN-based Language Models are trained directly on sentences, learn to model the distribution of sentences in a language, and can efficiently sample from them. More Formally, we assume

[10]The largest NLP models tend to be an order of magnitude larger than models used for vision tasks, up to 10^{11} parameters

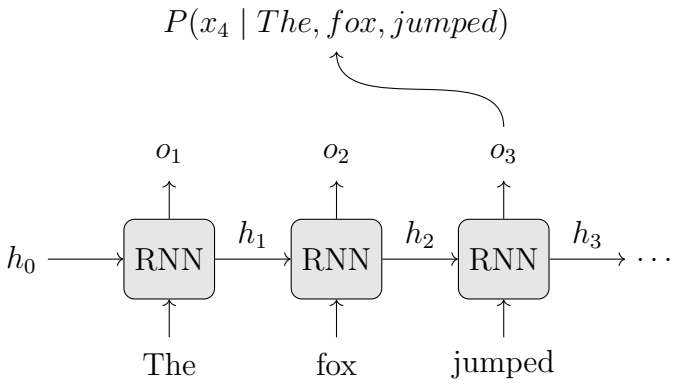


Figure 2.7: A simple language model trained on a sentence.

a language of n tokens. Now, considering a sequence of tokens $\mathcal{X} = (x_1, \ldots, x_k)$ as a random variable, the RNN learns to model the distribution as:

$$P(\mathcal{X}) = P(x_1) \prod_{i=2}^{k} P(x_i \mid x_1, \ldots, x_{i-1}) \tag{2.14}$$

Where the dependency on previous tokens factors through the hidden state of the network (Fig. 2.7). Recall from Sec.2.2 that the RNN is a function $(i_k, s_{k-1}) \longmapsto (o_k, s_k)$. We make sure the dimension of o_k is precisely the number of tokens in the language n, and treat it as logits representing the output distribution of the the k-th word in the sentence, which allows us to use classification loss (where each token in the dictionary is a class). Language Models trained this way can capture the structure and style of natural language texts, and are often used as pre-trained feature extractors for other downstream NLP tasks. With enough data they can be trained to produce texts that mimic anything from Obama's speeches to Chinese poetry, and perform related tasks (for example, classify a paragraph to those two categories). This versatility motivated researchers to apply it to more unnatural languages. For example, what if we train a model on a text corpus in the "style" of C programs? These were the first steps towards whats called today "Big Code" [2] - training models on a large set of computer programs, and using them towards a host of coding related tasks - from actual Program Synthesis [94], to correcting syntax errors [19], obfuscating programs [81], improved error reporting [29], and more.

While our focus in this work is on RNNs as a starting point towards our goal of representing formulas, its worth mentioning that more recently, it was discovered that transformer-based language models scaled up to the order of 10^{11} parameters achieve state of the art results on many *different* language tasks [38, 109], even without additional task-specific training! [28]

Sequence to Sequence Tasks

Sequence to Sequence, sometimes called seq2seq, is an approach originally developed by Google to be used in machine translation [136], and later used for other language tasks such as question answering and text summarization [126]. As the name implies, the approach is suitable for problems which can be formulated as processing a sequence of tokens, and producing another sequence. Indeed, the classical example is machine translation, where the input is a sentence, say, in English, and the output is the translation in French.[11]

The model is built from two parts, an *Encoder* and a *Decoder*. Both of them are RNNs. The Encoder processes the sentence in English one word at a time, where each word is embedded as described in section 2.2. The last hidden state, containing information about the entire sentence, is passed to the Decoder. The decoder in turn takes in the state from the encoder as its initial hidden state, and starts producing the translation, one word at a time. This is done the opposite way to how we encoded the input sentence - the decoder outputs on each step a logits vector the size of the French dictionary, and a softmax layer assigns probability to each potential word. At each time step, the decoder receives the hidden state from the previous step, and the previous word it emitted. This goes in until the decoder chooses to emit an "End Sequence" token, whereas the translation is complete. While similar to a Language Model in its architecture, the decoder uses the encoder-provided hidden state as its h_0, rather than a fixed initial state. The other main difference is the training - an encoder-decoder model is trained using labeled, task-specific data.

The seq2seq approach and its variants achieved impressive results on machine translation and was flexible enough to be used successfully in other contexts - for example, for Image Captioning, only the decoder part is used, and the hidden state is the output of a convolutional network rather than the encoder. For sentiment analysis, where the output is a single score, only the encoder is used. This success motivated researchers to test whether this approach could work on more structured tasks. As seen in Fig. 2.8, any list manipulation algorithm such as sorting or inverting a list can be seen as doing "translation". In fact, any terminating computer program with inputs and outputs can be formulated similarly. What if we train an Encoder-Decoder model on a lot of such pairs of sequences (there's certainly no shortage of data!) and then evaluate it on a sequence it hasn't seen before? It turns out that the Encoder-Decoder architecture does not perform very well. The code snippet in example no. 3 in the figure comes from Zaremba and Sutskever [165]. While the RNN can learn to "translate" short programs, the authors discovered it has severe limitations. Such a translation works only when trained and tested on a limited subset of extremely simple programs, with bounded nesting (no double loops), a few operations, and ending with a print statement. When trained to invert or sort sequences, the main limitation of RNNs is generalization to longer sequences. It can be trained to invert sequences of up to 20 elements, but fails entirely when tested on sequences longer than those seen in training [48].

[11] Modern state of of the art machine translation is done today using Transformers, not RNNs as in the time of the writing

(1) $7, 14, 9, 3, 2, 8 \longrightarrow 2, 3, 7, 8, 9, 14$

(2) $2, 8, 4, 1, 33 \longrightarrow 33, 1, 4, 8, 2$

(3)
```
j=8584
for x in range(8):
    j+=920
b=(1500+j)
print((b+7567))
```
$\longrightarrow 2, 5, 0, 1, 1$

Figure 2.8: Algorithmic "Translation" examples. (1) Sorting a list of Integers. (2) Inverting a list of integers. (3) Computing the output of a Python program (code snippet taken from Zaremba and Sutskever [165])

Text as Trees

Another two relevant lines of research in the NLP literature grew from the prevalence of structure and logical patterns within natural language, both stemming from the role of *composition* in language.

Syntax Trees and Recursive Tree NN

Researchers in Linguistics have long ago learned the structure common to sentences in a language. Words in the English language for example, belong to one of 9 different categories called *Parts of Speech* (POS), among them nouns, verbs, adjectives, etc'. These different categories of words serve different functions in a sentence, and importantly, are intertwined with the implied tree structure of a sentence, called a *Parse Tree*, or *Syntax Tree*. For example, using the standard POS tagger of the popular NLP toolkit nltk [84], the sentence "The quick brown fox jumps over the lazy dog" is parsed into the syntax tree in Fig. 2.9. Syntax parsing has long played an important part in computational language processing, and was used to derive semantic representations of sentences. Within the context of NN, it meant that the input of models in NLP are oftentimes trees rather than sequences.

In accordance with the induced bias principle of Section 2.2, it is not surprising therefore to find Tree-like NN architectures designed to process variable-sized syntax trees. There are several variants in the literature [130, 139], and all are forms of a *Recursive*[12] NN. At the heart of these architectures is the notion of learning how to *compose*. The intuition is not too

[12]Note the difference from Recurrent NN. Recurrent NN are a subclass of Recursive NN.

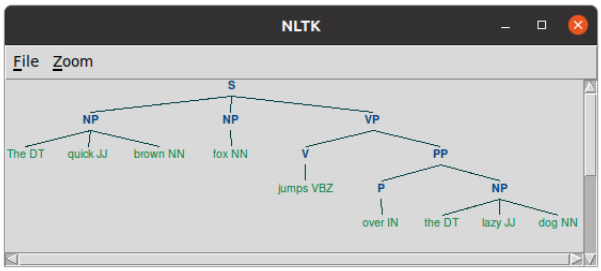


Figure 2.9: Parse Tree

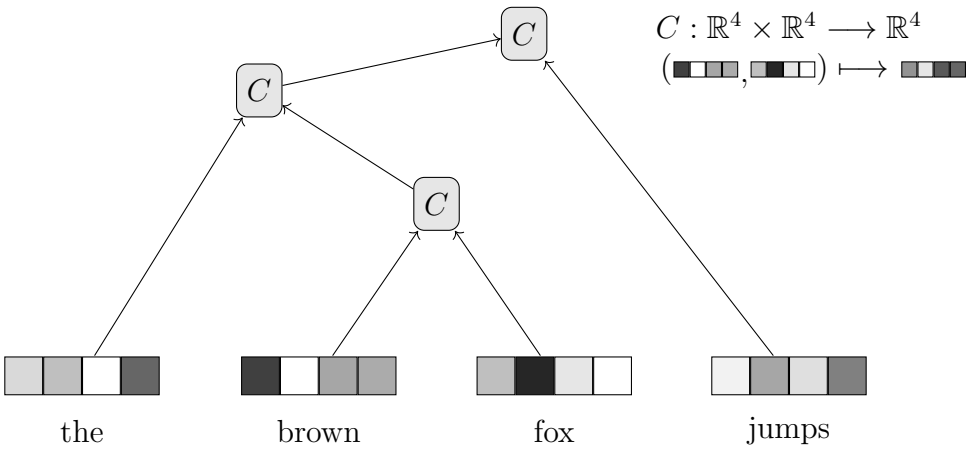


Figure 2.10: A Binary Tree-NN processing (a binary parse tree of) the beginning of a sentence. What the network actually learns is the combination function C. A simple implementation uses a single shared network for C. Incorporating more domain knowledge, we can learn multiple different combination functions according to the types (POS tags) non-leaf nodes.

different from the Recurrent case. The recursive element in a sequence processing RNN is also learning how to compose, albeit in a sequential, fold-like manner. It composes the hidden state which encapsulates the aggregated history with the current input, to produce a new hidden state. Similarly, A Binary Tree-NN builds an aggregate representation of its input tree by recursively composing representations of children into that of the parent (Fig. 2.10) via a learned composition function of type $\mathbb{R}^d \times \mathbb{R}^d \longmapsto \mathbb{R}^d$, where d is the embedding dimension of each node (leaves are words, and non-leaf nodes are combination of words, up to the entire sentence). In a non-binary Tree-NN, two common techniques are used for composition. Either a maximum number of children per node are assumed and required padding added, or an *aggregation* function is used, which is defined on a variable number of inputs - for example sum, average, or more generally attention-based mechanisms (see App. A.2). Every non-leaf node can have its own learned composition function

Informal Logic in Natural Language

The effort to discern and distill logical patterns from natural language has its roots run all the way back to the Syllogism of Aristotle. While these patterns need not be as rigid as formal propositional or predicate logic, most speakers of a language still recognize that the word 'not' preceding an adverb inverts its meaning, or that double negation cancels itself. Words and patterns like 'and', 'or', 'if X then Y', 'unless', 'all', 'implies', are interpreted by speakers of the language as operators in the "naive" predicate logic of everyday use. Question Answering, an important NLP task, is well known for requiring reasoning ("Socrates is a man. All men are mortal. Is Socrates mortal?"), but even more mundane tasks such as automatic sentiment analysis of movie reviews implicitly require reasoning about logical operators and composition. A tree structured NN is suitable for capturing those patterns, as demonstrated in Socher et al. [131]. For example, in the sentence "not a very good movie", the model would give the intermediate node "good movie" a positive score. Composing the word 'very' would give "(a) very good movie" an even higher positive score, yet the composition with the word 'not' inverts the sentiment, and produces an overall negative score at the root node.

Logical Formulas as Trees

From some perspectives, learning the semantics of formal propositional operators is a trivial task for A Tree-NN. So much so, that it is used as a sort of a sanity test in Socher et al. [131], where they train a Tree-NN to evaluate binary expressions represented as shallow binary trees over the dictionary $\{\mathbf{true}, \mathbf{false}, \neg, \wedge\}$. They found that with no more than 6 such training examples, they could learn weights for the two operators that achieve zero training error (and of course, zero error on all such deeper trees, by the compositional nature of the problem). This is somewhat unsurprising, as NLP usually deals with the "forward" problem, of evaluating a sentence. The difficulty is in the ambiguity of the language, not in the underlying computational problem, which is clearly in $\mathcal{P}$. When we remove the ambiguity of natural language, the expressiveness of NN easily captures the semantics of negation and conjunction as non-linear functions.

In Allamanis et al. [3], the authors explicitly set out to to learn a dense embedding for logical formulas, addressing some of the issues that arise once we tackle problems more complex than evaluation. The authors present a tree-NN based architecture called SemVecs, which learns to represent propositional formulas with variables, and more generally symbolic expressions such as polynomials. In their settings[13], the input data is composed of Boolean propositional formulas over a fixed set of named variables $\{a, b, c, \dots\}$, such as:

$$\neg c \wedge (a \wedge (a \vee c) \wedge b) \tag{2.15}$$

Their goal is to find vector embeddings for formulas like Eq. 2.15 such that they would overcome the mismatch between syntax and semantics. Specifically, Boolean propositional

[13] We will focus only the logical propositional formulas part of that work.

formulas on n variables are a function of type $2^n \longmapsto 2$. There are 2^{2^n} different such functions, yet there are far more syntactically correct formulas over n variables (technically there are infinite such formulas, think of $(a, a \wedge a, a \wedge a \wedge a, \dots)$). And so we say two such formulas are equivalent if they induce the same function, and this equivalence relation partitions all formulas to 2^{2^n} equivalence classes. For example, $a \vee b$ and $b \vee a$ are semantically equivalent, even though as a sequence of tokens they're different. In fact, it is easy to see that equivalent formulas can have drastically different syntax. Ideally, we'd want the representation of formulas to be depend on semantics only, and be invariant to the syntax. What this amounts to in the context of learning embeddings, is that the Euclidean distance between the vector representations of two semantically equivalent formulas in $\mathbb{R}^d$ is small. With such an embedding, if we check the nearest neighbors of a formula, we should expect to find many of them are from the same equivalence class.

Given such an embedding which clusters equivalent formulas together, it should be relatively easy to learn on top of it a classifier. And so, at least in theory, we could also invert this intuition - if we train a model to classify formulas into their equivalence classes, it should learn embeddings that are productive towards this downstream task. After all, the same phenomenon, to some degree, exists in natural language processing. Sentences with more or less the same meaning can be composed of largely different words and expressions. And yet, models trained on downstream tasks such as translation that learn embeddings of entire sentences learn to overcome this, and cluster similar expressions/sentences together. While this could theoretically work for formulas as well, In Allamanis et al. [3] the authors show that at least with modest amounts of data (and accordingly, computation), this is not enough. Standard RNNs and Tree-NN architectures are too attached to the syntax. In order to achieve good clustering, or "invariance to syntax", they extend the standard Tree-NN architecture with a component that explicitly encourages a clustering behaviour on the embeddings (with respect to equivalence classes).

The authors observe a connection between semantics and reversibility. Specifically, if we have a formula $F(a, b) = F_1(a, b) \vee F_2(a, b)$, knowing the semantic equivalence class of F, F_1 allows us to predict the equivalence class of F_2 *better than it allows for predicting its syntax*. Translating this observation to the embedding space, reconstructing F_2's embedding from those of F, F_1 will be easier if those embeddings represent semantics rather than syntax. They exploit this observation by introducing a denoising autoencoder component (See Appdendix A.1) they call SubExpAe. At each non-leaf node of a tree, they combine the children embeddings $(r_{c_0}, \dots, r_{c_k})$ into r_p as described in Sec. 2.3, using a different function (that is, weight matrices) for different operators, so in practice we have multiple combine functions such as $\text{COMBINE}_\vee, \text{COMBINE}_\wedge$ at different non-leaf nodes. They then randomly choose a child node (WLOG, c_0), and then use SubExpAe to reconstruct r_{c_k} from $(r_{c_1}, \dots, r_{c_k}, r_p)$. They add the reconstruction loss as a regularization to the classification loss. On several synthesized datasets of boolean formulas, they show this specialized component significantly improves on the standard Tree-NN baseline when measuring how clustered together are the embeddings of semantic equivalence classes.

Challenges of Formula Representation in the Context of Constraint Solvers

When it comes to representing formulas in the context of modern Constrain Solvers, the approaches we've reviewed from the literature have some significant shortcomings. Let us go over the challenges of representation in this domain.

Invariance

We've seen invariance, where the convolution operation in a CNN reflects the invariance to translation in images - A triangle is a triangle, however we translate it across the image. The convolution bakes this assumption into the architecture. There are other invariants in image processing however, that are not reflected in the CNN. Rotation, for example, or inversion, should probably not change the meaning of an image (or, in practice, the class of an ImageNet data point). One way practitioners make NN "invariant" to these operations is by *Data Augmentation*. We can, for each image, synthesize new versions of it by randomly applying the operations we want to NN to be invariant to, and add those synthesized versions to the training set under the same class. The intuition is that by training on the downstream classification task, the "invariance" is baked into the parameters of the network.

A bit more formally, we have the input space $\mathcal{X}$, a function $f : \mathcal{X} \longmapsto \mathcal{Y}$, and a family of mappings $\mu_{i\in I} : \mathcal{X} \longmapsto \mathcal{X}$. We say f is invariant to μ if $\forall i \in I, x \in \mathcal{X}$ we have $f(\mu_i(x)) = f(x)$. For another function $g : \mathcal{X} \longmapsto \mathcal{Z}$ (representing the NN), invariance to operations in μ is defined the same. We say g is invariant to f if $f(x_1) = f(x_2) \implies g(x_1) = g(x_2)$.

Armed with our new definitions, let us turn to invariants in Boolean formulas. The work described in Sec. 2.3 defines f to be the mapping that sends an element of $\mathcal{X}$ - formula over n variables - to its interpretation, which can be seen as a number in $\{1 \ldots, 2^{2^n}\}$ (the semantic equivalence classes). The SemVecs architecture is the function g, which takes inputs in $\mathcal{X}$ and outputs the formula's embedding in $\mathcal{Z} = \mathbb{R}^d$. Note, the architecture does not incorporate any domain knowledge about the invariance of formulas - for example, their $\text{COMBINE}_\vee$ function is not symmetric in its variables, so it has to learn that $\vee$ is symmetric on its own, through training.[14] Of course, no amount of training with SGD will ever make a NN truly invariant by our definition. For $x_1 \neq x_2$ we shouldn't generally expect a NN g to output $g(x_1) = g(x_2)$. But by means of the SubExpAe and the downstream task itself, the training process encourages g to be "approximately invariant" to f. That is, $f(x_1) = f(x_2) \implies g(x_1) \approx g(x_2)$. In words, embeddings for semantically equivalent formulas should be "close" to each other. It should be noted though, that even encouraging a network to be approximately invariant to syntax is a technique with inherent limitations. The computational task of deciding whether two formulas are semantically equivalent is NP-hard - it is at least as difficult as deciding equivalence to an UNSAT formula. In general, a NN implements a bounded computation graph. There are no magic weights that can make it

[14] Though, this is quite probably on purpose, since their point was explicitly to **learn** the invariants by means of the SubExpAe component.

solve the problem as we scale the number of variables. And UNSAT is just one equivalence class, as we recall their total number grows doubly exponential.

Variables

The approaches we've seen so far have one unwanted legacy from their NLP origins. In Allamanis et al. [3], the formulas the network learns to embed are interpreted over a **fixed** set of variables, $\{a, b, c, \dots\}$. Both the variables and the Boolean operators ($\vee, \wedge, \neg, \implies$) are taken as tokens, as if they were words of a sentence. For the operators this makes sense - logical OR has the same meaning in two different formulas. This is similar to the word "dog" having more or less the same meaning in every sentence, with some statistical relation to other words from the dictionary, for example "cat" or "bark". It therefore makes sense to learn a word embedding for "dog" which stands for the word any time its being processed. However, the variables in formulas in the context of CS work differently. They are anything but a fixed set, and they take a very different role from a word in a sentence. The variable x_1 in one formulas has absolutely no relation to the variable x_1 in a different formula. Much like `i` in a `for` loop block, it is a bounded variable, meaningful only within the context of a single formula. It makes no sense to learn a unique embedding for x_1 and another for x_2 as if they were distinct words. In fact, in most scenarios, A CS couldn't care less if we exchange every occurrence of x_1 with x_2 and vice versa - it does not change the satisfiability of the formula, nor its number of solutions. For all practical purposes, a SAT solver is invariant to variable names.

Computational Limitations

RNNs, and to a lesser degree Tree-NN, have some practical and conceptual limitations with scale. RNNs were always prone to the infamous "vanishing/exploding gradient" optimization problem (see App A.3). Due to their "back propagation through time" (BPTT), from the perspective of computing gradients, RNNs have as many layers as the length of the input sequence. The optimization problems were largely solved by popular RNN variants like the "Long Short Term Memory" (LSTM) networks [53], but both computation and memory grow linearly with sequence length, and since the RNN architecture is sequential by nature, it cannot be easily parallelized [89]. They have technical difficulties with what is called in NLP jargon Long-term dependencies [144], remembering relevant information that will only be used many time steps in the future. Some of these problems can be overcome with the Transformer architecture [147, 161], but at the price of $O(n^2)$ computation complexity.

If we write down a typical formula solved by a modern CS, it'd easily be a sequence of $\geq 10^6$ tokens. This is about 3 orders of magnitude above what is considered standard for LSTM networks. A constraint appearing at the beginning of a formula can easily "become" critical a million tokens later, as in $\neg x_1 \wedge \cdots \wedge (x_1 \vee \neg x_{37931})$.

Formulas as Graphs

The challenges outlined above are all related to the inherent mismatch between sequence processing architectures and the structure of logical formulas. We address them drawing inspiration directly from the design of CS, and how they represent propositional formulas.

CNF Representation

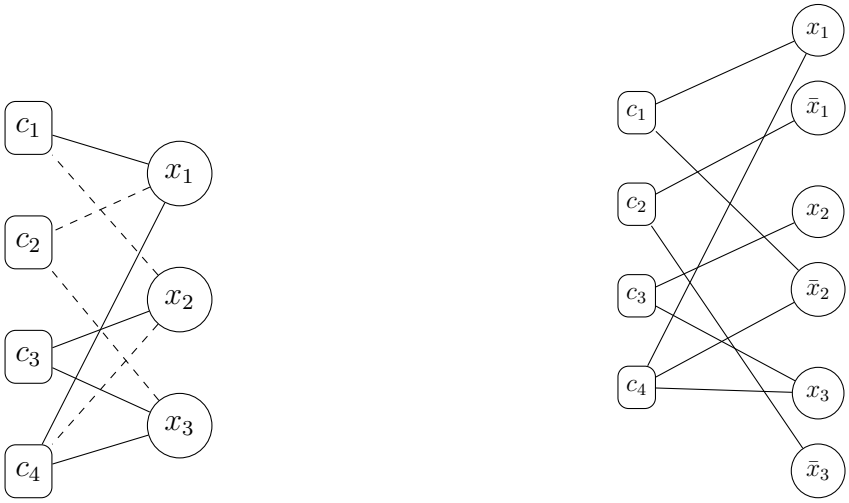
We start with describing *propositional* (i.e. quantifier-free) Boolean logic. Propositional Boolean logic allows us to use the constants 0 (false) and 1 (true), variables, and the standard Boolean operators like $\wedge$ (*"and"*), $\vee$ (*"or"*), and $\neg$ (*"not"*). We assume that the reader is familiar with their semantics and that it is clear that all other Boolean operators can be defined in terms of these operators.

A *literal* of variable v is either the variable itself or its negation $\neg v$. By $\bar{l}$ we denote the logical negation of literal l. $P(v, c) \in \{+, -, \bot\}$ returns the polarity of variable v in c (or $\bot$ if $v \notin c$). A *clause* c of length s is a disjunction of s literals, $l_1 \vee \cdots \vee l_s$. We say that s is the size of c, and that $l \in c$ if c contains the literal l. A formula Γ is said to be in *conjunctive normal form* (CNF) if it is a conjunction of clauses, $\Gamma = c_1 \wedge \cdots \vee c_t$. We then say that $c_i \in \Gamma$. Any Boolean formula can be efficiently transformed into CNF. The Tseitin Transformation [145] considers a formula as a circuit, and replaces each operator with a fixed small set of clauses on the inputs and a new auxiliary variable which represents the output. This increases the size only linearly, which is quite fortunate, as it turns out (for reasons we will explore in the next chapter) that CNF is precisely how CDCL based CS expect their input formulas to be represented. For the time being, we thus assume that all formulas are given in CNF.

Incidence Graphs

An (undirected) Graph is a tuple $G = (V, E)$, where V is a set of nodes, and $E \subseteq V \times V$ is a symmetric relation on V. A bipartite graph is a tuple (V_1, V_2, E) such that $E \subseteq V_1 \times V_2$. A *Heterogeneous Graph* is a graph where both nodes and edges are labeled. They are a tuple $(V_1, \ldots, V_j, E_1, \ldots, E_k)$, understood as j node types and k edge types. Note, our definition of heterogeneous graph also generalizes a *Multigraph* - we allow to differently typed edges between the same pair of nodes. With a slight abuse of notation, we will say of a heterogeneous graph that it is bipartite if it has exactly 2 node types, and all edges go between the partitions. It is well known (and cleverly used in the design of SAT solver heuristics) [7] that a CNF formula can be represented as a graph in a few variants. They are called *CNF Incidence Graphs*, or, in this work, CNF Graphs (see Fig. 2.11):

Definition (Variable Incidence Graph (VIG))**.** *Given a CNF formula $c_1 \wedge \cdots \wedge c_t$ over the set of variables $X = \{x_1, \ldots, x_n\}$, the Variable Incidence Graph is a (X, E), where $(x_i, x_j) \in E \iff \exists c\, x_i, x_j \in c$*



(a) CVIG (b) CLIG

Figure 2.11: CNF Incidence Graphs of $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$

Definition (Clause-Literal Incidence Graph (CLIG))**.** *Given a CNF formula $c_1 \wedge \cdots \wedge c_t$ over the set of variables $X = \{x_1, \ldots, x_n\}$, the Clause-Literal Graph is a bipartite graph $(X \cup \bar{X}, C, E)$, where $\bar{X} = \{\bar{x}_1, \ldots, \bar{x}_n\}$, C is the set of clauses, and $E = \{(l, c) | l \in c\}$.*

Definition (Clause-Variable Incidence Graph (CVIG))**.** *Given a CNF formula $c_1 \wedge \cdots \wedge c_t$ over the set of variables $X = \{x_1, \ldots, x_n\}$, the Clause-Variable Graph is a Heterogeneous Graph (X, C, E_p, E_n), where C is the set of clauses, $E_p = \{(x, c) | x \in c\}$, and $E_n = \{(x, c) | \bar{x} \in c\}$.*

Processing CNF graphs

According to the "induced bias" principle, to process input data in the form of graphs, we should be looking for an architecture that mirrors this structure. Unsurprisingly, the architecture that is most helpful here is called a *Graph Neural Network* (GNN). First described in Scarselli et al. [117], it resurfaced a few years later based on modern ML components and pipeline in Li et al. [79], and have since been used in many domains [16]. We will describe the architecture in general, and then its specific application to CNF graphs.

A Gentle Introduction to Graph Neural Networks

Hundreds of variants of GNNs have been described in the literature over the past few years. This is not meant to be a thorough review of this still developing field (we direct the interested reader to Battaglia et al. [16]), but rather enough intuition and definitions to understand the application to CNF graphs. A GNN is a network that takes as its input a graph, and produces an embedding for each of its nodes. It does so iteratively, for n iterations, the final

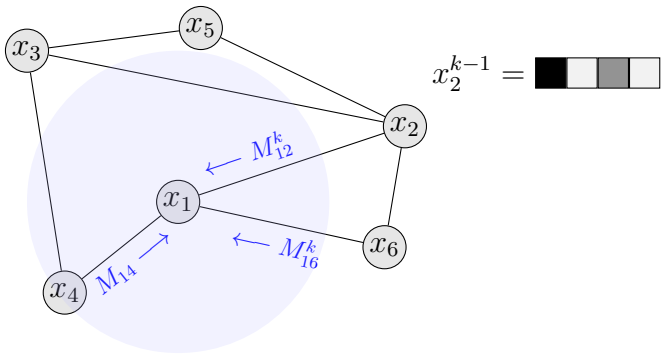


Figure 2.12: A Graph Neural Network learns a function that combines embeddings of neighboring nodes. Here we see the computation of $x_1^k = F(x_1^{k-1}, A(\{M_{14}^k, M_{12}^k, M_{16}^K\}))$, where 3 messages from x_1's neighbors are combined to generate its k'th embedding.

iteration being the output embedding. At iteration i, every node rebuilds its embeddings according to the $(i-1)$'th embeddings of itself, and its neighbors in the graph (see Fig. 2.12).

Formally, Let us assume an (undirected) Graph $G = (X, E)$, where $X = \{x_1, \ldots, x_n\}$, and the existence of an edge between x_i, x_j is equivalent to $(x_i, x_j) \in E$. We denote the k'th iteration embedding of x_i as $x_i^k \in \mathbb{R}^d$, and $X^k = \{x_1^k, \ldots, x_n^k\}$. The equation defining the GNN (up to X^0) is:

$$x_i^k = F(x_i^{k-1}, A(\{M_{ij}^k | (x_i, x_j) \in E\})) \tag{2.16}$$

$$M_{ij}^k = M(x_i^{k-1}, x_j^{k-1}) \tag{2.17}$$

Where F and M are transformations, and A is an aggregation function, symmetric in its arguments. All of them are learned functions implemented by NN. This equation is rather general, but its meaning is quite intuitive. M_{ij}^k can be thought of as a Message sent to x_i from each of its neighbors in iteration k.

Formally, A, the aggregation function is a mapping finite **sets** of vectors in $\mathbb{R}^d$ to $\mathbb{R}^d$. Less formally, its an operation that takes a variable number of vectors and produces a single aggregate vector. Common choices are `sum`, `max`, or more generally some form of learned attention mechanism as described in App A.2. A aggregates all the messages M_{ij}^k that arrive from x_i's neighbors. Finally F takes the result and combines it with the previous iteration's embedding of x_i to get the k'th version. The remaining undefined component is X^0, the initial embeddings of the nodes. There are two ways to bootstrap this initial embedding. One option is to use a fixed random vector for all the nodes in the graph. This makes sense when the nodes are featureless, and it incorporates the principle that the nodes are indistinguishable apart from the topology of their neighborhood. In this scenario, the final k'th node embeddings of two nodes with isomorphic k-neighborhoods are identical. A second option is to initialize each node according to its features in the relevant domain. We will see examples of both uses soon.

In different contexts, one of two interpretations of the GNN's operation tend to be useful[15]:

- A GNN learns to execute a message-passing algorithm. F, M, A as defined above induce a "message-update" rule. Specifically, as has been demonstrated in Dai, Dai, and Song [35], A GNN can learn message-update that are akin to variational inference algorithms in graphical models, like mean field inference or loopy belief propagation.
- In every iteration every node gathers information from its direct neighbors, and after k iterations, each node potentially has information from all nodes up to k hops away. And so A GNN with k iterations can be seen as performing a convolution of size k, building an embedding for each node based on a patch of size k around it. If the graph is a perfect grid, we get the regular convolution[16] on images. There's a big difference between grids and general graphs though - the pooling part. In a grid, its quite obvious how to "compress" the grid by down-sampling between convolution layers. It is not at all clear how that down-sampling works in general graphs.

Processing CNF graphs with GNN

While a GNN is usually defined on a general graph, we can adjust it to fit the specific structure of CNF graphs. Of the incidence graphs, we chose to concentrate on the CVIG and CLIG, the two non-lossy representations (It is generally impossible to recover a formula from its VIG representation). Both variants of the CNF graph are bipartite, and so we match this structure with a 2-step message propagation process.

Let us consider a CLIG. It has two types of nodes, or two partitions, which we denote L and C, literals and clauses respectively. As our goal is learning branching heuristics, we are chiefly interested in literal embeddings, so we start with them. We split each iteration into two half-iterations, going from literals to clauses, and back again (see Fig. 2.13). Since clauses and variables/literals have inherently different meanings, we also choose to propagate distinct messages from them. Formally, this means we "double" the edges between literals and clauses, getting $G = (L, C, E_{lc}, E_{cl})$ (Alternatively, we can think of it as a directed bipartite graph). For CVIG, we do the same "doubling" of the edges, thus ending up with $G = (V, C, E^+_{vc}, E^-_{vc}, E^+_{cv}, E^-_{cv})$.

Graphs vs. Sequences

The graph structure addresses the challenges of representing formulas in a way that is useful for a CS. It abstracts away by construction much of the spurious syntax in formulas as sequences of tokens - Invariance over order, and variable names. As we shall see, in CLIG

[15]There's also a popular Harmonic Analysis point of view, but it is not used in this work.

[16]More or less the regular convolution. There's no padding, and the function is parameterized differently. For example, within a regular convolution patch, distance doesn't matter, there's no actual message-passing going on, the entire patch is looked at "together". Whereas in a GNN there's clear difference between how information from different hop-distances is processed.

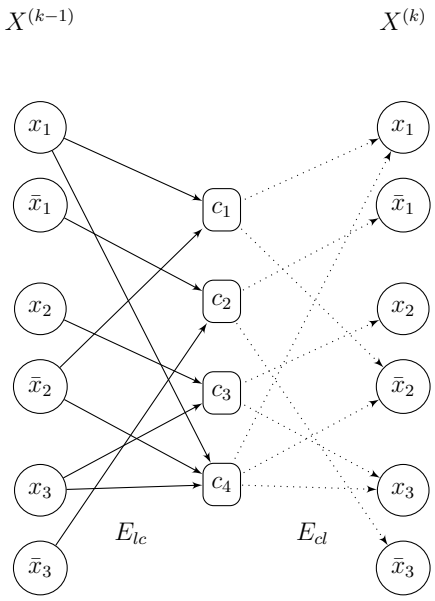


Figure 2.13: Processing the k'th iteration of a CLIG - from Literals to Clauses and back again.

we can achieve invariance to negation. The graph computation is local - the embedding of each node only considers its τ-neighborhood. Thus, it can be parallelized by partitioning the graph. This locality also means that unlike in sequences or trees, the path of gradient propagation is bounded by τ, unrelated to the size of the problem.

Implementation

We implemented our model in PyTorch. Its dynamic graph auto-differentiation [101] makes it possible to support variable size of graphs.[17] We begin with the structure of the input and how to batch it. We then present a simple implementation of an encoder to illustrate its structure, and proceed with several variations around this structure.

Input and Batching

Each of our data points is a bipartite graph, which represents a formula in CNF format. Suppose it has n variables and m clauses, both ordered. It is specified as a tuple of matrices (V, C, A), where:

- V is a matrix in $\mathbb{R}^{n \times d_{v_0}}$, where d_{v_0} is the dimension of initial variable feature vector. It can be zero, which represents identical, featureless nodes.

[17] At the time of implementation (2017) PyTorch was unique in that respect compared to static autodiff frameworks. By now other frameworks such as TensorFlow 2.0 have this functionality.

- C is a matrix in $\mathbb{R}^{m \times d_{c_0}}$, where d_{c_0} is the dimension of initial clause feature vector. It can be (and often is) zero.
- A is the adjacency matrix of the bipartite CNF graph, of dimension $m \times n$. We will sometimes speak of its first and second dimensions as "clause dimension" and "variable dimension", respectively. It is a sparse matrix, using the default sparse representation of PyTorch, which saves a list of indices and values instead of the dense matrix. We define it as:

$$A_{ij} = \begin{cases} 1 & \text{if } v_j \in c_i \\ -1 & \text{if } \bar{v}_j \in c_i \\ 0 & \text{otherwise.} \end{cases}$$

As mentioned in Sec. 2.1, in practice most times training is done using batching. When the input is a tensor of fixed dimension, batching is an easy matter of stacking a bunch of tensors along a new dimension. With variable-sized graphs we have to work a bit more. The main observation is that multiple graphs can be seen as distinct connected components of a single graph made of their union.

Suppose we are given a batch of b samples $((V^{(1)}, C^{(1)}, A^{(1)}), \ldots, (V^{(b)}, C^{(b)}, A^{(b)}))$, where each sample is from a different formula with n_i variables and m_i clauses. We take the adjacency matrices $(A^{(1)}_{n_1 \times m_1}, \ldots, A^{(b)}_{n_b \times m_b})$, and put them along the diagonal of a single adjacency matrix $A^{\text{batch}}_{\sum_i n_i \times \sum_i m_i}$, as seen in Fig. 2.14. With the sparse representation in mind, this reduces to renumbering the k'th graphs indices vector by adding $\sum_{i=1}^{k-1} n_i$ $\sum_{i=1}^{k-1} m_i$ to its clause and variable dimensions respectively. We concatenate $(V_1, \ldots, V_b)$ along their existing n_i dimension, and $(C_1, \ldots, C_b)$ along its m_i dimension, to get $V^{\text{batch}}_{\sum_i n_i \times d_{v_0}}$ and $C^{\text{batch}}_{\sum_i m_i \times d_{c_0}}$.

Model Implementation

Our main component is called an *Encoder*. It takes a graph as the tuple described above (which could in fact be a set of graphs batched as described above), and returns an embedding for each node in the variable/literal partition. It implements the functions F, M, A defined in Sec. 2.3, and as outlined there, we will specify how to compute a single, i'th iteration, where the total number of iterations τ is a hyperparameter.

CVIG Implementation In CVIG representation, we have a heterogeneous graph with two different edge types, positive and negative, which are then doubled to get $G = (V, C, E^+_{vc}, E^-_{vc}, E^+_{cv}, E^-_{cv})$. Accordingly, we split the adjacency matrix A into $A = A^+ - A^-$, the positive and negative adjacency matrix, respectively, where $A^+_{ij} = max(0, A_{ij})$. We define the GNN iteration as:

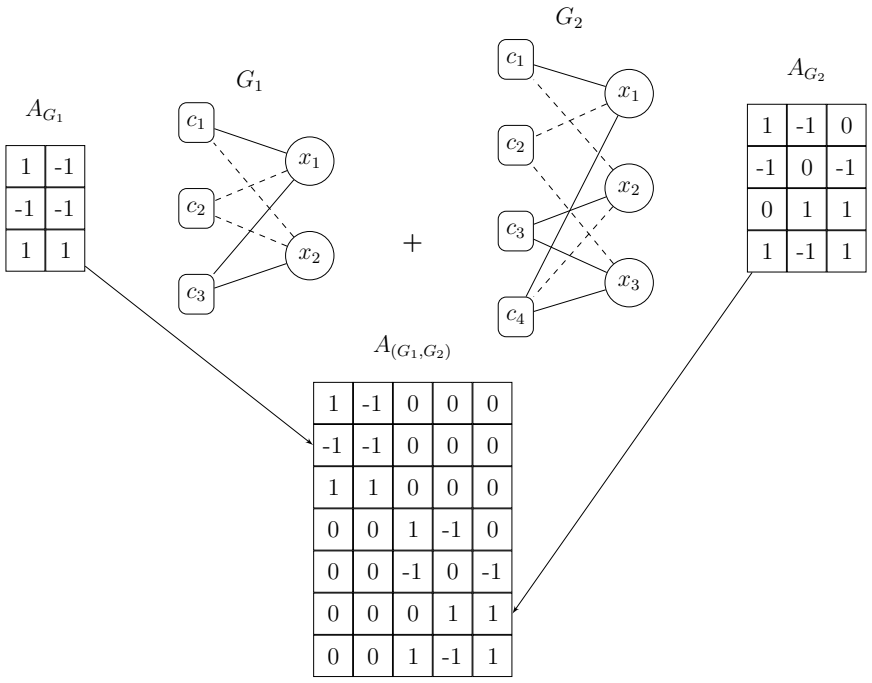


Figure 2.14: Batching adjacency matrices

$$\tilde{c}^{(t)} = \mathrm{ReLU}\left(\sum_{v \in c} W_{vc}^{P(v,c)} v^{(t-1)} + B_{vc}^{P(v,c)}\right) \tag{2.18}$$

$$\tilde{v}^{(t)} = \mathrm{ReLU}\left(\sum_{v \in c} W_{cv}^{P(v,c)} c^{(t)} + B_{cv}^{P(v,c)}\right) \tag{2.19}$$

$$v^{(t)} = GRU(v^{(t-1)}, \tilde{v}^{(t)}), \quad c^{(t)} = GRU(c^{(t-1)}, \tilde{c}^{(t)}) \tag{2.20}$$

Where matrices denoted $W_{vc}^{+}, W_{vc}^{-} \in \mathbb{R}^{d_c \times d_v}, W_{cv}^{+}, W_{cv}^{-} \in \mathbb{R}^{d_v \times d_c}, B_{vc}^{+}, B_{vc}^{-} \in \mathbb{R}^{d_c}, B_{cv}^{+}, B_{cv}^{-} \int \mathbb{R}^{d_v}$ are learned parameters, and GRU is A Gated Recurrent Unit with the appropriate dimensions. We compute the sum over negative occurrences and positive occurrences of variables separately with A^{+}, A^{-}. For example, for the sum in the computation of $\tilde{c}^{(t)}$ we multiply A^{+} by the matrix in which the i'th column is $W_{vc}^{+} v_i^{(t-1)}$, A^{-} by the matrix in which the i'th column is $W_{vc}^{-} v_i^{(t-1)}$, and add the two terms. This way the actual computation is done using efficient vector operations.

CLIG Implementation In the CLIG representation the two node types are literals and clauses, and after doubling we have just two edge types, in $G = (L, C, E_{vc}, E_{cv})$. The GNN iteration is defined a little differently:

$$\tilde{c}^{(t)} = \text{ReLU}\left(\sum_{l \in c} W_{lc} l^{(t-1)} + B_{lc}\right) \tag{2.21}$$

$$\tilde{l}^{(t)} = \text{ReLU}\left(\sum_{l \in c} W_{cl} c^{(t)} + B_{cl}\right) \tag{2.22}$$

$$\hat{l}^{(t)} = [\tilde{l}^{(t)}, \tilde{\bar{l}}^{(t)}] \tag{2.23}$$

$$l^{(t)} = GRU(l^{(t-1)}, \hat{l}^{(t)}), \quad c^{(t)} = GRU(c^{(t-1)}, \tilde{c}^{(t)}) \tag{2.24}$$

Where $W_{lc} \in \mathbb{R}^{d_c \times 2d_v}, W_{cl} \in \mathbb{R}^{d_v \times d_c}, B_{lc} \in \mathbb{R}^{d_c}, B_{cl} \in \mathbb{R}^{d_v}$. The main difference from CVIG is Eq. 2.23, that "ties" the two literals by concatenating them. The intuition behind this step is twofold. Unlike in the CVIG, there's nothing in the graph itself that relates a literal x to its negation $\bar{x}$. "Tying" the literals between iterations represents the connection between them that is missing in CLIG. When we tie them using concatenation (or a function of it), we also make the resulting embedding invariant to negation, in that if we rename a variable x to $\bar{a}$ (and so $\bar{x} = a$), the resulting literal embeddings are equal. Note, this doubles the actual literal embedding dimension to $2d_v$, and accordingly, the respective dimension in the parameter matrices.

To implement this computation with vector operations we again split the adjacency matrix into A^+, A^-, but this time we stack them along the variable axis, to get a $m \times 2n$ literal-clause adjacency matrix, which we multiply by the $2n \times d_v$ matrix of the literal embeddings from the previous iteration (see Fig. 2.15).

Model Variations

There are multiple variants of the model(s) described above, all of them sharing a similar structure, but with different components. More importantly, they are motivated by several design choices. We describe the former through the latter.

Message-update Note that the functions F, A, M defined in Eq. 2.17, 2.16 are independent of the iteration. This makes sense when viewing a GNN as learning a message-passing algorithm - these are usually run an unspecified number of iterations, "until convergence". Indeed, in the original GNN paper, the function on the entire graph had to be a contraction in order to assure convergence. But when running a fixed number of iterations, it is easy to relax this constraint and define a different message-update per iteration. Technically, in our (CLIG) implementation from Eq. 2.22, 2.21, it means changing the matrices W_{lc}, W_{cl} to be parametrized by iteration, $W_{lc}^{(t)}, W_{cl}^{(t)}$. This makes the network more expressive, at the usual cost of more weights, computation time, etc'.

The messages in Eq. 2.22, 2.21 and as shown in Fig. 2.15 are represented as a 1 layer MLP for simplicity. In practice they can be efficiently implemented by deeper networks. This

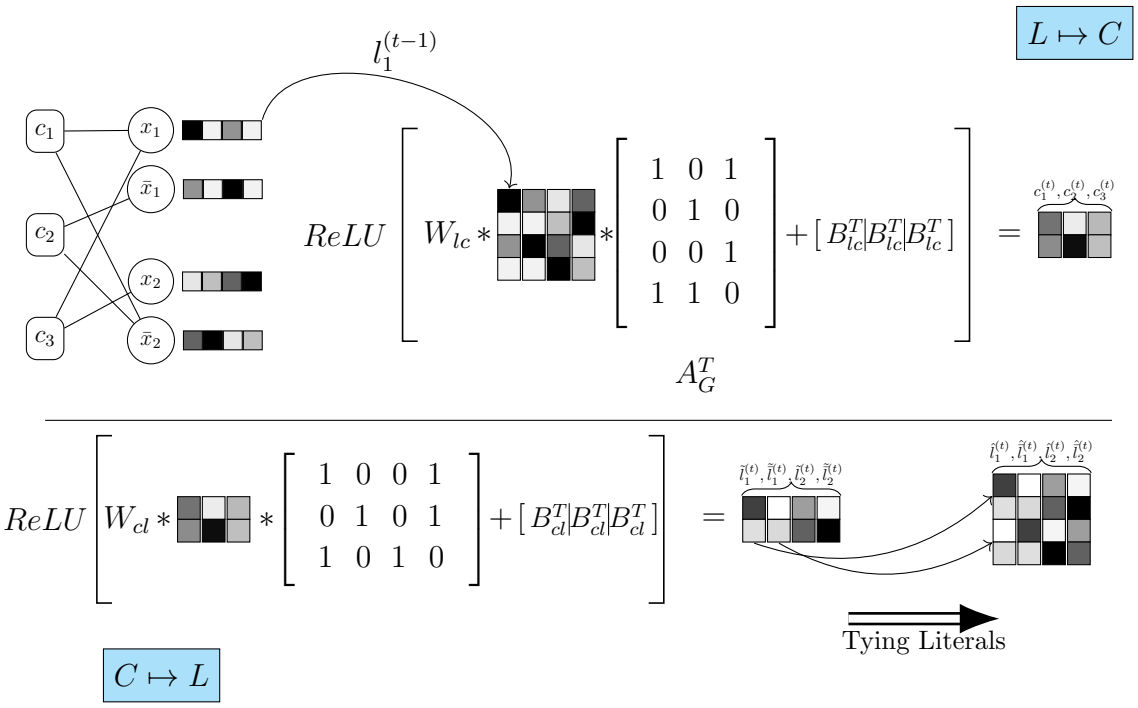


Figure 2.15: Efficient implementation of CLIG half-iterations. Dimensions in the figure are: $d_c = 2, d_v = 2, n = 2, m = 3$.

too increases the expressiveness of the GNN at the cost of more weights. The total number of parameters in the message-update is therefore proportional to both the number of parameters the message-update and to the number of iterations.

Propagation When using several iterations in a GNN, there's the issue of information propagation through the network to think of. Let us denote kth (out of τ) iteration embedding of variables as $V^{(k)}$, and the iteration itself (both halves) denoted $\mathcal{I}$ s.t $V^{(k+1)} = \mathcal{I}(V^{(k)}) = \mathcal{I}^{k+1}(V^{(0)})$. For example, if $\tau = 2$, it is possible that some information about a variable that can be gathered from its 1-hop neighbors (thus appearing already in $V^{(1)}$) is important for the downstream task that only sees $V^{(2)}$. In theory, if $V^{(1)}$ contains an important feature, the next propagation should be able to preserve it. In practice, its been shown that NN have issues with implementing an identity function, therefore just "passing" information [50]. This has been addressed in the literature with either residual connections (basically, redefining a network component representation from $y = f(x)$ to $y = x + f(x)$), or gating mechanisms. We've tested both residual and GRU cells. We can also, instead of trying to preserve and propagate the information, simply include it. One approach is to concatenate all the intermediate embeddings of a variable as the final embedding we pass to the downstream task. So in a 2 iteration GNN, the final embeddings would be $[V^{(0)}, V^{(1)}, V^{(2)}]$. However, this way only the

downstream task has access to intermediate embeddings. Its also possible for each iteration to have access to all the previous intermediate embeddings, by defining different message updates for different iterations as described above.

2.4 Empirical Results

Our ultimate goal is to use our CNF formulas representation within a CS. However, before we turn to that, we've implemented the architecture described above, and run several experiments to test its different variations. The results we detail here do not fall under what is usually termed "experiments" in machine learning literature. The main difference is that we do not include comparison to any other baselines on the tasks we describe.

The main reason we do not (yet) compare to any baseline is our motivation. We are interested in measurable improvements to specific CS, on real-world tasks, what is referred to as "industrial problems" in the Formal Methods literature [6]. The ultimate usefulness of a representation is derived backwards from its contribution to that real-world task. "Being good in representing logical formulas" is not a meaningful, measurable property of an architecture independent of what its used for.[18] And so, there is no natural baseline to compare to.

Our motivation in this section is rather to develop an understanding the properties of the architecture, and its behaviour under different hyper-parameters. We tested on two downstream tasks that, if not of practical use in themselves, would seem to require the ability to represent CNF graphs.

Equivalence Experiments

The first downstream task is taken directly from the work we described in Sec. 2.3. It is to classify a set of Boolean formulas over some named variables (also called "grounded" variables, or input variables) into their equivalence class.

Data

We start with the BOOLEAN8 dataset from Allamanis et al. [3]. It contains formulas over 3 variables, $\{a, b, c\}$, which fall into one of 193 (out of 265 possible) equivalence classes. There is a total of 146488 formulas in the training set, unequally divided between the classes. The largest class ('True') has over 17k formulas, the smallest, only 15. We filter the training set to include only classes with a minimal number of representatives. We bias the sampling during training so that the model is equally probable to see a sample from any given equivalence class. There are also validation and test sets, of 36508 and 60929 formulas, respectively.

Each formula is composed of the 3 variables and the operators (**And, Or, Not, Xor, Implies**). For each formula, we transform it into CNF using the Tseitin transformation [145]. This

[18] At least not yet. When there will be enough downstream tasks that involve processing formulas, a meaningful measure of how good a "general representation" could be developed, just like for natural language.

introduces new tseitin variables into the formula, so we note which variables are the grounded ones. We also note the topmost tseitin variable, which evaluates to the formula.

Implementation Specifics

There are a still few missing pieces to the implementation, as described in Sec 2.3. First, note that while the encoder model goes from the i'th embedding to the $i+1$'th one, it doesn't include the initial, 0'th iteration embeddings, which we call "ground embeddings". These are set according to the domain in which the GNN is used. What should the ground embeddings be in our case? Surely they must allow the model to distinguish between the ground variables. after all a is not equivalent to b, and for the model to have any hope of classifying them, they must seem different to it. The tseitin variables, however, do not have a fixed name across formulas, they are created as necessary, and are therefore indistinguishable except by the topology of their neighborhood. We therefore denote a, b, c with id's $1, 2, 3$, and all other tseitin variables with id 4, making them interchangeable. We represent their id's in one-hot encoding as $l^{(0)}$.

Second, note that a GNN produces an embedding for each of its nodes. For CNF graphs, we produce an embedding per variable (or literal), and every formula can have a different number of variables. The downstream task though is to predict one (equivalence) class for the entire graph. How do we go from n embeddings to predicting one class?

Transforming a formula to CNF creates additional tseitin variables, which are all in fact a function of the ground variables(a, b, c). One of those additional variables is the "top level" variable, meaning, as a function of the ground variables, it is of the same equivalence class as the entire original formula. And so, one approach to classifying the graph is to focus only on the final embedding of the top level variable, which at least in theory contains information about the entire original formula. If the variable dimension is d_v and the number of equivalence classes is K, a final layer of $MLP(d_v, K)$ will take the top variable embedding and produce classes logits.

A more general technique can be used whenever we have to reduce a variable number of n d-dimensional embeddings $\{x_i\}_{1 \le i \le n}$ to a single d-dimensional vector. In fact, we note that as part of the GNN computation we already do something similar - we combine any number of a node's neighbors previous embeddings from iteration $t-1$ into its new t'th level embedding. In Li et al. [79] they exploit this fact by adding a new auxiliary node to the graph, that is connected by *directed* edges to every other node (they have to be directed so that the auxiliary node doesn't change the topology of information propagation within the graph, it only receives information). By definition, the embedding of this auxiliary node gathers information from all nodes in the graph, which is a little bit like simulating a "top level variable". Equivalently, we just take the final n variable ($2n$ literal) embeddings and combine them directly. We use a parametrized (i.e, learned) combination of the final n embeddings by implementing a computation somewhat similar to gating mechanisms. If the final embeddings are denoted $\{v_0, \ldots, v_n\}$, we output:

Depth	Accuracy
1	0.76
2	0.885
3	0.889
4	0.891
5	0.901
6	0.918
7	0.915
8	0.92

Table 2.1: Accuracy on 8 classes increases with depth of models

$$V_{aggregate} = \frac{1}{n} \sum_{1 \leq i \leq n} Iv_i * Sigmoid(Jv_i) \tag{2.25}$$

Where I, J are two $d_v \times d_v$ matrices. I transforms the embedding, and J, through a Sigmoid function which maps its input to $(0, 1)$, is used as a point-wise "gating" mechanism. Once we have $V_{aggregate}$, we compute the K logits as before.

Protocol and Results

We started by comparing the different hyperparameters on an easy problem - we filtered the dataset for the 8 largest equivalence classes. We trained using a small embedding dimension of 10. We trained the CVIG model described in 2.3, and found a clear pattern when it came to the number of iterations, or "depth" of the GNN. As can be seen in Table 2.1, increasing the depth of the model improves the overall accuracy. We can also see that the the big gains are when going from 1 to 2 iterations, which is not surprising considering in this dataset, a 2-hop neighborhood is exactly the size of the entire formula. It should be noted that this increased accuracy involves no increase in the number of weights, but does increase both training and inference time. At this point it is not of much interest to us, but inference time will come to play a larger part when we embed a network into an optimized CS. It should also be noted that we found deeper models to be more difficult to train. Training is more prone to "fail" the deeper the model is, as in, getting stuck in some local minimum of the loss function (this can be detected early and aborted, so doesn't cost much computation time). The results in the table are averages over 8 successful runs. The variance in results also increases with number of iterations. This instability makes sense, and had been observed in other deep NN. The parameterized function $\mathcal{I}^8_W$ is more sensitive to W than $\mathcal{I}_W$ is.

We make the task more challenging by increasing the number of classes. The same pattern persists with regards to the number of iterations when we train on 28 classes and 45 classes, but the accuracy drops. The highest accuracy (achieved with 8 iterations) on 28

# Classes	Embedding Dimension	Accuracy
28	20	0.99
45	32	0.989
95	38	0.988
130	44	0.987
164	64	0.965

Table 2.2: Accuracy of 3 iterations model with varying embedding size

and 45 classes, respectively, was 0.56 and 0.24. It turns out that in order to classify formulas into more classes, we need to increase the embedding space dimension. Higher embedding dimensions get better accuracy, until at some point it saturates. In Table. 2.2, we report the accuracy and the embedding dimension required to achieve it for different number of classes. All results are achieved using 3 iterations of the same model. With a modest embedding dimension of 64, we can get 96% accuracy classifying formulas into 164 classes! Following this logic, it seems reasonable that classifying formulas into fewer classes requires less space. Indeed, going the other way, we tested what is the minimal embedding dimension that allows us to classify just two classes. We used the same dataset, but classified all formulas into the class FALSE, and all other classes. Effectively, this means classifying formulas according to their satisfiability. We found that with an embedding dimension of 2, accuracy of 97% was achieved.

In conclusion, the BOOLEAN8 dataset of 3 variable formulas turned out to be quite trivial for a GNN to classify. As detailed earlier, classifying formulas to equivalence classes is not a useful task in itself, and we do not compare to other baselines. We did compare our results against a standard LSTM network, to get an idea of how difficult the task is for a sequential . It should be noted that it is not entirely clear what a "fair" comparison should be in this context - to test our model, we reduce each formula to a CNF graph. This plays to the strengths of a GNN, but should hardly help an LSTM network. Quite the contrary, the CNF representation, when viewed as a sequence, is longer (up to 5x tokens) than the original formula. We ended up testing an LSTM network with cell dimension 128, both on the original formulas tokens and on their equivalent CNF representation. We used 164 classes, and the accuracy achieved was 21% and and 13%, respectively, compared to 96.5% with the GNN.

Random Structured SAT Experiments

Given the results on classifying formulas on 3 variables to their equivalence classes, we decided to test the model on more challenging formulas. We had two issues to consider - first, it is not trivial to generate Boolean formulas over many variables while making sure there are enough samples per equivalence class. Second, the formulas we will eventually want to represent are of a different nature - they do not include any fixed named variables with a meaning that

transcends a single formula. And so with this in mind, we decided to focus on random SAT problems.

Data

We generated larger random formulas (though still toy problems in the context of modern SAT solvers - every one of them was solved by Minisat in under 2ms) using FUZZSAT[19], which generates random Boolean circuits, translates them into CNF, and adds some random clauses for good measure. We generated around 100k random formulas over 4-8 input variables, split evenly into SAT and UNSAT (using rejection sampling) with an average of 33 variables and 96 clauses (compared to an average of 6 and 13 for the equivalence classes classification task). We split them randomly into training, validation and test set (70/15/15).

Implementation Specifics

The problems synthesized by fuzzsat do not have any fixed, named variables, and so in this dataset, there's nothing that distinguishes between variables apart from the topology of their neighborhood (at least in the CNF representation). And so, we define the ground embeddings of all variables to be the same learned parameter vector of size d_v. To go from the per-variable embedding produced by the encoder to the final 2-class decision, we use the same aggregation as in Eq. 2.25.

Several encoder versions give very similar results, but the one we ended up using a variation of the CLIG model with $d := d_v = d_c$:

$$c^{(t)} = M_{lc}\left(c^{(t-1)} + \sum_{l \in c} l^{(t-1)}\right) \tag{2.26}$$

$$\tilde{l}^{(t)} = l^{(t-1)} + \sum_{l \in c} c^{(t)}) \tag{2.27}$$

$$l^{(t)} = M_{cl}\left([\tilde{l}^{(t)}, \tilde{\tilde{l}}^{(t)}]\right) \tag{2.28}$$

Where M_{lc} is a two-layer MLP with layer normalization, specifically, $MLP_{LN}(d, d, d)$. M_{cl} is similar, but for the different first dimension, $MLP_{LN}(2d, d, d)$. Note that for both half-iterations, we aggregate not only the neighbors embedding, but also the embedding of the node itself from the previous iteration. MLP_{LN} is similar to the standard MLP seen before, but with added Layer Normalization [10] after every non-linearity. Specifically, $MLP_{LN}(d, d, d)(x) := W_2(LayerNorm(ReLU(W_1 x + B_1))) + B_2$, where $W_1, W_2 \in \mathbb{R}^{d \times d}, B_1, B_2 \in \mathbb{R}^d$. No GRU was necessary.

Results

[19] http://fmv.jku.at/fuzzsat/

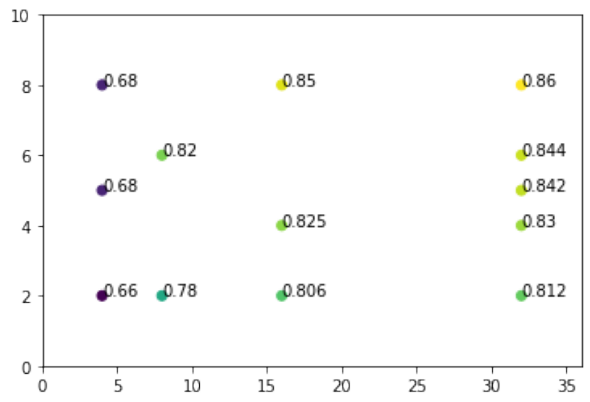


Figure 2.16: Accuracy of classifying SAT/UNSAT on random circuits

We've seen a similar pattern for the SAT classification as we have for the previous experiment with the BOOLEAN8 dataset. As in, more iterations result in better accuracy. To a lesser degree, increased embedding dimension also improves accuracy. As can be seen in the scatter plot in Fig. 2.16, all encoder versions with $d \geq 16$ and number of iterations $\tau \geq 2$ achieved an accuracy of $> 80\%$ on the test set. Consistently, the accuracy was 2-10% higher on the UNSAT instances than on the SAT. The best performance was achieved by training a 8 iterations model with $d = 32$, which achieved a combined accuracy of 86%, with 90% on the UNSAT instances and 84% on the SAT. Number of iterations seemed to be more important than embeddings size. A model with just a single iteration and a (relatively) large embedding size of 128 (not seen in the scatter plot) achieved an accuracy of 72%, whereas a model with 6 iterations and embedding size of merely 8 achieved an accuracy of 82%.

Solving SAT through Message-Passing

NeuroSAT As this was not the main focus of this work, in our SAT classification experiments we didn't go above 8 iterations of the encoder, nor tested other, more challenging datasets. In contemporary work[20], using a similar GNN architecture called NEUROSAT [121], the authors have thoroughly investigated the problem of classifying SAT/UNSAT problem instances on several synthesized datasets. They trained on a cleverly crafted dataset they call $SR(n)$, which contains randomly generated *pairs* of problems, where each pair includes to problems, one SAT, one UNSAT, and the difference between them is, by design, the polarity of a single variable in a single clause. The intuition behind this dataset is that the pair constructed this almost syntactically indistinguishable, making the task more challenging for the network. They demonstrated several properties of such a classification task, which shed some more light on how a GNN based model might be solving it:

- Instead of aggregating the topmost embedding of the literals and then deriving a decision from it, they first derived a single scalar per literal, which can then be interpreted as that literal's "vote". They then average the votes to get a global decision.
- They showed a qualitative difference between embeddings in SAT and UNSAT instances. Essentially, on SAT instances, the literals eventually settle into a confident decision after

[20] At the time of writing

some iterations. On UNSAT instances, literals remain perplexed, not very confident in their (negative) decision, regardless of how many iterations are run. Moreover, in SAT instances, oftentimes a satisfying assignment can be decoded from the embeddings of the literals, as it turns out the topmost literal embeddings are clustered according to *some* satisfying assignment.

- They used a larger model, larger dataset (with more variables and clauses, 40/200 on average), and more GNN iterations during training, 26, to be exact. Moreover, they showed that once trained, the network can solve larger problems, up to $SR(200)$ (albeit with a lower accuracy of about 25%), by simply increasing the number of iterations, up to hundreds of them!

- They showed the same model trained on $SR(40)$ can then solve problems from different synthesized families, such as k-coloring, clique detection and vertex cover of random graphs from several distributions. Those problems were up to 2.5 larger than those the model was trained on.

It is interesting to note that our model achieved an accuracy on the fuzzsat-generated dataset that is on the same order of magnitude as NeuroSAT achieved on the $SR(40)$ dataset, yet with fewer iterations (8 compared to 26), and a smaller embedding dimension (32 vs 128). We believe this is due to the difference in the nature of the datasets - $SR(40)$ has little internal regularity in its structure - they generate it by manually sampling clauses of some (random, smallish) size k, each clause being entirely independent of the others. Our fuzzsat dataset, on the other hand, is generated as a random Boolean circuit, which is then reduced to CNF through the Tseitin transformation. The result is quite different, and much more regular. There is a small finite number of Boolean gates types, and each gate type is transformed into an identical set of clauses (on different literals, obviously). The result is anything but independent clauses - all problem graphs in our dataset contain the same small isomorphic subgraphs that originate from the Boolean gates.

SAT Instances as Factor Graphs What exactly is going on here? How might a GNN classify SAT/UNSAT problems, or in the case of NeuroSAT, even come up with a satisfying assignment as a certificate? There are usually no mathematical proofs when dealing with NN, but we might set forth a conjecture that explains the rough outlines of how this might be done by a GNN.

What we called the Clause-Variable Incidence Graph (see Fig .2.11a) has another, more general name - A *Factor Graph*. Factor graphs [74] are graphical models that represent multi-variable functions which can be factored, such as (see Fig. 2.17):

$$g(x_1, x_2, x_3, x_4, x_5) = f_A(x_1) f_B(x_2) f_C(x_1, x_2, x_3) f_D(x_3, x_4) f_E(x_4, x_5)$$

The function g represented by factor graphs is usually of one of two types - either a set-membership indicator function, or a probability function. The factor graph of a satisfiable

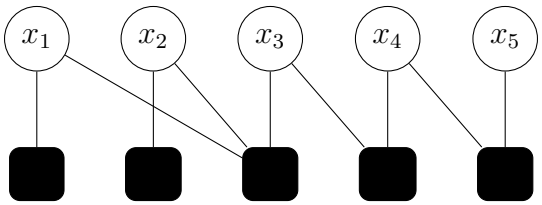


Figure 2.17: Factor graph of $f_A(x_1)f_B(x_2)f_C(x_1, x_2, x_3)f_D(x_3, x_4)f_E(x_4, x_5)$

SAT instance represents both functions, which are the same up to a scalar. To see that, we add some new definitions related to a SAT problem in CNF. Following Maneva, Mossel, and Wainwright [86], let V, C represent sets of indices of variables and clauses, such that $|V| = n, |C| = m$. We will use i, j, k to denote variable indices, and a, b, c for clause indices. We abuse notation and use i or a as the variables and clauses themselves when clear from context. For a set of indices S, $x_S := \{x_i | i \in S\}$. Now, the clause indexed by $a \in C$ is defined by a tuple $(V(a), J_a)$. $V(a) \subseteq V$ is a set of k variable indices. $J_a := (J_{a,i} | i \in V(a)$, where $J_{a,i} \in \{0, 1\}$ represents the polarity of the variable i in a. It is 0 for positive polarity of i in a, 1 if negative. The clause indexed by a is satisfied if and only if $x_{V(a)} \neq J_a$ (note, $V(a)$ is a set of variable indices. $x_{V(a)}$ is a k-tuple denoting the an actual partial assignment to variables in the a). Let $\delta(x, y)$ be the indicator function for the event $\{x = y\}$, and now we can define the function:

$$\psi_{J_a}(x) := 1 - \prod_{i \in V(a)} \delta(J_{a,i}, x_i) \tag{2.29}$$

And it is clear that for an assignment x, a is satisfied if and only if $\psi_{J_a} = 1$. The function $g(x) = \prod_{a \in C} \psi_{J_a}(x)$ can now be seen as the indicator function of the SAT problem defined by V, C. It can be transformed into a probability function by scaling:

$$p(x) := \frac{1}{Z} \prod_{a \in C} \psi_{J_a}(x) \tag{2.30}$$

Where $Z = \sum_{x \in \{0,1\}^n} \prod_{a \in C} \psi_{J_a}(x)$ is the normalization factor, sometimes called *Partition Function*. This definition only makes sense for satisfiable instances, and in that case it has a clear interpretation - It is the uniform distribution over the satisfying assignments. Tt assigns zero to every non-satisfying assignment, and $\frac{1}{Z}$ to every satisfying assignment, where Z is the total number of solutions.

One common computational task in factor graphs is inference of (conditional) *marginal probabilities*. That is, given a factored joint probability function $p(x)$ on variables $x = (x_1, \ldots, x_n)$ and two disjoint subsets of indices $S_1, S_2 \subset \{1, \ldots, n\}$, compute $p(x_{S_1} | x_{S_2})$. (a common case is where x_{S_2} are observed variables and x_{S_1} some Latent variables). In the context of a SAT instance, we are interested in the unconditional marginal probabilities $p(x_i)$. If we had an oracle that can compute the exact marginals, finding a satisfying assignment becomes trivial. $p(x_1 = 0)$ is the fraction of satisfying assignments in which $x_1 = 0$. Since

$p(x_1 = 0) + p(x_1 = 1) = 1$, one of the two assignments to x_1 has a probability greater than zero, and so can be completed to at least one full assignment that satisfies the formula. So we assign it, and solve the residual formula, which is also guaranteed to be SAT, using the same process. This iterative technique is called "decimation".

However, computing the exact marginals is intractable for most graphs. There are message-passing algorithms on factor graphs that approximate the marginals, such as the Sum-Product algorithm, also known as Belief Propagation [153] (BP). It is exact on trees, and while not guaranteed to converge on graphs with loops, in practice tends to work surprisingly well [96], and there are a few theoretical explanations as to why that is the case [40]. In the case of SAT instances, computing (approximate) marginals is known also as the Belief Propagation (BP) algorithm. Another algorithm on factor graphs which can be implemented as message passing is the Min-Sum algorithm, which approximates the mode of a function. In the context of SAT, it is called Warning Propagation (WP), which, if it converges, produces a satisfying solution. Another message-passing algorithm which approximates marginals is Survey Propagation [27, 86] (SP), which can be seen as BP on different probability functions.[21] Both BP and SP can be used in decimation loops to arrive at a satisfying assignment.

Could a GNN be approximating such a message-passing algorithm that can compute marginals or mode for some given distribution of factor graphs? Is it enough? In Dai, Dai, and Song [35], the authors offer an interpretation of GNN iterations as doing just that. Based on some rather fancy math [129], it can be shown that probability distributions can be embedded into some (possibly infinite) Hilbert Space, and that the message-updates of those distributions can be embedded as operations in said space. Specifically, they show that A GNN can compute marginals, learning to represent the probability distributions sent as the messages in the graph, and the message-update transformation, based on the distribution of the data its trained on, plausibly approximating BP by performing computations in a finite dimension embedding space. And so, we conjecture that classifying and solving SAT/UNSAT problems can be seen as a special case of Dai, Dai, and Song [35] in the domain of CNF factor graphs.

[21] It is a correction to the assumptions of BP, based on the the structure of the solution space of random $3 - SAT$ problems past some clause to variable threshold, which tends to fracture into distant clusters, therefore eluding local search methods.

Chapter 3

Learning Branching Heuristics for QBF

Having developed A neural representation for CNF formulas, we now turn to integrating it within a modern Constraint Solver (CS). We begin by describing our domain - we go over the operation of a basic *Conflict-Driven Clause Learning* (CDCL) SAT solver and its related Heuristics. Next we discuss the considerations underlying the interfacing of CS and NN - What other approaches were attempted, what are our goals, and how to measure them. We make the case that the process of solving a CS problem can be mapped to the *Reinforcement Learning* settings, and that the 2-QBF solver CADET [105] is a good candidate for such a method. We briefly discuss how CADET works as a CDCL based CS, and then present an implementation and experimental results of a version of cadet which incorporates a learning component. We conclude with a discussion of the results.

3.1 Anatomy of a SAT Solver

The satisfiability problem of propositional Boolean logics (SAT) is to find a satisfying assignment for a given Boolean formula or to determine that there is no such assignment. SAT is the canonical NP-complete problem and many other problems in NP can be easily reduced to it. There are no efficient algorithms for solving SAT of course, but modern SAT solvers are quite good at solving large, practical industrial problems. We will introduce the concepts of DPLL, Clause Learning and their related Heuristics in the context of the SAT problem, that in which they appeared.

DPLL

Modern SAT solvers such as MINISAT [39] and GLUCOSE [8] are able to solve instances of industrial problems that are of up to millions of variables/clauses. Perhaps surprisingly, the core of these modern tools is still the 1962' Davis–Putnam–Logemann–Loveland (DPLL)

algorithm [36], a complete, backtracking search algorithm that either gives a satisfying assignment or proves UNSAT.

First, let us consider the effect of assigning a variable on the individual clauses in a CNF problem. Formally, let $\Phi(x)$ be a formula over $x = (x_1, \ldots, x_n)$. We denote $\Phi_{x_1=1}$ the formula over $(x_2, \ldots, x_n)$ resulting from assigning $x_1 = 1$. To go from the CNF representation of Φ to that of $\Phi_{x_1=1}$, we go over the clauses. The ones that do not contain x_1 at all, we leave unchanged. If a clause contains x_1, we can safely remove it, because x_1 already satisfies it. If a clause contains $\bar{x}_i$, we remove the literal $\bar{x}_i$ from the clause, because for any Boolean formula A, $A \vee 0 \equiv A$.

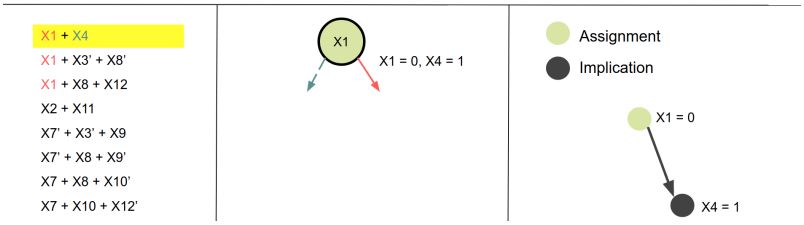
At the heart of DPLL is a 'naive' backtracking search algorithm over a binary tree. It **chooses** a variable and assignment, such as $x_1 = 1$, and updates the clauses accordingly as described above. This is also called *Branching* on a variable, or a branching decision. We continue to branch, working our way down the tree, until one of two things happen. If we assign all variables successfully, we end up with no clauses left, and have found a satisfying assignment. If when updating the clauses after setting $x_i = 1$ we find ourselves with an empty clause, that is a *conflict*. We backtrack and set $x_i = 0$. If that also leads to a conflict, we backtrack one step further. We continue until we find an assignment or finish searching the tree. DPLL takes this basic search algorithm and enhances it with two steps after each branching decision, meant to propagate entailment relations between the variables:

- **Unit Propagation** (UP) is performed when we have *unit clauses*, clauses with a single literal. A unit clause essentially forces an assignment. UP is the process of doing these forced assignments (an assignment can create new unit clauses which continue the propagation). Iterated application of UP is called Boolean Constraint Propagation (BCP).
- **Pure literal elimination** is a step that assigns any variable that happens to appear in the formula always in the same polarity, by deleting the clauses it appears in.

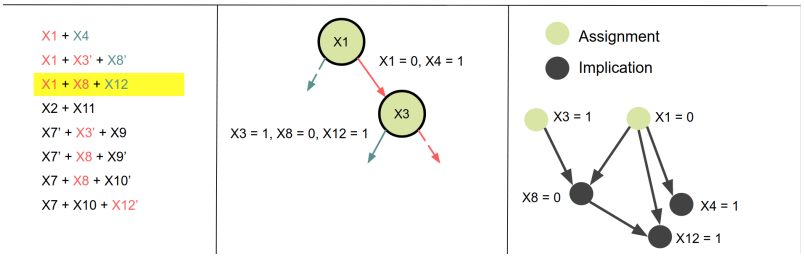
Its easier to understand DPLL by example. Fig. 3.1 shows DPLL running on a small CNF problem. Note the change in notation of clauses, where negation is denoted with ‘ and logical or with +.

CDCL

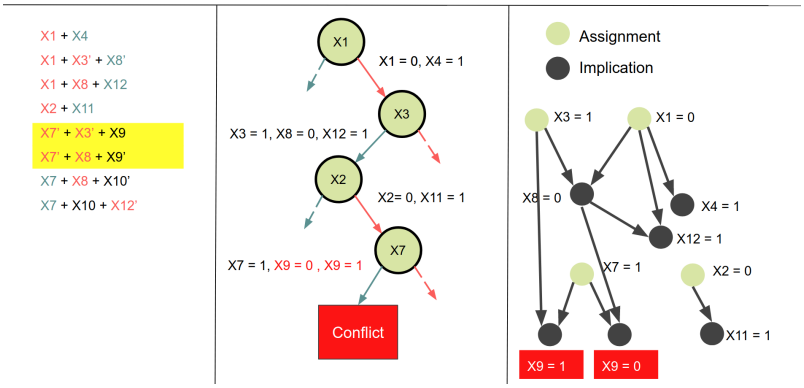
In 1997, GRASP [88] introduced the idea of Clause Learning, which leveraged conflicts to learn new clauses through conflict analysis. The intuition behind it is to find the cause of the conflict, and add a new clause that represents the conflict and prevents further searching. And so Conflict analysis adds new clauses over time, which cuts off large parts of the search space and thereby speeds up the search process. Continuing with the previous example from where a conflict was detected, Fig. 3.2 shows the steps of analysing the conflict, learning a clause, and backtracking in CDCL.



(a) Step 1, we branch on $x_1 = 0$. On the left pane, we mark x_1 red. This makes the clause $x_1 + x_4$ a unit clause, and so through UP we set $x_4 = 1$ and add it to the implication graph.

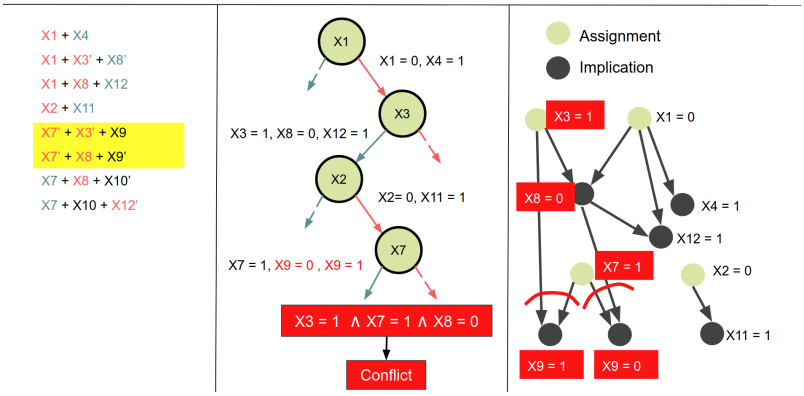


(b) Branching on $x_3 = 1$ forces $x_8 = 0$ through the clause $x_1 + x_3' + x_8'$, then $x_{12} = 1$ through the clause $x_1 + x_8 + x_{12}$.

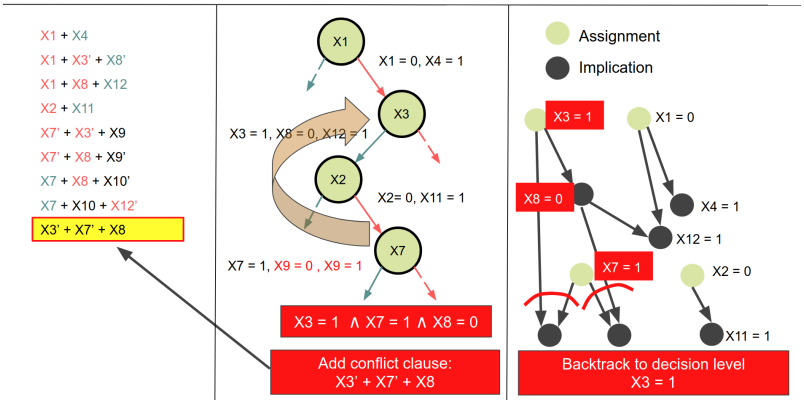


(c) After two more branching decisions, $x_2 = 0$ and then $x_7 = 1$, we reach a conflict on x_9. At this point, DPLL backtracks, sets $x_7 = 0$, and continues as before.

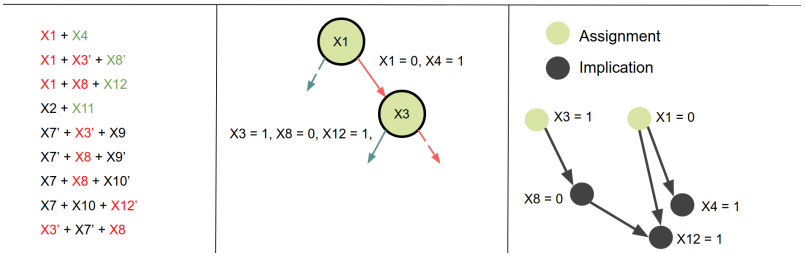
Figure 3.1: States of DPLL solving a formula. In each state there are 3 panes. On the left are the clauses, with literals colored according to their *satisfaction*. Green satisfies the clause, Red constrains it, and Black is not yet set. The green and red literals are "deleted" from their clauses. The middle pane shows the *Decision Graph*. This is effectively the search tree, and it shows what variables are set, by branch or propagation. In the right pane is the *Implication Graph*. It keeps track of implications, the results of UP.



(a) Conflict Analysis finds the literal assignments involved in the two clauses that caused the conflict, $x_3 = 1, x_7 = 1, x_8 = 0$.



(b) Negating the 3 assignments that led to the conflict produces the learned clause, which we add to the problem clauses.



(c) After learning the clause, CDCL backtracks to the earliest decision variable involved in the conflict, in this case, x_3.

Figure 3.2: CDCL diverges from the DPLL algorithm when a conflict is detected. Rather than just backtracking, it analyses the conflict, learns a clause, and backtracks non-chronologically, to the earliest decision variable in the learned clause.

Since its introduction, countless refinements of CDCL have been explored and clever data structures improved its efficiency significantly [93, 39, 46]. Today, the top-performing SAT solvers, such as Lingeling [20], Crypominisat [133], Glucose [8], and MapleSAT [80], all rely on CDCL and they solve formulas with millions of variables for industrial applications such as bounded model checking [22].

SAT Branching Heuristic

Solvers of SAT (and other constraint problems) are non-deterministic. At run-time, solvers employ specially crafted Heuristics to make choices, and different heuristics can have drastically different run-times. The most important heuristic in a SAT solver is the Branching Heuristic, choosing the next variable to branch on [67]. A number of branching heuristics have been developed over the years, such as DLIS, MOM, Jeroslow-Wang (JW) (see [87] for details). The intuition behind most of these heuristics is to (greedily) branch on literals that appear in or satisfy many clauses, preferably small ones. However, beginning with CHAFF [93], the preferred branching heuristic became the *Variable State Independent Decaying Sum* (VSIDS), which is highly coupled with the occurrence of conflicts, and the CDCL algorithm.

VSIDS gives an initial score to each variable (or, originally in Chaff, to each literal) according to the number of clauses it appears in, and then periodically scales the scores of all variables by a constant smaller than 1, making them decay exponentially to 0. For every clause learned (the result of a conflict), VSIDS increments the score of the variables in the new clause. The end result is a heuristic that concentrates on variables that took part in **recent** conflicts. Despite being a highly successful heuristic for solving industrial problems (unlike k-CNF problems for example, for which it adds very little), after 20 years there is still no definitive explanation for its success. There are a few theories backed by empirical evidence [56], mostly attributing the success of CDCL+VSIDS on industrial instances to the special structure of such problems - specifically, their "community structure" [7, 95] (A technical definition to do with the distribution of vertex degrees in the graph).

3.2 Combining Learning and Symbolic Reasoning

The question of whether (deep) learning can be effectively used for symbolic reasoning as in CS is intriguing for both theoretical and practical problems, and the benefits of combining deductive reasoning with inductive learning for automated reasoning and in formal methods for system design have been noted before (e.g., see [123]). In this section we present previous approaches that tackled this question, and discuss our goals and guiding motivation in our.

Existing Approaches

There exists an entire spectrum of approaches aiming to combine learning with CS, with different levels of integration: One extreme is to use learning for predicting which of a small

pool of algorithms (or heuristics) performs best, and run only that one to solve the given problem (e.g. SATzilla [158]). There's very little integration between the learning component and the actual solver in this case - they work separately. This approach is effective, but clearly limited by the availability of handwritten algorithms and heuristics (i.e. it can only solve problems for which we have written at least one algorithm that can solve it, based on expertise of human experts). On the other extreme, there are solutions where the classical CS side is eliminated entirely, and formulas are analyzed solely with deep learning architectures [3, 41, 121, 5]. While these approaches are intriguing from the ML perspective, even the best results are orders of magnitude below the state-of-the-art in the respective domains, despite the recent breakthroughs in deep learning. It is highly unlikely they will scale on their own.

There are a multitude of approaches that can work in different contexts. For example, if a good but expensive heuristic is known, such as in Mixed Integer Programming (MIP) problems, it is possible to learn a cheap approximation of it, as in [69], thereby speeding up the solver. There are works that enhance the VSIDS heuristic by using learning within a single run of the solver on a single formula, by means of optimizing a proxy [80] measure called Learning Rate (not to be confused with learning rate in SGD), the propensity of a variable to generate learned clauses. Other works [134] use supervised learning to approximate several proxies that should be important for the solving process, such as "clause usefulness". The downside of these approaches is that it is not entirely clear whether the proxies are indeed good enough. Of course, the more evidence there is for the usefulness of a proxy the better, but the fact is that for SAT, and more so for more complex CS, no such perfect proxy is known. Learning more clauses can be good, or bad, depending on the learned clauses, the formula, the solver state, and even other heuristics in the solver such as branching or restart policy. And so training a NN to approximate some manually designed proxy is not very different from manually designing a heuristic. See more on related work in Sec. 3.6.

Goals and Motivation

Instead of relying entirely on deep learning or on the availability of good handwritten algorithms, we explore the middle ground. We argue for tighter integration between deep learning and formal reasoning algorithms, which has the potential to both be unlimited by available manually designed heuristics, yet potentially scalable to real industrial problems. In the Boolean CS we're interested in, heuristics are generally fast, in fact, orders of magnitude faster beyond what a NN can achieve. That means that if we want to solve harder formulas, we need a smarter heuristic, not a faster approximation of an existing one. For that reason, our approach is intuitively more suited to complex solvers, where the branching heuristics cumulative running time is relatively a small part of the entire solving process.

We aim to avoid proxies, and concentrate on optimizing, directly, two goals, in decreasing order of importance: 1. Solve the problem. 2. Solve it fast. These goals are in fact derived from how solvers are tested in practice. Because solvers are complete, given enough time, any problem will be solved. Since we don't have unbounded time to run on every problem, in practice solvers are tested on a set of benchmark problems, where every problem is attempted

up to some bound, and then aborted and considered as failed to be solved. Judging the performance of solvers is therefore reduced to a combination of how many problems were solved within the bound, and the average time it took to solve them. And, of course, on the choice of the benchmark problems.

Speaking of sets of benchmark problems brings up another mismatch between performance measurements in the fields of ML and Formal Methods, which we have to address. In ML, the **distribution** of the data is explicitly considered, even "baked into" the formalism. For example, in supervised learning, We have a training, validation, and test set. All three sets are assumed to be drawn from the same data distribution. A trained model is said to have achieved "generalization" if it achieves an accuracy on the test set that is close to that of the training set, or, conversely, said to be "overfitting" if it achieves high accuracy on the training set but considerably lower accuracy on the unseen test set (The validation set is supposed to alert us to such overfitting, and there are techniques to deal with it, e.g, regularization). Generally speaking, if the input data is drawn from a different distribution, all bets are off.

With CS solvers, the distribution of the data isn't so neatly and explicitly defined, at least not in the context that is of interest to us. In the case of combinatorial problems such as random k-CNF on n variables and m clauses, there is indeed an explicit data distribution, where a problem instance is a random graph, where each of the m clauses, over k literals, is drawn uniformly from the $\binom{n}{k}$ options. But we're interested in industrial problems, and things there are not as clear. Solvers are often used as black-box tools, and are expected to work well on "all" industrial problems. Benchmarks are composed of CNF formulas that come from several different families of problems suggested by members of the community [51, 52, 104], each with their own encoding. They are either generated by some random process (for example, when trying to find sha-1 preimages, a pre-image is randomly generated and then hashed), or sometimes actual problems, such as those that arise in hardware verification. At any case, unlike in ML, Solvers and their heuristics are not generally "trained" to fit a specific input distribution, and are in fact entirely agnostic with respect to the distribution of their input, at least explicitly.

In theory, its quite possible there exists a superior branching heuristics that dominates VSIDS on every problem instance, which can be arrived at through training. In practice though, it is not at all clear what training set is representative of "all industrial problems", and so throughout this work, we explicitly limit ourselves to learning heuristics for specific families of problem distributions. We roughly follow the "no free lunch" intuition [1] (the formal theorems themselves are not applicable to our case, but we feel the intuition is still a useful illustration). No single heuristic is going to dominate in solving all instances of an NP-hard problem. The more we constrain the problem space, the better specific heuristic can be learned. Arguably, VSIDS already constrains the problem space to industrial problems, so our intuition can be thought of as taking this trend further.

That said, when it comes to CS, it is not very useful to train on some problem instances drawn at random from distribution $\mathcal{D}$ only to be able to solve some other problems from $\mathcal{D}$. The normal ML notion of generalization doesn't help us here. Informally, we want to be able to train on "easy" problems, and learn something that helps us solve "hard" problems that

are similar to the easy ones, but, well, harder. We will make this concept more concrete in the experimental section.

3.3 Method

In this section we describe how we cast the problem of learning better branching heuristics for backtracking search algorithms as a *Reinforcement Learning* (RL) problem, a family of learning algorithms that have demonstrated great success lately in learning to play combinatorial games such as Chess and Go [127]. We start with some background on how RL works, then show how to map our problem to the RL framework, and discuss the challenges that arise of it.

Constraint Solvers as RL Problems

RL

Reinforcement Learning is a more general optimization procedure than the supervised learning we've seen before. Like NN, RL was also inspired by biological systems, but at the level of the organism (rather then neural networks in the brain). It assumes an **Agent** taking actions within an **Environment**. For each action the agent takes it observes how the action effected the environment, and it receives a **Reward**. This process repeats in a loop (see Fig. 3.3), either a finite or infinite number of steps, which produces an *episode*. We can turn finite episodes in an MDP to infinite ones by adding a a dead-end node that loops to itself, so sometimes (for mathematical convenience) we choose to assume all episodes are infinite. The agent is is trying, through its actions, to maximize the sum of rewards it gets from the environment over the episode.

More formally, we consider an environment $\mathcal{E}$ which is modeled as a Markov Decision Process (MDP) over discrete time steps and accumulates reward. An MDP is a 5-tuple of states S, action space A, transition probabilities function $P : S \times A \to \mathcal{P}(S)$, reward function $R : S \times A \times S \to \mathbb{R}$, and initial state distribution $\rho_0 \in \mathcal{P}(S)$. The agent interacts with the environment by implementing a function that gets an *observation* from the environment, and returns an action. In our context, the observation is simply the state, and we use O_t, S_t interchangeably.[1] This function is called a (stochastic) *policy* - A mapping from observations to probability distributions over the actions $\pi : S \to \mathcal{P}(A)$.

The combined evolution of the policy and the environment throughout the episode produces a trajectory,formally a sequence of states and actions $\tau = (s_0, a_0, s_1, a_1, \dots)$. The reward of a trajectory is the sum of rewards over the timesteps, $R(\tau) := \sum_t r_t$. The trajectory τ itself can be seen as a random variable over the randomness of both the policy and the

[1] The terminology comes from Partially-Observed MDP, and is commonly used for MDPs as well in the RL literature

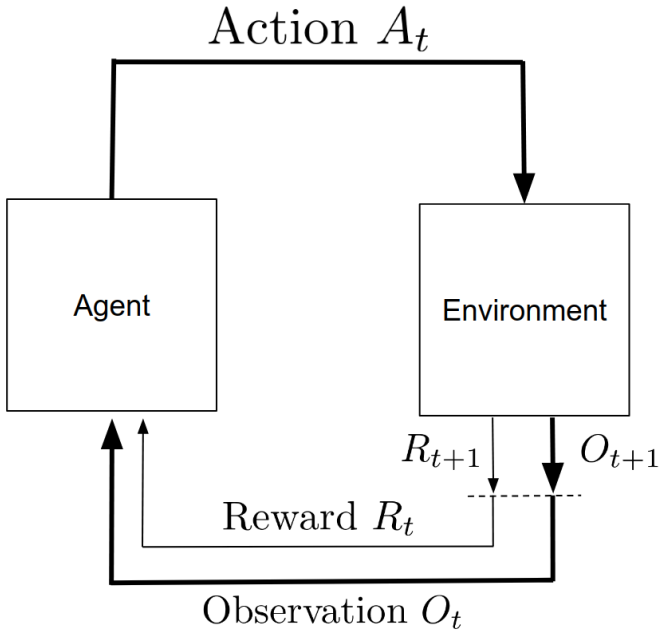


Figure 3.3: The RL loop. At time step t, the agent gets from the environment an observation O_t and a reward R_t. It produces an action A_t.

environment. Assuming its length is T, its probability is:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t) \tag{3.1}$$

The expected return of a policy is defined as $J(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)]$, and the goal of RL algorithms is to find the optimal policy, $\pi^* = \mathop{\text{argmax}}_{\pi} J(\pi)$.

RL Algorithms

We will only go over the very basics of the RL algorithms we use, and direct the interested reader to [137] for details. In order to understand the intuition behind RL, we need to go over a few more related concepts. The **Value Function** of a policy π assigns a score to a state s, its "value", which is the expected cumulative reward the agent gets from following π from s:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{3.2}$$

The closely related **Action-Value** function gives a value to each state-action tuple, which is the expected return when starting in state s with action a, and continuing according to π:

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \tag{3.3}$$

There are also the optimal versions of the value and value-action functions, which simply assume we start from a given state s (or a tuple (s, a)), and continue according to the optimal policy:

$$V^*(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi} [R(\tau)|s_0 = s] \tag{3.4}$$

$$Q^*(s, a) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi} [R(\tau)|s_0 = s, a_0 = a] \tag{3.5}$$

We will be using model-free RL, in which the algorithm doesn't have to know the transition function/distribution of the MDP (Although, a CS has a known, deterministic model. We will say more of this at the end of the chapter). There are in general two flavours of RL algorithms in this space, Q-Learning (QL) and Policy Gradient (PG). QL algorithms try to learn a *Q-function* (another name for the value-action function) that solves the γ-discounted Bellman equation for the optimal policy, which is:

$$V^*(s) = \max_{a} \mathop{\mathbb{E}}_{s'\sim P} [R(s, a, s') + \gamma V^*(s')] \tag{3.6}$$

$$Q^*(s, a) = \mathop{\mathbb{E}}_{s'\sim P} \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right] \tag{3.7}$$

Once we have the optimal Q function, extracting the optimal policy from it is easy - for each state s we simply choose the action that maximizes Q^*: $\pi^*(s) = \max_a Q^*(s, a)$.

Policy Gradient algorithms work differently, by directly optimizing a parametrized policy π_θ. They do this by approximating the gradient of the expected return $\frac{dJ(\pi_\theta)}{d\theta}$, and then proceeding with gradient ascent. Approximating the gradient is done with the "log-derivative trick", as follows:

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) = & \\
&= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) \\
&= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) \\
&= \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) \quad \text{Log-derivative trick} \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} [\nabla_\theta \log P(\tau|\theta) R(\tau)] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]
\end{aligned} \tag{3.8}$$

With this last expression it is straight-forward to estimate the gradient. All we have to do is sample a bunch of trajectories from the current policy π_θ, and average their gradients. Note, that in both algorithms the transition probabilities of the environment itself are either cancelled out or not used to begin with.

Problem Definition

Mapping our domain to the RL framework is straightforward on the one hand, yet a bit tricky to formalize:

- Our state space S is the infinite set of all potential partial states of a solver when working on any possible CNF formula.
- Formally speaking, our action space A is also infinite. For every given state, the agent, which plays the part of the branching heuristics, has to pick one variable (and polarity) from the unassigned variables. It can be seen as a disjoint union of $\biguplus_{s \in S} A_s$, where A_s is the discrete set of unassigned variables in state s.
- Since a CS is deterministic, the transition function is deterministic, and follows the rules of the specific solver. The only random part is the initial state, which is the choice of the formula. During training, this will be uniformly sampled from the training set.
- We use a reward function that resembles that of a maze-solving puzzle. We give the agent a small negative reward for every branching decision it takes, and a large positive reward on the last step for "winning". If the formula wasn't solved by some pre-defined maximum number of decisions we abort it, and it ends up with negative overall returns (the cost of the steps, with no winning reward). This reward structure encourages the agent to solve the formula in as few decisions as possible.

3.4 Implementation

We implement our method using the 2-QBF solver CADET as the environment, and a model based on the GNN architecture described in Sec. 2.3. We start with a short discussion of QBF and Cadet in particular, and then describe our architecture.

QBF & Cadet

QBF extends propositional Boolean logic by *quantifiers*, which are statements of the form "for all x" ($\forall x$) and "there is an x" ($\exists x$). The formula $\forall x.\, \varphi$ is true if, and only if, φ is true if x is replaced by 0 (false) and also if x is replaced by 1 (true). The semantics of $\exists$ arises from $\exists x.\, \varphi = \neg \forall x.\, \neg \varphi$. Whereas SAT is the canonical NP-complete decision problem, QBF[2] is the canonical PSPACE-complete problem, the class of languages that can be decided in polynomial space and unlimited time by a non-deterministic turing machine. Indeed, every SAT problem can be viewed as a QBF of the simplest complexity, by preceding it with an existential quantifier on all variables. We say that a QBF is in prenex normal form if all quantifiers are in the beginning of the formula. WLOG, we will only consider QBF that are

[2]technically, the decision problem is called TQBF, but we use QBF for clarity.

in prenex normal form and whose propositional part is in CNF. Further, we assume that for every variable in the formula there is exactly one quantifier in the prefix. An example QBF in prenex CNF is $\forall x.\, \exists y.\, (x \vee y) \wedge (\neg x \vee y)$.

We focus on 2QBF, a subset of QBF that admits only one quantifier alternation. WLOG we can assume that the quantifier prefix of formulas in 2QBF consists of a sequence of universal quantifiers $\forall x_1 \dots \forall x_n$, followed by a sequence of existential quantifiers $\exists y_1 \dots \exists y_m$. While 2QBF is less powerful than QBF, it is a useful class in itself, and we can encode into it many interesting applications from verification and synthesis, e.g. program synthesis [132, 4].

After the success of CDCL for SAT, CDCL-like algorithms have been explored for QBF as well [45, 82, 105, 107]. We focus on CADET, a solver that implements *Incremental Determinization* a generalized CDCL backtracking search algorithm [105, 107]. Instead of considering only Booleans as values, the Incremental Determinization algorithm assigns and propagates on the level of Skolem functions. We provide more details on Cadet in App. B.1, but for our purposes its enough to consider it as a generalized CDCL algorithm, which branches on variables and learns clauses. The reason why Incremental Determinization is particularly suitable to explore learning approaches is that its individual steps are significantly slower than competing QBF algorithms and that it takes much fewer steps to solve formulas. This is beneficial both due to the running time of NN as explained in Sec. 3.2, and also for optimization reasons - fewer steps means shorter episodes, which in RL means lower variance, and faster training time.

Architecture

Our model gets an observation, consisting of a formula and the state of the solver, and selects one of the formula's literals (= a variable and a Boolean value) as its action. The model has two components: An *encoder* that produces an embedding for every literal, and a *policy network* that rates the quality of each literal based on its embedding. We give an overview of the architecture in Fig. 3.4, describe the encoder in Sec. 3.4 and the policy network in sec. 3.4.

Encoder

For an encoder we used the CLIG implementation of the GNN described in Sec. 2.3. For each variable v, the variable label $\mathbf{v} \in \mathbb{R}^{\lambda_V}$, with $\lambda_V = 7$, indicates whether the variable is universally or existentially quantified, whether it currently has a value assigned, and whether it was selected as a decision variable already on the current search branch. We use the variable label for both of its literals and by $\mathbf{v}_l$ we denote the label of the variable of l. For each clause c, the clause label $\mathbf{c} \in \mathbb{R}$ is a single scalar (in $\{0, 1\}$), indicating whether the clause was original or derived during conflict analysis. See Appendix B.3 for details.

Literal embeddings have dimension $\delta_L = 16$ and clause embeddings have dimension $\delta_C = 64$. The GNN computes the embeddings over τ rounds. We define the initial literal

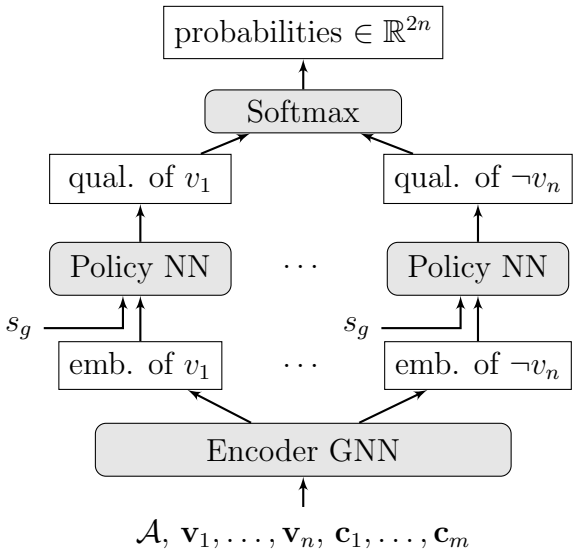


Figure 3.4: Sketch of the architecture for a formula φ with n variables v_i and m clauses. s_g is the global state of the solver, $\mathcal{A}$ is the adjacency matrix, and $\mathbf{v}_i$ and $\mathbf{c}_i$ are the variable and clause labels.

embedding as $l_0 = \mathbf{0}$, and for each round $1 \leq t \leq \tau$, we define the literal embedding $l_t \in \mathbb{R}^{\delta_L}$ for every literal l and the clause embedding $c_t \in \mathbb{R}^{\delta_C}$ for every clause $c \in C$ as follows:

$$c_t = \text{ReLU}\left(\sum_{l \in c} \mathbf{W}_L[\mathbf{v}_l^\top, l_{t-1}^\top, \bar{l}_{t-1}^\top] + \mathbf{B}_L\right)$$
$$l_t = \text{ReLU}\left(\sum_{c, l \in c} \mathbf{W}_C[\mathbf{c}^\top, c_t^\top] + \mathbf{B}_C\right)$$

The trainable parameters of our model are indicated as bold capital letters. They consist of the matrix $\mathbf{W}_L$ of shape $(2\delta_L + \lambda_V, \delta_C)$, the vector $\mathbf{B}_L$ of dimension δ_C, the matrix $\mathbf{W}_C$ of shape $(\delta_C + \lambda_C, \delta_L)$, and the vector $\mathbf{B}_C$ of dimension δ_L.

It is interesting to note that the GNN architecture is quite suited to modeling the CDCL solving process - new clauses that are learned along the way are represented as changes to the graph, and facilitate new information pathways in the GNN.

Policy Network

The policy network predicts the quality of each literal based on the literal embedding and the global solver state. The *global solver state* is a collection of $\lambda_G = 5$ values that include only the essential parts of solver state that are not associated with any particular variable or clause. We provide additional details in Appendix B.2. The policy network thus maps the *final literal embedding* $[\mathbf{v}_l^\top, l_\tau^\top, \bar{l}_\tau^\top]$ concatenated with the global solver state to a single numerical value

indicating the *quality* of the literal. The policy network thus has $\lambda_V + 2\delta_L + \lambda_G$ inputs, which are followed by two fully-connected layers. The two hidden layers use the ReLU nonlinearity. We turn the predictions of the policy network into action probabilities by a masked softmax. We mask all "illegal" actions, effectively ignoring the embeddings of variables which are universal, or are assigned already.

Note that the policy network predicts a score for each literal *independently*. All information about the graph that is relevant to the policy network must hence flow through the literal embedding. Since we experimented with graph neural networks with few iterations this means that *the quality of each literal is decided locally*. The rationale behind this design is that it is simple and efficient.

3.5 Experiments

We conducted several experiments to examine whether we can improve the heuristics of the logic solver CADET through our deep reinforcement learning approach. Throughout the following experiments, we try to explicitly answer the following questions

- Q0 Can we learn to predict good actions for one formula?
- Q1 Can we learn to predict good actions for a family of formulas?
- Q2 How does the policy trained on short episodes generalize to long episodes?
- Q3 How well does the learned policy generalize to formulas from a different family of formulas?
- Q4 Does the improvement in the policy outweigh the additional computational effort? That is, can we solve more formulas in less time with the learned policy?

Baselines

While there are no competing learning approaches yet, human researchers and engineers have tried many heuristics for selecting the next variable. As explained in Sec. 3.1, VSIDS is the best known heuristic for the solver we consider. It has been a dominant heuristic for SAT and several CDCL-based QBF algorithms for over 20 years now [92, 39, 23, 82, 105]. We therefore consider VSIDS as the main baseline. In QBF, much like how it works in SAT, VSIDS maintains an *activity score* per variable and always chooses the variable with the highest activity that is still available. The activity reflects how often a variable recently occurred in conflict analysis. To select a literal of the chosen variable, VSIDS uses the Jeroslow-Wang heuristic [64], which selects the polarity of the variable that occurs more often, weighted by the size of clauses they occur in. For reference, we also consider the *Random* heuristic, which chooses one of the available actions uniformly at random.

Data

Synthesized Data

We generated ourselves two random families of problems which we call *Boolean* and *Words*.

Boolean is a data set that starts with propositional formulas generated using FUZZSAT, similarly to those described in Sec. 2.4. To turn this kind of propositional formulas into QBFs,we randomly selected 4 variables to be universally quantified. This resulted in a more or less even split of true and false formulas. The formulas have 50.7 variables on average.

Words is a data set of random expressions over (signed) bitvectors. The top-level operator is a comparison ($=, \leq, \geq, <, >$), and the two subexpressions of the comparison are arithmetic expressions. The number of operators and leafs in each expression is 9, and all bitvectors have word size 8. The expressions contain up to four bitvector variables, alternatingly assigned to be existentially and universally quantified. The formulas are simplified using the circuit synthesis tool ABC, and then they are turned into CNF using the standard Tseitin transformation. The resulting formulas have 71.4 variables on average and are significantly harder for both Random and VSIDS. For example, the first formula from the data set looks as follows: $\forall z.\exists x.((x - z)\ \mathsf{xor}\ z) \neq z + 1$, which results in a QBF with 115 variables and 298 clauses. This statement happens to be true and is solved with just 9 decisions using the VSIDS heuristic.

Non-Synthesized Data

In contrast to most other works in the area, we evaluate our approach over a benchmark that (1) has been generated by a third party before the conception of this paper, and (2) is challenging to state-of-the-art solvers in the area. We take a pre-existing set of formulas assumed to share some common structure, some of them extremely challenging, and we want to see if by training on the easy ones we can solve more of the hard ones. We consider a set of formulas representing the search for reductions between collections of first-order formulas generated by [65], which we call *Reductions* in the following. Reductions is interesting from the perspective of QBF solvers, as its formulas are often part of the QBF competition. It consists of 4608 formulas of varying sizes and with varying degrees of hardness. On average the formulas have 316 variables; the largest formulas in the set have over 1600 variables and 12000 clauses. We filtered out 2573 formulas that are solved without any heuristic decisions. We further set aside a test set of 200 formulas, leaving us with a training set of 1835 formulas.

We additionally consider the 2QBF evaluation set of the annual competition of QBF solvers, QBFEVAL [104]. This will help us to study cross-benchmark generalization.

Training Details

We jointly train the encoder network and the policy network using a version of REINFORCE [156], a simple PG algorithm. We deviate from it only by normalizing the returns from episodes to expectation zero and variance one. For each batch we sample a *single*

formula from the training set, and generate b episodes by solving it multiple times. This bit is important - sampling different formulas in a single batch adds variance to the training, because not all formulas are inherently of the same difficulty. In each episode we run Cadet for up to 400 steps using the latest policy. Then we assign rewards to the episodes and estimate the gradient. We apply standard techniques to improve the training, including gradient clipping, normalization of rewards, and whitening of input data.

We assign a small negative reward of -10^{-4} for each decision to encourage the heuristic to solve each formula in fewer steps. When a formula is solved successfully, we assign reward 1 to the last decision. In this way, we effectively treat unfinished episodes (> 400 steps) as if they take 10000 steps, punishing them strongly.

Results

Preliminary Results

Before we started with our full experiments, we wanted to make sure question Q0 is answered positively - that our model can at least learn how to "navigate" a single formula, one which VSIDS solves in 11 steps. On this single formula we trained both with REINFORCE, and Deep Q-Learning. To our surprise, while REINFORCE quickly succeeded in finding a good 6 decision solution and converged on it, the QL algorithm failed to find a good solution. We conjectured that because the total solution time is highly sensitive to the branching decisions in Cadet, the Q-function landscape is difficult for QL to learn, and that PG, implementing a stochastic policy, finds it easier to navigate. At any rate, we proceeded with REINFORCE, though by no means are certain that QL algorithms can't work in this domain.

After succeeding in learning to solve one formula, we gradually increased the number of formulas in the data set. We discovered an interesting behaviour - for a small number of formulas (we assume how many exactly depends on the model capacity, in our case it was 5-6), the model was able to learn a good solution for them all. As we increased the number, it started straining, rather than converging it started "forgetting" the solution to one formula in order to remember another. With 20+ formulas, it seemed to learn nothing, leaving the average number of decisions mostly unchanged. Only when we trained on a training set of 500 formulas we started to see a clear pattern of improvement over the average number of decisions for the entire training set. We conjecture this is the point at which the model starts "generalizing" to the family of problems rather than remembering (or failing to remember) a solution to a few of them.

Main Results

We trained the model described in Section 4.2 on the *Reductions* training set. We denote the resulting policy *Learned* and present the aggregate results in Figure 3.5 as a *cactus plot*, as usual for logic solvers. The cactus plot in Figure 3.5 indicates how the number of solved formulas grows for increasing decision limits on the *test set* of the *Reductions* formulas. In a

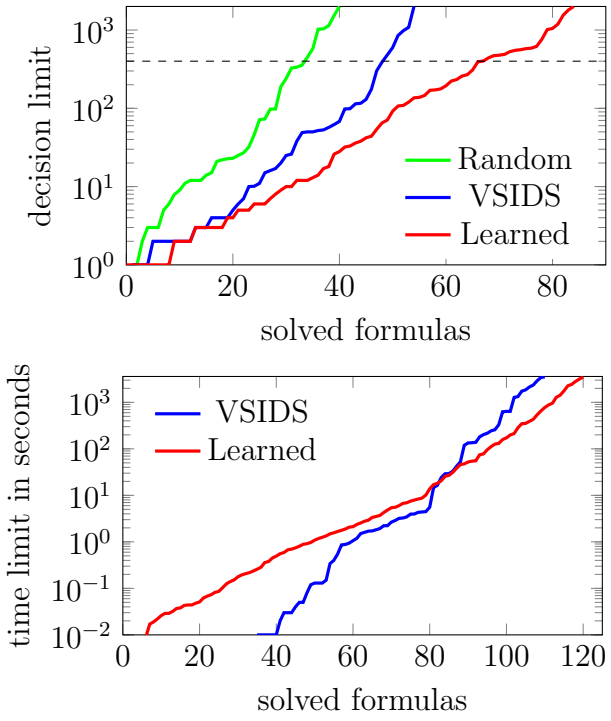


Figure 3.5: Two cactus plots showing how the number of solved formulas from the **test** set grows with increasing resource bounds. **Left:** Comparing the number of formulas solved with growing **decision limit** for Random, VSIDS, and our learned heuristic. **Right:** Comparing the number of formulas solved with growing **wall clock time**. Lower and further to the right is better.

cactus plot, we record one episode for each formula and each heuristic. We then sort the runs of each heuristic by the number of decisions taken in the episode and plot the series. When comparing heuristics, lower lines (or lines reaching further to the right) are thus better, as they indicate that more formulas were solved in less time.

We see that for a decision limit of 400 (dashed line in Fig. 3.5, left), i.e. the decision limit during training, Learned solved significantly more formulas than either of the baselines. The advantage of Learned over VSIDS is about as large as VSIDS over purely random choices. This is remarkable for the field and we can answer Q1 positively.

Figure 3.5 (left) also shows us that Learned performs well far beyond the decision limit of 400 steps that was used during its training. Observing the vertical distance between the lines of Learned and VSIDS, we can see that the advantage of Learned over VSIDS even grows exponentially with an increasing decision limit. (Note that the axis indicating the number of decisions is log-scaled.) We can thus answer Q2 positively.

A surprising fact is that small and shallow neural networks already achieved the best results. Our best model uses $\tau = 1$, which means that for judging the quality of each variable, it only looks at the variable itself and the immediate neighbors (i.e. those variables it occurs

together with in a constraint). The hyperparameters that resulted in the best model are $\delta_L = 16$, $\delta_C = 64$, and $\tau = 1$, leading to a model with merely 8353 parameters. The small size of our model was also helpful to achieve quick inference times.

Note, this stands in a stark difference to the results of the equivalence and SAT experiments from the previous chapter, where more iterations improved the results. Performance of GNN iterations in the RL experiments peaked at about 1-3, and then started declining. We conjecture this may be because unlike in deciding SAT (and possibly equivalence classes), there is no message-passing algorithm that "solves" the problem of which literal to branch on. We are no longer approximating marginal probabilities.

To answer Q3, we evaluated the learned heuristic also on our second data set of formulas from the QBF solver competition QBFEVAL. Random solved 67 formulas, VSIDS solved 125 formulas, and Learned solved 111 formulas. The policy trained on *Reductions* significantly improved over random choices, but does not beat VSIDS. This is hardly surprising, as our learning approach specialized the solver to a specific—different—distribution of formulas. Also it must be taken into account that the solver CADET has been tuned to QBFEVAL over year, and hence may perform much stronger on QBFEVAL than on the Reductions benchmark.

To answer our last question, Q4, we compare the runtime of CADET in with our learned heuristic to CADET with the standard VSIDS heuristic. In Fig. 3.5 (right) we see that for small time limits (up to 10 seconds), VSIDS still solves more formulas than the learned heuristic. But, for higher time limits, the learned heuristic starts to outperform VSIDS. For a time limit of 1 hour, we solved 120 formulas with the learned heuristic while only 110 formulas were solved with VSIDS (see right top corner). Conversely, for solving 110 formulas the learned heuristic required a timeout of less than 12 minutes, while VSIDS took an hour. Furthermore, our learning and inference implementation is written in Python and not particularly optimized. The NN agent is running in a different process from CADET, and incurs an overhead per step for inter-process communication and context switches, which is enormous compared to the pure C implementation of CADET using VSIDS. This overhead could be easily reduced, and so we expect the advantage of our approach to grow.

Additional Results

For the *Boolean* Dataset, we generated 5k formulas, and split them into 4k training and 1k testing formulas. The results are plotted in Figure 3.6, where we see that training a model on these formulas (we call this model *Boolean*, like the data set) results in significantly better performance than VSIDS, and well beyond the original 400 decision limit. The advantage of the learned heuristic over VSIDS and Random is smaller compared to the experiments on Reductions in the main part of the paper. We conjecture that this is due to the fact that these formulas are much easier to begin with, which means that there is not as much potential for improvement.

In Figure 3.7 we see that training a new model on the *Words* dataset again results in significantly improved performance. (We named the model *Words*, after the data set.)

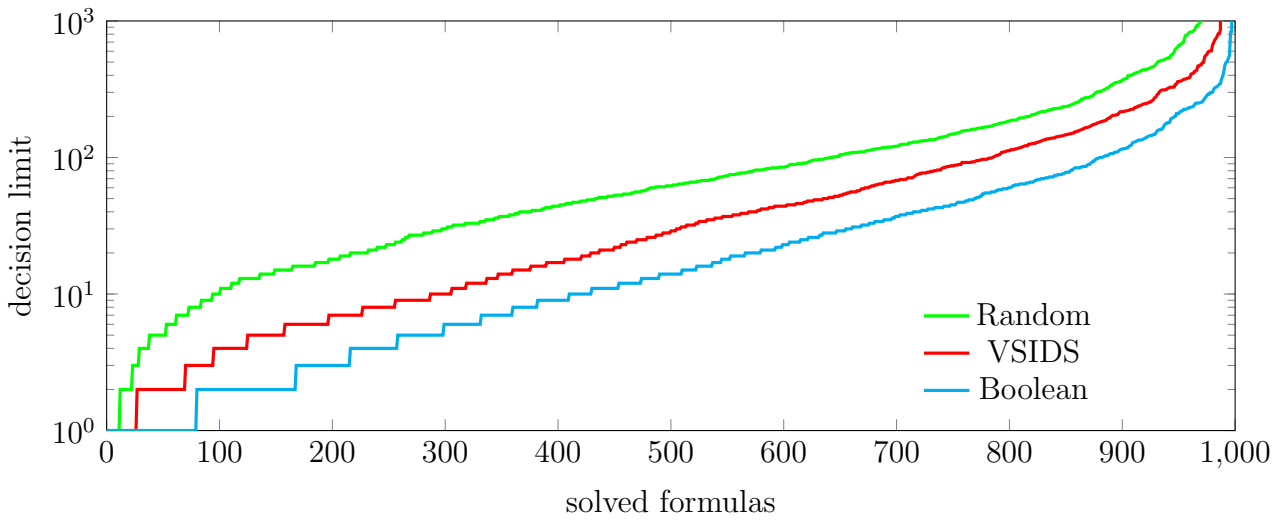


Figure 3.6: A cactus plot describing how many formulas from the **test** set were solved within growing decision limits on the *Boolean* test set. Lower and further to the right is better.

Additional Experiments on Generalization to Larger Formulas

An interesting observation that we made is that models trained on sets of small formulas generalize well to larger formulas from similar distributions. To demonstrate this, we generated a set of larger formulas, similar to the *Words* dataset. We call the new dataset *Words30*, and the only difference to *Words* is that the expressions have size 30. The resulting formulas have 186.6 variables on average. This time, instead of training a new model, we test the model trained on *Words* (from Figure 3.7) on this new dataset.

In Figure 3.8, we see that the overall hardness (measured in the number of decisions needed to solve the formulas) has increased a lot, but the relative performance of the heuristics is still very similar. This shows that the heuristic learned on small formulas generalizes relatively well to much larger/harder formulas.

In Fig. 3.5, we have already observed that the heuristic also generalizes well to much longer episodes than those it was trained on. We believe that this is due to the "locality" of the decisions we force the network to take: The graph neural network approach uses just one iteration, such that we force the heuristics to take very local decisions. Not being able to optimize globally, the heuristics have to learn local features that are helpful to solve a problem sooner rather than later. It seems plausible that this behavior generalizes well to larger formulas (Fig. 3.8) or much longer episodes (Fig. 3.5).

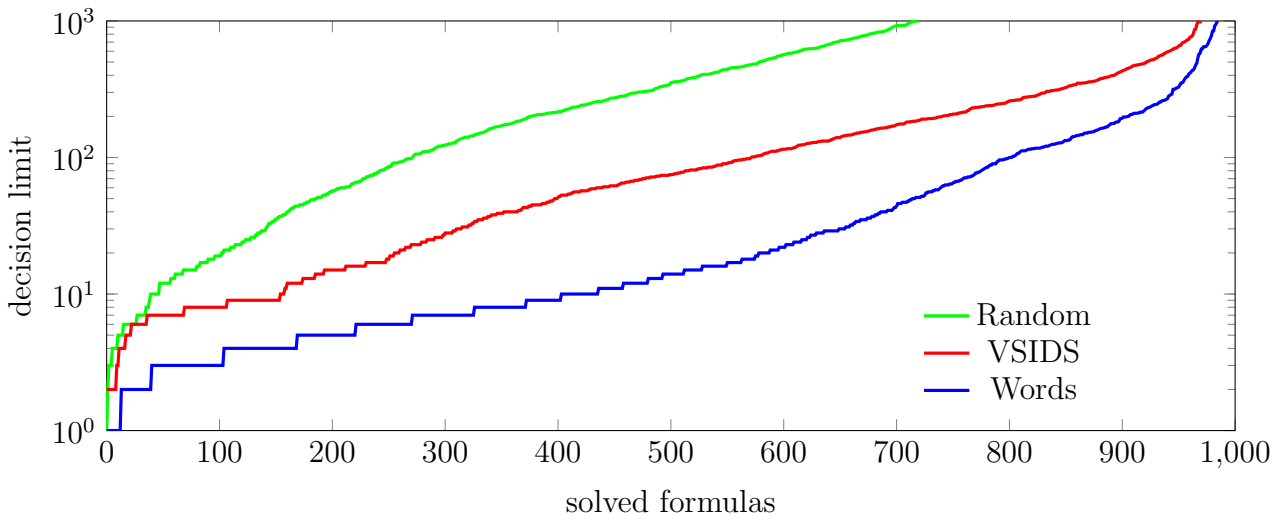


Figure 3.7: A cactus plot describing how many formulas from the **test** set were solved within growing decision limits on the *Words* test set. Lower and further to the right is better.

3.6 More Related Work

Independent from our work, GNNs for Boolean logic have been explored in NeuroSAT [121], where the authors use it to solve the SAT problem directly. While using a similar neural architecture, the network is not integrated in a state-of-the-art logic solver, and does not improve the state of the art in performance. Selsam and Bjørner [119] recently extended NeuroSAT to use its predictions in a state-of-the-art SAT solver. In contrast to their work, we integrate GNNs much tigher into the solver and train the heuristics directly through reinforcement learning. Thus allow deep learning to take direct control of the solving process. Also, we focus on QBF instead of SAT, which strongly affects the runtime tradeoffs between spending time on "thinking" about a better decision versus executing many "stupid" decisions.

Amizadeh, Matusevych, and Weimer [5] suggest an architecture that solves circuit-SAT problems. Unlike NeuroSAT, and similar to our approach, they train their model directly to find a satisfying assignment by using a differentiable "soft" satisfiability score as their loss. However, like NeuroSAT, their approach aims to solve the problem from scratch, without leveraging an existing solver, and so is difficult to scale to state-of-the-art performance. They hence focus on small random problems. In contrast, our approach improves the performance of a state-of-the-art algorithm. Furthermore, our learned heuristic applies to SAT and UNSAT problems alike.

Yang et al. [160] extended the NeuroSAT architecture to 2QBF problems. In contrast to our work, they do not embed their GNN model in a modern DPLL solver, and instead try to predict good counter-examples for a CEGAR solving approach. They focus on formulas with 18 variables, which are trivial for state-of-the-art solvers. Chen and Yang [32] showed that a

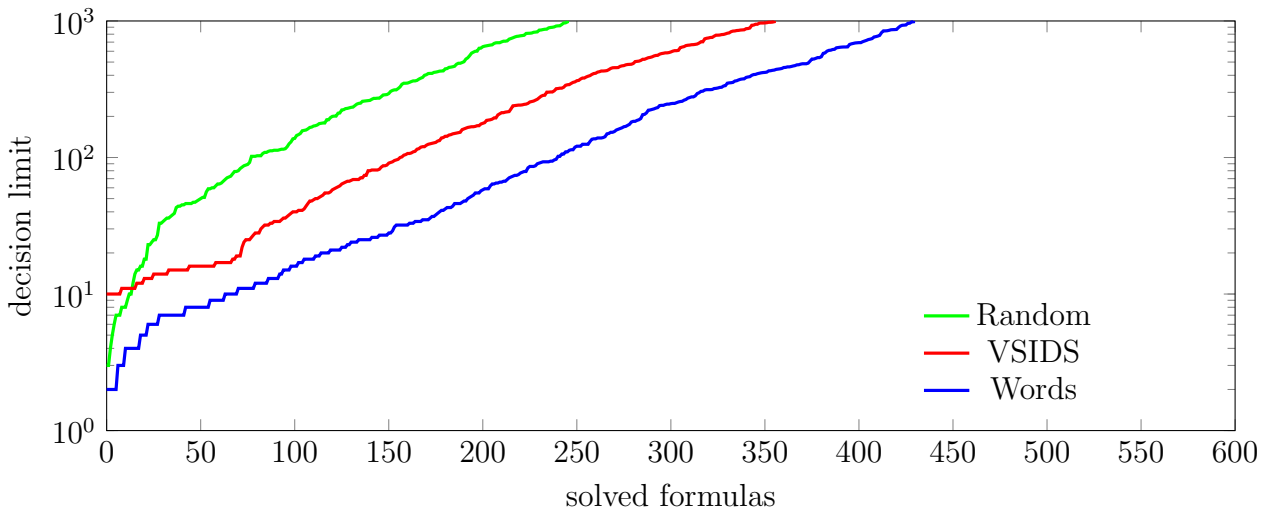


Figure 3.8: A cactus plot describing how many formulas were solved within growing decision limits on the *Words30* test set. Lower and further to the right is better. Note that unlike in the other plots, the model *Words* was not trained on this distribution of formulas, but on the same *Words* dataset as before.

pure GNN approach is unable to solve Boolean formulas when they are unsatisfiable, which in our work is addressed by combining GNNs with a logic reasoning engine.

Reinforcement learning has been applied to other logic reasoning tasks. Kaliszyk et al. [66] recently explored learning linear policies for tableaux-style theorem proving. Kurin et al. [75] follow a similar approach to ours for SAT solvers, but only evaluate on small synthetic formulas and do not improve the overall performance of the underlying SAT solver. Kusumoto, Yahata, and Sakai [76] applied reinforcement learning to propositional logic in a setting similar to ours; just that we employ the learning in existing strong solving algorithms, leading to much better scalability. Balunovic, Bielik, and Vechev [13] use deep reinforcement learning to improve the application of *high-level* strategies in SMT solvers, but do not investigate a tighter integration of deep learning with logic solvers. Also other works on combinatorial search explored the use of GNNs (some trained with reinforcement learning) for problems such as random SAT [162], coloring graphs [55], and MILP [42].

Most previous approaches that applied neural networks to logical formulas used LSTMs or tree models syntax-tree of formulas [26, 58, 3, 83, 41, 34, 31] or classical ML models [43, 66, 135]. Instead, we suggest a GNN approach, based on a graph-view on formulas in CNF. Recent work suggests that GNNs appear to be a good architecture for logics [100, 154]. [14, 54, 159] provide a learning environments around interactive theorem provers.

Other competitive QBF algorithms include expansion-based algorithms [21, 103], CEGAR-based algorithms [62, 61, 106], circuit-based algorithms [71, 141, 60, 59], and hybrids [63, 142]. Recently, [60] successfully explored the use of (classical) machine learning techniques to

address the generalization problem in QBF solvers.

3.7 Conclusions

We presented an approach to improve the heuristics of a backtracking search algorithm for Boolean logic through deep reinforcement learning. Our approach brings together the best of two worlds: The superior flexibility and performance of intuitive reasoning of neural networks, and the ability to explain (prove) results in formal reasoning. The setting is new and challenging to reinforcement learning; QBF is a very general, combinatorial problem class, featuring an unbounded input-size and action space. We demonstrate that these problems can be overcome, and that our method reduces the overall execution time of a competitive QBF solver by a factor of 10 after training on similar formulas.

This work demonstrates the huge potential that lies in the tight integration of deep learning and logical reasoning algorithms, and hence motivates more aggressive research efforts in the area. Our experiments suggest two challenges that we want to highlight: (1) We used very small neural networks, and—counterintuitively—larger neural networks were not able to improve over the small ones in our experiments. (2) The performance overhead due to the use of neural networks is large; however we think that with more engineering effort we could be significantly reduce this overhead.

Chapter 4

Learning for Model Counting

In previous chapters we've seen how to represent Boolean logical formulas in a way that allows for processing by NN, and the class of model architectures suitable for such formulas. We then showed how to cast the DPLL-based algorithm of the the 2QBF solver Cadet as an MDP, a kind of a game between the environment and the branching Heuristic, and how to leverage the graph representation of formulas to automatically learn this heuristic.

In this chapter, first of all, we show that this technique is not limited to the specific 2QBF problem, and demonstrate that it can achieve substantial improvements over the state of the art in the different, more difficult (from the perspective of computational complexity) constraint problem of *Model Counting*. Furthermore, we begin to expand our point of view beyond CNF formulas CS process, and into the higher-level problem domains from which they are encoded. We argue that the information lost during the encoding to CNF can be incorporated into the learning process, and show that it has great potential for improving solvers. We also concentrate on one specific problem domain and try to make sense of what the learned heuristic is actually learning, when examined in the original problem domain.

4.1 Background

#SAT

Notation

We use the notation from Sec. 2.3, and add a few more formula-centric definitions, which will be convenient when discussing the #SAT version of the DPLL algorithm. We denote the set of literals and clauses of a CNF formula ϕ by $\mathcal{L}(\phi)$ and $\mathcal{C}(\phi)$, respectively. As before, we assume that all formulas are in CNF.

A *truth assignment* for any formula ϕ is a mapping of its variables to $\{0, 1\}$ (**false**/**true**). Thus there are 2^n different truth assignments when ϕ has n variables. A truth assignment π satisfies a literal ℓ when ℓ is the variable v and $\pi(v) = 1$ or when $\ell = \neg v$ and $\pi(v) = 0$. It satisfies a clause when at least one of its literals is satisfied. A CNF formula ϕ is satisfied

when all of its clauses are satisfied under π in which case we call π a *satisfying assignment* for ϕ.

The #SAT problem for ϕ is to compute the number of satisfying assignments.

#SAT Algorithms

DPLL-based #SAT solvers [143, 116, 98] are called *Exact* model counters. There are also approximate model counters with different degrees of probabilistic guarantees [30, 90, 125], from none at all to full Probably-Approximately-Correct (PAC) algorithms which return a count with an ϵ-bounded error (ratio) with probability of $1 - \delta$. Probabilistic algorithms are beyond our scope, and from now on unless otherwise noted on we concentrate on exact solvers.

The simplest algorithm for #SAT is to extend DPLL to make it explore the full set of truth assignments. This is the basis of the CDP solver presented in [24], shown in Algorithm 1. In particular, when the current formula contains an empty clause it has zero models, and when it contains no clauses each of the remaining k unset variables can be assigned **true** or **false** so there are 2^k models (line 6).

Algorithm 1 DPLL extended to count all solutions (CDP)

```
1: function CDP(φ)
2:     if φ contains an empty clause then
3:         return 0
4:     if φ contains no clauses then
5:         k = # of unset variables
6:         return 2^k
7:     Pick a literal l ∈ φ
8:     return CDP(UP(φ, l)) + CDP(UP(φ, ¬l))
```

This algorithm is not very efficient, running in time $2^{\Theta(n)}$ where n is the number of variables in the input formula. Note that as in DPLL, the algorithm is actually a class of algorithms each determined by the procedure used to select the next literal to branch on. The complexity bound is strong in the sense that no matter how the branching decisions are made, we can find a sequence of input formulas on which the algorithm will take time exponential in n as the formulas get larger.However, we can improve on this naive

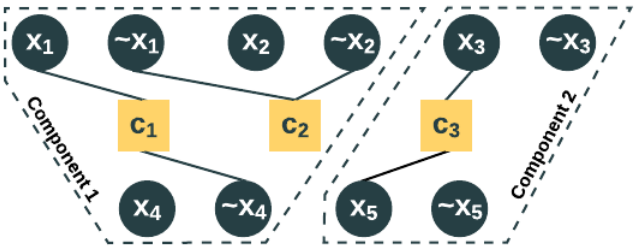


Figure 4.1: An example Clause-Literal Incidence Graph (CLIG) for a formula with two components: $(x_1 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_5)$.

algorithm by exploiting the way the problem decomposes into components, as first suggested in [17] and used in the RELSAT solver.

More formally, two sets of clauses are called *disjoint* if they share no variables. A component $C \subset \mathcal{C}(\phi)$ is a subset of ϕ's clauses that is disjoint from its complement $\mathcal{C}(\phi) - C$. In a CVIG representation this coincides with a maximal connected component of the graph. In CLIG representation, because the components were defined on variables and the graph is defined on literals, a component over variables can span multiple graph components over literals, as seen in Fig. 4.1. Although most formulas initially consist of only onecomponent, as variables are set by branching decisions and clauses are removed, the reduced formulas will often break up into multiple components.

The main observation is that each component can be solved independently. They can be thought of as different formulas, inverting the intuition behind batching of adjacency matrices described in Sec. 2.3. The maximal disjoint connected components of formula ϕ, C_1, ..., C_k, can be efficiently computed, and then we have: $\text{COUNT}(\phi) = \prod_{i=1}^{k} \text{COUNT}(C_i)$. Incorporating this observation into CDP is shown in Algorithm 2.

Algorithm 2 Using Components

```
1: function RELSAT(φ)
2:     Pick a literal l ∈ φ
3:     #l = COUNTSIDE(φ, l)
4:     #¬l = COUNTSIDE(φ, ¬l)
5:     return #l + #¬l

6: function COUNTSIDE(φ, l)
7:     φ_l = UP(φ, l)
8:     if φ_l contains an empty clause then
9:         return 0
10:    if φ_l contains no clauses then
11:        k = # of unset variables
12:        return 2^k
13:    K = FINDCOMPONENTS(φ_l)
14:    return ∏_{κ∈K} RELSAT(κ)
```

Breaking the formula into components can yield considerable speedups depending on n_0, the number of variables needed to be set before the formula is broken into components. If we consider a hypergraph in which every variable is a node and every clause is a hyperedge over the variables mentioned in the clause, then the branch-width [111] of this hypergraph provides an upper bound on n_0. As a result we can obtain a better upper bound on the run time of RELSAT of $n^{O(w)}$ where w is the branch-width of the input's hypergraph. However,

this run time will only be achieved if the branching decisions are made in an order that respects the branch decomposition with width w. In particular, there exists a sequence of branching decisions achieving a run time of $n^{O(w)}$. Computing that sequence would require time $n^{O(1)}2^{O(w)}$ [110], hence a run time of $n^{O(w)}$ can be achieved.

The final main element added is component caching [11, 12], which exploits the fact that, depending on decomposition to components (which depends on branching order), the same sub-components appear at different points in the search tree. That gives us the #DPLLCache algorithm in Algorithm 3 on which modern #SAT solvers are based. It has a better upper bound of $2^{O(w)}$, which again can be achieved with a $n^{O(1)}2^{O(w)}$ computation of an appropriate sequence of branching decisions.

Algorithm 3 Component Caching DPLL

```
1: #ℓ = CountSide(φ, ℓ)
2: function #DPLLCache(φ)
3:    if inCache(φ) then
4:       return cacheLookUp(φ)
5:    Pick a literal ℓ ∈ L(φ)
6:    #ℓ = CountSide(φ, ℓ)
7:    #¬ℓ = CountSide(φ, ¬ℓ)
8:    addToCache(φ, #ℓ + #¬ℓ)
9:    return #ℓ + #¬ℓ

10: function CountSide(φ, ℓ)
11:   φ_ℓ = UP(φ, ℓ)
12:   if φ_ℓ contains an empty clause then
13:      return 0
14:   if φ_ℓ contains no clauses then
15:      k = # of unset variables
16:      return 2^k
17:   K = findComponents(φ_ℓ)
18:   return ∏_{κ∈K} #DPLLCache(κ)
```

In practice, the branch-width of most instances is very large, making a run time of $2^{O(w)}$ infeasible. Computing a branching sequence to achieve that run time is also infeasible. Fortunately, in practical instances unit propagation is also very powerful. This means that making only a few decisions ($< w$) often allows unit propagation to set w or more variables thus breaking the formula apart into separate components. Furthermore, most instances are falsified by a large proportion of their truth assignments. This makes clause learning an

effective addition to #SAT solvers, as with it the solver can more effectively traverse the non-solution space.

In sum, for #SAT solvers the branching decisions try to achieve complex and sometimes contradictory objectives. Making decisions that split the formula into larger components near the top of the search tree (i.e., after only a few decisions are made) allows greater speedups, while generating many small components near the bottom of the search trees (i.e., after many decision are made) does not help the solver. Making decisions that generate the same components under different branches allows more effective use of the cache. And making decisions that allow the solver to learn more effective clauses allows the solver to more efficiently traverse the often large space of non-solutions.

For that reason, as in the case of QBF, the intuition behind the branching heuristics in #SAT are not as well understood as in the SAT case. We focus on the exact solver `SharpSAT` [143] which is based on Algorithm 3 augmented with clause learning. `SharpSAT` uses the VSADS heuristic [115], which is a linear combination of a heuristic aimed at making clause learning effective (VSIDS) and a count of the number of times a variable appears in the current formula.

Evolution Strategies

Evolution Strategies (ES) are a class of zeroth order black-box optimization algorithms [18, 155]. Inspired by natural evolution, a population of parameter vectors (genomes) is perturbed (mutated) at every iteration, giving birth to a new generation. The resulting offspring are then evaluated by a predefined fitness function. Those offspring with higher fitness score will be selected for producing the next generation.

We adopt a version of ES that has shown to achieve great success in the standard RL benchmarks [114]: Let $f : \Theta \to \mathbb{R}$ denote the fitness function for a parameter space Θ, e.g., in an RL environment, f computes the stochastic episodic reward of a policy π_θ. To produce the new generation of parameters of size n, Salimans et al. [114] uses an additive Gaussian noise with standard deviation σ to perturb the current generation: $\theta_{t+1}^{(i)} = \theta_t + \sigma\epsilon^{(i)}$, where $\epsilon^{(i)} \sim \mathcal{N}(0, I)$. We then evaluate every new generation with fitness function $f(\theta_{t+1}^{(i)})$ for all $i \in [1, \ldots, n]$. The update rule of the parameter is as follows,

$$\begin{aligned}\theta_{t+1} &= \theta_t + \eta\nabla_\theta \mathbb{E}_{\theta\sim\mathcal{N}(\theta_t,\sigma^2 I)}[f(\theta)] \\ &\approx \theta_t + \eta\frac{1}{n\sigma}\sum_i^n f(\theta_{t+1}^{(i)})\epsilon^{(i)},\end{aligned}$$

where η is the learning rate. The update rule is intuitive: each perturbation $\epsilon^{(i)}$ is weighted by the fitness of the corresponding offspring $\theta_{t+1}^{(i)}$. We follow the rank-normalization and mirror sampling techniques of Salimans et al. [114] to scale the reward function and reduce the variance of the gradient, respectively.

4.2 Method

We formalize the problem of learning the branching heuristic for #DPLLCache as a *Markov Decision Process* (MDP), similar to how we did in Sec. 3.3. The environment is now `SharpSAT`, and because every component is solved independently by #DPLLCache, the input to the policy is only the current component rather than the entire formula graph. The objective function is again to reduce the number of decisions the solver makes while solving the counting problem and so we use the same reward function:

$$R(s) = \begin{cases} 1 & \textit{if } s \textit{ is a terminal state with} \\ & \text{“instance solved”} \textit{ status} \\ -r_{penalty} & \textit{otherwise} \end{cases}$$

If not finished, episodes are aborted after a predefined max number of steps, without receiving the termination reward.

Evolution Strategies vs RL

One problem that became apparent during the training process of Cadet described in Chapter 3 was the dependence of training time to convergence on the average length of episodes, which in turn depends on both the problem's “inherent difficulty”[1], and the branching policy.

Part of this dependence on episode length is unavoidable. Longer episodes contain more information, and at the very least it takes a linear amount of work to process it. However, part of it is a property of RL algorithms, and as shown in Vemula, Sun, and Bagnell [151], the exploration complexity of an action-space exploration RL algorithm (e.g, Q-Learning, Policy Gradient) increases with the size of the action space and the problem horizon. The intuition behind this is quite simple - All RL algorithms use (pseudo) randomness for exploration of the solution space. In the case of the Policy Gradient algorithm that we use, this randomness is implicitly injected through the estimation of the policy gradient in Eq. 3.8. To compute the gradient we sample a batch of on-policy “rollouts”. This means generating episodes by running the current policy on a formula, and on each time step sample an action according to the action distribution the model outputs (remember, a stochastic policy is a function $\pi : S \to \mathcal{P}(A)$ from observation to distribution over actions). The randomness we inject is therefore clearly a function of the number of actions taken and the entropy of the distribution sampled on every step, which usually correlates with the size of the action space A. This translates into a larger variance in the policy gradients estimation, and therefore longer training time.

On the other hand, a parameter-space exploration algorithm like ES is independent of episode length or action space size. It injects the randomness directly into model parameters space, and is in fact entirely oblivious to the entire MDP formulation - the policy and the

[1] We leave this vague for now, but it is not difficult to think of formal measures such as length of optimal branching order

environment are a black box from the ES perspective, including the "episode". ES simply sees it as a (stochastic) function where a fitness function is evaluated for a given point in model parameter space. Since both our action space and episode horizon can be quite large (up to 20,000 and 1,000, respectively), and our model size is relatively modest (a few thousands of parameters), we decided to turn to parameter-space exploration using ES algorithm. Therefore, we choose to use a version of ES proposed by Salimans et al. [114] (see Sec. 4.1) for optimizing our agent. This allowed us to train on much longer episodes than with Policy Gradient, in a fraction of the training time (an improvement of roughly one order of magnitude).

Processing SharpSAT Components with GNNs

In the language of Sec. 2.3, we use a CLIG representation with literal tying when given the current component from `SharpSAT`. If the initial vector representation are denoted by $h_c^{(0)}$ for each clause $c \in C$ and $h_l^{(0)}$ for each literal $l \in L$, both of which are learnable model parameters. We run the following message passing steps iteratively:

- Literal to Clause (L2C):

$$h_c^{(k+1)} = \mathcal{A}\Big(h_c^{(k)}, \sum_{l \in c} [h_l^{(k)}, h_{\bar{l}}^{(k)}]; W_C^{(k)}\Big), \quad \forall c \in C,$$

- Clause to Literal (C2L):

$$h_l^{(k+1)} = \mathcal{A}\Big(h_l^{(k)}, \sum_{c, l \in c} h_c^{(k)}; W_L^{(k)}\Big), \quad \forall l \in L,$$

where $\mathcal{A}$ is a nonlinear aggregation function, parameterized by $W_C^{(k)}$ for clause aggregation and $W_L^{(k)}$ for literal aggregation at the k^{th} iteration. We use an aggregation function from a recently suggested GNN architecture [157] named *Graph Isomorphism Network* (GIN), where they show it has some beneficial expressiveness properties. Specifically, set $\mathcal{A}(x, y; W) = \text{MLP}((1+\epsilon)x + y; W)$, where ϵ is a hyperparameter.

Semantic Features

In practice, CNF formulas are encoded from a higher level problem domain with additional semantics. These features of the original problem domain, which we call *semantic features*, are all but lost during the encoding process. Classical constraint solvers only process CNF formulas, and so their heuristics by definition are entirely independent of any specific problem domain, and only consider internal solver properties, such as variable activities. These internal solver properties are a function of the CNF representation and internal solver dynamics, and quite detached from the original problem domain. Thus, it is not unreasonable that semantic features of the original problem domain could contain additional useful structure that can be exploited by the low-level solver heuristic.

One such semantic feature that often naturally arises in real-world problems is *time*. Many problems, such as dynamical systems and bounded model checking, are iterative in nature, with a distinct temporal dimension to them. In the original problem domain, there is often a state that is evolved through time via repeated applications of a state transition function. A structured CNF encoding of such problems usually maps every state s_t to a set of variables, and adds sets of clauses to represent the dynamical constraints between every transition (s_t, s_{t+1}). Normally, all temporal information is lost in reduction to CNF. However, with a learning-based approach, the time-step feature from the original problem can be readily incorporated as an additional input to the network, effectively annotating each variable with its time-step. In our experiments, we represented time by appending to each literal embedding a scalar value (representing the normalized time-step t) before passing it through the output MLP. We perform an ablation study to investigate the impact of this additional feature in Sec. 4.4.

Another example is spatial information. For example, many planning problems in dynamical systems have what amounts to a grid structure (either because they originate from grid-world problems, or discretized versions of real-world continuous problems). In the encoding process described above, such spatial structure is entirely shattered as we encode the high-level grid structure and the system dynamics into a circuit. It is not unreasonable that preserving the original grid structure and processing it through a separate convolutional network could help the learning process (for example, suppose a grid world has lots of squares colored blue to mark "water". While this information is encoded all over the place in the CNF representation, a fairly shallow convolutional network can already easily detect it). We describe this direction as well in the Ablation study.

4.3 Data Generation

Our choice of datasets is guided by a few goals and constraints. First, we try to choose problems that are application-oriented ("industrial") rather than random graphs such as in Yolcu and Póczos [163], Selsam et al. [122], and Kurin et al. [75]. Ideally, from SAT benchmarks or existing literature. We use families of problems with different properties (local vs. global connectivity, sequential circuits, etc.) to better evaluate the regime in which our method is effective. Finally, we use problem families for which there is an available generative process, which is required to generate "easy" problems for training. To those ends, we searched SAT and planning benchmarks for problems whose generative processes were publicly available or feasible to implement. To test the versatility of our method, we made sure that these problems cover a diverse set of domains: sudoku, blocked n-queens, cell *(combinatorial)*; sha-1 preimage attack *(cryptography)*; island, grid_wrld (*planning*), bv_expr, it_expr *(circuits)*:

- cell(R, n, r): Elementary (i.e., one-dimensional, binary) Cellular Automata are simple systems of computation where the cells of an n-bit binary state vector are progressed

through time by repeated applications of a rule R (seen as a function on the state space). Figure 4.7a shows the evolution grid of rules 9, 35 and 49 for 20 iterations.

Reversing Elementary Cellular Automata: Given a randomly sampled state T, compute the number of initial states I that would lead to that terminal state T in r applications of R, i.e., $\left|\{I : R^r(I) = T\}\right|$. The entire r-step evolution grid is encoded by mapping each cell to a Boolean variable ($n \times r$ in total). The clauses impose the constraints between cells of consecutive rows as given by R. The variables corresponding to T (last row of the evolution grid) are assigned as unit clauses. This problem was taken from SATCOMP 2018 [51].

- grid_wrld(s, t): This bounded horizon planning problem from Vazquez-Chanlatte et al. [150] and Vazquez-Chanlatte, Rabe, and Seshia [149] is based on encoding a grid world with different types of squares (e.g., lava, water, recharge), and a formal specification such as *"Do not recharge while wet"* or *"avoid lava"*. We randomly sample a grid world of size s and a starting position I for an agent. We encode to CNF the problem of counting the number of trajectories of length t beginning from I that always avoid lava.

- sudoku(n, k): Randomly generated partially filled $n \times n$ Sudoku problems ($n \in \{9, 16\}$) with multiple solutions, where k is the number of revealed squares.

  Sudoku problems are typically designed to have only one solution but as our goal is to improve a model counter, we relaxed this requirement to count the number of solutions instead.

- n-queens(n, k): Blocked N-Queens problem of size n with k randomly blocked squares on the chess board. This is a standard SAT problem and the task is to count the number of ways to place the n queens on the remaining $n^2 - k$ squares such that no two queens attack each other.

- sha-1(n, k): SHA-1 preimage attack of randomly generated messages of size n with k randomly chosen bits fixed. This problem was taken from SATRACE 2019 and we used the CGEN[2] tool to generate our instances.

- island(n, m): This dataset was introduced by Geffner and Geffner [44] as a *Fully-Observable Non-Deterministic* (FOND) planning problem. There are two grid-like islands of size $n \times n$, each connected by a bridge. The agent is placed at a random location in island 1 and the goal is for it to go to another randomly selected location in island 2. The short (non-deterministic) way is to swim from island 1 to 2, where the agent may drown, and the long way is to go to the bridge and cross it. Crossing the bridge is only possible if no animals are blocking it, otherwise the agent has to move the animals away from the bridge before it can cross it. The m animals are again randomly positioned on the two grids. We used the generative process of Geffner and Geffner [44] to encode compact policies for this task in SAT.

[2]CGEN: `https://github.com/vsklad/cgen`

- bv_expr(n, d, w): Randomly generated arithmetic sentences of the form $e_1 \prec e_2$, where $\prec \in \{\leq, \geq, <, >, =, \neq\}$ and e_1, e_2 are expressions of maximum depth d over n binary vector variables of size w, random constants and operators $(+, -, \wedge, \vee, \neg, \text{XOR}, |\cdot|)$. The problem is to count the number of integer solutions to the resulting relation in $([0, 2^w] \cap \mathbb{Z})^n$.

- it_expr(s, i): Randomly generated arithmetic expression circuits of size s (word size fixed at 8). Effectively implementing a function with input and output of a single word. This function is composed i times. We choose a random word c, and count the number of inputs such that the output is less than c. Formally, if the random circuit is denoted by f, we compute $|\{x | f^i(x) < c\}|$.

4.4 Experiments

To evaluate our method, we designed experiments to answer the following questions: **1) I.I.D. Generalization:** Can a model trained on instances from a given distribution generalize to unseen instances of the same distribution? **2) Upward Generalization:** Can a model trained on small instances generalize to larger ones? **3) Wall-Clock Improvement:** Can the model improve the runtime substantially? **4) Interpretation:** Does the sequence of actions taken by the model exhibit any discernible pattern at the problem level? Our baseline in all comparisons is `SharpSAT`'s heuristic. Also, to make sure that our model's improvements are not trivially attainable without training we tested a `Random` policy that simply chooses a literal uniformly at random. We also studied the impact of the trained model on a variety of solver-specific quality metrics (e.g., cache-hit rate, ...), the results of which are in Appendix **??**.

The grid_wrld, being both a problem of an iterative nature (i.e., steps in the planning problem) and with a clear spatial structure, was a natural candidate for testing our hypothesis regarding the effect of adding the spatial features of Section 4.2. We report the main results for grid_wrld with those features included, and later in this section we perform an ablation study on their effects.

Experimental Setup For each dataset, we sampled 1,800 instances for training and 200 for testing. We trained for 1000 ES iterations. At each iteration, we sampled 8 formulas and 48 perturbations ($\sigma = 0.02$). With mirror sampling, we obtained in total $96 = 48 \times 2$ perturbations. For each perturbation, we ran the agent on the 8 formulas (in parallel), to a total of $768 = 96 \times 8$ episodes per parameter update. All episodes, unless otherwise mentioned, were capped at 1k steps during training and 100k during testing. The agent received a negative reward of $r_{penalty} = 10^{-4}$ at each step. We used the Adam optimizer [70] with default hyperparameters, a learning rate of $\eta = 0.01$ and a weight decay of 0.005.

Hardware Infrastructure We used a small cluster of 3 nodes each with an AMD Ryzen Threadripper 2990WX processor with 32 cores (64 threads) and 128GB of memory. We trained for an average of 10 hours on a dataset of 1000 instances for each problem. For testing the wall-clock time we ran the problems sequentially, to avoid any random interference due to parallelism.

Range of Hyperparameters The model is relatively "easy" to train. The criteria for choosing the hyperparameters was the performance of generalization on i.i.d test set, i.e, lowest possible average number of branching decisions. Once calibrated on the first dataset (cell(35)), we were able to train all models on all datasets without further hyperparameters tuning. The minimal number of episodes per optimization step that worked was 12. We tested a few different GNN architectures, and none was clearly superior over the others. We also varied the number of GNN message-passing iterations but going beyond 2 iterations had a negative effect (on i.i.d generalization) so we settled on 2. GNN messages were implemented by an MLP with ReLU non-linearity. The size of literal and clause embeddings were 32 and the dimensionality of C2L *(resp. L2C)* messages was $32 \times 32 \times 32$ *(resp.* $64 \times 32 \times 32$*)*. As mentioned above, we used $T = 2$ message passing iterations and final literal embeddings were passed through the MLP policy network of dimensions $32 \times 256 \times 64 \times 1$ to get the final score. When using the extra "time" feature, the first dimension of the decision layer was 33 instead of 32. The initial ($T = 0$) embeddings of both literals and clauses were trainable model parameters.

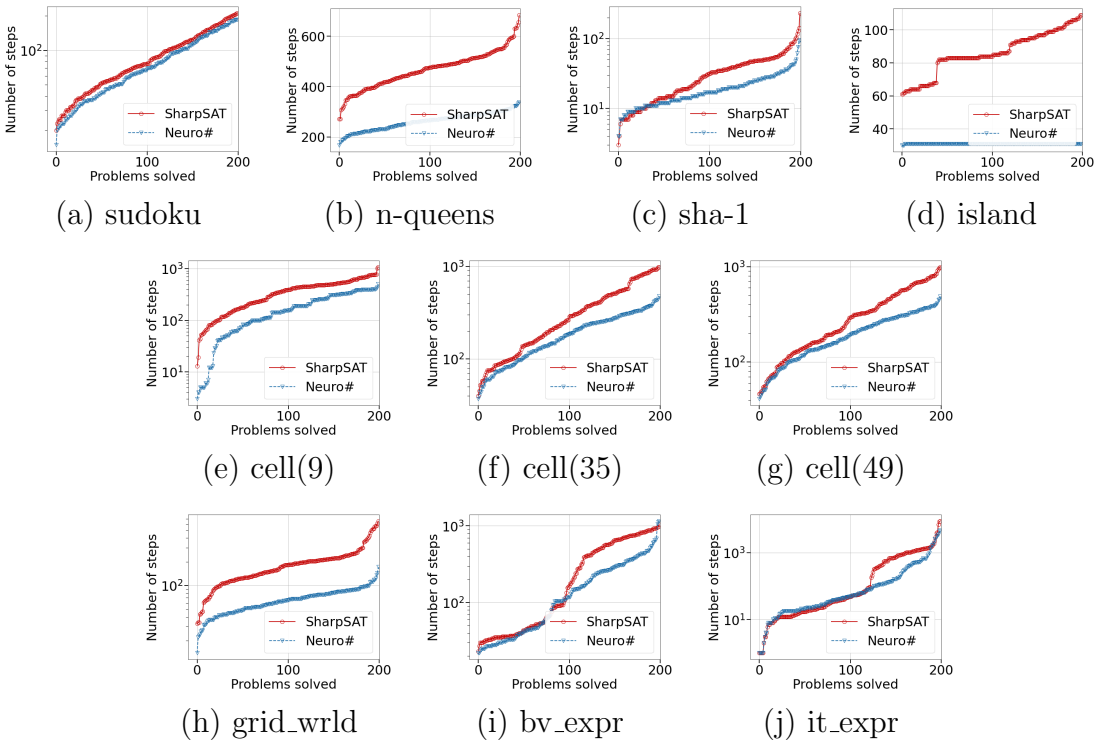
Results

I.I.D. Generalization Table 4.1 summarizes the results of the i.i.d. generalization over the problem domains of Section 4.3. We report the average number of branching steps on the test set. `Neuro#` outperformed the baseline across all datasets. Most notably, on grid_wrld, it reduced the number of branching steps by a factor of 3.0 and on cell, by an average factor of 1.8 over the three cellular rules.

Figure 4.2 shows cactus plots for all of the i.i.d. benchmark problems. Unsurprisingly, the improvements on sudoku are relatively modest, albeit consistent across the dataset. On all cell datasets, and grid_wrld, a superlinear growth is observed with `Neuro#`'s lead over `SharpSAT` growing as the problems get more difficult (moving right along the x axis). The problems of the islandi.i.d dataset all had the same model counts and they were isomorphic to one another. Because `Neuro#` operates on the graph of the problem, it was capable of utilizing this fact and solve all problems in the same number of steps. However we observe that `SharpSAT`'s performance is function of variable ids and clauses orderings of the input CNF, and thus isomorphic problems are solved in different number of steps. Lastly, on bv_expr, `Neuro#` does better almost universally, except near the 100 problems mark and at the very end.

Dataset	# vars	# clauses	Random	SharpSAT	Neuro#
sudoku(9, 25)	182	3k	338	220	**195(1.1x)**
n-queens(10, 20)	100	1.5k	981	466	**261(1.7x)**
sha-1(28)	3k	13.5k	2,911	52	**24(2.1x)**
island(2, 5)	1k	34k	155	86	**30(1.8x)**
cell(9, 20, 20)	210	1k	957	370	**184(2.0x)**
cell(35, 128, 110)	6k	25k	867	353	**198(1.8x)**
cell(49, 128, 110)	6k	25k	843	338	**206(1.6x)**
grid_wrld(10, 5)	329	967	220	195	**66(3.0x)**
bv_expr(5, 4, 8)	90	220	1,316	328	**205(1.6x)**
it_expr(2, 2)	82	264	772	412	**266(1.5x)**

Table 4.1: `Neuro#` generalizes to unseen i.i.d. test problems often with a large margin compared to `SharpSAT`.



(a) sudoku (b) n-queens (c) sha-1 (d) island

(e) cell(9) (f) cell(35) (g) cell(49)

(h) grid_wrld (i) bv_expr (j) it_expr

Figure 4.2: Cactus Plot – `Neuro#` outperforms `SharpSAT` on all i.i.d benchmarks (lower and to the right is better). A cut-off of 100k steps was imposed though both solvers managed to solve the datasets in less than that many steps.

Upward Generalization We created instances of larger sizes (up to an order of magnitude more clauses and variables) for each of the datasets in Section 4.3. We took the models trained from the previous i.i.d. setting and directly evaluated on these larger instances without further training. The evaluation results in Table 4.2 show that `Neuro#` generalized to the larger instances across all datasets and in almost all of them achieved substantial gains compared to the baseline as we increased the instance sizes.

Dataset	# vars	# clauses	Random	SharpSAT	Neuro#	
sudoku(16, 105)	1k	31k	7,654	2,373	**2,300**	**(1.03x)**
n-queens(12, 20)	144	2.6k	31,728	12,372	**6,272**	**(1.9x)**
sha-1(40)	5k	25k	15k	387	**83**	**(4.6x)**
island(2, 8)	1.5k	73.5k	1,335	193	**46**	**(4.1x)**
cell(9, 40, 40)	820	4k	39,000	53,349	**42,325**	**(1.2x)**
cell(35, 192, 128)	12k	49k	36,186	21,166	**1,668**	**(12.5x)**
cell(35, 256, 200)	25k	102k	41,589	26,460	**2,625**	**(10x)**
cell(35, 348, 280)	48k	195k	54,113	33,820	**2,938**	**(11.5x)**
cell(49, 192, 128)	12k	49k	35,957	24,992	**1,829**	**(13.6x)**
cell(49, 256, 200)	25k	102k	47,341	30,817	**2,276**	**(13.5x)**
cell(49, 348, 280)	48k	195k	53,779	37,345	**2,671**	**(13.9x)**
grid_wrld(10, 10)	740	2k	22,054	13,661	**367**	**(37x)**
grid_wrld(10, 12)	2k	6k	100k≤	93,093	**1,320**	**(71x)**
grid_wrld(10, 14)	2k	7k	100k≤	100k≤	**2,234**	(–)
grid_wrld(12, 14)	2k	8k	100k≤	100k≤	**2,782**	(–)
bv_expr(7, 4, 12)	187	474	35,229	5,865	**2,139**	**(2.7x)**
it_expr(2, 4)	162	510	51,375	7,894	**2,635**	**(3x)**

Table 4.2: `Neuro#` generalizes to much larger problems than what it was trained on, sometimes achieving orders of magnitude improvements over `SharpSAT`. Episodes are capped at 100k steps, which skews averages of `SharpSAT` downwards.

Figure 4.3 shows this effect for multiple sizes of cell(49) and grid_wrld by plotting the percentage of the problems solved within a number of steps. The superlinear gaps get more pronounced once we remove the cap of 10^5 steps, i.e., let the episodes run to completion. In that case, on grid_wrld(10, 12), `Neuro#` took an average of 1,320 branching decisions, whereas `SharpSAT` took 809,408 (613x improvement).

On some datasets, namely cell(49) and grid_wrld, `Neuro#`'s lead over `SharpSAT` becomes more pronounced as we test the upwards generalization (using the model trained on smaller instances and testing on larger ones). Cactus plots of Figure 4.4 show this effect clearly for these datasets. In each figure, the i.i.d. plot is included as a reference on the left and on the right the plots for test sets with progressively larger instances are depicted.

Figure 4.5 compares the percentage of the problems solvable by `SharpSAT` vs. `Neuro#` under a given number of steps. Notice the robustness of the learned model in cell(35&49) and grid_wrld. As these datasets get more difficult, `SharpSAT` either takes more steps or completely fails to solve the problems altogether, whereas `Neuro#` relatively sustains its performance.

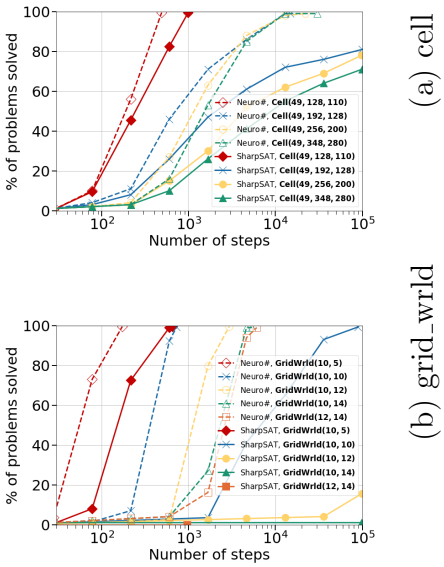
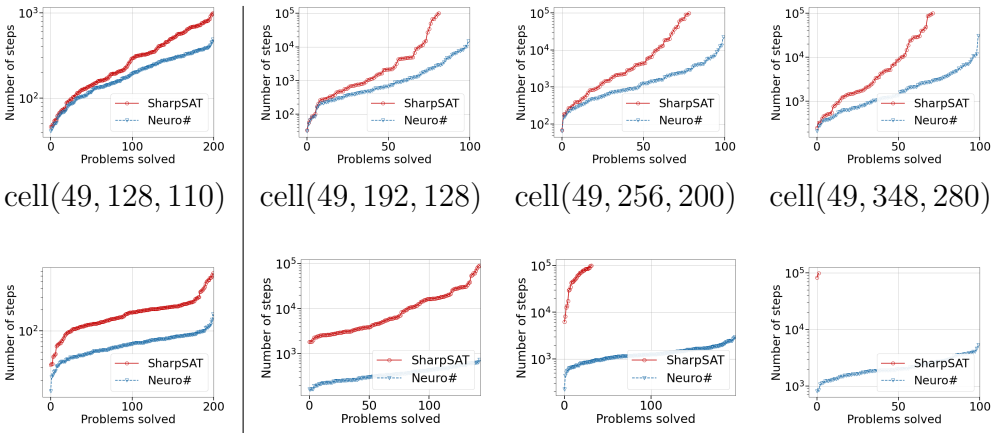


Figure 4.3: `Neuro#` generalizes well to larger problems. Compare the robustness of `Neuro#` vs. `SharpSAT` as the problem sizes increase. Solid and dashed lines correspond to `SharpSAT` and `Neuro#`, respectively. All episodes are capped at 100k steps.

Wall-Clock Improvement Given the scale of improvements on the upward generalization benchmark, in particular cell(49) and grid_wrld, we measured the runtime of `Neuro#` vs. `SharpSAT` on those datasets (Figure 4.6). On both problems we observe that as `Neuro#` widens the gap in the number of steps, it manages to beat `SharpSAT` in wall-clock. Note that the query overhead could still be greatly reduced in our implementation through GPU utilization, loading the model in the solver's code in C++ instead of making out-of-process calls to Python, etc.

Model Interpretation

Formal analysis of the learned policy and its performance improvements is difficult, however we can form some high-level conjectures regarding the behaviour of `Neuro#` by how well it decomposes the problem. To interpret `Neuro#`'s policy at the original problem-level we focus on cell. This is because encodings to CNF can in general be quite removed from the original problem domain. Consider grid_wrld: the problems are encoded from an MDP to a state machine, then to a circuit, and finally to CNF, as many new variables are created along this process. In contrast, cell has a straightforward encoding that directly relates the CNF representation to an easy-to-visualize evolution grid which coincides with the standard representation of Elementary Cellular Automata. Our conjecture was that the model learns to solve the problem from the bottom up. It translates to a policy that starts from the known state T (bottom row) and tries to "guess" the preimage, row by row from the bottom up



cell(49, 128, 110) cell(49, 192, 128) cell(49, 256, 200) cell(49, 348, 280)

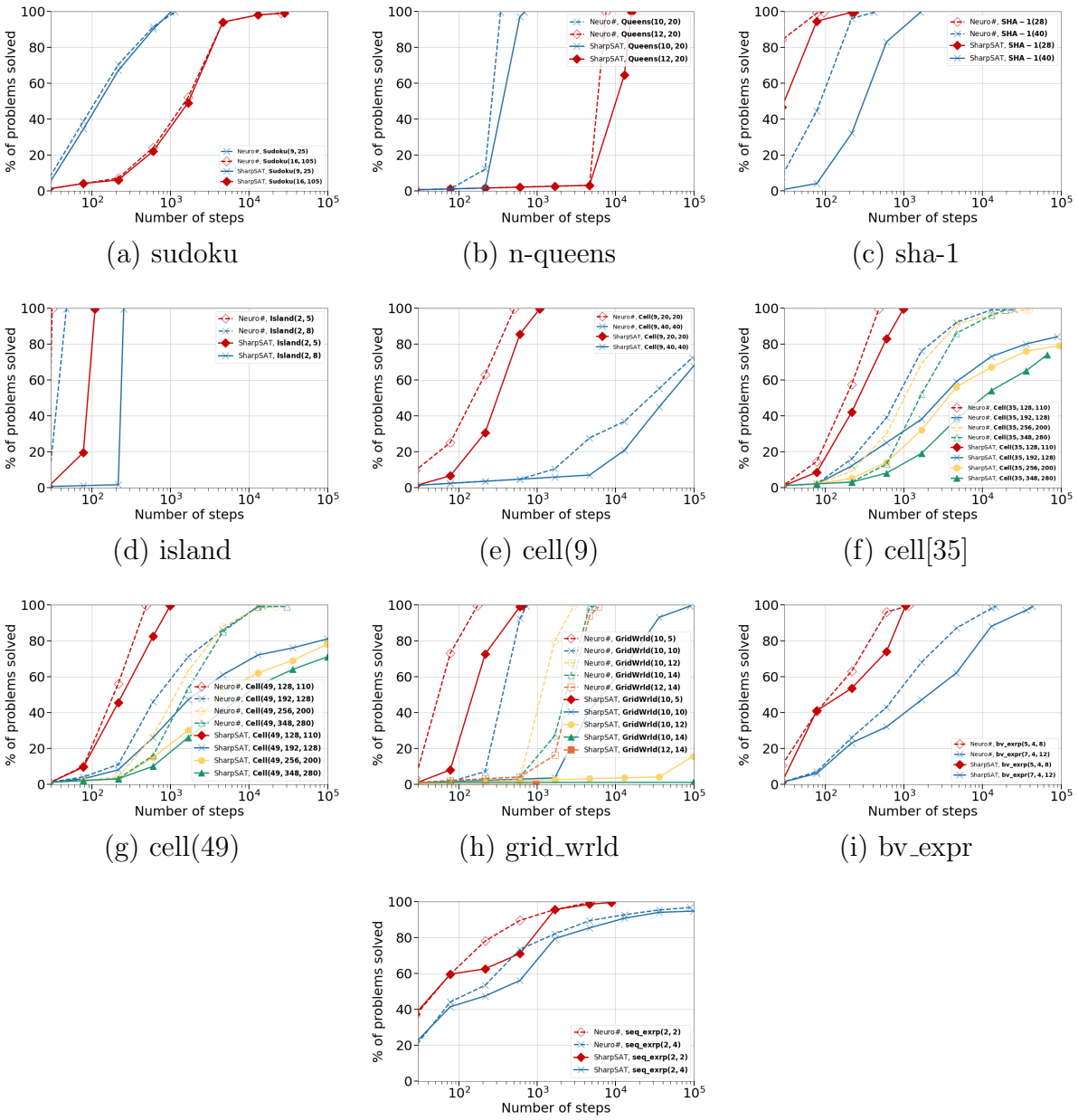
grid_wrld(10, 5) grid_wrld(10, 10) grid_wrld(10, 12) grid_wrld(10, 14)

(a) (b)

Figure 4.4: Cactus Plot: `Neuro#` maintains its lead over `SharpSAT` on larger datasets (lower and to the right is better). A cut-off of 100k steps was imposed. **(a)** i.i.d. generalization; **(b)** Upward generalization of the model trained on cell(49, 128, 110) (top row) and grid_wrld(10, 5) (bottom row) over larger datasets.

through variable assignment. The point being that different preimages can be computed *independently* upwards, and indeed, this is how a human would approach the problem.

Heat maps in Fig. 4.7 (c & d) depict the behaviour under `SharpSAT` and `Neuro#` respectively. The heat map aligns with the evolution grid, with the terminal state T at the bottom. The hotter coloured cells indicate that, on average, the corresponding variable is branched on earlier by the policy. The cooler colours show that the variable is often selected later or not at all, meaning that its value is often inferred through UP either initially or after some variable assignments. That is why the bottom row T and adjacent rows are completely dark, because they are simplified by the solver before any branching happens. We show the effect of this early simplification on a single formula per dataset in Figure 4.7 (b). Notice that in cell(35&49) the simplification shatters the problem space into few small components (dark triangles), while in cell(9) which is a more challenging problem, it only chips away a small region of the problem space, leaving it as a single component. Regardless of this, as conjectured, we can see a clear trend with `Neuro#` focusing more on branching early on variables of the bottom rows in cell(9) and in a less pronounced way in cell(35&49). Moreover, as more clearly seen in the heatmap for the larger problem in Figure 4.8, `Neuro#` actually branches early according to the pattern of the rule (!).



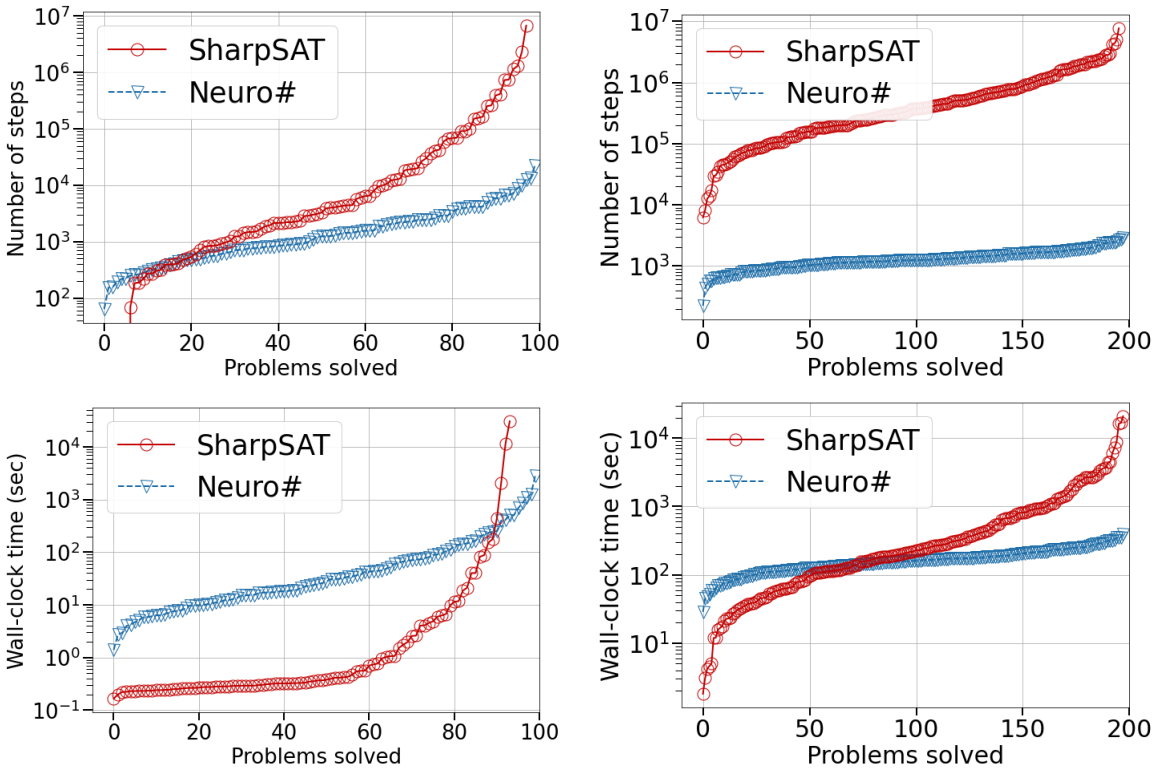
(a) sudoku (b) n-queens (c) sha-1

(d) island (e) cell(9) (f) cell[35]

(g) cell(49) (h) grid_wrld (i) bv_expr

(j) it_expr

Figure 4.5: `Neuro#` generalizes well to larger problems on almost all datasets (higher and to the left is better). Compare the robustness of `Neuro#` vs. `SharpSAT` as the problem sizes increase. Solid and dashed lines correspond to `SharpSAT` and `Neuro#`, respectively. All episodes are capped at 100k steps.



(a) cell(49, 256, 200) (b) grid_wrld(10, 12)

Figure 4.6: Cactus plots comparing `Neuro#` to `SharpSAT` on cell and grid_wrld. Lower and to the right is better: for any point t on the y axis, the plot shows the number of benchmark problems that are individually solvable by the solver, within t steps (top) and seconds (bottom).

Ablation Study

Semantic Features We tested the degree to which the "time" feature contributed to the upward generalization performance of grid_wrld. We compared three architectures with `SharpSAT` as the baseline: **1.** *GNN*: The standard architecture proposed in Section 4.2, **2.** *GNN+Time*: Same as *GNN* but with the variable embeddings augmented with the "time" feature and **3.** *Time*: Where no variable embedding is computed and only the "time" feature is fed into the policy network. As depicted in Figure 4.9, we discovered that "time" is responsible for most of the improvement over `SharpSAT`. This fact is encouraging, because it demonstrates the potential gains that could be achieved by simply utilizing problem-level data, such as "time", that otherwise would have been lost during the CNF encoding.

We've also tested the effect of adding the original spatial information of the grid, in form of an "image", which we processed with a convolutional network, and then added to the input given to the GNN as an independent "blob" of information (much like the global solver

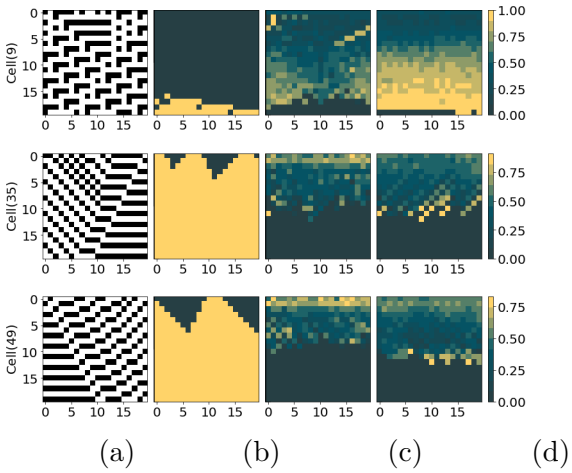


Figure 4.7: Contrary to `SharpSAT`, `Neuro#` branches earlier on variables of the bottom rows. (a) Evolution of a bit-vector through repeated applications of Cellular Automata rules. The result of applying the rule at each iteration is placed under the previous bit-vector, creating a two-dimensional, top-down representation of the system's evolution; (b) The initial formula simplification on a *single* formula. Yellow indicates the regions of the formula that this process prunes; (c & d) Variable selection ordering by `SharpSAT` and `Neuro#` averaged over the entire dataset. Lighter colours show that the corresponding variable is selected earlier on average.

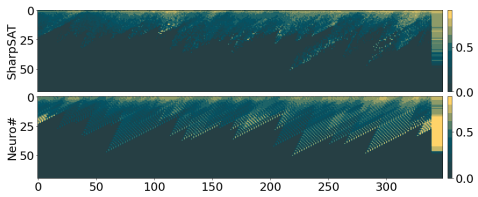


Figure 4.8: Full-sized variable selection heatmap on dataset cell$(35, 348, 280)$. We show the 99th percentile for each row of the heatmap in the last column.

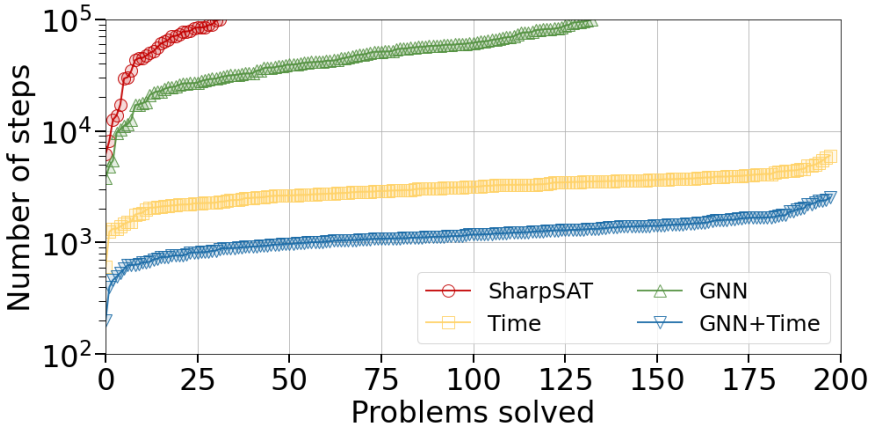


Figure 4.9: Ablation study on the impact of the "time" feature on upward generalization on grid_wrld$(10, 12)$.

information from the previous chapter). We've discovered that unlike the time feature, the additional spatial information didn't contribute, and models trained with or without it were virtually indistinguishable.

Variable Score We mentioned in Section 4.1 that `SharpSAT`'s default way of selecting variables is based on the VSADS score which incorporates the number of times a variable v appears in the current sub-formula, and (a function of the) number of conflicts it took part in. At every branching juncture, the solver picks a variable among the ones in the current component with maximum score and branches on one of its literals (see Algorithm 3). As part of our efforts to improve the performance of our model, we performed an additional ablation study over that of Section 4.4. Concretely, we measured the effect of including the variable scores in our model. We start with a feature vector of size 2 for each literal, and pass it through an MLP of dimensions $2 \times 32 \times 32$ to get the initial literal embedding. We tested on cell$(49, 256, 200)$ and grid_wrld$(10, 12)$ datasets (Figures 4.10 & 4.11). For both datasets, the inclusion of the variable scores produced results inferior to the ones achieved without them! This is surprising, though consistent with what was observed in the QBF case [78]. It is an intriguing result that repeats itself in two different domains, and leads to the following two conjectures - 1. VSIDS Activity scores are simply not that useful for CDCL based solvers that are not SAT solvers. 2. In a learning context, they provide a "cheap", short-sighted and easy signal to latch on to, which results in convergence to inferior models.

Comparison with Ganak and Centrality It was recently shown [25] that branching according to the centrality scores of variable nodes of the CNF graph leads to significant performance improvements. We obtained their centrality enhanced version of `SharpSAT` and compared it with `Neuro#` trained on specific problem family. We found that, although

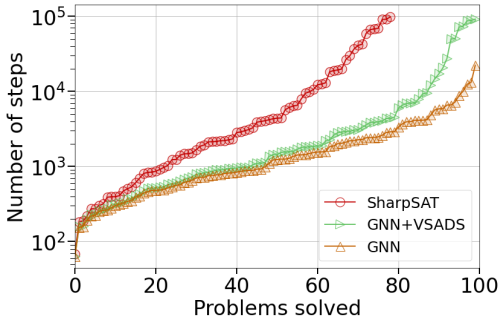


Figure 4.10: Cactus Plot – Inclusion of VSADS score as a feature hurts the upward generalization on cell$(49, 256, 200)$ (lower and to the right is better). A termination cap of 100k steps was imposed on the solver.

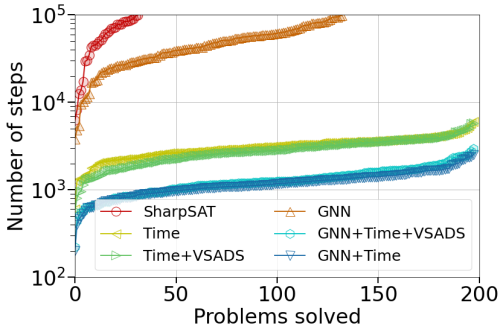
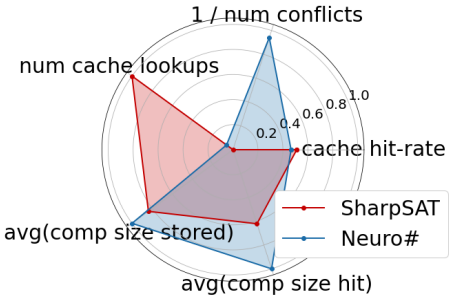
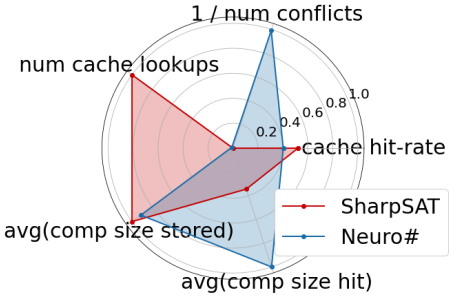


Figure 4.11: Cactus Plot – Ablation study on the impact of the "time" and VSADS features over upward generalization on grid_wrld$(10, 12)$ (lower and to the right is better). A termination cap of 100k steps was imposed on the solver.

centrality enhanced `SharpSAT`, `Neuro#` retained its orders of magnitude superiority over it. This indicates that whatever structure `Neuro#` is exploiting from the graph, it is not exclusively centrality. Also we compared the performance of `Neuro#` against the state-of-the-art *probabilistic* model counter Ganak [125]. Note that this solver performs the easier task of providing a model count that is only *probably correct within a given error tolerance*. Thus to make the comparison more fair we set the error tolerance of Ganak to a small value of 10^{-3} and observed that its performance was again inferior to `Neuro#`. An interesting future direction would be to investigate if our method could also be used to customize Ganak's heuristics.



(a) cell$(49, 256, 200)$



(b) grid_wrld$(10, 12)$

Figure 4.12: Radar charts showing the impact of each policy across different solver-specific performance measures.

Trained Policy's Impact on Solver Performance Measures

In this section we analyze the impact of `Neuro#` on solver's performance through the lens of a set of solver-specific performance measures. These measures include: **1.** Number of conflict clauses that the solver encounters while solving a problem (NUM CONFLICTS), **2.** Total (hit+miss) number of cache lookups (NUM CACHE LOOKUPS), **3.** Average size of components stored on the cache (AVG(COMP SIZE STORED)), **4.** Cache hit-rate (CACHE HIT-RATE) and **5.** Average size of the components that are successfully found on the cache (AVG(COMP SIZE HIT)).

A conflict clause is generated whenever the solver encounters an empty clause, indicating that the current sub-formula has zero models. Thus the number of conflict clauses generated is a measure of the amount of work the solver spent traversing the *non-solution space* of the formula. Cache hits and the size of the cached components, on the other hand, give an indication of how effectively the solver is able to traverse the formula's *solution space*. In particular, when a component with k variables is found in the cache (a cache hit) the solver does not need to do any further work to count the number of solutions over those k variables. This could potentially save the solver $2^{O(k)}$ computations. This $2^{O(k)}$ worst case time is rarely occurs in practice; nevertheless, the number of cache hits, and the average size of the

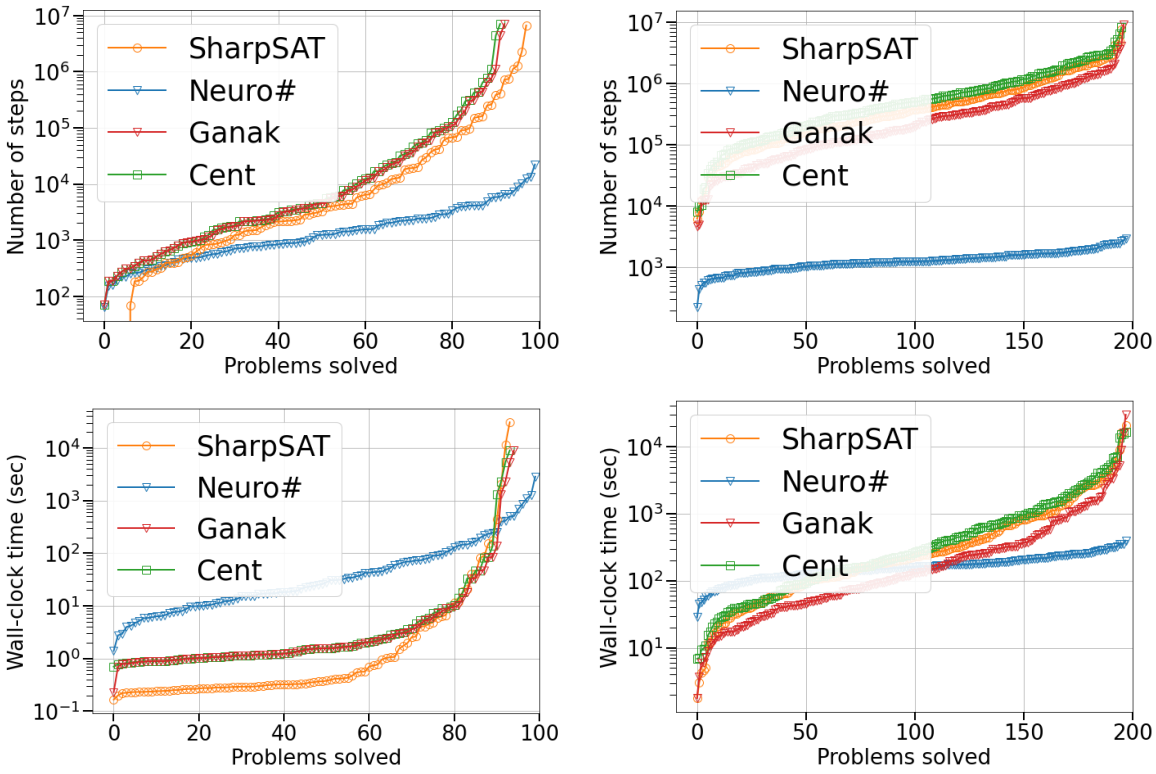
components in those cache hits give an indication of how effective the solver is in traversing the formula's solution space. Additional indicators of solver's performance in traversing the solution space are the number of components generated and their average size. Every time the solver is able to break its current sub-formula into components it is able to reduce the worst case complexity of solving that sub-formula. For example, when a sub-formula of m variables is broken up into two components of k_1 and k_2 variables, the worst case complexity drops from $2^{O(m)}$ to $2^{O(k_1)} + 2^{O(k_2)}$. Again the worst case rarely occurs (as indicated by the fact that #SAT solvers do not display worst case performance on most inputs), so the number of components generated and their average size provide only an indication of the solver's effectiveness in traversing the formula's solution space.

In Figure 4.12 we plot these measures for cell$(49, 256, 200)$ and grid_wrld$(10, 12)$. Looking at the individual performance measures, we see that the `Neuro#` encounters fewer conflicts (larger 1/NUM CONFLICTS), meaning that it is traversing the non-solution space more effectively in both datasets. The cache measures, indicate that the standard heuristic is able to traverse the solution space a bit more effectively, finding more components (NUM CACHED LOOKUPS) of similar or larger average size. However, `Neuro#` is able to utilize the cache as efficiently (with comparable cache hit rate) while finding components in the cache that are considerably larger than those found by the standard heuristic. In sum, the learnt heuristic finds an effective trade-off of learning more powerful clauses, with which the solver can more efficiently traverse the non-solution space, at the cost of a slight degradation in its efficiency traversing the solution space. The net result in an improvement in the solver's run time.

Figure 4.13 shows the results of running Bliem and Järvisalo [25]'s centrality-based solver (henceforth "Cent") and GANAK on the two datasets that we tested wall-clock time on (i.e., cell and grid_wrld). We observe that `SharpSAT`, Cent and GANAK behave more or less in the same performance regime, whereas `Neuro#` deviates from the pack and emits the superlinear performance on the step counts. This results in wall-clock improvements for `Neuro#`, which again happens as the problems get more and more difficult.

Discussion

We observed varying degrees of success on different problem families. This raises the question of what traits make a problem more amenable for improvement via `Neuro#`. One of the main contributing factors is the model's ability to observe similar components many times during training. In other words, if a problem gets shattered into smaller components either by the initial simplification (e.g., UP) or after a few variable assignments, there is a high chance that the model fits to such distribution of components. If larger problems of the same domain also break down into similar component structures, then `Neuro#` can generalize well on them. We visualized this "shattering" phenomena for the cell dataset via heat maps, as can be seen in Fig. 4.14. This might explain why sequential problems like grid_wrld benefit from our method, as they are naturally decomposable into similar iterations and addition of the "time" feature demarcates the boundaries between iterations even more clearer.

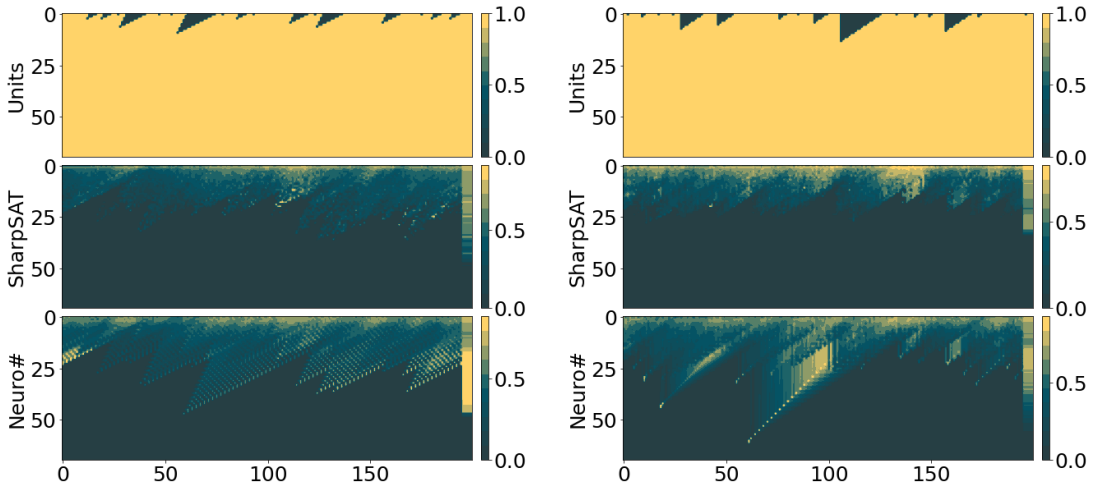


(a) cell(49, 256, 200) (b) grid_wrld(10, 12)

Figure 4.13: Orders of magnitude reduction in the number of branching steps which translates to wall-clock improvements as problems get harder. Note, as explained in the text, Python startup overhead skews results on easy problems.

4.5 Conclusions

We've shown in this chapter that our "neural guidence" approach can improve on a state of the art exact model counter, sometimes in several orders of magnitude, and have conjectured and experimentally verified that the black-box Evolutionary Strategies optimization algorithms achieve superior results when compared to RL algorithms in our domain. We've introduced the concept of "Semantic Features", elements of the problem that are present at the level of the original problem domain, yet are lost during conversion to CNF, and how keeping them around can improve the learned heuristics. Finally, we've shed some light onto what one such learned heuristic actually does, interpreting its behaviour in a the specific problem domain of cellular automata.



(a) cell$(35, 192, 128)$ (b) cell$(49, 192, 128)$

Figure 4.14: Clear depiction of `Neuro#`'s pattern of variable branching. The "Units" plots show the initial formula simplification the solvers. Yellow indicates the regions of the formula that this process prunes. Heatmaps show the variable selection ordering by `SharpSAT` and `Neuro#`. Lighter colours show that the corresponding variable is selected earlier on average across the dataset.

Chapter 5

Challenges in end-to-end learning for SAT

In this shorter chapter we describe some of our attempts to adapt our method to SAT solvers. Unlike the work described in previous chapters we achieved very little success in this context. We describe some of the challenges we faced and what we discovered, for the benefit of those who may research it after us, so that they could avoid at least a few pitfalls.

From a practical point of view, SAT is without doubt the most important CSP. Arguably, it's due to its relative simplicity. The fact that SAT is relatively simple has two implications. First, SAT solvers scale to solve problems that are orders of magnitude larger (in terms of number of variables) than what other more complex solvers can manage. And second, it is used within some more complex solvers, so improvement to SAT can indirectly improve other solvers.

In the early days of personal computing, a concept that developed around microprocessor architecture design was 'speed demons' vs. 'brainiacs'. Originally, it referred to the trade-off between clock speed and instructions per cycle, but the metaphor can be carried to the domain of Constraint Solvers (CS). A SAT solver is a speed demon compared to a QBF solver like Cadet. What this means is that while Cadet takes an average of less than 10^4 branching decisions per second, the popular solver Minisat takes 10^5-10^6. This difference in the frequency of the branching decisions has drastic implications with regard to the implementation of our method for SAT. We start this short chapter with outlining these implications and the challenges they pose, and then go over which other approaches in the literature achieved a degree of success. We then present two possible solution, learning Clause Deletion heuristics, and scheduling CSIDS activities "intervention", and describe our partial results.

5.1 The Problem with Speed Demons

One of the main strengths of our method is that it is in some way 'end-to-end'. Both optimization algorithms we use, Policy Gradient (PG) and Evolutionary Strategies (ES),

require episodes to be **completed** at least some of the time in order to learn anything. Remember, the reward structure we use is somewhat akin to solving a maze - a large positive reward for solving, and a small negative reward (penalty) for every step taken along the way. An episodes will be aborted after a fixed number of decisions, without the positive winning bonus. All aborted episodes therefore achieve the exact same(negative) reward. If a problem is always aborted regardless of what actions (branching decisions) the policy takes, we gain absolutely no information from it, and might as well not have included it in training.

In the case of QBF or model counting, we could train on "small" problems, taking anywhere between 30 and 1000 decisions, which took at most several seconds to complete (or sometimes abort) through our training loop. The heuristics we learned scaled to be effective on problems that took up to 100k decisions. In the case of SAT, the math doesn't add up. We can still only train on problems of up to around 1000 decisions - the bottleneck is not in the solver, but rather on the training framework side, which computes forward and backward passes for a NN. But for SAT, even problems that are considered small (but still resemble "real" problems) often require many thousands of decisions. At test time, such a learned heuristic would have to scale not to 100k decisions, but to up to 10^8 total decisions.

While training constraints are the main problem, there are a few others worth mentioning. The time it takes to process the CNF graph through a NN depends on its size, regardless of whether it came from a SAT problem or a QBF. This means that the ratio between the running time of the original and the learned heuristic depends for the most part of the absolute running time of the original heuristic. In SAT, where branching decisions are orders of magnitude faster, the learned heuristic (when measured in wall clock time, not number of decisions) is comparatively that much worse. Another issue, though less important, is the absolute size of the CNF graphs. As mentioned above, SAT solvers deal with graphs that are orders of magnitude larger than the other CS we've seen. This means our NN policy has to process this larger graph. This is not a huge issue in comparison though, because processing the graph can be parallelized to multiple GPUs by partitioning the graph. It adds complexity, but is doable.

5.2 Existing Approaches

The only other approach we are aware of that learns heuristics in SAT in an end-to-end manner is of Kurin et al. [75], where they effectively use a version similar to our "naive" approach. They get around the training and test time constraints by complementing VSIDS rather than replacing it. VSIDS is known to take some time to learn reasonable activity scores for variables, which means its initial decisions are less informed. Their solution therefore was to only take the first N branching decisions (With N on the order of 10-1000), and then leave the rest to the standard VSIDS. They show improvement in the number of decisions for some 3-SAT and graph coloring problems, all relatively small (less than 1000 variables). However, the weakness here is precisely the fact that the NN has only a limited effect on the solver, it

is uncertain precisely how useful are the first decisions taken within a split second in guiding a solution process that can take more than an hour, with hundreds of millions of decisions.

The other approaches that address SAT solvers invariably use some sort of a proxy, through supervised learning, thereby getting around the requirement for completing each episode. NeuroCore [120] takes a large training set of unsatisfiable problems, and trains a GNN architecture on predicting which variables take part in the unsatisfiable core of the problem, extracted from its unsatisfiability proof.[1] At run-time, it periodically refocuses the solver on the (predicted) unsat core variables by directly setting activity values. This method is based on the domain intuition that it is better to branch on variables that lead to conflicts, and the authors have shown improvement on a single family of problems from the 2018 SAT competition. The main advantage of this method is the (relative) ease of training. Problems are solved and the unsat cores are found in advance, and from there its standard supervised classification task.

The disadvantages of this method are the standard ones for proxies - instead of directly learning to optimize the target metric (wall clock time) or a relatively close proxy (number of propagations, number of branching decisions) it relies on the intuition that it's best to branch on variables that lead to conflicts. This makes sense, but only to a point. It is known for example that not all conflicts are equally useful. Satisfiable problems don't have unsat cores at all, and it is not clear that predicting non-existing unsat cores is a good way of finding variables likely to lead to conflicts. Even in an unsat problem there may be different unsat cores, and the model was trained on just one of them (for a given problem). In short, we're directing the learning agent towards some pre-known domain intuitions rather than try to learn a heuristic from scratch.

A similar method using more classical multi-armed bandit techniques is presented in Liang et al. [80], where they modify VSIDS to learn a "learning rate" for variables, roughly predicting how many conflicts they will participating on producing, cumulatively, over the solving process.

5.3 Method - Clause Deletion Heuristic

In light of these technical constraints, the approach we've taken is to learn a heuristic that is queried at a lower frequency throughout the SAT solving process.

Clause Deletion in SAT

As mentioned earlier, choosing branching variables is not the only heuristic is a CS. One of the other heuristics that has a large influence on solver performance is the *Clause Deletion Heuristic*. As we recall, Conflict-Driven Clause Learning (CDCL) works by learning new clauses that summarize conflicts and prevent the solver from searching down entire sub-trees.

[1] An unsatisfiability proof in SAT is a sequence of resolutions (clause learning is a form of resolution) leading all the way to an empty clause.

Every clause learned this way is added to the "Learned Clauses Database". However, one complication which we glossed over before is that the solver cannot afford to just keep learning new clauses and adding them indefinitely. First, their number is too great. Even if the solver wanted to keep them all, it would soon run out of memory. But beyond that, even with infinite memory it is not advisable to keep all learned clauses. More clauses cause more Boolean Constraint Propagation (BCP), and not all of those propagations are productive (note, these issues are related to why encouraging more conflicts is not always desirable). Not all learned clauses are of the same quality, and we want to keep only the useful ones. And so, SAT solvers periodically delete some of the learned clauses, a process that is called *Garbage Collection*.

Literal Block Distance As our baseline we take the LBD heuristic [9], one of most popular heuristics used to delete clauses, and implemented in Minisat and Glucose. It is queried on a schedule based on the number of learned clauses, and crucially for us, in practice at a much lower frequency than the branching heuristic, on the order of seconds. For every learned clause, it derives a single integer, the LBD score, and then drops the half of the learned clauses with LBD above the median.[2] To understand the LBD score, recall that each assigned variable is also assigned a *decision level*, which is essentially the level in which it was assigned. The variable branched on and all variables assigned in the resulting BCP are assigned the same decision level, which is then increased by one. When reaching a conflict and backtracking, it decreases. The LBD score of the a learned clause is the number of different decision levels its variables have, and in Audemard and Simon [9] the authors show empirically that clauses with low LBD scores are more useful, in that they take place in more propagations and conflicts.

Action Space for Clause Deletion

Our approach is then to employ a similar method to the one that worked for branching heuristic, but instead of ranking literals, our model now ultimately has to output a Boolean decision for every learned clause, telling the solver whether to keep or drop it.

Naive Version - Independent Decisions

The most straightforward solution is a minor modification of the architecture we've seen before. Recall, the Graph Neural Network (GNN) we used computed variable (or literal) embeddings through multiple iterations composed of two half-iterations each, from variables (literals) to clauses and back to variables (see Fig. 2.13). While we did not use them directly for branching heuristic, the GNN also computes clause embeddings. We can just take the latest learned clauses embeddings, and pass them through a policy network that outputs two logits per clause (a simple Multi-Layered Perceptron (MLP) will do this), which translate

[2]To be precise, half of the clauses with LBD score greater than 2, clauses scored 2 are never dropped

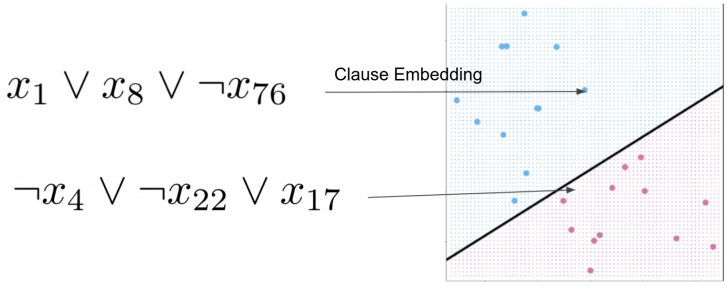


Figure 5.1: Classifying embedded clauses with the Hyperplane approach

to a Bernoulli distribution $B(1, p_c)$ per learned clause c. The parameter p_c is interpreted as the probability the model thinks the clause c should be kept. During training, the Policy Gradient algorithm (see Sec. 3.3) samples a Boolean decision for each clause, and sends back to the solver a Boolean array of decisions as the action. The environment, a modified glucose solver, keeps or drops the clauses according to the decisions array.

This solution could work in theory, but it has a serious flaw. As we discussed before in Sec. 4.2, PG algorithms explore the solution space implicitly by sampling on the action space. In this naive solution, it means we're sampling for each clause independently of all the others. For N clauses, this means the action space is the discrete space 2^N, and independent Bernoulli distributions means we're flipping N coins per action. When N is in the 10^3-10^4 range, this is a lot of randomness to inject into the estimator, which translates to variance, and leads to extremely slow convergence, if at all. Note, it is in fact entirely equivalent to replacing every action with N separate actions, one per learned clause, which would leave us with episodes with length in the tens of thousands of time steps. Nothing is gained by 'coalescing' the coin flips together into one action.

Dealing with Randomness

Evidently, care must be taken when we define the action of our policy. Let us consider again what doesn't work in the naive version. When we sample a decision for each clause independently, it also means that two identical clauses (there are no identical clauses, but say, nearly identical) with identical embeddings can result in different decisions. If c_1, c_2 have close embeddings, that means $p_{c_1} \approx p_{c_2}$. If they happen to be close to 0.5, PG is just as likely to sample similar decisions for them as it is to sample a different one. This is not how we want the clause embeddings to work. Surely similar embeddings should be more likely to result in equal decisions, regardless of what that decision is.

What we're actually looking for is a classifier. The GNN embeds each clause into n-dimensional space. As an action then, the agent should output a separating (affine) Hyperplane, as in Fig. 5.1. Of course, PG must sample in order to explore, and since our action *is* the Hyperplane, we have to sample the Hyperplane. The way to do it is to have the

model output a vector in $\mathbb{R}^{n+1}$ (where n is the clause embedding dimension), and sample from a normal distribution around each coordinate.[3] Note that this way we bound the randomness we're injecting - n-dimensional normal distribution, independent of the number of learned clauses.

Implementation and Empirical Results

We implemented our model using a similar architecture to that in Sec. 4.2, with Glucose as an environment. We included a global Solver State with some statistics about the clause database, and 5 features per variable, including its activity and LBD score. For our tests, we used the Cellular automata dataset from SATCOMP 2018 described before in Sec. 4.3.

We couldn't get the training of this model to converge, meaning, even when staying on the training set, before generalization, the model didn't improve on the standard LBD heuristic. Taking a couple of steps back, we decided to simplify the problem, reduce the degrees of freedom of the policy. Specifically, we dropped the GNN and used the variable features directly, and still were unable to see any improvement even on the training set itself.

LBD Based Policies

Finally, we decided to use the LBD heuristic itself for further simplification, and learned to output a dynamic LBD threshold. That is, we created a model that, on every garbage collection, and based on the global solver state, learned to output two real numbers, a and b. The solver environment, for each clause c with LBD score L_c, computed the decision according to the sign of $aL_c + b$. For exploration, we sampled from a normal distribution around a, b. Note, this simplified version of the model can be thought of as the Hyperplane solution, where the embedding space is one-dimensional, and the single feature of a clause is its LBD score.

This simplified version was able to converge on the training set, but improving only very slightly on LBD. We found we could improve convergence time considerably by outputting a discrete probability over the possible (reasonable) LBD thresholds. Instead of outputting 2 numbers, we output 29 different numbers that are taken as logits for selecting (through sampling) an integer between 2 and 30, which is used as the LBD threshold. We found that a policy with discrete outputs for the threshold could explore the solution space better, and choose better thresholds. This is quite expected, because when using a continuous action by sampling from a normal distribution around some mean, the learning process only moves this mean continuously, and it can easily get stuck in a suboptimal minimum. The discrete version gets to quickly explore all possible LBD thresholds and converge on beneficial ones. Still, the gains over standard LBD were small, and no generalization to larger problems was observed.

[3] In practice our model now outputs a continuous action

5.4 Conclusions

When it comes to "speed demons" like SAT solvers, our method faces serious technical limitations. Learning a branching heuristic is infeasible, and it's necessary to consider other Heuristics with a lower query frequency, such as for clause deletion. But decisions for clause deletion come with their own constraints, especially with regards to size of the action space. We suggested a few ways to get around this constraint, but achieved only very limited success.

We feel that due to the technical constraints involved with SAT solving there is still a considerable distance to cross before an end-to-end method like ours could integrate with a modern solver, and it is still an open question whether this is possible.

Chapter 6

Conclusions and Further Work

6.1 Conclusion

We set out at the beginning of this thesis to research how to combine techniques from Deep Learning with modern Boolean Constraint Solvers (CS), thus allowing them to learn from experience. By the time it comes to a close, this thesis had made the following contributions:

- We presented for the first time[1] a Graph Neural Network based representation for logical formulas, matching the symmetries of such formulas, and demonstrated its applicability to the task through empirical experiments.
- We presented a method, based on the representation mentioned above, of automatically learning heuristics in CS for specific families of problems through training, using either Reinforcement Learning or Evolutionary Strategies. This method can be adapted to any solver which works with Boolean formulas, subject to some limitations.
- We implemented our method for two different modern solvers, the Quantified Boolean Formulas solver Cadet, and the Model Counter `SharpSAT`. Although both are based on DPLL, they solve different problems of different complexity classes, based on different intuitions. In both, we demonstrated that our method improves over the state of the art by up to an order of magnitude on challenging problems.
- We showed how to incorporate some higher-level domain information on top of the common CNF representation of a problem. We call them Semantic Features, and show that at least under some circumstances (transition systems) they can have a drastic effect on solving time.
- We presented an interpretation of the learned heuristics in the context of model counting. We look both at the solver's internal metrics, and from the point of view of a specific higher-level domain, of cellular automata.

[1] Independently, Selsam et al. [121] came up with a similar version of this architecture at about the same time.

- We open-sourced our code[2], a framework which supports integration with the solvers Cadet, `SharpSAT`, MiniSAT and Glucose through an open-AI compatible interface. On the RL side, it is compatible with the popular RLLIB framework.[3]

Limitations Our method works impressively well in solvers where the heuristic is relatively "heavy", a common property of both QBF and Model Counting (but also potentially of SMT and other non-boolean solvers). It is less suited to be used in SAT solvers, where the heuristic is quick and simple. We have described some approaches that to get around this problem, but so far they achieved partial success at best, and require further research.

This work lies at the intersection of several lines of research that have drawn interest recently. From the side of AI, while deep learning techniques have broken new ground over the last decade, researchers in the field repeatedly confront its limitations. The classic Symbolic AI of the past is making a comeback, and there are several different attempts to combine Symbolic and Neural learning in new ways, to the benefit of either or both. We believe the trend of narrowing the gap between Automatic Reasoning and Neural systems will continue.

Likewise, formal methods and modern ML will also continue to be integrated in both directions - ML methods that enhance classic formal reasoning, like our work, and in the opposite direction, formal methods that allow for verifying systems with ML in the loop. Finally, there are many tools and algorithms in research and industry solving "hard" problems in practice. We have already reviewed some of the research applying ML to learn heuristics in some of these tools, and we believe this trend too will continue, where NN will be used to replace hand-designed heuristics.

6.2 Further Work

We conclude with some notes on promising further directions for research that naturally arise from the work described in this thesis.

Architecture

We've generally discovered that the specific GNN architecture used is not very important, as in, all "reasonable" encoders, whether based on CVIG or CLIG representation, perform roughly equally well. Two directions we think are worthwhile to pursue regarding architecture are solver-recurrent models and transformers:

- GNN models could be said to already be recurrent, since (potentially) the same message-update transformation is applied a fixed number of times. However, this is not to be confused with recurrence over the MDP time steps (that is, observation and actions). This means that when the heuristic is called, it is given not only the current solver

[2]https://github.com/lederg/learningCNF
[3]https://docs.ray.io/en/master/rllib.html

state, but also the last N states and decisions. Formally, if the observation to the model on step i is O_i, we transform it to $(O_{i-N}, a_{i-N}, \ldots, O_{i-1}, a_{i-1}, O_i)$. The intuition is that this allows the heuristic to detect **dynamic** features of the CNF graph (such as the variable property of "having been recently branched on"), and implement policies that take advantage of it. If VSIDS can make use of a variables summarized "history," perhaps a learned heuristic can do the same.

- Ever since the Transformer architecture (see App. A.2) was introduced [147] in the context of machine translation, it has been used to improve on the state of the art on many language tasks - translation, language modeling, captioning, question answering, even proving mathematical theorems.

  The philosophy behind its use runs somewhat contrary to that of an "induced bias." A Transformer takes a fixed-size **set** of token embeddings and returns the same fixed-size set of processed embeddings. It induces no pre-defined structure on the input embeddings - in fact, when processing sequences the linear order over the sequence has to be specially encoded in the input embeddings (this is called "positional encoding" and was used in the original Transformer paper). The intuition is that with enough computation power and data, the transformer can detect the relations between any pair of embeddings on its own, regardless of the distance (within the sequence) between them. This added expressiveness comes at a cost though - the Transformer requires quadratic time (in the size of the input set N) compared to linear time of RNN.

  It can be adapted to process graphs by generalizing the concept of positional encoding. If the position of an element in a sequence is its ordinal number, the "position" of a node in a graph is its neighborhood. With this in mind, a Transformer can be added to our branching policy model by defining a maximum problem size N (Transformers can process sets that are smaller than the fixed size by masking), and using the original GNN outputs as positional encodings for the Transformer. The main advantage of such an architecture is that it goes beyond locality - every node gets to see a summary of the neighborhood of every other node, no matter how "far" they are in the graph. This could be especially useful for `Neuro#`. Due to its dependence on caching components, being able to detect isomorphic neighborhoods in distant parts of the graph can give the policy hints regarding the resulting partition into components. On the other hand, a quadratic computation time within the loop of a CS could simply be too slow for our purposes. Here too, `Neuro#` is the better candidate, because the model only processes components rather than the entire graph.

Optimization algorithms

We've used both RL and ES for optimization in the context of MDP. Because our main motivation was to prove the feasibility of the approach, its scope and limitations, we've implemented only the most basic, naive versions of those algorithms. It is known that modern

RL algorithms like PPO have a sample complexity that is 2-3 times better than vanilla policy gradient, and it should be relatively straight-forward to achieve faster training time using them..

Moreover, while for QBF we were not able to get Q-learning based algorithms to work, Some similar approaches for SAT [75] have demonstrated it can work in principle. Getting another RL algorithm to work roughly the same as Policy Gradient is not very exciting in itself, but it becomes important because estimating the Q function opens the way to implement elements from the AlphaZero algorithm [127]. Specifically, one element, the Monte-Carlo Tree Search (MCTS). MCTS is a technique that can be used in the settings of games like Chess and Go, and in fact whenever we have an environment with deterministic, known dynamics, that is relatively easy to simulate. The normal intuition behind MCTS is that by expanding the search tree and simulating the "game" (the solution process in a CS) we can find much better Q values - we project them from the future, sort of like a human player simulating the game in his head. Here it may be possible to exploit the fact that the DPLL algorithm is already back-tracking in its nature, and in practical implementations contains even more backtracking in the form of restarts.

Combine with Portfolio methods

Its been known that different solvers or different (manual) heuristics excel for different families of problems. Leveraging this fact, SATZilla [158] used a "portfolio" of different solvers (7 in the original paper), and learned a classifier sending each problem to its designated solver according to its features - a set of graph statistics, and other statistics collected over short "test runs" of DPLL and Local Search on the problem. For Industrial SAT, portfolio methods are highly successful.

The idea would be to do something similar, but with a portfolio of learned heuristics, which could be updated from time to time with new trained models, and in that respect is more flexible than a set of solvers. One could argue that instead of learning a classifier and N different heuristics separately we might as well throw more capacity at the original model and hope that if there's a useful classification the network will find it on itself. One could be right, but we feel this is a similar issue to the one we encountered in Sec. 3.2, why not just learn a heuristic that is "better on everything"? Some empirical results could guide us forward here.

Encoding-Solving Interaction & Semantic Features

The channel between encoding and solving

As we often repeated, CNF problems do not appear our of thin air, nor (usually) drawn from a distribution. They are encoded from problems in a higher-level domain, problems that are actually of interest to practitioners in industry and research. The pattern that has developed is using a CS as a backend service, seen as a blackbox from the user's perspective. Encoding

problems is a bit like compiling code into assembly - every programming language is different and has its own compiler, but they all output the same (type of) assembly instructions, which the assembler then takes care of.

From this perspective, The CNF representation is like the assembly language of formal methods. The encoding process itself is oftentimes hierarchical, implemented in layers. Arguably, it reflects the structure of the software tools humans use to design and generate these problems to begin with. An MDP (or even a non-markovian planning) problem like the grid_wrld of Sec. 4.3 is a good example. It was generated using PYAIGER [148], a set of tools for processing combinatorial and sequential circuits. The grid dynamics is encoded into a circuit, using common components such as boolean operators and vector arithmetic. A specification in LTL is transformed into a monitor circuit, and the two circuits are composed together. The resulting circuit is then *unrolled* into a sequential circuit representing an entire trajectory, and then encoded into an AIG circuit[4] by encoding each of its components in turn. From an AIG, the Tseitin transformation (or some version of it) takes us to CNF. Along the way, the language of the encoding becomes more uniform - from grid dynamics with different properties and specs, inputs and outputs, to a set of components, then to two Boolean gate type (And & Not), and finally to a list of clauses. This layered approach has a clear advantage - a solver needs only "know" about clauses and variables, not the endless mathematical structures defining problems in multitude of domains.

The disadvantage, though, is that the encoding processes losses a lot of information along the way. Information that could have been used to direct the branching heuristics. For example, in the same grid world example, knowing how action inputs are ordered in time, could give us a hint - it is better to branch on variables that represent the earlier inputs to the system. Indeed, when we added the single feature of the time step to each variable, we got orders of magnitude improvement.

But the time step is only one semantic feature - there are others that could be useful. Moreover, the layered approach to encoding means that semantic features from the lower levels can be added automatically, as part of the encoding framework. the grid example is but one example of monitoring an MDP, every such problem has a time step feature. Every problem that is encoded to a circuit and then to CNF has the same circuit-related semantic features that can be associated with the variables and can potentially provide hints to a branching policy. It is not difficult to augment pyAiger such that it produces "extended" CNF files, where each variable has extra annotated features such as time step, whether it is an input, latch, belongs to the MDP or to its monitor. If we go one level up, we can associate variables with the high-level component (such as a ripple-carry adder) that generated them. In effect, this approach widens the channel between the encoding and the solving process, expanding its "assembly language" beyond the bare CNF representation.

[4] And-Inverter-Graph is a common representation for circuits, which, as the name hints, is based only on 'And' and 'Not' gates.

Learning to Encode and Solve

Encoding and solving CS problems could be said to follow a pattern of co-evolution. We've seen how to learn heuristics that adapt to the data, but of course, the 'data' depends (in part) on the encoding. In practice there is often more than one possible encoding to CNF of a given domain, and users (of CS) make choices that essentially try to tune the encoding to the solver.

Why then not to do so jointly? An interesting direction is to train two networks together on the entire process. One network is incorporated into the encoder, taking decisions about how to encode. The other is the one we've studied here, incorporated into the solver and taking branching decisions. The same pipeline can be trained end-to-end with some minor modifications, where the input is the high-level problem, and the episode is the sequence of encoding decisions followed by the solving decisions.

Bibliography

[1] Stavros Adam et al. "No Free Lunch Theorem: A Review". In: May 2019, pp. 57–82. ISBN: 978-3-642-54671-6. DOI: 10.1007/978-3-030-12767-1_5.

[2] Miltiadis Allamanis et al. "A Survey of Machine Learning for Big Code and Naturalness". In: *CoRR* abs/1709.06182 (2017). arXiv: 1709.06182. URL: http://arxiv.org/abs/1709.06182.

[3] Miltiadis Allamanis et al. "Learning continuous semantic representations of symbolic expressions". In: *arXiv preprint arXiv:1611.01423* (2016).

[4] Rajeev Alur et al. "Syntax-Guided Synthesis". In: *Proceedings of the IEEE International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Oct. 2013, pp. 1–17.

[5] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. "Learning To Solve Circuit-SAT: An Unsupervised Differentiable Approach". In: *ICLR*. 2019.

[6] Carlos Ansótegui, Maria Bonet, and Jordi Levy. "On the Structure of Industrial SAT Instances". In: vol. 5732. Sept. 2009, pp. 127–141. ISBN: 978-3-642-04243-0. DOI: 10.1007/978-3-642-04244-7_13.

[7] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. "The community structure of SAT formulas". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2012, pp. 410–423.

[8] Gilles Audemard and Laurent Simon. "Glucose in the SAT 2014 Competition". In: *SAT COMPETITION 2014* (2014), p. 31.

[9] Gilles Audemard and Laurent Simon. "Predicting learnt clauses quality in modern SAT solvers". In: *Twenty-first International Joint Conference on Artificial Intelligence*. Citeseer. 2009.

[10] Lei Jimmy Ba, Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization". In: *CoRR* abs/1607.06450 (2016). arXiv: 1607.06450. URL: http://arxiv.org/abs/1607.06450.

[11] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. "Algorithms and Complexity Results for #SAT and Bayesian Inference". In: *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*. https://doi.org/10.1109/SFCS.2003.1238208. IEEE Computer Society, 2003, pp. 340–351. DOI: 10.1109/SFCS.2003.1238208.

[12] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. "Solving #SAT and Bayesian Inference with Backtracking Search". In: *J. Artif. Intell. Res.* 34 (2009). https://doi.org/10.1613/jair.2648. DOI: 10.1613/jair.2648.

[13] Mislav Balunovic, Pavol Bielik, and Martin Vechev. "Learning to Solve SMT Formulas". In: *NeurIPS*. Curran Associates, Inc., 2018. URL: http://papers.nips.cc/paper/8233-learning-to-solve-smt-formulas.pdf.

[14] Kshitij Bansal et al. "HOList: An Environment for Machine Learning of Higher-Order Theorem Proving". In: *CoRR* abs/1904.03241 (2019). arXiv: 1904.03241. URL: http://arxiv.org/abs/1904.03241.

[15] Clark Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Hans van Maaren, and Toby Walsh. IOS Press, 2009. Chap. 26, pp. 825–885.

[16] Peter W Battaglia et al. "Relational inductive biases, deep learning, and graph networks". In: *arXiv preprint arXiv:1806.01261* (2018).

[17] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. "Counting Models Using Connected Components". In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*. http://www.aaai.org/Library/AAAI/2000/aaai00-024.php. AAAI Press / The MIT Press, 2000, pp. 157–162.

[18] Hans-Georg Beyer and Hans-Paul Schwefel. "Evolution strategies - A Comprehensive Introduction". In: *Nat. Comput.* 1.1 (2002). https://doi.org/10.1023/A:1015059928466, pp. 3–52.

[19] Sahil Bhatia and Rishabh Singh. "Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks". In: *CoRR* abs/1603.06129 (2016). arXiv: 1603.06129. URL: http://arxiv.org/abs/1603.06129.

[20] Armin Biere. "Lingeling, plingeling, picosat and precosat at sat race 2010". In: *FMV Report Series Technical Report* 10.1 (2010).

[21] Armin Biere. "Resolve and expand". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2004, pp. 59–70.

[22] Armin Biere et al. "Bounded model checking". In: *Advances in computers* 58.11 (2003), pp. 117–148.

[23] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[24] Elazar Birnbaum and Eliezer L. Lozinskii. "The Good Old Davis-Putnam Procedure Helps Counting Models". In: *J. Artif. Intell. Res.* 10 (1999). https://doi.org/10.1613/jair.601, pp. 457–477.

[25] Bernhard Bliem and Matti Järvisalo. "Centrality Heuristics for Exact Model Counting". In: *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*. IEEE, 2019, pp. 59–63. URL: https://doi.org/10.1109/ICTAI.2019.00017.

[26] Samuel R Bowman, Christopher Potts, and Christopher D Manning. "Recursive neural networks can learn logical semantics". In: *arXiv preprint arXiv:1406.1827* (2014).

[27] Alfredo Braunstein, Marc Mezard, and Riccardo Zecchina. "Survey propagation: An algorithm for satisfiability". In: *Random Struct. Algorithms* 27 (Sept. 2005), pp. 201–226. DOI: 10.1002/rsa.20057.

[28] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[29] Hazel Victoria Campbell, A. Hindle, and J. Amaral. "Syntax errors just aren't natural: improving error reporting with language models". In: *MSR 2014*. 2014.

[30] Supratik Chakraborty et al. "Distribution-Aware Sampling and Weighted Model Counting for SAT". In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. AAAI Press, 2014, pp. 1722–1730.

[31] Xinyun Chen and Yuandong Tian. "Learning to Progressively Plan". In: *CoRR* abs/1810.00337 (2018). arXiv: 1810.00337. URL: http://arxiv.org/abs/1810.00337.

[32] Ziliang Chen and Zhanfu Yang. *Graph Neural Reasoning May Fail in Certifying Boolean Unsatisfiability*. 2019. arXiv: 1909.11588 [cs.LG].

[33] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[34] Karel Chvalovsky. "Top-Down Neural Model For Formulae". In: *ICLR*. 2019.

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

[36] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[37] J. Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[38] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://www.aclweb.org/anthology/N19-1423.

[39] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2003, pp. 502–518.

[40] Bethe Energy et al. "Bethe free energy, Kikuchi approximations and belief propagation algorithms". In: (Dec. 2000).

[41] Richard Evans et al. "Can Neural Networks Understand Logical Entailment?" In: *arXiv preprint arXiv:1802.08535* (2018).

[42] Maxime Gasse et al. "Exact Combinatorial Optimization with Graph Convolutional Neural Networks". In: 2019.

[43] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. "TacticToe: Learning to Reason with HOL4 Tactics". In: *EPiC Series in Computing* 46 (2017), pp. 125–143.

[44] Tomas Geffner and Hector Geffner. "Compact Policies for Fully Observable Non-Deterministic Planning as SAT". In: *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*. AAAI Press, 2018, pp. 88–96.

[45] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. "QUBE: A system for deciding quantified boolean formulas satisfiability". In: *International Joint Conference on Automated Reasoning*. Springer. 2001, pp. 364–369.

[46] Eugene Goldberg and Yakov Novikov. "BerkMin: A fast and robust SAT-solver". In: *Discrete Applied Mathematics* 155.12 (2007), pp. 1549–1561.

[47] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[48] A. Graves et al. "Symbolic Reasoning with Differentiable Neural Comput". In: 2016.

[49] K. He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[50] Kaiming He et al. *Deep residual learning for image recognition. CoRR abs/1512.03385 (2015)*. 2015.

[51] Marijn JH Heule, Matti Juhani Järvisalo, and Martin Suda. "Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions". In: (2018). http://hdl.handle.net/10138/237063.

[52] Marijn JH Heule, Matti Järvisalo, and Martin Suda. "Proceedings of SAT Race 2019: Solver and Benchmark Descriptions". In: (2019). http://hdl.handle.net/10138/306988.

[53] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[54] Daniel Huang et al. "Gamepad: A learning environment for theorem proving". In: *arXiv preprint arXiv:1806.00608* (2018).

[55] Jiayi Huang, Mostofa Patwary, and Gregory Diamos. "Coloring Big Graphs with AlphaGoZero". In: *arXiv preprint arXiv:1902.10162* (2019).

[56] Jia Hui Liang et al. "Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers". In: June 2015. DOI: 10.1007/978-3-319-26287-1_14.

[57] Frank Hutter et al. "Boosting Verification by Automatic Tuning of Decision Procedures". In: *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*. IEEE Computer Society, 2007, pp. 27–34. DOI: 10.1109/FAMCAD.2007.9. URL: https://doi.org/10.1109/FAMCAD.2007.9.

[58] Geoffrey Irving et al. "Deepmath-deep sequence models for premise selection". In: *NeurIPS*. 2016, pp. 2235–2243.

[59] Mikoláš Janota. "Circuit-based Search Space Pruning in QBF". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2018.

[60] Mikoláš Janota. "Towards generalization in QBF solving via machine learning". In: *AAAI Conference on Artificial Intelligence*. 2018.

[61] Mikolás Janota and Joao Marques-Silva. "Solving QBF by Clause Selection." In: *IJCAI*. 2015, pp. 325–331.

[62] Mikoláš Janota and Joao Marques-Silva. "Abstraction-based algorithm for 2QBF". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2011, pp. 230–244.

[63] Mikoláš Janota et al. "Solving QBF with counterexample guided refinement". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2012, pp. 114–128.

[64] Robert G Jeroslow and Jinchang Wang. "Solving propositional satisfiability problems". In: *Annals of mathematics and Artificial Intelligence* 1.1-4 (1990), pp. 167–187.

[65] Charles Jordan and Łukasz Kaiser. "Experiments with reduction finding". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2013, pp. 192–207.

[66] Cezary Kaliszyk et al. "Reinforcement Learning of Theorem Proving". In: *arXiv preprint arXiv:1805.07563* (2018).

[67] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. "Empirical Study of the Anatomy of Modern Sat Solvers". en. In: *Theory and Applications of Satisfiability Testing - SAT 2011*. Ed. by Karem A. Sakallah and Laurent Simon. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 343–356. ISBN: 978-3-642-21581-0.

[68] Henry Kautz and Bart Selman. "Planning as Satisfiability." In: Jan. 1992, pp. 359–363.

[69] Elias B. Khalil et al. "Learning to Branch in Mixed Integer Programming". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, pp. 724–731. URL: http://dl.acm.org/citation.cfm?id=3015812.3015920 (visited on 11/15/2018).

[70] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6980. 2015.

[71] William Klieber. *GhostQ QBF Solver System Description*. 2012.

[72] Mark A. Kramer. "Nonlinear principal component analysis using autoassociative neural networks". en. In: *AIChE Journal* 37.2 (1991). _eprint: https://aiche.onlinelibrary.wiley.com/doi/pdf/1 pp. 233–243. ISSN: 1547-5905. DOI: https://doi.org/10.1002/aic.690370209. URL: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.690370209 (visited on 04/28/2021).

[73] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[74] F. R. Kschischang, B. J. Frey, and H. -. Loeliger. "Factor graphs and the sum-product algorithm". In: *IEEE Transactions on Information Theory* 47.2 (2001), pp. 498–519. DOI: 10.1109/18.910572.

[75] Vitaly Kurin et al. "Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning". In: *CoRR* abs/1909.11830 (2019). http://arxiv.org/abs/1909.11830. arXiv: 1909.11830.

[76] Mitsuru Kusumoto, Keisuke Yahata, and Masahiro Sakai. "Automated Theorem Proving in Intuitionistic Propositional Logic by Deep Reinforcement Learning". In: *arXiv preprint arXiv:1811.00796* (2018).

[77] Guillaume Lample and François Charton. "Deep Learning for Symbolic Mathematics". In: *CoRR* abs/1912.01412 (2019). arXiv: 1912.01412. URL: http://arxiv.org/abs/1912.01412.

[78] Gil Lederman et al. "Learning Heuristics for Quantified Boolean Formulas through Reinforcement Learning". In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. https://openreview.net/forum?id=BJluxREKDB. OpenReview.net, 2020.

[79] Yujia Li et al. *Gated Graph Sequence Neural Networks*. 2017. arXiv: 1511.05493 [cs.LG].

[80] Jia Hui Liang et al. "Learning rate based branching heuristic for SAT solvers". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2016, pp. 123–140.

[81] H. Liu. "Towards Better Program Obfuscation: Optimization via Language Models". In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 2016, pp. 680–682.

[82] Florian Lonsing and Armin Biere. "DepQBF: A Dependency-Aware QBF Solver". In: *JSAT* 7.2-3 (2010), pp. 71–76.

[83] Sarah Loos et al. "Deep network guided proof search". In: *arXiv preprint arXiv:1701.06972* (2017).

[84] Edward Loper and Steven Bird. "NLTK: The Natural Language Toolkit". In: *In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics. Philadelphia: Association for Computational Linguistics*. 2002.

[85] Sharad Malik and Lintao Zhang. "Boolean satisfiability from theoretical hardness to practical success". In: *Communications of the ACM* 52.8 (2009), pp. 76–82.

[86] Elitza N. Maneva, Elchanan Mossel, and Martin J. Wainwright. "A New Look at Survey Propagation and its Generalizations". In: *CoRR* cs.CC/0409012 (2004). URL: http://arxiv.org/abs/cs.CC/0409012.

[87] João Marques-silva. "The impact of branching heuristics in propositional satisfiability algorithms". In: *In 9th Portuguese Conference on Artificial Intelligence (EPIA*. 1999, pp. 62–74.

[88] João P Marques-Silva and Karem A Sakallah. "GRASP - A new search algorithm for satisfiability". In: *Computer Aided Design*. IEEE. 1997, pp. 220–227.

[89] Eric Martin and Chris Cundy. "Parallelizing Linear Recurrent Neural Nets Over Sequence Length". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=HyUNwulC-.

[90] Kuldeep S. Meel and S. Akshay. "Sparse Hashing for Scalable Approximate Model Counting: Theory and Practice". In: *CoRR* abs/2004.14692 (2020). https://arxiv.org/abs/2004.14692. arXiv: 2004.14692.

[91] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[92] Matthew W. Moskewicz et al. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings DAC*. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2. DOI: 10.1145/378239.379017.

[93] Matthew W Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. ACM. 2001, pp. 530–535.

[94] Lili Mou et al. "On End-to-End Program Generation from User Intention by Deep Neural Networks". In: *ArXiv* abs/1510.07211 (2015).

[95] Nathan Mull, Daniel J. Fremont, and Sanjit A. Seshia. "On the Hardness of SAT with Community Structure". In: *arXiv:1602.08620 [cs]* (Feb. 2016). arXiv: 1602.08620. URL: http://arxiv.org/abs/1602.08620 (visited on 11/28/2018).

[96] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. "Loopy Belief Propagation for Approximate Inference: An Empirical Study". In: *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*. UAI'99. Stockholm, Sweden: Morgan Kaufmann Publishers Inc., 1999, pp. 467–475. ISBN: 1558606149.

[97] Daniel W Otter, Julian R Medina, and Jugal K Kalita. "A survey of the usages of deep learning for natural language processing". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[98] Umut Oztok and Adnan Darwiche. "A Top-Down Compiler for Sentential Decision Diagrams". In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. http://ijcai.org/Abstract/15/443. AAAI Press, 2015, pp. 3141–3148.

[99] Jayashree Padmanabhan and Melvin Jose Johnson Premkumar. "Machine learning in automatic speech recognition: A survey". In: *IETE Technical Review* 32.4 (2015), pp. 240–251.

[100] Aditya Paliwal et al. "Graph Representations for Higher-Order Logic and Theorem Proving". In: *CoRR* abs/1905.10006 (2019). arXiv: 1905.10006. URL: http://arxiv.org/abs/1905.10006.

[101] Adam Paszke et al. "Automatic differentiation in PyTorch". In: 2017.

[102] Matthew E. Peters et al. *Deep contextualized word representations*. 2018. arXiv: 1802.05365 [cs.CL].

[103] Florian Pigorsch and Christoph Scholl. "An AIG-based QBF-solver using SAT for preprocessing". In: *Proceedings of the 47th Design Automation Conference*. ACM. 2010, pp. 170–175.

[104] Luca Pulina. "The Ninth QBF Solvers Evaluation-Preliminary Report." In: *QBF@ SAT*. 2016, pp. 1–13.

[105] Markus N Rabe and Sanjit A Seshia. "Incremental determinization". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer International Publishing. 2016, pp. 375–392.

[106] Markus N Rabe and Leander Tentrup. "CAQE: A certifying QBF solver". In: *Formal Methods in Computer-Aided Design (FMCAD), 2015*. IEEE. 2015, pp. 136–143.

[107] Markus N Rabe et al. "Understanding and Extending Incremental Determinization for 2QBF". In: *International Conference on Computer Aided Verification (accepted)*. 2018.

[108] A. Radford. "Improving Language Understanding by Generative Pre-Training". In: 2018.

[109] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2018). URL: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[110] Neil Robertson and Paul D. Seymour. "Graph Minors XXIII. Nash-Williams' Immersion Conjecture". In: *J. Comb. Theory, Ser. B* 100.2 (2010). https://doi.org/10.1016/j.jctb.2009.07.003, pp. 181–205.

[111] Neil Robertson and Paul D. Seymour. "Graph Minors. X. Obstructions to Tree-Decomposition". In: *J. Comb. Theory, Ser. B* 52.2 (1991). https://doi.org/10.1016/0095-8956(91)90061-N, pp. 153–190.

[112] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[113] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[114] Tim Salimans et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning". In: *CoRR* abs/1703.03864 (2017). arXiv: 1703.03864. URL: http://arxiv.org/abs/1703.03864.

[115] Tian Sang, Paul Beame, and Henry A. Kautz. "Heuristics for Fast Exact Model Counting". In: *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*. Vol. 3569. Lecture Notes in Computer Science. https://doi.org/10.1007/11499107_17. Springer, 2005, pp. 226–240.

[116] Tian Sang et al. "Combining Component Caching and Clause Learning for Effective Model Counting". In: *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*. http://www.satisfiability.org/SAT04/programme/21.pdf. 2004.

[117] Franco Scarselli et al. "The Graph Neural Network Model". In: *Trans. Neur. Netw.* 20.1 (Jan. 2009), pp. 61–80. ISSN: 1045-9227. DOI: 10.1109/TNN.2008.2005605. URL: http://dx.doi.org/10.1109/TNN.2008.2005605.

[118] Joseph Scott et al. "MachSMT: A Machine Learning-based Algorithm Selector for SMT Solvers". In: *Tools and Algorithms for the Construction and Analysis of Systems* 12652 (2020), p. 303.

[119] Daniel Selsam and Nikolaj Bjørner. "NeuroCore: Guiding High-Performance SAT Solvers with Unsat-Core Predictions". In: *CoRR* abs/1903.04671 (2019). arXiv: 1903.04671. URL: http://arxiv.org/abs/1903.04671.

[120] Daniel Selsam and Nikolaj Bjørner. "NeuroCore: Guiding High-Performance SAT Solvers with Unsat-Core Predictions". In: *CoRR* abs/1903.04671 (2019). http://arxiv.org/abs/1903.04671. arXiv: 1903.04671.

[121] Daniel Selsam et al. "Learning a SAT Solver from Single-Bit Supervision". In: *arXiv preprint arXiv:1802.03685* (2018).

[122] Daniel Selsam et al. "Learning a SAT Solver from Single-Bit Supervision". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. https://openreview.net/forum?id=HJMC_iA5tm. OpenReview.net, 2019.

[123] Sanjit A. Seshia. "Combining Induction, Deduction, and Structure for Verification and Synthesis". In: *Proceedings of the IEEE* 103.11 (2015), pp. 2036–2051.

[124] Sanjit A Seshia, Shuvendu K Lahiri, and Randal E Bryant. "A hybrid SAT-based decision procedure for separation logic with uninterpreted functions". In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*. IEEE. 2003, pp. 425–430.

[125] Shubham Sharma et al. "GANAK: A Scalable Probabilistic Exact Model Counter". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. ijcai.org, 2019, pp. 1169–1176. URL: https://doi.org/10.24963/ijcai.2019/163.

[126] Tian Shi et al. "Neural abstractive text summarization with sequence-to-sequence models". In: *ACM Transactions on Data Science* 2.1 (2021), pp. 1–37.

[127] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[128] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556.

[129] A. Smola et al. "A Hilbert Space Embedding for Distributions". In: *Algorithmic Learning Theory, Lecture Notes in Computer Science 4754*. Max-Planck-Gesellschaft. Berlin, Germany: Springer, Oct. 2007, pp. 13–31.

[130] Richard Socher, Christopher Manning, and Andrew Ng. "Learning Continuous Phrase Representations and Syntactic Parsing with Recursive Neural Networks". In: (Jan. 2010).

[131] Richard Socher et al. "Semantic Compositionality through Recursive Matrix-Vector Spaces". In: *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Jeju Island, Korea: Association for Computational Linguistics, July 2012, pp. 1201–1211. URL: https://www.aclweb.org/anthology/D12-1110.

[132] Armando Solar-Lezama et al. "Combinatorial sketching for finite programs". In: *ACM Sigplan Notices* 41.11 (2006), pp. 404–415.

[133] Mate Soos. "CryptoMiniSat v4". In: *SAT Competition* (2014), p. 23.

[134] Mate Soos, Raghav Kulkarni, and Kuldeep S Meel. "CrystalBall: Gazing in the Black Box of SAT Solving". en. In: (), p. 17.

[135] Mate Soos, Raghav Kulkarni, and Kuldeep S. Meel. "CrystalBall: Gazing in the Black Box of SAT Solving". In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*. July 2019.

[136] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *CoRR* abs/1409.3215 (2014). arXiv: 1409.3215. URL: http://arxiv.org/abs/1409.3215.

[137] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[138] Christian Szegedy et al. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV].

[139] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. "Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks". In: *CoRR* abs/1503.00075 (2015). arXiv: 1503.00075. URL: http://arxiv.org/abs/1503.00075.

[140] Lei Tai et al. "A survey of deep network solutions for learning control in robotics: From reinforcement to imitation". In: *arXiv preprint arXiv:1612.07139* (2016).

[141] Leander Tentrup. "Non-prenex QBF solving using abstraction". In: *International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Springer. 2016, pp. 393–401.

[142] Leander Tentrup. "On expansion and resolution in CEGAR based QBF solving". In: *International Conference on Computer Aided Verification*. Springer. 2017, pp. 475–494.

[143] Marc Thurley. "SharpSAT - Counting Models with Advanced Component Caching and Implicit BCP". In: *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*. Ed. by Armin Biere and Carla P. Gomes. Vol. 4121. Lecture Notes in Computer Science. https://doi.org/10.1007/11814948_38. Springer, 2006, pp. 424–429.

[144] Trieu Trinh et al. "Learning longer-term dependencies in rnns with auxiliary losses". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4965–4974.

[145] Grigori S Tseitin. "On the complexity of derivation in propositional calculus". In: *Studies in constructive mathematics and mathematical logic* 2.115-125 (1968), pp. 10–13.

[146] Pashootan Vaezipoor et al. "Learning Branching Heuristics for Propositional Model Counting". In: *ArXiv* abs/2007.03204 (2020).

[147] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[148] Marcell Vazquez-Chanlatte. *mvcisback/py-aiger*. Aug. 2018. DOI: 10.5281/zenodo.1326224. URL: https://doi.org/10.5281/zenodo.1326224.

[149] Marcell Vazquez-Chanlatte, Markus N. Rabe, and Sanjit A. Seshia. "A Model Counter's Guide to Probabilistic Systems". In: *CoRR* abs/1903.09354 (2019). http://arxiv.org/abs/1903.09354. arXiv: 1903.09354.

[150] Marcell Vazquez-Chanlatte et al. "Learning Task Specifications from Demonstrations". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. http://papers.nips.cc/paper/7782-learning-task-specifications-from-demonstrations. 2018, pp. 5372–5382.

[151] Anirudh Vemula, Wen Sun, and J. Andrew Bagnell. "Contrasting Exploration in Parameter and Action Space: A Zeroth-Order Optimization Perspective". In: *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*. Vol. 89. Proceedings of Machine Learning Research. http://proceedings.mlr.press/v89/vemula19a.html. PMLR, 2019, pp. 2926–2935.

[152] Athanasios Voulodimos et al. "Deep learning for computer vision: A brief review". In: *Computational intelligence and neuroscience* 2018 (2018).

[153] Martin J. Wainwright and Michael I. Jordan. "Graphical Models, Exponential Families, and Variational Inference". In: *Found. Trends Mach. Learn.* 1.1–2 (Jan. 2008), pp. 1–305. ISSN: 1935-8237. DOI: 10.1561/2200000001. URL: https://doi.org/10.1561/2200000001.

[154] Mingzhe Wang et al. "Premise Selection for Theorem Proving by Deep Graph Embedding". In: *NIPS*. 2017.

[155] Daan Wierstra et al. "Natural Evolution Strategies". In: *J. Mach. Learn. Res.* 15.1 (2014). http://dl.acm.org/citation.cfm?id=2638566, pp. 949–980.

[156] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[157] Keyulu Xu et al. "How Powerful are Graph Neural Networks?" In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. https://openreview.net/forum?id=ryGs6iA5Km. OpenReview.net, 2019.

[158] Lin Xu et al. "SATzilla: portfolio-based algorithm selection for SAT". In: *Journal of artificial intelligence research* 32 (2008), pp. 565–606.

[159] Kaiyu Yang and Jia Deng. "Learning to Prove Theorems via Interacting with Proof Assistants". In: *arXiv preprint arXiv:1905.09381* (2019).

[160] Zhanfu Yang et al. "Graph Neural Reasoning for 2-Quantified Boolean Formula Solvers". In: *arXiv preprint arXiv:1904.12084* (2019).

[161] Zhilin Yang et al. "Xlnet: Generalized autoregressive pretraining for language understanding". In: *arXiv preprint arXiv:1906.08237* (2019).

[162] Emre Yolcu and Barnabás Póczos. "Learning Local Search Heuristics for Boolean Satisfiability". In: 2019.

[163] Emre Yolcu and Barnabás Póczos. "Learning Local Search Heuristics for Boolean Satisfiability". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*. http://papers.nips.cc/paper/9012-learning-local-search-heuristics-for-boolean-satisfiability. 2019, pp. 7990–8001.

[164] Wojciech Zaremba, Karol Kurach, and Rob Fergus. "Learning to Discover Efficient Mathematical Identities". In: *CoRR* abs/1406.1584 (2014). arXiv: 1406.1584. URL: http://arxiv.org/abs/1406.1584.

[165] Wojciech Zaremba and Ilya Sutskever. "Learning to Execute". In: *CoRR* abs/1410.4615 (2014). arXiv: 1410.4615. URL: http://arxiv.org/abs/1410.4615.

Appendix A

Additional NN Architectures and General Lore

A.1 Auto-Encoders

Auto-Encoders [72] are a family of architectures that through unsupervised training learn to represent the distribution of some data. The intuition behind the architectures is for the network to try to reconstruct the input on its outputs, using a simple MSE loss. The tricky part is that the input is somehow constrained or corrupted, and so in order to reconstruct the original, the network must learn good representations of the distribution of inputs.

The simplest autoencoder is an $MLP(n, k, n)$, where $k < n$. The small k creates an information bottleneck in the network, and forces it to learn a compact representation, in effect learning lossy compression. Other variants constrain the middle layer not by its width but by enforcing sparseness. Yet other variants corrupt the input with random noise during training rather than constraining it.

A.2 Attention

Attention is a mechanism used to learn how to 'attend' to parts of some larger whole. Take for example translating a sentence from English to French. We generate the french translation one word at a time, and as we generate every word in turn, we have to look at different parts of the English sentence. The same can be done for generating sentences to caption images. As we generate the words in the caption, we attend to different relevant parts of the image.

Formally, we have a set of (m, l)-dimensional key-value tuples $\{(K_1, V_1), \ldots, (K_n, V_n)\}$, given a Query vector $Q \in \mathbb{R}^m$, we can compute an *attention mask* as $Softmax(\{a_1 =< Q, K_1 >, \ldots, a_n =< Q, K_n >\})$, where $< \cdot, \cdot >$ is inner product in $\mathbb{R}^m$. With the mask, the result of attending by Q over $\{(K_1, V_1), \ldots, (K_n, V_n)\}$ is $\sum_{i=1}^{n} a_i V_i$. It is important to note that n, the number of keys/values, is not fixed. This makes attention the general way to combine any set of vectors into one.

A.3 Gating Mechanisms

Gating mechanisms were developed in the context of RNNs and their dual problem of vanishing gradient / representing identity. The two most common examples are Long Short-Term Memory (LSTM) networks [53] and Gated Recurrent Unit (GRU) [33], and whenever RNN is mentioned, in practice it is one of these two. The intuition is easiest to understand from the equations for GRU taking a previous state h_{t-1} and input x_t to produce the next state h_t:

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z) \tag{A.1}$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r) \tag{A.2}$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \tag{A.3}$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t \tag{A.4}$$

Where $\odot$ is the Hadamard product, a pointwise product between elements of matrices of equal dimensions, $W_z, W_r, W_h, U_z, U_r, U_h, b_z, b_r, b_h$ are learned parameter matrices, and σ_g is a Sigmoid. Given the previous state h_{t-1}, $\hat{h}_t$ is the "update candidate". In Eq. A.4 we see that the output h_t is a (pointwise) convex combination of the previous state and the new update candidate, where the z_t, called the "update gate", a matrix of elements in $[0, 1]$, decides for each element how much of the old state is kept vs. how much of the new state is taken.

What this gating trick allows for is easily passing-through the hidden state - all that it has to do is make sure z_t is close to zero. In contrast, if the new state h_t is computed directly as the output of some network operating on the previous state $U(h_{t-1})$, the network U has to work extremely hard (in terms of arriving at good parameters) to be able to merely "do-nothing" and pass the information. Thus, gating in architectures like LSTM and GRU ease the flow of information (and the flow of gradients back), allowing for longer RNNs to be trained.

A.4 Auto-Diff

Before autodiff, doing ML was hard. You had to compute by hand the gradients of your model in order to do back-propagation. Take for example the softmax function $(y_1, y_2, y_3) = \sigma(x_1, x_2, x_3)$, specifically, $y_i = \frac{e^{x_i}}{e^{x_1}+e^{x_2}+e^{x_3}}$. You are responsible for computing both $\sigma(x_0)$ (for some $x_0 \in \mathbb{R}^3$) and its gradient $\left.\frac{\partial \sigma(x)}{\partial x}\right|_{x=x_0}$, a 3×3 matrix of partial derivatives.

Auto-diff frameworks such as those implemented in TensorFlow[1] or PyTorch[2] make this computation for you automatically. They include hundreds of function components that can

[1] https://www.tensorflow.org/.

[2] https://pytorch.org/

be composed into a computation graph. Each component implements both evaluation of the function at a point, which is called the *forward pass*, and the evaluation of the gradient at that point. Combining this with the back-propagation algorithm (which is essentially function composition) allows the user to seamlessly compute gradients of entire computation graphs.

Appendix B

Additional Information for Chapter 3

B.1 Additional details about Cadet

B.2 Global Solver State

1. Current decision level
2. Number of restarts
3. Restarts since last major restart
4. Conflicts until next restart
5. Ratio of variables that already have a Skolem function to total variables. Formula is solved when this reaches 1.

B.3 Literal Labels

Here we describe the details of the variable labels presented to the neural network described in Section 4.2. The vector $\mathbf{v}$ consists of the following 7 values:

$y_0 \in \{0, 1\}$ indicates whether the variable is universally quantified,
$y_1 \in \{0, 1\}$ indicates whether the variable is existentially quantified,
$y_2 \in \{0, 1\}$ indicates whether the variable has a Skolem function already,
$y_3 \in \{0, 1\}$ indicates whether the variable was assigned constant True,
$y_4 \in \{0, 1\}$ indicates whether the variable was assigned constant False,
$y_5 \in \{0, 1\}$ indicates whether the variable was decided positive,
$y_6 \in \{0, 1\}$ indicates whether the variable was decided negative, and

B.4 The QDIMACS File Format

QDIMACS is the standard representation of quantified Boolean formulas in prenex CNF. It consists of a header "`p cnf <num_variables> <num_clauses>`" describing the number of variables and the number of clauses in the formula. The lines following the header indicate the quantifiers. Lines starting with '`a`' introduce universally quantified variables and lines starting with '`e`' introduce existentially quantified variables. All lines except the header are terminated with `0`; hence there cannot be a variable named 0. Every line after the quantifiers describes a single clause (i.e. a disjunction over variables and negated variables). Variables are indicated simply by an index; negated variables are indicated by a negative index. Below give the QDIMACS representation of the formula $\forall x.\, \exists y.\, (x \vee y) \wedge (\neg x \vee y)$:

```
p cnf 2 2
a 1 0
e 2 0
1 2 0
-1 2 0
```

There is no way to assign variables strings as names. The reasoning behind this decision is that this format is only meant to be used for the computational backend.

B.5 Hyperparameters and Training Details

We trained a model on the reduction problems training set for 10M steps on an AWS server of type C5. We trained with the following hyperparameters, yet we note that training does not seem overly sensitive:

- Literal embedding dimension: $\delta_L = 16$
- Clause embedding dimension: $\delta_C = 64$
- Learning rate: 0.0006 for the first 2m steps, then 0.0001
- Discount factor: $\gamma = 0.99$
- Gradient clipping: 2
- Number of iterations (size of graph convolution): 1
- Minimal number of timesteps per batch: 1200