Machine Learning for Automated Theorem Proving: Learning to Solve SAT and QSAT

Sean B. Holden
University of Cambridge
sbh11@cam.ac.uk

October 25, 2021

Abstract

The decision problem for Boolean satisfiability, generally referred to as SAT, is the archetypal NP-complete problem, and encodings of many problems of practical interest exist allowing them to be treated as SAT problems. Its generalization to quantified SAT (QSAT) is PSPACE-complete, and is useful for the same reason. Despite the computational complexity of SAT and QSAT, methods have been developed allowing large instances to be solved within reasonable resource constraints. These techniques have largely exploited algorithmic developments; however machine learning also exerts a significant influence in the development of state-of-the-art solvers. Here, the application of machine learning is delicate, as in many cases, even if a relevant learning problem can be solved, it may be that incorporating the result into a SAT or QSAT solver is counterproductive, because the run-time of such solvers can be sensitive to small implementation changes. The application of better machine learning methods in this area is thus an ongoing challenge, with characteristics unique to the field. This work provides a comprehensive review of the research to date on incorporating machine learning into SAT and QSAT solvers, as a resource for those interested in further advancing the field.

1 Introduction

Automated theorem proving represents a significant and long-standing area of research in computer science, with numerous applications. A large proportion of the methods developed to date for the implementation of *automated theorem provers (ATPs)* have been algorithmic, sharing a great deal in common with the wider study of heuristic search algorithms (Harrison, 2009). However in recent years researchers have begun to incorporate *machine learning (ML)* methods (Murphy, 2012) into ATPs in an effort to extract better performance.

ATPs represent a compelling area in which to explore the application of ML. It is well-known that theorem-proving problems are computationally intractable, with the exception

of specific, limited cases. Even in the apparently simple case of *propositional logic* the task is NP-hard, and adding *quantifiers* makes the task PSPACE-complete (Garey and Johnson, 1979). Taking a small step further we arrive at *first-order logic (FOL)*, which is undecidable (Boolos *et al.*, 2007). In addition to the general computational complexity of theorem-proving problems, they have a common property that makes them challenging as a target for ML: even the most trivial change to the statement of a problem can have a huge impact on the complexity of any subsequent proof attempt (Fuchs and Fuchs, 1998; Hutter *et al.*, 2007; Hutter *et al.*, 2009; Biere and Fröhlich, 2015; Biere and Fröhlich, 2019).

The aim of this work is to review the research that has appeared to date on incorporating ML methods into solvers for propositional *satisfiability (SAT)* problems, and also solvers for its immediate variants such as *quantified SAT (QSAT)*.

In a sense, these are some of the simplest possible ATP problems. (Any instance of a SAT problem can be represented as a Boolean formula in conjunctive normal form, and it is undeniably hard to propose anything much simpler.) But the combination of the computational challenges such problems present, and the enormous range of significant, practical applications that can be addressed this way, makes general solvers for SAT and its friends a compelling target for research. Marques-Silva (2008) reviews applications of SAT solvers circa 2008, and the interested reader might consult work applying them to bounded model checking (Biere *et al.*, 1999; Clarke *et al.*, 2001), planning (Kautz and Selman, 1992; Kautz, 2006), bioinformatics (Lynce and Marques-Silva, 2006; Graça *et al.*, 2010), allocation of radio spectrum (Fréchette *et al.*, 2016), and software verification (Babić and Hu, 2007). A further notable application has been the solution of the *Boolean Pythagorean triples* problem by Heule *et al.* (2016), resulting in what is currently considered the longest mathematical proof in history.

1.1 Coverage

Work on applying ML in this context appears to have started with Ertel *et al.* (1989) and Johnson (1989). At that time the limited availability of computing power and the limitations of existing solvers made the studies necessarily small by current standards, in terms of the size of the problems addressed, and also of the ML methods applied. This review is the result of a systematic search for literature appearing from then until late 2020.

SAT/QSAT solving and machine learning are both large and long-standing areas of research, and each has a correspondingly large literature. As these are two apparently rather unrelated fields, it is therefore inevitable that any reader versed in one might feel less confident with the other. (It has certainly been my experience in talking to researchers from both domains that this is often the case.) It would not be feasible to explain either, let alone both, areas in full detail here; and in any case, this is not intended to be a textbook on

either subject. I have provided an introduction to each, but experts in either area might find one presentation overly elementary and the other too brief. The aim has been to provide sufficient information to make this work self-contained for both sides while maintaining a manageable length; however I expect that for many there will be areas where further reading will be necessary.

I wrote this work guided by two central aims for what the reader should gain from it. First, they should know *what has been tried*. In presenting the material, I concentrate on the learning methods used and the way in which they have been incorporated into solvers. As the literature rarely if ever presents methods not leading to performance improvements of some kind, less consideration is given to the details of the level of improvement achieved, because I believe such details are secondary to my second aim, which is: that the reader should understand the often *complex interaction between ATP and ML that is needed for success* in these undeniably challenging applications.

In order to achieve these aims it was necessary to be quite selective in the level of detail used to present various methods. Some research is presented in very great detail, relating to the learning method and its relationship with a solver, the description of the data used, or the experimental method employed. Other research is presented in less detail, although I hope at a level sufficient to allow the reader to understand what was done, and why. With the exception of the Chapters on ATP and ML, each Chapter presents a discussion summarizing what I believe are the central lessons to be taken from it. Where methods have been presented in greater detail, it is generally in the service of these arguments.

1.2 Outline of the review

Chapter 2 presents an introduction to the SAT problem, and to contemporary methods for its solution. Much of this section is devoted to summarizing the operation of *Conflict-Driven Clause Learning (CDCL)* solvers;[1] first, as these form the core of many of the most successful SAT solvers available; and second, because there are many distinct areas of their operation that have provided a point at which to introduce ML, and this therefore provides a road map for a large portion of the review. This section also briefly describes *portfolio solvers* and *local search* solvers, which have also been targets for ML, and which will be described further in later Chapters.

Chapter 3 provides a complementary introduction to some of the ML methods most

[1] There is an important distinction to be made here for the avoidance of confusion. The term 'learning' in the context of a CDCL solver is, at least at first glance, unconnected to the idea of machine learning. It is used to describe the addition of one or more new clauses to a problem after analysing a conflict during the search for a satisfying assignment; this is explained in more detail in Section 2.4.4. The use of the term 'learning' in both contexts is ubiquitous however, and we expand on the distinction a little further in Section 3.1.5.

commonly applied to SAT and QSAT solvers; this work spans supervised and unsupervised learning in addition to n-armed bandits, reinforcement learning, neural networks and evolutionary computing. In addition we describe some of the main sources of problems available for testing SAT and QSAT solvers; as these are often annotated such that we know which problems are satisfiable, and which are not, they provide a valuable resource for training ML methods.

Many applications of ML in this domain have required a phase of *feature engineering*, whereby a problem, typically expressed in *conjunctive normal form (CNF)*, is converted into a vector of real numbers suitable for use by an ML method. Chapter 4 reviews common sets of features that have been used, and that continue to form the basis for many ongoing studies. More recent work has made significant use of *graph neural networks* to (partially) automate the feature engineering process, and we introduce these here also.

There are, broadly-speaking, four ways in which ML has been applied to SAT solvers: by treating SAT directly as a classification problem; by building *portfolios* of existing SAT solvers; by modifying CDCL solvers; and by treating the problem as a form of *local search*.

In Chapter 5 we describe work aiming to identify satisfiability directly, without necessarily also obtaining a satisfying assignment of variables if one exists. Here, the SAT problem is treated as a classification problem: given a formula f, we aim to return the answer 'yes' or 'no', indicating whether or not the problem is satisfiable. In some cases it may be possible to extract a satisfying assignment as a side-effect.

Portfolio solvers are addressed in Chapter 6. Here, a collection of different SAT solvers is used in some combination to attack a problem. Chapter 7 then reviews the application of ML to CDCL solvers, addressing in turn the way in which ML has been applied to the individual elements described in Chapter 2. Chapter 8 describes the application of ML to local search SAT solvers.

In Chapter 9 we address attempts to introduce ML into solvers for QSAT. This area has received comparatively little attention, but work has appeared addressing ML for both portfolio solvers, and individual solvers.

While this review mainly addresses solvers for SAT and QSAT—these problems having received considerable attention as they have clear and significant applications—in Chapter 10 we briefly address machine learning applied to *intuitionistic propositional logic (IPL)* (Dalen, 2001). While this logic is of more foundational interest, having few applications beyond the philosophy of mathematics, it is related sufficiently closely to propositional logic that I feel attempts to apply machine learning to the search for proofs in IPL are relevant.

Chapter 11 concludes.

1.3 Limits to Coverage

A body of research exists addressing methods for automatically configuring algorithms that expose parameters—a process sometimes referred to as the *algorithm configuration problem*. Effective methods such as ParamILS (Hutter *et al.*, 2009) and, perhaps the best-known system of this kind, *Sequential Model-based Algorithm Configuration (SMAC)* (Hutter *et al.*, 2011), are now common. Algorithms in this class can clearly be applied to SAT/QSAT and related solvers, which invariably have parameters governing aspects of their operation. In compiling this review, I have aimed to focus on material that has a *specific emphasis* on SAT, QSAT and (closely) related problems. As a result, I decided not to describe in detail work such as that of Kadioglu *et al.* (2010) and Malitsky *et al.* (2013), that develops a general method for algorithm configuration and uses SAT as a test case, or Hutter *et al.* (2007) and Mangla *et al.* (2020), that is predominantly an application of an existing algorithm configuration method to SAT. For the same reasons, I have not included work that mainly relies on the application of general methods for selecting an algorithm from a collection of candidates; see Kotthoff (2016) for a review of such methods.

1.4 What Should the Reader Gain?

It is my hope that ML researchers might gain from this work, an understanding of state-of-the-art SAT and QSAT solvers that is sufficient to make new opportunities for applying their own ML research to this domain clearly visible. It is equally my hope that ATP researchers will gain a complementary understanding, giving them a clear appreciation of how state-of-the-art machine learning might help them to design better solvers. For both constituencies, I aim to show what has already been achieved at the time of writing, at a level of detail sufficient to provide a basis for new work.

2 Algorithms for Solving SAT

SAT is a central problem in computer science, and consequently its literature is huge; the reader seeking a comprehensive review of the field as it stood in 2009 might consult Biere *et al.* (2009), and a great deal has happened since then. As a result, it will be impossible to be comprehensive in this work; rather, it is the principal aim of this Chapter to give a view of current SAT solving methods that is sufficiently detailed for ML practitioners to appreciate how ML has been applied to them.

Attempts to use ML to improve SAT solvers have, inevitably, tended to focus on adding to methods that are already known to be effective. (This has not always been the case, as we shall see.) Such methods fall into three broad categories—local search methods (Section 2.3),

CDCL solvers (Section 2.4) and portfolios (Section 2.5)—and we present each of these in turn. We begin however with some essential definitions, and by introducing the DPLL algorithm.

2.1 The SAT Problem

Let V be a set of propositional variables taking values T or F ('true' or 'false'). A *literal* l is either a variable $v \in V$ or its negation $\neg v$. A *clause* c is a disjunction $l_1 \vee \cdots \vee l_n$ of n literals where $\vee$ denotes the logical 'or' connective. A *formula* f is a conjunction $(c_1 \wedge \cdots \wedge c_m)$ of m clauses where $\wedge$ denotes the logical 'and' connective. The symbols $\neg$, $\vee$ and $\wedge$ are interpreted in the usual way. For example, with $V = \{v_1, v_2, v_3\}$ we might define

$$f = (\neg v_2 \vee v_3) \wedge (v_1 \vee v_3) \wedge (v_2). \tag{2.1}$$

This is a *conjunctive normal form (CNF)* formula and any Boolean formula involving the usual extended set of connectives ($\rightarrow$ and so on) can be equivalently expressed in this way. The majority of SAT solvers use CNF as the format for their input. (We will make specific note where necessary of solvers that do not; this will be particularly relevant when we discuss quantified SAT solvers, many of which favour circuit-based representations.)

An *assignment* A is a partial function from V to the set $\{T, F\}$. We write for example $A(v_1) = F$ to denote that A assigns v_1 to be false. A variable v for which $A(v)$ is not defined is *unassigned*. The truth or falsity of a formula is then defined by extending A in the usual way. That is,

$$\begin{aligned} A(\neg v) = T &\iff A(v) = F \\ A(f_1 \vee f_2) = T &\iff A(f_1) = T \text{ or } A(f_2) = T \\ A(f_1 \wedge f_2) = T &\iff A(f_1) = T \text{ and } A(f_2) = T. \end{aligned}$$

A formula f is *satisfiable* if there is an assignment A such that $A(f) = T$. Such an assignment is known as a *model*. The set of satisfiable formulas is denoted SAT, and we are interested in the following decision problem: given an arbitrary CNF formula f, is $f \in$ SAT? For example in the case of the formula in Equation (2.1), we see that $f \in$ SAT with the model $(v_1 = F, v_2 = T, v_3 = T)$. In this example other models also suffice. If an algorithm always provides an answer, given an instance f, it is *complete*; if this answer is always correct, the algorithm is *consistent*.

The SAT problem has numerous variations. For example, the #SAT problem, asks *how many* satisfying assignments f has, and the *quantified SAT (QSAT)* problem considers the addition of quantifiers to propositional formulas. #SAT has received little attention from ML researchers and will not be discussed further beyond a single example in Chapter 7. QSAT is described separately in Chapter 9.

There are numerous algorithms for deciding whether a given formula f is in SAT, and these algorithms fall naturally into three categories:

1. Solvers based on *local search*. These are described in Section 2.3.

2. Distinct from local search solvers, the category that has proved the most influential is based on extensive modifications to the *Davis, Putnam, Logemann, Loveland (DPLL) algorithm* (Davies *et al.*, 1962) described in the next Section. The modifications needed, leading to *Conflict-Driven Clause Learning (CDCL)* solvers, are extensive, and are described in Section 2.4.

3. *Portfolio Solvers* employ collections, or *portfolios* of multiple, different SAT-solvers. These are introduced in Section 2.5.

These categories have very different characteristics, and this is reflected in the methods used to apply ML to them. For example, DPLL can be considered a search algorithm, and this observation leads to natural methods for adding ML—we shall discuss this further below. In extending DPLL to CDCL many other points become available at which ML can be applied in specific ways. Local search solvers are incomplete, in that they can not prove unsatisfiability, while in portfolio solvers the emphasis is on learning to select which solver to use for a given problem. These differences entail the use of different approaches to ML.

2.2 The DPLL Algorithm

Starting with a formula in CNF, we need some further definitions. If a literal l only appears with one polarity—that is, for some variable v either $l = v$ throughout the formula or $l = \neg v$ throughout the formula—it is known as a *pure* literal. A clause containing a single literal is called a *unit* clause. The DPLL algorithm works by repeating the following steps:

1. Propagate the effects of unit clauses: for each unit clause with literal l, remove all other clauses containing l, and remove $\neg l$ from any clause in which it appears. The last step may produce further unit clauses, so we continue until either all clauses are satisfied, we detect an inconsistency (because a clause becomes empty), or there are no unit clauses remaining.

2. For each pure literal l, remove all clauses containing l.

3. Choose a variable v and perform a split, recursively testing for satisfiability when $v = T$ and when $v = F$.

The process can easily be arranged as a depth-first search with chronological backtracking. If at any point in the process the empty clause is generated then the formula is unsatisfiable

with the set of assignments currently made, and the process backtracks; alternatively, if the process generates a consistent collection of literals then a satisfying assignment can be returned.

One critical observation that can be made allowing ML to be applied to theorem provers is that many of them can be regarded as *search algorithms* wherein at each step of the search a critical choice needs to be made. For example, in the DPLL algorithm we have a search problem where the central choice involved is which variable to split on.[1] The hope is that by *learning from existing proofs* we might succeed in making this choice more effectively. In Section 2.4 we briefly review the modifications made to DPLL solvers allowing them to be applied to large problems, and in the process uncover several further areas where ML can be introduced.

2.3 Local search SAT solvers

While the DPLL algorithm makes a systematic, tree-based search for a satisfying assignment, an alternative method employed by state-of-the-art SAT solvers is that of *local search*. These solvers employ a fundamentally straightforward algorithm:

```
1 Select an initial solution candidate x ∈ {0,1}^n;
2 totalFlips = 0;
3 while totalFlips < flipLimit do
4     if x is a satisfying assignment then
5         return x;
6     totalFlips = totalFlips + 1;
7     Choose a variable to flip;
8     Flip the chosen variable in x;
```

Typically, a solver begins with a random assignment to the variables in an instance. If the assignment satisfies the instance we are done; otherwise, a variable is selected and its assignment flipped. This continues until we find a satisfying assignment, or reach a time-out. Solvers differ mainly in how a variable is chosen at each iteration. This process can be considered an application of hill-climbing search, where a step in the search space

[1] In a wider context, the same can be said of methods for theorem proving in equational reasoning, first-order logic (FOL) and so on. While modern FOL solvers for example rely on different proof methods, in particular resolution using some variant of the *given clause method* (McCune, 2003), there is a similar choice to be made in the search process: which given clause to use next. The most important single factor in making an effective FOL prover based on resolution is the ability to make such selections effectively; this was recognised early in the construction of ATPs with the use of simple heuristics such as *unit preference* (Wos, 1964) and many more advanced heuristics have since been developed for making such choices.

corresponds to flipping a variable, and the function being maximized is the number of satisfied clauses. Further refinements can be made to the method. For example, a solver might incorporate *restarts*, where the search process is stopped and restarted at a new, typically random assignment.

Since the introduction of GSAT (Selman *et al.*, 1992) and WalkSAT (Selman *et al.*, 1996), solvers of this kind, while they are generally incomplete as they can not prove unsatisfiability, have proven to be very effective for many practical problems.

Local search solvers have proved particularly popular as a target for ML based on *evolutionary algorithms* (Chapter 3, Section 3.6). This is because such algorithms can naturally represent heuristics similar to the hand-crafted heuristics used by solvers such as GSAT and WalkSAT, and modify them over time. We describe these methods in Chapter 8.

2.4 Conflict-Driven Clause Learning

DPLL is typically a depth-first search problem—in the third step we have to choose a variable along with a value to assign to it. In its basic form backtracking is chronological; that is, on exhausting the assignments for a variable we always backtrack to the immediately preceding variable in the search. In practice DPLL is ineffective at solving realistic problems, and modern SAT solvers use various techniques in an attempt to manage both time and space requirements. CDCL has become a general term applied to this class of SAT-solvers, although strictly speaking the title refers to only one of the methods that builds on DPLL. As these methods have often introduced new opportunities for applying ML, we now provide a brief summary. This is by no means a complete introduction and the interested reader is directed to Biere *et al.* (2009) and to the more recent papers cited below.[2]

A number of solvers are available taking different approaches to the implementation of these methods; see for example GRASP (Marques-Silva and Sakallah, 1999), CHAFF (Moskewicz *et al.*, 2001), BERKMIN (Goldberg and Novikov, 2002), MINISAT (Eén and Sörensson, 2003), PICOSAT (Biere, 2008b), MAPLESAT (Liang *et al.*, 2016b), GLUCOSE (Audemard and Simon, 2018) and KISSAT (Biere *et al.*, 2020). This list is by no means exhaustive; the annual *International SAT Competition*[3] provides an ongoing, up-to-date reference to the state-of-the-art.

[2] Descriptions of the relevant algorithms are often somewhat informal, which in the past has made reference to source code necessary for a definitive description. In recent work efforts have been made to provide formal descriptions allowing formally verified solvers to be constructed, and this work also provides a clear and precise description of many of the algorithms—see for example Nieuwenhuis *et al.* (2006), Fleury *et al.* (2018), and Blanchette *et al.* (2018).

[3] www.satcompetition.org

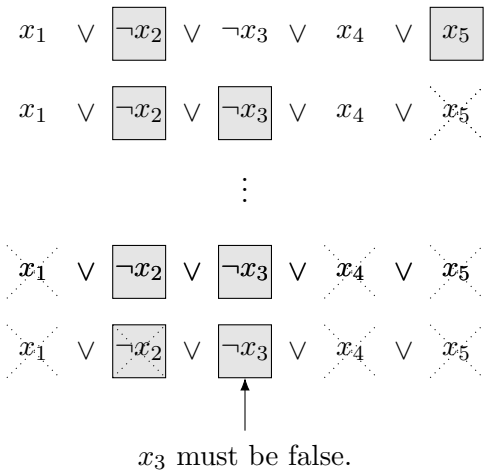

Figure 2.1: Illustration of the two watched literals algorithm. Each row is a single clause, represented at a different point in an attempt to find a satisfying assignment. Initially all literals are unassigned, and $\neg x_2$ and x_5 are watched. In the second step $\neg x_5$ is assigned, and as we are watching its negation the watched literal moves to an alternative that is unassigned or true. Later, $\neg x_1$ and $\neg x_4$ are assigned so the unwatched literals x_1 and x_4 have become false. Finally, assignment x_2 falsifies watched literal $\neg x_2$. There is no unwatched literal that is unassigned or true, so we need to propagate the assignment $\neg x_3$.

2.4.1 Preprocessing and Inprocessing

SAT solvers can be made more effective by attempting to simplify a formula at the start of the process—referred to as *preprocessing* (Eén and Biere, 2005)—or while the search for a solution is under way—referred to as *inprocessing* (Han and Somenzi, 2009; Hamadi *et al.*, 2010; Järvisalo *et al.*, 2012). Related methods employ simplification at other points in the SAT solving process, in particular by simplifying learned clauses (Sörensson and Biere, 2009; Luo *et al.*, 2017; Li *et al.*, 2020).

2.4.2 Efficient Unit Propagation

The time taken to solve realistic SAT problems is overwhelmingly dominated by the cost of performing unit propagation. Consequently, great care and ingenuity has been applied to optimizing this process. Several data structures have been devised that help in this process (see for example Lynce and Marques-Silva (2005)) however one in particular—the *two watched literals* algorithm—has become dominant.

The idea is straightforward and is illustrated in Figure 2.1. Initially all literals in all clauses are unassigned. The key insight is that, as long as at least two literals in a clause are unassigned, no unit propagation can occur from that clause. For each clause we choose (arbitrarily) two unassigned literals l_1 and l_2 to be watched. Each time a literal l is assigned

we only consider clauses watching its negation for unit propagation. For those clauses, without loss of generality let us assume that $l_1 = \neg l$. If the other watched literal l_2 is true then the clause is satisfied and there is nothing left to do. Otherwise, we search for another literal to watch instead of l_1. If there is an unwatched literal that is either unassigned or true then we watch it instead of l_1. Otherwise only two outcomes are possible:

1. The other watched literal l_2 is false, so there is a conflict as all literals in the clause are now false.

2. The other watched literal l_2 is free, so we need to perform a unit propagation.

It turns out that this process has the desirable property that the literals being watched do not need to be updated on backtracking. Additionally, further gains can be made by treating clauses of sizes 2 or 3 as special cases (Biere, 2008b).

2.4.3 Standard Variable Choice Heuristics

Considerable effort has been expended in the development of heuristics for choosing the next variable on which to branch in DPLL-based solvers. Perhaps the best known and most effective, until this class of heuristics was superseded by the *activity-related* heuristics described in Section 2.4.5, was the *Jeroslow-Wang heuristic* (Jeroslow and Wang, 1990). Let f be a CNF formula and for each literal l in f define

$$J(l) = \sum_{l \in c, c \in f} 2^{-|c|}.$$

The *one-sided Jeroslow-Wang (OS-JW)* heuristic selects the literal maximizing $J(l)$ as the next assignment. The *two-sided Jeroslow-Wang (TS-JW)* heuristic selects the variable v maximizing $J(v) + J(\neg v)$, and chooses its polarity according to whether $J(v) > J(\neg v)$.

Various further heuristics of this kind are reviewed in Marques-Silva (1999). However they are now little-used on account of the developments described in Section 2.4.5.

2.4.4 Clause Learning and Backjumping

It is worth re-iterating that, while the term 'clause learning' is ubiquitous in the SAT-solving literature, it is considered a distinct form of learning for the purposes of this review. The relationship between the two concepts is discussed further in Section 3.1.5.

Henceforth we need to make a distinction between assignments *chosen* after unit propagation terminates, and assignments that are *forced by* unit propagation. We will refer to the former as *asserted* or as *decision variables*, and the latter as *implied*. We also need to consider the *level* at which each assignment is made. Initially the level is 0. The level increases by 1 each time an assertion is made, and any assignments implied by an assertion share its level.

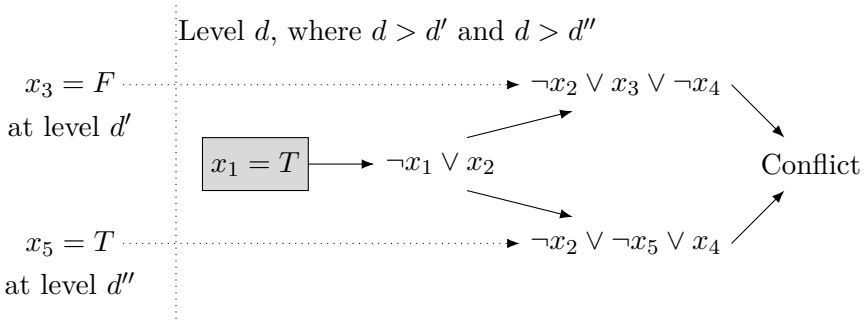

Figure 2.2: Learning a clause and backjumping. The assignment $x_1 = T$ causes unit propagations leading to a conflict. Resolving clauses as described in the text leads to clauses implied by the original problem; in this case the clause asserting $\neg x_3 \wedge x_5 \rightarrow \neg x_2$ can be added. In addition we can immediately backjump to the higher of levels d' and d''.

The key idea in clause learning is that the satisfiability of a set of clauses is unaffected if we add a new clause that is implied by the original set. Thus, further clauses can be added to the original problem. Typically when a *conflict* is detected, indicating that a current partial assignment to the variables can not be extended to a solution, the situation is analysed and information about the conflict is encoded in a new clause. Such clauses can be used to aid the continuing search. In practice the finding and addition of such clauses has three primary benefits:

1. Chosen carefully, they force unit propagations allowing the solver to avoid repeated steps.

2. The process of deriving them can provide information allowing us to backjump many levels, instead of a single level as with chronological backtracking, without any danger of missing out areas of the search space.

3. The degree to which variables are being used in the process of finding new, implied clauses[4] can be exploited to devise improved heuristics for which literal to assert next.

Many approaches have been suggested to the derivation of implied clauses; we will describe the most common: learning at the first *Unique Implication Point (UIP)*. Figure 2.2 illustrates the key idea. At some level d in the search we assert that $x_1 = T$. This triggers unit propagations leading to a conflict. Specifically, the clause $\neg x_1 \vee x_2$ propagates the assignment $x_2 = T$. This, in conjunction with assignments $x_3 = F$ and $x_5 = T$ made at earlier decision

[4]The term 'implied clause' has been defined in different ways by different authors. We use it largely in the sense termed a *resolvent* by Eén and Biere (2005), but the reader should be aware that others, such as Moskewicz *et al.* (2001), may use it differently.

levels d' and d'', causes two further unit propagations implying both $x_4 = T$ and $x_4 = F$. At this point we have a conflict and therefore need to backtrack.

Resolving the two latest clauses in this process leads to the clause $\neg x_2 \vee x_3 \vee \neg x_5$, which is implied by the original formula. Stepping back again and resolving with $\neg x_1 \vee x_2$ results in the clause $\neg x_1 \vee x_3 \vee \neg x_5$, which is also implied by the original formula. We can add any clause derived in this way to the original formula without changing its satisfiability.

Looking at the first clause obtained using resolution, note that it contains only a single variable at decision level d. Notice also that it can equivalently be written

$$\neg x_3 \wedge x_5 \rightarrow \neg x_2.$$

This means that, not only can we add it to the clauses available, but that it encodes enough knowledge about the original problem potentially to be of use in continuing the search. In particular, assuming without loss of generality that $d' > d''$, we can backjump to the last unit propagation at level d' and immediately assert $x_2 = F$.

A more general description of this process can be given using an *implication graph* (Zhang *et al.*, 2001) as illustrated in Figure 2.3. Here, the black node is a decision variable $v_5 = T$ at decision level 10, denoted $10|v_5 = T$. White nodes are implied assignments, and grey nodes are assignments at earlier decision levels. The problem clauses themselves are now implicit; for example, the implied assignment $v_6 = T$ can be seen to be due to a clause $\{\neg v_5, v_3, \neg v_{12}, v_6\}$ by examining nodes with outgoing edges connected to v_6.

We say that a node n_1 *dominates* node n_2 if any path from the decision variable at n_1's decision level to n_2 must pass through n_1. A *Unique Implication Point* is then a node that dominates both nodes causing the conflict, in this case the two nodes for v_{17}. In the present example, both v_5 and v_{29} are UIPs. The *first* UIP is the first such node encountered, working backwards from the right-hand-side of the graph, in this case v_{29}.

Multiple learned clauses can be obtained by *cutting* the implication graph as illustrated. (The two examples shown are not exhaustive.) A cut divides the graph into a *reason side* (including all nodes with no incoming edge) and a *conflict side* (including the conflicting variable). A learned clause is obtained by negating the variables of the nodes directly adjacent to the cut on its reason side—the reader may easily verify that this corresponds to the resolution process described earlier. In our example, Cut 1 allows us to learn the clause $\{\neg v_7, \neg v_6, v_{16}, v_{19}, \neg v_{13}\}$.

The cut corresponding to the first UIP places all assignments following the first UIP, and that have a path to the conflicting variable, on the conflict side, and all remaining assignments on the reason side. In our example this is Cut 2, which allows us to learn the clause $\{\neg v_7, \neg v_{29}, \neg v_{13}\}$ corresponding to the first UIP. This contains only one variable assigned at level 10, and is equivalent to $v_7 \wedge v_{13} \rightarrow \neg v_{29}$, so we can backjump to level 6 (the highest level among variables v_7 and v_{13}) and assert $v_{29} = F$.

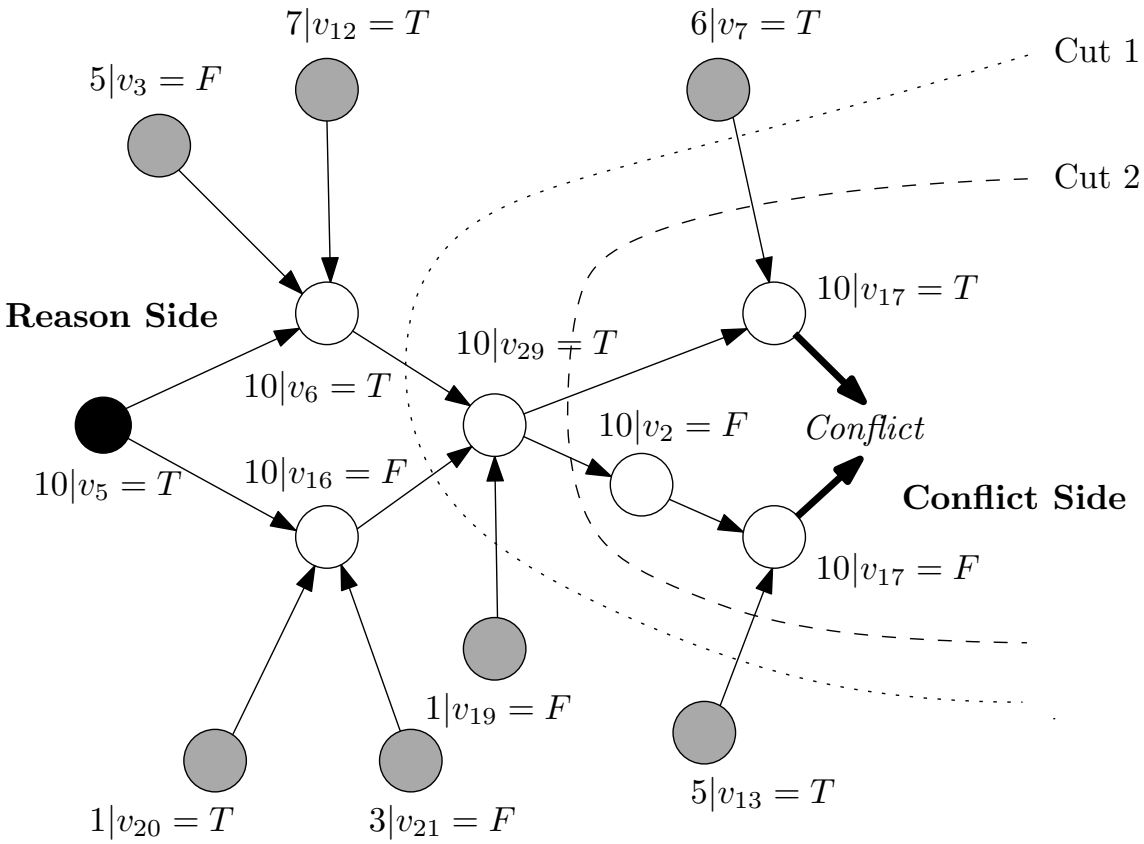

Figure 2.3: Example of an implication graph, with cuts used to generate two possible learned clauses.

In practice the learned clause for the first UIP can be obtained by resolving clauses starting with the conflict, and working backwards along the sequence of assignments, at each stage resolving with the clause that caused the assignment. We continue this process until the resulting clause contains only one variable at the current level. Thus in our example we start by resolving $\{\neg v_7, \neg v_{29}, v_{17}\}$ and $\{v_2, \neg v_{13}, \neg v_{17}\}$ to obtain $\{\neg v_7, \neg v_{29}, v_2, \neg v_{13}\}$. This has two variables at the current level. Immediately preceding the conflict is the assignment $v_2 = F$, caused by clause $\{\neg v_{29}, \neg v_2\}$. Resolving with this clause gives $\{\neg v_7, \neg v_{29}, \neg v_{13}\}$ as required.

Learned clauses can be simplified before adding them to the problem, as described by Sörensson and Biere (2009) and Hamadi *et al.* (2010).

Backjumping is an elegant and effective idea. Because of this, for some time it entirely replaced DPLL's *chronological* backtracking; that is, the simpler strategy of backtracking to the last decision and flipping the assignment. (Or backtracking further if both assignments have been tried.) However recent work by Nadel and Ryvchin (2018) argues that chronological backtracking can be beneficial, and suggests a backtracking strategy that applies it if, first, sufficient conflicts have occurred since solving began, and second, the chronological and backjumping levels are sufficiently different.

2.4.5 Activity-Related Variable Choice Heuristics

The use of clause learning has a further useful side-effect. It appears that variables used in deriving learned clauses often make good candidates when we need to choose the next variable to assert. This has led to the development of a large number of variable choice heuristics, the best-known being the *Variable State Independent Decaying Sum (VSIDS)* heuristic (Moskewicz *et al.*, 2001). With these heuristics the central idea is to maintain a record of how often each variable has been used recently in the derivation of a learned clause—such measures are usually referred to as a variable's *activity*—and to choose the variable with highest activity to assert next. For a comparison of several related heuristics of this kind see Biere and Fröhlich (2015).

The solver MiniSat has become ubiquitous in applications of ML to SAT. This solver chooses variables using the *Exponential VSIDS (EVSIDS)* method. It assigns to each variable v in a problem instance an *activity* $a(v)$, initialized to zero. At each conflict, the variables appearing in a learned clause have their activities incremented as

$$a(v) \leftarrow a(v) + u$$

and at each conflict u is increased as

$$u \leftarrow u \times u'$$

where u' takes a value typically between 1.01 and 1.2, and which in some variations may vary. Activities $a(v)$ and the value u are re-scaled when necessary to avoid numerical overflow.

Thus, variables involved in more recent conflicts attain higher activities, and when choosing the next variable to assign, the variable with the highest activity is chosen and set to F first.

These heuristics are often used only to choose the variable on which to branch, and not its polarity. A common method for making the latter choice is simply to set it to whichever polarity was used last time it was assigned, as suggested by Pipatsrisawat and Darwiche (2007). However some solvers take a different approach, for example by always asserting the negative literal.

More recent developments have been made in improving variable choice heuristics. As these rely on the introduction of ML we defer further discussion to Chapter 7.

2.4.6 Clause Forgetting

While clause learning has definite benefits, it also has an important shortcoming: it is possible to learn so many new clauses that memory becomes a limiting factor. It is therefore common also to forget learned clauses on a periodic basis. This is often achieved by keeping a record of each clause's activity, in a way analogous to that of variable activity—clauses used often are considered more active.

MiniSat maintains an activity score $a(c)$ for each learned clause c. An activity $a(c)$ is incremented each time c is used to analyse a conflict, and the size of the increment increased over time as in the case of the variable activities. The activities are used to prune learned clauses at various points in the search, with learned clauses having small $a(c)$ being discarded.

More recently, alternative notions of how the quality of a clause might be assessed have received scrutiny, and a particularly significant outcome has been the introduction of the *Literals Blocks Distance (LBD)* measure of clause quality (Audemard and Simon, 2009). When a clause is learned, its literals each have a level at which they were assigned. The LBD of a clause is the number of levels that appear, and clauses with a low LBD are preferred.

2.4.7 Restarting

We noted above that local search solvers can employ restarts as part of their search process. Restarts can also be effective in the context of CDCL solvers. Here, restarting involves simply stopping the search process at some point, typically after a specified number of conflicts, and starting again. While this initially seems an unlikely way to make any improvement, the key idea is that we keep all learned clauses, variable and clause activities, and any other relevant information inferred by the searches made so far. It has been established that frequently restarting can lead to faster successful searches, and numerous methods have been explored for deciding when to force a restart. The best-known restart method, and a common one in practice, is the *Luby Restart* originally devised for minimizing the expected

time taken by a randomized algorithm (Luby *et al.*, 1993). Other methods of varying levels of sophistication have also been employed, see for example Biere (2008a).

We will have further need to discuss the Luby restart sequence in Section 7.4. Say we have an algorithm whose runtime $t(x)$ for input x is randomized, having probability distribution $p(t)$. Instead of starting the algorithm and just waiting for a solution, we run it for a time t_1. If it produces a result within this limit we are done, otherwise we start an independent run for time t_2, and so on. For a sequence $\mathbf{t} = (t_1, t_2, \ldots)$ we are interested in the expected value of the resulting total runtime $T(\mathbf{t}, x)$. Luby *et al.* (1993) showed that if $p(t)$ is known then the *optimal* sequence

$$\hat{\mathbf{t}} = \operatorname*{argmin}_{\mathbf{t}} \mathbb{E}[T(\mathbf{t}, x)]$$

is uniform, such that $\hat{\mathbf{t}} = (\hat{t}, \hat{t}, \ldots)$ for a specified value $\hat{t}$. The value $\hat{t}$ is

$$\hat{t} = \operatorname*{argmin}_{t} \mathbb{E}[T((t, t, \ldots), x)] = \frac{1}{P(t)} \left(t - \int_o^t P(T) dT \right) \tag{2.2}$$

where P is the cumulative distribution function corresponding to p. In practice $p(t)$ is of course rarely known however. Let $\hat{T} = \mathbb{E}[T(\hat{\mathbf{t}}, x)]$ be the expected total runtime for the optimal sequence. The Luby sequence $\mathbf{t}_L$ is non-uniform, but has the property that its expected total runtime is $O(\hat{T} \log \hat{T})$ for any $p(t)$.

While the Luby sequence is widely known and understood, its use has fallen out of favour in the SAT-solving world since the introduction of methods allowing more frequent restarts, in particular those pioneered by the Glucose (Audemard and Simon, 2018) solver. This development has more recently been challenged by Oh (2015), whose approach is described in Section 2.4.9.

2.4.8 Cache-Friendly Data Structures

In implementing a SAT solver there are various points at which suitable data structures need to be chosen. While it might appear that a sophisticated data structure would be be preferred, it has been found that often the best performance can be attained using carefully implemented array-type data structures. This is because their default layout across adjacent memory locations leads to a better cache hit rate: locality of reference seems to be crucial here. In practice, SAT solvers now pay careful attention to optimizing cache performance, often to the extent that they will implement their own memory management.

This issue is studied in detail by Chu *et al.* (2010), where data structures for various parts of the process are considered. An example of how attention to the cache can be beneficial is as follows. Often in detecting unit propagations, it is necessary only to examine the first few literals in a clause. A clause can thus be represented using a data structure specifically aligned to the cache, where a single cache line contains the first few literals along with a pointer to the remainder of the clause.

2.4.9 Recent Developments

Should satisfiable (S) instances be treated differently to unsatisfiable (U) instances? For some time it has been known that these classes appear to have contrasting characteristics, but research on improving CDCL solvers has mostly proceeded without differentiating them. Oh (2015) argues that this has been counterproductive, and provides the origin of a method, now known as *phasing*, that attempts to address the difference. In this work, it is shown that, for problems in the industrial domain, different approaches to learned clauses, restarts, and variable selection can lead to better performance for S or U instances. Specifically:

1. Learned clauses are more significant in proving U instances, while variable choice is more important for S instances.

2. Luby restarts can perform better than more frequent restarts for S instances.

3. Different settings governing the VSIDS heuristic can be appropriate under different circumstances.

Evidence is then provided showing that improvements can be achieved by mixing, during a solution attempt, phases in which different approaches are used. For example, a solver might alternate between phases with and without restarts, and use different approaches to VSIDS for each phase.

2.5 Portfolio Solvers

The third broad class of SAT solvers that has benefited from the use of ML is that of *portfolio solvers*, with the best-known example being SATzilla (Xu *et al.*, 2012c). The reader might reasonably suspect by now that different approaches to SAT solving have complementary strengths and weaknesses. This is indeed the case, and so the question arises of how, given a problem of interest, we might choose a specific solver with which to address it.

In a portfolio solver the inclusion of ML focuses on learning to select, from a portfolio of available SAT-solvers, which solver to use given the problem at hand. For example, if we have a portfolio $S = \{s_1, \ldots, s_n\}$ of n solvers, we might apply ML to learn n functions $\{h_1, \ldots, h_n\}$. Given a problem p, $h_i(p)$ is then an estimate of the time solver s_i will take to solve p, and these estimates can be used to choose the most promising solver.

This idea can be expanded such that if we have a portfolio $S = \{s_1, \ldots, s_n\}$ of solvers and a time limit T, we can learn to construct a *schedule* based on the problem at hand. Here, a schedule takes the form of a permutation π of $\{1, \ldots, n\}$ and a sequence $\{t_1, \ldots, t_m\}$

of times where $m \leq n$ and

$$\sum_{i=1}^{m} t_i \leq T.$$

Given such a schedule, we run solver $s_{\pi(1)}$ with time limit t_1. If it is successful, we are done, otherwise we run $s_{\pi(2)}$ with time limit t_2, and so on.

A considerable degree of success has been achieved in applying ML to the construction of portfolios for SAT, and we describe this work in Chapter 6.

2.6 Standard Input File Formats

SAT and QSAT solvers need to read an input describing the problem to be solved, typically from a file. Researchers have converged on a small number of file formats that are now standard:

- The *DIMACS* format describes SAT problems in CNF form. Its description can be found at: www.satcompetition.org/2009/format-benchmarks2009.html.
- The *QDIMACS* format extends the DIMACS format to a CNF-based description of QSAT problems. Its description can be found at: www.qbflib.org/qdimacs.html.
- The *QCIR* format describes QSAT problems in a circuit-based form. Its description can be found at: www.qbflib.org/qcir.pdf.

3 Machine Learning

While theorem-proving and ML are very different fields, they share the characteristic of having a vast associated literature. As with the previous chapter, we can not attempt to be comprehensive in the following introduction to ML. Rather, having presented an outline of the SAT-solving world for the benefit of ML researchers, the present Chapter aims to provide a complementary presentation of ML for researchers in SAT and the wider ATP world.

Historically, the majority of deployed applications of ML have been of the *supervised learning* kind (Section 3.1), and consequently we shall cover supervised learning in the greatest detail. It is only quite recently that *unsupervised learning* (Section 3.2) and *reinforcement learning (RL)* (Section 3.4) have started to gain more widespread application.[1] The subject of *multi-armed bandits* (Section 3.3) can be seen as an introduction to the more

[1] Other approaches to ML, in particular *semi-supervised learning* and *active learning* have so far not been applied in the SAT and QSAT spheres.

general idea of reinforcement learning; however, it has an important rôle to play specifically in the application of ML to SAT-solvers, so we give it its own Section.

Neural Networks (NNs) (Section 3.5) have alternated several times between periods of great popularity and deep unfashionability, while currently enjoying the former status. They can be applied to all the variants of the ML problem mentioned so far. Evolutionary methods, particularly genetic algorithms and genetic programs (Section 3.6), are essentially optimization procedures that can be applied to ML; as they have very specific characteristics we introduce them separately.

Finally, applications of ML depend fundamentally on the availability of good data. Over many years, SAT and QSAT researchers have built extensive collections of problems that can be used for this purpose (Section 3.8).

3.1 Supervised Learning

We begin by providing a very brief introduction to the fundamental idea of supervised learning, and to some of the algorithms most commonly used in this area; for a more detailed, but gentle introduction see Mitchell (1997), and for an in-depth view see Murphy (2012).

3.1.1 Feature Engineering

To ground the discussion with a relevant example, consider the problem of deciding directly the satisfiability of an arbitrary Boolean formula f. (This is a problem that has been studied extensively, as we shall see in Chapter 5.) The formula f can be encoded as an *instance* $\mathbf{x} \in \mathbb{R}^n$, the n elements of which are called *features*. Traditionally the ML practitioner has needed to find an effective way of transforming a CNF, or other logical representation of f, into the vector $\mathbf{x}$, using their domain knowledge about the problem at hand. (And often a considerable degree of trial, error, and experimentation.) This process is commonly called *feature engineering*.

Features, when engineered in this way, are generally chosen according to the problem domain; in this case for example we might characterise f by features such as the average size of its clauses, the number of variables and so on. As we shall see in Chapter 4, the problem of SAT solving is unusual in that, unlike most supervised learning problems, there exists a quite standardized set of features, and this set (or a subset) has been used in multiple applications of ML to SAT.

While traditionally features have been chosen by the designer, in recent years there has been a move more generally in ML towards methods that work directly on a representation of the input, inferring relevant features without human intervention. We address this issue

at greater length in Chapter 4, but for now it better fits the discussion to assume that features are derived from f in the more traditional way.

3.1.2 Classifiers

Let $\mathcal{F}$ be the space of all Boolean formulas. A method as described for turning formulas into feature vectors can be regarded as a function $F : \mathcal{F} \to \mathbb{R}^n$. Once an arbitrary formula f can be represented using a set of features $\mathbf{x} = F(f)$, solving SAT then becomes the problem of designing a *classifier* $h : \mathbb{R}^n \to \{0, 1\}$ such that $h(\mathbf{x}) = 1$ for any instance derived from a satisfiable f and $h(\mathbf{x}) = 0$ otherwise. The classifier h is not designed by hand; rather, we gather a collection of *training data*. To do this we take m formulas

$$(f_1, \ldots, f_m)$$

where in each case we know whether or not the formula is satisfiable, and label the corresponding instances according to satisfiability. This results in the *training sequence*

$$\mathbf{s} = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m))$$

with $\mathbf{x}_i = F(f_i)$ and $y_i \in \{0, 1\}$ for $i = 1, \ldots, m$. A *learning algorithm* takes $\mathbf{s}$ and outputs h. Let $\mathcal{H}$ be a set, often called the *hypothesis space*, containing all allowed functions h. The learning algorithm is in essence a function

$$\mathbb{A} : \bigcup_{i=1}^{\infty} (\mathbb{R}^n \times \{0, 1\})^i \to \mathcal{H}.$$

3.1.3 Generalization

It is critical to note that h is a *function*. As a result, it will provide a value in $\{0, 1\}$ *for instances derived from formulas that were not represented in* $\mathbf{s}$. We might reasonably expect that h does not always provide the correct answer, but the ability to provide the correct answer in the majority of cases is referred to as *generalization*.

Can this possibly work? How can we hope to choose h on the basis of $\mathbf{s}$ such that it can be successful in predicting outcomes for formulas the learning algorithm has never seen?

This is a deep and interesting issue. Put simply: the relationship between instances and their labels is not random. As a result, the examples in $\mathbf{s}$ can form the basis for a statistical model of a general probability distribution on the space $\mathbb{R}^n \times \{0, 1\}$—we shall see a simple example of this below. By carefully controlling the nature of the function $h \in \mathcal{H}$ that $\mathbb{A}$ picks, in particular properties relating to its *smoothness*, generalization can be achieved.

This is of course a very simplified explanation, but the reader can, for the purposes of understanding the rest of this review, safely assume that modern, standard methods for ML can achieve a usable, even profound level of generalization. The reader in need of further

justification can find extensive theoretical models of the process based on *computational learning theory* (Anthony and Bartlett, 2009), *empirical risk minimization* (Vapnik, 2006) and *statistical physics* (Engel and Broeck, 2001), amongst other areas.

3.1.4 Classifiers Making Non-Binary Predictions

As we do not expect the *classifier* h to guarantee a correct answer, we might wish to relax the requirement that it produces a prediction in $\{0, 1\}$. The problem of predicting a real number—that is, learning a function $h : \mathbb{R}^n \to \mathbb{R}$—is known as *regression*. For example, we might then decide satisfiability by predicting $f \in \text{SAT}$ if $h(\mathbf{x}) > 0$ and $f \notin \text{SAT}$ otherwise, while using $|h(\mathbf{x})|$ as a measure of the *certainty* of the prediction.[2] Regression has also proved useful for tasks such as predicting the time that a solver might take to solve a problem; this idea is central to the construction of solver portfolios, and we explore it further in Chapter 6.

We can also formulate the problem as one of predicting the *probability* that $f \in \text{SAT}$, allowing h to be a function $h : \mathbb{R}^n \to [0, 1]$ with the interpretation

$$h(\mathbf{x}) = \Pr(f \in \text{SAT}|\mathbf{x}) = \Pr(C = 1|\mathbf{x}) \tag{3.1}$$

where we have introduced the random variable C used to denote class. For a classification problem with two classes, thresholding h at $1/2$ then corresponds in certain circumstances to the *Bayes-optimal classifier* (see for example Duda *et al.* (2000)).

We now provide three examples of simple classification algorithms.

Example 1: The *naïve Bayes classifier* represents perhaps the easiest approach to learning a classifier of the form $h(\mathbf{x}) = \Pr(C = 1|\mathbf{x})$. While it makes very strong assumptions regarding the underlying probability distribution governing the generation of labelled data, it often works well in practice and has the advantage of being computationally lightweight—a property that, as we shall see, is often of great significance when attempting to prove theorems.

We can in principle compute the value $\Pr(C|\mathbf{x})$ as

$$\Pr(C|\mathbf{x}) = \frac{1}{Z}\Pr(\mathbf{x}|C)\Pr(C)$$

where

$$Z = \sum_c \Pr(\mathbf{x}|c)\Pr(c)$$

[2] A *Bayesian* approach to ML allows us to treat the issue of certainty more rigorously, by inferring a *distribution* on the output value, and using the variance of this distribution as an indicator of certainty. We will not pursue this subject further here; see Bishop (2006) for details.

normalises the distribution. While the prior class probabilities $\Pr(C)$ can in general be estimated easily by observing the numbers of training examples in $\mathbf{s}$ corresponding to each class, the distribution $\Pr(\mathbf{x}|C)$ will be much harder to obtain—if we wish to estimate its value for a specific $\mathbf{x}$ then we are likely to need a great deal of data in $\mathbf{s}$. The naïve Bayes classifier therefore makes the assumption that features are conditionally independent given the class

$$\Pr(\mathbf{x}|C) = \prod_{i=1}^{n} \Pr(x_i|C).$$

The estimation of the distributions $\Pr(x_i|C)$ from $\mathbf{s}$ is in some cases significantly more straightforward, particularly when the features x_i are discrete; Mitchell (1997) for example describes the use of this form of classifier in the automated classification of news stories. □

Example 2: A second simple but often very effective machine learning algorithm is the *nearest neighbour* algorithm, or its close relative the *k-nearest neighbours* algorithm. Consider a training sequence $\mathbf{s}$ as already described. In order to classify a new instance $\mathbf{x}$ the nearest neighbour algorithm works as follows: find the example $(\mathbf{x}_i, y_i)$ in $\mathbf{s}$ with $\mathbf{x}_i$ closest to $\mathbf{x}$ according to some metric, then classify $\mathbf{x}$ using y_i.

This idea can easily be extended to take account of the k instances $\mathbf{x}_i$ closest to $\mathbf{x}$. For example, let $S_{\mathbf{x}}$ be the set of the k examples in $\mathbf{s}$ that are closest to $\mathbf{x}$. Let $S_y = \{(\mathbf{x}_i, y_i) \in S_{\mathbf{x}} | y_i = y\}$. Then for a two-class classification problem with labels in $\{0, 1\}$ we might define the classifier h as

$$h(\mathbf{x}) = \begin{cases} 0 & \text{if } |S_0| > |S_1| \\ 1 & \text{otherwise} \end{cases}.$$

Algorithms of this kind have been very extensively studied and applied; many variations can be constructed using different metrics, different ways of taking the neighbouring examples into account, and so on. For a comprehensive treatment see Devroye *et al.* (1996). □

Many learning algorithms do not output a classifier h directly; rather, they output a vector of parameters, usually called *weights*, that define h implicitly. This kind of learning algorithm is the norm for NNs, and the simplest possible NN is equivalent to the method of *linear regression*.

Example 3: In linear regression we learn a function

$$h(\mathbf{x}, \mathbf{w}) = w_0 + \sum_{i=1}^{n} w_i x_i \tag{3.2}$$

where[3]

$$\mathbf{x}^T = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix}$$

and

$$\mathbf{w}^T = \begin{bmatrix} w_0 & w_1 & \cdots & w_n \end{bmatrix}.$$

It is notationally convenient to assume that $\mathbf{x}$ is always prefixed with an extra element equal to 1, so

$$\mathbf{x}^T = \begin{bmatrix} 1 & x_1 & \cdots & x_n \end{bmatrix}$$

and Equation (3.2) can be re-written

$$h(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \mathbf{x}. \tag{3.3}$$

Learning now corresponds to mapping the training sequence $\mathbf{s}$ to the vector $\mathbf{w}$. To do this we can choose $\mathbf{w}$ to minimize some measure of the *error* that h makes when trying to classify the available training data. For example, define the error

$$E(\mathbf{w}) = \sum_{i=1}^{m} \left(h(\mathbf{x}_i, \mathbf{w}) - y_i\right)^2. \tag{3.4}$$

The training algorithm for linear regression should then choose

$$\mathbf{w}_{\text{opt}} = \underset{\mathbf{w}}{\text{argmin}}\, E(\mathbf{w})$$

and is straightforward to derive as it corresponds to solving a set of linear equations—we present a more general version below.

Equation (3.2) clearly describes a linear relationship between input and output. In order to make the relationship nonlinear while maintaining the overall simplicity of the method, it is common to introduce p *basis functions* $\phi_i : \mathbb{R}^n \to \mathbb{R}$ and modify Equation (3.2) to

$$h(\mathbf{x}, \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}) \tag{3.5}$$

where

$$\boldsymbol{\phi}^T(\mathbf{x}) = \begin{bmatrix} 1 & \phi_1(\mathbf{x}) & \cdots & \phi_p(\mathbf{x}) \end{bmatrix}.$$

A drawback of this approach, when applied directly, is that it can suffer from *overfitting*; that is, the tendency if the data is noisy to fit it *too* closely, in essence attempting to learn the noise. *Ridge-regression* modifies the learning algorithm to compute

$$\mathbf{w}_{\text{opt}} = \underset{\mathbf{w}}{\text{argmin}} \left(E(\mathbf{w}) + \lambda ||\mathbf{w}||\right) \tag{3.6}$$

[3] It is now the norm in the literature on ML to assume vectors are column vectors, hence this perhaps unfamiliar notation.

where $\lambda \in \mathbb{R}$ is a further parameter that can be set experimentally, typically using a method such as *crossvalidation* (Kohavi, 1995) to estimate the generalization performance expected for a selection of values.[4] It is straightforward to show that the solution to Equation (3.6) is

$$\mathbf{w}_{\text{opt}} = \left(\mathbf{\Phi}\mathbf{\Phi}^T + \lambda \mathbf{I}\right)^{-1} \mathbf{\Phi}\mathbf{y}$$

where

$$\mathbf{\Phi} = \begin{bmatrix} \boldsymbol{\phi}(\mathbf{x}_1) & \cdots & \boldsymbol{\phi}(\mathbf{x}_m) \end{bmatrix}$$

and

$$\mathbf{y}^T = \begin{bmatrix} y_1 & \cdots & y_m \end{bmatrix}.$$

In the case where we wish to perform classification rather than regression, we can for example address two-class problems with labels in $\{0, 1\}$ by modifying Equation (3.5) to

$$h(\mathbf{x}, \mathbf{w}) = \sigma\left(\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x})\right). \tag{3.7}$$

If

$$\sigma(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

then we have the *perceptron*; in this case there are numerous applicable training algorithms (Duda *et al.*, 2000). To obtain a probabilistic output we can use a function such as the *sigmoid* function

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{3.9}$$

In this case the error defined by Equation (3.4) is no longer appropriate and is generally replaced by the *cross-entropy loss*

$$E(\mathbf{w}) = -\sum_{i=1}^{m} \left(y_i \log h(\mathbf{x}_i) + (1 - y_i) \log(1 - h(\mathbf{x}_i))\right). \tag{3.10}$$

There is no longer a closed-form solution to the corresponding minimization problem, however a solution can be found iteratively using the *iterative re-weighted least squares* algorithm (Bishop, 2006). □

We will at some points in the following also need to refer to a classifier known as the *Support Vector Machine (SVM)* (Shawe-Taylor and Cristianini, 2000). This is based on the idea that, taking Equations (3.7) and (3.8) as a starting point, the two classes are divided by a hyperplane in $\mathbb{R}^p$, the position and orientation of which depend on $\mathbf{w}$. The SVM chooses $\mathbf{w}$ to place this hyperplane *as far away as possible* from the training examples, while still classifying them correctly. This idea can be adapted to situations where the training data are not separable.

[4] Parameters such as λ are called *hyperparameters* and their careful estimation is critical. In training NNs there can be many hyperparameters, and the process of estimating expected performance for many combinations of hyperparameter values is referred to as *grid-search*.

3.1.5 Supervised Learning Versus Clause Learning

In Section 1.2 we noted that the term 'learning' has distinct meanings, depending on whether the context involves machine learning or clause learning. It is worthwhile at this point briefly to expand on the relationship between these ideas.

One way in which we can model supervised learning is to propose the existence of a *target function* $t : \mathbb{R}^n \to L$ where L denotes an appropriate space of labels. This function is never revealed to our learning algorithm. It is however used to generate the training set $\mathbf{s}$, using some process to generate feature vectors $\mathbf{x}_i$, and labelling them as $y_i = t(\mathbf{x}_i)$. The labels y_i may additionally be perturbed by noise, however the training set ultimately provides a limited representation of the function t. The purpose of supervised learning is to infer a hypothesis h, based on $\mathbf{s}$, that represents t well even in areas not represented by $\mathbf{s}$. Each time a new example is added to $\mathbf{s}$, we expect to be able to improve the quality of the inferred h. We are, in essence, attempting to reconstruct t in the form of h, using the incomplete information represented by $\mathbf{s}$.

In the case of clause learning, assume we have a CNF formula f. Whenever a solver generates a partial assignment leading to a conflict, one or more clauses can be learned and added to f; thus over time the problem takes the form

$$f \wedge \bigwedge_{i=1}^{n} c_i$$

where the c_i are learned clauses. These clauses are derived using the partial assignments, which in essence act as training examples. As the learned clauses are implied by f, their addition does not alter the meaning of the problem, but they do encode knowledge about the problem uncovered by the partial assignments and resulting conflicts, and which is useful in the ongoing proof search. We can thus regard the conjunction $c_1 \wedge \cdots \wedge c_n$ as analogous to h in the ML context.

3.2 Unsupervised Learning

In unsupervised learning we attempt to infer structure in data without the assistance of labels as in supervised learning. Given m examples, often in the form of vectors $\mathbf{x}_i \in \mathbb{R}^n$, we typically aim to identify *clusters* within the data. Numerous algorithms are available that aim to solve this problem. For example, denote by $N(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ the multivariate normal density with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. Assume that the $\mathbf{x}_i$ are drawn at random from a probability distribution with density $p(\mathbf{x})$ of the form

$$p(\mathbf{x}) = \sum_{i=1}^{K} \pi_i N(\mathbf{x}; \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i) \tag{3.11}$$

where $\pi_i \geq 0$, $\sum_{i=1}^{K} \pi_i = 1$. Equation (3.11) is a natural description of a density that will produce data from K clusters centred on the $\boldsymbol{\mu}_i$ and with shapes described by the $\boldsymbol{\Sigma}_i$.

Given only this model and m samples $(\mathbf{x}_1, \ldots, \mathbf{x}_m)$, how might we then infer the parameters $\boldsymbol{\mu}_i$, $\boldsymbol{\Sigma}_i$ and π_i? Collecting the parameters as $\boldsymbol{\theta} = \{\pi_i, \boldsymbol{\mu}_i, \boldsymbol{\Sigma}_i | i = 1, \ldots, K\}$ one approach is to maximize the *likelihood*

$$L(\boldsymbol{\theta}) = \prod_{i=1}^{m} p(\mathbf{x}_i).$$

This can be achieved using the *EM algorithm*; for the details see Bishop (2006).

3.3 Multi-Armed Bandits

Multi-armed bandits provide the simplest possible model for a learning task where an agent learns by interaction with the environment, rather than simply by observing it. We provide only a minimal introduction; see Sutton and Barto (2018) for further details.

Imagine you are faced with n 1-armed bandit machines. Each has a probability distribution over the rewards payed out when it is played, and this distribution is not known to you. You are given m plays and asked how you might choose a sequence of machines to play in order to optimize the reward obtained. If the reward distributions were known to you it would make sense to play only the machine with the highest expected reward; but when these distributions are unknown you need to try some machines to learn how they perform.

You might for example try a small number of machines and find that one gives you some large rewards. One approach is then to *exploit* this machine by continuing to play it. Alternatively, you might forego playing this machine, and try out others in the hope of finding an even better one. Thus you would *explore* the characteristics of the machines in an attempt to gain better rewards. When the distributions of rewards are unknown to you the problem becomes one of how to balance exploration and exploitation.

The problem becomes more complicated if the reward distributions can evolve over time, in which case rewards gained recently should carry more weight in our strategy than those gained earlier in the process. A simple idea then is to compute weighted averages. If a machine gives a sequence $r_1, r_2, \ldots, r_n$ of rewards we might compute the weighted average

$$\hat{r}_n = \sum_{i=1}^{n} (1-\alpha)^{n-i} \alpha r_i$$

where $0 < \alpha < 1$ is a *discount* parameter. This can be achieved without storing the entire reward sequence as

$$\hat{r}_{n+1} = (1-\alpha)\hat{r}_n + \alpha r_{n+1}.$$

This is known as the *exponential recency weighted average (ERWA)* algorithm.

Space again precludes any comprehensive presentation of algorithms for learning to play multi-armed bandits. We limit ourselves to an example of a simple algorithm with provable performance guarantees. The discounted *Upper-Confidence Bound (UCB)* algorithm (Garivier and Moulines, 2011) addresses multi-armed bandit problems where the distribution of rewards is stationary for periods of time but occasionally changes. It takes the idea of discounting rewards further, and in doing so provides one of many algorithms for addressing multi-armed bandit problems. Once again, a discount factor $0 < \gamma < 1$ is introduced, allowing immediate rewards to be treated as more valuable than delayed rewards. Let $\mathbb{I}$ denote the indicator function, such that for a predicate P, $\mathbb{I}(P) = 1$ when P is true and $\mathbb{I}(P) = 0$ otherwise. Let $r_{i,t}$ be the reward obtained from playing arm i at time t, and let p_t denote the arm played at time t. Then the discounted UCB algorithm estimates for the arm i

$$\hat{r}_T(i) = \frac{1}{N_T(i)} \sum_{t=1}^{T} \gamma^{T-t} r_{i,t} \mathbb{I}(p_t = i)$$

where

$$N_T(i) = \sum_{t=1}^{T} \gamma^{T-t} \mathbb{I}(p_t = i).$$

It then introduces a further term

$$c_T(i) = 2b\sqrt{\frac{\xi \log N_T}{N_T(i)}}$$

where b is a bound on the size of the rewards, ξ is a parameter and

$$N_T = \sum_{i=1}^{n} N_T(i).$$

Finally, it plays the arm maximizing $\hat{r}_T(i) + c_T(i)$.

3.4 Reinforcement Learning

In a multi-armed bandit system our only choice in accumulating reward is which arm to choose. In reinforcement learning our aim is still to accumulate reward, but the actions we take can change the state of the world, and rewards can be state and action dependent.

Our aim now is to learn a *policy* $p : \mathcal{S} \to \mathcal{A}$ where $\mathcal{S}$ is a set of *states* and $\mathcal{A}$ is a set of *actions*. Taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ results in us moving to a new state $s' \in \mathcal{S}$ and receiving a reward $r \in \mathbb{R}$; we do not however know at the outset how new states and rewards are assigned, and in general there are probability distributions that govern the assignments. Our aim is to learn a policy that maximizes some measure of the rewards

accumulated over time, typically the expected value of the quantity

$$R_i = r_i + \gamma r_{i+1} + \gamma^2 r_{i+2} \cdots = \sum_{j=0}^{\infty} \gamma^j r_{i+j} \tag{3.12}$$

where r_i denotes the reward received at the ith step and $\gamma \in [0, 1]$ is a discount factor used to emphasise near-term rewards over those received some time in the future, in exactly the same way as in a multi-armed bandit problem.

Once again, there are numerous algorithms that can be used to infer a policy through interaction with the environment; see Sutton and Barto (2018) for a detailed introduction. For example, in *Q-learning* we attempt to learn a function $Q(s, a)$ that estimates the reward attained by performing action a in state s and thereafter acting optimally. The corresponding policy then simply selects the a maximizing $Q(s, a)$ for the current state s.

3.5 Neural Networks

It is to some extent unnecessary to give NNs a section separate from those preceding, as they can be applied in some form to all of the learning tasks seen so far, either as self-contained methods or as components of larger systems. However the proliferation of architectures now available, aimed at diverse elements of the ML problem, makes it convenient to treat them in a single place.

NNs have been applied in many ways in attempts to improve SAT and QSAT solvers, and these attempts have covered the full spectrum of complexity available. In this section we present a short introduction to several NN architectures, attempting to give a flavour of the main ones used to date. In reading it, we advise the non-specialist to be mindful of two important points:

1. We mentioned above that a training algorithm can specify a hypothesis implicitly by specifying a set $\mathbf{w}$ of parameters, or *weights*. Hence for example, in Equation (3.2) a linear function was denoted $h(\mathbf{x}, \mathbf{w})$, where $\mathbf{x}$ is an instance. NNs can for our purposes almost always be regarded as a way of specifying parameterized hypotheses that are much more flexible than the linear regressor.

2. The design of these, more flexible hypotheses can be regarded as a process of selecting the correct building blocks and combining them in an effective way. Different building blocks, such as multilayer perceptrons, convolution layers, long short-term memories and so on are designed to address particular kinds of problem.

3.5.1 The Perceptron

The simplest possible neural network is illustrated in Figure 3.1, and corresponds to the

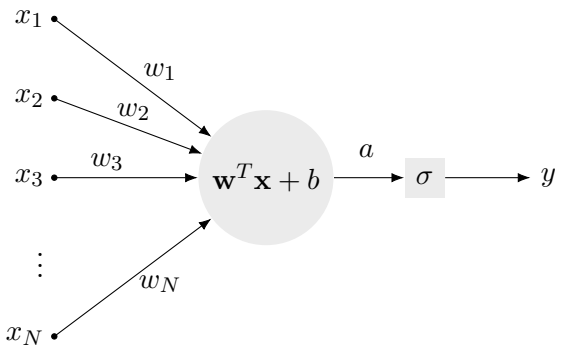

Figure 3.1: Representation of the perceptron model. The N inputs are $x_1, \ldots, x_N$, collected into the vector $\mathbf{x}$. A linear function is formed from these, parameterized by the weights $w_1, \ldots, w_N$ and the *bias* b. The value of this function is the activation a, and is modified by the activation function σ to produce the output y.

linear model of Equation (3.2), usually with the edition of a nonlinearity σ known as the *activation function*, and taking the shape of a step or a smoothed version of a step as discussed above.[5] The value

$$a = \sum_{i=1}^{N} w_i x_i + b$$

is referred to as the *activation*. We will not discuss these simple models further, other than to note that they form the basic building block for the *multi-layer perceptron (MLP)*.

3.5.2 The Multilayer Perceptron, and Friends

By connecting many perceptrons into a more complex structure we obtain a *multilayer perceptron (MLP)*. The typical format is illustrated in Figure 3.2. There are two points to note here. First, the network is a directed acyclic graph (DAG), beginning on the left with the input nodes, and with one or more output nodes on the right. Second, it has a layered structure: input nodes connect directly to the first layer, these nodes connect directly to the second layer, and so on. It is entirely possible to deal with more complex structures—for example by allowing connections to skip layers. While this is not unusual in the more recent work discussed below, we confine ourselves for the time being to this structure, which provides a useful and very common basis.

Each node in the MLP is itself a perceptron as described in the previous section. By collecting together the weights and biases of all the nodes in the structure into a single

[5] Henceforth in this work we will use σ to denote the *sigmoid* function defined by Equation (3.9). In the literature, many functions with a similar shape are used, depending on the circumstances, and these will be introduced as required and denoted using a different symbol.

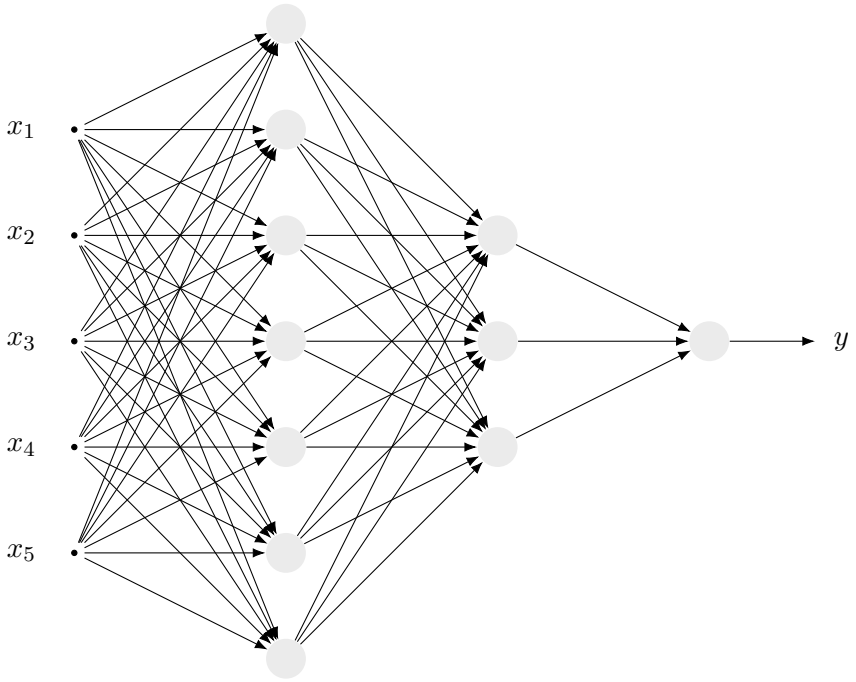

Figure 3.2: Representation of a multilayer perceptron having five inputs, two hidden layers with seven and three nodes respectively, and a single output layer with one node. Each node in the network is a perceptron as illustrated in Figure 3.1.

vector $\boldsymbol{\theta}$, and assuming for the moment that there is a single output y as illustrated, we can consider the MLP to compute a function $h_{\boldsymbol{\theta}} : \mathbb{R}^n \to \mathbb{R}$ just as in previous sections; however the more complex structure now allows for more expressive functions, depending on the number of layers and the number of nodes in each layer. In any case, the process of computing $h_{\boldsymbol{\theta}}(\mathbf{x})$ by applying an input $\mathbf{x}$ to the network and successively computing the outputs of each of the following layers is called *forward propagation*.

Given a training set $\mathbf{s}$ the network can also be trained in the same manner as already discussed, by minimizing a measure $E(\boldsymbol{\theta})$ of error such as those in Equations (3.4) and (3.10), where we have changed the notation from $\mathbf{w}$ to $\boldsymbol{\theta}$ to emphasise that $\boldsymbol{\theta}$ contains parameters collected from *multiple* perceptrons. However, the resulting optimization problems do not have closed-form solutions and must therefore be addressed using other methods. The key requirement for such methods is to obtain the gradient $\nabla_{\boldsymbol{\theta}} E$ of the error. For example, if this gradient is available then in the simplest form of optimization, namely *gradient descent*, we can try to compute a good vector $\boldsymbol{\theta}$ in an iterative manner by starting at some random solution $\boldsymbol{\theta}_0$ and applying updates

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \mu \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}_i) \tag{3.13}$$

where μ is a parameter denoting step size. Note that as the error $E(\boldsymbol{\theta})$ is almost always a sum of individual terms, one for each example in $\mathbf{s}$, its gradient can also be expressed as a sum of gradients corresponding to individual examples; that is,

$$\nabla_{\boldsymbol{\theta}} E = \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} E_i. \tag{3.14}$$

A variant of gradient descent, known as *stochastic gradient descent* performs updates on a per-example basis using $\nabla_{\boldsymbol{\theta}} E_i$ instead of $\nabla_{\boldsymbol{\theta}} E$. In either case the question remains though of how to compute the gradient.

The process of *backpropagation* refers to the algorithm for finding $\nabla_{\boldsymbol{\theta}} E_i(\boldsymbol{\theta})$ for a single labelled example $(\mathbf{x}_i, y_i)$ and the current parameters $\boldsymbol{\theta}$. Its name alludes to the fact that, once $\mathbf{x}_i$ has been applied to the inputs of the network and forward propagation performed, $\nabla_{\boldsymbol{\theta}} E_i(\boldsymbol{\theta})$ can be computed in a straightforward manner by starting at the output node and working backwards, layer by layer. Each step backwards through the network requires only the use of the chain rule of calculus. This process is illustrated in many existing works on NNs, and we will not repeat it in detail here, referring the reader to Bishop (2006) or one of many alternatives for the details. Similarly, in practice the basic method of gradient descent is usually not sufficient for the training of large networks, and numerous more powerful optimization methods exist; for example a recent, effective development is the *Adam optimizer* (Kingma and Ba, 2015).

Thus far we have limited the discussion to networks that have a single output, such

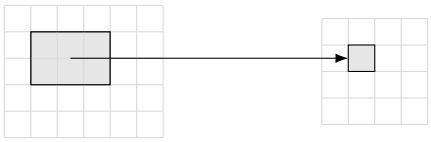

Figure 3.3: Convolutional layer mapping a 5×6 image to a 4×4 image. Each pixel in the smaller image is computed by applying the convolution operation to the larger image using a 2×3 kernel. The weights defining the kernel are the same for all pixels.

that in performing classification we are limited to problems with two classes.[6] Given c classes, ideally we want a learner to provide a distribution over these classes as its output for any new instance $\mathbf{x}$. This is typically achieved using a network with c real-valued output nodes. Denote the values produced by these nodes as $o_1(\mathbf{x}), \ldots, o_c(\mathbf{x})$. These outputs can be transformed to a distribution using the *softmax* function, such that the c final outputs $y_1(\mathbf{x}), \ldots, y_c(\mathbf{x})$ are

$$y_i(\mathbf{x}) = \frac{\exp(o_i(\mathbf{x}))}{\sum_{i=1}^{c} \exp(o_i(\mathbf{x}))}.$$

It is straightforward to adapt the cross-entropy loss and the relevant gradient calculations to this format.

3.5.3 Convolutional Neural Networks

NNs are often used to process images, in which case the instance $\mathbf{x}$ is arranged as a matrix $\mathbf{X} \in \mathbb{R}^{n_1 \times n_2}$. *Convolutional neural networks (CNNs)* map from one layer to the next using a modified computation. Let $\mathbf{K} \in \mathbb{R}^{k_1 \times k_2}$ be a *kernel* matrix, where $k_1 \leq n_1$ and $k_2 \leq n_2$. It is convenient to think of a convolutional layer as computing a new image $\mathbf{X}' \in \mathbb{R}^{n_1 - k_1 + 1 \times n_2 - k_2 + 1}$, as illustrated in Figure 3.3. Pixel $X'_{i,j}$ in the output image is

$$X'_{i,j} = \sum_{l=1}^{k_1} \sum_{m=1}^{k_2} K_{l,m} X_{i+l-1, j+m-1}.$$

In this case the learned parameters are the elements of $\mathbf{K}$, and during learning they are constrained such that each pixel of $\mathbf{X}'$ is computed using the same values; this is easily accounted for in computing the relevant gradients.

3.5.4 Recurrent Neural Networks and Long Short-Term Memory

The learning methods described so far take individual feature vectors $\mathbf{x}$ and map them to one of a finite number of classes. In many problems it is more natural to think in

[6] While we could, for instance, divide a single real-valued output into an arbitrary number of intervals, one for each class, such methods are in general uncompetitive.

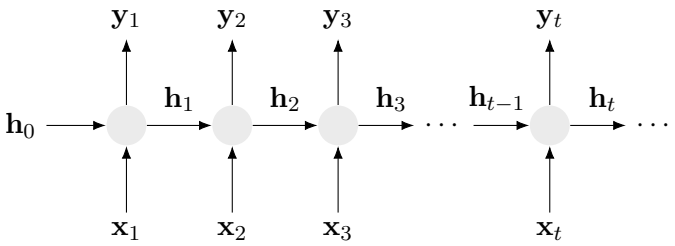

Figure 3.4: Common format for a recurrent neural network. Inputs $\mathbf{x}_t$ at each time step are combined with the state vector $\mathbf{h}_{t-1}$ for the previous time step to produce the current state $\mathbf{h}_t$. The current state is passed forward, and also used to compute the current output $\mathbf{y}_t$. Parameters used during the computation in each node are constrained to be equal for all nodes.

terms of classifying a *sequence* $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t, \ldots$ indexed by the time t, either producing a classification after some final time T, or at each point in the sequence. Many methods have been explored for addressing such problems, and at present the *recurrent neural network (RNN)* is a common solution.

The basic architecture for an RNN is shown in Figure 3.4. Intuitively, each node combines an input vector $\mathbf{x}_t$ with a vector $\mathbf{h}_{t-1}$ representing the *state* carried over from the previous time step, and passes the resulting state $\mathbf{h}_t$ forward. A separate computation computes the output $\mathbf{y}_t$ from the state. For example, a common format for the forward propagation computes

$$\begin{aligned} \mathbf{s}_t &= \mathbf{b}_s + \mathbf{S}_1 \mathbf{h}_{t-1} + \mathbf{S}_2 \mathbf{x}_t \\ \mathbf{h}_t &= \tanh(\mathbf{s}_t) \\ \mathbf{y}_t &= \mathbf{b}_y + \mathbf{Y} \mathbf{h}_t \end{aligned} \tag{3.15}$$

at each step. The critical point to note here is that the parameters contained in $\mathbf{S}_1$, $\mathbf{S}_2$, $\mathbf{Y}$, $\mathbf{b}_s$ and $\mathbf{b}_y$ are *not* indexed by time, but are identical at all time steps. As with CNNs, this requires only minor changes to be made in using backpropagation to compute the gradients needed for learning; essentially, small modifications are needed to constrain the parameters to be identical across time steps, and the resulting process is called *backpropagation through time*.

An unfortunate side-effect of the architecture of the RNN is that the gradients computed can either disappear or grow exponentially. Numerous modified architectures have been proposed to address this, with the *long short-term memory (LSTM)* introduced by Hochreiter and Schmidhuber (1997) having become the dominant model. Once again there are numerous related approaches and variations on the LSTM and we will not attempt to introduce them all here; Hu *et al.* (2019) review the LSTM and some of its many variations. Several recent applications of RNNs to the SAT problem have employed architectures very close to the original LSTM, and so we limit ourselves to presenting this in full.

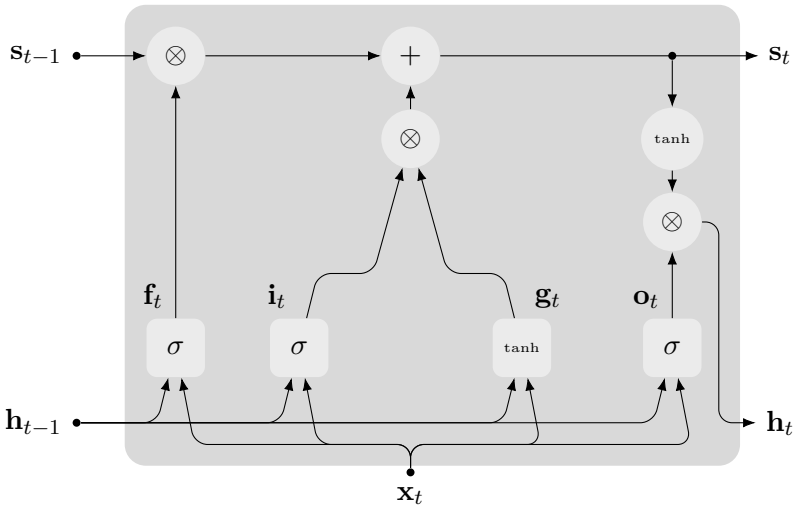

Figure 3.5: Representation of the standard LSTM node. The structure and components of this node are explained in the text.

The central idea involves replacing the RNN nodes described above in Equation (3.15) with the LSTM node illustrated in Figure 3.5. The forward propagation process for this node is defined by the equations

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{b}_f + \mathbf{F}_1\mathbf{h}_{t-1} + \mathbf{F}_2\mathbf{x}_t) \\
\mathbf{i}_t &= \sigma(\mathbf{b}_i + \mathbf{I}_1\mathbf{h}_{t-1} + \mathbf{I}_2\mathbf{x}_t) \\
\mathbf{o}_t &= \sigma(\mathbf{b}_o + \mathbf{O}_1\mathbf{h}_{t-1} + \mathbf{O}_2\mathbf{x}_t) \\
\mathbf{g}_t &= \tanh(\mathbf{b}_g + \mathbf{G}_1\mathbf{h}_{t-1} + \mathbf{G}_2\mathbf{x}_t) \\
\mathbf{s}_t &= (\mathbf{s}_{t-1} \otimes \mathbf{f}_t) + (\mathbf{i}_t \otimes \mathbf{g}_t) \\
\mathbf{h}_t &= \tanh(\mathbf{s}_t) \otimes \mathbf{o}_t
\end{aligned}$$

where $\otimes$ denotes element-by-element multiplication. Each LSTM node now has two state vectors $\mathbf{s}_t$ and $\mathbf{h}_t$ at time t, with $\mathbf{h}_t$ also acting as the output for the node. Intuitively, $\mathbf{s}_t$ passes long-term memory forward, and $\mathbf{h}_t$ short-term memory. The node combines the input $\mathbf{x}_t$ and previous short-term state $\mathbf{h}_{t-1}$ to form four new vectors:

- The vector $\mathbf{f}_t$ allows the node to *forget* elements of the long-term state.
- The vector $\mathbf{i}_t$ represents new information to be incorporated into the long-term state.
- The vector $\mathbf{g}_t$ controls the way in which $\mathbf{i}_t$ is incorporated.
- The vector $\mathbf{o}_t$ is combined with the new long-term state to produce an output.

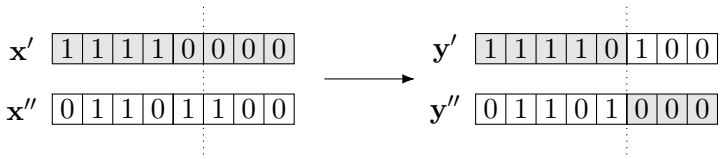

Figure 3.6: Generating new individuals in a GA using crossover. The initial individuals $(\mathbf{x}', \mathbf{x}'')$ are split at a random point—in this case between bits 5 and 6, and their tails swapped to generate the new individuals $(\mathbf{y}', \mathbf{y}'')$.

3.6 Genetic Algorithms and Genetic Programming

Several attempts at combining ML with theorem proving have employed *genetic algorithms (GAs)* (Mitchell, 1998) or *genetic programming (GP)* (Koza, 1992). Both of these methods are general approaches to the wider field of *optimization* (Luenberger, 2003). They are however loosely based on ideas from biological evolution, and rename various terms from the optimization literature accordingly. We shall refer to them under the common term *evolutionary programming (EP)*.

In the usual optimization parlance we have an *objective function* $O : \mathbb{R}^n \to \mathbb{R}$ and aim to find

$$\mathbf{x}_{\text{opt}} = \operatorname*{argmax}_{\mathbf{x}} O(\mathbf{x}).$$

As EP addresses precisely this problem, it should come as no surprise that such ideas have been used as alternatives to traditional optimization methods, such as gradient descent with gradients computed using backpropagation. One of their strengths however is their ability to search for good functions that are not described by, for example, a fixed network structure.

In EP, possible solutions $\mathbf{x}$, rather than necessarily being real vectors, are structured. This allows new algorithms to be devised. For the purpose of the following brief description of a basic GA we will assume individuals are represented as binary strings $\mathbf{x} \in \{0, 1\}^n$.

We have a *fitness function* F assigning a fitness $F(\mathbf{x})$ to any proposed solution $\mathbf{x}$—this is simply a renaming of the usual objective function. We start with a *population* $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ of N *individuals*. These are used to produce the next *generation* of individuals using some variant of the following process:

- Pairs $(\mathbf{x}', \mathbf{x}'')$ of individuals are *selected*, often with a bias towards individuals with a high fitness, and often stochastically.
- These pairs are combined in some way to form a new pair of individuals $(\mathbf{y}', \mathbf{y}'')$. The combination is usually through some variant of *crossover*, as illustrated in Figure 3.6. The new individuals become part of the next generation.

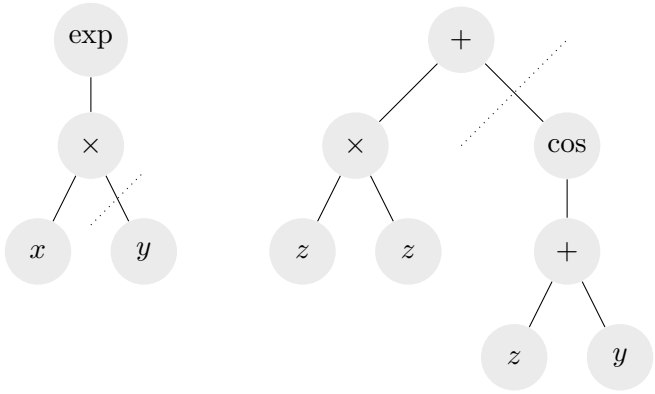

Figure 3.7: Generating new individuals in a genetic program using crossover. Each individual is represented as a tree. In this case the left tree computes $\exp(xy)$ and the right tree $z^2 + \cos(z + y)$. Subtrees are cut at random—in this case at the points indicated by dotted lines—and swapped. The resulting individuals compute $\exp(x \cos(z + y))$ and $z^2 + y$.

- Individuals $\mathbf{x}$ in a new generation may be subject to *mutation*, whereby one or more bits of $\mathbf{x}$ are randomly flipped.

This process is iterated, forming a sequence of new generations, and it is often found that the fitness of the generations improves over time.

In GP the primary difference is that individuals are represented in the form of trees, typically representing simple computations. Crossover is achieved by swapping subtrees, as illustrated in Figure 3.7. The remainder of the approach is essentially unchanged, and we aim to obtain individuals representing more effective computations as the number of generations increases.

3.6.1 Machine Learning Versus Optimization

It is worth emphasizing at this point that EP methods are in fact optimization methods, and not in themselves machine learning methods—at least, when applied without modification. (In many ways they are more akin to *local search* (Russell and Norvig, 2020).) The distinction is an important one.

Earlier in this Chapter we saw that supervised learning using a system such as an MLP can be described as an optimization problem involving the minimization of a function $E(\boldsymbol{\theta})$.[7] Once $\boldsymbol{\theta}$ is determined it implicitly defines the function $h_{\boldsymbol{\theta}} : \mathbb{R}^n \to \mathbb{R}$ used to classify new data. One might reasonably argue that a GP achieves the same result, the only difference

[7] Indeed, a significant body of work exists applying EP methods to precisely this problem.

being that the function obtained is represented as a computation tree, rather than in terms of a structure built from perceptrons. So where does the distinction lie?

In Section 3.1.3 we mentioned the need for a learner to *generalize*. If we were to naïvely train an MLP, GP or other learner only by minimizing its error as measured on a training set, it is very unlikely that we would obtain generalization. This is a phenomenon known as *overfitting* (Bishop, 2006), and it is avoided using one or more of numerous methods. It is this that sets supervised learning apart from straightforward optimization.

Clearly though, there is considerable common ground, and EP methods are included here on account of the extensive literature applying them to SAT and related problems. The reader should nonetheless keep this important point in mind in what follows.

3.7 Choosing a Learning Algorithm

Given the huge variety of learning algorithms available, how does one choose which to use? There is no straightforward answer to this, beyond choosing the method—supervised, unsupervised, reinforcement and so on—to best fit the problem, and thereafter experimenting.

For supervised learners there is some degree of further guidance. Holte (1993) reviews evidence suggesting that in many cases, methods that learn very simple rules can perform comparably to more complex approaches. He then presents extensive experimental results verifying this observation for sixteen data sets, comparing rules that make a classification based on only one feature in an example against more complex classifiers. It is found that the simpler classifier usually suffers by only a few percent in accuracy compared to C4's decision trees (Quinlan, 1986).

While Holte's results were published in 1993, and we might therefore be tempted to suspect that they have limited relevance today, they are in fact reinforced by the more recent study of Fernández-Delgado *et al.* (2014). This work compares 179 classifiers on 121 data sets. They find that overall the best two classifiers are Random Forests (Breiman, 2001) and Support Vector Machines with Gaussian kernels (Shawe-Taylor and Cristianini, 2000; Shawe-Taylor and Cristianini, 2004), and that a rather small number of classifiers dominate the remaining majority.[8] Ultimately the lesson to be learned from extensive research in machine learning is that, regardless of the problem domain, there is no rule for choosing the best algorithm: one must choose algorithms whose computational complexity is appropriate to the problem, and thereafter be guided by experimentation.

The more recent opportunities presented by large, deep NN architectures only serve to make this issue more problematic. The experimental work just described dealt with more standard ML methods; the variety of NN architectures available, and the ease with which

[8]Since this work appeared some doubt has been cast on its conclusion for random forests (Wainberg *et al.*, 2016). These should certainly not be discounted however, and have been used with great success in ATP applications both within and outside of the SAT/QSAT domain.

they can be combined to build hugely complex systems makes specific advice impossible to provide, and the fact that their complexity needs to be balanced against the characteristics of the available data leaves only one conclusion: one must be willing to experiment, and choose what works.

These issues are of course not limited to the application of ML to ATP, and consequently recent work on *Automated Machine Learning (AutoML)* attempts to further automate the process. A description of recent progress can be found in Hutter *et al.* (2019).

3.8 Sources of Data

In order to apply ML in any field of study, a source of training data is required. In SAT and QSAT solving, and indeed in the wider area of ATP, we are lucky in having access to large quantities of good quality data. In particular, for SAT and QSAT:

- The *Satisfiability Library (SATLIB)* (Hoos and Stützle, 2000) contains a large collection of data sets. The library is available at Hoos and Stützle (2019).
- One reason for the good availability of data is the competitive nature of SAT research. The *International SAT Competition* (Heule *et al.*, 2019) allows developers of SAT-solvers to compete using collections of problems that are made freely available.
- A further competition serves a similar purpose for QSAT solvers. Narizzano *et al.* (2006) describe the *Quantified Boolean Formulas Satisfiability Library (QBFLIB)*, which includes a large collection of problems used for competitive evaluations. The library is available at Giunchiglia *et al.* (2005).

It is worth noting that these data collections are not limited to small or synthetic problems. On the contrary, they contain a mixture covering the range from small, algorithmically generated problems to very large instances derived from real industrial problems.

4 Extracting Features from a Formula

In general, in the context of ML research, 'feature engineering' refers to the process of using domain knowledge to design a set of features that the designer expects to be effective in representing problem instances—in the present case, propositional formulas. Accordingly, in most of the work to date, a fundamental need in applying ML to SAT solvers has been a method to extract a set of features from a given formula. More recent NN-based methods, based in particular on *graph neural networks (GNNs)*, have attempted to circumvent this need by using a CNF or other propositional formula directly as input and encoding it as a real-valued vector. In this Chapter we describe both of these approaches.

The SAT problem is in a sense unusual in that there is a single, very widely studied feature that is known to be an indicator of the likely difficulty of an instance: the ratio R of the number c of clauses to the number v of variables. Specifically, in the case of 3-SAT the point at which there is a probability of $1/2$ that a randomly selected instance will be satisfiable occurs at a phase transition with $R = c/v \simeq 4.26$, and instances in this region tend to be hard to solve. See Saitta *et al.* (2011) for further details. We might expect though, that more sophisticated features are needed to address realistic SAT and related problems. This has been verified in practice: features other than R are invariably required.

One of the earliest attempts to apply ML to SAT solvers was described by Ertel *et al.* (1989), and this work provides a very simple example of how features can be obtained. While the problems addressed were small—having at most three variables, four clauses and three literals per clause—the work provided an initial indication that learning might be beneficial in this context. The aim was to show that, when using the prover SETHEO (Letz *et al.*, 1992) to solve propositional problems, it was possible to predict the limit on depth of search needed to allow a proof to be found quickly. CNF formulas were encoded into binary strings indicating whether or not a literal appeared in each clause; this made it possible to present a formula to a NN using twenty-four inputs. Five outputs were used, and again these were binary and indicated that the depth limit should be one, two, and so on up to five. Labelled examples were obtained by randomly generating suitable formulas and using exhaustive search to find labels. A basic NN with a single hidden layer of sixteen units was trained successfully to solve this problem.

We now discuss the approaches used to develop further useful features.

4.1 Feature-Engineered Representations

Many of the features used to describe CNF formulas are based on three graphs.[1] In the following we describe each graph in terms of its nodes N and edges E. We denote by V the set of variables and C the set of clauses in a CNF formula.

1. The *clause-variable incidence graph (CVIG)* is a weighted bipartite graph with $N = V \cup C$ and edges
$$e_{v,c} = \begin{cases} |c|^{-1} & \text{if } v \in c \\ 0 & \text{otherwise} \end{cases}$$
where $v \in V$ and $c \in C$.

[1] The definitions here are those used by Ansótegui *et al.* (2017). Variations on these definitions are also used; for example, Nudelman *et al.* (2004) do not use edge weighting.

2. The *variable incidence graph (VIG)* is a weighted graph with nodes $N = V$ and edges

$$e_{v_1,v_2} = \sum_{\substack{c \in C \\ v_1, v_2 \in c}} \binom{|c|}{2}^{-1}.$$

3. The *clause incidence graph (CIG)* has nodes $N = C$ and edges

$$e_{c_1,c_2} \in E \iff c_1, c_2 \in C \text{ and share a negative literal.}$$

4.1.1 The 'Standard Features'

Perhaps the earliest attempt to define a single, widely effective set of features was that of Nudelman *et al.* (2004) in the context of the *empirical hardness models*, which we will discuss further in Chapter 6. In this work a total of ninety-one features in nine groups were introduced, and some evidence advanced for which might be the most useful in some circumstances. The set of features was expanded to 138 in twelve groups by Xu *et al.* (2012b). Table 4.1 shows the groups of features proposed and gives some examples of each. This collection of features has become sufficiently ubiquitous, and widely used in whole or in part in subsequent research, that from here on we will refer to the collection as the *standard features*.

It is noteworthy that some of the standard features—often referred to as *probing* features—are computed by performing limited runs of a SAT solver. Intuitively, this should represent an effective means of obtaining useful features, as they are derived using the behaviour of a system while it investigates the problem at hand. This approach to feature engineering is also attractive in that it can be applied by a user not well-versed in the subtleties of the SAT problem, by simply running tools that are already available. The use of such features is also not limited to the SAT problem; for example Bridge *et al.* (2014) have employed such features in the domain of first-order logic, and in a non-logical context *landmark features*, which use runs of simple machine learning systems to provide features for selecting more complex learners appropriate to a specific problem, have been employed (Pfahringer *et al.*, 2000).

4.1.2 Features for Industrial Problems

As it became clear that SAT solvers had important potential applications in industrial problems, interest increased in whether such problems have exploitable structure. Ansótegui *et al.* (2017) considered feature engineering with a specific emphasis on problems arising in industry applications. There is indeed evidence that problems of this kind, in contrast to problems such as randomly generated 3-SAT problems for example, possess three specific kinds of structure:

Table 4.1: The *standard features* for SAT instances. The first nine groups were introduced by Nudelman *et al.* (2004) and the final three by Xu *et al.* (2012b). The number of features in a group appears in brackets—numbers correspond to the original publication as some of the original groups differ in the later work. The number of variables is v and the number of clauses c. Statistics collected are usually the minimum, maximum, mean, coefficient of variation and entropy.

Origin of features in group	Example(s) of feature(s)
Basic size of the problem (11)	v, c, $R = c/v$, R^2, R^3, R^{-1}, $\lvert 4.26 - R \rvert$.
Variable-clause graph (10)	Statistics of the degrees of variable and clause nodes in the formula's CVIG.
Variable graph (4)	Statistics for the degrees of nodes in the formula's VIG.
Clause graph (10)	Statistics for the degrees of nodes and the weighted clustering coefficient in the formula's CIG.
Balance of items in a formula (13)	Fractions of clauses that are unary, binary or ternary. Statistics of the ratio of positive/negative variable occurrences.
Similarity to Horn (6)	Statistics for how many times a variable occurs in a Horn clause.
Features from a linear program (6)	Value of the objective function after a linear program related to the formula is solved. Fraction of variables that are 1 in the solution, and further statistics.
Features from the DPLL search (7)	Estimates of the size of the search space. Number of unit propagations measured at different depths. Mean depth leading to a contradiction when instantiating variables at random and performing propagation.
Features from local search (24)	Search for local minima with two local search algorithms. Each is run multiple times, and statistics are derived from the runs.
Features from clause learning (18)	Run a SAT solver for 2 seconds. Statistics for number and length of clauses learned.
Features from survey propagation (18)	Estimate $\Pr(v_i = T)$ and $\Pr(v_i$ is not constrained) for each v_i. Statistics are derived from these values.
Features from timing (12)	The time taken to compute the features in each group.

1. *Power-law* or *scale-free structure*. This is related to the existence of heavy-tailed distributions in some real-world date. Specifically, denoting by V the set of variables in a SAT instance define

$$f_v(n) = \frac{|\{v \in V | v \text{ occurs } n \text{ times}\}|}{|V|}.$$

   Under the assumption that $f_v(n) \simeq cn^{-\alpha}$ the single feature α can be estimated using the methods described by Clauset *et al.* (2009).

2. *Community structure*. This relates to the tendency of some graphs observed in practice to be structured as a number of *communities*. A single feature can be obtained for a SAT instance by estimating the *modularity* of the VIG (Ansótegui *et al.*, 2012) using an algorithm described by Blondel *et al.* (2008).

3. *Fractal* or *self-similar structure* as measured by estimating the fractal dimension of the VIG and CVIG graphs (Ansótegui *et al.*, 2014).

Ansótegui *et al.* (2017) compared the use of 115 of the standard features by the portfolio SAT solver SATzilla (described in Chapter 6) with a set of only five features consisting of the four structure-related features mentioned along with the clause/variable ratio R. It was found that, for the industry applications considered, similar performance can be obtained.

4.2 Graph Representations

The structure-related features described above suggest that it may be profitable to represent a CNF formula as a graph, as part of the feature engineering process. Several machine learning algorithms have now been proposed that aim to take graphs directly as input, and this provides a way of partially avoiding the feature engineering process. These methods are collected under the heading of *Graph Neural Networks (GNNs)*; a survey can be found in Wu *et al.* (2019), and Hamilton (2020) provides a readable introduction. Some of these methods have the advantage that they can be made invariant to symmetries that appear in the SAT problem; for example, we can re-order the clauses in a CNF formula without changing its SAT status, and similarly we can change the polarity of a single variable throughout a formula.

Gilmer *et al.* (2017) show that many of the proposed algorithms have significant elements in common, and propose the *Message-Passing Neural Network (MPNN)* as a single unified architecture for such tasks. The underlying structure of the MPNN is shown in Figure 4.1. Each node n_i has features $\mathbf{x}_i$ and hidden state $\mathbf{h}_i^t$. Edges $e_{i,j}$ have features $\mathbf{x}_{i,j}$ and hidden state $\mathbf{h}_{i,j}^t$. Updates are made over T steps, and the superscript t denotes the current step.

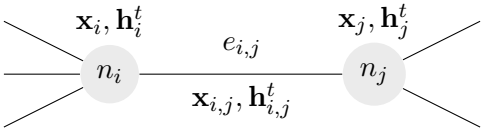

Figure 4.1: Message passing for graph neural networks, in the general MPNN case based on Gilmer *et al.* (2017). Nodes n_i and n_j are connected by an edge $e_{i,j}$, and have features $\mathbf{x}_i, \mathbf{x}_j$ and hidden state $\mathbf{h}_i^t, \mathbf{h}_j^t$ respectively. The edge has features $\mathbf{x}_{i,j}$ and hidden state $\mathbf{h}_{i,j}^t$. The superscript t denotes the time step at which messages and hidden states are calculated.

During each update *messages* $\mathbf{m}_i^t$ are passed between nodes. Let $N(i)$ denote the neighbours of the ith node in the graph. Updates over the T steps are

$$\mathbf{m}_i^{t+1} = \sum_{j \in N(i)} M_t(\mathbf{h}_i^t, \mathbf{h}_j^t, \mathbf{x}_{i,j})$$
$$\mathbf{h}_i^{t+1} = U_t(\mathbf{h}_i^t, \mathbf{m}_i^{t+1}).$$

Here, the M_t and U_t are *message functions* and *vertex update functions* respectively. Hidden states $\mathbf{h}_i^0$ can be initialized, for example, using the features $\mathbf{x}_i$, if necessary padded out to a higher dimension. After these updates a single output vector $\mathbf{y}$ is obtained as

$$\mathbf{y} = R(\mathbf{h}_1^T, \ldots, \mathbf{h}_N^T)$$

where R is the *readout function* and N is the number of nodes in the graph. The functions M_t, U_t and R are typically parameterized, and learned during the training process; thus the output $\mathbf{y}$ obtained directly from the input graph is the outcome of a process of learning from the data, rather than explicit feature engineering. We should note however that some degree of feature engineering might remain, in that node and edge features $\mathbf{x}_i$ and $\mathbf{x}_{i,j}$ specific to the problem at hand may still be required.

The various methods proposed for turning graphs into vectors $\mathbf{y}$ in this way differ according to the functions used, and to other details such as whether the edge hidden states are included in the computation. For example, functions can vary from simple fixed concatenations, through parameterized linear functions, to full NNs. While in general, functions M_t and U_t can be trained for each iteration, it is common to employ weight tying such that single functions M and U can be used for all iterations.

Clearly there is considerable scope for adapting the MPNN model to the SAT problem. As an example, we describe the architecture used by Selsam *et al.* (2019) in a system that will be described further in Chapter 5. Initially a formula f is represented as a graph with its literals and clauses as nodes, as shown in Figure 4.2. There are two kinds of edge: an edge of the first kind occurs between each clause and the literals it contains; an edge of the second kind occurs between complementary pairs of literals.

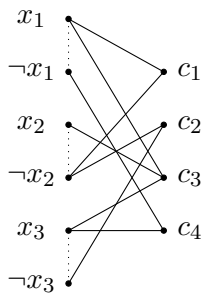

Figure 4.2: Representation of the CNF formula $(x_1 \vee \neg x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_3)$ as a graph in the manner of Selsam *et al.* (2019).

For a formula with v variables and c clauses, the ultimate aim is to derive matrices $\mathbf{L}^t \in \mathbb{R}^{2v \times d}$ and $\mathbf{C}^t \in \mathbb{R}^{c \times d}$, the rows of which contain vector-valued *embeddings* for the literals and clauses respectively, each in $\mathbb{R}^d$ where d is a parameter of the system. Once again t denotes the time step and the process is run for T steps. Updates are made using four learned functions: $\mathbf{L}$ and $\mathbf{C}$ are MLPs, and $\mathbf{L}'$ and $\mathbf{C}'$ are layer-norm LSTMs (Ba *et al.*, 2016). Hidden states $\mathbf{L}_h^t$ and $\mathbf{C}_h^t$ have the same dimensions as $\mathbf{L}^t$ and $\mathbf{C}^t$ respectively. The updates are

$$(\mathbf{C}^{t+1}, \mathbf{C}_h^{t+1}) = \mathbf{C}(\mathbf{C}_h^t, \mathbf{A}^T \mathbf{L}'(\mathbf{L}^t)) \tag{4.1}$$

$$(\mathbf{L}^{t+1}, \mathbf{L}_h^{t+1}) = \mathbf{L}(\mathbf{L}_h^t, f\mathbf{L}^t, \mathbf{A}\mathbf{C}'(\mathbf{C}^{t+1})) \tag{4.2}$$

where $\mathbf{A}$ is the adjacency matrix

$$A_{i,j} = \begin{cases} 1 & \text{if literal } i \text{ is in clause } j \\ 0 & \text{otherwise} \end{cases}$$

and $f(\mathbf{L}^t)$ swaps each row representing a literal with the row representing its negation.

4.3 Discussion

It has long been known that both the quality and the number of features used in an application of ML can have a significant effect on the performance attained. It is therefore common in ML to concentrate on the quality of extracted features first—in terms of their contribution to performance—and for any difficulty involved in *computing* such features to be secondary. SAT solving, as a target for ML, is perhaps unique in having such a standardized set of hand-engineered features, with a large body of research demonstrating their effectiveness.

SAT solving can however carry a further requirement, that can make feature engineering notably challenging. It should be clear that many of the features described above are likely to be time-consuming to compute; this was, for example, observed directly by Devlin and O'Sullivan (2008) while learning to identify satisfiability directly—a procedure described further in the next Chapter. They also observed that the time taken to compute some of the standard features can differ for satisfiable and unsatisfiable problems. Complex features (in the sense of their relative difficulty of computation) are unlikely to be a problem in *some* applications of ML to the SAT problem. A common example is that of portfolio solvers, where the time taken to compute features can be heavily outweighed by the time taken to run one or more standard SAT solvers. The standard features are often not problematic in this case; this is unsurprising as they were designed with portfolio methods in mind.

However, there are numerous applications where feature complexity is highly significant. In particular, we shall see in what follows that any attempt to introduce ML into the operation of a CDCL solver can force us to use only features that are very cheap indeed to compute. (And for the same reasons, any classifier will need to be able to compute its result very quickly). Otherwise, the ML components can easily slow the solver to the extent that it is entirely uncompetitive.

Attempts to mitigate the complexity of feature computation can take multiple forms. For example, *feature selection* algorithms are common in the ML literature; these attempt to find a subset of features that can be used alone without unduly reducing performance. Another approach, related to the GNN methods described above, involves learning features from experience without designing them by hand; an example is the work of Loreggia *et al.* (2016) that we describe in Section 6.5. There is some evidence (Cameron *et al.*, 2020) that such features can be faster to compute than engineered alternatives. We shall discuss these issues further as they arise in later Chapters.

5 Learning to Identify Satisfiability Directly

Some of the earliest attempts to apply ML to the SAT problem involved treating the problem purely as a classification task—given an instance of SAT in the form of a propositional formula, classify it into one of two classes: satisfiable or unsatisfiable. This work is described in Section 5.1. While limited by the resources available at the time, it provides us with a historical starting point—one that has a great deal in common with more recent developments.

Later, the straightforward correspondence between variable assignments and bit-strings led researchers to explore the use of GAs to solve SAT directly. This work is described in Section 5.2.

More recently, the former approach has been revisited. In particular, developments in

NNs and GNNs have been applied to SAT as a classification problem. Section 5.3 describes several such applications using different architectures and different representations for the SAT problems of interest.

Closely related to the problem of identifying satisfiability is the problem of identifying *sequents*. Given a pair X and Y of propositional formulas, a *sequent* $X \models Y$ denotes the property that any satisfying assignment for X is also a satisfying assignment for Y. Section 5.4 describes research using NNs to learn to solve this variant of the problem.

Finally, in Section 5.5 we describe work with the aim of embedding solvers into larger NN-based systems, in a way that allows them to learn from examples how to solve a problem.

5.1 Early Approaches to SAT as Classification

Johnson (1989) attempted to solve 3-SAT using an approach building on that applied by Hopfield and Tank (1985) to the travelling salesman problem (Garey and Johnson, 1979). Given an instance of 3-SAT a continuous, recurrent neural network was constructed in such a way that its dynamics minimize a specified error function; in turn, it can be shown that the global minimum of this function corresponds to a satisfying assignment, if one exists, and that this assignment can be extracted. While no proof was provided that local minima will be avoided, experimental evidence suggested this might not be problematic.

Spears (1996) took a similar approach, presenting a method for directly converting a CNF formula into a corresponding neural network. An algorithm was presented allowing the network to evolve over time, the activations of the nodes being updated probabilistically at each step. The degree of randomness at each update was controlled by a temperature parameter that is reduced over time—a form of *simulated annealing* (Laarhoven and Aarts, 1987). The dynamics were again designed such that the network tends to converge in a way that allows a satisfying assignment to be read from its nodes. However convergence to a satisfying assignment is again not guaranteed.

Devlin and O'Sullivan (2008) used forty-eight of the standard features along with multiple learning methods, including MLPs, naïve Bayes, decision trees, random forests, and nearest-neighbour classifiers, to learn directly whether or not an instance is satisfiable. They found that no single classifier dominates, but the best appears to depend on the nature of the problem set.

The reader should have two worries at this point. The first two approaches work by setting up a dynamic system that attempts to converge to a solution, but in neither case is convergence guaranteed. The third approach attempts to classify instances directly, but using ML methods that we might expect to make mistakes. In logical terms, they act as provers that might fail to be *consistent* or *complete*. Problems such as this are also present in the more recent work described below, and we might legitimately ask whether such

provers are in any way useful.

We leave this thought for further discussion in Section 5.6. For the time being, let it suffice to note that such provers are indeed useful. For example, in constructing portfolio solvers a hierarchical approach can be taken, because the time taken to solve a SAT problem can be dependent on whether the instance is satisfiable or unsatisfiable. Predicting this time can therefore be done most effectively using different predictors for the two types of instance, and a choice between these can be made by first predicting whether an instance is satisfiable.

5.2 GAs for Solving SAT Directly

The SAT problem lends itself very naturally to solution by GAs. Any potential solution is a bit-string with one bit per variable, the fitness of a solution is the number of satisfied clauses, and standard crossover and mutation operators can be applied. Consequently, considerable effort has been expended in approaching SAT-solving in this way.

The research published up to 2002 was the subject of its own review by Gottlieb *et al.* (2002), to which the reader is directed for details. We shall however mention some overall lessons. First, there are other ways of representing a potential solution to a SAT problem; for example, attempts have been made to represent potential solutions for problems with n variables as real vectors $\mathbf{x} \in [-1, 1]^n$. However, the evidence suggests that the bit-string representation is preferred. Furthermore:

- It is important to use *adaptive* fitness functions. Two specific examples are given. For a CNF formula $f = c_1 \wedge \cdots \wedge c_m$ let $c_i(A)$ take the value 1 if clause c_i is satisfied by assignment A and 0 otherwise. The first adaptive fitness function is

$$F(A) = \sum_{i=1}^{m} w_i c_i(A)$$

  and the second is

$$F(A) = \sum_{i=1}^{m} c_i(A) + \alpha r(A)$$

  where r is a *refining function* and α sets the degree to which r affects the fitness. In the former case the parameters w_i are adapted during the evolution process, and in the latter case α and any parameters associated with r are adapted.

- Small populations, and the use of mutation without crossover are preferred.

A notable work since this review appeared is that of Aksoy and Gunes (2005). Here, the underlying GA uses the bitstring representation and the number of satisfied clauses as fitness, but modifies both initialization of the population and crossover using properties of

the specific instance formula f. In addition it mixes the evolutionary process with runs of a local search solver, and uses a lack of progress by this solver as an indication of having found a local optimum, thus triggering the use of mutation.

5.3 SAT as Classification Using GNNs and NNs

In Chapter 4 we noted that recent work on GNNs provides a way to deal with formulas represented as graphs directly using ML methods. Bünz and Lamm (2017) attempted to learn to classify formulas as satisfiable or unsatisfiable using this approach to formula representation. They represented formulas in a direct graph-based format and trained a GNN to perform the classification. They also provided evidence—by applying a recurrent neural network to a textual representation—that trying to treat propositional formulas in a similar way to written text is less successful, giving weight to the GNN approach.

This work in essence represents an attempt to re-start the research described in Section 5.1, using a mixture of more sophisticated ML methods and more capable hardware. Several researchers have taken this approach further still.

5.3.1 GNNs Applied to CNF Formulas

In Chapter 4 we also gave a partial description of the NeuroSAT system presented by Selsam *et al.* (2019), which has further developed the idea of treating SAT directly as a classification problem. In that Chapter we introduced its use of a GNN incorporating a combination of MLPs and LSTMs to map each literal and each clause of a candidate CNF formula to a vector representation. The mapping was constructed to be invariant to some important symmetries.

The key equations defining this process are Equations (4.1) and (4.2) (page 45), and the matrix $\mathbf{L}^t \in \mathbb{R}^{2v \times d}$ obtained at step $t = T$ in this process contains a representation in $\mathbb{R}^d$ for each of the $2v$ literals. The final step needed was to introduce a further function $\mathtt{V}$, implemented by a trainable MLP, to produce a *vote* for each literal. Thus we obtain a vector $\mathbf{v} = \mathtt{V}(\mathbf{L}^T) \in \mathbb{R}^{2v}$ of votes. The output of the system is essentially the average of the votes in $\mathbf{v}$. The entire system—including all the MLPs and LSTMs mentioned, is trained to minimize a measure of loss between this average and a single bit denoting satisfiability versus unsatisfiability for a set of training problems.

This work diverges from the mainstream of SAT-solver research, in the kind of data used for training and testing. (There are good reasons for this, which we return to in our Discussion in Section 5.6.) The problems used are not derived from repositories such as SATLIB, or those associated with the International SAT Competitions. Rather, they are randomly generated. As the method of problem generation used has been taken up by other

researchers, particularly those interested in NNs and related methods, we describe it in detail.

The problem class SR(n) denotes by n the number of variables and generates CNFs in pairs, such that one instance is satisfiable, one unsatisfiable, and they differ by a single flipped literal in a single clause. To make such a pair, a CNF is grown a clause at a time. A clause is made as follows: a value n' is generated at random using a distribution with mean a little greater than four, n' variables are chosen from n at random without replacement, and each is negated with probability 1/2. Clauses are added until the CNF becomes unsatisfiable, and at this point a single literal in the last clause is flipped to give a satisfiable CNF.

A second method for generating problem pairs involves the idea of an *Unsatisfiable Core (UC)*. A UC C is a set of clauses, possibly part of a larger CNF, that are known to be unsatisfiable. The problem class SRC(n, C) is similar to SR(n), but each unsatisfiable CNF is known to have C as a UC. To generate pairs, a single literal in a clause in C is flipped to make C', which is satisfiable. Clauses are added to C' as described above to make the satisfiable instance, in which C' is replaced with C to obtain the unsatisfiable instance.

The results obtained using this system show some interesting characteristics. When trained on problems from SR(n) with n between ten and forty, an accuracy of 85% was achieved on a representative test set from SR(40). Of greater significance however is that, using the activations for each literal in the input formula, a satisfying assignment can often be extracted for satisfiable instances—this was possible 70% of the time for the test set used.

A further significant observation is that, while the main training and test problems are randomly generated, when applied to larger problems and to formulas from several graph-based problems, some success can be achieved simply by increasing the number of iterations the system is allowed to run for. However when applied to problems differing significantly from those used in training, accuracy is reduced.

A final outcome of interest was obtained by attempting to learn using SRC($40, C$) with a selection of three known UCs for C. In this case the system showed some ability to identify the variables involved in the contradictions.

5.3.2 GNNs Applied to Formulas as Circuits

Amizadeh *et al.* (2019) take an alternative approach to this problem, based on a circuit representation instead of the CNF representation. A formula is represented as a DAG: nodes correspond to `AND`, `OR` or `NOT` gates, and edges define connections between them. The SAT problem is then to find inputs to the circuit (now corresponding to variables) to produce a logical T at the output.

The proposed system has two components, the first attempting to find a satisfying assignment, and the second verifying that this assignment in fact works. The first of these

networks uses a NN that processes an input circuit based on its graph structure. Nodes n are converted to vectors $v(n)$, using a one-hot encoding to denote whether they are a variable or one of the logic gates. Using the ordering implied by the DAG, a state vector $\mathbf{s}_n$ is then propagated as

$$\mathbf{s}_n = S(v(n), \mathbf{s}_{\text{pred}})$$
$$\mathbf{s}_{\text{pred}} = T(\{\mathbf{s}_{n'} : n' \in p(n)\})$$

where $p(n)$ denotes the set of predecessors of node n, and S and T are learned.

This is sufficient to construct a mapping from variables to a state. How do we then propose an assignment to the variables? In order to do this the idea is extended in two ways. First, it is also applied to the DAG with its edges reversed, and multiple forward and reverse layers with this structure are employed. Second, the computation of these layers is itself repeated in a recurrent manner. The only constraint is that the last layer should be a reverse layer, so that its outputs—corresponding to variables—can be used to compute a potential assignment. A final, feedforward network is used to convert those outputs to soft assignments with values between 0 and 1.

When the first component has proposed a soft assignment to the variables, the second component is used to check whether this assignment does in fact satisfy the problem at hand. It does this by reproducing the input circuit using soft versions of the gates. For example, each `NOT` gate is modelled by the function

$$N(x) = 1 - x$$

and each `AND` gate by the function

$$A(x_1, \ldots, x_n) = \frac{\sum_i x_i \exp(-x_i/T)}{\sum_i \exp(-x_i/T)}$$

where T is a parameter used to adjust the characteristics of the function. An analogous function models each `OR` gate.

The output of this network is then used to train the overall system, using a loss function that is essentially a smooth version of the step function and thus rewards situations where the second network verifies that a satisfying assignment has been found. This training process is related to the use of policy gradient methods in reinforcement learning, and the parameter T can be used to trade-off exploration against exploitation.

This system was initially applied by training using satisfiable problems from the SR(n) class described above, with n between three and ten, that had been converted from CNFs to circuits. Using the finding of a satisfying assignment as a measure of success, it again showed a level of ability to generalize to larger problems, from SR(20) up to SR(80). A similar result was achieved generalizing to random graph k-colourability problems, for graphs with six to ten nodes and k from two to four. Performance was generally improved when compared against NeuroSAT.

5.3.3 Exchangeable Matrix Layers

Another recent approach to learning to identify satisfiability directly, that has much in common with Selsam *et al.* (2019) but employs contrasting methods, appears in Cameron *et al.* (2020). This work also treats the problem as a two-class classification problem, and addresses it using an NN. It shares the aim of explicitly modelling the symmetries inherent in this problem, however it achieves this using a different approach: instead of using a GNN to learn a new representation of the input, it uses a $v \times c \times 2$-dimensional tensor input, where the i, jth entry is a two-bit one-hot encoding denoting whether variable i or its negation appears in clause j. This input is passed to successive further layers initially based on *exchangeable matrix layers* (Hartford *et al.*, 2018), and followed by pooling and an MLP. It is trained much as in Selsam *et al.* (2019), by minimizing the cross-entropy loss between the output of the network and a single bit indicating satisfiability.

Training and testing concentrated on randomly generated CNFs with between 100 and 600 variables, and with a clause-to-variable ratio placing them in the difficult, phase transition region. (This makes problems with 600 variables reasonably challenging.) In terms of classification accuracy, when trained and tested on comparable problems there was little to differentiate this approach from NeuroSAT. However this approach turns out to have two advantages. First, the training process is more memory-efficient. Second, and perhaps more significantly, it shows a much greater ability to generalize from small to large problems. It was found that training on problems with one hundred variables produced classifiers that still performed well on problems with six hundred variables.

5.4 Learning to Recognize Sequents

The closely-related problem of identifying *sequents* in propositional logic was addressed by Evans *et al.* (2018). Given formulas X and Y, a sequent $X \models Y$ denotes that any satisfying assignment for X is also a satisfying assignment for Y; equivalently, $\neg(X \to Y)$ is unsatisfiable. In this work, several NN architectures were explored for solving this as a classification problem. *Encoding* architectures take the form

$$\Pr(X \models Y) = \sigma(f(g(X) \circ g(Y)))$$

and *relational* architectures take the form

$$\Pr(X \models Y) = \sigma(f'(X, Y))$$

where σ is a function with range $[0, 1]$, f is a feedforward NN, g is a NN embedding formulas into $\mathbb{R}^d$, f' is a combination of NNs, and $\circ$ denotes concatenation. While some of these architectures have moderate success in identifying sequents, considerably greater accuracy was achieved using an architecture based on replacing the sequent with a continuous

approximation. For a sequent $X \models Y$, we know that a satisfying assignment for X is a satisfying assignment for Y. Allowing an assignment a to be modelled as a vector $\mathbf{a} \in \mathbb{R}^d$, we can represent this statement by

$$\texttt{Imp}(\texttt{Sat}(X, \mathbf{a}), \texttt{Sat}(Y, \mathbf{a}))$$

by learning functions $\texttt{Sat}$ with range $\mathbb{R}^d$, and $\texttt{Imp} : \mathbb{R}^d \times \mathbb{R}^d \to [0, 1]$, again implemented by NNs. Taking a set $A = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$ of assignments chosen at random we then have

$$\Pr(X \models Y) = \prod_{\mathbf{a}_i \in A} \texttt{Imp}(\texttt{Sat}(X, \mathbf{a}_i), \texttt{Sat}(Y, \mathbf{a}_i)).$$

Perhaps the most notable result obtained with this architecture is that, for some of the data sets employed, in excess of 90% of sequents can be correctly classified using a set A that is extremely small in comparison with the total number of possible assignments.

Chvalovský (2019) have also addressed this problem, using a circuit-based representation for formulas. It is assumed that a formula is represented as a circuit consisting of variables along with connectives `NOT`, `AND`, `OR` and `IMPLIES`. The key idea is to employ a vector $\mathbf{p}$, which can either be specified or learned, representing some property of the formula to be determined, such as satisfiability; it can therefore in principle be applied to problems other than recognizing sequents.

Variables are represented as vectors in $\mathbb{R}^p$ for some p. Feedforward networks are trained to represent reversed applications of the available connectives. For example, a network for a two-input `OR` operation computes $f_{\texttt{OR}} : \mathbb{R}^p \to \mathbb{R}^p \times \mathbb{R}^p$, and should be interpreted as computing from the output to the inputs of an `OR` gate.

Given an input formula, the trained networks are assembled to reflect its structure, and $\mathbf{p}$ is applied as an input to the combined network to produced output vectors corresponding to the variables in the original problem. A second collection of networks is then used to process those outputs. Three distinct processes are needed for this step:

1. It is possible for several outputs from the first stage to correspond to the same variable. A recurrent neural network is used to combine these into a single vector.

2. Having obtained a single vector for each variable, these are combined by a second recurrent neural network to give a single vector.

3. The final stage is a feedforward network converting that single vector into a direct indication of whether or not the formula has the property represented by $\mathbf{p}$.

The overall system is trained by optimization using the sum of squared errors as a loss function.

5.5 Differentiable Solvers

In this, final section we describe a system that departs somewhat from the usual conception of a SAT-solver. In essence, the idea is to parameterize a solver in such a way that the problem it addresses can be specified by setting the parameters, and these parameters can be *learned* by seeing *examples* of the problem. The aim is to allow a SAT-solver to be treated as a learnable module, possibly as part of a larger ML system. This requires some further explanation.

Convolution nodes, LSTMs and so on are two examples of a plethora of substructures available to the NN designer, from which NN architectures can be built targeting specific applications. Given the flexibility of the SAT representation the following question presents itself: how might a SAT solver be incorporated into a larger, NN-based system?

The training of a supervised, feedforward NN using gradient descent typically requires two phases. First, in the forward propagation an input vector is applied to the network and the necessary computations are allowed to propagate to the output. Second, in the backward propagation the gradients of the network's loss function with respect to its parameters are computed, starting at the output and working back to the inputs. Both forward and backward propagations are straightforward for the kinds of structure commonly employed.

If we wish to include a SAT-solver in this kind of system, there are three inherent challenges. First, while a SAT-solver clearly has an input and an output, the intervening process is NP-complete, and is thus likely to present an unacceptable bottleneck. This suggests a need to design *approximate*, rather than exact solvers. Second, a SAT-solver is not differentiable with respect to its parameters, so computing gradients of the overall network's loss function with respect to them is not possible. Third—and this is perhaps the most interesting source of new possibilities—it is not immediately clear what the input to a solver should be.

The preceding statement needs elaboration. Consider a SAT-solver applied to deciding graph colourability. We start with a graph, use a standard encoding method to obtain a CNF, and the solver provides an answer. We might regard the graph as the input; alternatively, we could consider the input to be the CNF formula, which *also* encodes the problem to be solved. Might a solver be parameterized in a way allowing a solver for the colourability problem, or some other problem, to be learned from examples?

When dealing with the CNF representation, the SAT problem can be modified to give us the *MAXSAT problem*: rather than looking for a satisfying assignment we try to find an assignment satisfying as many clauses as possible. Let the n variables in the problem be denoted $v_i \in \{+1, -1\}$ and let c_{ij} denote the status of variable v_i in clause j, taking value 1 if the variable is positive, -1 if it is negative, and 0 if it is not present. If there are m

clauses then the MAXSAT problem is to find

$$\underset{v_1,\ldots,v_n}{\operatorname{argmax}} \sum_j \bigvee_i \mathbb{I}\left[c_{ij} v_i > 0\right] \tag{5.1}$$

where $\mathbb{I}[P]$ is the indicator function.

Wang *et al.* (2019) have used the MAXSAT problem as a means of producing a differentiable solver. The first step is to represent MAXSAT in a continuous rather than discrete form. Each variable v_i is mapped to a corresponding unit vector $\mathbf{v}_i \in \mathbb{R}^k$, with the interpretation that, with respect to some *truth direction* unit vector $\mathbf{v}_T$

$$\Pr(v_i = 1) = \frac{1}{\pi} \cos^{-1}(-\mathbf{v}_i^T \mathbf{v}_T).$$

Let $\mathbf{c}_i^T = \begin{bmatrix} c_{i1} & \cdots & c_{im} \end{bmatrix}$, and let $\mathbf{1}_{ij}$ be the i by j matrix with unit elements. Also, define the matrices

$$\mathbf{V} = \begin{bmatrix} \mathbf{v}_T & \mathbf{v}_1 & \cdots & \mathbf{v}_n \end{bmatrix}$$
$$\mathbf{C} = \begin{bmatrix} -\mathbf{1}_{m1} & \mathbf{c}_1 & \cdots & \mathbf{c}_n \end{bmatrix} \operatorname{diag}((4|\mathbf{c}_i|)^{-1/2}).$$

Let $\langle ., . \rangle$ denote the inner product

$$\langle \mathbf{M}, \mathbf{N} \rangle = \sum_i \sum_j m_{ij} n_{ij}.$$

Then an alternative to solving Equation (5.1) is to solve

$$\underset{\mathbf{V}}{\operatorname{argmin}} \langle \mathbf{V}^T \mathbf{V}, \mathbf{C}^T \mathbf{C} \rangle$$

under the constraints $||\mathbf{v}_T|| = 1$ and $||\mathbf{v}_i|| = 1$ for $i = 1, \ldots, n$. This optimization problem can be solved using a co-ordinate descent method.

Having transformed the MAXSAT solver into a continuous form, it can be incorporated into an NN by treating the c_{ij} values as parameters, variables with known values as inputs, and the remaining variables as outputs. The co-ordinate descent is then applied only to find the output variables, and this constitutes the forward propagation process.

Finally, if L denotes the NN's loss function, then gradients such as $\partial L/\partial \mathbf{c}_i$, $\partial L/\partial \mathbf{v}_i$ and so on can be computed. (The conversion of variables to their vector representations and the corresponding inverse transformations must also be accounted for in this process.) As these gradients are available, the backpropagation phase of NN training can also be implemented. As a result, the parameters c_{ij} can be adapted in response to examples, *and these parameters define the MAXSAT problem being solved.* Wang *et al.* (2019) give an example of how this can be used to learn to solve Sudoku puzzles purely by seeing examples. This ability is demonstrated first by using a direct encoding of the input. An extension is then demonstrated adding CNN layers to the network such that learning can take place using images.

5.6 Discussion

It is notable that the earliest work on applying ML to the SAT problem has a great deal in common with some of the most recent: both treat SAT in terms of learning to classify an instance as satisfiable or unsatisfiable, and both show an ability to extract satisfying assignments as a side-effect of this process.

At present, this approach to SAT has become dominated by research based on NNs. While great attention has been payed to GAs for solving essentially the same problem, and with considerable success, this line of development appears to have stalled. The conclusion that there is a preference for mutation without crossover moves the GA approach rather close to a form of random search; while more recent work such as Aksoy and Gunes (2005) is more sophisticated, it remains to be seen whether there is further room for improvement using evolutionary methods.

The work on differentiable solvers represents a distinctive move beyond SAT as classification, and is reflected in a wider interest in modelling reasoning tasks in a connectionist manner. There is clearly considerable opportunity for further development here.

5.6.1 Should We Use Inconsistent Solvers?

From the perspective of an ML researcher, the treatment of SAT using ML is a legitimate and undoubtedly interesting problem, as SAT is known to be a computationally hard task—surely any ability demonstrated by an ML system to solve it must be significant? Arguably however, the inherent behaviour demonstrated by ML methods—that of *generalization*, with occasional errors appearing in the output—makes the approach of less interest to those predominantly interested in theorem-proving: any prover should at least be consistent. This is a relevant issue even when accuracy generally better than 90%, and in some cases better than 99% as reported even in relatively early work (Devlin and O'Sullivan, 2008), is obtained. Many of the architectures described above, while achieving good classification accuracy, do fall short of attaining an accuracy of 100%; in other words, they are not consistent theorem-provers, even when used to classify quite small—by the standards of CDCL solvers—problems.

We noted earlier in this Chapter that readers most interested in the ATP aspects of this review might be uncomfortable with the idea that some of the provers described here are not consistent. (We assume that *incompleteness* is more palatable, as local-search solvers share this property and are a long-standing and accepted tool.) There is however a single very good reason that such solvers are useful: one of the key applications of learners that treat SAT as classification is in the *portfolio solvers* we discuss in the next Chapter. Here, the classifier is not asked to provide a final answer, given an instance; rather, it is used to choose a consistent solver that might be more capable for satisfiable or unsatisfiable

instances.

5.6.2 Specialization and Generalization

The use of a wide range of NN-related methods in this context provides a further set of reasons for accepting the study of imperfect classifiers. Such studies are seen by many authors as addressing a scientific—as opposed to purely ATP-centric—pursuit. There is clear merit in this position: as generalization is a key ability in ML, questions regarding the ability of a learner to solve problems not seen during training, when faced with a difficult problem such as SAT, are of inherent interest.

Questions are also asked here that are closely aligned with multiple areas of research to be described in later Chapters. In particular:

1. Can a classifier trained using instances specific to a particular problem class also perform well in solving problems from outside that class?

2. Can similar generalization be achieved from small instances used in training, to larger instances used in testing?

As we shall see, such issues gain considerable significance when NN-based methods are taken beyond SAT as classification.

5.6.3 Architecture, Data and Resources

Throughout this review we shall see two recurring characteristics of some of the modern ML methods applied in the SAT domain. First, there is a huge variation in the architectures used. (And very clear opportunities for others.) Second, these architectures are often large enough to represent very significant computational challenges; in the current context, for the training process.

The second of these issues has sometimes made it infeasible to train models using the kinds of large and challenging SAT problems that are the staple means of assessing CDCL solvers. It is to be hoped that ongoing improvements in NN-specific hardware will solve this issue.

Despite these computational difficulties, Cameron *et al.* (2020) make an intriguing suggestion. While training such systems is currently time-consuming, features are effectively learned—rather than hand-engineered—as part of that process. Once learning is complete, the computation of those features for new inputs can be much more efficient than the computation of the standard features. (We might ask however, to what extent this observation is limited to that work.)

5.6.4 The Question of Unsatisfiability

Much of the direct ML treatment of SAT has focused on satisfiable instances, and the process of extracting a satisfying assignment. There are indications that unsatisfiable instances might be more problematic for ML.

This is perhaps unsurprising. Satisfiable instances are easy to certify, and we only need to find one assignment that works; unsatisfiability on the other hand appears much harder to handle. While we don't necessarily have to check all 2^n assignments for n variables to demonstrate unsatisfiability, a modern CDCL solver will still need to construct a proof, and these proofs can be quite lengthy. (Many solvers now provide proofs in the *Deleted Resolution Asymmetric Tautology (DRAT)* format (Wetzler *et al.*, 2014).)

Chen and Yang (2019) argue that the apparent difficulty that GNN-based methods have in dealing with unsatisfiable problems—in the absence of small unsatisfiable cores—might represent an inherent limitation of this approach. They provide some discussion in support of this position, but stop short of providing a fully definitive proof. They do however also argue that GNN-based methods can represent the standard heuristic used by the WalkSAT local-search solver, that we shall discuss further in Chapter 8. This seems an interesting connection, as local search solvers are also not capable of proving unsatisfiability.

6 Learning for Portfolio SAT Solvers

It has long been observed in experimental SAT research, that different solvers can show widely differing performance when applied to a problem instance. This should not be surprising: on the one hand, SAT is an extremely rich problem, and instances can possess a variety of structure, as discussed for example in Chapter 4. Also, solvers may be designed with specific kinds of problem in mind.

If a user needs solutions to problems from a known class, then use of a single, dedicated solver may seem a good strategy; but even then, there is evidence that behaviour can be very different for satisfiable versus unsatisfiable instances.

The following question thus arises naturally: is it possible to select a solver, or perhaps a sequence of solvers, most suitable for attacking a specific problem instance? This is the domain of solver *portfolios*.

Solver portfolios can be constructed using the more general idea of *empirical hardness models (EHMs)*, which we describe in Section 6.1. These form the basis for SATzilla, which is undoubtedly the most widely successful SAT solver portfolio in practice. We describe its development in Section 6.2, and an alternative approach more rooted in statistics in Section 6.3.

While these methods show significant sophistication, parallel developments have demonstrated that effective portfolios can also be constructed using only very simple ML techniques.

These are described in Section 6.4. Finally, an attempt to apply NNs to the problem is described in Section 6.5.

6.1 Empirical Hardness Models

Many attempts to apply ML to the SAT and QSAT problems have focused on the construction of EHMs (Nudelman *et al.*, 2004). Essentially, the aim is to learn to predict how long some solver will take to solve a given instance, and the approach is therefore closely related to the wider field of algorithm selection as reviewed by Kotthoff (2016), and to the use of *runtime performance prediction* (Allan and Minton, 1996), which uses short runs of solvers to estimate their ultimate performance. Similar work has appeared for problems closely related to SAT, such as solving constraint satisfaction problems (CSPs); see for example O'Mahony *et al.* (2008) and Yun and Epstein (2012).[1]

The construction of an EHM for a solver S works as follows:

1. Collect a set $\{P_1, \ldots, P_m\}$ of problems for training, and convert each to an appropriate set of features $\mathbf{x}_i = \boldsymbol{\phi}(P_i)$.

2. Run solver S on each of the training problems P_i to obtain its running time y_i. This results in the training sequence

$$\mathbf{s} = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)).$$

3. Train a regression method on $\mathbf{s}$ to obtain a predictor h that can be used to predict the running time $h(\boldsymbol{\phi}(P))$ for a new problem P, when solved by solver S.

4. Repeat this process for each solver of interest. This results in a set of predictors, one for each solver.

It is common for this procedure to be applied to a collection of solvers known as a *portfolio*, allowing the resulting classifiers to be used to select a single solver to use for a new problem. The process for this is straightforward: given a new problem, compute its features and apply them to each EHM, then choose the solver predicted to require the smallest solution time.

The process just described is straightforward, but many variations have been explored. For example, Nudelman *et al.* (2004) show that it may be advantageous to treat the prediction of running time differently for satisfiable and unsatisfiable instances. In particular, different classifiers using different features may be appropriate. (We expand on both of these points later in this Chapter.)

[1] To be clear, these works do not explicitly construct regression models to predict run-time. Instead they employ *case-based reasoning* (Mitchell, 1997). The design space for problems of this kind is extremely large, and admits multiple approaches.

6.2 Portfolios: Learning to Select a SAT Solver

The SATzilla solver is a *portfolio solver*—it uses a number of different SAT-solver algorithms. It employs numerous ML methods to do this.

In the version of SATzilla described in Xu *et al.* (2007) and Xu *et al.* (2008) the central ML method involves the construction of EHMs as described in the previous Section. These are based on the standard features, after some have been discarded through the application of *feature selection* (Guyon *et al.*, 2006). In addition, products of features are computed to form further features, which are subjected to a second phase of feature selection. Using the resulting features, EHMs predicting the runtime of various individual SAT-solvers are learned using ridge regression.

The system incorporates several further subtleties:

- The portfolios constructed run *pre-solvers* before attempting to apply the EHMs to select an alternative solver. (In fact, before even computing features for a new problem: as the computation of the features and application of the EHMs can itself be time-consuming, and many instances will be solved in less time than it takes to perform these steps, this can be a sensible first step. The EHMs themselves are trained only on instances that can not be solved by the pre-solvers, and thus concentrate on harder instances.

- In training an EHM the question arises of how to approach an instance for which the solver of interest times out. When this is the case, it is impossible to correctly label the instance according to its runtime; such data is known as *censored* data. An iterative method due to Schmee and Hahn (1979) is used to incorporate censored data.

- The authors explore the prediction of more general scores as an alternative to runtime. This can introduce further difficulties. For example, some scores used in competitions may depend on the performance of competing solvers.

- The EHMs are *hierarchical models* (Xu *et al.*, 2007). Instead of attempting to predict the runtime of a solver directly from the features of an instance, they first use a classifier to predict whether the instance is satisfiable or unsatisfiable. Two EHMs are trained for each solver—one for satisfiable and one for unsatisfiable instances. The predictions of the two EHMs are combined to obtain an overall predicted runtime.

- In some versions of SATzilla the hierarchical modelling approach is taken a step further. The data used for training relates to three groups of problems, with problems in a common group considered similar. The hierarchical modelling is extended from two to six EHMs, one for each combination of group and satisfiability.

Later versions of SATzilla (Xu *et al.*, 2009; Xu *et al.*, 2012a; Xu *et al.*, 2012c) introduce further refinements:

- Further ML is introduced attempting to predict whether computation of the features will take too long. This step uses a very simple set of features, such as the numbers of clauses and variables in the instance. If it is predicted that computing the full set of features will take too long, then a backup solver is run without making any use of the EHMs.

- A somewhat different approach to EHMs is introduced. Instead of learning to predict the runtime for a solver S on an instance, classifiers are trained for each *pair* (S_i, S_j) of solvers to predict which is superior. To choose a solver for a new instance, all classifiers make a prediction for that instance and the solver with the most winning predictions is chosen.

  The learning method applied here is somewhat more sophisticated, employing random forests (Breiman, 2001) and based on a classifier proposed by Ting (2002) that takes account of training examples having different weights. If solvers S_i and S_j have performance t_i and t_j respectively for an instance, then the corresponding training example is given weight $|t_i - t_j|$.

In the most recent version, now re-named **Zilla* (Cameron *et al.*, 2017), some further refinements are made, particularly in terms of using a general scheduling algorithm due to Streeter and Golovin (2008).

6.3 Learning Portfolios using Latent Classes

SAT problems can be divided into classes of related instances. For example, one source of SAT problems is SATLIB (Hoos and Stützle, 2000), and this contains classes such as graph colouring, model checking, fault analysis in circuits, planning, inductive inference and so on. While it is known that small changes to an instance of SAT can have a significant effect on the difficulty of finding a solution, it is natural to ask whether problems from the same class might be related, to the extent that similar methods can be applied to all instances within the class.

Taking a more general view, we can consider a scenario where instances fall into classes in this way, but where these classes are not known at the outset. In general, variables such as this—that form part of the structure of a problem but are not directly observed—are known as *latent variables*. An approach to portfolio construction suggested by Silverthorn and Miikkulainen (2010) achieved performance comparable to SATzilla, but required minimal

domain-specific knowledge in the form of feature engineering.[2]

In the following we denote by $[n]$ the set $\{1, \ldots, n\}$. Assume we have P problems, each of which falls into one of G groups. Assume there are S solvers. Given a parameter $\boldsymbol{\theta} \in [0, \infty)^G$ we can introduce a distribution over the groups using a conjugate pair of Dirichlet and multinomial distributions

$$\mathbf{d} \sim \texttt{Dirichlet}(\boldsymbol{\theta})$$
$$g_p \sim \texttt{Multinomial}(\mathbf{d}) \text{ for } p \in [P]$$

where g_p denotes the group associated with problem p.

Assume the result of running one of the S available solvers can be one of O outcomes. Denote by $o(i, s, p)$ the outcome of the ith run of solver s on problem p, where we consider the possibility of multiple runs to account for any randomization in the solver. Let $N_{s,p}$ be the number of times s is run on p. Given parameters $\boldsymbol{\theta}_{s,g} \in [0, \infty)^O$ with $s \in [S]$ and $g \in [G]$, the outcomes can then be modelled using a second conjugate pair as

$$\mathbf{d}_{s,p} \sim \texttt{Dirichlet}(\boldsymbol{\theta}_{s,g_p}) \text{ with } s \in [S], p \in [P]$$
$$o(i, s, p) \sim \texttt{Multinomial}(\mathbf{d}_{s,p}) \text{ with } s \in [S], p \in [P] \text{ and } i \in [N_{s,p}].$$

With this model for the data, and given training data obtained by observing the outcomes of solvers on a collection of test problems, the *Expectation Maximization (EM)* algorithm (Bishop, 2006) can be used to infer the parameter vectors $\boldsymbol{\theta}_{s,g}$. The learned model can then be used to choose an action to take when faced with a new problem. Actions are regarded as pairs specifying a solver to use and a duration for which to run it, where durations are specified using values from a finite set of possibilities.

6.4 Simplified Approaches to Portfolio SAT Solvers

SATzilla and other methods have proved extremely effective in practice. However, Malitsky *et al.* (2011) note that prediction of run time for a SAT solver is a difficult problem—with SATzilla's own methods often showing significant inaccuracy—and one whose solution might not be necessary for the construction of an effective portfolio.

They explore an alternative approach based on the k-nearest neighbour algorithm. (A related approach using k-nearest neighbours to perform heuristic selection in the context of CDCL solvers has been employed by Nikolić *et al.* (2009), and is described in Section 7.2. In Section 9.1 we describe an application of k-nearest neighbours to portfolios for quantified satisfiability, due to Pulina and Tacchella (2007), and the method has also been applied to

[2] Silverthorn and Miikkulainen (2010) in fact propose two models, and we describe only the more successful of these here—the *Dirichlet compound multinomial* model. Further variations can be found in Silverthorn (2012).

portfolio construction in the context of constraint satisfaction problems (Amadini *et al.*, 2015).) Given a collection S of solvers, a set P of training problems, and a time-out T, they measure the time taken (bounded by T) by each solver in S to solve each problem in P. They also compute the standard features for the problems in P, and parameters needed to normalise each to the range $[0, 1]$.

Given a new problem $p \notin P$ to solve, they compute its standard features, normalize them, and find the k problems $p_1, \ldots, p_k$ in P whose normalized standard features are closest (according to Euclidean distance) to those of p. The *PAR10 score*[3] is then calculated for each solver on the problems $p_1, \ldots, p_k$, and the solver with the best score is chosen to solve p.

A good value for k was determined using cross-validation (Kohavi, 1995), and in experiments it was found that this method performs significantly better than SATzilla.

Kadioglu *et al.* (2011) present a two-stage SAT solver which runs a single solver selected by the k-nearest neighbour algorithm for 90% percent of the available time, and a schedule of solvers precomputed using an integer program for the remainder.

In the first stage, problems are represented using a subset of the standard features, and training data specifies the run-times for all available solvers on all available training problems. For a new problem, the approach selects the k closest problems in the training sequence. The value of k is again selected using cross-validation. A single solver is selected from these: the solver that would perform the best on the k selected problems during the time limit, according to the PAR10 score.

The work also explores two further refinements. First, the use of *weighted* k-nearest neighbours, where the neighbours *closer* to a new problem have greater influence. Second, the use of variable k, where the value used depends on the new problem.

This method was extended by Malitsky *et al.* (2012) for the case where more than one processor is available. Here, having selected the k instances, a parallel schedule is chosen that would again solve the largest number of these instances.

Nikolić *et al.* (2013) also take a simplified approach to the construction of SAT solver portfolios. Their method is very similar to Malitsky *et al.* (2011), but differs as follows:

1. In order to make feature computation fast it uses a smaller subset of the standard features; this is also justified on the basis that the learning task is likely to be simplified as it is no longer necessary to estimate run times.

2. Features are not scaled.

3. If more than one solver minimizes the PAR10 score, ties are broken by selecting the one with the best performance on P.

[3] For a time limit T, the PAR10 score is an average computed using the time taken for solved problems, and $10T$ for unsolved problems.

4. The Euclidean distance is replaced by

$$d(\mathbf{x}, \mathbf{x}') = \sum_i \frac{|x_i - x_i'|}{1 + \sqrt{x_i x_i'}}.$$

Experimental evaluation shows that, while this system differs only in limited respects from the earlier work, it nonetheless outperforms it.

6.5 NNs for Portfolio Solvers

More recently, Loreggia *et al.* (2016) have attempted to extend various successes achieved by NNs, mostly in the area of image processing, to algorithm portfolios. The fundamental idea is to transform an algorithm selection problem directly into an image classification problem. The method for achieving this relies of the fact that algorithm selection, regardless of whether applied to SAT solving, constraint satisfaction, or some other task, usually starts with a text file describing a problem instance.

In order to transform an arbitrary text file into a suitable image, the string of characters is replaced by the corresponding sequence of ASCII codes. Assuming this sequence has length l, it is re-arranged as a $\sqrt{l} \times \sqrt{l}$ image, with the ASCII codes interpreted as grey-scale. This image is then re-sized using a standard image processing algorithm to an $n \times n$ image, where n is a fixed parameter.

The process described essentially allows any text file at all to be reduced to an $n \times n$ image, and thus the common method of *convolutional neural networks* (Goodfellow *et al.*, 2016) can be applied to solve the problem of selecting an algorithm from a portfolio: if m algorithms are available then the classifier has m corresponding outputs, which can be used directly to learn algorithm choice, or as features for use by a further classifier.

The authors apply this idea to problems from a recent international SAT competition, using both interpretations of the network outputs: direct solver selection, and features for a further portfolio builder. As a benchmark, they use cost-sensitive hierarchical clustering (Malitsky *et al.*, 2013) with hand-engineered features. They find that the approach outperforms individual solvers, but does not match existing state-of-the-art methods. However it provides a further interesting example of the way in which features can be learned rather than hand-engineered.

6.6 Discussion

The success of ML-based portfolio methods is a significant indicator that ML can profitably be applied to SAT-solving.

This area of research was for some time dominated by SATzilla, which has achieved a significant degree of sophistication and correspondingly high performance. There is evidence however that general approaches to portfolio construction, such as *cost-sensitive hierarchical clustering* (Malitsky *et al.*, 2013) can equal or exceed its performance.

There is an observation regarding ML in portfolio SAT solvers that seems particularly noteworthy: it is possible to achieve a great deal using what are, in ML terms, quite standard and relatively lightweight methods. While decision forests and statistical models based on latent variables are certainly sophisticated, they can be very much more compact and undemanding—in computational terms—than the various deep learning methods available. Also, the standard features form a relatively small set in comparison to the features employed in many other applications of ML. The fact that, as we saw in Section 6.4, such good performance can be attained using something as simple as k-nearest neighbour algorithms is particularly impressive.

Given the success of lightweight ML methods, the observation that the application of CNNs in this context does not equal the state-of-the-art might be taken to suggest that this is an area in which deep learning is not required. (The lack of the need for hand-engineered features however remains a notable advantage.) On the other hand, one might argue that the representation of SAT problems as images is, despite its merits, an unnatural one, in addition to lacking invariance to numerous symmetries inherent to SAT problems, that many GNN-based approaches have been careful to address. It remains to be seen whether other applications of deep learning to SAT portfolios, perhaps based on GNNs, can improve matters.

This chapter has limited itself to discussing only SAT portfolios. Portfolio solvers for QSAT have also been studied, and we shall take up this discussion again in Chapter 9. For now, we note that a similar conclusion is reached. In fact, there is evidence that excellent portfolio QSAT solvers can be constructed using only three features.

7 Learning for CDCL Solvers

In Chapter 2 we described quite an extensive collection of methods that have been employed to bring CDCL solvers to a level of performance making them capable of attacking large and interesting problems. Essentially any of these methods can be regarded as a target for improvement by the application of ML. It is probably not surprising that, given the importance of the SAT problem, numerous attempts have been made by researchers to apply ML wherever there is an opportunity.

In this chapter we take each target area for ML in a CDCL solver in turn, and describe how ML has been applied to it, beginning with learning to select a preprocessor in Section 7.1.

The effectiveness of any SAT solver is highly dependent on its ability to choose which

variable to branch on next. We saw in Chapter 2 that, prior to the development of activity-related heuristics such as VSIDS, several variable choice heuristics were proposed. We describe ML for selecting a good heuristic of this type in Section 7.2. These heuristics were largely superseded by more sophisticated activity-related heuristics, but one area in which ML has had a particular impact has been in moving beyond activity measures based on simple counts of variable use. We describe these developments in Section 7.3.

We describe the use of ML for selecting a restart strategy in Section 7.4, and for deletion of learned clauses in Section 7.5.

Researchers interested in evolutionary computing have tended to approach ML for SAT solvers in a manner complementary to that of mainstream ML. We shall see further examples of this in later Chapters; for now, let it suffice to observe that this should not seem surprising, as the methods used by GA and GP algorithms to *represent* solutions to problems tend to be very different. Such representations have been used to learn CDCL heuristics *combining* known approaches, as we shall see in Section 7.6.

Such differences carry over to the methods discussed in the final Sections. All current CDCL solvers expose numerous user-specifiable parameters. This review does not address the use of general-purpose algorithm configuration methods to choose such parameters; however, research that is *specific* to CDCL solvers exists on setting them. We address such methods in Section 7.7. Finally, in Section 7.8 we describe evolutionary computing methods attempting to directly modify the source code of a CDCL solver.

It will quickly become clear to the reader that a common shared factor in this research has been the use of MiniSat as a target solver. To date it has been the subject of work setting its parameters, modifying its internal heuristics, automatically editing its source code, and beyond.

7.1 Learning to Select a Preprocessor

Chen *et al.* (2014) apply a simple supervised learning model to select a preprocessor to use in advance of running MiniSat. Three preprocessors are available, with a fourth option of using no preprocessor.

- The *SatELite* preprocessor (Eén and Biere, 2005) applies logically sound simplification methods to the initial CNF formula with the aim of making it easier to solve.

- The second preprocessor inverts all literals in the original CNF, and if a solution is found inverts its literals accordingly. The aim here is to exploit the fact that the solver favours choosing negative literals to branch on first, and thus one choice of polarity may lead to a faster solution than the alternative.

- The preprocessor sets the ten most common literals to be false, simplifies the CNF accordingly, and runs the solver on the resulting CNF, but with a timeout. If the solver times out or indicates unsatisfiability, then the ten literals are set to true and the process repeated. If this leads to unsatisfiability then the solver is run on the original CNF. The aim here is to exploit the fact that setting common literals to guessed values may simplify the formula sufficiently to make the solving process much faster.

A training set is obtained by running each possibility on a set of training problems. Each problem is labelled according to the possibility that leads to a solution most quickly. A decision tree classifier (Quinlan, 1993) is trained on this data using the original standard features. The trained model is then used to predict which approach to use on a new problem.

7.2 Learning to Select a Heuristic

Lagoudakis and Littman (2001), building on Lagoudakis and Littman (2000), apply RL to SAT-solvers based on the DPLL procedure. Their aim was to learn to choose which of seven heuristics to use when selecting a literal at each point where the algorithm branches. They found that, using this approach, it was possible to perform better than when using any individual heuristic. They address the #SAT problem, which, rather than simply asking for a satisfying assignment, attempts to count the total number of satisfying assignments. This can be achieved by a small modification to the DPLL algorithm: instead of choosing a variable and polarity at each decision, we try both polarities and count the number of satisfying assignments achieved from each.

An RL problem is constructed as follows. The state is some representation of the formula at the point when a new decision is required. In fact, the authors find that a formula can be represented using only the number of variables it contains, and that using further features is not helpful. An action corresponds to a choice of one of the seven available, standard heuristics. The reward for choosing a particular heuristic is the number of nodes generated in the search tree by propagation as a result. (Some of the heuristics need to generate nodes as part of their computation. For these, the extra nodes generated are also included.)

The learning algorithm is a form of *Q-learning* (Sutton and Barto, 2018) based on trying to learn a function $Q(s, a)$, denoting the cost of taking action a in state s and thereafter acting optimally. It relies on the use of the update equation

$$Q(s, a) = r + \min_a Q(s', a) + \min_a Q(s'', a). \tag{7.1}$$

In Equation (7.1), r is the cost of taking action a in state s, and a corresponds to selection of a heuristic h. If h selects a literal l, then s' is the state reached from s using l, and s'' is

the state reached from s using $\neg l$. In order to learn $Q(s,a)$ it is represented as

$$\log Q(s,a) = \mathbf{w}_a^T \boldsymbol{\phi}(s)$$

where $\mathbf{w}_a$ is a vector of parameters for action a and

$$\boldsymbol{\phi}^T(s) = \begin{bmatrix} s & s^2 & \cdots & s^7 \end{bmatrix},$$

so we are using a polynomial approximation to $\log Q(s,a)$. As this is linear in its parameters a straightforward linear regression method is used to learn the weight vectors. If $\hat{Q}(s,a)$ is the current estimate for $Q(s,a)$, then during training we choose an action in state s, and the weights are adjusted to move $\hat{Q}(s,a)$ closer to the value

$$y = r + \min_a \hat{Q}(s',a) + \min_a \hat{Q}(s'',a).$$

This is achieved using a set of problems similar to those we expect to need to solve.

Taking a very different approach, Nikolić *et al.* (2009) explore the use of a k-nearest neighbour classifier with thirty-three syntactic features, derived from the standard features, to predict which of a predefined set of problem classes a new problem instance belongs to; that is, the label in the classification problem is now drawn from the collection

$$\{\texttt{graph\_colouring}, \texttt{model\_checking}, \ldots\}.$$

Having established that this prediction can be made very effectively, they use it as a means of selecting the best heuristic for the DPLL solver ArgoSAT (Marić, 2009). Starting with a collection of heuristics for choosing the next literal to search on, and the point at which to restart, they select, for each problem class in a training set, the combination of heuristics performing best for that class in the sense that it solves the largest number of problems. When faced with a new problem, the classifier is used to find the most appropriate problem class. The best heuristic combination for that class, as learned in the training phase, is then applied.

7.3 Learning to Select Decision Variables

7.3.1 Learning to Initialise Variable Activities

MiniSat assigns to each variable v in a problem instance an *activity* $a(v)$, initialized to zero. Kibria and Li (2006) aim to learn to *initialise* the activities in a way that is more effective than the default of setting them to zero. To do this a function f is learned and used to initialise activities as

$$a(v) = \sum_{c \in C(v)} \sum_{l \in c(v)} f(c, v, l, \ldots)$$

where $C(v)$ denotes the clauses containing v, and $c(v)$ denotes the literals in c not containing v.

The function f is represented as a LISP expression (McCarthy, 1960) constructed from simple functions including `exp`, `sqrt`, `if` $a > 0$ `then` b `else` 0, the basic arithmetic functions and so on. The additional arguments of f are values such as the number of times v appears as a positive/negative literal in the instance, the number of times l appears as positive/negative in the instance, the polarity of v in c, the number of clauses in the instance, and so on. A standard GP is used to learn the form of f, using as a fitness function the time taken to solve the instances in a training set of SAT problems.

This approach was significantly extended in later research described in Section 7.7.

7.3.2 Learning to Select Variable Polarity

The work of Grozea and Popescu (2014) is unusual in that it deliberately applies machine learning to a SAT solver which does not include state-of-the-art heuristics. Starting with an entirely basic backtracking solver the authors attempt to learn, first, to predict the *polarity* to try first for the current variable in the search. (Clearly, for any satisfiable instance the ability to predict polarity correctly will result in a solution being found immediately with no backtracking required.) This is then taken further in an attempt also to predict which variable to assign next.

The authors base their work on forty-eight of the standard features; in fact, they include two copies of these features, re-computed setting the variable of interest to true or false respectively. They also add twelve further features considered informative for the problem; for example, the size of the smallest clause containing v or $\neg v$, where v is the variable of interest.

Training data for supervised learning using a random forest classifier is extracted by running a solver on random 3-SAT problems and extracting, from each successful run, a collection of training examples during the final backtracking. After training it is found that the system has considerable success in reducing branching.

7.3.3 Learning to Improve on VSIDS

The VSIDS and related heuristics described in Section 2.4.5 maintain a measure $a(v)$ of *activity* for each variable v. Every time a variable is used in the derivation of a learned clause its activity is increased—or "bumped" in the usual parlance—and whenever the search needs to select a variable to assign it selects the variable having the highest activity. The alternative algorithms vary in how they bump variables, how the activities are allowed to *decay* over time, and so on.

Attempts have been made to improve such heuristics, in particular by increasing the amount by which variables involved in learning small clauses or clauses with a small

LBD are bumped, and increasing the bump for variables involved in computing large backjumps (Carvalho and Marques-Silva, 2004; Chang *et al.*, 2017; Chang *et al.*, 2018). While improvements have been demonstrated through such modifications they do not in themselves use ML as part of their operation.

Recently, researchers have incorporated ML into CDCL solvers by using multi-armed bandits to allow variable activities to be learned as a proof attempt progresses. This represents an important variation on many of the methods seen so far: we are now learning activities on a *per-proof basis*—there is no requirement to learn using a set of similar proofs before a learned model can be applied.

The Conflict History-Based Heuristic

The first method of this kind was proposed by Liang *et al.* (2016a) and Liang *et al.* (2016b). The underlying motivation was that experience suggests picking variables that are likely to be involved in the generation of many learned clauses. The *conflict history-based (CHB)* branching heuristic addresses the problem as a multi-armed bandit learning procedure, in which one arm is assigned to each variable and playing an arm corresponds to selecting a variable. The activity $a(v)$ of each variable v is updated using the ERWA algorithm every time v is chosen to branch on, propagated during unit propagation, or asserted as a result of clause learning. That is, we update as

$$a(v) \leftarrow (1-\alpha)a(v) + \alpha r_v.$$

The parameter α is initialised to 0.4 and is reduced by 10^{-6} at each conflict until it reaches 0.06, where it remains. The reward r_v for playing variable v is defined in order to provide high rewards to variables that have recently been involved in clause learning, with the aim of maximising the rate at which clauses are learned. Specifically,

$$r_v = \frac{\beta}{\mathcal{C} - \mathcal{C}(v) + 1}$$

where $\mathcal{C}$ is the number of conflicts seen so far, $\mathcal{C}(v)$ is the number of conflicts seen at the last point where v was used in learning a clause, and β is 1 if a conflict exists and 0.9 otherwise.

The Learning Rate Branching Heuristic

Liang *et al.* (2016b) extended the approach underlying the CHB heuristic by more explicitly optimizing the rate at which the heuristic generates conflict clauses. The *learning rate branching (LRB)* heuristic employs the same multi-armed bandit setup—with one arm per variable and the use of the ERWA algorithm—and uses the same schedule for the parameter α. It differs in its use of a more sophisticated concept of the reward r_v, and outperforms the CHB heuristic as a result.

Let $\mathcal{I}$ denote the interval beginning when a variable v is assigned and ending when it is unassigned during backtracking. A variable is said to *participate in generating* a clause c if, either it is a member of c, or it was resolved in generating c. Let $p(v, \mathcal{I})$ denote the number of clauses v participates in generating during interval $\mathcal{I}$, and let $\mathcal{L}(\mathcal{I})$ denote the number of clauses learned during $\mathcal{I}$. The *learning rate* $\mathrm{LR}(v)$ of v for the interval $\mathcal{I}$ is defined as

$$\mathrm{LR}(v) = \frac{p(v, \mathcal{I})}{\mathcal{L}(\mathcal{I})}.$$

A second component of the reward is generated by also considering literals on the reason-side of the cut during clause learning. A variable is said to *reason* in learning clause c if it does not appear in c, but does appear in a reason clause for a variable in c. Let $q(v, \mathcal{I})$ be the number of clauses v reasons in generating during $\mathcal{I}$ and let $\mathrm{RR}(v)$ denote the *reason rate*

$$\mathrm{RR}(v) = \frac{q(v, \mathcal{I})}{\mathcal{L}(\mathcal{I})}.$$

Then the overall reward is

$$r_v = \mathrm{LR}(v) + \mathrm{RR}(v).$$

Note that the reward in this case can not be computed until a variable is *unassigned*. Each time a variable is unassigned, including those originally assigned due to unit propagation, a play for the corresponding variable is made using reward r_v.

Finally, based on the observation that VSIDS can exploit community structure by preferring variables forming a particular community, at each conflict the activities of unassigned variables are scaled by a factor of 0.95.

The Global Learning Rate Heuristic

The LRB heuristic assesses variables individually, in terms of how they contribute to the generation of conflicts. Liang *et al.* (2017) take a more global approach, based on the *global learning rate (GLR)*. This is defined as the number of conflicts per decision generated by a solver. Empirical evidence demonstrates that high GLR correlates with shorter solver runs.

In initial experiments, decision variables were chosen by explicitly favouring those that would lead to a conflict if chosen. This is not a feasible heuristic to use in practice as it is very expensive to compute; however, it was found to be more effective than VSIDS if the time taken to compute it was discounted. Denote by $\mathcal{P}$ the set of partial assignments, and by $f : \mathcal{P} \to \{0, 1\}$ the function mapping a partial assignment to 1 if it will lead to a conflict when propagated and 0 otherwise. If available, this function would allow the full heuristic to be computed.

In an attempt to efficiently approximate the heuristic, the authors applied L_2-regularized linear logistic regression to learn an approximation to the function f. For a problem with

V variables, they learn an approximation $\hat{f} : \mathbb{R}^V \to [0, 1]$ of the probability that a partial assignment leads to a conflict. The feature vector $\mathbf{x} \in \mathbb{R}^V$ is simply

$$x_i = \begin{cases} 1 & \text{if } v_i \text{ is currently assigned} \\ 0 & \text{otherwise.} \end{cases}$$

The training data is extracted during the clause learning process, with a pair of examples being generated at each conflict: a positive example is generated using the negation of the learned clause along with the conflict-side literals; a negative example is constructed using the current partial assignment, having removed the literals at the current decision level and those in the positive example.

Two modifications can be made to this process. First, as the negative example tends to include many more literals than the positive example, these can be limited to one literal per decision level. Second, the reason-side literals adjacent to the learned clause can be added to the positive example.

A single step of gradient descent is then performed using these examples. This application of online learning is appropriate as it is efficient, and as the heuristic should be expected to vary over time as a side-effect of the clause learning process changing the state of the solver.

After training, the heuristic—known as the *Stochastic Gradient Descent Branching (SGDB) heuristic*—picks the variable predicted by $\hat{f}$ to have the highest probability of leading to a conflict, and achieves a level of performance comparable to VSIDS.

7.3.4 Perceptron-Based Methods for Learning to Select Variables

Heuristic Search (Pearl, 1984; Russell and Norvig, 2020) is a staple area of interest within artificial intelligence. In heuristic search we are searching on a graph of states, trying to move from start state s' to a goal state s'' in the best possible way. During a search, we wish to choose a state to explore next. The A^* method, for example, assesses a state s using the sum of the (known) cost $p(s', s)$ of moving from s' to s, and a *heuristic* $h(s, s'')$ that *estimates* the cost of moving from s to s''. A good heuristic can vastly improve the effectiveness of such searches. Fink (2007) has explored the use of ML to learn a good heuristic: each time a problem is solved, a single training example is generated and used to improve the current estimate for h.

Heuristic search clearly has a great deal in common with the search conducted by a DPLL/CDCL solver. In the latter, states represent the state of the solver after a number of decisions have been made, moving to a new state corresponds to selecting and assigning a variable, s' corresponds to no variables having been assigned, and s'' (for a satisfiable instance) to all variables having been assigned.

Flint and Blaschko (2012) have developed this correspondence further to learn to improve variable selection. At any point in the proof search tree, we will have reached a state s.

This state is described by the trail,[1] and by the state of the formula of interest given that decisions and propagations will have potentially simplified it. The key idea is to learn a perceptron, defined by a vector $\mathbf{w}$ of weights as described in Chapter 3.[2] Given such a weight vector, we need to use it to choose a literal (that is, the variable and its polarity) to assert, such that it is likely to lead to a satisfying assignment. The literal is chosen as

$$l(s) = \operatorname*{argmax}_l \mathbf{w}^T \boldsymbol{\phi}_l(s).$$

This is an unusual format, but one that is derived from attempts to apply ML when outputs are highly structured, rather than taken from simple sets such as $\{0, 1\}$; see for example Daumé and Marcu (2005).

A feature vector $\boldsymbol{\phi}_l(s)$ is computed for each possible literal assignment. Specifically, the work uses 25 simple features mostly related to standard DPLL search heuristics. For example, three of the features correspond to the number of times l appears in clauses of length two, three and four. A further feature is the current activity assigned to l's variable by MiniSat.

It is assumed that a solver will see a sequence of related problems. After a problem is solved using the current setting for $\mathbf{w}$, nodes in the resulting search tree can be labelled, provided that a satisfying assignment was found. If it was, then on the path in the search tree leading to that assignment, pairs (l, s) are labelled $+1$; all other pairs are labelled -1. The algorithm then attempts to find two sets of pairs, S^+ and S^-, such that pairs in S^+ are labelled $+1$, pairs in S^- are labelled -1, but pairs in S^+ are scored lower than pairs in S^-. It does this by finding a threshold θ and the two sets such that

$$\forall (l, s) \in S^+, (l', s') \in S^-. \mathbf{w}^T \boldsymbol{\phi}_l(s) < \theta < \mathbf{w}^T \boldsymbol{\phi}'_l(s')$$

and $|S^+||S^-|$ is maximized. Having done so, $\mathbf{w}$ is updated as

$$\mathbf{w} \leftarrow \mathbf{w} - \epsilon \boldsymbol{\nabla}$$

where ϵ is a learning rate parameter and

$$\boldsymbol{\nabla} = \frac{1}{|S^-|} \sum_{(l,s) \in S^-} \boldsymbol{\phi}_l(s) - \frac{1}{|S^+|} \sum_{(l,s) \in S^+} \boldsymbol{\phi}_l(s).$$

When used to modify MiniSat, it was found that this method delivered an order of magnitude increase in speed when tested using a set of hardware verification problems.

[1] The term *trail* is commonly used to refer to the list of assignments currently made, either as decisions or through propagation.

[2] In the original paper the authors use a more general Hilbert space-based formulation. I have simplified this in order to maintain a consistent notation and to keep the exposition straightforward. The reader should consult the original work for a fully general presentation.

7.3.5 GNNs for Learning to Select Variables

In Chapter 5 we described the NeuroSAT system of Selsam *et al.* (2019). While this does not offer a solver that guarantees to provide a correct answer, it has inspired a considerable body of further research in the context of consistent solvers.

Learning using Unsatisfiable Cores and Glue Variables

The first such extension is the *NeuroCore* solver described by Selsam and Bjørner (2019). This solver uses a simplified version of the original NN architecture, combined with standard CDCL solvers employing the EVSIDS variable selection heuristic, in a hybrid approach. It makes extensive use of the idea of *unsatisfiable cores (UCs)* for unsatisfiable problem instances, as described in Chapter 5, Section 5.3.1.

The work is based on the intuition that if a variable is likely to be part of a UC, then it makes sense to branch on it, as it is likely to lead quickly to a conflict. The authors therefore prepare a training set of unsatisfiable problems, and, by analysing proofs of unsatisfiability extracted from them, identify the variables in UCs. The problems forming the basis for this generation process include problems from the international SAT competitions up to 2017. A collection of NNs—in this case constructed from MLPs and without including LSTMs—is trained to produce outputs predicting how likely any variable is to be in a UC.

Denote the resulting prediction $\texttt{core}(v_i)$ for each variable v_i. In order to incorporate the prediction into an existing CDCL solver, let $E(v_i)$ denote the activity for v_i according to the EVSIDS heuristic. Periodically, the solver is interrupted and the current activities are replaced. In order to do this, a subset of the current problem and its learned clauses is provided to the predictor, and the activities are reset as

$$E(v_i) = v\kappa \times \texttt{softmax}(\texttt{core}(v_i)/\tau)$$

where v is the number of variables and τ and κ are parameters.

Results using test problems from the 2018 international SAT competition indicate an improvement over the unmodified CDCL solver. With a time limit of $5,000$ seconds, improvements of between 6% and 11% in the numbers of problems solved were obtained, depending on the underlying solver and the schedule used to reset the activations. The improvement comes mostly from an increased ability to solve satisfiable instances, despite the training employing only unsatisfiable ones. Results for a specific class of problems, rather than the full, and quite diverse set of problems were better still.

Two extensions to the approach of NeuroCore have appeared. The first, named *NeuroGlue*, is due to Han (2020a). This proposes that an alternative to assessing variables by their presence in UCs is to identify *glue variables*, which are variables likely to appear in clauses having LBD at most two. An architecture similar to one used by Lederman *et al.* (2019)—and which is described further in Chapter 9, Section 9.2.4—is trained to predict a corresponding

distribution over variables. This prediction is again used to periodically reset EVSIDS scores.

Han (2020a) also experiments with an RL-based method, constructed as follows:

1. For an instance f and trail t, let $f(t)$ be the formula obtained by making the assignments in t and carrying out any resulting propagations. Such formulas form the states for the RL.

2. An action consists of choosing a variable.

3. State transitions on choosing a variable v are implemented by choosing a polarity for v at random, adding the resulting assignment to the current trail to make a new trail t', and moving to the new state $f(t')$.

4. If the new state proves satisfiability the reward is 0; for unsatisfiability it is $1/G^2$, where G is the LBD of the conflict clause. Otherwise the reward is $-1/V$ where V is the number of variables in f.

The second extension to NeuroCore was proposed in Han (2020b). This work applies to *cube and conquer* solvers (Heule *et al.*, 2011; Heule *et al.*, 2016; Heule *et al.*, 2017), which work as follows. Initially, an instance f is addressed essentially using tree search, with a carefully hand-crafted heuristic for variable selection. This process aims to find a set $T = \{t_1, \ldots, t_n\}$ of trails. Considering the trails t_i—generally referred to as *cubes*—the aim is to find a set T for which the problems $f(t_i)$ are likely to be solvable efficiently by standard CDCL solvers, which can then if necessary be run in parallel.

Han (2020b) takes the view that variables appearing often in proofs of unsatisfiability might be good candidates for building cubes. A system based on NeuroCore is trained using variable counts extracted from *Deletion Resolution Asymmetric Tautology (DRAT)* proofs (Wetzler *et al.*, 2014). Given a new instance, the model is used to find the m top-ranked variables, and these are used to produce 2^m cubes.

Learning to Predict Satisfying Assignments

In Jaszczur *et al.* (2019) a graph-based representation for CNF formulas was used to learn a literal-choice heuristic directly. In addition to learning to predict whether or not an instance is satisfiable, the approach involves learning to predict whether or not each literal will appear in a satisfying assignment. This is done by augmenting the training data with further labels for each literal l in an instance formula ϕ, denoting whether or not $\phi \wedge l$ is satisfiable.

The same data generation process was used as by Selsam *et al.* (2019), with models trained using examples from SR(n) with n taking values 30, 50, 70 and 100. Using the trained networks for literal selection in DPLL solvers it was found that, using only satisfiable test

instances, performance over $1,000$ steps was similar to that of the one-sided Jeroslow-Wang (OS-JW) heuristic (Jeroslow and Wang, 1990), with OS-JW better on problems from SR(50) and SR(70), and the learned system better for SR(90) and SR(110).

A hybrid approach—testing the SR(50) trained model with satisfiable instances but this time with no step limit—where the learned heuristic was used until satisfiability is predicted with probability less than 0.3 and OS-JW is used thereafter, was also shown usually to use fewer steps in solving a problem than OS-JW alone, when used with DPLL or CDCL.

While these results are interesting, it is worth noting that:

1. The OS-JW heuristic is by no means a state-of-the art heuristic, having been eclipsed by VSIDS and others.

2. The problems addressed are much smaller than those that can be solved by existing CDCL solvers.

Finally, it is worth noting that the work concentrates on using the number of steps as a performance measure. It is common in attempts to apply deep learning methods to theorem-proving problems, even when bolstered by special-purpose hardware, to observe that, while the deep learning approach can require fewer steps, this must be offset by the fact that it may generate predictions relatively slowly, leading to an overall slow-down. We see further examples of this problem elsewhere.

Reinforcement Learning for Variable Choice

Kurin *et al.* (2019) use GNNs in an RL framework to learn a variable choice heuristic for a MiniSat-based solver. Their aim is to learn heuristics specialized to particular problem classes. Their *GQSAT* system places variable choice in an RL framework as follows:

1. A state s contains the currently unassigned variables and the set of unsatisfied clauses.

2. An action a is the selection of a variable and its polarity.

3. A state transition is obtained by asserting a variable and polarity, and allowing propagation and so on to occur.

4. Reward is $-r$ for some constant r when no solution is attained, and 0 otherwise.

The method of *deep Q-networks* (Mnih *et al.*, 2015) is used to learn an approximation $Q_{\boldsymbol{\theta}}(s, a)$ to an optimal function, the value of which is the expected reward if action a is taken in state s, and behaviour is optimal thereafter. Function $Q_{\boldsymbol{\theta}}(s, a)$ is based on a GNN, and $\boldsymbol{\theta}$ is the set of learned parameters. Once learned, it can be used to implement a policy $p(s)$ as

$$p(s) = \operatorname*{argmax}_a Q_{\boldsymbol{\theta}}(s, a).$$

The system is trained on a collection of problems from the SATLIB library. As the learned heuristic is computationally demanding compared with EVSIDS, it is used only at the beginning of a solution attempt, allowing EVSIDS to take over thereafter. This is justified using the observation that EVSIDS needs time for its initial state, with all variable activities set to 0, to become effective by identifying useful variables.

Several conclusions are drawn. The main conclusion is that the learned heuristic is more efficient than EVSIDS, in the sense that it generally leads to fewer decisions being needed to solve an instance. The system also demonstrates some ability to generalize from SAT to UNSAT problems, and to problems with more variables than those used in training. Finally, the system can, to a lesser degree, generalize to problem classes other than those used for training.

7.4 Learning to Select a Restart Strategy

7.4.1 Predicting the Number of Conflicts

Haim and Walsh (2008) note that predicting the run-time of a solver can be complicated by the presence of clause learning or restart strategies. This is because, in the first case, learning a clause changes the formula being processed, and in the second case, use of restarting generates a new search tree.

They address this by using features computed during the early part of the search. They initially compute seventeen features. These are specifically designed to be computed quickly, and are based on information generated during an *observation window* starting a short time after processing begins and persisting for a limited time. Using these features they use linear ridge regression to predict the logarithm of the number of conflicts. They attempt to allow solvers with a restart strategy to use information learned in earlier restarts to improve the prediction for later restarts. This is achieved by adding the predictions for the earlier restarts to the feature vector for the later one. Finally, they apply their method in a simple two-solver portfolio where the solvers differ in terms of the restart strategy used, in order to select the best alternative.

7.4.2 Predicting Which Strategy To Use

Haim and Walsh (2009) take the work on selecting a restart strategy a step further. They apply ML directly to the problem of selecting the best restart strategy from a choice of nine. To do this, they use sixty features based partly on the standard features and partly on those from Haim and Walsh (2008). Some features are computed at the outset and some during an observation window. They train a linear logistic classifier to predict satisfiability. In addition, for each restart strategy, they train a pair of linear ridge regression models to predict CPU time for satisfiable and unsatisfiable instances respectively. If the first classifier

predicts that an instance is satisfiable with probability p, and for a given restart strategy the predictions for CPU time are t_{SAT} and $t_{\neg\text{SAT}}$, then the prediction for that restart strategy is

$$\text{predicted time} = pt_{\text{SAT}} + (1-p)t_{\neg\text{SAT}}$$

and the strategy with the lowest prediction is selected.

7.4.3 Restarting Using Learned Clause Quality

More recently, Liang *et al.* (2018) have developed a learned restart method competitive with the state-of-the-art, based on the observation that restarts tend to improve the quality of learned clauses, where quality is measured in terms of the LBD—lower LBD clauses being considered of higher quality. (Clause LBD was defined in Chapter 2, Section 2.4.6.) On the other hand, restarting too frequently is to be avoided as it comes at the cost of having to reconstruct the search tree. They make two specific observations: first, that the tail of the distribution of learned clause LBDs is well-approximated by a normal distribution; second, that if LBDs of learned clauses are observed in the order they are generated then they are correlated, and thus it should be possible to predict the LBD of the next clause to be learned.

Taken together these observations make it possible to predict when to restart using the following approach:

1. By keeping track of the estimated mean μ and variance σ^2 of the LBDs of clauses learned so far, the z-score for the normal distribution can be used to detect when an LBD falls in a particular percentile. The authors use the 99.9th percentile, meaning that the threshold is
$$\text{LBD} > \mu + 3.08\sigma.$$

2. Each time a clause is learned, a training example is generated. As the aim is to predict LBD, the LBD of this clause is used as the target. The features are the LBDs L_1, L_2, L_3 of the three immediately preceding learned clauses. This example is used to make a single online update to the parameters of the linear function
$$L = w_0 + w_1L_1 + w_2L_2 + w_3L_3 + w_4L_1L_2 + w_5L_1L_3 + w_6L_2L_3$$
used to predict the next LBD.

3. Whenever unit propagation completes without detecting a conflict, the current set of weights is used to predict the LBD of the next clause that will be learned, and if this exceeds the threshold a restart is triggered.

This method is known as the *Machine Learning-based Restart (MLB)* heuristic. As the LBD of the next clause is predicted without actually generating it, a restart is therefore

attempting to better use the time that would have been needed to do this by rebuilding the search tree instead.

7.4.4 Restarting using Multi-Armed Bandits

Nejati *et al.* (2017) describe the *MapleCrypt* solver. While aimed specifically at the problem of inverting cryptographic hash functions, this solver uses a multi-armed bandit approach to learning to select a restart strategy. This approach—known as the *Multi-Armed Bandit (MAB)* restart strategy—is generally applicable.

A 4-armed bandit is used with the arms corresponding to the uniform, linear, Luby and geometric restart strategies. When a strategy is chosen the solver is allowed to run until a restart is indicated by that strategy, keeping track of the average LBD of the clauses learned during the run. The reciprocal of this average is then used as the reward for training using the discounted UCB algorithm.

In Section 2.4.7 we briefly reviewed some results on restarting due to Luby *et al.* (1993). In summary, if the run-time distribution is known then a uniform restart strategy is optimal; if not, then a single, fixed strategy exists with a guaranteed performance bound. Gagliolo and Schmidhuber (2007) propose a method employing MABs, involving several elements:

1. It attempts to learn the run-time distribution while solving a series of problems.

2. It uses a 2-armed bandit trained using the algorithm of Auer *et al.* (1995) to mix the use of an (approximate) optimal restart based on the estimated distribution, and the fixed Luby sequence.

3. While solving a problem the bandit is used to probabilistically select the next time limit to run for—either the next value in the Luby sequence, or the current estimate of the optimal time limit.

4. If the algorithm is unsuccessful within this time limit then the bandit learning algorithm is updated with a reward of 0, and the process repeats.

5. However, if the problem is solved then two things occur. First, the bandit algorithm is updated with a reward related to the total run-time. Second, the successful run-time, in addition to the times for unsuccessful runs—which act as censored samples—are used with an algorithm of Kaplan and Meier (1958), to maintain an estimate of the cumulative distribution function corresponding to the run-time distribution; this is represented in a form making the integral in Equation (2.2) easy to evaluate, and so the estimated optimal time limit for the fixed strategy can be updated as a sequence of problems is solved.

7.5 Learning to Delete Learned Clauses

When a CDCL solver learns a clause, the clause may then be used in producing future propagations, conflicts, learned clauses, and so on. It has been observed that such uses often happen shortly after the clause is learned, but much less frequently later in the search. As keeping all learned clauses entails costs both in memory and time, solvers therefore periodically delete clauses that are deemed in some sense to no longer be useful.

Soos *et al.* (2019) attempt to use ML to decide when to delete clauses. Their approach is a standard application of supervised learning, but incorporates a number of insights related specifically to this problem:

Feature engineering The standard features are used, but they are recomputed every $100,000$ conflicts because they will change during a solution attempt. Features more specific to individual clauses, such as the number of literals and the LBD are also used; in addition, features related to the state of the solver at the current and previous restart are used, such as averages of the LBDs and trail depth. Finally, features for a clause are included in an attempt to reflect its level of performance.

Labelling In order to label examples derived from learned clauses, it is necessary to assess how useful they were in finding a solution. In order to achieve this, training is based only on unsatisfiable instances. This is because learned clauses can be used in this case to construct a proof of the unsatisfiability of the original instance, and the degree to which a clause is involved in such a proof can be used as a measure of its importance. Proofs are written using the standard DRAT format described by Wetzler *et al.* (2014). Given an interval t, a clause is labelled to be kept for that period if the number of times it is used in the proof of unsatisfiability exceeds the average number of uses for all clauses over interval t.

One effect of this approach to labelling is that training data can only be obtained from unsatisfiable instances. This appears to have little negative effect on performance.

Classifier choice The work uses decision trees and random forests to perform classification, first because decision trees provide some interpretability, but also because in both cases the learned rules can be converted to C++ and thereby incorporated directly into the solver. In addition, random forests make assessment of the importance of individual features straightforward, allowing the size of the collection of features to be reduced to twenty-two.

Classifier training As deletion of a clause is in some sense more final than keeping it, classifiers are deliberately biased in favour of keeping clauses.

The work includes a notable observation on the difficulties involved in incorporating machine learning into state-of-the-art SAT-solvers: the use of twenty-two features increases the amount of information stored per clause by sixty-eight bytes, and at present the system increases cache misses enough to affect performance.

It is worth noting that part of the reason for this method's success is that ML has been applied outside the central loop of variable selection and propagation. Vaezipoor *et al.* (2020) have also used the approach of applying ML outside of this loop to allow relatively time-consuming processes to be used, applying RL to the problem of clause deletion. They place the problem into an RL format as follows:

1. The state of the solver includes the ratio of the number of learned to the number of original clauses, the histogram of LBDs for recently learned clause, and their average, and a moving average of recent decision levels and trail size.

2. An action consists of specifying a limit on clause LBD at the point clause deletion occurs: any clause with an LBD above the specified level is deleted.

3. To assign reward, the number n of times clauses are accessed during propagation is maintained. A limit l is placed on n, and should the limit be exceeded a run is deemed unsuccessful. Unsuccessful runs receive a reward of 0, and successful runs receive $200 - 10^{-7}n$ for the limit $l = 10^9$ employed.

7.6 GAs for Learning CDCL Heuristics

The term *hyperheuristic* is commonly used within the GA/GP literature when referring to a method that either chooses the best of an existing set of heuristics (a *selective* hyperheuristic), or constructs a new heuristic (a *generative* hyperheuristic). In Bertels (2016), Bertels and Tauritz (2016), and Illetskova *et al.* (2017), the *ADSSEC* system is described. This employs GPs to construct heuristics for variable selection and learned clause ranking, by combining elements of existing heuristics used by CDCL solvers. The central aim is to allow an existing CDCL solver to be specialized for solving a particular class of problems.

The way in which a GP can represent an individual corresponding to a heuristic, in such a way as to incorporate elements of multiple existing heuristics, is perhaps best illustrated using the example of learned clause ranking. Here, MiniSat for example updates the score for a clause additively each time the clause is used in analysing a conflict. On the other hand, Glucose scores clauses using their LBD. ADSSEC constructs heuristics using the usual tree-based representation for a function based on the following elements:

- Current score assigned to the clause.
- MiniSat increment for additive updates.

- LBD and its inverse.
- Number of conflicts so far.
- Number of literals in the clause.
- A finite set of constants.
- Addition, subtraction, multiplication and division operators.

An individual heuristic in this representation is incorporated directly into the source code of a SAT solver and assessed in the usual way. ADSSEC uses a fitness measure based on the number of variable decisions made and also, in the case of learned clause ranking, the number of literals in learned clauses, in order to promote short learned clauses to save memory.

7.7 Learning to Select Solver Parameters

7.7.1 GPs for Selecting Multiple Solver Parameters

In Section 7.3.1 above we described the approach taken by Kibria and Li (2006) to learning to initialize variable activities. Kibria (2007) build on this idea to target multiple solver parameters covering variable activities, clause activities, clause deletion, and restarts. For example, MiniSat maintains a bound on the number of conflicts allowed during a search, after which it restarts. At each restart the thresholds for the number of conflicts before the next restart, and the number of learned clauses that can be added before pruning them, are increased. The work involves replacing the fixed thresholds and parameters used by MiniSat with a more dynamic system that is learned from examples.

The following constants are introduced:

1. θ_1 is used to increase the increment for variable activities. If i_v is the increment, such that activities are updated as $a(v) = a(v) + i_v$, then the increment is updated as $i_v = \theta_1 i_v$.
2. θ_2 is used to increase the increment for clause activities, in an analogous way to θ_1.
3. θ_3 and θ_4 are thresholds for forcing a restart and forcing pruning of the learned clauses respectively.
4. θ_5 is used to decide which learned clauses to prune: if there are n learned clauses and the clause activity increment is i_c then clauses with $a(c) < \theta_5(i_c/n)$ are discarded.

In addition, a simple neural network is added. This network has eleven inputs, and at each conflict it is applied to each literal in a learned clause. The inputs specify the size of the the learned clause, the number of clauses learned since they were last pruned, the polarity of the literal, and so on. The network has four outputs N_i. N_1 is used to update the activity of the current literal's variable v as $a(v) = a(v) + N_1 i_v$. N_2 and N_3 specify values added to the restart and clause pruning counters. N_4 is used to decide whether T or F should be assigned to v. This neural network, along with the constants θ_i, are learned using an EP method. The fitness is once again the total time taken to solve a set of training instances.

7.7.2 Evolutionary Strategies for Learning CDCL Solver Parameters

Kibria (2011) applies both GAs, and a variant of the GA approach known as an *evolutionary strategy (ES)*, to try to learn good settings for seven of MiniSat's parameters. Real-valued parameters are represented in the form $G = (v, l, u, \sigma)$, where v is a parameter value, l and u are lower and upper bounds for v, and σ is a parameter that we will explain below. A potential solution, if we are trying to learn m parameters, then takes the form $S = (G_1, \ldots, G_m)$.

The ES takes essentially the same form as a GA: a population of potential solutions is maintained, solutions are selected and combined to produce offspring, and these can be subject to mutation. A new population is then selected, either from the offspring alone, or allowing parents also to persist. In an ES however the recombination and mutation operations differ from those generally used in GAs. Recombination for example might take one or more potential solutions $S_1, \ldots, S_k$, and either average corresponding components within them, or select offspring components at random from the parents. Mutation for example might take a component and add zero-mean normally-distributed noise having variance σ. The variances are attached to the components themselves as described above, and consequently are subject to adaptation as part of the algorithm. As with GAs, there are many possible variations on ESs; a more comprehensive introduction can be found in Emmerich *et al.* (2018).

7.7.3 AvatarSAT

Singh *et al.* (2009) describe the *AvatarSAT* system. This applies ML to the problem of setting two of the parameters used by MiniSat. The version of MiniSat used has ten parameters controlling its operation, and empirical experience suggests that the *variable decay rate* used by the VSIDS heuristic, and the *restart factor*, are particularly effective in changing the behaviour of the solver.

AvatarSAT uses two multi-class SVM classifiers. The first uses features of the initial CNF problem to choose one of nine decay rates and one of three restart factors to use when starting MiniSat. After MiniSat has learned a number of clauses exceeding 80% of

those in the original problem, a second classifier repeats the process, this time also using features from the learned clauses. It then has the option of changing the two parameters. Training examples for the SVMs are obtained from a set of training problems by running MiniSat for each problem using all twenty-seven combinations of parameters, and making an example labelled with the combination that achieves the fastest solution. Features are a mixture of the standard features and some designed specifically for this application.

7.7.4 Other Methods for Selecting Solver Parameters

While we do not aim in this work to address in detail the application of generic parameter optimization methods to SAT solvers, it is worth noting two research efforts that have achieved considerable success through this approach.

The *Configurable SAT Solver Challenge* described by Hutter *et al.* (2017) applied three algorithm configuration methods—ParamILS (Hutter *et al.*, 2009), SMAC (Hutter *et al.*, 2011) and GGA (Ansótegui *et al.*, 2009)—to a collection of SAT solvers in a competitive environment, and found that the resulting parameter settings were often much better than the defaults that would otherwise be used.

The *SATenstein* solver (Khudabukhsh *et al.*, 2016) is a local search SAT solver that has been constructed to include elements of existing solvers along with other methods, and highly parameterized with the aim of providing a high degree of flexibility to an algorithm configuration method, in this case ParamILS. The combination of this flexibility and automated parameter tuning again leads to notably high performance.

7.8 Specializing a SAT Solver at the Source Code Level

The *MiniSat Hack* track of the annual SAT Solver Competition[3] allowed modified versions of MiniSat to compete. This part of the competition was based on the observation that many improvements to SAT solvers involve changing only a small number of lines of source code, often at most 10. Consequently, entries were limited to modifications comprising changes of at most a thousand non-space characters.

Petke *et al.* (2013) and Petke *et al.* (2014) have attempted to apply a form of GP known as *Genetic Improvement*, previously applied to automated software patching, in order to produce improved versions of MiniSat. In their work, individual versions of a program are allowed to compete against one-another. Each is a variant of a single core program—in this case MiniSat—and individuals are represented as a list of modifications to be made to the source code of the core program. These modifications operate on single lines of source code, and consist of three kinds: deletions, replacements, and copying. A variety of constraints

[3]The MiniSat Hack track was held in 2009, 2011, 2013 and 2014. Information on all past SAT Solver Competitions can be found at `satcompetition.org`.

are imposed, such as the maintenance of pairs of brackets. A key idea was to include entries from the MiniSat Hack track as a source of potential modifications.

The algorithm itself runs much as any GP: a population of individuals is maintained, and a fitness function based on number of lines executed and accuracy on a source of test SAT problems is used to decide which individuals survive. Crossover consists of concatenating lists of modifications. A key difference is that a postprocessing step is used in an attempt to produce a fully optimized final individual.

It was found that, while this approach provided minimal benefit in improving MiniSat for general problem classes—one problem here was that individuals simply learned to remove code such as that used for gathering statistics—a significant improvement could be achieved when searching for a specific version aimed at a single class—in this case Combinatorial Interaction Testing.

7.9 Discussion

The immediate, striking conclusion to be drawn from this Chapter is that researchers have approached ML for SAT-solvers in an extraordinary variety of ways: at essentially every point in the solver structure that is open to ML, and using essentially every tool in the ML toolbox. (In fact, a state-of-the-art SAT solver may be using several ML algorithms simultaneously.) It is however possible to identify some important core lessons and themes.

7.9.1 Lightweight Versus Heavyweight Methods

It is easy to destroy the performance of a CDCL solver: in adding a new component at any key point, that component must either achieve a sufficient positive outcome quickly, or be so effective that the impact of the time taken to do its job is less than the improvement it delivers. (Ideally, it should of course be very fast and very effective, but there is an inevitable trade-off.)

This is reflected in the results to date on learning to select preprocessors, existing heuristics, and restart strategies, and learning to delete learned clauses. In each of these areas the results are dominated by what might be termed *lightweight* methods: linear regression, decision trees, smaller MLPs and so on.[4]

A notable, and worrying, characteristic of much work applying lightweight methods has been that often little attention is given to optimizing hyperparameters. (There are of course many exceptions—the AutoFolio system for example (Lindauer *et al.*, 2015) performs hyperparameter optimization as an integral part of its operation.) This applies also to work

[4]Given the considerable body of work applying GNNs to SAT, it is surprising that no attempt appears to have been made to apply *graph kernels* (Vishwanathan *et al.*, 2010) with SVMs. The latter are known to be extremely effective, while falling into the lightweight category.

described in other Chapters. Even for a very simple method such as ridge regression, with a single hyperparameter (λ in Equation (3.6)), correctly setting that parameter can hugely affect performance. It is interesting to wonder how better attention to this detail might improve results. Wainer and Fonseca (2021) and Bischl *et al.* (2021) both provide recent discussions of methods available for achieving this most effectively.

In the area of learning to select variables there is a greater diversity of methods; here, everything from the simplest of ML methods to deep architectures requiring significant specialized computing resources has been applied. It is unclear at present whether any particular level of complexity will turn out to be appropriate; this is very much a developing area.

It has certainly been demonstrated that there is great potential for the more complex methods. A common conclusion is that they demonstrate a clear ability to make better decisions than heuristics such as (E)VSIDS in the sense that fewer decisions are needed to solve problems. At present this is often at the expense of requiring more time to find these solutions, on account of the time taken for these methods to make their choice.[5]

The situation is complicated by the disparity of the methods used to implement state-of-the-art CDCL solvers—typically in highly optimized C or C++—versus deep architectures using a mixture typically of Python with specialized GPUs. This begs both the question of how CDCL solvers might benefit from specialized hardware, and how deep architectures might benefit from further optimization of their code.

It seems inevitable that deep architectures will indeed prove to be of long-term benefit in this area; there is perhaps significant opportunity for further research applying them to restart strategy selection, clause deletion and so on. However I expect that significant opportunities also remain for lightweight methods. One reason for this belief is that work such as that of Flint and Blaschko (2012) demonstrates that carefully-judged use of such a method, in this case a perceptron, in a central part of the solver can still achieve very significant results. The other reason relates to the question of precisely what we want a variable selection heuristic to achieve: generality or specialization.

7.9.2 General Versus Specialized Heuristics

(E)VSIDS might itself be considered a learning algorithm: it observes how variables are used and updates activities accordingly. The CHB, LRB and GLR heuristics are most definitely ML methods. The application of ML here has, for the purpose of our discussion, three critical characteristics:

1. It is extremely lightweight.

[5] This situation is by no means unique to applications in the ATP domain. For example, deep architectures also suffer from problems when translated to mobile devices, as their power and memory requirements can be limiting, and significant research currently attempts to address this.

2. It learns on a *per-instance* basis: the heuristic learns from scratch whenever it is faced with a new instance of the SAT problem.

3. As a result, it is a *general* heuristic: it has not been specialized for any class of problems.

It is common when using more involved learning methods to treat the problem as one of learning *specialized* heuristics, in the sense that the learning employs a collection of related problems, and it is assumed that new problems will share the same characteristics.

It is worth emphasizing that these two scenarios differ in their use of training data. In the former case, there is no fixed set of training examples; rather, the training data is generated as a side-effect of the proof process on a per-instance basis. (We shall see a further notable example of such a process, which is central to the QFUN algorithm described in Section 9.2.2.) As a result, learning adapts the solver to specific instances and generalization across problem classes is automatic. In the latter case a specific collection of SAT problems is typically required; as a result learning often has the aim of specializing a solver to problems represented by the collection. In this case learning happens once and the outcome is used for all future problems, meaning that the time-scale for learning is fundamentally different. (And while any ability the learner gains to generalize to other problem classes is undoubtedly of interest, it is reasonable to expect that this might be limited.) Of course, whether the aim is to obtain a general solver or one specialized to some particular problem class will in practice be dependent on the desired application.

It seems then, that one must exert great caution in directly comparing the two approaches: they work on different timescales (per-instance versus per-domain) and thus solve different problems. In fact, shouldn't one aim to exploit the abilities of both?

7.9.3 Are Hybrid Methods the Sweet Spot?

To date, when demanding deep architectures have been added to SAT solvers, tested using *realistic data*, and shown to improve both decision quality *and the time taken to find solutions*, it has usually been by *combining* the strengths of existing heuristics such as (E)VSIDS with the more demanding methods. In the present Chapter this was particularly apparent when two kinds of hybrid approach were employed. First, when predictions by a deep architecture were used periodically, to re-set the variable activities (for example Selsam and Bjørner (2019) and Han (2020a)). Second, when existing variable scores were used as inputs for a modified calculation of variable scores (for example Flint and Blaschko (2012) and Illetskova *et al.* (2017)).

In Chapter 9 we shall describe work by Lederman *et al.* (2019) that has achieved similar success using a hybrid approach exploiting elements of an existing solver in the context of the

QSAT problem. Perhaps it is hybrid solvers that provide some of the greatest opportunities for advancing the field?

7.9.4 Other Unanswered Questions

We end this Chapter by noting three further potential directions for improving SAT and QSAT solvers using ML.

First, many of the results described in this review rely on reinforcement learners. Kurin *et al.* (2019) note that, despite the successes achieved, one might expect to encounter very considerable challenges in trying to scale such methods to address realistic problems. These challenges are not limited to the SAT/QSAT domain, but are likely to be encountered in any sufficiently complex environment.

Second, we have described a single piece of research applying ML to the cube and conquer approach to SAT solving. This area presents multiple opportunities for new research.

Finally, the disparity between satisfiable and unsatisfiable instances seems underexplored. It has been noted by multiple authors, and several have also noted that training only on unsatisfiable instances can improve performance on satisfiable ones. There are clearly opportunities here.

8 Learning to Improve Local-Search SAT Solvers

Local search solvers take a fundamentally different approach to the SAT problem than most of the solvers to which ML has been applied. The basic algorithm was presented in Chapter 2, and for convenience is repeated here:

```
1 Select an initial solution candidate x ∈ {0,1}^n;
2 totalFlips = 0;
3 while totalFlips < flipLimit do
4     if x is a satisfying assignment then
5         return x;
6     totalFlips = totalFlips + 1;
7     Choose a variable to flip;
8     Flip the chosen variable in x;
```

The only point in this algorithm at which we can reasonably apply a heuristic choice that might be improved by ML is in the choice of the next variable to flip. Hand-engineered heuristics for this choice have attracted great attention, and provide a basis for some of the learning methods used; we provide some initial background material in Section 8.1. It is perhaps because of the limited choice of targets that relatively little work applies ML to

local search SAT solvers. Nonetheless, some successful attempts have been made, which we now describe.

Local search has attracted particularly keen interest from evolutionary computing researchers, and several successful systems have been developed. These are described in Section 8.2.

Local search solvers are often parameterized. Section 8.3 describes methods related to empirical hardness models, that aim to allow good parameters to be selected with the aim of minimizing a solver's runtime.

Recently GNNs have been applied in conjunction with reinforcement learning in an attempt to learn good variable selection heuristics. This work is described in Section 8.4.

The *STAGE* method is unusual in that, rather than targeting variable selection, it attempts to learn to identify good restarting points. The process here is analogous to the use of restarts by CDCL solvers: it can be beneficial periodically to stop the search process and restart, only now a restart involves choosing a new initial solution candidate $\mathbf{x}$ in the algorithm above. This method is described in Section 8.5.

8.1 Standard Variable Selection Heuristics for Local Search

The literature on variable choice heuristics, in the absence of an ML component, is extensive, and relies on some basic definitions that we will need to re-use (Fukunaga, 2008). Let f be a CNF formula and A an assignment. Define

- $\text{UNSAT}(f, A)$ to be the number of clauses in f that are unsatisfied for A.
- A_v to be the assignment that flips the assignment of variable v but is otherwise identical to A.

The *net gain* $G(v, A)$ of a variable v for assignment A is then

$$G(v, A) = \text{UNSAT}(f, A_v) - \text{UNSAT}(f, A).$$

Similarly, the *negative gain* of a variable for A is the number of clauses that were satisfied but will be made unsatisfied by flipping v, and the *positive gain* is the number of clauses that were unsatisfied but will be made satisfied by flipping v. The *age* of a variable v is the number of flips made to variables since v was last flipped.

8.2 Evolutionary Learning of Local Search Heuristics

8.2.1 CLASS

We saw in Chapter 5 that GAs have been used to solve SAT directly. In Fukunaga (2002), Fukunaga (2004), and Fukunaga (2008) the *CLASS* method is developed; this moves

beyond the straightforward application of evolutionary methods to SAT-solving. The central observation leading to these developments was that the variable selection heuristics used by existing local search solvers could generally be represented as simple combinations of a small number of basic primitives, such as net, positive or negative gain. For example, GSAT simply selects a variable with the best net gain; on the other hand, WalkSAT's heuristic is as follows:

```
1 Randomly select an unsatisfied clause C = {l_1, ..., l_m};
2 Let V = {v ∈ C | v has 0 negative gain};
3 if |V| > 0 then
4     select v ∈ V at random;
5 else
6     if binomial(p) then
7         select v ∈ C at random;
8     else
9         select the v ∈ C having smallest negative gain;
```

Algorithms such as these can be represented as LISP s-expressions (McCarthy, 1960), and, as s-expressions have a clear interpretation as trees, any pair of s-expressions can be combined using familiar GP methods.

The original CLASS system described in Fukunaga (2002) takes this approach. If S_1 and S_2 are s-expressions implementing heuristics, then CLASS combines them to generate ten offspring. The composition method is not a standard one, but was designed specifically for this application. Two examples of the ten offspring produced are as follows; in both cases v_1 and v_2 are the variables selected by S_1 and S_2:

1. The s-expression that returns v_1 or v_2, according to which has the smaller negative gain.

2. The s-expression that returns v_1 or v_2, according to which is oldest.

The remainder of the system operates as a standard GP method, with the initial population containing randomly generated s-expressions. In addition, it was found that improved heuristics could be developed by modifying the composition method such that it sometimes introduces a heuristic taken from a collection of known good heuristics.

In Fukunaga (2004) numerous improvements were made to CLASS, leading to the CLASS2.0 system. The improvements were mostly aimed at improving efficiency, but also led to the development of further good heuristics without needing to introduce known good heuristics during composition. In Fukunaga (2009) a further effort involving the use

of parallel computing led to further improvements in the learned heuristics. It is notable that, until significant effort was expended to improve efficiency, CLASS suffered from the problem commonly encountered when attempting to improve heuristics for CDCL solvers: while improvements were found in the number of flips needed by the learned heuristics to find a solution, such improvements were negated because these heuristics took too long to evaluate.

8.2.2 Inc* and Friends

Further GP-based approaches to learning heuristics for local search SAT solvers can be found in Bader-El-Den and Poli (2007), Bader-El-Den and Poli (2008b), Bader-El-Den and Poli (2008c), and Bader-El-Den and Poli (2008a). There are two main contributions here: the first is the *Inc** algorithm, and the second a set of methods for evolving heuristics more closely related to Fukunaga's work.

The initial development of the Inc* algorithm by Bader-El-Den and Poli (2008c) begins with the idea that a SAT problem can be addressed by local search in an incremental manner. (This differs somewhat from the form of incremental solving used in CDCL solvers, but is certainly related.) The idea is to work with a subset of the problem clauses called the *clauses active list* (CAL). The algorithm begins with a small CAL and runs a local search solver. If the solver is successful then further problem clauses are added to the CAL and a further solution attempt is made. If a solution attempt is unsuccessful then the CAL is reduced in size; the clauses to be removed are chosen using a weighting scheme that measures the number of variable flips needed to satisfy the CAL after the addition of each clause.

A GP is used to improve this process by learning a function called `ifSuccess`. The GP learns a tree representation of this function having root node `ifSuccess`(p_1, p_2), where p_1 and p_2 are themselves evolved programs. The interpretation is that if a solution attempt is successful then p_1 computes the number of clauses to add to the CAL; conversely if it is unsuccessful then p_2 computes the number of clauses to remove.

A further variant of this algorithm called Inc** is presented in Bader-El-Den and Poli (2008a), where an improved clause weighting scheme is introduced along with an improvement to the way in which the algorithm increases the allowed number of flips after an unsuccessful solution attempt.

Bader-El-Den and Poli (2007) takes a more direct approach by using a relatively standard GP to evolve a variable-selection heuristic. The aim here is similar to Fukunaga's, in that a collection of basic primitives used by existing successful heuristics is identified, and these are then combined in the usual way by a GP to make new, more complex heuristics. The work differs from Fukunaga's in that standard GP recombination is used instead of the composition method described above, which is criticised for producing heuristics which are

time-consuming to compute.

The work by Bader-El-Den and Poli (2007) is related to that by Bain *et al.* (2005a) and Bain *et al.* (2005b), who have also attempted to learn heuristics for SAT and MAXSAT problems, but again use standard GP crossover. Experiments suggest that, in learning heuristics for local search, the standard operators may perform better under various measures; conversely Fukunaga (2008) reports an inability to obtain good performance using standard operators.

Bader-El-Den and Poli (2008b) attempt to improve on the GP approach to evolving heuristics by allowing such heuristics to interact with the Inc* algorithm. This is achieved by allowing them to use, for example, the size of the CAL, and a Boolean variable indicating whether the last solution attempt was successful, in the corresponding evolved computation.

8.3 Learning Good Parameters for Local Search Solvers

Hutter *et al.* (2006) observe that, as local search algorithms are typically randomized, the run-time for a given problem can differ for consecutive runs of the solver. Based on this, they develop a method extending the idea of an empirical hardness model.

As run-times can vary it is necessary to consider the *distribution* of run-times for an instance, instead of trying to predict a single run-time. It has been observed experimentally that the run-time distribution often conforms to a standard distribution, such as the exponential distribution, and so this might be achieved by learning to predict a small number of parameters. The work applies this idea to the solvers Novelty$^+$ (Hoos, 1999) and SAPS (Hutter *et al.*, 2002). Instances are described by features derived from the standard features, which are reduced by forward selection (Hastie *et al.*, 2009). Supervised learning is applied using ridge regression with basis functions corresponding to individual features, and products of pairs of features.

This modification is successful, and it is then further extended to incorporate the parameters associated with Novelty$^+$ and SAPS. For each method, a set of parameter settings is chosen, and these parameters are added to the features. After training, it becomes possible to use the classifier for parameter *selection*, as the classifier can be used to predict the median of the run-time distribution based on both instance features and parameters. The parameters expected to lead to the shortest run-time can then be chosen.

8.4 GNNs for Learning in Local Search

Yolcu and Póczos (2019) have applied RL to the learning of variable selection heuristics, using a GNN to represent the learned policy, with the aim of specializing local search solvers to particular classes of SAT problem.

For a formula ϕ with n variables v_i, the problem can be cast in an RL format as follows:

1. A state is the pair (ϕ, A), where A is the current assignment to the variables.
2. An action consists of flipping one of the n variables.
3. State transition simply modifies A to reflect the flipping of the relevant variable.
4. Reward is 1 if a satisfying assignment is found, and 0 otherwise.

To learn a policy, a GNN is used. This represents ϕ in a manner related to those used by Selsam *et al.* (2019) and others, and described in Chapter 5, Section 5.3, although with notable differences needed to adapt it to this scenario. The complete, learned NN represents a function

$$f_{\boldsymbol{\theta}}(A) = d(v_1, \ldots, v_n).$$

Here, $\boldsymbol{\theta}$ is a vector of learned parameters, and d is a probability distribution on the variables. Note that ϕ is implicitly involved as its structure is used to define f.

The policy derived from f selects a variable by sampling from d with probability $1/2$, and otherwise selecting a variable at random from a randomly selected, unsatisfied clause. The overall system is trained using the REINFORCE algorithm (Williams, 1992) and a curriculum learning approach starting with simple problems and using their solution as a basis for learning harder problems.

After training on a collection of relatively straightforward SAT problems—numbers of variables range from 50 to 100, with 213 to $1,725$ clauses—the authors find that the system makes more effective decisions than the standard WalkSAT heuristic, but at increased computational cost in terms of runtime.

8.5 Other Approaches to Learning in Local Search

The STAGE method presented in Boyan (1998), Boyan and Moore (1998), and Boyan and Moore (2000) is a general optimization method, but one applied successfully to the SAT problem. Consider a local search method π for solving SAT, based on the bit-string representation $\mathbf{x} \in \{0,1\}^n$ of a solution and aiming to minimize some cost function $F(\mathbf{x})$, such as the number of unsatisfied clauses. Any run of π starting at $\mathbf{x}_1$ gives rise to a sequence

$$\pi(\mathbf{x}_1) = ((\mathbf{x}_1, F(\mathbf{x}_1)), \ldots, (\mathbf{x}_m, F(\mathbf{x}_m)) \tag{8.1}$$

of potential solutions with corresponding costs. How might we choose a starting point $\mathbf{x}_1$? Define the function $V^\pi(\mathbf{x})$ to be the best cost encountered if we run π from starting point

$\mathbf{x}$.[1] If we had access to this function, and could find its minimum, then the problem would be solved; however the function is not known.

Rather than using $\mathbf{x}$ values directly, STAGE uses features derived from $\mathbf{x}$ using a function $f : \{0,1\}^n \to \mathbb{R}^{n'}$, and learns a function V' with the aim of approximating $V^\pi(\mathbf{x})$ as $V'(f(\mathbf{x}))$. Now, any run of π provides a sequence as in Equation (8.1) and this provides m training examples of the form $(f(\mathbf{x}_i), F_{\min})$ where $i = 1, \ldots, m$ and $F_{\min}$ is the smallest value of $F(\mathbf{x}_i)$ occurring in $\pi(\mathbf{x}_1)$. By interleaving runs of the solver WalkSAT, with learning using polynomial regression to estimate V', it was found that good restarting points for WalkSAT could be found as

$$\mathbf{x}' = \operatorname{argmin} V'(f(\mathbf{x}))$$

using hill-climbing search to find $\mathbf{x}'$.

8.6 Discussion

Local search has provided fertile ground for evolutionary learning, which has achieved considerable success. There appears to be some disagreement regarding whether standard or specialized GP crossover methods are preferred but, aside from this, little if any controversy.

What is particularly clear is that, once again, lightweight ML methods—particularly those inspired by EHMs—can show great success. More involved methods however—here, based on evolutionary computing or the combination of GNNs with RL—can sometimes learn heuristics that make better decisions, but at the expense of being expensive to compute, sometimes to the extent that they are, in their current form, counterproductive. Later work on learners of the Inc* type has attempted to address this, but in general the trade-off of effectiveness against complexity continues to present opportunities for new work.

As we have now seen several times, there is a need to draw a distinction between heuristics that are intentionally *general*, which have usually been the target of earlier research, and heuristics that are *specialized* to a particular problem class, which more recent work has tended to address. It is not unusual for the former to be used as a baseline for comparison with the latter. This is a sensible way to check that the specialized heuristic reaches a basic level of performance. However care is needed, as it is often possible for the general heuristic to outperform the specialized one if the class of problems used for testing changes.

Finally, it is perhaps surprising that such significant effort has been expended on local-search for SAT without similar attention being paid to the use of *large neighbourhood search* algorithms (Shaw, 1998; Pisinger and Ropke, 2010). Here perhaps lies significant opportunity for further application of ML.

[1] The notation V^π is similar to that commonly used to denote a *value function* in reinforcement learning. This is not accidental; see Boyan and Moore (2000) for a discussion.

9 Learning to Solve Quantified Boolean Formulas

Perhaps the simplest way to generalize the SAT problem is to introduce quantifiers, leading to *quantified Boolean formulas (QBFs)*. In a QBF, variables can be arbitrarily universally ($\forall$) or existentially ($\exists$) quantified. The interpretation of the quantifiers is straightforward: let ϕ denote some Boolean formula, and let $\phi[x = B]$ with $B \in \{T, F\}$ denote the formula obtained by replacing the variable x throughout ϕ with its assigned value, T or F. Then

$$\forall x.\phi \equiv \phi[x = T] \wedge \phi[x = F] \tag{9.1}$$

$$\exists x.\phi \equiv \phi[x = T] \vee \phi[x = F]. \tag{9.2}$$

This definition is extended such that if Φ is a QBF with a free variable x, and $Q \in \{\forall, \exists\}$, then $Qx.\Phi$ is also a QBF, in which x is now bound, and with an interpretation that is the obvious extension of the above. Let free(Φ) denote the set of variables that are free in Φ. If free(Φ) $= \emptyset$ then Φ is *closed*, and Φ can only take the value T or F.

When free(Φ) $\neq \emptyset$ let $\mathbf{x}$ = free(Φ). Then, if the identities in Equations (9.1) and (9.2) are applied, we can treat Φ as a Boolean formula $\Phi(\mathbf{x})$. The aim of the QSAT problem in general is to establish whether or not $\Phi(\mathbf{x})$ is satisfiable. However, as this is equivalent to establishing the truth value of $\exists\mathbf{x}.\Phi(\mathbf{x})$ it is usual only to consider closed QBFs.

In general a QBF will have nested quantifiers. Let $Q \in \{\forall, \exists\}$ denote a quantifier, and let $\overline{Q} = \forall$ when $Q = \exists$, and $\overline{Q} = \exists$ when $Q = \forall$. We abbreviate a formula of the form $Qx_1Qx_2 \cdots Qx_n.\phi$ to $Q\mathbf{x}.\phi$ where $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$. Similarly, we abbreviate a formula of the form

$$Q_1 x_1^{(1)} \cdots Q_1 x_{n_1}^{(1)} Q_2 x_1^{(2)} \cdots Q_2 x_{n_2}^{(2)} \cdots Q_m x_1^{(m)} \cdots Q_m x_{n_m}^{(m)}.\phi$$

where $Q_{i+1} = \overline{Q}_i$ to

$$Q_1\mathbf{x}_1 Q_2\mathbf{x}_2 \cdots Q_m\mathbf{x}_m.\phi$$

where $\mathbf{x}_i = \{x_1^{(i)}, \ldots, x_{n_i}^{(i)}\}$, and the quantifier in $Q_i\mathbf{x}_i$ is said to have *level* i. If ϕ is propositional then the formula is said to be in *prenex* form, the initial block $Q_1\mathbf{x}_1 Q_2\mathbf{x}_2 \cdots Q_m\mathbf{x}_m$ is called the *prenex*, and ϕ is called the *matrix*.

Interest in finding good QSAT solvers is driven by the development of encodings for various practical problems; for example, in conditional planning (Rintanen, 1999), knowledge representation for non-monotonic reasoning (Egly *et al.*, 2000) and bounded model checking (Dershowitz *et al.*, 2005). Further, there is evidence that QSAT codings can be more compact than SAT encodings for similar problems. Conversely. the expansions in Equations (9.1) and (9.2) allow a QSAT problem to be converted to a SAT problem, with the obvious potential for an exponential increase in the size of the formula to be tested for satisfiability. The trade-off is not straightforward, and so research on solvers of both kinds is ongoing.

The possibility of an exponential increase in problem size suggests that this obvious approach to solving QSAT is unlikely to be successful; and indeed, this is the case in practice. In complexity terms, QSAT is PSPACE-complete (Garey and Johnson, 1979): we should expect its solution to be challenging in general. Nonetheless, methods have appeared that can deal with interesting problems of realistic size.

As might be expected, these methods fall into different camps, depending on their underlying approach. For example, the testing of QBFs for validity can be achieved by a search process that is a modification of the DPLL procedure (Samulowitz and Memisevic, 2007). At each node in the search tree the formula is modified—for example, by choosing a variable and setting it to be true or false—and each sub-tree recursively tries to solve a modified version of the formula. Heuristics can be used here for example to choose which variable to assign. We will not describe further the multitude of solvers available, other than to the extent necessary to explain how ML has been incorporated.

9.1 Learning for Portfolios of QSAT Solvers

Clearly one way of applying machine learning to the QSAT problem is through the construction of portfolio solvers. This can be achieved using precisely the same set of methods as we saw in Chapter 6: a classifier is trained to predict, on the basis of syntactic features computed from a formula, which of a collection of solvers will be most effective in solving that formula.

Pulina and Tacchella (2007) were the first to explore this possibility. While their approach is essentially as expected, it is notable in that it applies a collection of classifiers, rather than a single classifier, to the problem. Specifically, decision trees (Quinlan, 1993), decision lists (Rivest, 1987), logistic regression and nearest-neighbour methods are used. By moving beyond the simpler techniques, decision trees in particular are found to perform very well.

The initial set of 141 features used in Pulina and Tacchella (2007) was related to the existing standard features for SAT solvers—described in Chapter 4—augmented using simple syntactic features specific to QBFs; it was later shown, with the help of a straightforward *forward selection algorithm* (Hastie *et al.*, 2009), that the initial 141 features could be reduced to 20 with little loss of performance. In addition, having grouped the available QBF solvers into two fundamental types—*search-based* and *hybrid*—the authors applied the *Partition Around Medoids* (Kaufman and Rousseeuw, 1990) clustering algorithm to show that these can be treated as latent classes, in that formulas tend to be more successfully addressed by one of the two types of solver.

A further contribution of note in Pulina and Tacchella (2007) is the introduction of two methods, specific to the assessment of portfolio solvers, for measuring the performance of a classifier. This is important as it provides an approach more closely aligned to the relevant

problem than a standard measure such as accuracy. It was first noted that the cost of a solver-selection method should take account of any limit imposed on how long a solver is run for. To take account of this the following cost was used:

$$1 - \frac{1}{l|F|} \sum_{f \in F} t(f)$$

where F is a set of formulae, $t(f)$ is the time taken by the solver selected for f, and l is the time limit. This cost varies between 0 and 1, with values close to 1 indicating that selected solvers rarely need the full time limit. The second cost proposed is

$$1 - \frac{1}{|F|} \sum_{f \in F} \frac{t(f) - t'(f)}{t(f)}$$

where $t'(f)$ is the time taken by the best solver for f. This cost indicates how far the selected solvers fall short of the best available solvers.

Pulina and Tacchella (2009) take this work further. The authors note that the system described fails to prove 24% of the theorems that one or more of its constituent provers is capable of proving. In addition, by studying the distribution of run-times for the constituent solvers over a large number of problems, it is noted that a solver will tend either to solve a problem within a few seconds, or run for considerably longer.[1] To address this they introduce a more sophisticated method for scheduling solvers. This method is assigned an overall limit on the time available. A learner is used as in the earlier system to choose a solver, and a time limit set for its execution. The solver is run for the assigned time; if a solution is found we are done, otherwise alternative solvers are tried according to a specified ordering, each being assigned some portion of the remaining time available, until a solution is found or no time remains. If at any point in this process a solver other than the first, chosen using the learner, finds a solution then we have essentially generated a new training example. This example is added to the training set and the learner is re-trained.[2]

There are essentially three areas in this algorithm that are open to variation: the underlying machine learner used, the ordering imposed on solvers, and the approach to assigning a time limit, given the remaining time available. Several approaches to each of these areas are explored in Pulina and Tacchella (2009), ultimately allowing the system to achieve a considerable improvement in performance over its earlier version.

[1] This is in fact a common observation made by authors when experimenting with QSAT solvers on large collections of problems: most problems are either easy, or not solvable within the time limit of interest; solvers are differentiated largely according to their ability to solve the remaining problems.

[2] The full version of the algorithm also accounts for the case where all solvers fail but the time-out is not reached. The reader should consult the original work for full details.

9.1.1 Good Features for QSAT Portfolios

Hoos *et al.* (2018) note that QSAT solvers might be expected to benefit from a portfolio approach, because it has been observed (Lonsing and Egly, 2018) that solvers based on *expansion* tend to be more successful for problems with a small number of quantifier alternations, while solvers based on *search* tend to dominate for problems with many quantifier alternations. They use a general method called *AutoFolio* (Lindauer *et al.*, 2015) to build a portfolio using the solvers QuAbS (Tentrup, 2019; Giunchiglia *et al.*, 2006), QFUN (Janota, 2018), Qute (Peitl and Slivovsky, 2017) and GhostQ (Klieber *et al.*, 2010).

In order to construct the portfolio they design a set of features specifically with the circuit-based QCIR problem format in mind. The first set of twenty-three features is computed before any solver is run. This set contains features relating to the numbers of existential and universal variables, number and sizes of quantifier blocks, sizes of gates, numbers of gates of different types, and so on. The second set of eight features is computed after a short run of the Qute solver, and contains features such as the number of backtracks, the number of learned clauses, fraction of assignments due to branching versus propagation, and so on.

The study has two notable outcomes. First, that the second set of features is essentially unhelpful.[3] Second, that only three of the features in the first set are needed to attain 99% of the full performance. The work identifies the following four features as being particularly useful:

1. The circuit depth. (The best, single feature.)
2. The number of quantifier blocks.
3. The average size of the blocks.
4. The relative standard deviation of the gate depths.

9.2 Learning in Non-Portfolio QSAT Solvers

9.2.1 Learning to Select a Heuristic

Samulowitz and Memisevic (2007) were the first to apply non-portfolio machine learning methods to the solving of QBFs. Their approach is to learn to select a good heuristic for a single solver: they train a linear logistic regression model to predict the best heuristic for each of ten available heuristics, under the assumption that the heuristic chosen at the

[3] While relevant to the domain of first-order logic, rather than SAT/QSAT, it is perhaps relevant that a similar observation was made by Bridge *et al.* (2014).

beginning of the search is then fixed and used at all other nodes.[4] They use seventy-eight features, some related to the standard features for SAT based on Nudelman *et al.* (2004) and some specific to quantified formulas. In addition to simple heuristic selection, they go further by allowing the trained classifier to choose a heuristic at every node in the search tree, allowing the heuristic to change as the search progresses, and find that this is on the whole advantageous.

9.2.2 Learning Strategies

The method of *Counterexample Guided Abstraction Refinement (CEGAR)* (Janota *et al.*, 2016) forms the basis for several QSAT solvers. Janota (2018) developed the solver *QFUN*, which added machine learning to CEGAR with considerable success.

This application of ML has certain similarities to (E)VSIDS and related heuristics, in that learning takes place from scratch every time a solver is run on a problem, and the solver generates the learner's training data as it runs. Learning takes place as an integral part of the solver; consequently, we will need to present considerably more machinery to explain this method. In the following, some details are omitted for brevity, but the critical themes remain.

Solving a QBF can be cast as a two-player game, where one player makes choices for the universally quantified variables, and the other for the existentially quantified variables. We denote the two players by $P_\forall$ and $P_\exists$. Let ϕ be a Boolean formula and let Φ be a prenex, closed QBF

$$\Phi = Q_1 \mathbf{x}_1 Q_2 \mathbf{x}_2 \cdots Q_m \mathbf{x}_m . \phi$$

where as usual $Q_{i+1} = \overline{Q}_i$. The fundamental idea is that $P_\forall$ tries to choose values for its variables to make ϕ false, while $P_\exists$ tries to choose values for its variables to make ϕ true. Play begins at Q_1 and moves along the sequence of quantifiers. Players alternate along the sequence of quantifiers, and $P_\forall$ wins if the eventual formula ϕ—now with all variables set to T or F—is F, and $P_\exists$ wins otherwise. A *strategy* for player P_Q at level i is a mapping from the values chosen for $\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}$ to a set of values to be assigned to $\mathbf{x}_i$.

In order to illustrate the ideas needed to add ML to this process it suffices to use the specific example of a QBF of the form

$$\Phi = \forall \mathbf{x} . \exists \mathbf{y} . \phi \tag{9.3}$$

and so we shall use this as a working example; the reader should consult Janota (2018) for the details involved in extending the following to the general case.

[4] There is some mixing of nomenclature here. The authors refer to this as a portfolio. While essentially correct, the term 'portfolio' is more usually applied not to a single solver that can use one of a selection of fixed heuristics, but to a collection of solvers each of which can employ its own collection of heuristics.

A *winning move* for $P_\forall$ in Φ is an assignment $\mathbf{x}'$ to $\mathbf{x}$ such that $\exists \mathbf{y}.\phi[\mathbf{x} = \mathbf{x}']$ is F, or equivalently $\phi[\mathbf{x} = \mathbf{x}']$ is unsatisfiable. If $P_\forall$ has made a move $\mathbf{x}'$ then a *counter-move* for $P_\exists$ in Φ is an assignment $\mathbf{y}'$ to $\mathbf{y}$ such that $\phi[\mathbf{x} = \mathbf{x}', \mathbf{y} = \mathbf{y}']$ is T. Given $\mathbf{x}'$, $P_\exists$ can try to find such an assignment using a SAT solver: it simply passes $\phi[\mathbf{x} = \mathbf{x}']$ to the solver and receives a result $\mathrm{SAT}(\phi[\mathbf{x} = \mathbf{x}'])$, which can either be $\mathbf{y}'$ or an indication of unsatisfiability. The formula is false precisely when there is a winning move for $P_\forall$, and an indication of unsatisfiability establishes that Φ is F; but how do we proceed if we obtain a countermove $\mathbf{y}'$? The key idea is to build a sequence

$$S = ((\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n))$$

in which each $\mathbf{y}_i$ is a counter-move to $\mathbf{x}_i$. Starting with a random assignment for $\mathbf{x}_1$, a countermove $\mathbf{y}_1$ is found by running a SAT solver as described. If this is successful, we attempt to extend $S = ((\mathbf{x}_1, \mathbf{y}_1))$ as follows. If at any point we have managed to construct a sequence S, we can find a candidate for $\mathbf{x}_{n+1}$ by running a SAT solver on the formula

$$\neg \bigvee_{i=1}^{n} \phi[\mathbf{y} = \mathbf{y}_i]. \tag{9.4}$$

This is simply the negation of an *incomplete* expansion of the existential quantifier, referred to as an *abstraction*. If there is no satisfying assignment then the QBF is T. However if the SAT solver returns $\mathbf{x}_{n+1}$ then we can use this as a candidate winning move. This process continues until the status of Φ is resolved.

In order to incorporate learning into this process the key observation is that S can act as a training sequence for supervised learning, where we wish to infer a classifier C that can map any candidate winning move $\mathbf{x}$ to a countermove $\mathbf{y}$. In fact if

$$\mathbf{y}^T = \begin{bmatrix} y_1 & \cdots & y_m \end{bmatrix}$$

has m elements the idea is to train m binary classifiers C_j. Provided these classifiers can be expressed as a Boolean formulas $\psi_{C_j} : \mathbf{x} \mapsto y_j$, we can use them to extend the abstraction in Equation (9.4) not just by substituting specific countermoves $\mathbf{y}_i$, but by substituting strategies. Equation (9.4) is then

$$\neg \left(\bigvee_{i=1}^{n} \phi[\mathbf{y} = \mathbf{y}_i] \vee \phi[\psi_{C_1}, \ldots, \psi_{C_m}] \right)$$

where $\phi[\psi_{C_1}, \ldots, \psi_{C_m}]$ is the formula obtained by replacing y_j by ψ_{C_j} throughout ϕ for $j = 1, \ldots, m$. This has the effect of further strengthening the abstraction for $\exists$.

The QFUN solver uses precisely this technique, extended to the case of an arbitrary number of quantifiers. It collects pairs of candidate and countermove vectors, and periodically uses them to learn classifiers, adding multiple formulas ψ derived from the learners to the

abstraction. The need for a classifier that can be expressed as a logical formula limits the pool of methods that can be applied, but the authors successfully employ a *decision tree* learner (Quinlan, 1993) in this way.

Clearly, decision trees are not the only method that can be used in this approach. Santos Silva (2019) explores the use of six classifiers as alternatives to decision trees in QFUN, finding that two were worse, two gave similar performance, and two gave a limited improvement.

9.2.3 GNNs for Ranking Candidates and Countermoves

Yang *et al.* (2019) have attempted to extend the earlier work of Selsam *et al.* (2019) (discussed in Chapter 5, Section 5.3) to the QSAT problem. This essentially requires a modification allowing existential and universal literals to be treated as separate groups. They attempt to predict the status of QBFs of the form of Equation (9.3) directly, and also to predict witnesses, but find that this has limited success. (The problems used had 18 variables, with 10 existential and 8 universal. Clauses had 5 literals, 3 selected at random from the existential variables and 2 from the universal variables.) They argue that this is due to the difficulty GNNs appear to have in addressing unsatisfiability.

As an alternative they then address the use of GNN-based models to rank potential candidates and countermoves in a CEGAR solver, finding that this approach has greater success.

9.2.4 Learning to Improve a VSIDS-Based Heuristic

The method of *incremental determinization* described by Rabe and Seshia (2016) extends many of the ideas used by CDCL SAT solvers, such that propagation of variables is replaced by propagation of *Skolem functions*. The QSAT solver *CADET* implements this idea using an analogue of VSIDS as its central variable-choice heuristic. CADET again applies specifically to QBFs of the form

$$\Phi = \forall \mathbf{x}.\exists \mathbf{y}.\phi.$$

Here, we can try to establish the truth value of Φ by searching for a *Skolem function* f that maps assignments to $\mathbf{x}$ to assignments to $\mathbf{y}$. The formula Φ is true if and only if there is a Skolem function f such that

$$\forall \mathbf{x}.\phi[\mathbf{y} = f(\mathbf{x})] \tag{9.5}$$

is true. If f can be expressed by substituting a suitable Boolean formula in $\mathbf{x}$ for each element of $\mathbf{y}$ then the truth of the formula in Equation (9.5) can be decided by applying a SAT solver to the formula $\neg\phi[\mathbf{y} = f(\mathbf{x})]$.

Lederman *et al.* (2019) applies GNNs and reinforcement learning to replace CADET's use of the VSIDS-based heuristic. (There are noteworthy similarities with the work of Kurin

et al. (2019) described in Chapter 7, Section 7.3.5.) A GNN is used to compute embeddings for each clause and each literal in a problem, and in this application the GNN takes a small number of inputs, corresponding to more traditional features for the clauses and variables. For example, such features indicate whether or not a clause was in the original formula, and whether a variable is universal, existential and so on.[5] In a similar way, the literal embeddings computed for each variable are concatenated with the variable's features and with 5 features describing the current state of the solver; for example, the decision level and the number of restarts so far. Each of the resulting vectors is then passed to its own three-layer feedforward NN, and the outputs of these networks provide a heuristic measure of how desirable each literal is for selection.

The GNN and the feedforward NNs are trained using the *REINFORCE* algorithm described in Williams (1992) and Sutton and Barto (2018). To do this, training problems are executed with a limit of 400 steps, and rewards are assigned, with a reward of -10^{-4} for each step and 1 for a successful proof.

This system was evaluated using a collection of problems involving the finding of logical reductions (Jordan and Kaiser, 2013), and compared against the VSIDS heuristic and a heuristic that selects an action at random in terms of the number of decisions needed and the actual time for finding solutions. This data is quite challenging—the largest problems exceed $1,600$ variables and $12,000$ clauses. The system shows a clear improvement in terms of the number of decision made, but it is of particular note that, for larger time limits, it achieves an increase in performance in terms of time taken despite the learned heuristic having an overhead of around ten times compared to the standard heuristic. This suggests a significant increase in the quality of literal selection obtained. In addition the method shares the ability of some other methods to generalize performance attained when training with these problems to problems in a different set—in this case taken from the 2QBF problem set for one of the QBFEVAL competitions. However in this case it fails to match VSIDS.

9.3 Discussion

The QSAT problem has seen less progress to date than SAT; this reflects its underlying difficulty. Consequently, attempts to improve QSAT solvers using ML are also relatively rare. There is though, sufficient material to suggest that many, now familiar lessons can be learned: much of our discussion for this chapter has a close correspondence to discussions in earlier Chapters, and many of the conclusions are the same.

[5] In an earlier version of this work, the features for a variable included its current VSIDS score, however it turns out that this leads to a small decrease in performance.

9.3.1 Portfolios for QSAT

Portfolios of QSAT solvers have been built very successfully using primarily lightweight learning methods and hand-engineered features; the demonstration that only three, simple features essentially suffice (Hoos *et al.*, 2018) is particularly compelling. It is also interesting that the QSAT domain has been innovative in designing better cost measures and more flexible schedules. The main conclusion here though parallels the lessons learning in the SAT domain: relatively straightforward ML methods can yield very high-quality portfolios.

9.3.2 Can SAT-based ML Methods be Extended?

It is interesting to consider whether the many approaches seen so far to the introduction of ML to SAT might be extended to the QSAT domain. Many QSAT solvers employ a SAT solver at some point in their operation, and we can of course reduce the QSAT problem entirely to a SAT problem by application of Equations (9.1) and (9.2). To this extent the use of ML to solve SAT can easily be exploited for QSAT; in both cases however, this would be to ignore significant opportunities. Taking the QFUN solver described above for example, the introduction of ML that it employs exploits elements of the CEGAR method entirely unrelated to the applications of ML to SAT seen earlier. Multiple methods have now been introduced for the solution of QSAT—Tentrup (2019) provides a concise summary—and these often require methods moving beyond CDCL. It seems therefore that further work on adding ML to QSAT solvers should focus on targeting the specific characteristics of these solvers, rather than simply extending work from the SAT domain.

9.3.3 Complexity and Generality

As in our discussion in Chapter 7, we note that there is a clear and significant contrast between two kinds of learning in the QSAT domain. First, we have methods that learn on a per-instance basis and are essentially general. These are exemplified by the attempts to add learning to CEGAR solvers. Second, we have methods used to specialize solvers to particular problem classes.

It is particularly noteworthy that the *combination* of an existing solver with advanced ML methods, for example in the work of Lederman *et al.* (2019), has led to significant gains for realistic problems. This precisely parallels the results noted in Chapter 7.

10 Learning for Intuitionistic Propositional Logic

Intuitionistic Propositional Logic (IPL) (Dalen, 2001) is a variant of propositional logic in which the *law of the excluded middle* $P \vee \neg P$ and *law of double negation* $\neg\neg P \rightarrow P$ can

not be assumed to hold. Instead we have the *principle of explosion*

$$\neg P \to (P \to Q)$$

and the *law of contradiction*

$$(P \to Q) \to [(P \to \neg Q) \to \neg P].$$

As a result, the process of proving a formula in IPL differs from that needed for classical (non-intuitionistic) propositional logic.

Proving a classical propositional formula f corresponds to showing that $\neg f$ is unsatisfiable. Thus, while we are primarily interested in the SAT problem for classical propositional logic, the fact that some attention has been directed at ML for IPL provers is relevant to this study. More so in fact as the specific characteristics of IPL provide openings for ML to be exploited in ways not available for the classical case.

10.1 Methods Employing the Curry-Howard Correspondence

Sekiyama *et al.* (2017) present a method for learning to prove formulas in IPL, building on the success of *sequence to sequence (seq2seq) learning* (Sutskever *et al.*, 2014) for language translation. The process of finding a proof P for a given formula f in IPL is cast as a problem of *translation* from f to P. As the intuitionistic variant is used, the *Curry-Howard Correspondence* (Pierce, 2002) allows f to be represented as a type and P as a term in simply-typed λ-calculus; this in turn provides a problem representation suitable for training a seq2seq learner.

The authors find that, after training, the network has limited success in predicting correct proofs directly, as the output is often not typeable. However, they then add a search mechanism that attempts to find a typeable term that is *close* to the output term. With this addition, the system is often successful in finding a proof. Termination is not guaranteed if f is not a tautology (and hence not provable), so as a prover the system is incomplete.

This work was later extended. As is commonly the case, the learner in the work just described tries to learn a conditional distribution $\Pr(P|f)$—the probability that P is a proof of f—directly from data. It is the difficulty of learning this distribution directly that led to the limited success of the method. In Sekiyama and Suenaga (2018a) and Sekiyama and Suenaga (2018b) it is noted that as statements in the simply-typed λ-calculus have a hierarchical structure, the probability $\Pr(P|f)$ can be expressed in terms of simpler distributions, and this leads to an approach where a learned probability distribution can be used to guide a search for a proof.

Constructors in the λ-calculus are represented with holes that need to be filled. Paths denoted by `path` specify the locations of holes in an incomplete proof. For example, we

$$\frac{}{\Gamma, A \Rightarrow A}\,\text{Axiom} \qquad \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A \wedge B \Rightarrow \Delta}\,\wedge\text{L}$$

$$(A \text{ is atomic})\ \frac{\Gamma, A, B \Rightarrow \Delta}{\Gamma, A, A \to B \Rightarrow \Delta}\,\to\text{L}_1 \qquad \frac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \to B}\,\to\text{R}$$

Figure 10.1: Examples of rules in the LJT sequent calculus. The subscript for rule L_1 denotes that it is the first of a set of four rules in the LJT calculus that together cover various possibilities for $\to$ used of the left.

might feel that the top-level constructor for a proof of f is function abstraction, which would be represented as $\lambda x.[]$ where [] denotes a hole to be filled.

The aim of the ML part of the system is to learn an approximation to a distribution $\Pr(\texttt{rule}|f, g, \texttt{path})$, which can then be used to guide a search procedure that systematically attempts to fill holes until none remain. We will not describe the search process in further detail here, but instead focus on the ML part of the system.

The probability distribution is learned using a collection of NNs. As f and g correspond to formulas they are represented as abstract syntax trees with nodes labelled by one of three connectives or a propositional variable. These are in turn one-hot encoded, although variables are encoded as $(n(x), 0, 0, 0)$ where $n(x)$ denotes the number of the variable. One or more convolution layers are then used to combine each node with its parent and children, and the nodes are then aggregated to give a single vector representation $F(f) \in \mathbb{R}^n$. A similar process is used to embed g as a vector $G(g)$. A second NN takes the vector encoding of f and the path `path` and combines them to form a real vector $\texttt{extract}(F(f), \texttt{path})$. This is concatenated with the vector encoding of g and passed to a multilayer feedforward NN with eight outputs corresponding to the possible proof inference rules.

The overall architecture is considerably more successful in finding proofs than the earlier version. While any proof found will be a correct one, the system still suffers from the problem that it may not terminate if no proof exists.

10.2 Methods Employing Sequent Calculus

A different approach to finding proofs in the IPL was taken by Kusumoto *et al.* (2018). Here, the system is based on the *sequent calculus* LJT introduced by Dyckhoff (1992).

A sequent takes the form $\Gamma \Rightarrow \Delta$ where Γ is a multiset of formulas and Δ is a formula. The LJT calculus has eleven rules, each of which has one or more premises, and a single conclusion, and can act on the left or right of a sequent. Rules without premises are axioms. Figure 10.1 shows examples of four of these rules. In order to prove f, we build a tree starting with the sequent $\Rightarrow f$ as root node. If a node matches the conclusion of a rule then

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, B \Rightarrow B}\,\text{Axiom}}{A, A \to B \Rightarrow B}\,{\to}\text{L}_1}{A \wedge (A \to B) \Rightarrow B}\,{\wedge}\text{L}}{\Rightarrow [A \wedge (A \to B)] \to B}\,{\to}\text{R}$$

Figure 10.2: Example of a proof in the LJT sequent calculus.

its children contain the premises. A tree with axioms at all leaves is a proof. Figure 10.2 shows an example proof of the assertion $[A \wedge (A \to B)] \to B$, employing the 4 rules from figure 10.1.

In Kusumoto *et al.* (2018) the search for a proof is approached as a reinforcement learning problem. In attempting to prove f:

1. States s are sets of sequents that remain to be proved. The initial state is $s_0 = \{\Rightarrow f\}$ and the goal is to obtain $s = \{\}$. In general $s = \{p_1, \ldots, p_n\}$ where each p_i is a sequent.

2. An action a taken in state s involves selecting a sequent in s corresponding to the conclusion of some rule, and replacing it with the corresponding premises from that rule.

3. A transition from state s to state s' as a result of an action a in the manner described is denoted $s' = \mathcal{S}(s, a)$.

4. The reward on reaching state s is denoted $\mathcal{R}(s)$ and is

$$\mathcal{R}(s) = \begin{cases} 1 & \text{if } s \text{ is the goal} \\ 0 & \text{otherwise} \end{cases}.$$

   Denoting by r_i the reward received at the ith step, the overall reward for a proof taking n steps is the usual reward introduced in Chapter 3, Equation (3.12)

$$\mathcal{R} = \sum_{i=0}^{n} \gamma^i r_i = \gamma^n$$

   with the convention that $\mathcal{R} = 0$ if no proof is obtained. In general, as each premise in a state $s = \{p_1, \ldots, p_n\}$ must be proved, we have

$$\mathcal{R}(s) = \prod_{i=1}^{n} \mathcal{R}(\{p_i\}).$$

Say we have a policy $\mathcal{P} : s \mapsto a$ that can be used to search for proofs. This will have a corresponding reward function $\mathcal{R}^{\mathcal{P}}(s)$ denoting the reward attained by starting at s and

acting according to policy $\mathcal{P}$. This reward function, if known, can always be used to derive an improved policy, by applying the following:

$$\forall s.\mathcal{P}'(s) = \operatorname*{argmax}_a \mathcal{R}^{\mathcal{P}}(\mathcal{S}(s,a)). \tag{10.1}$$

A GNN is used to represent a parameterized estimate of reward, denoted $\mathcal{R}_{\boldsymbol{\theta}} : s \mapsto \mathbb{R}$, where $\boldsymbol{\theta}$ denotes the network's parameters. From this, a policy $\mathcal{P}_{\boldsymbol{\theta}} : s \mapsto a$ can be obtained using Equation (10.1). The GNN is trained using approximate *policy iteration* (Sutton and Barto, 2018), which involves beginning with some initial policy $\mathcal{P}_{\boldsymbol{\theta}_0}$ and producing a sequence $\mathcal{P}_{\boldsymbol{\theta}_0}, \mathcal{P}_{\boldsymbol{\theta}_1}, \mathcal{P}_{\boldsymbol{\theta}_2} \ldots$ of improved policies. At each step, a training set D_i is selected from the example problems available, with each example being labelled as $(s, \mathcal{R}^{\mathcal{P}_{\boldsymbol{\theta}_i}}(s))$. The GNN is then trained as follows:

$$\boldsymbol{\theta}_{i+1} = \operatorname*{argmin}_{\boldsymbol{\theta}} \sum_{D_i} \left(\mathcal{R}_\theta(s) - \mathcal{R}^{\mathcal{P}_{\boldsymbol{\theta}_i}}(s)\right)^2.$$

10.3 Discussion

IPL has received much less attention as a target for ML than the classical SAT or QSAT problems. This is to be expected: the wealth of problems—many of considerable practical relevance—that can be addressed by translating them to the SAT or QSAT form has no analogue for IPL. IPL is close enough to SAT and QSAT to be of interest; but also enough of a niche subject that opportunities for discussion are limited.

I am not aware of any attempt to date to apply portfolio methods to IPL. This is perhaps surprising, for two reasons. First, there are several theorem provers for IPL available: at the time of writing the *ILTP Library* (Raths *et al.*, 2007) available at www.iltp.de lists seven solvers, and to these can be added *Imogen* (McLaughlin and Pfenning, 2008), fCube (Ferrari *et al.*, 2010) and possibly others. Second, there appears to be little if any reason to suppose that the application of portfolio methods to these solvers would require much beyond the straightforward extension of existing approaches.

The Curry-Howard correspondence has provided opportunities for the application of seq2seq learners and other methods that are in striking contrast to the other work we have seen. Whether this was a unique opportunity, or one that can be exploited further remains an open question.

The approach to IPL based on using RL to guide search within the sequent calculus is more familiar. While we have seen numerous applications of RL in earlier Chapters, it has not to date been applied in the SAT/QSAT domain to structured proof methods of this kind. Recent work exists applying ML to proofs based on the *connection calculus* in first-order logic (Färber *et al.*, 2021), so there is perhaps room for further work here.

More generally, it is interesting to wonder whether the success of local search for SAT, or of the many methods developed allowing ML to be applied to CDCL solvers, can be

repeated by adapting them in some way to the IPL context. The same question was raised earlier in the case of QSAT solvers, and the conclusion is the same. Methods for the solution of IPL problems often seem sufficiently removed from those used for SAT that researchers might be better advised to target the specifics of IPL solvers, rather than to attempt to re-apply ML approaches from SAT.

11 Conclusion

In writing this work I have been struck by the extraordinary variety of approaches that have been taken to mixing ML with ATP. In retrospect this should not have been so surprising, as the problems being addressed are rich, and the available methods both powerful and diverse. I am now equally struck by the extent to which there are opportunities to take the work further, and I hope that my original aims have been satisfied to the extent that the reader might feel equally inspired.

My main conclusions have been set out in the Discussion sections of the preceding Chapters, and do not need to be repeated. I will therefore conclude with some more general observations that did not fit naturally into that structure.

11.1 The Structure of Solvers

Thus far, theorem-provers have not been built from the outset with learning in mind—learning has been retro-fitted. This is not necessarily good. As noted by Samulowitz and Memisevic (2007), the use of a portfolio approach limits behaviour to that provided by existing solvers. In the case of CDCL solvers, the fact that they have been so forcefully engineered with performance in mind makes modifications possible, but somewhat uncomfortable, in the sense that one has to tread very carefully to avoid destroying the performance of the solver. It is perhaps worth considering whether re-engineering SAT and QSAT solvers with such modifications in mind might be a profitable exercise.

In Section 7.7.4 we described the SATenstein solver, which deliberately exposed a large number of parameters, allowing great flexibility in modifying solver behaviour. While the aim in that work was to allow those parameters to be optimized automatically, it represents one way in which future solvers might be made more accessible to ML researchers. (While many existing solvers are parameterized, the extent of the parameterization is often limited in comparison.) In Section 7.8 we described a system that learns at the source code level. Additionally, in the context of constraint satisfaction problems, Minton (1996) describes the MULTI-TAC system, which produces solver programs optimized for specific classes of problem. Such systems represent another way of providing great flexibility in solver operation, but it is again not clear whether this flexibility can be harnessed by other ML researchers in a

straightforward way.

11.2 What is the Appropriate Level of Complexity?

The worlds of ML and ATP currently possess a relatively new toolbox in the form of the large set of methods collectively entitled 'deep learning'. It is unsurprising that these new tools are being applied freely.

In scientific terms, there is clear evidence that large-scale learning of this kind can improve decision-making in SAT and QSAT solvers. This is an important lesson, and it is likely that ongoing developments will mitigate the negative effects that sometimes accompany it: in particular, the overall slow-down imposed by the increased time taken to make the improved decisions.

For now, where deep learning has made improvements in both numbers of problems solved *and* speed, and when this has been achieved with realistic data, it has been through being *combined* with existing methods.

Of course, heavyweight tools are sometimes the right ones. But not for brain surgery; and modifying the inner workings of a SAT solver is certainly the computer scientist's answer to brain surgery. It is important to maintain a balanced view, as much of the success in applying ML to SAT and QSAT has been achieved using lightweight (in comparison with deep learning) methods. And again, there appears to be merit in continuing to develop both.

It may be that deep learning turns out to be a universal tool in this field. But the jury is at present still deliberating.

11.3 What About Parallel Solvers?

Work on incorporating ML into SAT solvers has almost entirely targeted sequential approaches; so for example portfolio solvers select a single solver from the portfolio to run on a single processor. This is rather surprising in the light of the extensive literature on portfolio design in general, addressing the running of multiple algorithms in parallel, switching between algorithms on a single processor, or treating portfolio construction as a form of restarting (Huberman *et al.*, 1997; Gomes and Selman, 2001).

More recent work, as reviewed by Holldobler *et al.* (2011) has turned to SAT solvers exploiting parallel architectures, and the question arises of how learning methods might be extended to these more recent solvers. Using again the example of portfolio solvers, we might for example try to identify the best subset of solvers to run in parallel on the available resources. In particular it has become clear that to attain the best performance parallel solvers must be designed specifically to exploit the properties of the underlying hardware,

and it is of interest to ask whether ML has a further contribution to make in this context.

11.4 Solver Competitions

The field of automated theorem proving places great importance on competitions. This has driven huge innovation and progress. At present the standard metrics used to judge provers are solution time and number of theorems proved within some time limit. Such metrics have an influence on the way in which solvers are developed, and as proving more theorems more quickly is a perfectly reasonable target, these metrics, and competitions themselves, are useful tools.

Historically, the use of predominantly time-based metrics has allowed advances to be made through careful and ingenious programming. Given the well known computational complexity of theorem proving problems, even in the simplest SAT variant, we might only expect this approach to provide limited benefit in the long-term, regardless of its undoubted merits. Indeed, it seems that this approach has in recent years been less influential, and that advances have been dominated by more foundational developments. Again, a positive outcome based on time-based metrics as a driver for competitive solvers.

Might it be productive however to consider additional metrics? In writing this work it has become clear that ML methods can be used to make better decisions during a proof attempt, although often at the cost of increased runtime. The fact that decisions of better quality can be made is surely an indication that fundamental advances are being made, even if not yet in a context that allows them to be fully exploited. It seems therefore important that researchers devote time to developing methods that, while perhaps not winning competitions in the short-term—and under the current metrics—have the potential to allow us to address harder problems in the future. This might be facilitated by adding alternative ranking methods within existing competitions. For example, by rewarding solvers finding solutions using *fewer decisions* separately to solvers that solve problems *faster*.

Acknowledgements

In 2016 Josef Urban invited me to speak at the *1st Conference on Artificial Intelligence and Theorem Proving (AITP)*. I offered to give a survey talk on applications of machine learning to automated theorem provers. Having given the talk it seemed like a good idea to write it up in full.

I thought this would be a straightforward process, but it did not take long to discover that the full extent of the literature on the subject is genuinely impressive. In any case, here is the result.

Thanks for the invitation Josef.

I've done my share of reviewing, and I'm aware that reviewing a work of this length is a major undertaking. I therefore offer great thanks to the anonymous reviewer for their careful reading and numerous useful suggestions.

In reading the literature underlying this work there were inevitably occasions where I felt the need to contact the original authors for clarification. All responded quickly and helpfully. Thanks to all of them.

A Abbreviations

Abbreviation	Meaning
ATP	Automated theorem-prover
CAL	Clauses active list
CDCL	Conflict-Driven Clause Learning
CHB	Conflict history-based
CIG	Clause incidence graph
CNF	Conjunctive normal form
CNN	Convolutional neural network
CSP	Constraint satisfaction problem
CVIG	Clause-variable incidence graph
DAG	Directed acyclic graph
DPLL	Davis, Putnam, Logemann, Loveland
DRAT	Deletion Resolution Asymmetric Tautology
EHM	Empirical hardness model
EP	Evolutionary programming
ES	Evolutionary strategy

Continued overleaf...

Abbreviation	Meaning
ERWA	Exponential recency weighted average
EVSIDS	Exponential VSIDS
GA	Genetic algorithm
GLR	Global learning rate
GNN	Graph neural network
GP	Genetic program
IPL	Intuitionistic propositional logic
LBD	Literals blocks distance
LRB	Learning rate branching
LSTM	Long short-term memory
MAB	Multi-armed bandit
ML	Machine learning
MLB	Machine learning-based restart
MLP	Multi-layer perceptron
MPNN	Message-passing neural network
NN	Neural network
QSAT	Quantified satisfiability
RL	Reinforcement learning
SAT	Satisfiability
SVM	Support vector machine
SGDB	Stochastic Gradient Descent Branching
UC	Unsatisfiable core
UCB	Upper confidence bound
UIP	Unique implication point
VIG	Variable incidence graph
VSIDS	Variable State Independent Decaying Sum

B Symbols

General	
$\mathbf{I}$	Identity matrix
$\mathbb{R}^i$	Set of i-dimensional vectors with real elements
v_i	Element i of a vector $\mathbf{v}$
$\mathbb{R}^{i \times j}$	Set of i by j matrices with real elements
$M_{i,j}$	Element at row i, column j of a matrix $\mathbf{M}$
$\mathbb{I}$	Indicator function: $\mathbb{I}[P]$ is 1 if P is true and 0 otherwise
$\mathbf{1}_{ij}$	i by j matrix with all elements equal to 1.
$N(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Multivariate normal density with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$
$\otimes$	Element-by-element multiplication of vectors
$[n]$	The set $\{1, \ldots, n\}$

Continued overleaf...

The SAT Problem	
V	Set of variables
C	Set of clauses
v	A variable. or |V|, according to context
c	A clause, or |C|, according to context
l	A literal
f, ϕ, ψ	Propositional formulas
A	Assignment
$a(v)$	Activity of a variable
$a(c)$	Activity of a clause
Machine Learning	
n	Dimension of feature space for a classifier
m	Size of training set
$\mathbf{s}$	Sequence containing m training examples
F	Function mapping instances of a problem to feature vectors
$\mathbb{A}$	Learning algorithm
$\mathcal{H}$	Hypothesis space
Z	Constant normalizing a probability distribution
C	Random variable denoting a class
$\mathbf{x}$	Instance vector
k	Dimension of the extended space
p	Number of basis functions
ϕ_i	Basis functions
λ	Regularization parameter
$\boldsymbol{\phi}(\mathbf{x})$	Mapping from instance $\mathbf{x}$ to the extended space
$\boldsymbol{\Phi}$	Matrix of $\boldsymbol{\phi}(\mathbf{x})$ for $\mathbf{x}$ in a training sequence
$\sigma(x)$	Step or sigmoid function
$\boldsymbol{\theta}$, $\mathbf{w}$	Vectors of parameters
K	Number of clusters

Continued overleaf...

Machine Learning	
r_i	Reward sequence
$r_{i,t}$	Reward from arm i of a multi-armed bandit at time t
α	EWMA discounting factor
γ	Bandit or reinforcement learning discount factor
$\hat{r}_T$	Estimated bandit reward at time T
$\mathcal{S}$	RL state set
$\mathcal{A}$	RL action set
p	RL policy
R	RL discounted reward
μ	Step size for gradient descent
c	Number of classes in a problem
$\mathbf{K}$	CNN kernel
t	Step in a sequence
T	Final step in a sequence
O	Objective function

References

Aksoy, L. and E. O. Gunes. 2005. "An Evolutionary Local Search Algorithm for the Satisfiability Problem". In: *Proceedings of the 14th Turkish Symposium on Artificial Intelligence and Neural Networks (TAINN)*. Ed. by F. A. Savakı. Vol. 3949. *Lecture Notes in Computer Science*. Springer. 185–193.

Allan, J. A. and S. Minton. 1996. "Selecting the Right Heuristic Algorithm: Runtime Performance Predictors". In: *Advances in Artificial Intelligence: 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*. Ed. by G. McCalla. Vol. 1081. *Lecture Notes in Computer Science*. Springer. 41–53.

Amadini, R., M. Gabbrielli, and J. Mauro. 2015. "A Multicore Tool for Constraint Solving". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*. Ed. by Q. Yang and M. Wooldridge. AAAI Press/International Joint Conferences on Artificial Intelligence. 232–238.

Amizadeh, S., S. Matusevych, and M. Weimer. 2019. "Learning To Solve Circuit-SAT: An Unsupervised Differentiable Approach". In: *Proceedings of the International Conference on Learning Representations*.

Ansótegui, C., M. L. Bonet, J. Giráldez-Cru, and J. Levy. 2014. "The Fractal Dimension of SAT Formulas". In: *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by S. Demri, D. Kapur, and C. Weidenbach. Vol. 8562. *Lecture Notes in Computer Science*. Springer. 107–121.

Ansótegui, C., M. L. Bonet, J. Giráldez-Cru, and J. Levy. 2017. "Structure instances for SAT instances classification". *Journal of Applied Logic*. 23(Sept.): 27–39.

Ansótegui, C., J. Giráldez-Cru, and J. Levy. 2012. "The Community Structure of SAT Formulas". In: *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by A. Cimatti and R. Sebastiani. Vol. 7317. *Lecture Notes in Computer Science*. Springer. 410–423.

Ansótegui, C., M. Sellmann, and K. Tierney. 2009. "A Gender-Based Genetic Algorithm for the Automatic Configuration of Algorithms". In: *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP)*. Ed. by I. P. Gent. Vol. 5732. *Lecture Notes in Computer Science*. Springer. 142–157.

Anthony, M. and P. L. Bartlett. 2009. *Pattern Recognition and Machine Learning*. Cambridge University Press.

Audemard, G. and L. Simon. 2018. "On the Glucose SAT Solver". *International Journal on Artificial Intelligence Tools*. 27(1).

Audemard, G. and L. Simon. 2009. "Predicting learnt clauses quality in modern SAT solvers". In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann. 399–404.

Auer, P., N. Cesa-Bianci, Y. Freund, and R. E. Schapire. 1995. "Gambling in a rigged casino: The adversarial multi-armed bandit problem". In: *Proceedings of the 36th IEEE Annual Symposium on Foundations of Computer Science*. IEEE. 332–331.

Ba, J. L., J. R. Kiros, and G. E. Hinton. 2016. "Layer Normalization". arXiv:1607.06450v1.

Babić, D. and A. J. Hu. 2007. "Structural Abstraction of Software Verification Conditions". In: *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*. Ed. by W. Damm and H. Hermanns. *Lecture Notes in Computer Science*. No. 4590. Springer. 366–378.

Bader-El-Den, M. and R. Poli. 2007. "Generating SAT Local Search Heuristics Using a GP Hyper-Heuristic Framework". In: *Proceedings of the 8th International Conference on Artificial Evolution*. Ed. by N. Monmarché, E.-G. Talbi, P. Collet, M. Schoenauer, and E. Lutton. Vol. 4926. *Lecture Notes in Computer Science*. Springer. 37–49.

Bader-El-Den, M. and R. Poli. 2008a. "Analysis and extension of the Inc* on the satisfiability testing problem". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. IEEE. 3342–3349.

Bader-El-Den, M. and R. Poli. 2008b. "Evolving Effective Incremental Solvers for SAT with a Hyper-Heuristic Framework Based on Genetic Programming". In: *Genetic Programming Theory and Practice VI*. Ed. by B. Worzel, T. Soule, and R. Riolo. *Genetic and Evolutionary Computation*. Springer. 1–16.

Bader-El-Den, M. and R. Poli. 2008c. "Inc*: An Incremental Approach for Improving Local Search Heuristics". In: *Proceedings of the 8th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP)*. Ed. by J. van Hemert and C. Cotta. Vol. 4972. *Lecture Notes in Computer Science*. Springer. 194–205.

Bain, S., J. Thornton, and A. Sattar. 2005a. "A Comparison of Evolutionary Methods for the Discovery of Local Search Heuristics". In: *Proceedings of the 18th Australasian Joint Conference on Artificial Intelligence*. Ed. by S. Zhang and R. Jarvis. Vol. 3809. *Lecture Notes in Computer Science*. Springer. 1068–1074.

Bain, S., J. Thornton, and A. Sattar. 2005b. "Evolving Variable-Ordering Heuristics for Constrained Optimisation". In: *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*. Ed. by P. van Beek. Vol. 3709. *Lecture Notes in Computer Science*. Springer. 732–736.

Bertels, A. R. and D. R. Tauritz. 2016. "Why Asynchronous Parallel Evolution is the Future of Hyper-heuristics: A CDCL SAT Solver Case Study". In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. Ed. by T. Friedrich. Association for Computing Machinary. 1359–1365.

Bertels, A. R. 2016. "Automated design of boolean satisfiability solvers employing evolutionary computation". 7549. *MA thesis*. Missouri Institute of Science and Technology.

Biere, A. 2008a. "Adaptive Restart Strategies for Conflict Driven SAT Solvers". In: *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by H. K. Büning and X. Zhao. Vol. 4996. *Lecture Notes in Computer Science*. Springer. 28–33.

Biere, A. 2008b. "PicoSAT Essentials". *Journal of Satisfiability, Boolean Modeling and Computation*. (75–97).

Biere, A., A. Cimatti, E. Clarke, and Y. Zhu. 1999. "Symbolic Model Checking without BDDs". In: *Proceedings of the 5th International Conference on Construction and Analysis of Systems (TACAS)*. Ed. by W. R. Cleaveland. Vol. 1579. *Lecture Notes in Computer Science*. Springer. 197–207.

Biere, A., K. Fazekas, M. Fleury, and M. Heisinger. 2020. "CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020". In: *Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions*. Ed. by T. Balyo, N. Froleyks, M. J. Heule, M. Iser, M. Järvisalo, and M. Suda. *Department of Computer Science Report Series B*. No. B-2020-1. Department of Computer Science, University of Helsinki.

Biere, A. and A. Fröhlich. 2015. "Evaluating CDCL Variable Scoring Schemes". In: *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by M. Heule and S. Weaver. Vol. 9340. *Lecture Notes in Computer Science*. Springer. 405–422.

Biere, A. and A. Fröhlich. 2019. "Evaluating CDCL Restart Schemes". In: *Proceedings of Pragmatics of SAT 2015 and 2018*. Vol. 59. *EPiC Series in Computing*. EasyChair. 1–17.

Biere, A., M. Huele, H. van Maaren, and T. Walsh. 2009. *Handbook of Satisfiability*. Vol. 85. *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Bischl, B., M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, D. Deng, and M. Lindauer. 2021. "Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges". arxiv:2107.05847v2 [stat.ML].

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning*. Springer.

Blanchette, J. C., M. Fleury, P. Lammich, and C. Weidenbach. 2018. "A Verified SAT Solver Framework with Learn, Forget, Restart and Incrementality". *Journal of Automated Reasoning*. 61(1–5): 333–365.

Blondel, V. D., J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. 2008. "Fast unfolding of communities in large networks". *Journal of Statistical Mechanics: Theory and Experiment*. 2008(10).

Boolos, G. S., J. P. Burgess, and R. C. Jeffrey. 2007. *Computability and Logic*. 5th Edition. Cambridge University Press.

Boyan, J. A. 1998. “Learning Evaluation Functions for Global Optimization”. CMU-CS-98-152. *PhD thesis*. Pittsburgh, PA 15213: Carnegie Mellon University.

Boyan, J. A. and A. W. Moore. 1998. “Learning Evaluations Functions for Global Optimization and Boolean Satisfiability”. In: *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*. 3–10.

Boyan, J. A. and A. W. Moore. 2000. “Learning Evaluation Functions to Improve Optimization by Local Search”. *Journal of Machine Learning Research*. 1: 77–112.

Breiman, L. 2001. “Random Forests”. *Machine Learning*. 45(1): 5–32.

Bridge, J. P., S. B. Holden, and L. C. Paulson. 2014. “Machine Learning for First-Order Theorem Proving: Learning to Select a Good Heuristic”. *Journal of Automated Reasoning*. 53(Feb.): 141–172.

Bünz, B. and M. Lamm. 2017. “Graph Neural Networks and Boolean Satisfiability”. arXiv:1702.03592 [cs.AI].

Cameron, C., R. Chen, J. Hartford, and K. Leyton-Brown. 2020. “Predicting Propositional Satisfiabilty via End-to-End Learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI-20)*. Vol. 34. No. 4. AAAI Press.

Cameron, C., H. H. Hoos, K. Leyton-Brown, and F. Hutter. 2017. “OASC-2017: *Zilla Submission”. In: *Proceedings of Machine Learning Research*. Ed. by M. Lindauer, J. N. van Rijn, and L. Kotthoff. Vol. 79. 15–18.

Carvalho, E. and J. Marques-Silva. 2004. “Using Rewarding Mechanisms for Improving Branching Heuristics”. In: *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*.

Chang, W., G. Wu, and Y. Xu. 2017. “Adding a LBD-based Rewarding Mechanism in Branching Heuristic for SAT Solvers”. In: *Proceedings of the 12th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*.

Chang, W., Y. Xu, and S. Chen. 2018. “A New Rewarding Mechanism for Branching Heuristic in SAT Solvers”. *International Journal of Computational Intelligence Systems*. 12(1): 334–341.

Chen, W., A. Howe, and D. Whitley. 2014. “MiniSAT with Classification-based Preprocessing”. In: *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*. Ed. by A. Belov, D. Diepold, M. J. Heule, and M. Järvisalo. *Department of Computer Science Report Series B*. No. B-2014-2. Department of Computer Science, University of Helsinki. 41–42.

Chen, Z. and Z. Yang. 2019. “Graph Neural Reasoning May Fail in Certifying Boolean Unsatisfiability”. arXiv:1909.11588 [cs.LG].

Chu, G., A. Harwood, and P. J. Stuckey. 2010. “Cache Conscious Data Structures for Boolean Satisfiability Solvers”. *Journal of Satisfiability, Boolean Modeling and Computation*. 6(1-3): 99–120.

Chvalovský, K. 2019. “Top-Down Neural Model For Formulae”. In: *Proceedings of the International Conference on Learning Representations*.

Clarke, E., A. Biere, R. Raimi, and Y. Zhu. 2001. “Bounded Model Checking Using Satisfiability Solving”. *Formal Methods in System Design*. 19: 7–34.

Clauset, A., C. R. Shalizi, and M. E. J. Newman. 2009. “Power-Law Distributions in Empirical Data”. *SIAM Review*. 51(4): 661–703.

Dalen, D. van. 2001. “Intuitionistic Logic”. In: *The Blackwell Guide to Philosophical Logic*. Ed. by L. Goble. Blackwell Publishers. Chap. 11. 224–257.

Daumé, H. and D. Marcu. 2005. “Learning as Search Optimization: Approximate Large Margin Methods for Structured Prediction”. In: *Proceedings of the 22nd International Conference on Machine Learning (ICML)*. 169–176.

Davies, M., G. Logemann, and D. Loveland. 1962. “A machine program for theorem-proving”. *Communications of the ACM*. 5(7): 394–397.

Dershowitz, N., Z. Hanna, and J. Katz. 2005. “Bounded Model Checking with QBF”. In: *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*. Ed. by F. Bacchus and T. Walsh. Vol. 3569. *Lecture Notes in Computer Science*. 408–414.

Devlin, D. and B. O’Sullivan. 2008. “Satisfiability as a Classification Problem”. In: *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*. Ed. by D. Bridge, K. Brown, B. O’Sullivan, and H. Sorensen. University College Cork.

Devroye, L., L. Györfi, and G. Lugosi. 1996. *A Probabilistic Theory of Pattern Recognition*. Vol. 31. *Stochastic Modelling and Applied Probability*. Springer.

Duda, R. O., P. E. Hart, and D. G. Stork. 2000. *Pattern Classification*. 2nd Edition. Wiley.

Dyckhoff, R. 1992. “Contraction-Free Sequent Calculi for Intuitionistic Logic”. *The Journal of Symbolic Logic*. 57(3): 795–807.

Eén, N. and A. Biere. 2005. “Effective Preprocessing in SAT Through Variable and Clause Elimination”. In: *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by F. Bacchus and T. Walsh. Vol. 3569. *Lecture Notes in Computer Science*. Springer. 61–75.

Eén, N. and N. Sörensson. 2003. “An Extensible SAT-solver”. In: *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by E. Giunchiglia and A. Tacchella. Vol. 2919. *Lecture Notes in Computer Science*. Springer. 502–518.

Egly, U., T. Eiter, H. Tompits, and S. Woltran. 2000. “Solving Advanced Reasoning Tasks using Quantified Boolean Formulas”. In: *Proceedings of the Seventeenth National Conference on Artificial Intelligence*. The AAAI Press. 417–422.

Emmerich, M., O. M. Shir, and H. Wang. 2018. “Evolution Strategies”. In: *Handbook of Heuristics*. Ed. by R. Marti, P. Panos, and M. G. C. Resende. Springer. 1–31.

Engel, A. and C. V. den Broeck. 2001. *Statistical Mechanics of Learning*. Cambridge University Press.

Ertel, W., J. M. P. Schumann, and C. B. Suttner. 1989. "Learning Heuristics for a Theorem Prover using Back Propagation". In: *5. Osterreichische Aertificial-Intelligence-Tagung*. Ed. by J. Retti and K. Leidlmair. Vol. 208. *Informatik-Fachbericht*. 87–95.

Evans, R., D. Saxton, D. Amos, P. Kohli, and E. Grefenstette. 2018. "Can Neural Networks Understand Logical Entailment?" In: *Proceedings of the 6th International Conference on Learning Representations*.

Färber, M., C. Kaliszyk, and J. Urban. 2021. "Machine Learning Guidance for Connection Tableaux". *Journal of Automated Reasoning*. 65: 287–320.

Fernández-Delgado, M., E. Cernadas, S. Barro, and D. Amorim. 2014. "Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?" *Journal of Machine Learning Research*. 15: 3133–3181.

Ferrari, M., C. Fiorentini, and G. Fiorino. 2010. "fCube: An Efficient Prover for Intuitionistic Propositional Logic". In: *Proceedings of the 17th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*. Ed. by C. G. Fermüller and A. Voronkov. Vol. 6397. *Lecture Notes in Computer Science*. Springer. 294–301.

Fink, M. 2007. "Online Learning of Search Heuristics". *Proceedings of Machine Learning Research*. 2: 115–122.

Fleury, M., J. C. Blanchette, and P. Lammich. 2018. "A verified SAT solver with watched literals using imperative HOL". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proof*. 158–171.

Flint, A. and M. B. Blaschko. 2012. "Perceptron Learning of SAT". In: *Proceedings of the 25th International Conference on Neural Information Processing (NIPS)*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Vol. 2. Curran Associates Inc. 2771–2779.

Fréchette, A., N. Newman, and K. Leyton-Brown. 2016. "Solving the Station Repacking Problem". In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. 702–709.

Fuchs, M. and M. Fuchs. 1998. "Feature-based learning of search-guiding heuristics for theorem proving". *AI Communications*. 11(3,4): 175–189.

Fukunaga, A. S. 2002. "Automated Discovery of Composite SAT Variable-Selection Heuristics". In: *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*. The AAAI Press. 641–648.

Fukunaga, A. S. 2004. "Evolving Local Search Heuristics for SAT Using Genetic Programming". In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*. Ed. by K. Deb. Vol. 3103. *Lecture Notes in Computer Science*. Springer. 483–494.

Fukunaga, A. S. 2008. "Automated Discovery of Local Search Heuristics for Satisfiability Testing". *Evolutionary Computation*. 16(1): 31–61.

Fukunaga, A. S. 2009. "Massively Parallel Evolution of SAT Heuristics". In: *Proceedings of the IEEE Congress on Evolutionary Computation*. 1478–1485.

Gagliolo, M. and J. Schmidhuber. 2007. "Learning Restart Strategies". In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*. 792–797.

Garey, M. R. and D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

Garivier, A. and E. Moulines. 2011. "On Upper-Confidence Bound Policies for Switching Bandit Problems". In: *Proceedings of the 22nd International Conference on Algorithmic Learning Theory*. Ed. by J. Kivinen, C. Szepesvári, E. Ukkonen, and T. Zeugmann. Vol. 6925. *Lecture Notes in Computer Science*. Springer. 174–188.

Gilmer, J., S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. 2017. "Neural message passing for Quantum chemistry". In: *Proceedings of the 34th International Conference on Machine Learning (ICML)*. Vol. 70. 1263–1272.

Giunchiglia, E., M. Narizzano, L. Pulina, and A. Tacchella. 2005. "Quantified Boolean Formulas satisfiability library (QBFLIB)". URL: www.qbflib.org.

Giunchiglia, E., M. Narizzano, and A. Tacchella. 2006. "Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas". *Journal of Artificial Intelligence Research*. 26(Aug.): 371–416.

Goldberg, E. and Y. Novikov. 2002. "BerkMin: A fast and robust SAT solver". In: *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition*. IEEE. 142–149.

Gomes, C. P. and B. Selman. 2001. "Algorithm portfolios". *Artificial Intelligence*. 126: 43–62.

Goodfellow, I., Y. Bengio, and A. Courville. 2016. *Deep Learning*. MIT Press.

Gottlieb, J., E. Marchiori, and C. Rossi. 2002. "Evolutionary Algorithms for the Satisfiability Problem". *Evolutionary Computation*. 10(1): 35–50.

Graça, A., J. Marques-Silva, and I. Lynce. 2010. "Haplotype Inference Using Propositional Satisfiability". In: *Mathematical Approaches to Polymer Sequence Analysis and Related Problems*. Ed. by R. Bruni. Springer. 127–147.

Grozea, C. and M. Popescu. 2014. "Can Machine Learning Learn a Decision Oracle for NP Problems? A Test on SAT". *Fundamenta Informaticae*. 131: 441–450.

Guyon, I., S. Gunn, M. Nikravesh, and L. A. Zadeh, eds. 2006. *Feature Extraction: Foundations and Applications. Studies in Fuzziness and Soft Computing*. Springer.

Haim, S. and T. Walsh. 2008. "Online Estimation of SAT Solving Runtime". In: *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by H. K. Büning and X. Zhao. Vol. 4996. *Lecture Notes in Computer Science*. Springer. 133–138.

Haim, S. and T. Walsh. 2009. "Restart Strategy Selection Using Machine Learning Techniques". In: *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by O. Kullmann. Vol. 5584. *Lecture Notes in Computer Science*. Springer. 312–325.

Hamadi, Y., S. Jabbour, and L. Saïs. 2010. "Learning for Dynamic Subsumption". *International Journal on Artificial Intelligence Tools: Architectures, Languages, Algorithms*. 19(4): 511–529.

Hamilton, W. L. 2020. "Graph Representation Learning". *Synthesis Lectures on Artificial Intelligence and Machine Learning*. 14(3): 1–159.

Han, H. and F. Somenzi. 2009. "On-The-Fly Clause Improvement". In: *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by O. Kullmann. Vol. 5584. *Lecture Notes in Computer Science*. Springer. 209–222.

Han, J. M. 2020a. "Enhancing SAT solvers with glue variable predictions". arXiv:2007.02559v1 [cs.LO].

Han, J. M. 2020b. "Learning cubing heuristics for SAT from DRAT proofs". In: *Conference on Artificial Intelligence and Theorem Proving (AITP)*.

Harrison, J. 2009. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press.

Hartford, J., D. Graham, K. Leyton-Brown, and S. Ravanbakhsh. 2018. "Deep Models of Interactions Across Sets". In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. *Proceedings of Machine Learning Research*. 1909–1918.

Hastie, T., R. Tibshirani, and J. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd Edition. Springer.

Heule, M., M. Järvisalo, and M. Suda. 2019. *The international SAT Competitions web page*. URL: http://www.satcompetition.org/.

Heule, M. J. H., O. Kullmann, and V. W. Marek. 2016. "Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-And-Conquer". In: *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by N. Creignou and D. L. Berre. Vol. 9710. *Lecture Notes in Computer Science*. Springer. 228–245.

Heule, M. J. H., O. Kullmann, S. Wieringa, and A. Biere. 2011. "Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads". In: *Proceedings of the 7th International Haifa Verification Conference*. Ed. by K. Eder, J. Lourenço, and O. Shehory. Vol. 7261. *Lecture Notes in Computer Science*. Springer. 50–65.

Heule, M. J., O. Kullmann, and V. W. Marek. 2017. "Solving Very Hard Problems: Cube-and-Conquer, a Hybrid SAT Solving Method". In: *Proceedings of the 26th International Conference on Artificial Intelligence (IJCAI)*. Ed. by C. Sierra. 4864–4868.

Hochreiter, S. and J. Schmidhuber. 1997. "Long Short-Term Memory". *Neural Computation*. 9(8): 1735–1780.

Holldobler, S., N. Manthey, V. H. Nguyen, J. Stecklina, and P. Steinke. 2011. "A short overview of modern parallel SAT-solvers". In: *Proceedings of the International Conference on Computer Science and Information Systems*. IEEE. 201–206.

Holte, R. C. 1993. "Very Simple Classification Rules Perform Well on Most Commonly Used Datasets". *Machine Learning*. 11: 63–91.

Hoos, H., T. Peitl, F. Slivovsky, and S. Szeider. 2018. "Portfolio-Based Algorithm Selection for Circuit QBFs". In: *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP)*. Ed. by J. Hooker. Vol. 11008. *Lecture Notes in Computer Science*. Springer. 195–209.

Hoos, H. H. 1999. "On the Run-Time Behavious of Stochastic Local Search Algorithms for SAT". In: *Proceedings of the 16th National Conference on Artificial Intelligence*. Association for the Advancement of Artificial Intelligence. AAAI Press. 661–666.

Hoos, H. H. and T. Stützle. 2000. "SATLIB: An Online Resource for Research on SAT". In: *SAT2000: Highlights of Satisfiability Research in the Year 2000*. Ed. by I. P. Gent, H. V. Maaren, and T. Walsh. Vol. 63. *Frontiers in Artificial Intelligence and Applications*. IOS Press. 283–292.

Hoos, H. H. and T. Stützle. 2019. *SATLIB—The Satisfiability Library*. URL: https://www.cs.ubc.ca/~hoos/SATLIB/.

Hopfield, J. J. and D. W. Tank. 1985. "'Neural' Computation of Decisions in Optimization Problems". *Biological Cybernetics*. 52(July): 141–152.

Hu, Y., X. Si, C. Hu, and J. Zhang. 2019. "A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures". *Neural Computation*. 31: 1235–1270.

Huberman, B. A., R. M. Lukose, and T. Hogg. 1997. "An Economics Approach to Hard Computational Problems". *Science*. 275(Jan.): 51–54.

Hutter, F., D. Babić, H. H. Hoos, and A. J. Hu. 2007. "Boosting Verification by Automatic Tuning of Decision Procedures". In: *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design*. IEEE. 27–34.

Hutter, F., Y. Hamadi, H. H. Hoos, and K. Leyton-Brown. 2006. "Performance Prediction and Automated Tuning of Randomized and Parametric Algorithms". In: *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*. Ed. by F. Benhamou. Vol. 4204. *Lecture Notes in Computer Science*. Springer. 213–228.

Hutter, F., H. H. Hoos, and K. Leyton-Brown. 2011. "Sequential Model-Based Optimization for General Algorithm Configuration". In: *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION)*. Ed. by C. A. C. Coello. Vol. 6683. *Lecture Notes in Computer Science*. Springer. 507–523.

Hutter, F., H. H. Hoos, K. Leyton-Brown, and T. Stützle. 2009. "ParamILS: An Automatic Algorithm Configuration Framework". *Journal of Artificial Intelligence Research*. 36(Oct.): 267–306.

Hutter, F., L. Kotthoff, and J. Vanschoren, eds. 2019. *Automated Machine Learning: Methods, Systems, Challenges. The Springer Series on Challenges in Machine Learning*. Springer.

Hutter, F., M. Lindauer, A. Balint, S. Bayless, H. Hoos, and K. Leyton-Brown. 2017. "The Configurable SAT Solver Challenge (CSSC)". *Artificial Intelligence*. 243: 1–25.

Hutter, F., D. A. D. Tompkins, and H. H. Hoos. 2002. "Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT". In: *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*. Ed. by P. V. Hentenryck. Vol. 2470. *Lecture Notes in Computer Science*. Springer. 233–248.

Illetskova, M., A. R. Bertels, J. M. Tuggle, A. Harter, S. Richter, D. R. Tauritz, S. Mulder, D. Bueno, M. Leger, and W. M. Siever. 2017. "Improving performance of CDCL SAT solvers by automated design of variable selection heuristics". In: *Proceedings of the IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE. 617–624.

Janota, M. 2018. "Towards Generalization in QBF Solving via Machine Learning". In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*. AAAI Press. 6607–6614.

Janota, M., W. Klieber, J. Marques-Silva, and E. Clarke. 2016. "Solving QBF with counterexample guided refinement". *Artificial Intelligence*. 234: 1–25.

Järvisalo, M., M. J. H. Heule, and A. Biere. 2012. "Inprocessing Rules". In: *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by B. Gramlich, D. Miller, and U. Sattler. Vol. 7364. *Lecture Notes in Computer Science*. Springer. 355–370.

Jaszczur, S., M. Łuszczyk, and H. Michalewski. 2019. "Neural heuristics for SAT solving". In: *Proceedings of the 7th International Conference on Learning Representations*.

Jeroslow, R. G. and J. Wang. 1990. "Solving propositional satisfiability problems". *Annals of Mathematics and Artificial Intelligence*. 1(1–4): 167–187.

Johnson, J. L. 1989. "A Neural Network Approach to the 3-Satisfiability Problem". *Journal of Parallel and Distributed Computing*. 6: 435–449.

Jordan, C. and Ł. Kaiser. 2013. "Experiments with Reduction Finding". In: *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Ed. by M. Järvisalo and A. V. Gelder. Vol. 7962. *Lecture Notes in Computer Science*. Springer. 192–207.

Kadioglu, S., Y. Malitski, A. Sabharwal, H. Samulowitz, and M. Sellmann. 2011. "Algorithm Selection and Scheduling". In: *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*. Ed. by J. Lee. Vol. 6876. *Lecture Notes in Computer Science*. Springer. 454–469.

Kadioglu, S., Y. Malitsky, M. Sellman, and K. Tierney. 2010. "ISAC—Instance-Specific Algorithm Configuration". In: *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI)*. 751–756.

Kaplan, E. L. and P. Meier. 1958. “Nonparametric Estimation from Incomplete Observations”. *Journal of the American Statistical Association*. 53(282): 457–481.

Kaufman, L. and P. J. Rousseeuw. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis. Wiley Series in Probability and Statistics*. John Wiley & Sons, Inc.

Kautz, H. and B. Selman. 1992. “Planning as Satisfiability”. In: *Proceedings of the 10th European Conference on Articifial Intelligence (ECAI)*. Wiley. 359–363.

Kautz, H. A. 2006. “Deconstructing Planning as Satisfiability”. In: *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*. Vol. 2. 1524–1526.

Khudabukhsh, A. R., L. Xu, H. H. Hoos, and K. Leyton-Brown. 2016. “SATenstein: Automatically building local search SAT solvers from components”. *Artificial Intelligence*. 232: 20–42.

Kibria, R. H. 2007. “Evolving a Neural Network-Based Decision and Search Heuristic for DPLL SAT Solvers”. In: *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*. 765–770.

Kibria, R. H. and Y. Li. 2006. “Optimizing the Initialization of Dynamic Decision Heuristics in DPLL SAT Solvers Using Genetic Programming”. In: *Proceedings of the 9th European Conference on Genetic Programming (EuroGP)*. Ed. by P. Collett, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt. Vol. 3905. *Lecture Notes in Computer Science*. Springer. 331–340.

Kibria, R. H. 2011. “Soft Computing Approaches to DPLL SAT Solver Optimization”. *PhD thesis*. Technische Universität Darmstadt.

Kingma, D. P. and J. Ba. 2015. “Adam: A Method For Stochastic Optimization”. In: *Proceedings of the International Conference on Learning Representations*.

Klieber, W., S. Sapra, S. Gao, and E. Clarke. 2010. “A Non-prenex, Non-clausal QBF Solver with Game-State Learning”. In: *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by O. Strichman and S. Szeider. Vol. 6175. *Lecture Notes in Computer Science*. Springer. 128–142.

Kohavi, R. 1995. “A study of cross-validation and bootstrap for accuracy estimation and model selection”. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*. Vol. 2. Morgan Kaufmann. 1137–1143.

Kotthoff, L. 2016. “Algorithm Selection for Combinatorial Search Problems: A Survey”. In: *Data Mining and Constraint Programming: Foundations of a Cross-Disciplinary Approach*. Ed. by C. Bessiere, L. D. Raedt, L. Kotthoff, S. Nijssen, B. O’Sullivan, and D. Pedreschi. Vol. 10101. *Lecture Notes in Computer Science*. Springer. 149–190.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press.

Kurin, V., S. Godil, S. Whiteson, and B. Catanzaro. 2019. “Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning”. arXiv:1909.11830 [cs.LG].

Kusumoto, M., K. Yahata, and M. Sakai. 2018. "Automated Theorem Proving in Intuitionistic Propositional Logic by Deep Reinforcement Learning". arXiv:1811.00796 [cs.LG].

Laarhoven, P. J. van and E. H. Aarts. 1987. *Simulated Annealing: Theory and Applications*. Springer.

Lagoudakis, M. G. and M. L. Littman. 2000. "Algorithm Selection using Reinforcement Learning". In: *Proceedings of the 17th International Conference on Machine Learning (ICML)*. Morgan Kaufmann. 511–518.

Lagoudakis, M. G. and M. L. Littman. 2001. "Learning to Select Branching Rules in the DPLL Procedure for Satisfiability". *Electronic Notes in Discrete Mathematics*. 9(June): 344–359.

Lederman, G., M. N. Rabe, E. A. Lee, and S. A. Seshia. 2019. "Learning Heuristics for Quantified Formulas through Deep Reinforcement Learning". arXiv:1807.08058v3 [cs.LO].

Letz, R., J. Schumann, S. Bayeri, and W. Bibel. 1992. "SETHEO: A high-performance theorem prover". *Journal of Automated Reasoning*. 8(2): 183–212.

Li, C.-M., F. Xiao, M. Luo, F. Manyà, Z. Lü, and Y. Li. 2020. "Clause vivification by unit propagation in CDCL SAT Solvers". *Artificial Intelligence*. 279(Feb.).

Liang, J. H., V. Ganesh, P. Poupart, and K. Czarnecki. 2016a. "Exponential recency weighted average branching heuristic for SAT solvers". In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16)*. 3434–3440.

Liang, J. H., V. Ganesh, P. Poupart, and K. Czarnecki. 2016b. "Learning Rate Based Branching Heuristic for SAT Solvers". In: *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by N. Creignew and D. L. Berre. Vol. 9710. *Lecture Notes in Computer Science*. Springer. 123–140.

Liang, J. H., C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh. 2018. "Machine Learning-Based Restart Policy for CDCL SAT Solvers". In: *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing*. Ed. by O. Beyersdorff and C. M. Wintersteiger. Vol. 10929. *Lecture Notes in Computer Science*. Springer. 94–110.

Liang, J. H., H. G. P. Poupart, K. Czarnecki, and V. Ganesh. 2017. "An Empirical Study of Branching Heuristics Through the Lens of Global Learning Rate". In: *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by S. Gaspers and T. Walsh. Vol. 10491. *Lecture Notes in Computer Science*. Springer. 119–135.

Lindauer, M., H. H. Hoos, F. Hutter, and T. Schaub. 2015. "Autofolio: an automatically configured algorithm selector". *Journal of Artificial Intelligence Research*. 53(1): 745–778.

Lonsing, F. and U. Egly. 2018. “Evaluating QBF Solvers: Quantifier Alternations Matter”. In: *Proceedings of the 24th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by J. Hooker. Vol. 11008. *Lecture Notes in Computer Science*. Springer. 276–294.

Loreggia, A., Y. Malitsky, H. Samulowitz, and V. Saraswat. 2016. “Deep Learning for Algorithm Portfolios”. In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. The AAAI Press. 1280–1286.

Luby, M., A. Sinclair, and D. Zuckerman. 1993. “Optimal speedup of Las Vegas algorithms”. In: *Proceedings of the 2nd Israel Symposium on Theory and Computing Systems*. IEEE. 128–133.

Luenberger, D. 2003. *Linear and Nonlinear Programming*. 2nd Edition. Kluwer Academic Publishers.

Luo, M., C.-M. Li, F. Xiao, F. Manyà, and Z. Lü. 2017. “An Effective Learnt Clause Minimization Approach for CDCL SAT Solvers”. In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*. 703–711.

Lynce, I. and J. Marques-Silva. 2005. “Efficient data structures for backtrack search SAT solvers”. *Annals of Mathematics and Artificial Intelligence*. 43(1–4): 137–152.

Lynce, I. and J. Marques-Silva. 2006. “Efficient Haplotype Inference with Boolean Satisfiability”. In: *Proceedings of the 21st AAAI Conference on Artificial Intelligence*. Vol. 1. The AAAI Press. 104–109.

Malitsky, Y., A. Sabharwal, H. Samulowitz, and M. Sellmann. 2011. “Non-Model-Based Algorithm Portfolios for SAT”. In: *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by K. A. Sakallah and L. Simon. Vol. 6695. *Lecture Notes in Computer Science*. Springer. 369–370.

Malitsky, Y., A. Sabharwal, H. Samulowitz, and M. Sellmann. 2012. “Parallel SAT Solver Selection and Scheduling”. In: *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming*. Ed. by M. Milano. Vol. 7514. *Lecture Notes in Computer Science*. Springer. 512–526.

Malitsky, Y., A. Sabharwal, H. Samulowitz, and M. Sellmann. 2013. “Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering”. In: *Proceedings of the 23rd International Joint Conference on Artifical Intelligence*. Ed. by F. Rossi. AAAI Press. 608–614.

Mangla, C., S. Holden, and L. Paulson. 2020. “Bayesian Optimization of Solver Parameters in CBMC”. In: *Proceedings of the 18th International Workshop on Satisfiability Modulo Theories (SMT)*.

Marić, F. 2009. “Formalization, Implementation and Verification of SAT Solvers”. *PhD thesis*. University of Belgrade.

Marques-Silva, J. 1999. “The Impact of Branching Heuristics in Propositional Satisfiability Algorithms”. In: *Proceedings of the 9th Portugese Conference on Artificial Intelligence (EPIA)*. Ed. by P. Barahona and J. J. Alferes. Vol. 1695. *Lecture Notes in Computer Science*. Springer. 62–74.

Marques-Silva, J. 2008. “Practical Applications of Boolean Satisfiability”. In: *Proceedings of the 9th International Workshop on Discrete Event Systems*. IEEE. 74–80.

Marques-Silva, J. and K. A. Sakallah. 1999. “GRASP: a search algorithm for propositional satisfiability”. *IEEE Transactions on Computers*. 48(5): 506–521.

McCarthy, J. 1960. “Recursive functions of symbolic expressions and their computation by machine, Part I”. *Communications of the ACM*. 3(4): 184–195.

McCune, W. 2003. “Otter 3.3 Reference Manual”. *Tech. rep.* No. MCS-TM-263. 9700 South Cass Avenue, Argonne, IL 60439: Argonne National Laboratory.

McLaughlin, S. and F. Pfenning. 2008. “Imogen: Focusing the Polarized Inverse Method for Intuitionistic Propositional Logic”. In: *Proceedings of the 15th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*. Ed. by I. Cervesato, H. Veith, and A. Voronkov. Vol. 5330. *Lecture Notes in Computer Science*. Springer. 174–181.

Minton, S. 1996. “Automatically Configuring Constraint Satisfaction Programs: A Case Study”. *Constraints: An International Journal*. 1: 7–43.

Mitchell, M. 1998. *An Introduction to Genetic Algorithms*. The MIT Press.

Mitchell, T. 1997. *Machine Learning*. McGraw-Hill.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. 2015. “Human-level control through deep reinforcement learning”. *Nature*. 518(Feb.): 529–533.

Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. 2001. “Chaff: engineering an efficient SAT solver”. In: *Proceedings of the 38th Design Automation Conference*. IEEE. 530–535.

Murphy, K. P. 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press.

Nadel, A. and V. Ryvchin. 2018. “Chronological Backtracking”. In: *Proceedings of the 21st International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Ed. by O. Beyersdorff and C. M. Wintersteiger. Vol. 10929. *Lecture Notes in Computer Science*. Springer. 111–121.

Narizzano, M., L. Pulina, and A. Tacchella. 2006. “The QBFEVAL Web Portal”. In: *Proceedings of the 10th European Workshop on Logics in Artificial Intelligence (JELIA)*. Ed. by M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa. Vol. 4160. *Lecture Notes in Computer Science*. Springer. 494–497.

Nejati, S., J. H. Liang, C. Gebotys, K. Czarnecki, and V. Ganesh. 2017. "Adaptive Restart and CEGAR-Based Solver for Inverting Cryptographic Hash Functions". In: *Proceedings of the 9th International Working Conference on Verified Software: Theories, Tools, and Experiments*. Ed. by A. Paskevich and T. Weis. Vol. 10712. *Lecture Notes in Computer Science*. Springer. 120–131.

Nieuwenhuis, R., A. Oliveras, and C. Tinelli. 2006. "Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T)". *Journal of the ACM*. 53(6): 937–977.

Nikolić, M., F. Marić, and P. Janičić. 2009. "Instance-Based Selection of Policies for SAT Solvers". In: *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by O. Kullman. Vol. 5584. *Lecture Notes in Computer Science*. Springer. 326–340.

Nikolić, M., F. Marić, and P. Janičić. 2013. "Simple algorithm portfolio for SAT". *Artificial Intelligence Review*. 40: 457–465.

Nudelman, E., K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. 2004. "Understanding Random SAT: Beyond the Clauses-to-Variables Ratio". In: *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*. Ed. by M. Wallace. Vol. 3258. *Lecture Notes in Computer Science*. Springer. 438–452.

O'Mahony, E., E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan. 2008. "Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving". In: *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*.

Oh, C. 2015. "Between SAT and UNSAT: The Fundamental Difference in CDCL SAT". In: *Proceedings of the 18th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Ed. by M. Heule and S. Weaver. Vol. 9340. *Lecture Notes in Computer Science*. Springer. 307–323.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Coimputer Problem Solving*. Addison-Wesley.

Peitl, T. and F. Slivovsky. 2017. "Dependency Learning for QBF". In: *Proceedings of the 20th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by S. Gaspers and T. Walsh. Vol. 10491. *Lecture Notes in Computer Science*. Springer. 298–313.

Petke, J., M. Harman, W. B. Langdon, and W. Weimer. 2014. "Using Genetic Improevement and Code Transplants to Specialise a C++ Program to a Problem Class". In: *Proceedings of the 17th European Conference on Genetic Programming (EuroGP)*. Ed. by M. Nikolau, K. Krewiek, M. I. Heywood, M. Castelli, P. Garcia-Sánchez, J. J. Morello, V. M. R. Santos, and K. Sim. Vol. 8599. *Lecture Notes in Computer Science*. Springer. 137–149.

Petke, J., W. B. Langdon, and M. Harman. 2013. "Applying Genetic Improvement to MiniSAT". In: *Proceedings of the 5th International Symposium on Search Based Software Engineering (SSBSE)*. Ed. by G. Ruhe and Y. Zhang. Vol. 8084. *Lecture Notes in Computer Science*. Springer. 257–262.

Pfahringer, B., H. Bensusan, and C. Giraud-Carrier. 2000. "Meta-Learning by Landmarking Various Learning Algorithms". In: *Proceedings of the 17th International Conference on Machine Learning (ICML)*. Ed. by P. Langley. Morgan Kaufmann. 743–750.

Pierce, B. C. 2002. *Types and Programming Languages*. The MIT Press.

Pipatsrisawat, K. and A. Darwiche. 2007. "A Lightweight Component Caching Scheme for Satisfiability Solvers". In: *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Tesing*. Ed. by J. Marques-Silva and K. A. Sakallah. Vol. 4501. *Lecture Notes in Computer Science*. 294–299.

Pisinger, D. and S. Ropke. 2010. "Large Neighborhood Search". In: *Handbook of Metaheuristics*. Ed. by M. Gendreau and J.-Y. Potvin. Vol. 146. *International Series in Operations Research and Management Science*. Springer. 399–419.

Pulina, L. and A. Tacchella. 2007. "A Multi-engine Solver for Quantified Boolean Formulas". In: *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*. Ed. by C. Bessière. Vol. 4741. *Lecture Notes in Computer Science*. Springer. 574–589.

Pulina, L. and A. Tacchella. 2009. "A self-adaptive multi-engine solver for quantified Boolean formulas". *Constraints*. 14: 80–116.

Quinlan, J. 1993. *C4.5: Programs for Machine Learning*. 1st Edition. Morgan Kaufmann.

Quinlan, J. R. 1986. "Induction of Decision Trees". *Machine Learning*. 1: 81–106.

Rabe, M. N. and S. A. Seshia. 2016. "Incremental Determinization". In: *Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Ed. by N. Creignou and D. L. Berre. Vol. 9710. *Lecture Notes in Computer Science*. Springer. 375–392.

Raths, T., J. Otten, and C. Kreitz. 2007. "The ILTP Problem Library for Intuitionistic Logic". *Journal of Automated Reasoning*. 38: 261–271.

Rintanen, J. 1999. "Constructing Conditional Plans by a Theorem-Prover". *Journal of Artificial Intelligence Research*. 10: 323–352.

Rivest, R. L. 1987. "Learning Decision Lists". *Machine Learning*. 2(3): 229–246.

Russell, S. and P. Norvig. 2020. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson.

Saitta, L., A. Giordana, and A. Cornuéjols. 2011. *Phase Transitions in Machine Learning*. Cambridge University Press.

Samulowitz, H. and R. Memisevic. 2007. "Learning to Solve QBF". In: *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*. The AAAI Press. 255–260.

Santos Silva, R. J. M. dos. 2019. “Machine learning of strategies for efficiently solving QBF with abstraction refinement”. *MA thesis*. Instituto Superior Técnico, Universidade de Lisboa.

Schmee, J. and G. J. Hahn. 1979. “A Simple Method for Regression Analysis with Censored Data”. *Technometrics*. 21(4): 417–432.

Sekiyama, T., A. Imanishi, and K. Suenaga. 2017. “Towards Proof Synthesis Guided by Neural Machine Translation for Intuitionistic Propositional Logic”. arXiv:1706:06462v1 [cs.PL].

Sekiyama, T. and K. Suenaga. 2018a. “Automated proof synthesis for propositional logic with deep neural networks”. arXiv:1805:11799v1 [cs.AI].

Sekiyama, T. and K. Suenaga. 2018b. “Automated Proof Synthesis for the Minimal Propositional Logic with Deep Neural Networks”. In: *Proceedings of the 16th Asian Symposium on Programming Languages and Systems (APLAS)*. Ed. by S. Ryu. Vol. 11275. *Lecture Notes in Computer Science*. Springer. 309–328.

Selman, B., H. Kautz, and B. Cohen. 1996. “Local search strategies for satisfiability testing”. In: *Cliques, Coloring and Satisfiability*. Ed. by D. S. Johnson and M. A. Trick. Vol. 26. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society. 521–532.

Selman, B., H. Levesque, and D. Mitchell. 1992. “A New Method for Solving Hard Satisfiability Problems”. In: *Proceedings of the 10th National Conference on Artificial Intelligence*. Association for the Advancement of Artificial Intelligence. AAAI Press. 440–446.

Selsam, D. and N. Bjørner. 2019. “Guiding High-Performance SAT Solvers with Unsat-Core Predictions”. In: *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Ed. by M. Janota and I. Lynce. *Lecture Notes in Computer Science*. No. 11628. Springer. 336–353.

Selsam, D., M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. 2019. “Learning a SAT Solver from Single-Bit Supervision”. arXiv:1802.03685 [cs.AI].

Shaw, P. 1998. “Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems”. In: *Proceedings of the 4th International Conference on Principles and Practice of Constraint Solving (CP)*. Ed. by M. Maher and J.-F. Puget. Vol. 1520. *Lecture Notes in Computer Science*. Springer. 417–431.

Shawe-Taylor, J. and N. Cristianini. 2000. *Support Vector Machines and Other Kernel-Based Learning Methods*. Cambridge University Press.

Shawe-Taylor, J. and N. Cristianini. 2004. *Kernel Methods for Pattern Analysis*. Cambridge University Press.

Silverthorn, B. 2012. “A Probabilistic Architecture for Algorithm Portfolios”. *PhD thesis*. The University of Texas at Austin.

Silverthorn, B. and R. Miikkulainen. 2010. “Latent Class Models for Algorithm Portfolio Methods”. In: *Proceedings of the 24th AAAI Conference on Artificial Intelligence*. Ed. by M. Fox and D. Poole. The AAAI Press. 167–172.

Singh, R., J. P. Near, V. Ganesh, and M. Rinard. 2009. “AvatarSAT: An Auto-tuning Boolean SAT Solver”. *Tech. rep.* No. MIT-CSAIL-TR-2009-039. MIT Computer Science and Artificial Intelligence Laboratory.

Soos, M., R. Kulkami, and K. S. Meel. 2019. “CrystalBall: Gazing in the Black Box of SAT Solving”. In: *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Ed. by M. Janota and I. Lynce. Vol. 11628. *Lecture Notes in Computer Science*. Springer. 371–387.

Sörensson, N. and A. Biere. 2009. “Minimizing Learned Clauses”. In: *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by O. Kullmann. Vol. 5584. *Lecture Notes in Computer Science*. Springer. 237–243.

Spears, W. M. 1996. “A NN Algorithm for Boolean Satisfiability Problems”. In: *Proceedings of the IEEE International Conference on Neural Networks*. 1121–1126.

Streeter, M. and D. Golovin. 2008. “An online algorithm for maximizing submodular functions”. In: *Proceedings of the 21st International Conference on Neural Information Processing Systems (NIPS)*. Ed. by D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou. Curran Associates Inc. 1577–1584.

Sutskever, I., O. Vinyals, and Q. V. Lee. 2014. “Sequence to Sequence Learning with Neural Networks”. In: *Advances in Neural Information Processing Systems (NIPS)*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Vol. 2. 3104–3112.

Sutton, R. S. and A. G. Barto. 2018. *Reinforcement Learning: An Introduction*. 2nd Edition. MIT Press.

Tentrup, L. 2019. “CAQE and QuAbS: Abstraction based QBF solvers”. *Journal on Satisfiability, Boolean Modeling and Computation*. 11(1): 155–210.

Ting, K. M. 2002. “An instance-weighting method to induce cost-sensitive trees”. *IEEE Transactions on Knowledge and Data Engineering*. 14(3): 659–665.

Vaezipoor, P., G. Lederman, Y. Wu, R. Grosse, and F. Bacchus. 2020. “Learning Clause Deletion Heuristics with Reinforcement Learning”. In: *Proceedings of the Conference on Artificial Intelligence and Theorem Proving (AITP)*.

Vapnik, V. 2006. *Estimation of Dependencies based on Empirical Data*. Springer.

Vishwanathan, S. V. N., N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. 2010. “Graph Kernels”. *Journal of Machine Learning Research*. 11: 1201–1242.

Wainberg, M., B. Alipanahi, and B. J. Frey. 2016. “Are Random Forests Truly the Best Classifiers?” *Journal of Machine Learning Research*. 17(1–5).

Wainer, J. and P. Fonseca. 2021. "How to tune the RBF SVM hyperparameters? An empirical evaluation of 18 search algorithms". *Artificial Intelligence Review*. 54: 4771–4797.

Wang, P.-W., P. L. Donti, B. Wilder, and Z. Kolter. 2019. "SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. *Proceedings of Machine Learning Research*. 6545–6554.

Wetzler, N., M. J. H. Heule, and W. J. H. Jr. 2014. "DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs". In: *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing (SAT)*. Ed. by C. Sinz and U. Egly. Vol. 8561. *Lecture Notes in Computer Science*. 422–429.

Williams, R. J. 1992. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". *Machine Learning*. 8(3–4): 229–256.

Wos, L. 1964. "The Unit Preference Strategy in Theorem Proving". In: *Proceedings of the Fall Joint Computer Conference (AFIPS)*. Association for Computing Machinery. 615–622.

Wu, Z., S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. 2019. "A Comprehensive Survey on Graph Neural Networks". arXiv:1901.00596v4.

Xu, L., H. H. Hoos, and K. Leyton-Brown. 2007. "Hierarchical Hardness Models for SAT". In: *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*. Ed. by C. Bessière. Vol. 4741. *Lecture Notes in Computer Science*. Springer. 696–711.

Xu, L., F. Hutter, H. Hoos, and K. Leyton-Brown. 2012a. "Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors". In: *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*. Ed. by A. Cimatti and R. Sebastiani. Vol. 7317. *Lecture Notes in Computer Science*. Springer. 228–241.

Xu, L., F. Hutter, H. H. Hoos, and K. Leyton-Brown. 2008. "SATzilla: Portfolio-based Algorithm Selection for SAT". *Journal of Artificial Intelligence Research*. 32: 565–606.

Xu, L., F. Hutter, H. H. Hoos, and K. Leyton-Brown. 2009. "SATzilla2009: an Automatic Algorithm Portfolio for SAT". In: *SAT 2009 competative events booklet*. 53–55.

Xu, L., F. Hutter, H. Hughes, and K. Leyton-Brown. 2012b. "Features for SAT". *Tech. rep.* University of British Columbia. URL: http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/.

Xu, L., F. Hutter, J. Shen, H. H. Hoos, and K. Leyton-Brown. 2012c. "SATzilla2012: Improved Algorithm Selection Based on Cost-sensitive Classification Models". In: *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*. Ed. by A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, and C. Sinz. *Department of Computer Science Report Series B*. No. B-2012-2. Department of Computer Science, University of Helsinki. 57–58.

Yang, Z., F. Wang, Z. Chen, G. Wei, and T. Rompf. 2019. "Graph Neural Reasoning for 2-Quantified Boolean Formula Solvers". arXiv: 1904.12084v1 [cs.AI].

Yolcu, E. and B. Póczos. 2019. "Learning Local Search Heuristics for Boolean Satisfiability". In: *Proceedings of the 32nd Conference on Neural Information Processing Systems (NeurIPS)*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett. 7992–8003.

Yun, X. and S. L. Epstein. 2012. "Learning Algorithm Portfolios for Parallel Execution". In: *Proceedings of the 6th International Conference on Learning and Intelligent Optimization (LION)*. Ed. by Y. Hamadi and M. Schoenauer. Vol. 7219. *Lecture Notes in Computer Science*. Springer. 323–338.

Zhang, L., C. F. Madigan, M. H. Moskewicz, and S. Malik. 2001. "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver". In: *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*. 279–285.