Learning Interpretable Heuristics for WalkSAT

Yannet Interian[1] , **Sara Bernardini**[2]
[1]University of San Francisco
[2]Royal Holloway University of London
yinterian@usfca.edu, Sara.Bernardini@rhul.ac.uk

Abstract

Local search algorithms are well-known methods for solving large, hard instances of the satisfiability problem (SAT). The performance of these algorithms crucially depends on heuristics for setting noise parameters and scoring variables. The optimal setting for these heuristics varies for different instance distributions. In this paper, we present an approach for learning effective variable scoring functions and noise parameters by using reinforcement learning. We consider satisfiability problems from different instance distributions and learn specialized heuristics for each of them. Our experimental results show improvements with respect to both a WalkSAT baseline and another local search learned heuristic.

1 Introduction

The satisfiability problem (SAT), one of the most studied NP-complete problems in computer science, consists in determining if there exists an assignment that satisfies a given Boolean formula. SAT algorithms typically assume that formulas are expressed in conjunctive normal form (CNF). A CNF formula is a conjunction of clauses; a clause is a disjunction of literals; and a literal is a variable or its negation. SAT has a wide range of practical applications, including electronic design automation, planning, scheduling and hardware verification.

Stochastic local search (SLS) algorithms are well-known methods for solving hard, large SAT instances (Kautz, Sabharwal, and Selman 2009). They are incomplete solvers: they typically run with a pre-set number of iterations, after which they produce a valid assignment or return "unsolved." Algorithm 1 shows the pseudo-code of a generic SLS algorithm. Like most SLS solvers, it starts by generating a random assignment. If the formula is satisfied by this assignment, a solution is found. Otherwise, a variable is chosen by a variable selection heuristic (*pickVar* in Algorithm 1) and that variable is flipped. The loop is repeated until a solution is found or the maximum number of iterations is reached.

WalkSAT (Selman, Kautz, and Cohen 1993; Selman, Kautz, and Cohen 1994; McAllester, Selman, and Kautz 1997) and other successful local search algorithms select the variable to flip from an unsatisfied clause (see Algorithm 2). After picking a random unsatisfied clause c, the choice of which variable in c to flip is made in two possible ways: either a random variable is chosen, or a scoring function

Algorithm 1 *SLS* algorithm

```
Input: A formula $F$ in CNF form
Parameter: max_flips, max_tries
Output: If found, a satisfying assignment

for $i = 1$ to max_tries do
    $flips = 0$
    $X$ be a random initial assignment
    while $flips <$ max_flips do
        if $X$ satisfies $F$ then
            return $X$
        Pick a variable $x$ using $pickVar$
        $X \leftarrow flipVar(x)$
        $flips$ ++
return unsolved
```

Algorithm 2 *pickVar* for WalkSAT

```
Pick a random unsatisfied clause $c$
if $rand() < p$ then
    Pick a random variable $x$ from $c$
else
    Pick a variable $x$ from $c$ with the smallest break value
return $x$
```

is used to select the best variable to flip. The version of WalkSAT in Algorithm 2 picks a variable with the smallest "break" value, where $break(x)$ of a variable x given an assignment X is the number of clauses that would become false by flipping x. Other algorithms and other versions of WalkSAT use different heuristics (Balint and Schoning 2012; McAllester, Selman, and Kautz 1997) for choosing x.

WalkSAT-type algorithms also use a noise parameter p (see Algorithm 2) to control the degree of greediness in the variable selection process. This parameter has a crucial impact on the algorithms' performance (Hoos 2002; Selman, Kautz, and Cohen 1994; McAllester, Selman, and Kautz 1997; Hoos 1999). Hoos et al. (2002) propose a dynamic noise adaptation algorithm in which high noise values are only used when the algorithms appear to not be making progress.

Designing SLS algorithms requires substantial problem-specific research and a long trial-and-error process by the algorithm experts. Also, algorithms seldom exploit the fact that real-world problems of the same type are solved again and again on a regular basis, maintaining the same combinatorial structure, but differing in the data. Problems of this type include, for example, SAT encodings of AI Planning instances (Robinson et al. 2008) and Bounded Model Checking instances (Benedetti and Bernardini 2005).

Recently, there has been increased interest in applying machine learning techniques to design algorithms to tackle combinatorial optimization problems (Bello et al. 2016; Khalil et al. 2017; Bengio, Lodi, and Prouvost 2021; Zhang et al. 2020). In line with this work, our paper focuses on using machine learning to design algorithms for SAT. More specifically, we investigate the use of reinforcement learning to learn both adaptive noise strategies and variable scoring functions for WalkSAT-type algorithms. We call the resulting strategy *LearnWSAT*. The main contributions of this paper are as follows:

- Our technique automatically learns a scoring function and an adaptive noise strategy for WalkSAT-type algorithms.
- Our scoring functions are simple and interpretable. When coded efficiently, they would have a running time per iteration similar to WalkSAT.
- Our approach outperforms both a WalkSAT baseline algorithm and a previously published learned SLS-type algorithm (Yolcu and Póczos 2019).
- Our technique uses a "warm-up" strategy designed to substantially decrease training time.
- Our algorithm, when trained on a specific distribution, generalizes well to both unseen instances and larger instances of the same distribution.

We remark that our goal in this paper is to show how reinforcement learning could be leveraged to make WalkSAT-type algorithms more efficient and their design more practical; we do not aim to offer the fastest WalkSAT implementation, which we leave as future work. [1]

2 Related Work

The literature regarding SAT is vast. We focus here only on the following two topics, which are the most pertinent to our contribution.

2.1 Machine Learning for SAT

Guo et al. (2022) give an in-depth survey of machine learning for SAT. In their classification, our work falls into the category described as "modifying local search solvers with learning modules". There are two other works (Yolcu and Póczos 2019; Zhang et al. 2020) that fall into the same category.

Yolcu and Poczos (2019) use reinforcement learning with graph neural networks to learn an SLS algorithm. The graph neural network takes a factor graph associated which the SAT formula and the current assignment to score each variable. Scoring each variable at every iteration incurs a large overhead, which leads the authors to run experiments only on small SAT instances. Our work is similar to Yolcu and Poczos's (2019) in that we also use a model to score variables. On the other hand, our approach differs from theirs in four ways. Our scoring model is a linear function of a small set of features, which is simple and interpretable. At every iteration, we only score variables from one unsatisfied clause, which makes our model much more scalable and practical. Our features are able to encode time dependencies (e.g. last time a variable was flipped). We learn a separate noise strategy.

Zhang et al. (2020) propose a system (NLocalSAT) for guiding the assignment initialization of an SLS solver with a neural network. Their model feeds the CNF formula into a Gated Graph neural network for feature extraction. The neural network predicts an assignment for the SAT formula. The model is trained to predict a satisfying assignment. The output of the neural network is used to initialize SLS solvers. Whereas NLocalSAT modifies the initialization of the SLS algorithm, our algorithm modifies its internal loop. Those two improvements are potentially compatible.

Selsam et al. (2018) trained a message-passing neural network called NeuroSAT to predict the satisfiability (SAT) or unsatisfiability (UNSAT) of problem instances. The authors trained and evaluated NeuroSAT on random problem instances that are similar to the ones used in our paper. NeuroSAT achieved an accuracy of 85% and successfully solved 70% of the SAT problems. It is worth noting that our approach focuses on predicting satisfiability and does not directly address unsatisfiability. However, our approach demonstrates a significantly higher accuracy on SAT instances.

2.2 Stochastic Local Search for SAT

Various strategies have been proposed for picking the variables to flip within WalkSAT.

McAllester et al. (1997) analyze six strategies. In all the strategies, a random unsatisfied clause c is selected, and the variable is chosen within c. With probability p, a random variable is selected from c; otherwise, one of the six following strategies is implemented. 1) Pick the variable that minimizes the number of unsatisfiable clauses. 2) Pick the variable that minimizes the break value (Algorithm 2). 3) Same as the previous strategy, but never make a random move if one with break value 0 exits. 3) Pick the variable that minimizes the number of unsatisfied clauses, but refuse to flip any variable that has been flipped in the last t steps. 5) Sort the variables by the total number of unsatisfied clauses, then pick the one with the smallest value. Break ties in favor of the least recently flipped variable. 6) Pick a variable using a combination of least recently picked variable and number of unsatisfied clauses.

ProbSAT (Balint and Schoning 2012) uses a scoring function based on the values $make(x)$ and $break(x)$ and samples the variable to pick based on that scoring function. Given a variable x and an assignment X, $make(x)$ is the number of clauses that would become true by flipping x. Note that $make(x) - break(x)$ is the number of unsatisfiable clauses after flipping x. Balint and Schoning (2012)

[1]The implementation can be found here https://github.com/yanneta/learning_heuristics_sat

Algorithm 3 *pickVar* for LearnWSAT

```
Pick a random unsatisfied clause $c$
if $rand() < p_w$ then
    Pick a random variable $x$ from $c$
else
    Compute score $f_\theta(z)$ for each variable $z$ in $c$
    $x \leftarrow$ sample $z$ with prob $\frac{e^{f_\theta(z)}}{\sum_{y \in c} e^{f_\theta(y)}}$
return $x$
```

experiment with various types of scoring functions based on $make$ and $break$ and find that $make$ values can be ignored.

Hoos (2002) proposes a dynamic noise strategy that uses higher values of noise only when the algorithm is in an "stagnation" stage, which is when there is no improvement in the objective function's value over the last $\frac{m}{6}$ search steps, where m is the number of clauses of the given problem instance. Every incremental increase in the noise value is realized as $p \leftarrow 0.8p + 0.2$; the decrements are defined as $p \leftarrow 0.6p$ where p is the noise level.

The work by McAllester et al. (1997) inspired our selection of features for the variable ranking, and the paper by Balint and Schoning (2012) led us to use features based on $break(x)$ and ignore $make(x)$. Finally, the work in Hoos (2002) inspired us to learn an automated noise strategy.

3 Methodology

Algorithm 3 shows the pseudo-code for our $pickVar$ module. Our objective is to learn the functions p_w and f_θ in such a way that they minimize the number of flips needed to solve a SAT problem. We now describe these functions in detail.

3.1 Variable Representation

To score each variable, we first compute some features that represent the state of the variable at the current iteration t. From our discussion of previous work in Section 2.2, we know that $break(x)$ is an important feature in deciding the score of a variable. We also know, from previous work, that we want to avoid flipping variables back and forth. We design features encoding that information.

Let $age_1(x)$ be the last iteration in which x was flipped and $age_2(x)$ the last iteration in which x was flipped and selected by the algorithm using $f_\theta(x)$. Let $last_K(x) = 1$ if x was flipped in the last K iterations by $f_\theta(x)$. Let $\tilde{x} = min(x, 10)$.

Based on this notation, we represent each variable via the following features:

- $bk(x) = \log(1 + break(\tilde{x}))$
- $\Delta_1(x) = 1 - \frac{age_1(x)}{t}$
- $\Delta_2(x) = 1 - \frac{age_2(x)}{t}$
- $last_5(x)$
- $last_{10}(x)$

We use $\tilde{x}$ and $\log$ in the feature $bk(x)$ to make the feature independent of the size of the formulas. $bk(x)$ it is also normalized to be between 0 and 1.

We have selected these features based on an extensive preliminary evaluation performed on a variety of features and formulas. It would be easy to expand our technique to include additional features whenever relevant.

Let $\mathbf{f}(x) = (bk(x), \Delta_1(x), \Delta_2(x), last_5(x), last_{10}(x))$ be the vector representing the variable x at iteration t, given a current assignment X for a formula F. Note that, to compute the vector, we keep updating variables $age_1, age_2, last_{10}$, which is very cheap. Similar to WalkSAT, $break(x)$ is only computed for variables on one clause at each iteration.

3.2 Models for Scoring Variables and Controlling Noise

Our goal is to make our algorithm interpretable and fast, so we use a linear model for scoring variables. Given a feature vector $\mathbf{f} = \mathbf{f}(x)$ for a variable x, $f_\theta(x)$ is a linear model on $\mathbf{f}$:

$$f_\theta(x) = \theta_0 + \sum_i \theta_i \cdot \mathbf{f}_i$$

Inspired by the dynamic noise strategy discussed in Section 2.2, we define the stagnation parameter δ as the number of iterations since the last improvement in the number of satisfied clauses, divided by the number of clauses. Instead of increasing or decreasing it at discrete intervals as in Hoos (2002), our noise is a continuous function of δ, defined as

$$p_w(\delta) = 0.5 \cdot Sigmoid(w_0 + w_1\delta + w_2\delta^2)$$

We use the sigmoid function to ensure p_w being between 0 and 0.5. Those are commonly used values for noise. Parameters w_0, w_1, w_2 are learned together with parameters $\{\theta_i\}_{i=0}^5$ by using reinforcement learning.

After running our initial experiments, we noticed that the effect of the stagnation parameter δ was almost negligible. Therefore, in most of our experiments, we use a noise parameter that is a constant learned for each instance distribution, that is, $p_w = 0.5 \cdot Sigmoid(w_0)$.

3.3 Simplicity and Interpretability of Models

Domingos (1999) states that one interpretation of Occam's razor in machine learning is the following: "*Given two models with the same generalization error, the simpler one should be preferred because simplicity is desirable in itself.*"

Following this basic principle, in our technique, we use simple functions (linear and sigmoid functions) involving a small set of input variables and show that we get better results than related algorithms that use much more complex models, e.g. Yolcu and Poczos's one (2019). Simplicity is also valuable because simple linear models are very fast to evaluate, which is crucial to practical SAT solvers.

Interpretability refers to a model's capacity to be "*explained or presented in understandable terms to a human*" (Doshi-Velez and Kim 2017). Linear models that use only a few simple variables are typically considered highly interpretable. Our variable-scoring model, which has just six coefficients, is therefore highly interpretable. The interpretability of a model is useful because it allows us to identify which features are significant and important and thus

size	θ_1	θ_2	θ_3	θ_4	θ_5	θ_0
$rand_3$						
$(50, 213)$	-21.1	-1.8	-2.9	-0.9	-1.3	0.1
$(75, 320)$	-19.0	-1.8	-2.3	-0.8	-1.1	0.5
$(100, 426)$	-18.1	-1.7	-2.0	-1.2	-1.4	0.6
$(200, 852)$	-19.4	-2.4	-2.6	-1.0	-1.5	-0.2
$rand_4$						
$(30, 292)$	-20.2	-1.2	-3.2	0.9	-2.5	0.28
$(50, 487)$	-14.3	-1.0	-1.4	0.7	-2.1	-0.31

Table 1: Coefficients of the scoring variable model learned for $rand_k(n, m)$. The first column specifies the size of the formulas (n, m). θ_0 is the model's bias.

make decisions about adding or subtracting features. If a feature has a coefficient close to 0, we can infer that the feature lacks statistical significance and should be eliminated. By providing insight into the impact of each model feature, interpretability can help algorithm designers simplify the process of adding, removing, and designing features.

Table 1 provides an example of the scoring parameters associated with random 3-SAT formulas of various sizes. The absolute value of each coefficient in the table allows us to gauge the contribution of each variable to the model. As demonstrated by the coefficients in Table 1, the $bk(x)$ feature has a notably negative impact on the variable score, indicating its strong influence compared to other features. Conversely, the coefficients associated with the noise function $p_w(\delta)$ showed that δ was not a crucial feature, allowing us to simplify our assumptions regarding the noise parameter. This kind of insight can be extremely valuable.

3.4 Training with Reinforcement Learning

To learn heuristics by using reinforcement learning (Sutton and Barto 2018), we formalize local search for SAT as a Markov Decision Process (MDP). For clarity, we describe the MDP assuming that the noise parameter is 0, that is, the algorithm always picks a variable x from a random unsatisfied clause c using features $\mathbf{f}(x)$.

For each problem distribution D, we have an MDP represented as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where:

- $\mathcal{S}$ is the set of possible states. The state encodes the information needed at iteration t to pick a variable to flip. In our setting, a state is a tuple $(X, c, \{\mathbf{f}(x)\}_{x \in c}, t)$, where X is our current assignment, c is a clause unsatisfied by X, and $\{\mathbf{f}(x)\}_{x \in c})$ are the set of features for all variables in c and t is the current step. The formula F uniformly sampled from D is also part of the state, but it is fixed through the episode. There are also two end states: end_{sat} and $end_{unsolved}$.
- $\mathcal{A}$ is the set of actions. Given a state $s = (X, c, \{\mathbf{f}(x)\}_{x \in c}, t)$, the set of actions corresponds to picking a variable to flip from the state's clause c.
- $\mathcal{P}$ is the transition probability function, defining the probability of going from a state-action pair (s, a) to the next state s'. Let $s = (X, c, \{\mathbf{f}(x)\}_{x \in c}, t)$ be our current state, we pick a variable x in c with probability $\frac{e^{f_\theta(x)}}{\sum_{y \in c} e^{f_\theta(y)}}$, which gets us X', the assignment obtained from X by flipping variable x. If X' satisfies the formula F, we move to the end_{sat} state. If the max number of steps is reached and X' does not satisfy F, we move to $end_{unsolved}$. Otherwise, we move to $(X', c', \{\mathbf{f}(x)\}_{x \in c'}, t+1)$, where c' is a random unsatisfied clause by the new assignment X'.
- $\mathcal{R}(s)$ is the immediate reward after transitioning to state s. $\mathcal{R}(end_{sat}) = 1$ and 0 otherwise.
- $\gamma \in (0, 1)$ is the discount factor, which we set to less than 1 to encourage finding solutions in fewer steps.

We reformulate the problem of learning informative heuristics for SAT into the problem of finding an optimal policy π for the MDP described above. We use the well-known REINFORCE algorithm (Williams 1992). Our policy $\pi(s)$ is determined by the function $f_\theta(x)$ that we use to sample the variable to flip based on the feature vector of each variable.

At each training iteration, we sample a batch of formulas from the distribution D and generate trajectories for each formula. We accumulate the policy gradient estimates from all trajectories and perform a single update of the parameters. Algorithm 4 shows the pseudo-code of the REINFORCE algorithm for the case of constant noise and batch size of one.

Algorithm 4 *Reinforce*

```
Input: Training set (Train_ds) from a problem distribution
$D$, policy $\pi_\theta$, discount rate $\gamma$, learning rate $\alpha$

  Initialize parameters of the policy $\pi_\theta$
  Wam-up $\pi_\theta$ by fitting it to WalkSAT scoring function
  for $i = 1$ to Epocs do
    Initialize $g \leftarrow 0$;
    for $j = 1$ to len(Train_ds) do
      $F$ = Train_ds[j]
      Init random assignment $X$; state $S_0$; $history = ()$

      for $t = 0$ to max_flips do
        if $X$ satisfies $F$ then
          break
        Sample action $a \sim \pi_\theta(S_t)$
        Append $(S_t, a)$ to $history$
        $X \leftarrow flip(X, a)$
        Update state $S_{t+1}$
      Set reward $r = 1$ if $X$ satisfies $F$ and 0 otherwise.

      for $t = 0$ to $T = len(history)$ do
        $(S_t, a) \leftarrow history(t)$
        $\hat{p} \leftarrow \pi_\theta(S_t)$
        $g \leftarrow g + \gamma^{T-t} r \nabla \log \hat{p}(a)$
    $\theta \leftarrow \theta + \alpha g$
```

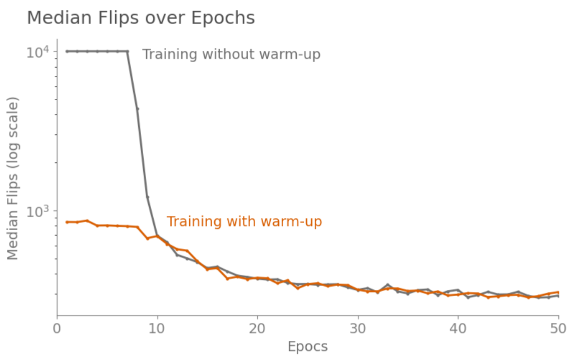

Figure 1: Comparing median flips (log-scale) over epochs on training data for LearnWSAT with and without a 5 epoch warm-up. Training with 1800 formulas of $rand_3(75, 320)$.

3.5 Training with a Warm-Up Strategy

By performing an extensive experimental evaluation, we found that the training of our algorithm takes too long for formulas with over 50 variables when using completely random heuristics and not initially finding a satisfying assignment. Trials without satisfying assignments are not useful for training since they have a reward of zero. To cope with this problem, we design a warm-up strategy to speed up the training process. For a few epochs, we train the function f_θ in such a way that the sampling mimics the *pickVar* strategy from WalkSAT with probability $\frac{e^{f_\theta(z)}}{\sum_{y \in c} e^{f_\theta(y)}}$. We cast this as a classification problem and use log-loss and gradient descent to train f_θ. Figure 1 displays the training with and without warm-up for formulas in $rand_3(75, 320)$, showing the benefit of our approach.

4 Experimental Setting

4.1 Data

We perform experiments using random formulas generated from the following problems: random 3-SAT, random 4-SAT, clique detection, graph coloring and dominating set. These distributions, except for random 4-SAT, are used in the evaluation of GnnSLS by Yolcu and Poczos (2019). To facilitate comparison, we use the same problem distributions. They also used a vertex covering problem that the CNFgen package (Lauria et al. 2017) no longer supports, so we do not include this problem in our experiments.

It has been observed empirically that random K-SAT problems are hard when the problems are critically constrained, i.e. close to the SAT/UNSAT phase boundary (Mitchell et al. 1992; Selman, Mitchell, and Levesque 1996). These problems are used as common benchmarks for SAT. The threshold for 3-SAT is when problems have roughly 4.26 times as many clauses as variables. To generate hard problems for random 4-SAT, we set the number of clauses to be 9.75 times the number of variables (Gent and Walsh 1994). The other three problems are NP-complete graph problems. For each of these problems, a random Erdos–Rényi graph $G(N, p)$ is sampled. To sample from $G(N, p)$, a graph with N nodes is generated by sampling each edge with probability p.

For all these problem distributions, we generate random instances and keep those that are satisfiable. We use the CNFgen package (Lauria et al. 2017) to generate all instances and Minisat (Eén and Sörensson 2003) to filter out the unsatisfiable formulas.

4.2 Algorithms

For comparison, we use the SLS algorithm learned via reinforcement learning developed by Yolcu and Poczos (2019), which we call GnnSLS, and follow the same experimental setup. We also consider one of the WalkSAT versions, as described in Selman et al. (1993). Again, we follow Yolcu and Poczos (2019) in using this particular WalkSAT version.

We wrote our algorithms in Python and PyTorch, which does not make them competitive with state-of-the-art SAT solvers with respect to running time. Indeed, our goal in this paper is to explore the power of reinforcement learning for formulating effective SAT heuristics. To this aim, we offer a prototype algorithm that proves the concept. Although we do not try here to beat highly-optimized current SAT solvers, our results suggest that our technique has the potential to compete with them if written efficiently.

For each problem distribution, we generate 2500 satisfiable formulas. From these, 500 are used for testing, 1900 for training and 100 for validation.

As metrics, we use the median of the median number of flips, the average number of flips and the percentage of instances solved.

4.3 Training with Reinforcement Learning

We train GnnSLS as described in Yolcu and Poczos (2019)'s paper and use their code from the related GitHub repository. The paper uses curriculum learning, where training is performed on a sequence of problems of increasing difficulty. For example, to train problems for $rand_3(50, 213)$, the authors start by first training on $rand_3(5, 21)$, using the resulting model to subsequently train on $rand_3(10, 43)$, $rand_3(25, 106)$ and $rand_3(50, 213)$.

As mentioned above, for experiments with random formulas, our models are trained using 1900 instances. The 100 validation instances are used to select the model with the best median number of steps. We train for 60 epochs using one cycle training (Smith and Topin 2017) and AdamW (Loshchilov and Hutter 2017) as the optimizer (a link to our GitHub repository will be provided in due course). Most of our experiments are run with a discount factor of 0.5.

4.4 Evaluation

For evaluation, we use $max\_tries = 10$ and $max\_flips = 10000$ unless otherwise specified. As said above, for randomly generated problems, we use 500 instances for testing. The noise probability for WalkSAT and GnnSLS is set to $p = \frac{1}{2}$ as in the experiments by Yolcu and Poczos (2019).

5 Experimental Results

Comparison to GnnSLS and WalkSAT. Table 2 summarizes the performance of LearnWSAT compared to GnnSLS

	LearnWSAT	GnnSLS	WalkSAT
$rand_3(50, 213)$			
m-flips	**119**	352	356
a-flips	**384**	985	744
solved	100%	99.6%	100%
$color_5(20, 0.5)$			
m-flips	**103**	137	442
a-flips	**225**	497	787
solved	100%	100%	100%
$clique_3(20, 0.05)$			
m-flips	**68**	200	176
a-flips	**91**	345	238
solved	100%	100%	100%
$domeset_4(12, 0.2)$			
m-flips	**65**	72	171
a-flips	**97**	242	288
solved	100%	100%	100%
$rand_4(50, 487)$			
m-flips	**685**	-	2044
a-flips	**1484**	-	3302
solved	**100%**	-	96%

Table 2: Performance of LearnWSAT compared to GnnSLS and WalkSAT. Three metrics are presented: median (m-flips) and average number of flips (a-flips), and percentage solved (solved).

and WalkSAT. We present results for five classes of problems, $rand_3(50, 213)$, $rand_4(30, 292)$, $color_5(20, 0.5)$, $clique_3(20, 0.05)$ and $domeset_4(12, 0.2)$ and three metrics, median number of flips (m-flips), average number of flips (a-flips), and percentage solved (solved). Table 3 indicates the number of variables and clauses in the sampled formulas and gives a sense of the size of the SAT problems we tackle. Table 2 shows that, after training, LearnWSAT requires substantially fewer steps than GnnSLS and WalkSAT to solve the respective problems. Our algorithm performs better than WalkSAT because it optimizes the variable scoring and the noise parameter to the particular distribution of SAT problems. Our technique is also better than GnnSLS because of the following two reasons. First, we speculate that GnnSLS underfits the problem. The SAT encoding and the model used by GnnSLS are more sophisticated but also much more complex than our approach. It is not possible to directly train the GnnSLS algorithm with problems that have a few variables (e.g. 50 variables). To get the GnnSLS encoding to work well, smarter training and more data are needed. Second, our approach uses time-dependent variables (the last time a variable has been flipped), which GnnSLS is unable to encode.

Generalization to larger instances. In Table 4, we compare the performance of LearnWSAT trained on data sets

Distribution	n	m
$rand_k(n, m)$	n	m
$color_5(20, 0.5)$	100	770
$clique_3(20, 0.05)$	60	1758
$domeset_4(12, 0.2)$	60	996

Table 3: Size of the formula used in our evaluation. The distribution $rand_k(n, m)$ has exactly n variables and m clauses. For all the other distributions, we show the maximum number of variables n and clauses m in the sampled formulas.

of different sizes to assess how well the algorithm generalizes to larger instances after having been trained on smaller ones. We consider random 3-SAT instances of different sizes, $rand_3(n, m)$. As in Table 2, we consider three metrics: median number of flips (m-flips), average number of flips (a-flips), and percentage solved (solved). The second column reports the performance of LearnWSAT (indicated LWSAT for brevity) on instances of different sizes when the algorithm is trained on $rand_3(50, 213)$ only. In the third column, for comparison, we report the performance of LearnWSAT when it is trained and evaluated on instances of the same size. The fourth column reports the performance of GnnSLS when the algorithm is trained on $rand_3(50, 213)$ only. Finally, the last column reports the WalkSAT (indicated WSAT) baseline.

The table shows that our model evaluated on $rand_3(50, 213)$ performs similarly or better than models trained on larger instances. Training becomes much more expensive as a function of the size of the formula, but this result suggests that we can train on smaller formulas of the same distribution. GnnSLS trained on smaller instances can also be evaluated on larger problems of the same distribution, but the results seem to degrade as the formulas get larger.

Table 5 shows results on instances that are harder than the ones shown before. In particular, Minsat is not able to solve some of the instances of $rand_3(500, 2130)$ and $rand_4(200, 1950)$ in less than ten hours. We generated 100 problems from $rand_3(300, 1278)$, $rand_3(500, 2130)$ and $rand_4(200, 1950)$, respectively.

These instances are generated at the SAT/UNSAT threshold, therefore around 50% of them are supposed to be satisfiable. In the case of $rand_4(200, 1950)$, it seems that a few more are satisfiable since LearnWSAT is able to solve 68% of them.

Noise parameter. In our initial experiments, we learned a noise function that depended on the stagnation parameter δ. After inspecting the function, we noticed that the effect of δ is negligible. In Figure 2, we show the learned noise function as used by the algorithm at evaluation time. We plot the noise function against the iteration until the formula is solved. The stagnation parameter varies per iteration, but these curves show very little variation. We ran experiments in which we fixed p_w to be a constant dependent on the distribution and found that the results are similar

	LWSAT (50, 213)	LWSAT	GnnSLS (50, 213)	WSAT
$(50, 213)$				
m-flips	119	119	352	356
a-flips	384	384	985	744
solved	100%	100%	99.6%	100%
$(75, 320)$				
m-flips	260	286	969	880
a-flips	904	948	2253	1772
solved	100%	100%	96.6%	98%
$(100, 426)$				
m-flips	503	575	2264	1814
a-flips	1650	1682	3816	3132
solved	100%	100%	85.6%	93%
$(200, 852)$				
m-flips	4272	4005	10000	10000
a-flips	5329	5085	8359	7497
solved	96.2%	95.6%	26%	46.2%

Table 4: Performance of our algorithm evaluated on different instances of the same distribution. We consider $rand_3(n, m)$ formulas of different sizes (n and m refer to the number of variables and clauses in the sampled formulas). The second column corresponds to evaluating our algorithm (indicated as LWSAT) trained on formulas from $rand_3(50, 213)$ only. The third column corresponds to evaluating the algorithm on instances of the same size used for training. The fourth corresponds to evaluating GnnSLS on the algorithm trained on $rand_3(50, 213)$. The last column reports the WalkSAT (indicated WSAT) baseline. We consider three metrics: median (m-flips) and average number of flips (a-flips), and percentage solved (solved).

to when the noise function depends on δ. In particular, we optimize $p_w = 0.5 \cdot Sigmoid(w)$ by finding a single parameter w per distribution. After these initial experiments, we ran all the others (as they are reported here) with fixed constants. Note that these constants are small compared to typical values used for WalkSAT ($p = 1/2$). This is because our $PickVar$ algorithm (shown in Algorithm 3) injects noise by sampling instead of deterministically picking variables as in the original $PickVar$ algorithm of WalkSAT (Algorithm 2).

Impact of the discount factor. We ran experiments to understand the dependencies of our results on the value of the discount factor for reinforcement learning. Figure 3 shows the median flips as a function of the discount factor. The gray area shows the confidence intervals for each curve. We find that various discount factors give similar results.

Impact of the size of training data. Figure 4 shows the median flips as a function of the size of the training data. The experiment uses formulas from $rand_3(50, 213)$. The plot shows that we need a training size of at least 40 to learn an algorithm that is better than WalkSAT. For optimal results,

	LearnWSAT	WalkSAT
$rand_3(300, 1278)$	48%	26%
$rand_3(500, 2130)$	36%	9%
$rand_4(200, 1950)$	68%	0%

Table 5: Performance of our algorithm evaluated on larger instances of $rand_k(n, m)$. These instances have not been checked for satisfiability. Around 50% of them are expected to be satisfiable. The metric shown is percentage solved. The max_flip parameter is set to 50000 for both solvers. LearnWSAT was trained on $rand_3(50, 213)$ for the first two rows and on $rand_4(50, 487)$ for the last row.

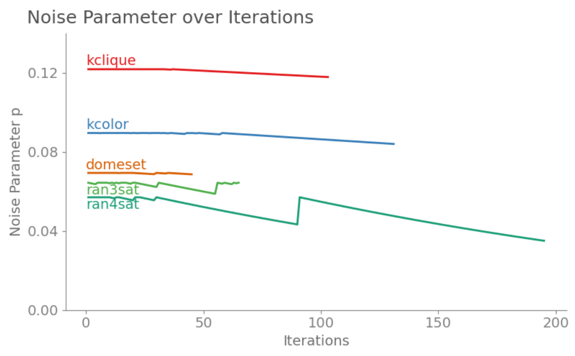

Figure 2: The data corresponding to each line comes from running an evaluation on a single formula for each of the five problem distributions ($color_5(20, 0.5)$, $clique_3(20, 0.05)$, $domeset_4(12, 0.2)$, $rand_3(50, 213)$, $rand_4(30, 292)$) until the SAT assignment is found. The plot shows the noise parameter as a function of the iteration.

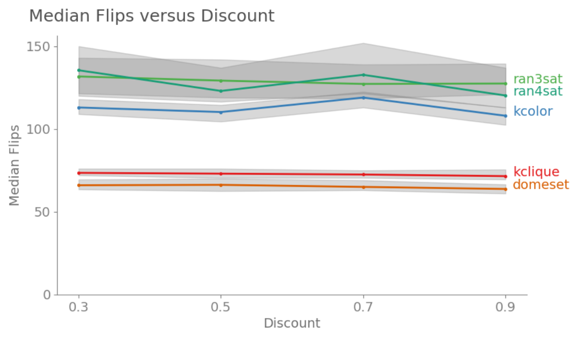

Figure 3: Comparing median flips as a function of the discount factor for various datasets. The lines correspond to training and evaluation on instances of the following distributions: $rand_3(50, 213)$, $rand_4(30, 292)$, $color_5(20, 0.5)$, $clique_3(20, 0.05)$ and $domeset_4(12, 0.2)$

we need at least 160 formulas. To run the experiments with smaller datasets, we increased the number of warm-up steps from 5 to 50 and the amount of epochs from 60 to 200.

6 Conclusions and Future Work

In this paper, we present LearnWSAT, a technique that discovers effective noise parameters and scoring variable func-

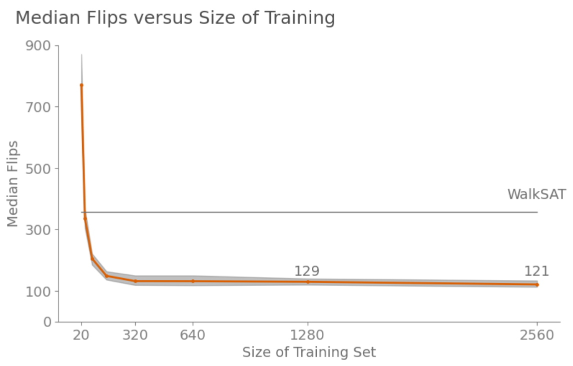

Figure 4: Comparing median flips as a function of the size of the training data. Training with $rand_3(50, 213)$ formulas. Gray region corresponds to confidence intervals. Gray line corresponds to median flips for the WalkSAT baseline.

tions for WalkSAT-type algorithms. Thanks to them, LearnWSAT uses substantially fewer flips than a WalkSAT baseline, as well as an existing learned SLS-type algorithm, to solve the satisfiability problem. Although we do not focus on optimizing the implementation of LearnWSAT in this paper, our experiments suggest that, when coded efficiently, our technique could compete with state-of-the-art solvers.

Despite improving over algorithms in the literature, we note that a limitation of LearnWSAT is the need to predefine a set of features. In addition, training is slow for formulas with 150 variables or more. The last limitation is mitigated by the fact that, as we have shown in the experiments, models trained on smaller formulas generalize well to larger ones. Overcoming these limitations is part of our future work.

Finally, we remark that the ideas presented in this work are general and could be adapted to solve other hard combinatorial problems.

References

Balint, A., and Schoning, U. 2012. Choosing probability distributions for stochastic local search and the role of make versus break. In Cimatti, A., and Sebastiani, R., eds., *Theory and Applications of Satisfiability Testing – SAT 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

Benedetti, M., and Bernardini, S. 2005. Incremental compilation-to-sat procedures. In Hoos, H. H., and Mitchell, D. G., eds., *Theory and Applications of Satisfiability Testing*, 46–58. Berlin, Heidelberg: Springer Berlin Heidelberg.

Bengio, Y.; Lodi, A.; and Prouvost, A. 2021. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research* 290(2):405–421.

Domingos, P. 1999. The role of occam's razor in knowledge discovery. *Data mining and knowledge discovery* 3(4):409–425.

Doshi-Velez, F., and Kim, B. 2017. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*.

Eén, N., and Sörensson, N. 2003. An extensible sat-solver. In Giunchiglia, E., and Tacchella, A., eds., *SAT*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Gent, I. P., and Walsh, T. 1994. The sat phase transition. In *ECAI*, volume 94, 105–109. PITMAN.

Guo, W.; Yan, J.; Zhen, H.-L.; Li, X.; jie Yuan, M.; and Jin, Y. 2022. Machine learning methods in solving the boolean satisfiability problem. *ArXiv* abs/2203.04755.

Hoos, H. H. 1999. On the run-time behaviour of stochastic local search algorithms for SAT. In Hendler, J., and Subramanian, D., eds., *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA*, 661–666. AAAI Press / The MIT Press.

Hoos, H. H. 2002. An adaptive noise mechanism for walksat. In *AAAI/IAAI*.

Kautz, H. A.; Sabharwal, A.; and Selman, B. 2009. Incomplete algorithms. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 185–203.

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems* 30.

Lauria, M.; Elffers, J.; Nordström, J.; and Vinyals, M. 2017. Cnfgen: A generator of crafted benchmarks. In *Theory and Applications of Satisfiability Testing – SAT 2017*, Lecture Notes in Computer Science, 464–473. Germany: Springer. 20th International Conference on Theory and Applications of Satisfiability Testing, SAT 2017.

Loshchilov, I., and Hutter, F. 2017. Fixing weight decay regularization in adam. *CoRR* abs/1711.05101.

McAllester, D. A.; Selman, B.; and Kautz, H. A. 1997. Evidence for invariants in local search. In Kuipers, B., and Webber, B. L., eds., *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, 321–326. AAAI Press / The MIT Press.

Mitchell, D.; Selman, B.; Levesque, H.; et al. 1992. Hard and easy distributions of sat problems. In *Aaai*, volume 92, 459–465.

Robinson, N.; Gretton, C.; Pham, D. N.; and Sattar, A. 2008. A compact and efficient sat encoding for planning. In *ICAPS*, 296–303.

Selman, B.; Kautz, H. A.; and Cohen, B. 1993. Local search strategies for satisfiability testing. In Johnson, D. S., and Trick, M. A., eds., *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, volume 26 of *DIMACS*

Series in Discrete Mathematics and Theoretical Computer Science, 521–531. DIMACS/AMS.

Selman, B.; Kautz, H. A.; and Cohen, B. 1994. Noise strategies for improving local search. In Hayes-Roth, B., and Korf, R. E., eds., *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, 337–343. AAAI Press / The MIT Press.

Selman, B.; Mitchell, D. G.; and Levesque, H. J. 1996. Generating hard satisfiability problems. *Artificial intelligence* 81(1-2):17–29.

Smith, L. N., and Topin, N. 2017. Super-convergence: Very fast training of residual networks using large learning rates. *CoRR* abs/1708.07120.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement Learning: An Introduction*. The MIT Press, second edition.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8(3):229–256.

Yolcu, E., and Póczos, B. 2019. Learning local search heuristics for boolean satisfiability. *Advances in Neural Information Processing Systems* 32.

Zhang, W.; Sun, Z.; Zhu, Q.; Li, G.; Cai, S.; Xiong, Y.; and Zhang, L. 2020. Nlocalsat: Boosting local search with solution prediction. *CoRR* abs/2001.09398.