Neural Guidance for SAT Solving

Sebastian Jaszczur[1], Michał Łuszczyk[1], and Henryk Michalewski[1][2]

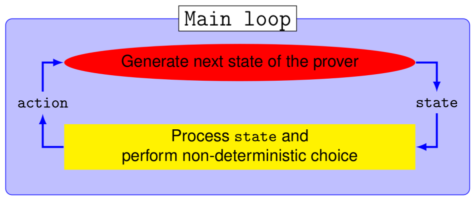
[1] University of Warsaw
[2] deepsense.ai

Abstract

We use neural guidance to direct search of the DPLL algorithm. We compare SAT-solving performance of various heuristics and two neural architectures: LSTM and a message-passing architecture. By a large margin the best one is the message passing architecture, which has more desirable theoretical properties and which is capable of solving complicated instances of SAT problems even when used with a naive implementation of the DPLL algorithm.

1 Introduction

The main processing loop present in all modern fully automatic theorem provers [Sch13, KV13, Ott08] is presented in Figure 1 (left). The loop requires a non-deterministic choice which is usually guided by heuristics.


Main loop
Generate next state of the prover
action
state
Process state and perform non-deterministic choice

Algorithm 1 Simplified propositional SAT algorithm

```
1: function DPLL($\Phi$)
2:     if $\Phi$ is empty then return True
3:     if $\Phi$ contains an empty clause then return False
4:     $l \leftarrow$ choose-literal($\Phi$)
5:     return DPLL($\Phi \wedge l$) or DPLL($\Phi \wedge$ not $l$)
```

Figure 1: The general prover loop on the left. On the right a simplified propositional SAT algorithm

In machine-learning driven approaches, each choice within the main loop will be guided by machine learning methods. This can involve a training corpus which may include prior problems and successful proofs. As an important instantiation of this high-level algorithm in this work we will consider the case of propositional satisfiability. Algorithm 1 is a simplified version of the Davis Putnam Logemann Loveland (DPLL) algorithm [DLL62]. The algorithm assumes that a propositional formula Φ is given in conjunctive normal form (CNF). At every step we choose a propositional literal l (line 4). We then check whether l or its negation can be consistently added as an assumption (line 5), using a recursive call. Of course, state-of-the-art propositional theorem provers such as Glucose [AS09] or MiniSat [ES03], are much more complex, and involve more refined choices. For first-order logic provers, the situation is even more involved [Sch13, KV13, Ott08]. Let us notice that from the point of view of deep learning, dealing with guidance of the DPLL algorithm is a very convenient task and provides a testing ground for various ideas which can be later on employed in the context of more expressive logics. The main advantage over experiments with more expressive logics, such as first-order logic, is presence of strong oracles, that is SAT-solvers capable of solving big propositional problems. This implies that we can apply supervised learning methods without risks present in current experiments with stronger logics, where in absence of a good oracle one has to rely on various heuristics which decide whether a given data point is classified as a negative example [LISK17, PU18].

nodes	samples per hour	samples per node	days to 1e8
8	28693	3587	145
16	58080	3630	72
32	108672	3396	38
64	189504	2961	22
128	352683	2755	12
256	691328	2701	6

problem	Glucose 3	MiniSat 2.2
SR(10)	0.003	0.002
SR(30)	0.009	0.007
SR(50)	0.021	0.015
SR(70)	0.031	0.024
SR(90)	0.052	0.053
SR(110)	0.171	0.137
SR(130)	0.1238	0.504
SR(150)	6.091	3.406

Figure 2: Performance of our multinode training architecture tested using up to 256 servers each equipped with Xeon E5-2680v3@2,5 GHz. The days *column shows how many days of computations are required to generate and train the message passing architecture on 100 million samples drawn from the SR(100) distribution. The table on the right shows the amount of time in seconds required by Glucose 3 [AS09] and Minisat 2.2 [ES03] to solve a randomly sampled problem from SR(n) family (see Appendix B for more details about SR(n) problems). Solvers were accessed using the PySAT interface [IMM18].*

Moreover, for propositional logic we can use interesting random examples of formulas, see Appendix B for definitions of SR and RF random distributions. Here our notion of being interesting is based on two premises: (1) using currently available CPU resources (see Appendix H and Figure 2) we can solve the problems and generate more than 100 millions of examples for training using a modern computer cluster, (2) standard SAT-solvers such as MiniSat and Glucose start to struggle with SAT instances drawn from the distribution, that is solving times are reaching seconds per problem on a modern hardware (see Figure 2); in particular, when standard propositional solvers require seconds to solve a given problem, neural methods become a viable alternative to heuristics employed in standard solvers. While hundreds of time slower in terms of tree search, neural-guided DPLL solver is capable of visiting much less nodes in a proof tree when compared to heuristic methods, see Table 3. We claim that combination of parallelized training and data collection and presence of strong oracles create a situation where current SAT-solvers may be used to bootstrap learning process of a better neural solver.

An important advantage of propositional logic in comparison to other logics is availability of a neural representation of formulas proposed in NeuroSAT [SLB+18], which satisfy the following properties (see also Figure 3):

1. Variable renaming shouldn't change the output.
2. Permutation of literals in a clause shouldn't change the output.
3. Permutation of clauses in a formula shouldn't change the output.
4. Negation of all occurrences of variable shouldn't change the output, except for swapping predicted probabilities for literals of this variable.

For first-order logic some of these postulates are satisfied by the FormulaNet architecture proposed in [WTWD17]. An advantage of our approach when compared to NeuroSAT is an ability to combine a learned guidance with a manual heuristic. This is an approach previously used in [LISK17]: proof is generated to a certain level by a neural guidance, but starting from a fixed depth proof is guided by a conventional heuristic. More details are provided in Appendices.

References

[AAB$^+$15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[ACKS16] Miltiadis Allamanis, Pankajan Chanthirasegaran, Pushmeet Kohli, and Charles A. Sutton. Learning continuous semantic representations of symbolic expressions. *CoRR*, abs/1611.01423, 2016.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[ESA$^+$18] Richard Evans, David Saxton, David Amos, Pushmeet Kohli, and Edward Grefenstette. Can neural networks understand logical entailment? *CoRR*, abs/1802.08535, 2018.

[IMM18] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[KUMO18] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Mirek Olsák. Reinforcement learning of theorem proving. *CoRR*, abs/1805.07563, 2018.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.

[LISK17] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, pages 85–105, 2017.

[Ott08] Jens Otten. leancop 2.0and ileancop 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, pages 283–291, 2008.

[PU18] Bartosz Piotrowski and Josef Urban. Atpboost: Learning premise selection in binary setting with ATP feedback. *CoRR*, abs/1802.03375, 2018.

[Sch13] Stephan Schulz. System description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, pages 735–743, 2013.

[SLB$^+$18] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. *CoRR*, abs/1802.03685, 2018.

[SS18] Taro Sekiyama and Kohei Suenaga. Automated proof synthesis for propositional logic with deep neural networks. *CoRR*, abs/1805.11799, 2018.

[WTWD17] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. *CoRR*, abs/1709.09994, 2017.

A Related Work

NeuroSAT The architecture proposed in [SLB$^+$18] is the most important inspiration for our work. Our most effective implementation follows closely the message-passing architecture proposed in NeuroSAT, but instead of using it directly for generation of assignments, we use a similar message-passing architecture for guidance of Algorithm 1.
TreeNN The PossibleWorldNet architecture described in [ESA$^+$18] is based on the TreeNN architecture, with an additional idea of checking multiple possible worlds. We consider the DPLL algorithm as an alternative approach to exploration of possible worlds. It is worth noting that PossibleWorldNet could be modified to use a message-passing architecture while keeping exploration of possible worlds. Another application of TreeNN for proof synthesis in propositional logic was proposed in [SS18].
FormulaNet FormulaNet [WTWD17] solves a problem of premise selection in theorem proving in higher-order logic. It uses a graph representation of formulas similar to NeuroSAT, but more general in order to accommodate for higher-order logic. To the best of our knowledge the architecture was never used for neural guidance.
Eqnet EqNet [ACKS16] solves a more general problem. It embeds a boolean (alternatively: arithmetic) expression into an n-dimensional sphere, in such a way that expressions within a single equivalence class (same semantics) should be close to each other. Assuming correctness of such an embedding, one could infer satisfiability of a formula by measuring its Euclidean distance to a fixed unsatisfiable formula, e.g. $a \wedge \neg a$. However, in our experiments using this metric for guidance of DPLL proved to be problematic. For example, we found that an unsatisfiable formula $b \wedge \neg b$ was closer to a satisfiable formula than to $a \wedge \neg a$.

B Random Generation of Formulas

In our experiments we use two different methods of generating random formulas. The first method, $RF(k, n, m)$, generates CNF formulas with m clauses. Each clause is independently drawn from uniform distribution across all possible clauses of size k on n variables.

Second method, $SR(n)$ is a reproduction of method used in NeuroSAT and it is described in detail in [SLB$^+$18]. It is parametrized only by n - the number of variables used in a formula. Both the size and number of clauses vary to keep the dataset balanced in terms of number of satisfiable and unsatisfiable examples. In general, RF may produce unbalanced dataset with overall easier examples than SR.

C Heuristics

We consider three simple heuristics which are used to compare with neural guidance

Random Variable (randomvar) Select random variable a from the set of the variables occurring in the formula, then choose a random literal out of a and ¬a.

Random Clause (randomclause) Select one of formula's clauses randomly and then select random literal within the clause. This reduces the problem by at least one clause (and one variable).

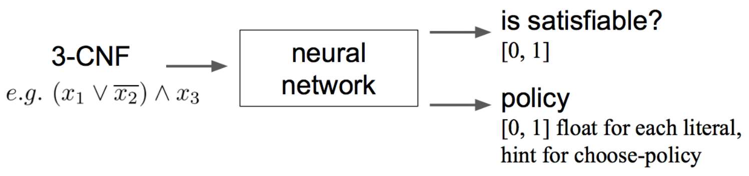
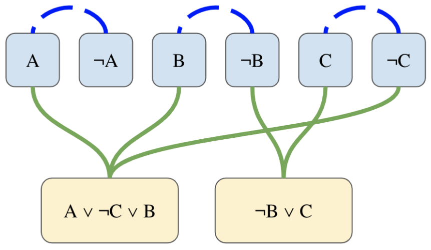

3-CNF
$e.g.\ (x_1 \vee \overline{x_2}) \wedge x_3$
neural network
is satisfiable?
[0, 1]
policy
[0, 1] float for each literal, hint for choose-policy

Figure 4: High level architecture of the neural networks used for `choose-policy`.

Most Common Literal (mostcommon) Select the most frequently occurring literal within the formula.

D Architectures



Property	EqNet	TreeNN	LSTM	Message-passing
1. Variable renaming	No	No	No	**Yes**
2. Permutation of literals	No	Maybe*	Maybe*	**Yes**
3. Permutation of clauses	No	Maybe*	No	**Yes**
4. Negation of variable	No	No	No	**Yes**

Figure 3: Left: a graph representation of formula $(A \vee \neg C \vee B) \wedge (\neg B \vee C)$ used in our work and in NeuroSAT. In the model nodes are unlabeled (variable names are included only for the reader convenience). Different colors mark two distinct types of nodes, and two distinct types of edges. Right: the table compares various architectures with respect to four desirable properties of an embedding described in the Introduction. Maybe* *refers to the fact that the property may be achieved using a particular method of aggregation.*

We consider a general framework for neural network architectures for SAT problems. It should be a function with input being a CNF formula, that is a list of clauses, each clause being a list of literals. It should have two outputs: (1) probability of the whole formula being satisfiable, (2) for each literal, probability of the formula being satisfiable with this literal, assuming current formula is satisfiable. This corresponds to a decision about branching in DPLL.

In order to train the model predicting those probabilities we sum together losses for SAT predictions and policy predictions. We define SAT loss as cross-entropy loss between the prediction and the ground truth. We define policy loss as zero if formula is unsatisfiable and as the average of cross-entropy losses between policy predictions and ground truths if formula is satisfiable. More detailed description of architectures is presented in Appendix E.

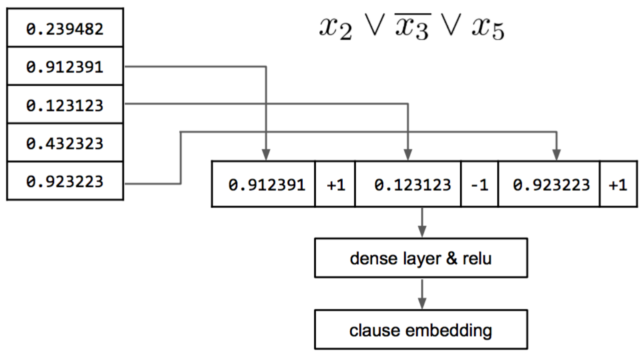


Figure 5: Literal embeddings and clause embeddings. Embedding for each variable is stored in a trainable lookup table. Embeddings for all variables in the clause are appended a ±1 indicating whether the literal is positive or negative and concatenated. Such a tensor is passed through a dense layer with ReLU activation to form the clause embedding.

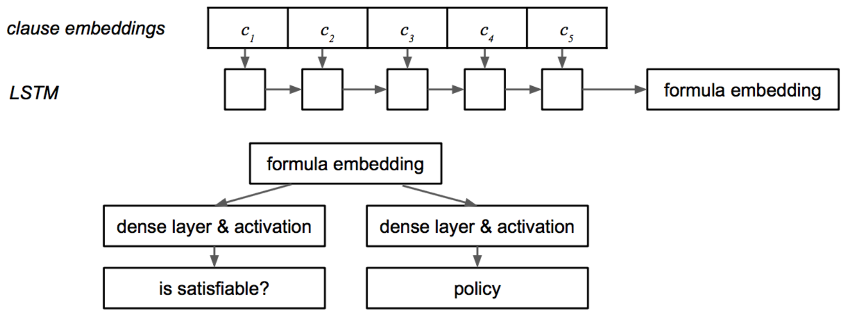


Figure 6: Formula embedding is the output of LSTM on the sequence of clause embeddings. The network outputs: is satisfiable? and the policy are calculated based on the formula embedding.

E Details of Architectures

E.1 LSTM

Taking into account the one dimensional sequential structure of a 3-CNF formula (or two dimensional for n-CNF), we have evaluated a sequential model that calculates embeddings on 3 levels: variable, clause and the formula.

Variable embedding Variable embedding is read from a trainable lookup table with a single row for each variable.

Clause embeddings Clause embedding (Figure 5) is generated with a dense layer with an activation that takes the concatenation of the following tensors as its input: for each literal in the clause, the embedding of the variable and a single ±1 number indicating whether the literal is positive or negative.

Formula embedding Formula embedding is the output of an LSTM layer (Figure 6). The input of the LSTM layer is a sequence of clause embeddings.

SAT and policy The output tensors SAT and policy are generated from a formula embedding with separate dense layers with activations.

E.2 Message-passing/graph-based/NeuroSAT

Unfortunately, the LSTM-based architecture doesn't have any of the desirable properties described in the Introduction (see also Figure 3 Right). We can however use architecture similar to one described in [SLB+18], which is message-passing neural network.

We represent a formula as a graph. We have two types of nodes: a clause node for each clause and a literal node for each literal (two per variable).

We also have two types of edges. Negation edge is a symmetrical edge between each literal and its negation. Clause edge is an asymmetrical edge between each clause and each literal it contains.

Every node contains an embedding. We initialize them with a trainable initial embedding for each type of node. Then we run a number of iterations (from 20 to 50 in our experiments), each consisting of three stages:

1. Message: Each node sends a message based on its embedding, to every connected node.
2. Aggregate: Each node aggregates received messages according to edge type it received it from.
3. Update: Each node updates it's embedding based on it's current embedding and aggregated messages it received.

(1) is realized by sending unmodified embedding. (2) is realized by averaging together all messages coming to the node, grouped by edge type. Averaged embeddings of different types are then concatenated. (3) is realized by hidden dense layers with ReLU activations, ended with dense layer with tanh activation. They together take aggregated embeddings from a previous step, concatenated with current node's embedding, and return node's updated embedding. The parameters of those layers are shared between all the nodes of a given type.

For predicting policy we add on top of each literal's embedding a logistic regression (with shared parameters). For predicting SAT we add a linear regression on top of each literal's embedding, and then apply a softmax on sum of their outputs.

We usually use 20-50 iterations during training, depending on the size of formulas. In each iteration we use cross-entropy losses as described in Appendix D. To get a loss of the whole model we sum together losses for every iteration, so model tries to make good predictions for every iteration.

F Future Modifications to the Sequential Approach

We consider the following modifications of the sequential (LSTM) approach for the future work.

Sequence at clause level Instead of the dense layer converting literal embeddings into the clause embedding, an LSTM layer can be used. This yields a more general model: size of clauses needs not to be fixed at train time.

size	number	parameters
1	2	2
128	10	1280
32768	1	32768
16384	4	65536
49152	1	49152

Table 1: The table lists all layers present in our implementation of the message passing architecture. The left column lists sizes of layers and the center column lists corresponding numbers of layers. In total the network has 148738 *parameters.*

Bag of words of literal embeddings To build the clause embedding, the bag of words approach can be used rather than dense or LSTM layer from the previous variant. To each literal embedding an additional dense layer can be applied (the dense layers share the weights) and then the outputs are aggregated with a sum or an average. This changes makes the model invariant to the permutations of literals within a clause.

Bag of words of clause embeddings Just as above, bag of words can be used rather than LSTM layer to calculate the formula embedding out of clause embeddings. This change makes the model invariant to the permutations of clauses within the formula.

G Implementation

The code for our architecture is available at `https://github.com/neuralguidanceforsat/neuralguidanceforsat`. Neither the code or experiments are optimized with respect to performance. In particular we use the same hyperparameters in all experiments with the message-passing architecture and all our code, including the main DPLL loop, are written in Python using the TensorFlow [AAB+15] framework.

H Experimental Results

In each training we fixed either RF or SR distribution of formulas, see Appendix B for more details. In every step of training we draw a new propositional formula from the selected distribution. This is a costly procedure in the case of the SR distribution, but reduces chances of overfitting. Following [LISK17] we assume that neural guidance is more useful when applied in first iterations of the proving loop. Hence in experiments marked as `hybrid` we switch from neural guidance to the `mostcommon` heuristic (see Appendix C for a definition) after 200 steps of the proof. We expect that neural guidance may limit the search space, in particular that it will be feasible to find a proof after limited number of steps: see Tables 2 and 4. In Table 3 we compare various proving strategies on datasets SR(n) for $n \leq 50$. Training of LSTM on SR problems and on large RF problems proved to be quite difficult. Figure 7 (left) summarizes performance of LSTM models on relatively small RF problems. Training details are described in more details in Appendix I. Figure 7 (right) shows performance of the message-passing architecture with respect to the number of layers. The results show in particular that policy accuracy as well as satisfiability accuracy substantially improves with subsequent iterations.

guidance	SR(16)	SR(18)	SR(20)	SR(25)	SR(30)	SR(40)	SR(50)	SR(70)	SR(90)
hybrid SR(8)	**99**	83	**76**	57	49	22	23	11	2
hybrid SR(30)	93	89	71	58	42	25	12	6	0
hybrid SR(50)	93	**90**	65	**61**	**53**	**46**	**30**	**19**	**16**
hybrid SR(100)	94	85	68	58	31	25	6	4	2
mostcommon	**99**	**90**	60	26	10	5	3	0	0

Table 2: Problems solved in 1000 steps using various DPLL heuristics. In the first row we consider guidance trained on the SR(8) dataset and test it against SR(n) up to $n = 90$*. In subsequent row we test other instances of guidance. Compare with Table 4 in the Appendix.*

variable number	6		8		10	
clause number	30	60	40	80	50	100
lstm6_avg_error	0.00	0.02	NaN	NaN	NaN	NaN
lstm6_avg_step	**6.85**	7.20	NaN	NaN	NaN	NaN
lstm6_std_error	0.00	0.14	NaN	NaN	NaN	NaN
lstm6_std_step	0.36	1.71	NaN	NaN	NaN	NaN
lstm8_avg_error	0.20	0.09	0.26	0.13	NaN	NaN
lstm8_avg_step	10.06	8.16	16.90	11.34	NaN	NaN
lstm8_std_error	0.45	0.29	0.54	0.34	NaN	NaN
lstm8_std_step	9.38	4.48	19.69	7.80	NaN	NaN
lstm10_avg_error	0.33	0.17	0.43	0.32	0.47	0.28
lstm10_avg_step	11.82	9.00	18.11	13.79	29.58	15.83
lstm10_std_error	0.53	0.38	0.59	0.55	0.61	0.47
lstm10_std_step	10.73	5.09	18.59	9.53	35.63	12.75
neurosat_avg_error	0.04	0.01	0.01	0.00	0.07	0.00
neurosat_avg_step	7.24	**7.03**	**9.49**	**8.99**	**12.64**	**10.98**
neurosat_std_error	0.24	0.10	0.10	0.00	0.26	0.00
neurosat_std_step	2.89	0.52	7.71	0.10	10.00	0.14
randomvar_avg_error	2.41	3.08	3.67	3.90	4.19	4.99
randomvar_avg_step	17.48	18.70	43.00	27.40	70.48	49.23
randomvar_std_error	1.18	1.15	1.46	1.39	1.48	1.73
randomvar_std_step	9.77	7.54	26.10	12.86	50.51	25.46
mostcommon_avg_error	0.49	0.39	0.65	0.46	0.91	0.54
mostcommon_avg_step	9.84	8.55	15.01	12.64	20.51	15.33
mostcommon_std_error	0.59	0.58	0.65	0.67	0.79	0.61
mostcommon_std_step	5.30	3.25	10.96	7.47	15.12	8.56

iteration	policy loss	sat loss	policy error	sat error
1	0.69	0.69	0.47	0.45
2	0.69	0.69	0.39	0.48
3	0.67	0.7	0.36	0.48
4	0.65	0.79	0.31	0.54
5	0.63	0.69	0.3	0.47
6	0.63	0.76	0.29	0.44
7	0.58	0.71	0.26	0.4
8	0.56	0.62	0.23	0.33
9	0.53	0.56	0.2	0.29
10	0.51	0.56	0.18	0.26
11	0.48	0.48	0.14	0.19
12	0.46	0.43	0.13	0.16
13	0.45	0.4	0.12	0.14
14	0.43	0.36	0.09	0.14
15	0.43	0.3	0.09	0.13
16	0.41	0.3	0.07	0.11
17	0.4	0.27	0.07	0.09
18	0.39	0.23	0.06	0.08
19	0.38	0.24	0.05	0.07
20	0.38	0.22	0.05	0.05

Figure 7: Left: DPLL with LSTM, NeuroSAT and heuristic `choose-literal` functions. LSTM was trained with 6, 8 and 10 variables, NeuroSAT was trained with 8 variables. Both architectures were trained on the RF distribution. The evaluation is performed on batches of 100 random satisfiable 3-CNFs from the RF distribution with fixed variable and clause numbers as specified in the table. For each testing dataset, the lowest step number is marked in bold. Right: Losses and accuracy for the model trained on the SR(8) dataset after training on 10 million samples.

I More Experimental Results

I.1 LSTM

We have evaluated the quality of the LSTM model as a `choose-literal` oracle for DPLL. We compared the results to 3 basic heuristics: `randomvar`, `randomclause` and `mostcommon` as well as to NeuroSAT.

I.1.1 Experiment setup

In each of the following experiment series, we trained LSTM on formulas up to the specified number of variables and clauses. Then, for each experiment, we selected fresh 100 random satisfiable 3-CNF formulas from the RF distribution with the parameters (number of variables, number of clauses) as noted and run DPLL with one of 5 aforementioned implementations of `choose-literal`. For each DPLL execution we measured two metrics: number of steps (recursive calls to the DPLL procedure) and number of errors (the number of times when a DPLL given a satisfiable formula chooses an unsatisfiable formula for the recursive call).

I.1.2 Results

LSTM trained on 6 vars and up to 30 clauses. LSTM outperforms heuristics `mostcommon`, `randomvar` and `randomclause` in all experiments with 6 variables in terms of the average step number and makes at most 2 wrong guesses per 100 DPLL executions as shown in Table 7 (left).

The LSTM model was trained on 24M samples and had loss 0.29 and policy error 0.05 (Figure 8). This experiment shows that the LSTM model generalizes well to up to 2x longer formulas.

LSTM trained on 8 vars and up to 40 clauses. LSTM has a lower average error than the basic heuristics on both 6 and 8 variable test cases. However, `mostcommon` has a lower average step number in three out of four cases. `mostcommon` has a higher error, but reduces the problem faster and thus recovers quicker from the errors and has lower average step number.

LSTM and `mostcommon` have lower average error and average step number than `randomclause` and `randomvar` in all experiments.

The LSTM model was trained on 32M samples and had loss 0.43 and accuracy policy error 0.085 (Figure 8).

LSTM trained on 10 vars and up to 50 clauses Just as in the 8 variable case, LSTM always has a lower average error than the basic heuristics. In the 10 variable case, the number of average steps is always higher compared to `mostcommon`.

The LSTM model was trained on 40M samples and had loss 0.42 and policy error 0.089 (Figure 8).

It is worth noting that all these problems (up to 10 variables) can be easily solved with a naive algorithm. The naive algorithm simply iterates through all possible valuations of the set of variables and checks whether the formula is satisfied for a given valuation.

I.2 Message-passing architecture

In this paper we present one model trained on 10 million samples drawn from the SR(8) distribution. We also present models trained on SR(30), SR(50), SR(100) distributions. Each of these models was trained using 50 millions samples. None of these models fully converged. In particular, policy error levels for the last iteration of the network are around 0.23 and errors in assessment of satisfiability are around 0.3.

model	SR(8)	SR(10)	SR(12)	SR(14)	SR(15)	SR(16)	SR(18)	SR(20)	SR(25)	SR(30)	SR(40)	SR(50)
hybrid SR(8)	**9**	17	25	101	**85**	**144**	326	633	2601	9964	158542	185995
SR(8)	**9**	17	24	134	125	254	746	1671	10687	NaN	NaN	NaN
hybrid SR(30)	20	20	**15**	173	172	298	339	783	5714	7774	169314	1893618
hybrid SR(50)	15	**14**	59	**77**	159	326	**229**	**485**	**2325**	6692	**29255**	**79886**
hybrid SR(100)	11	51	90	142	192	290	504	752	3397	**6022**	54051	2046069
mostcommon	24	53	101	183	240	338	313	1185	4556	13566	1651067	2837835
randomvar	65	144	270	538	862	1164	2436	5251	25275	125610	4358797	NaN
randomclause	51	103	216	394	548	745	1363	2788	13385	57509	1705605	NaN

Table 3: Average number of steps over 100 trials for models listed in the right column. Models were tested on datasets SR(n) for various $n \leq 50$. NaN refers to a timeout.

guidance	SR(50)	SR(70)	SR(90)
hybrid SR(8)	45	20	0
hybrid SR(30)	46	0	0
hybrid SR(50)	**71**	**30**	**24**
hybrid SR(100)	60	14	0
mostcommon	20	0	0

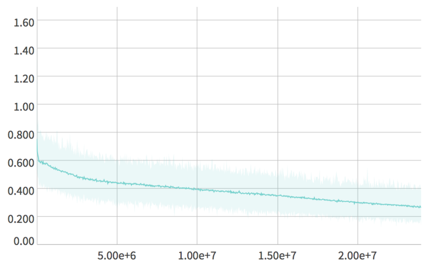
Table 4: Problems solved in 1 millions steps using various DPLL heuristics. In the first row we consider guidance trained on the SR(8) dataset and test it against SR(n) up to $n = 90$. In subsequent rows we test other instances of guidance. For $n \leq 40$ all methods solve all problems within 1 millions steps. Compare with Table 2.

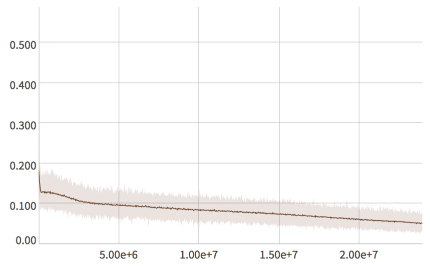
I.3 Parallelization

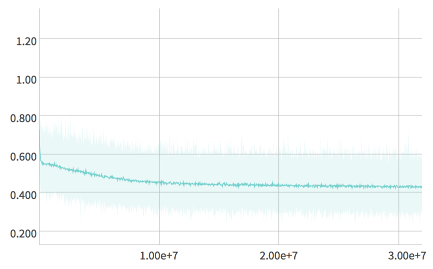
Parallelization was performed using Distributed TensorFlow and the SLURM framework.

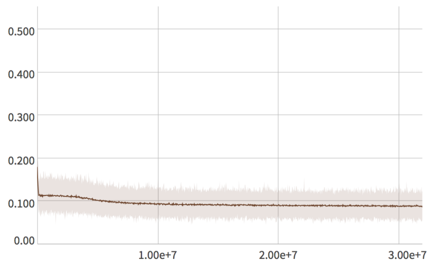
J Future Work

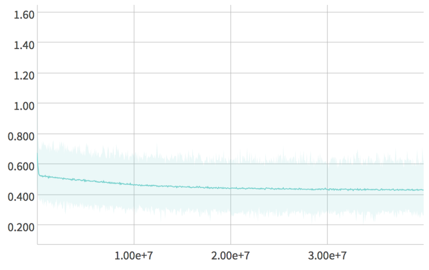
In this work we show preliminary evidence that SAT-solving can be augmented by neural networks. The evidence consists in showing that (1) data-collection and learning can be efficiently parallelized to hundreds of nodes, (2) neural guidance based on the message-passing architecture behaves competitively when evaluated on propositional problems with tens of variables. We believe that the learning process can be extended to more complex examples using an appropriately large computing infrastructure and that in a near future parallelization combined with a variant of the message architecture can be used to train models which will tackle SR and other SAT instances currently beyond reach of SAT-solvers. As future steps we consider (1) extending the prover so it can prove unsatisfiability, (2) refinement to the message-passing architecture, (3) deployment of the main prover loop on GPU. Once we exhaust the pool of available supervised data it would be interesting to apply reinforcement learning methods, including in particular methods recently presented in [KUMO18].


Loss for LSTM 6


Policy error for LSTM 6


Loss for LSTM 8


Policy error for LSTM 8


Loss for LSTM 10

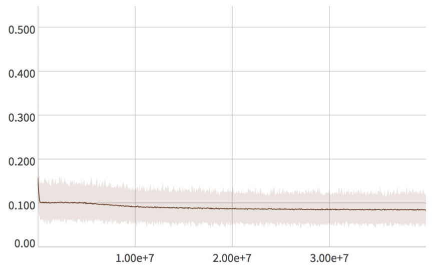

Policy error for LSTM 10

Figure 8: Training losses and policy errors for the LSTM models. 3 models are presented, each one fitting formulas with at most 6, 8 and 10 variables. The number of samples for training is $4 \cdot v \cdot 10^6$*, where* v *is the maximum number of variables.*