Applying Machine Learning to Improve SAT Solvers: Some Highlights

Dr Sean B Holden

sbh11@cl.cam.ac.uk

www.cl.cam.ac.uk/~sbh11/

Department of Computer Science and Technology

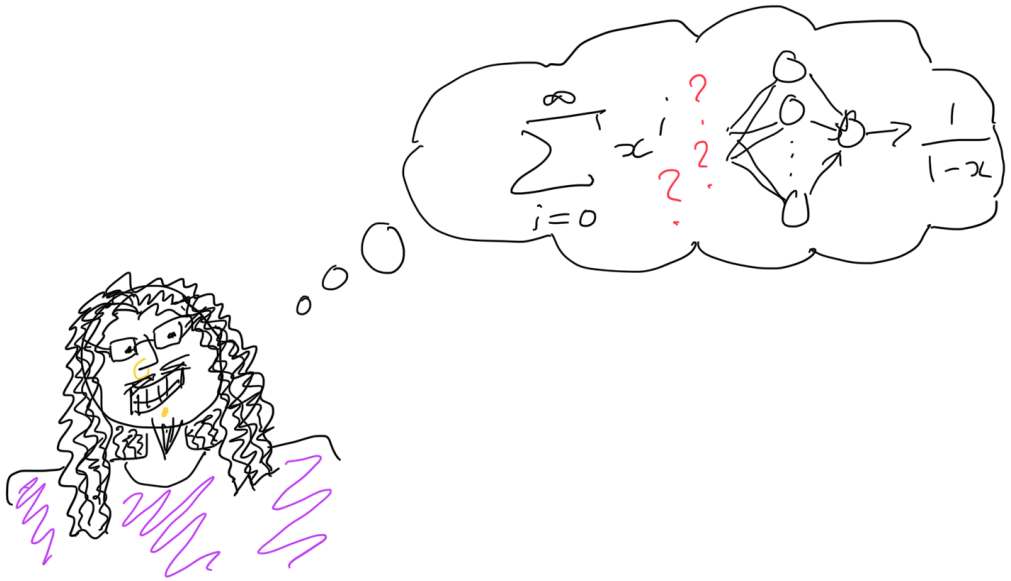
The Computer Laboratory

William Gates Building

JJ Thomson Avenue

Cambridge. CB3 0FD

Approximately 1989…



How can machine learning be applied to proving mathematical theorems?

…but my PhD research went elsewhere. (Well, I was in a DSP lab!) But the interest never went away.

Early 2010s...

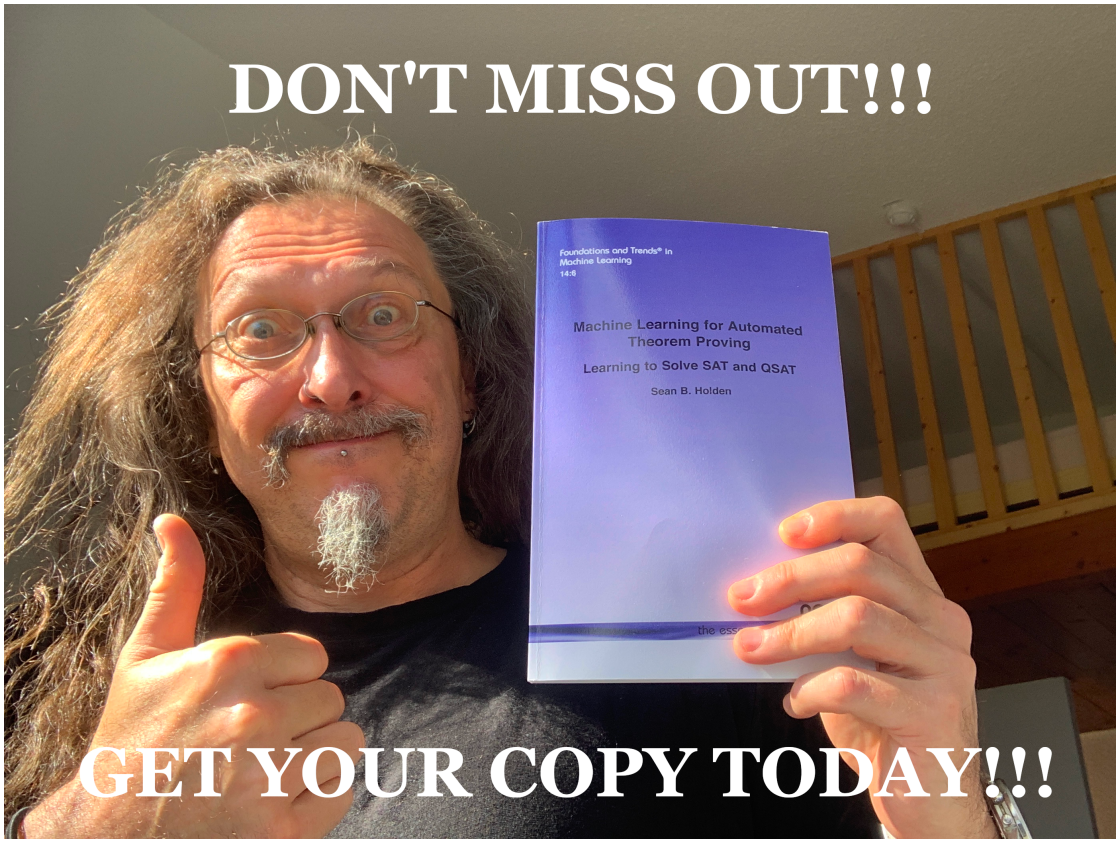
- I started to talk to Larry Paulson and Josef Urban about the problem.
- I worked with James Bridge and Larry Paulson on learning heuristic choice for the *E Prover*.
- This was fun!

In 2016 Josef Urban invited me to speak at the *International Conference on Artificial Intelligence and Theorem Proving (AITP)*.

- So I gave the talk, and then decided to write a review paper...
- ... and it turns out there's *lots* of material...
- ... *lots, and Lots, and LOTS!!!!* of material...
- ... so I concentrated on *SAT* and *QSAT*. (Still $\simeq 230$ relevant publications.)

Present day...

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



Book: https://www.nowpublishers.com/article/Details/MAL-081

CADE-28 half-day tutorial: https://www.cl.cam.ac.uk/~sbh11/research.html

Why is this challenging?

The application of ML is delicate:

- It is often straightforward to identify and solve a relevant ML problem…
- …*BUT* solvers can be very sensitive to small implementation changes.

As a result:

- ML research in this context has aspects *unique to the SAT/QSAT application*.

Also, in theorem-provers in general, a *small change to the problem* can make a huge *change to a subsequent attempt at proof*.

Finally, SAT and QSAT are *inherently hard*.

And *inherently interesting*.

The SAT problem

- *Variables* $v_1, v_2, \ldots$ can take values $\texttt{t}$ (true) or $\texttt{f}$ (false).
- A *literal* l is a variable v or its *negation* $\neg v$. $l = \texttt{t}$ iff $\neg l = \texttt{f}$ and vice versa.
- *Formulas:* $\texttt{t}$, $\texttt{f}$ are formulas. Any variable is a formula. Also, if A and B are formulas then so are:
  1. $\neg A$.
  2. $A \wedge B$.
  3. $A \vee B$.

  $A \to B$ is equivalent to $\neg A \vee B$ and $A \leftrightarrow B$ is equivalent to $(A \to B) \wedge (B \to A)$.
- *Assignment:* Let a formula F contain the variables $v_1, \ldots, v_n$. An *assignment* is a function $a : v_i \mapsto \{\texttt{t}, \texttt{f}\}$.
- *Semantics:* Given a formula F and an assignment a, the truth or falsity of F can be deduced. $\texttt{t}$ and $\texttt{f}$ take their own values. A variable v takes value $a(v)$.
  1. $\neg A = \texttt{t}$ iff $A = \texttt{f}$ and vice versa.
  2. $A \wedge B$ iff $A = \texttt{t}$ and $B = \texttt{t}$.
  3. $A \vee B$ iff $A = \texttt{t}$ or $B = \texttt{t}$.

The DPLL algorithm

A *clause* is a formula of the form $l_1 \vee l_2 \vee \cdots \vee l_n$ where the l_i are literals. An empty clause is $\texttt{f}$.

Any formula F can be written as $C_1 \wedge C_2 \wedge \cdots \wedge C_m$. This is known as *conjunctive normal form (CNF)*. If F has no clauses it is $\texttt{t}$.

The SAT Problem: given a formula F, is there an assignment a such that F is $\texttt{t}$ under a?

The *Davis-Putnam-Logemann-Loveland (DPLL)* algorithm takes an F in CNF and decides whether there exists such an a. Essentially, simple *depth-first search with chronological backtracking*.

1. For any *unit clause* $C = \{l\}$, if l has the form v then $a(v) = \texttt{t}$. if l has the form $\neg v$ then $a(v) = \texttt{f}$.
2. For any literal l where we've established $l = \texttt{t}$, remove $\neg l$ from all clauses in which it appears. If this makes a clause empty then $F = \texttt{f}$—it is *unsatisfiable*. Remove any clause containing l.
3. *Constraint propagation:* Step 2 may lead to new clauses of the form of Step 1. If so, we may have established truth values for new literals. Continue the process for as long as possible.
4. Choose a variable v on which to *split* the problem. Try setting $v = \texttt{t}$ and $v = \texttt{f}$, and applying the algorithm for each case.

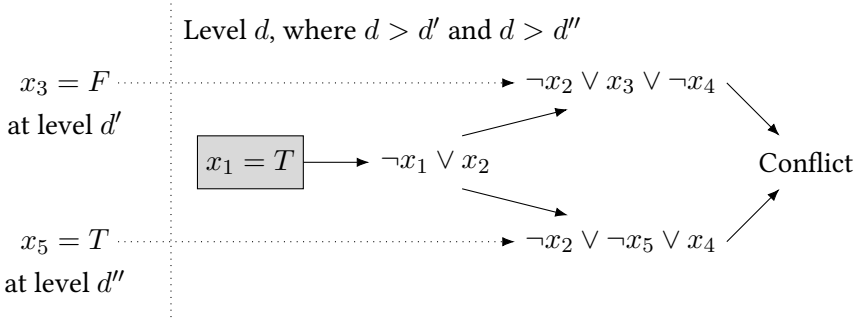
CDCL Solvers: Making DPLL Work

Conflict-driven clause learning (CDCL) solvers use numerous techniques to improve on DPLL:

- Better constraint propagation—the *two watched literals* method.
- Activity-based variable ordering.
- Clause learning and backjumping.
- Clause forgetting.
- Restarting.
- And others…

Today I will mostly be talking about *variable activity*.

CDCL Solvers: Clause Learning and Backjumping



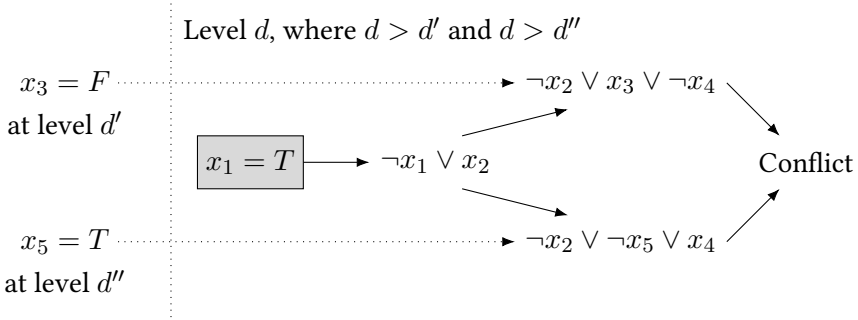
On finding a *conflict*, we can add one or more *learned clauses* to the original problem.

Why would we want to do this?

1. They can force later unit propagations.
2. In analyzing the conflict, we can establish *backjumps*.
3. Use of literals or clauses in analyzing the conflict leads to good *heuristics*.

The use of the term *learned* here is distinct from *machine learning*.

CDCL Solvers: Clause Learning and Backjumping



Example:

- *Resolve* the clauses causing the *conflict*:

$$\neg x_2 \vee x_3 \vee \neg x_5.$$

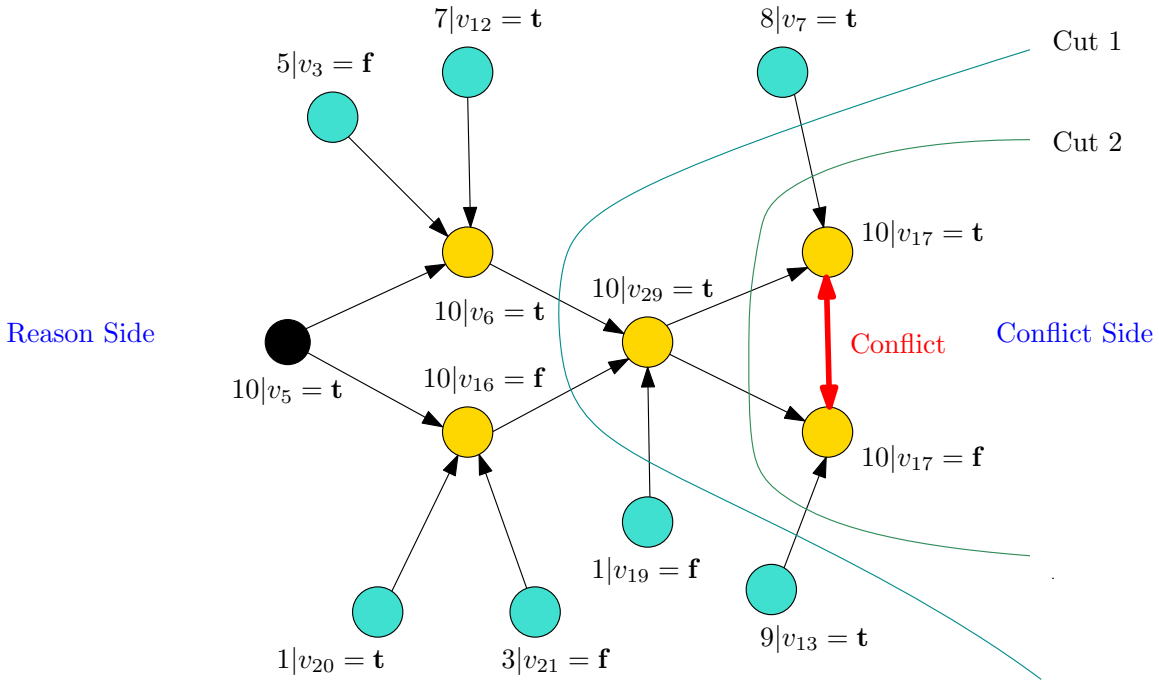
- *Equivalently*

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

- Backjump to the *most recent* of level d' or d'' and assert $\neg x_2$.

CDCL Solvers: Clause Learning and Backjumping

More generally:



Cut the *implication graph* to infer different *learned clauses*.

CDCL Solvers: Activity-Based Variable Ordering

Variable State Independent Decaying Sum (VSIDS) forms the basis for many current variable choice heuristics. The essentially idea is as follows:

- Measure a variable's importance by how often it has been *used to learn a clause*.
- Allow such activity measures to *decay over time*.

Example: EVSIDS updates variable activities at each conflict:

- For each variable v *appearing in a learned clause*

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

- To *decay activities* over time

$$u \leftarrow u \times u' \text{ where typically } 1.01 < u' < 1.2.$$

Various methods for choosing polarity, for example always `f`, or last polarity used.

Machine Learning Highlight 1

Lightweight Methods for *Per-Problem Learning* of *Variable Activities*

Multi-armed bandits

Learning by *interaction with the environment*.

I have n 1-*armed bandits*. Payouts have *unknown distributions*.

I am allowed to make m plays, and I aim to *maximize my winnings*.

- If I knew the distributions of payouts for the bandits this would be straightforward.
- As I don't, I have to *explore* the outcomes by trying different arms.
- If one gives me a good payout, perhaps I will *exploit* it.

What if the payout distributions *change over time*? I should place *greater value* on *recent payouts*. If a machine gives payouts $r_1, r_2, \ldots, r_p$ then estimate the *exponential recency weighted average (ERWA)*

$$\begin{aligned}\hat{r}_p &= \sum_{i=1^p}(1-\alpha)^{p-i}\alpha r_i \\ &= (1-\alpha)\hat{r}_{p-1} + \alpha r_p.\end{aligned}$$

Algorithms such as the *Discounted Upper Confidence Bound (UCB)* algorithm provide ways of choosing arms to play, depending on the characteristics of the problem, with differing *performance guarantees.*

Multi-armed bandits for improving VSIDS I: the CHB heuristic

Several applications of ML have employed *multi-armed bandits*, in particular the *ERWA* algorithm, in attempts to improve on VSIDS.

The *Conflict History Based (CHB) heuristic* attempts to reward variables that *lead to many learned clauses*.

- We have a v-*armed bandit*, that learns variable activities $a(v)$.
- $a(v)$ is updated whenever v is:
  - Chosen to branch on.
  - Propagated during unit propagation.
  - Asserted due to clause learning.
- The usual update is applied:

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

  but α is reduced over time.
- The reward r_v is carefully chosen to reward variables *recently used to learn a clause*

Multi-armed bandits for improving VSIDS I: the CHB heuristic

For the CHB heuristic, let:

- $\mathcal{C}$ be the number of conflicts seen so far.
- $\mathcal{C}(v)$ be the number of conflicts when v was last used to learn a clause.
- $\beta = 1$ if using v lead to a conflict or 0.9 otherwise.

Then

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

Multi-armed bandits for improving VSIDS II: the LRB heuristic

The *Learning Rate Branching (LRB)* heuristic extends CHB.

It attempts to *explicitly optimize the rate of generation of conflict clauses*.

It differs from CHB in the *definition of the reward* r_v.

- Let $\mathcal{I}$ be an interval starting when v is assigned and ending when v is unsassigned during backtracking.
- Say that v *participates in generating* C if $v \in C$ or v is resolved when generating C.
- Let $p(v, \mathcal{I})$ be the number of clauses that v participates in generating during $\mathcal{I}$.
- Let $L(\mathcal{I})$ be the number of clauses learned during $\mathcal{I}$.

Then the *learning rate* is

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

And we need some more definitions...

Multi-armed bandits for improving VSIDS II: the LRB heuristic

More definitions:

- Say that v *reasons in learning* C if it is not in C but is *in a reason clause for a variable in* C.
- Let $q(v, \mathcal{I})$ be the number of clauses v reasons in generating during $\mathcal{I}$.

Then

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

and the *reward* is

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

Finally:

- A play is made with reward r_v when v is *unassigned*.
- In order to *exploit community structure*, at each conflict the activities of unassigned variables are scaled by 0.95.

Improving VSIDS III: the GLR heuristic

The *number of conflicts per decision made by the solver* is known as the *Global Learning Rate (GLR)*.

High GLR is known to *correlate with short run-time*.

Favor variables *leading to a conflict if chosen*: better than VSIDS but *too expensive to compute*.

Let

- $\mathcal{P}$ be the set of *partial assignments*.
- $f : \mathcal{P} \to \{0, 1\}$ be a function taking value 1 if its argument *leads to a conflict when propagated* and 0 otherwise.

The central aim is to *learn a function*

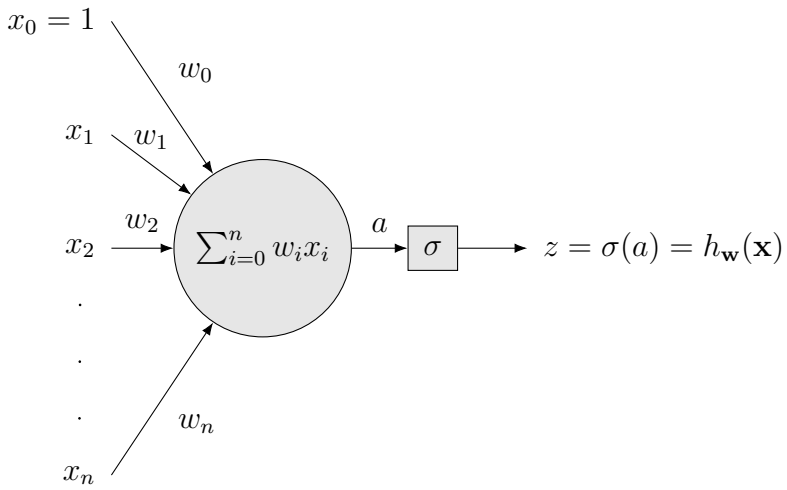
$$\hat{f} : \mathbb{R}^{|V|} \to [0, 1]$$

that approximates f. The features are simply

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

Linear Regression: A Very Quick Reminder

We can solve simple learning problems—like learning $\hat{f}$—using a *perceptron*:



Select the *weights* w_i to minimize some measure of *error* $E(\mathbf{w})$ on some *training examples*.

Linear Regression: A Very Quick Reminder

If we use a simple function $\sigma(x) = x$ then this is *extremely straightforward*.

Using for example

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - \mathbf{w}^T\mathbf{x}_i)^2.$$

If we can find the *gradient* $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ of $E(\mathbf{w})$ then we can minimize the error using *gradient descent*

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \left.\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}_t}.$$

To learn $\hat{f}$ we want an output between 0 and 1:

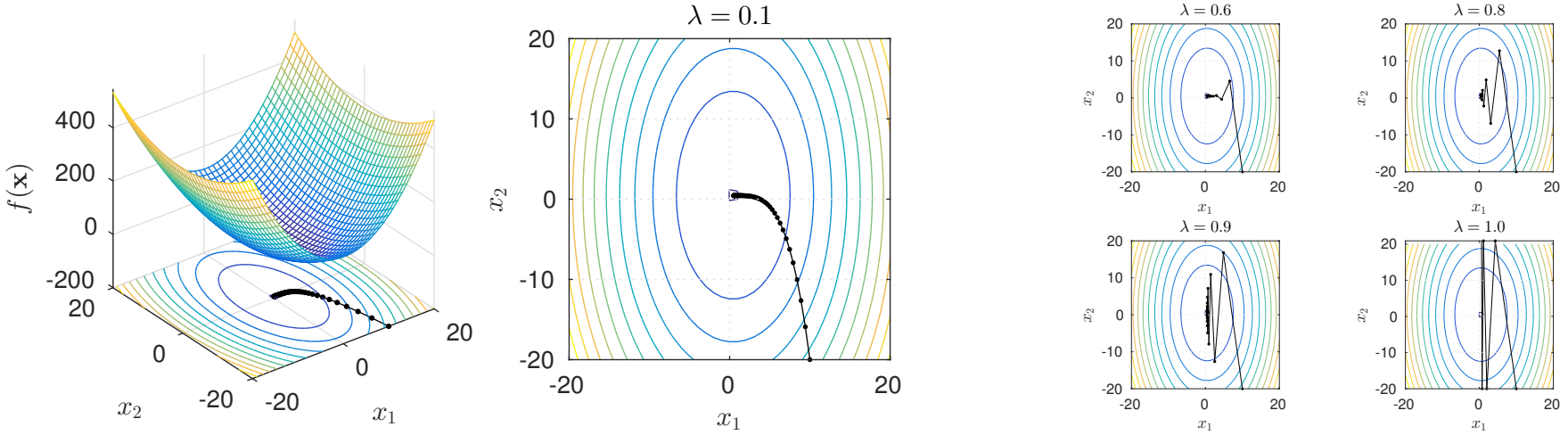
- *Regularized linear logistic regression* uses a function such as

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

- We use a different $E(\mathbf{w})$.
- The details are similarly straightforward.

Linear Regression: A Very Quick Reminder

Gradient descent: the *simplest possible method* for minimizing such functions:



Take *small steps downhill* until you reach the minimum.

But remember: in more general cases *there might be many minima*— some *local* and some *global*.

The *step size* matters.

Improving VSIDS III: the GLR heuristic

This is achieved using *regularized linear-logistic regression*.

The training examples are generated in *pairs* at each conflict:

- A positive example containing the *negation of the learned clause* and the *conflict-side literals*.
- A negative example containing literals in the *current partial assignment*, except those at the *current level* or *in the positive example*.
- There are some *variations* on this.

A *single step of gradient descent* is applied for each example.

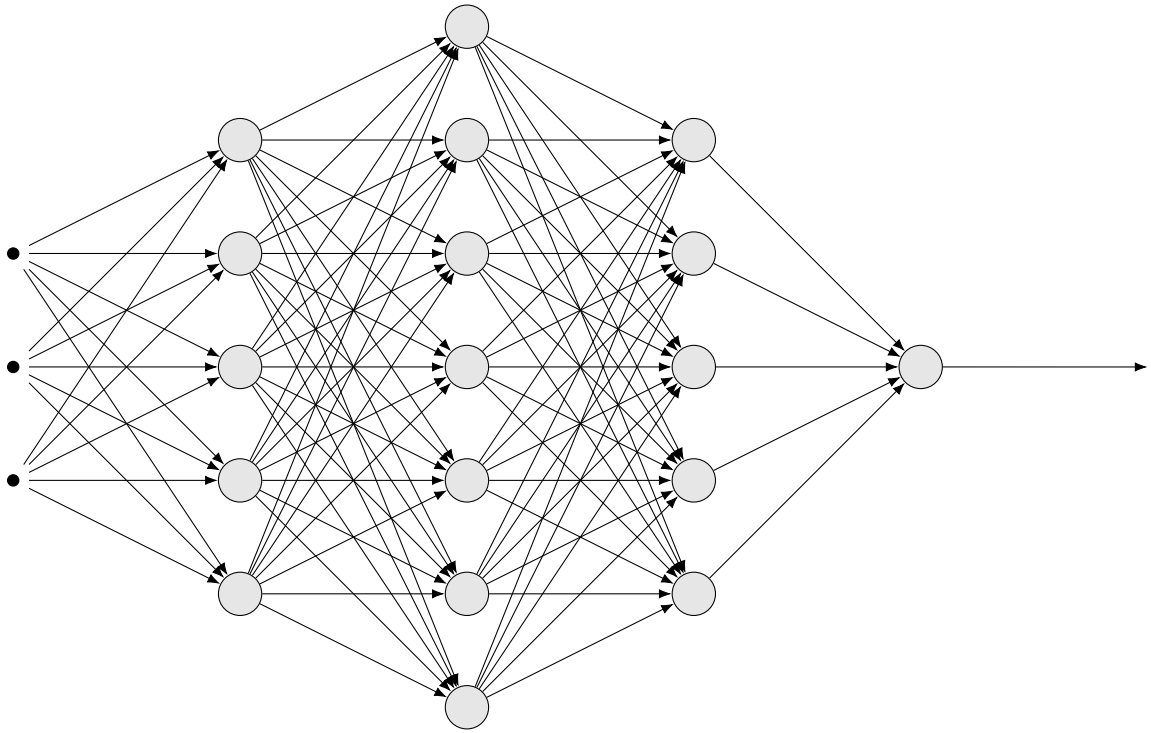
- This is *efficient*.
- Allows learned $\hat{f}$ to *adapt* as clauses are learned.

Machine Learning Highlight 2

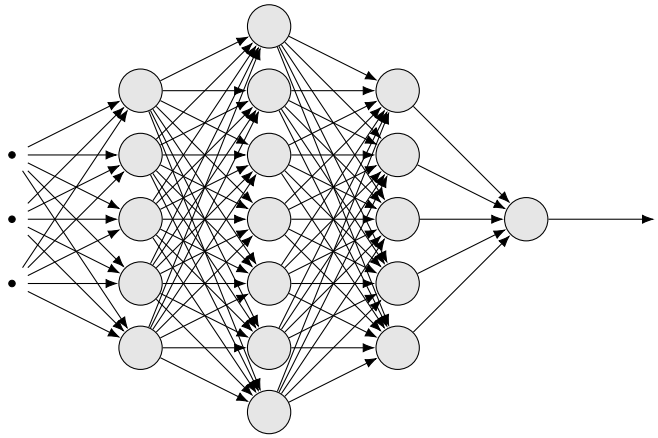
Heavyweight Methods for Learning *Variable Activities*

Real problems tend also to be *nonlinear*.

We can combine perceptrons to make a *multilayer perceptron*:



Here, each *node* is a perceptron and each *edge* has a weight attached.



- The network computes a function $h_{\mathbf{w}}(\mathbf{x})$.
- The trick remains the same: minimize an error $E(\mathbf{w})$.
- We do that by *gradient descent*

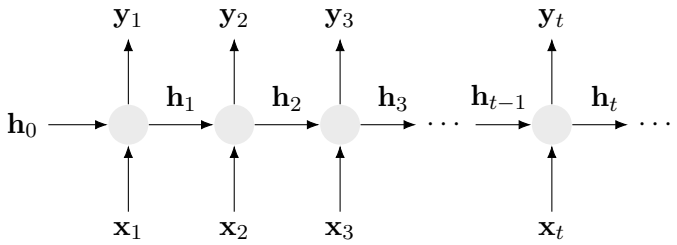
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \lambda \left. \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \right|_{\mathbf{w}_t}$$

- This can be achieved using *backpropagation*.
- Backpropagation is *just* a method for computing $\partial E(\mathbf{w})/\partial \mathbf{w}$.

Other kinds of NN computation: recurrent networks

Sometimes, we might want to classify *sequences* $\mathbf{x}_1, \mathbf{x}_2, \ldots$.

An output can be produced after T steps, or at every step.



The vector $\mathbf{h}_t$ denotes a *state* at time t.

- Updates are computed as follows:

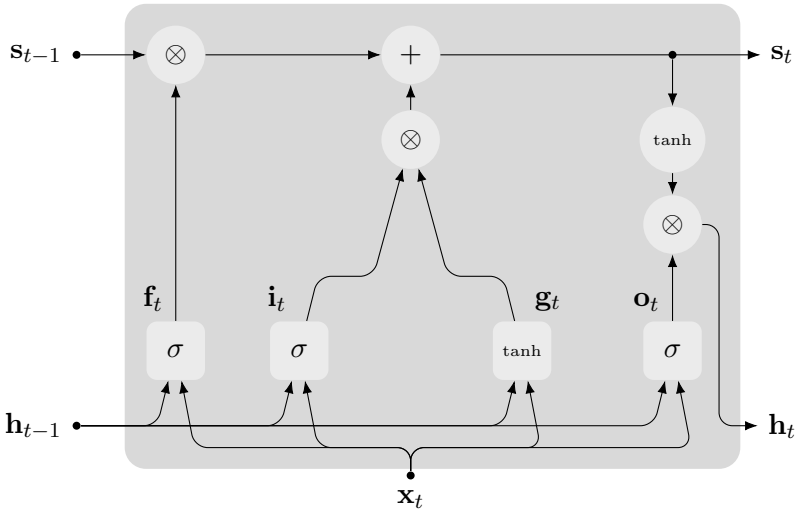
$$\begin{aligned}\mathbf{s}_t &= \mathbf{b}_s + \mathbf{s}_1\mathbf{h}_{t-1} + \mathbf{s}_2\mathbf{x}_t \\ \mathbf{h}_t &= \tanh(\mathbf{s}_t) \\ \mathbf{y}_t &= \mathbf{b}_y + \mathbf{Y}\mathbf{h}_t\end{aligned}$$

- *Learned parameters* $\mathbf{b}_s$, $\mathbf{s}_1$, $\mathbf{s}_2$, $\mathbf{b}_y$ and $\mathbf{Y}$ are the same at each step.

Other kinds of NN computation: LSTMs

When learning recurrent networks, gradients can *decay or grow exponentially*.

Long Short Term Memories (LSTMs) aim to solve this problem, and have other advantages.



This structure is *repeated at each time step*.

We now have two kinds of state: $\mathbf{s}_t$ denoting *long-term*, and $\mathbf{h}_t$ *short-term* state.

$\mathbf{h}_t$ also acts as the *output*.

The elephant(s) in the room: system design and hyperparameter choice

Machine Learning and AI have a *little secret...*

...nothing is really automatic!

- *Human input* is still required, particularly in *designing architectures*.
- Any architecture will generally have numerous *hyperparameters*, such as the *number of layers*, number of *nodes per layer*, *learning rate* and so on, and...
- ...*good performance depends critically* on setting these correctly.

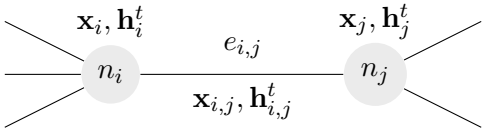
There are notable efforts aimed at increasing the level of automation, but for now do not underestimate the need for *blood, sweat and tears*!

Learning features from examples: graph neural networks

Another recent development is the use of *Graph Neural Networks (GNNs)* to *learn features from examples*.

This is an *alternative* to designing a *representation* of a SAT problem *by hand*.

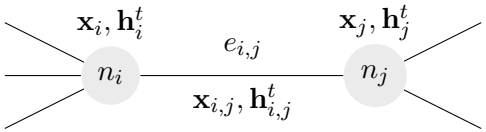
There are numerous variants of the GNN. Many employ some form of *message passing*:



- $\mathbf{x}_i$, $\mathbf{x}_j$, $\mathbf{x}_{i,j}$ are node and edge features.
- $\mathbf{h}_i^t$, $\mathbf{h}_j^t$, $\mathbf{h}_{i,j}^t$ represent *hidden state* at time t.
- Update over T steps using

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

Learning features from examples: graph neural networks



- The features are then

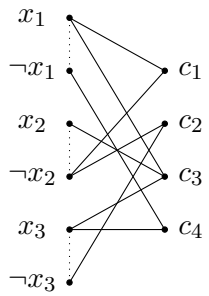
$$\mathbf{y} = R(\mathbf{h}_1^T, \ldots, \mathbf{h}_N^T).$$

- The functions M_t, U_t and R are typically parameterized and learned.

Aside from removing much (not all!) of the need for manual intervention, these can insure *invariance to isomorphism, symmetry (for example clause re-ordering) and so on*.

Learning features from examples: graph neural networks

Example: [Selsam et al., 2019]. Represent a CNF formula F with c clauses and v variables as follows:



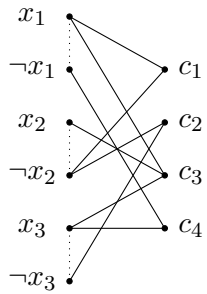
- Run for T steps. At each step

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

- Four functions are learned: L and C are MLPs and L' and C' are LSTMs with hidden states $\mathbf{L}_h^t$ and $\mathbf{C}_h^t$.
- $A_{i,j}$ is 1 if clause j contains literal i and 0 otherwise. f swaps a *row for a literal* with the *row for its negation*.
- The resulting $\mathbf{C}^T$ and $\mathbf{L}^T$ have rows representing *clauses and literals respectively*.

Revisiting the problem: NeuroSAT

NeuroSAT: represent a CNF formula F with c clauses and v variables as follows:



After some processing, we get $\mathbf{C} \in \mathbb{R}^{2v \times d}$ and $\mathbf{L} \in \mathbb{R}^{c \times d}$. These have rows representing *clauses and literals respectively*, each *embedded* as a vector in $\mathbb{R}^d$.

- Having obtained the literal embeddings, introduce an MLP that computes *votes for each literal*

$$\mathbf{v} = V(\mathbf{L}) \in \mathbb{R}^{2v}.$$

- The *output* of the system is obtained by *thresholding* the *average of the votes*.
- The *entire architecture* is trained to make the output match the single bit label for a training set of SAT/UNSAT problems.

GNNs for variable selection

Several systems have tried to extend *NeuroSAT* to learn to select variables.

Extension I: NeuroCore [Selsam and Bjørner, 2019] combines a *simplified version* of the NeuroSAT architecture with the standard *EVSIDS* heuristic:

- An *UNSAT Core (UC)* is a set of clauses known to be unsatisfiable.
- *Intuition:* if you know a variable is in a UC *then branch on it* as it's *likely to lead to a conflict*.
- Training set has only *unsatisfiable problems*.
- Analyze proofs to *find variables in UCs*.
- The system then learns the function $\text{core}(v_i)$ which predicts the likelihood that v_i is in a UC.
- Runs CDCL as usual using EVSIDS. Activity is $E(v_i)$.

Periodically *recompute activities*:

$$E(v_i) = |V|k + \text{softmax}\left(\frac{\text{core}(v_i)}{\tau}\right).$$

Kinds of learning: reinforcement learning

What if we want to learn from *rewards* rather than *labels*?

Reinforcement learning works as follows.

1. We are in a *state* and can perform an *action*.
2. When an *action* is performed we move to a *new state* and receive a *reward*. (Possibly *zero* or *negative*.)
3. New states and rewards can be *uncertain*.
4. We have *no knowledge in advance* of how actions affect either the new state or the reward.
5. We want to learn a *policy*. This tells us what action to perform in any state.
6. We want to learn a policy that in some sense *maximizes reward obtained over time*.

Typically, measure reward over time as

$$\begin{aligned} R_i &= r_i + \gamma r_{i+1} + \gamma^2 r_{i+2} + \cdots \\ &= \sum_{j=0}^{\infty} \gamma^j r_{i+j}. \end{aligned}$$

Note that this can be regarded as a form of *planning*.

GNNs for variable selection

Extension II: NeuroGlue [Han, 2020] takes a similar approach.

Literal Blocks Distance (LBD): number of *variable levels* in a *clause*.

Glue Variables: variables *in conflict clauses* with *LBD at most* 2.

This work considers a *reinforcement learning* approach.

- *States* are the $F(t)$ for trail t. (The problem *after* assignments t and resulting propagations.)
- *Actions* are taken by *choosing a variable to assign*.
- *State transition* from a state $F(t)$ corresponds to choosing polarity at random, adding the chosen assignment to the trail forming t', and moving to $F(t')$.
- *Rewards* are 0 for a proof of satisfiability, $1/G^2$ for finding a conflict, where G is the LBD of the conflict clause, and $-1/V$ where V is the number of variables in $F(t')$ otherwise.

This is *not the only approach* attempting to introduce reinforcement learning...

GNNs for variable selection

A similar system is *GQSAT* [Kurin et al., 2019]. In this system the states and transitions are similar, but:

- Actions choose *both variable and polarity*.
- Rewards are 0 if a solution is found and $-r$ otherwise.

A *GNN-based* approach is used to learn $Q_\theta(s, a)$, which is the *expected reward* if *action* a is taken in *state* s and *behavior thereafter is optimal*.

Having learned this function, a *policy* for *action choice* is

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

This turns out to be *demanding to compute*.

It is used only in the *initial part* of a CDCL run. After this, EVSIDS is used. (Another example of learning *variable initialization*.)

Once again, some evidence of *generalization* is noted.

Take-Home Messages

Take home messages

1. *Methods*: *LOTS* of things have been tried. Essentially *every usable element* of a CDCL solver has been a basis for adding ML, and *most of the tools in the ML toolbox* have been applied...
2. ...*BUT*, something not noted until now is that a worrying large body of research makes *little if any attempt to optimize hyperparameters*. One wonders *how much has been lost*!
3. *Lightweight versus heavyweight methods*:
   - When you put machine learning *inside the proof search* it has to be fast.
   - When *more demanding* methods are applied here, it's not unusual to make *better decisions* but *slow down* the process.
   - Deep methods have *analogous problems* when deployed to, for example, *mobile devices*.
4. Be aware of *what* you're learning to do: *per-problem* learning, versus learning to solve a *class of problems*.

References

[Babić and Hu, 2007] Babić, D. and Hu, A. J. (2007). Structural abstraction of software verification conditions. In Damm, W. and Hermanns, H., editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*, number 4590 in Lecture Notes in Computer Science, pages 366–378. Springer.

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without BDDs. In Cleaveland, W. R., editor, *Proceedings of the 5th International Conference on Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 197–207. Springer.

[Clarke et al., 2001] Clarke, E., Biere, A., Raimi, R., and Zhu, Y. (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34.

[Fréchette et al., 2016] Fréchette, A., Newman, N., and Leyton-Brown, K. (2016). Solving the station repacking problem. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 702–709.

[Graça et al., 2010] Graça, A., ao Marques-Silva, J., and Lynce, I. (2010). Haplotype inference using propositional satisfiability. In Bruni, R., editor, *Mathematical Approaches to Polymer Sequence Analysis and Related Problems*, pages 127–147. Springer.

[Han, 2020] Han, J. M. (2020). Enhancing SAT solvers with glue variable predictions. arXiv:2007.02559v1 [cs.LO].

[Kautz and Selman, 1992] Kautz, H. and Selman, B. (1992). Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI)*, pages 359–363. Wiley.

[Kautz, 2006] Kautz, H. A. (2006). Deconstructing planning as satisfiability. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, volume 2, pages 1524–1526.

[Kurin et al., 2019] Kurin, V., Godil, S., Whiteson, S., and Catanzaro, B. (2019). Improving SAT solver heuristics with graph networks and reinforcement learning. arXiv:1909.11830v1 [cs.LG].

[Lynce and ao Marques-Silva, 2006] Lynce, I. and ao Marques-Silva, J. (2006). Efficient haplotype inference with Boolean satisfiability. In *Proceedings of the 21st AAAI Conference on Artificial Intelligence*, volume 1, pages 104–109. The AAAI Press.

[Selsam and Bjørner, 2019] Selsam, D. and Bjørner, N. (2019). Guiding high-performance SAT solvers with unsat-core predictions. In Janota, M. and Lynce, I., editors, *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing (SAT)*, number 11628 in Lecture Notes in Computer Science, pages 336–353. Springer.

[Selsam et al., 2019] Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. (2019). Learning a SAT solver from single-bit supervision. arXiv:1802.03685 [cs.AI].