LEARNING SEARCH GUIDANCE FROM SOLVER-GENERATED GROUND TRUTH

Anonymous authors
Paper under double-blind review

ABSTRACT

AlphaZero made strides in the field of game playing not just because it trained via self-play, but also because it learned *tabula rasa*, i.e., from a random initialization on self-generated data. That learning philosophy has not yet made its way into the field of game solving, which requires deriving the game-theoretic value of the initial position rather than merely selecting a good move. In this paper, we apply AlphaZero's learning philosophy in the context of game solving to introduce AlphaSolve, a game solver that learns search guidance from the game-theoretic values of intermediate nodes it proves while solving a game. We verify it on previously solved games: Connect Four and three Breakthrough variants. Programs for solving games are evaluated on their efficiency, which is determined by how well they search the game tree for a proof of the game-theoretic value. Prior art has relied on estimating proof cost (the number of positions required to solve a node), but we show that on Connect Four training on win/loss outcomes (as AlphaZero does) is competitive with training on the measured proof cost as the value target. AlphaSolve generates roughly a hundred times more ground truth labels than self-play game trajectories, and on Connect Four we find that a small share of trajectories mixed into the verdicts is better than either stream alone. This is significant because reinforcement learning algorithms are notorious for investing heavily in data generation, but AlphaSolve generates ground truth labels in abundance.

1 INTRODUCTION

Game AI pursues two goals. Playing a game well means choosing strong moves under time pressure, and the dominant modern approach is reinforcement learning from self-play, of which the AlphaZero lineage is the most notable (Silver et al., 2018). Solving a game means determining the game-theoretic value of its initial position with certainty.[1] The practical difference between the two comes from the stopping criterion: a playing system stops once the selected move is probably best, while a solver stops only once the root's true value has been determined. That criterion also determines what each is judged on. A playing system is evaluated by how well it plays, and the cost of training is secondary; a solver is evaluated on how efficiently it produces a proof.

One of the key innovations in AlphaGo (Silver et al., 2016) was the choice of what to learn from. Where earlier programs evaluated positions with hand-crafted heuristics, AlphaGo's value network was trained simply to predict the outcome of the game from a given position. This choice is what allowed AlphaGo Zero (Silver et al., 2017) to learn *tabula rasa* because the system can learn from the data it generates itself. However, that same idea has not been applied to game solving; game solvers do learn (Gao et al., 2017; Wu et al., 2022) and can even be fine-tuned online (Wu et al., 2023), but their training data still comes from estimated labels rather than measured ones.

Game solving is special in that solving a node requires first solving many of the nodes beneath it. Each of those nodes is a ground truth verdict the solver generates itself, and this paper explores how much the solver can leverage this as training data. A solver solves positions much faster than it plays

[1] This work concerns weak solutions, which also provide a strategy for achieving the optimal outcome. A strong solution gives optimal play from every legal position, and an ultra-weak solution gives the value of the initial position without a strategy.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

games, every position it solves is an exact label, and we show that it can reliably learn from those exact labels to increase its efficiency. This is in contrast to learning from self-play games, where the training data is expensive to produce. For example, AlphaZero devoted five thousand first-generation TPUs to generating games versus sixteen second-generation TPUs to training on them (Silver et al., 2018). A solver could buy its data the same way, but it is judged on every unit of compute it spends, and it has a potential source it is already paying for.

We introduce AlphaSolve, a weak solver that learns its guidance from the solve itself. It starts off with AlphaZero's data generation and training loop: a policy and value network guides Monte Carlo tree search (Coulom, 2006; Kocsis & Szepesvári, 2006) through self-play episodes, and finished episodes are training examples. It is then augmented with the bookkeeping of an MCTS solver (Winands et al., 2008) and with it a second stream of training data. Whenever a node is solved, its verdict also becomes a training example: an exact win/loss label for the value head and a policy target built from the proof rather than from visit counts. The network begins at random initialization, trains concurrently with the solve, and only trains on self-generated data. AlphaSolve is the first weak solver whose guidance is learned this way, and it can be applied to any proof search over an AND/OR tree.

For our ablations, we test on games small enough to solve outright and large enough that guidance matters. We demonstrate our program by reproducing the verdicts on previously solved games: the standard 7×6 Connect Four board (Allen, 1989; Allis, 1988) and Breakthrough on the 5×5, 6×5, and 5×6 boards (Saffidine et al., 2011; Considine, 2025).[2] We report the number of simulations and unique nodes visited so readers can evaluate the efficiency of our solver. As a control, the same search with the network frozen at its random initialization solves 5×5 Breakthrough in 2 hours 26 minutes whereas learning during the solve brings that down to 23 minutes.

Three contributions follow.

1. Solver-emitted ground truth is enough to learn guidance *tabula rasa* (§4.1).
   (a) A network trained from random initialization, during the solve, on the verdicts the solve produces, weakly solves Connect Four and three Breakthrough boards.
2. Win/loss outcome is competitive with proof cost (Wu et al., 2022) as the value target (§4.2).
   (a) Trained online, the two targets solve in comparable time; here the proof cost label is the measured cost of proving the node, not an estimate.
3. Solver-emitted ground truth outnumbers self-play trajectories by two orders of magnitude, and a mixture of the two streams beats either alone (§4.3).
   (a) Weighting the streams at $f = 0.01$ solves 19.9% faster than ground truth alone.
   (b) At the rate of games that the solve plays, training on self-play trajectories alone does not finish within an eight-hour budget.

2 Background and Related Work

Weak solving. Weakly solving a two-player game means determining the game-theoretic value of its initial position and providing a strategy for achieving it. The literature is surveyed by van den Herik et al. (2002); its emblematic result is checkers (Schaeffer et al., 2007). Connect Four was weakly solved twice in 1988, by brute-force search with a large opening database (Allen, 1989) and by a knowledge-based approach (Allis, 1988). Breakthrough boards up to 6×5 were settled by hand-designed race patterns with job-level proof-number search (Saffidine et al., 2011), and 5×6, which had resisted search, was strongly solved by compressed retrograde analysis (Considine, 2025); 6×6 has since been solved as well (Björnsson et al., 2026). These works provide the verdicts our solves are checked against.

Solving as search over an AND/OR tree. One method of weak solving is via a search over an AND/OR tree. At a position where the first player is to move, an OR node, one winning move suffices; at a position where the second player is to move, an AND node, every reply must be answered. What

[2] Board sizes are written columns $\times$ rows throughout, following the convention of the solving literature.

the search produces is a solution tree (Stockman, 1979; Pijls & de Bruin, 2001).[3] The same machinery is not specific to games and has been carried to synthesis planning (Kishimoto et al., 2019) and theorem proving (Lample et al., 2022).

The optimal node to expand. A solver's efficiency is decided by which node it expands next. Let T be the search tree built so far and $L \subseteq T$ the frontier of unexpanded leaves. The optimal strategy minimizes the expected number of expansions still needed to solve the root (Smith, 1989):

$$C(T) = \begin{cases} 0 & \text{the root is solved in } T, \\ 1 + \min_{\ell \in L} \mathbb{E}\left[\, C(T') \mid T, \ell \,\right] & \text{otherwise,} \end{cases} \tag{1}$$

where T' is the tree after expanding ℓ, and the optimal node is the one that results in the fewest expected remaining expansions. The game is deterministic, so the expectation is over the solver's uncertainty about what an expansion will reveal: whether ℓ can be solved, and how large the tree beneath it is. No solver can select the optimal node, as discovering the shape of the game tree requires search, and the shape is only revealed as the solve progresses (Russell & Wefald, 1991). Instead, each selects ℓ according to a heuristic: proof-number search counts every unexpanded leaf as unit cost (Allis et al., 1994), MCTS solvers follow an estimated outcome (Winands et al., 2008), and Proof Cost Networks learn an estimate of the minimum proof tree size (Wu et al., 2022).

Parallel search. The strongest parallel solvers are hierarchical: a manager performs its own search and hands positions to workers as jobs (Wu et al., 2013). PN^2 runs one level of proof-number search above another (Breuker et al., 2001), and its parallel form has the workers report proof and disproof numbers back to guide the upper level (Saffidine et al., 2011). The online fine-tuning solver of Wu et al. (2023) keeps the same shape, with a manager that assigns positions and budgets and workers that return either a verdict or a heuristic value. The manager's job in each case is to decide which subproblems are worth a worker's time.

What a learned value head is asked to predict. Networks trained for strong play make imperfect solving heuristics. They are indifferent among winning moves, preferring a certain win to a fast one (Agostinelli et al., 2019; Wu et al., 2022), and they are brittle out of distribution (Lan et al., 2022). Proof Cost Networks (PCN) (Wu et al., 2022) answer the first of these by changing the training target: the value head predicts a logarithmically normalized estimate of proof tree size rather than a win rate, and the authors found it outperforms an AlphaZero network as a heuristic in both MCTS-based and df-pn solvers on 15×15 Gomoku and 9×9 Killall-Go. PCN defines its target as the size of a minimum proof tree, which it estimates from self-play, and notes that unlike a proof number the estimate includes the tree already searched beneath a node. Wu et al. (2023) hardcode the proof cost of solved nodes to 0; the online fine-tuning is intended to steer the network toward previously solved nodes.

Learning during the solve. Learned guidance for solvers has been trained elsewhere and imported: on games between strong programs (Gao et al., 2017), or by a separate run of self-play before the proof search begins (Wu et al., 2022). Wu et al. (2023) are the first to train during the solve. Their motivation is the observation that most positions a proof must verify lie far from the self-play distribution, and their answer is to fine-tune a Proof Cost Network online while the proof search runs, on a suite of 7×7 Killall-Go openings. Their network starts from roughly 52 GPU-hours of AlphaZero-style self-play, and its fine-tuning data is drawn 90% from self-play against 10% from solved positions, with the solved positions labeled at cost zero. AlphaSolve uses self-play as its search strategy and in §4.3 more than inverts that split: solved positions are the majority stream and self-play trajectories the minority. To our knowledge it is the first weak solver whose guidance is learned this way.

3 METHOD

AlphaSolve is a loop with four steps (Figure 1). A search solves a node. The solved node becomes a training example. The training example updates the network. The updated network guides the next

[3] For a proven root, one child at each OR node and every child at each AND node. A disproven root reverses it: every child at each OR node, and one child at each AND node.

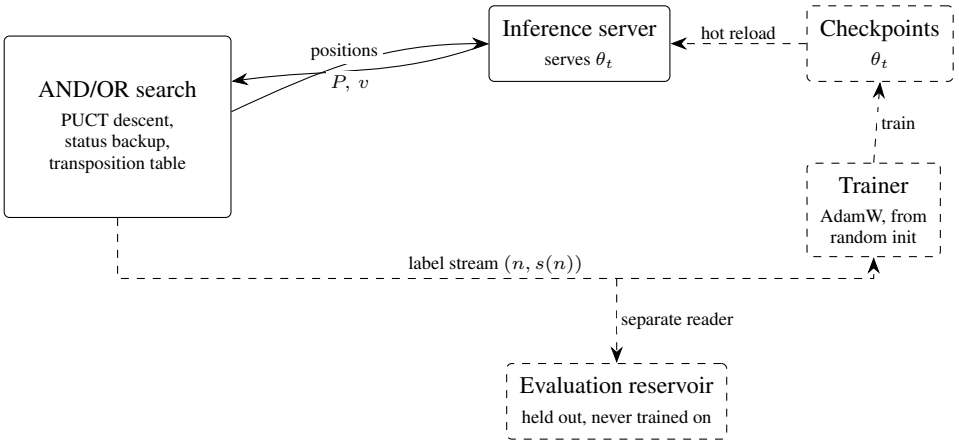


Figure 1: The AlphaSolve system. Solid components are present in every run. Dashed components – the label stream, trainer, periodic checkpoints with hot reload, and the held-out evaluation reservoir – are what the untrained control of Table 1 removes; with those gone, one fixed network serves the entire solve.

search. The search carries no game knowledge beyond the rules: no patterns, no move ordering, no endgame tables.

3.1 SEARCH

The search is structured using self-play games. Starting from the initial position, each worker performs a Monte Carlo tree search (Coulom, 2006; Kocsis & Szepesvári, 2006) guided by the network for up to 1,000 simulations (or until the root is solved), plays a move, and continues from the resulting position, reusing the subtree beneath the move it played. The tree lives for one game and is discarded when the game ends, which happens once the root has been solved. Every node carries a status (Proven, Disproven or Unknown) with respect to the proposition that the first player wins from that position. Appendix C gives the search in full.

Selection. Each simulation descends the tree from the root with the PUCT rule of AlphaZero (Silver et al., 2018). Each edge (s, a) stores a prior $P(s, a)$, a visit count $N(s, a)$ and an action value $Q(s, a)$, and the simulation selects the move that maximizes $Q(s, a) + U(s, a)$, where $U(s, a) = c_{\text{puct}}\, P(s, a)\sqrt{N(s)}/(1 + N(s, a))$. At the root, Dirichlet noise is mixed into the priors.

Once a child is solved, selection stops visiting it. At an OR node a Disproven child means the first player cannot win if they play that move, and at an AND node a Proven child means the second player cannot prevent the win if they play that move; neither requires any further search. The consequence is that every simulation lands on the unsolved frontier, which is what makes a solve finish. It also changes what visit counts mean. In AlphaZero the most-visited move is the best one; here it is the least resolved one, since the moves that have been settled are no longer being visited. The effect is that the search is steered away from solved nodes and toward yet-unsolved positions.

Expansion and evaluation. When the simulation reaches a leaf, it is expanded with a child for each legal move, the network evaluates it, $(p, v) = f_\theta(s)$, the children take their priors from p, and v is backed up along the path. Under the win/loss target of §3.3, the network's value $v \in [0, 1]$ is the probability that the first player wins. The network is used purely for search guidance and cannot decide a node's proof status. A search would solve the game with any network; a network with poor guidance only costs time but cannot yield an incorrect solve.

Backpropagation. A node backs up a value for each side: $Q(s, a)$ is the mean of the OR side at an OR node and of the AND side at an AND node, so each side maximizes its own chance of winning. Under the win/loss target the two sides are v and $1 - v$. A node is Proven or Disproven by the rules when it is terminal. Otherwise its status is propagated from its children: a node where the first player

is to move is an OR node, Proven if any child is Proven and Disproven if every child is; a node where the second player is to move is an AND node, Proven if every child is Proven and Disproven if any child is. Propagation runs up the tree each time a status changes, and the solve is finished when the root's status is no longer Unknown.

Parallelism. AlphaSolve uses root parallelization, where each worker gets its own tree, rather than tree parallelization, where workers parallelize across the same tree. A shared tree allows the workers to coordinate at the cost of contention; separate trees have no contention at the cost of additional memory and duplicate work. We found that the cost of contention far outweighed the cost of duplicate work in order to solve a game faster.

AlphaSolve uses a single-layered proof search rather than the manager-and-worker designs described in §2, so every worker runs the same search and consults the same network. This is a design choice; we make no claim that it is faster than those designs.

3.2 Data

A run produces two streams of training examples from the same games.

Trajectory stream. Every position of a finished game is recorded with the game's result and the visit counts at that position. Solved children are excluded from the visit count distribution, which effectively fine-tunes the network to steer toward unsolved children. Because a game always ends when its root is solved, the "result" is that root's verdict. Since that final position was solved via propagation, its search was cut short so we do not include it in this stream. Instead, it is included in the second stream.

Solved stream. Whenever propagation solves a node, that node is emitted with its verdict and proof cost. Nodes solved by the rules (the terminal positions) are not emitted. We calculate the proof cost as the size of the solution tree the search found beneath the node: one child at each OR node and every child at each AND node, and symmetrically for a disproof (Stockman, 1979; Pijls & de Bruin, 2001). Calculating the solution tree size this way counts a position once for every path that reaches it.

A solved node's verdict is the position's game-theoretic value, which satisfies the minimax form of the Bellman optimality equation (Sutton & Barto, 2018; Littman, 1994):

$$v^\star(s) = \begin{cases} z(s) & s \text{ terminal}, \\ \max_{a \in A(s)} v^\star(s') & s \text{ an OR node}, \\ \min_{a \in A(s)} v^\star(s') & s \text{ an AND node}, \end{cases} \tag{2}$$

where s' is the position reached by playing a in s, and $z(s) = 1$ if the first player has won and 0 otherwise, so a draw counts as 0.

For a solved node the policy target is built from the proof. When the side to move wins at that node, the target is one-hot on the move that settled it: the winning move at a Proven OR node, the refuting move at a Disproven AND node. When the side to move loses, every move loses, and the target is uniform over the legal moves. Appendix B draws the four cases. The one-hot target is the one place where search order enters this stream: the move that settled a node is the first child the search happened to prove, not the cheapest.

3.3 Learning

A second process trains the network while the search runs. Records from both streams enter one replay buffer, and the trainer draws batches from it, stratified by verdict so that Proven and Disproven examples are balanced; under the proof-cost target the strata are verdict by log-cost cell instead. The loss is the sum of the policy cross-entropy and the value loss, as in AlphaZero.

Win/loss outcome. Both streams supply a value label, and the label answers the same question in both, whether the first player wins, rather than being expressed from the side to move as AlphaZero's labels are. For a trajectory record the label is the game's result. For a solved node it is the verdict for that position. A trajectory record's label is $v^\star$ of the game's final position rather than of the recorded

Table 1: Online solves from random initialization on the rules alone, each paired where run with a control that holds a randomly initialized network frozen for the whole solve. Wall time is mean ± sd where $n > 1$. The Connect Four control was stopped at an eight-hour budget with the root unproven, so its figures are lower bounds. The 5×6 unique-position count is absent because the monitoring series that carried it has aged out of retention.

Board	Run	n	Wall time	Simulations	Unique positions	Verdict
Connect Four 7×6	Trained online	6	2h30m16s ± 13m32s	7.24B	696.4M	Proven
	Untrained	1	$>$ 8h (censored)	$>$ 17.17B	$>$ 1.44B	—
Breakthrough 5×5	Trained online	1	22m44s	0.63B	62.6M	Disproven
	Untrained	1	2h26m20s	3.92B	188.9M	Disproven
Breakthrough 6×5	Trained online	1	2h55m	4.43B	266.5M	Disproven
Breakthrough 5×6	Trained online	1	13h08m	19.4B	—	Proven

one. The value head learns this as a binary classifier. The head carries an estimate for the OR side and for the AND side; under this target the two are complementary, so one output suffices and the search recovers the other as $1 - v$ (§3.1).

Proof cost. The value head can instead be trained on proof cost; our label comes from the measured proof tree size rather than the estimated minimum proof tree size as in Wu et al. (2022). Under this target the head predicts a distribution over log-binned cost. The two sides are not complementary here, so the head predicts a distribution for each.

The two streams arrive at very different rates, roughly a hundred solved nodes for every trajectory record (§4.3), so their mixture is set in the loss rather than by sampling. A parameter f fixes the share of each batch's loss that trajectory records carry, and the weight that achieves it is recomputed every batch from the share that actually arrived. For a batch of n records, n_T of them trajectory records, with per-record loss ℓ_i, the batch loss is

$$L = \frac{1}{n} \sum_{i=1}^{n} w_i \ell_i, \qquad w_i = \begin{cases} f\, n/n_T & \text{trajectory record,} \\ (1-f)\, n/(n-n_T) & \text{solved node,} \end{cases} \tag{3}$$

so trajectory records carry a share f of the loss whatever their share of the batch. A batch holding only one kind of record, as before the first game finishes, takes $w_i = 1$. At $f = 0$ the batch takes the plain mean over solved nodes, and trajectory records are never collected. At $f = 1$ the batch takes the plain mean of trajectory records. The effect of this configuration is shown in Table 3.

4 Experiments

4.1 Solving from random initialization

First, we explored the effect of training on solved positions in isolation with the value head targeting win/loss outcome. AlphaSolve weakly solved all four boards with a randomly initialized network training on the solved data stream in parallel. Using only the network for guidance, AlphaSolve returned the published verdict in each case: the first-player win on 7×6 Connect Four (Allen, 1989; Allis, 1988), the second-player wins on 5×5 and 6×5 Breakthrough, and the first-player win on 5×6 (Saffidine et al., 2011; Considine, 2025) (Table 1). Every run used the loop of Section 3, with one A100 serving inference and a second training the network concurrently (Appendix A).

To measure what the learning contributes, we paired two of the solves with a control that holds the randomly initialized network frozen for the whole solve. On Connect Four the frozen search had not proved the root after eight hours, at which point it was stopped, against 2 hours 30 minutes for the search that learns as it goes; learning is therefore worth at least $3.2\times$. On 5×5 Breakthrough the frozen search finished in 2 hours 26 minutes against 23 minutes, a factor of $6.4\times$.

Table 2: Online solves of 7×6 Connect Four from random initialization on the rules alone, mean $\pm$ sd. Both arms use the cost-stratified replay buffer, so they differ in the value target and nothing else.

Value target	n	Wall time	vs. Outcome
Outcome	3	3h01m59s $\pm$ 9m	$1.000\times$
Proof cost	3	3h59m05s $\pm$ 44m	$1.314\times$

Table 3: Trajectory-mix sweep, online 7×6 Connect Four from random initialization on the rules alone, mean $\pm$ sd. f is the share of each batch's *loss* carried by trajectory records, with the per-record weight recomputed each batch from the realized arrival share. The $f = 1$ arm was stopped at an eight-hour budget without a verdict, so its entry is a lower bound.

f	n	Wall time	Simulations	vs. $f = 0$
0.00 (verdicts only)	6	2h30m16s $\pm$ 13m32s	7.24B	$1.00\times$
0.01	6	2h00m24s $\pm$ 15m40s	5.52B	$0.80\times$
0.25	3	2h57m39s $\pm$ 11m28s	6.85B	$1.18\times$
1.00 (trajectories only)	1	$>$ 8h (censored)	—	$> 3.19\times$

4.2 Outcome versus proof cost

The dominant family of solvers descends on proof cost (Kishimoto et al., 2012). So the next experiment we ran was a comparison between win/loss outcome and proof cost as a value target, once again training solely on the solved data stream. The proof cost label is the one defined in §3.2.

Changing the target changes one other thing: a proof-cost head is trained from a replay buffer stratified over verdict-by-log-cost cells, where an outcome head needs only two verdict cells. To hold the sampler fixed, we trained both heads on the cost stratification, which is why the outcome arm here is slower than the standard configuration of Table 1; the comparison is between targets and nothing else.

On the same sampler, a proof-cost head takes $1.314\times$ the wall time of an outcome head (Table 2). At three runs per arm the 95% interval on that ratio runs from $0.92\times$ to $1.71\times$, so we claim competitiveness rather than superiority: an outcome classifier pays no penalty for being the unconventional choice.

4.3 The two label streams

A run produces two populations of training examples from the same self-play games: the verdicts of solved nodes and the outcomes of finished games. On 7×6 Connect Four, trajectory records are about 1% of what reaches the replay buffer; the rest come from the stream of verdicts.

We swept f (§3.3) with everything else held at the $f = 0$ configuration. A small share helps and a large one hurts (Table 3). At $f = 0.01$ the solve is 19.9% faster than on verdicts alone; at $f = 0.25$ it is 18.2% slower, against a run-to-run spread of 9.0% on the control arm at $n = 6$. At $f = 1$, at the rate the solve produces trajectories, the run had not finished after eight hours, the same budget at which the untrained network of Table 1 was stopped, so trajectories alone were not measurably better than no learning at all. At this supply ratio that arm rehearses each record 124 times, so it is a negative control demonstrating the difference in data volume of the two streams rather than a reimplementation of AlphaZero (Silver et al., 2018).

Figure 2 shows where the $f = 0.01$ network differs from the $f = 0$ network. On held-out solved positions, where both networks train, the two are indistinguishable: the trajectory share costs nothing on what verdicts teach. On trajectory records, which the $f = 0$ network never sees, it predicts that stream far worse. The solver works its way up the tree, so the opening is underrepresented in the solved data stream, while the trajectory stream is uniformly spread across plies. The search reflects this. Under $f = 0.01$, a small share of trajectory loss buys guidance in the opening, where the verdict stream has almost no data, and that is where the speedup comes from.

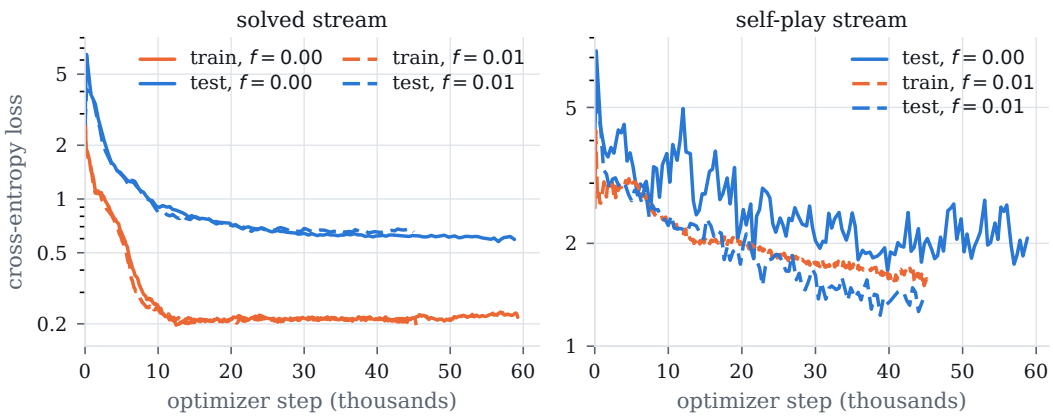


Figure 2: Prospective cross-entropy loss, the combined policy and value loss the trainer minimizes, of the $f = 0$ and $f = 0.01$ networks against optimizer step, for one pair of runs (`c6b40d7d` and `1aad4590`). Every point is the prediction the network made when it expanded a node, graded against the label that arrived later: the verdict for solved positions (left), the game outcome and visit distribution for trajectory records (right). The $f = 0$ network never trains on trajectory records, so its right-hand curve measures transfer and has no training counterpart. On the self-play stream, the $f = 0.01$ training curve rises above its own test curve partway through the run, which we attribute to target drift: as the solve progresses, the visit count distribution gets sharper faster than what training can keep up with.

5 Limitations

Faster than the same solver, not faster than the field. We did not compare AlphaSolve against classical solvers. The question here is whether learning win/loss outcomes from solved positions contributes during a solve. A comparison against df-pn (Nagai, 2002) or retrograde analysis (Considine, 2025) on the same board would measure the search's engineering rather than the learning, so our claim on efficiency is only that AlphaSolve is faster with learning on self-generated data and that win/loss outcome is a reasonable label for solved positions.

The mixture's direction, not its share. The mixture sweep ran only on one game. The best share may differ by game, and locating it on another board is a sweep we have not run.

The proof-cost arm is our implementation, not PCN. We label a solved node with the size of the proof the search actually found, because a method that trains on solved positions only should label them with a measurement rather than an estimate. Wu et al. (2023) choose otherwise: solved positions enter their training set at a cost of zero, and, as in Wu et al. (2022), the cost of an unsolved position is reconstructed from a self-play episode. A zero teaches the network a fact about the search's bookkeeping rather than about the position, which may be helpful in producing a more efficient solve, but we wanted to explore an alternative that was less sensitive to the search's bookkeeping (§3.2). That reasoning is why §4.2 compares the observed win/loss outcome against the observed proof cost as value targets rather than comparing AlphaSolve against PCN. We did not run PCN as published, so nothing here shows our solver is more efficient than theirs. Counting a position once per path also means the label overstates the work a search over a graph of positions actually does.

6 Conclusion

We introduced AlphaSolve, a weak solver that learns its search guidance from the solve itself, and showed that the verdicts a solver produces on its way to the root are enough to learn that guidance from random initialization. On the rules alone, it weakly solved 7×6 Connect Four and Breakthrough on 5×5, 6×5 and 5×6, returning the published verdict in each case. Against the same search with the network frozen at its random initialization, learning paid for itself: $6.4\times$ on 5×5 Breakthrough, and at least $3.2\times$ on Connect Four, where the frozen search had not finished at eight hours. A win/loss

classifier was competitive with the proof-cost target the dominant family of solvers descends on, and a small share of self-play trajectories mixed into the verdicts made the solve faster still, while a quarter made it slower and trajectories alone did not finish.

Our results rest on a fact about solving rather than about learning. A solver solves positions orders of magnitude faster than it plays games, and every position it solves is an exact label, so a solver can pay for its own guidance with data it was already producing. Reinforcement learning systems have traditionally bought their training data with generation compute, as AlphaZero did (Silver et al., 2018); a solver generates additional training data with the same compute.

Future work includes applying AlphaSolve to more complex games, analyzing the mixing ratio for different games, and evaluating AlphaSolve's ability in the game-playing setting in addition to game solving.

Game solvers generate ground truth data in abundance and can use that data to improve their own guidance.

Reproducibility Statement

The loop, the label construction and the training setup are described in Section 3, with the policy-target cases drawn out in Appendix B. Search constants, optimizer settings, checkpoint cadence and hardware are in Appendix A, together with the deviations for the Breakthrough runs and the transposition-table keying that the soundness argument depends on. Pseudocode for the search is given as Algorithm 1 in Appendix C. The runs consume no external data: every training example is generated by the solve itself from the rules of the game, so there is no dataset to obtain or reproduce. Every table states its n; solve times vary between seeds at a coefficient of variation of 9.0% over six replicates (§4.3), so a single reproduction should be read against that spread rather than against a point estimate.

AI Use Statement

The authors used a large language model as a coding and writing assistant. Its use covered drafting and editing prose, producing the figures from run records the authors' system had already written, verifying reported numbers against those records and against the solver's source, and locating references. It also gave feedback on the design of the experiments. The research questions, the solver and the interpretation of the results are the authors' own. The authors take full responsibility for all content.

References

Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

James D. Allen. A note on the computer solution of Connect-Four. In *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*, pp. 134–135, 1989.

L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

Louis Victor Allis. A knowledge-based approach of Connect-Four: The game is over, white to move wins. Master's thesis, Vrije Universiteit, Amsterdam, October 1988. Report No. IR-163, Faculty of Mathematics and Computer Science.

Yngvi Björnsson, Kristinn Vikar Jónsson, Dagur Benjamínson, Steinþór Bjarni Steinarsson, Edouard Saussoy, and Mark H. M. Winands. Solving Breakthrough for the 6×6 board. *ICGA Journal*, 48 (1):2–15, 2026. Published online 25 March 2026.

Dennis M. Breuker, Jos W. H. M. Uiterwijk, and H. Jaap van den Herik. The PN^2-search algorithm. In *Advances in Computer Games 9*, pp. 115–132. Universiteit Maastricht, 2001.

Jeffrey Considine. Compressed game solving. In *Computers and Games (CG 2024)*, volume 15550 of *Lecture Notes in Computer Science*, pp. 79–90. Springer, 2025. arXiv:2411.07273.

Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games (CG 2006)*, volume 4630 of *Lecture Notes in Computer Science*, pp. 72–83. Springer, 2006.

Chao Gao, Martin Müller, and Ryan Hayward. Focused depth-first proof number search using convolutional neural networks for the game of Hex. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, pp. 3668–3674, 2017.

Akihiro Kishimoto, Mark H. M. Winands, Martin Müller, and Jahn-Takeshi Saito. Game-tree search using proof numbers: The first twenty years. *ICGA Journal*, 35(3):131–156, 2012.

Akihiro Kishimoto, Beat Buesser, Bei Chen, and Adi Botea. Depth-first proof-number search with heuristic edge cost and application to chemical synthesis planning. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, pp. 7224–7234, 2019.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer, 2006.

Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. HyperTree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022.

Li-Cheng Lan, Huan Zhang, Ti-Rong Wu, Meng-Yu Tsai, I-Chen Wu, and Cho-Jui Hsieh. Are AlphaZero-like agents robust to adversarial perturbations? In *Advances in Neural Information Processing Systems 35 (NeurIPS 2022)*, 2022.

Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 157–163, 1994.

Ayumu Nagai. *Df-pn Algorithm for Searching AND/OR Trees and its Applications*. PhD thesis, 2002.

Wim Pijls and Arie de Bruin. Game tree algorithms and solution trees. *Theoretical Computer Science*, 252(1–2):197–215, 2001.

Stuart Russell and Eric Wefald. Principles of metareasoning. *Artificial Intelligence*, 49(1–3):361–395, 1991.

Abdallah Saffidine, Nicolas Jouandeau, and Tristan Cazenave. Solving Breakthrough with race patterns and job-level proof number search. In *Advances in Computer Games (ACG 2011)*, Lecture Notes in Computer Science, pp. 196–207. Springer, 2011.

Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587): 484–489, 2016.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

David E. Smith. Controlling backward inference. *Artificial Intelligence*, 39(2):145–208, 1989.

George C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2): 179–196, 1979.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1–2):277–311, 2002.

Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo tree search solver. In *Computers and Games (CG 2008)*, volume 5131 of *Lecture Notes in Computer Science*, pp. 25–36. Springer, 2008.

I-Chen Wu, Hung-Hsuan Lin, Der-Johng Sun, Kuo-Yuan Kao, Ping-Hung Lin, Yi-Chih Chan, and Po-Ting Chen. Job-level proof number search. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(1):44–56, 2013.

Ti-Rong Wu, Chung-Chin Shih, Ting Han Wei, Meng-Yu Tsai, Wei-Yuan Hsu, and I-Chen Wu. AlphaZero-based proof cost network to aid game solving. In *10th International Conference on Learning Representations (ICLR 2022)*, 2022.

Ti-Rong Wu, Hung Guei, Ting Han Wei, Chung-Chin Shih, Jui-Te Chin, and I-Chen Wu. Game solving with online fine-tuning. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023.

A Additional experimental details

Table 4: Search and training constants for the Connect Four measurements. Root Dirichlet noise and c_{puct} are identical across games and identical between online and frozen runs; only the move-selection temperature schedule differs by game. (Breakthrough deviations in training configuration are stated below.)

c_{puct}	1.0
Simulations per move	up to $1{,}000$, stopping early if the root is solved; the subtree beneath the played move is kept
$Q(c)$ at $N(c) = 0$	parent's running mean (0.5 at the root), $1 - v$ at AND nodes; no first-play-urgency reduction
Root exploration noise	Dirichlet, $\alpha = 10/\lvert C(n)\rvert$, mixed at weight 0.25; applied at every search, in the frozen solves as well as the online ones
Move-selection temperature	linear decay, 1.0 at ply 0 to greedy at ply 30
Transposition table	two tiers: in-process, 6.7×10^7 slots, direct-mapped one entry per slot, replace-always; backed by a distributed tier bounded at 40 GB, evicting least-recently-used, not persisted across runs
Checkpoint cadence (online)	every 200 optimizer steps ($\approx$ one every thirty seconds at the observed step rate)
Optimizer / LR / batch	AdamW / 3×10^{-4} / 16,384
Trajectory mixture (§4.3)	$f \in \{0, 0.01, 0.25, 1.0\}$; per-sample weight recomputed each batch from the realized arrival share; refused whenever the value target carries a proof-cost column

The untrained control. The untrained row of Table 1 serves a network that is never trained: weights are drawn once from a fixed seed, and the run carries no trainer, so they are the same weights at the eight-hour mark as at the first simulation. Every other quantity in the table above is unchanged. Its search still receives root Dirichlet noise, so the control is an uninformative prior over moves and values rather than a deterministic one.

Transposition table keys. Both tiers are keyed on a fixed-width lossless encoding of the position – 32 bytes, enough to hold any position of the games here – rather than on a hash of it. The in-process tier maps a position to a slot by a 64-bit hash, but stores the lossless encoding in the slot alongside the status and compares it in full on every read, so a lookup for a position occupying another position's

slot returns Unknown. Two positions can therefore evict each other, and cost the search the work of re-deriving a status, but cannot exchange statuses. Reads and writes are lock-free: a slot carries a verification word, written last and read first, that a reader checks against the status it read, so a read racing a write returns Unknown rather than a torn entry. The distributed tier is keyed on the same lossless encoding and stores no statuses under truncated keys.

Network and reported quantities. All runs share one trunk – a 4-layer, 4-head transformer encoder of model dimension 128 over one token per board cell – with a value head and a policy head above it, about 0.8M parameters for Connect Four's 42 tokens and 7 policy outputs. Only the input length, the policy width and the value head's width are game- or target-dependent. A *simulation* is one pass of the PUCT rule (§3.1). *Unique positions* counts the distinct positions the search reached, keyed by their lossless encoding, so a position is counted once however often the search returns to it, including through the transposition table; the cardinality is estimated by a sketch with standard error below 1%.

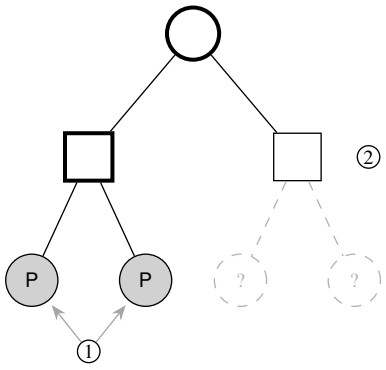
Connect Four measurements ran with search on a 16-vCPU high-memory instance (13 vCPUs available to the pod), inference served from a single on-demand A100. Training, where present, used a second A100 with AdamW at learning rate 3×10^{-4}, effective batch 16,384, bf16 autocast, and gradient clipping at 1.0. Inference instances were never preemptible, so that timing measurements could not be distorted by preemption.

Nodes already expanded keep the priors they were given. Re-evaluating them on every checkpoint would saturate the inference server, which is the bottleneck of the system, and it is not needed. A stale prior may cost search time, but its staleness is bounded by the prediction cache, which resets every 60 seconds.

The Breakthrough runs. These runs differ from the Connect Four experiments in several respects. The policy head is 900 outputs wide, since Breakthrough actions are ordered pairs of squares, raising the parameter count to approximately 0.9M. Training used a rolling replay buffer with learning rate 7×10^{-4} and batch 32,768. The search ran on a larger instance with different player counts and a shallower move-selection temperature schedule.

B LABEL CHANNELS AND POLICY TARGETS

Figure 3 draws the two channels of §3.2, only one of which emits a training label.



(1) The rules of the game. Every terminal position (shaded) is resolved outright. Status, no label.

(2) AND/OR propagation. A non-terminal position whose children settle it (bold) takes their verdict. Status *and* a training label.

Figure 3: The two channels through which a position acquires a status, only one of which emits a training label. Circles are OR nodes and squares are AND nodes; shading marks a position the rules resolve, bold marks one that propagation resolves, and grey dashed marks one the search never visited. Only channel (2) appends to the training stream.

Section 3.2 constructs the policy target from the game-theoretic label rather than from visit counts. Which construction applies depends on two things: whether the node is an OR node or an AND node, and whether its label is Proven or Disproven. The four combinations fall into two shapes, drawn in Figure 4.

A node is settled by one child when the side to move there is the side the label favors: a Proven OR node needed one winning move, and a Disproven AND node needed one refuting reply. The target is one-hot on the child that supplied it, and the remaining children may still be Unknown, since the search stopped asking about them. A node is settled by all of its children in the other two cases: a Disproven OR node had every move fail, and a Proven AND node had every reply answered. No single child is distinguished, so the target is uniform over the legal moves.

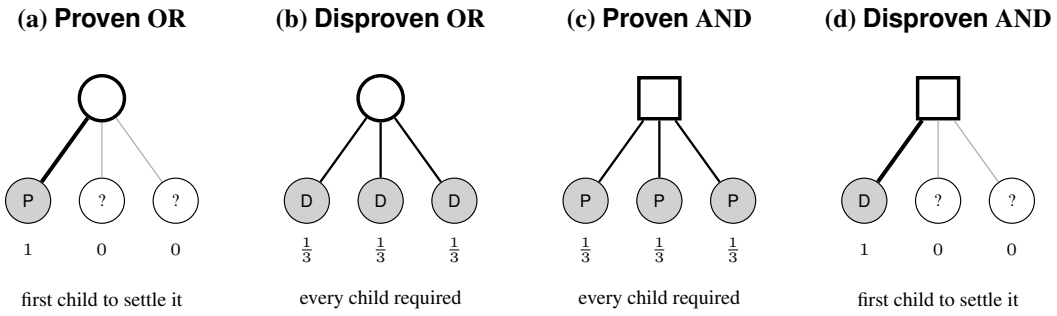


Figure 4: The policy target on a node with three legal moves, in each of the four cases. Circles are OR nodes and squares are AND nodes, the parent is bold because propagation resolved it, shaded children carry the verdict that settled it, and edge weight carries the target mass, printed beneath each child. In (a) and (d) one child settles the node and takes the whole target; in (b) and (c) every child was required and the target is uniform. Illegal moves, absent here, are masked from the loss rather than given zero mass.

The one-hot cases are the ones the search configuration can influence, as §3.2 notes: the child that settles a node is whichever the descent reached first, and every child the target can select is a correct move, so what varies between configurations is which of several correct moves is taught.

C THE SEARCH IN FULL

Algorithm 1: The search loop of §3.1, as one worker runs it. A worker calls PlayGame from the initial position until its status is known. Simulate recurses, so the value backup and the status update both happen at each node as the recursion unwinds. Statuses outlive the tree through the transposition table, so a later game inherits what an earlier one proved.

Input: position s, network f_θ, per-move budget B
Output: the verdict of s if this game settled it, and Unknown otherwise

```
procedure PlayGame(s, f_θ, B)
    r ← a fresh node for s; d ← 0
    while true do
        for B simulations or r.Status ≠ Unknown do
            Simulate(r, f_θ)
        if r.Status ≠ Unknown then
            CommitTrajectory(r.Status)
            return r.Status if d = 0, else Unknown
        RecordPosition(r)
        r ← SampleUnsolvedChild(r), keeping its subtree; d ← d + 1
procedure Simulate(n, f_θ)
    if n has no children then
        Expand(n)
        v ← Evaluate(n, f_θ)
        Backpropagate(n, v)
        return v
    c ← SelectChild(n)
    v ← Simulate(c, f_θ)
    Solve(n)
    Backpropagate(n, v)
    return v
procedure RecordPosition(r)
    π ← each unsolved child's share of the visits over r's unsolved children
    buffer (r, π) with its value pending
procedure CommitTrajectory(tv)
    foreach buffered record do
        label it tv and write it to the trajectory stream (§3.2)
    discard the buffer
procedure SampleUnsolvedChild(r)
    return a sample among r's unsolved children by visit count, at the move-selection
      temperature of Appendix A
```

Algorithm 2: The procedures Algorithm 1 calls: selection, expansion and evaluation on the way down, the status update and the value backup on the way back up.

procedure `SelectChild(`n`)`
  return $\arg\max_c Q(n,c) + U(n,c)$ *over the children* c *with* c.Status $=$ *Unknown*

procedure `Expand(`n`)`
  give n a child for each legal move

procedure `Evaluate(`n`,` f_θ`)`
  `// game rules`
  if `Solve(`n`)` $\neq$ *Unknown* **then return** `SolvedEstimate(`n`)`
  `// in-memory TT`
  n.Status $\leftarrow$ `TT.Read(`n`)`
  if n.Status $\neq$ *Unknown* **then return** `SolvedEstimate(`n`)`
  `// prediction cache`
  $(p, v, \mathit{found}) \leftarrow$ `PredCache.Read(`n`)`
  if found **then**
    set the prior of each child of n from p
    return v
  `// distributed TT`
  n.Status $\leftarrow$ `DistTT.Read(`n`)`
  if n.Status $\neq$ *Unknown* **then**
    `TT.Write(`n`)`
    return `SolvedEstimate(`n`)`
  `// the network`
  $(p, v) \leftarrow f_\theta(n)$
  `PredCache.Write(`n`,` p`,` v`)`
  set the prior of each child of n from p
  return v

procedure `SolvedEstimate(`n`)`
  if n.Status $=$ *Proven* **then return** $\{\text{Prove} : 1, \text{Disprove} : 0\}$
  return $\{\text{Prove} : 0, \text{Disprove} : 1\}$

procedure `Solve(`n`)`
  if n.Type $=$ OR **then**
    if *any child has* Status $=$ *Proven* **then** n.Status $\leftarrow$ Proven
    if *all children have* Status $=$ *Disproven* **then** n.Status $\leftarrow$ Disproven
  else
    if *any child has* Status $=$ *Disproven* **then** n.Status $\leftarrow$ Disproven
    if *all children have* Status $=$ *Proven* **then** n.Status $\leftarrow$ Proven
  if n.Status $\neq$ *Unknown* **then**
    `TT.Write(`n`)`
    `SolvedStream.Write(`n`) // the solved stream of §3.2`
  return n.Status

procedure `Backpropagate(`n`,` v`)`
  n.Visits $++$
  n.TotalProve $\leftarrow$ n.TotalProve $+$ v.Prove
  n.TotalDisprove $\leftarrow$ n.TotalDisprove $+$ v.Disprove