

Superhuman AI for Stratego Using Self-Play Reinforcement Learning and Test-Time Search

Samuel Sokota¹, Eugene Vinitzky², Hengyuan Hu³, J. Zico Kolter¹, and Gabriele Farina⁴

¹Carnegie Mellon University

²NYU Tandon School of Engineering

³Stanford University

⁴Massachusetts Institute of Technology

Abstract

Few classical games have been regarded as such significant benchmarks of artificial intelligence as to have justified training costs in the millions of dollars. Among these, Stratego—a board wargame exemplifying the challenge of strategic decision making under massive amounts of hidden information—stands apart as a case where such efforts failed to produce performance at the level of top humans. This work establishes a step change in both performance and cost for Stratego, showing that it is now possible not only to reach the level of top humans, but to achieve vastly superhuman level—and that doing so requires not an industrial budget, but merely a few thousand dollars. We achieved this result by developing general approaches for self-play reinforcement learning and test-time search under imperfect information.

Stratego is a board wargame resembling military chess (see Figure 1). It is distinctive among classical games regarded as major benchmarks of artificial intelligence (AI) in that it possesses a massive amount of hidden information. This contrasts not only with games that have no hidden information, like chess and Go, but also with games that have so little that AI systems can explicitly enumerate all possibilities, like Texas hold’em. The approaches that achieved superhuman performance in these other games [1, 2, 3, 4, 5, 6, 7, 8, 9] cannot even be meaningfully applied in settings with as much hidden information as Stratego.

Stratego also stands apart for its sheer difficulty as a benchmark. It has been the subject of research efforts spanning multiple years, involving dozens of researchers, and culminating in the expenditure of computational resources with commercial costs in the millions of dollars [10]. Yet, top human players have remained superior, perhaps making it the only classical game in which such well-resourced efforts failed to produce superhuman performance.

In this work, we introduce Ataraxos, an AI for Stratego. In a 20-game series, Ataraxos defeated the most decorated Stratego player of all time by a margin of victory unprecedented at the highest level of play: 15 wins, 4 draws, and 1 loss. Ataraxos achieved this result while only costing a few thousand dollars to train.

This simultaneous step change in performance and cost was made possible by innovations in reinforcement learning and search. The simplicity and generality of these innovations suggest that superhuman decision-making systems may now be attainable at modest computational cost in many strategic settings.

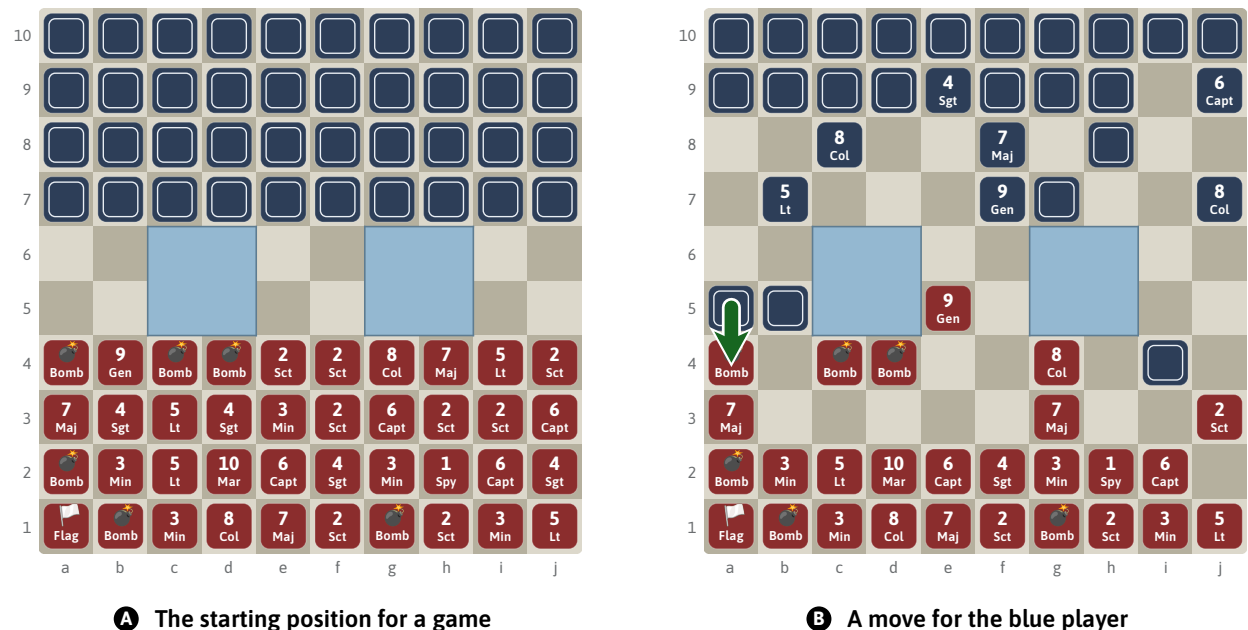


Figure 1: An illustration of the game of Stratego. **A** The starting position of a game from the perspective of the red player. Each player arranges their pieces in secret, so the types of the blue pieces are concealed from the red player (and vice versa). From this starting position, the players alternate moves, each of which is defined by the displacement of a single piece, as in chess. **B** A move in the game for the blue player. On this move, because a blue piece moves onto a square occupied by a red piece, there is a battle. During a battle, the types of both pieces are revealed and at least one of the two is captured (determined based on their types and attack-defense roles). If a Flag (see piece on a1) is captured in battle, the capturing player wins the game.

1 The Challenge of Imperfect Information

Imperfect information is a term of art that describes interactions in which it is possible for some agents to possess information that others do not. It is a core feature of many real-world settings: in financial markets, traders may not observe the rationales of trades made by others; in military

operations, forces may have incomplete intelligence about enemy positions and capabilities; and in negotiations, parties may lack knowledge about the priorities and constraints of their counterparts. Such settings contrast with perfect-information games, such as chess and Go, in which agents possess all strategically relevant information.

This distinction has important consequences. In games like chess and Go, the conceptual *right thing to do* is obvious: select a move that maximizes the value of the resulting position. But under imperfect information, it is much less clear. The value of a decision depends not only on the policies agents employ thereafter, but also on the policies agents used—and counterfactually would have used—prior to and at the time of the decision (as these all affect the posterior distribution over hidden information). As illustrated by Figure 2, even reasoning in isolation about the dependencies among contemporaneous counterfactuals is complex; incorporating dependencies across time is harder still.

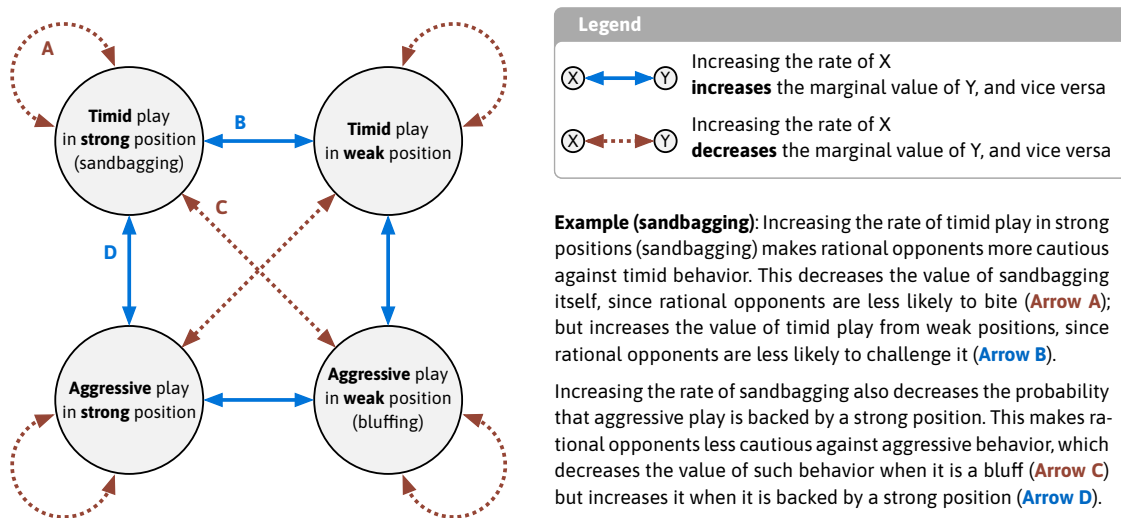


Figure 2: Graphical representation of differential relationships between decision frequencies and values among contemporaneous counterfactuals under imperfect information. Nodes represent decisions. The color/dash pattern of an edge represents whether changing the frequency of a decision on one end of the edge increases (blue/solid) or decreases (brown/dashed) the marginal value of the decision on the other end of the edge. The interconnectedness of frequencies and values across decisions makes it difficult to intuit the frequencies with which decisions should be made.

These dependencies have made developing AI for imperfect-information games challenging. The most successful approaches use sophisticated problem transformations based on public information [11, 12, 13, 9, 14]. But the cost of these transformations scales with the amount of hidden information, making them applicable only to games with small amounts of hidden information, like Texas hold'em (in which there are only 1,326 possible hands). Due to this fundamental limitation, and the absence of an alternative practical foundation, strategic decision making in settings with large amounts of hidden information has remained an open problem, as exemplified by sustained human supremacy at Stratego (in which there are over 10^{33} piece configurations).

2 Description of Ataraxos

This work introduces Ataraxos, an AI for Stratego based on *tabula rasa* self-play reinforcement learning and test-time search. The core innovation that enables Ataraxos to leverage reinforcement learning and search where past approaches have floundered is a coordinated interplay of regularization strength, policy update size, and policy strength. Ataraxos employs strong regularization and aggressive updates when the policy is weak, and weak regularization and small policy updates when the policy is strong.

Two complementary mechanisms motivate this interplay. First, keeping update sizes commensurate with regularization strength damps the otherwise cyclical, divergent, or chaotic learning dynamics engendered by imperfect information, leading to stable progress. Second, adapting both quantities to policy strength creates rapid improvements when the policy is weak and continued improvement when the policy is strong—avoiding both update sizes so small as to make improvements impractical and regularization so large as to compromise the policy.

These mechanisms together beget a policy improvement operator that remains reliable and productive throughout training- and test-time, driving Ataraxos to superhuman strategic decision making under massive amounts of hidden information.

2.1 Interdependent Self-Play Reinforcement Learning Processes

The foundation of Ataraxos is its self-play reinforcement learning module. This module comprises two separate but interdependent self-play processes associated with the two phases of Stratego. The first phase, wherein players privately determine starting positions for their pieces—called setups—is handled by one self-play process. The second phase, wherein players alternate moving their pieces, is handled by another. These processes learn separately but in tandem: the setup selection process determines the initial boards for the move selection process; and the move selection process determines game outcomes that both processes use for policy updates. This separation avoids tradeoffs involving design choices, network architecture, and hyperparameter regimes.

2.2 Transformer Architectures for Setup and Move Selection

To represent the policies and value functions for the two self-play processes, Ataraxos employs two transformers [15], which we call the setup network and the move network. These networks are diagrammed in Figure 3. Important choices included parameterizing the setup network as a decoder-only transformer, which allowed training on entire setups with single forward-backward passes; parameterizing the policy over moves using a key-query matrix product [16], which learned faster than less sophisticated parameterizations; and sizing the move network to balance sample efficiency against iteration speed, between which we observed significant tradeoffs.

2.3 Self-Play Training Data Generation

Ataraxos directly samples from its policy networks to generate self-play data. This is in contrast to other self-play deep reinforcement learning approaches to strategic decision making [2, 3, 5, 6, 9],

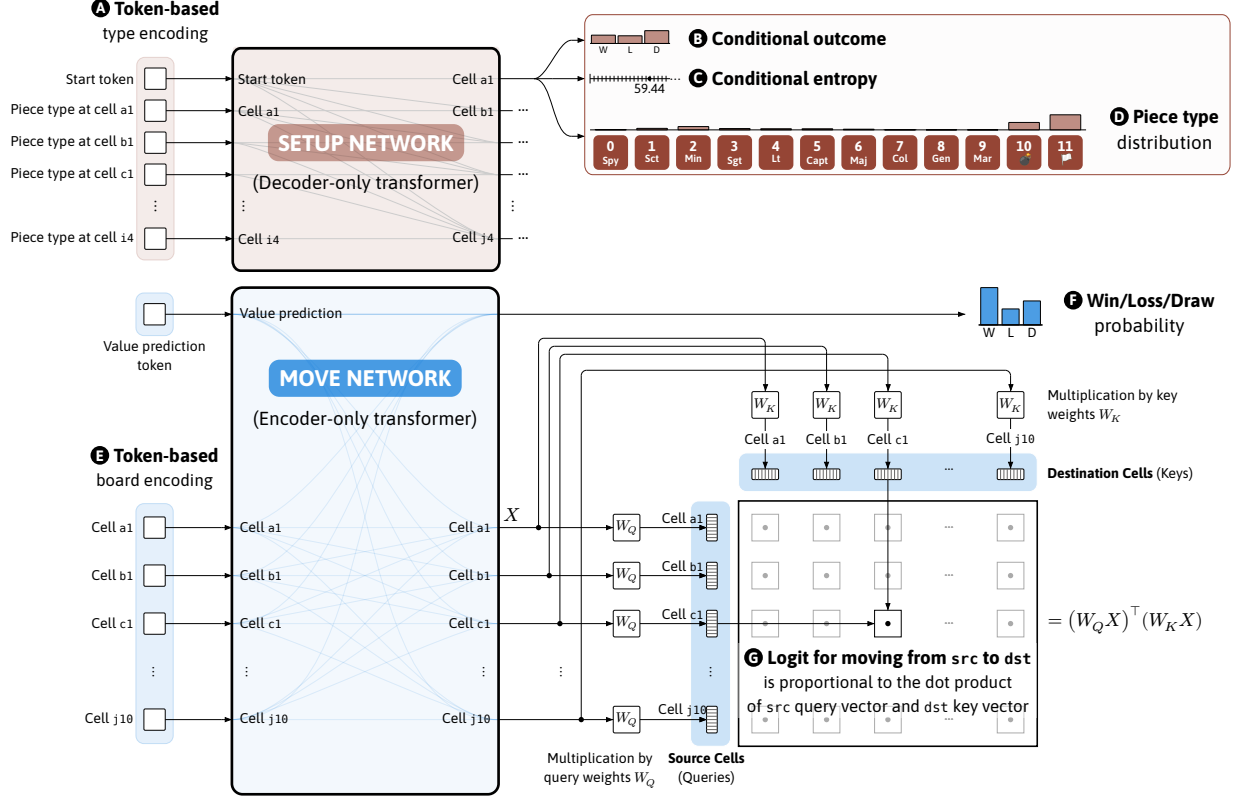


Figure 3: Setup and move network diagrams. **(Top)** The setup network is a decoder-only transformer that generates setups by autoregressively placing the 40 pieces onto squares of the board in row-major order (i.e., first row, first column; first row, second column; and so forth). **A** As input, the setup network takes tokenized piece types with learned absolute positional embeddings [17]. As output, for each setup prefix, the setup network provides: **B** an estimate of the conditional probability of winning, losing, or drawing a self-play game; **C** an estimate of the conditional entropy of the setup; and **D** probabilities for placing pieces of each type on that square. **(Bottom)** The move network is an encoder-only transformer that selects the piece to move and the square to which to move it. **E** As input, the move network takes tokenized representations of the occupiable squares with learned absolute positional embeddings, and an additional token for value prediction. The move network outputs: **F** a categorical prediction of the game outcome; and **G** probabilities for the moves computed using a key-query matrix product [16].

which use search to generate data. Motivated by the success of these approaches, we experimented with search-based self-play methods, but found that the slowdown on data generation imposed by the search methods we tried outweighed potential benefits. This could be because quantity of games matters more than quality of data both for learning a large-support distribution over setups and for learning to play the lines for each of the many possible pairs of setups in accordance with the associated posterior distributions.

For the moves played during these games, Ataraxos computes the estimates of expected cumulants [18] and advantages needed for training updates using λ -returns [19, 20] (with distinct λ

values), and trains only on those with large estimated advantage magnitudes [21]. Filtering in this way reduced the overall wall-clock time per reinforcement learning iteration by a factor of about 2.5, while simultaneously—in a phenomenon meriting further investigation—actually increasing both sample efficiency (per environment query) and asymptotic performance.

For the setups, Ataraxos uses Monte Carlo returns (i.e., final outcomes of games played by the current policy) to estimate these quantities, and does not employ filtering. The superiority of Monte Carlo returns for advantage estimation is unusual in reinforcement learning, but has also been observed in language model reasoning [22], possibly suggesting that this superiority is related to the bandit-like structure shared by the two problems.

2.4 Dynamically Damped Self-Play Reinforcement Learning

Ataraxos trains on on-policy or close to on-policy self-play data by dynamically damping its learning dynamics. This allows it to leverage standard policy optimization tools to regularize and control the size of its policy updates, obviating the need for more onerous techniques, such as trajectory importance reweighting and policy averaging, for handling imperfect information.

For regularization, Ataraxos incorporates additional terms into the losses of its networks. For the setup network, it uses a maximum entropy term [23]; for the move network, it uses a myopic reverse KL penalty toward the policy that selects a movable piece uniformly at random and then selects a legal move for this piece uniformly at random. Ataraxos anneals the coefficients of these regularization terms according to distinct power laws over training [24]. We found that this regularization functioned somewhat analogously to an energy reserve—annealing too cautiously left the playing ability of the model underdeveloped, while annealing too aggressively produced rapid initial gains but also collapsed the entropy of the model, depleting its capacity to learn thereafter and often making it easy to exploit.

For update size control, Ataraxos employs four mechanisms, which we found offered complementary benefits: a reverse KL penalty to the data collection policy, importance ratio clipping [25], gradient norm clipping [26], and the learning rate of Adam [27]. Ataraxos anneals the learning rate for its move network according to a power law over the course of training. We found that scheduling this learning rate was crucial both for rapid learning early in training and for preventing plateauing later in training.

2.5 Test-Time Search via Update Equivalence

To further improve its policy, Ataraxos employs additional computation immediately prior to each move to perform test-time search. This search is conceptually straightforward—it simply performs an additional damped self-play reinforcement learning update [28]. That such search is feasible at all, and moreover simple, is noteworthy, as test-time search in settings with as much hidden information as Stratego has been viewed as such a major technical challenge that prior work effectively forwent it [10].

Ataraxos facilitates this search by training a belief network to maximize the log-likelihood of hidden pieces over trajectories of the final self-play policy. This belief network processes the known information using a transformer encoder (similar to the move network) and decodes

the types of unknown pieces in row-major order using a transformer decoder. Ataraxos applies dropout [29] to the belief network during training to aid generalization to out-of-distribution positions reachable by opponents that play very differently from its setup and move networks, such as humans.

Ataraxos uses the belief network prior to each move to sample a collection of possible game states given its current position. For each candidate move, it runs depth-limited rollouts from these game states—starting with that candidate, then using its move network to simulate both players. Ataraxos estimates the value of the candidate move by averaging its network value predictions across the positions reached by rollouts starting from that move. Since the belief network approximates the posterior distribution of the self-play policy, the rollouts are executed by the self-play policy, and the network predicts self-play position values, these averaged values approximate self-play action values regardless of the policy of the opponent Ataraxos is facing.

Ataraxos uses these values to update its policy with a tabular step of magnetic mirror descent [24], which regularizes and controls the size of the update with the same two reverse KL divergences used during training. Importantly, this update can safely be more aggressive than those during training, both because the test-time update is tabular (and thus does not interfere with the policy at other positions), and because it is based on the more accurate advantage estimates enabled by the larger amount of computation per position at test time. The move Ataraxos plays is sampled from the updated policy.

2.6 Computational and Experimental Efficiency

To make the project feasible on modest academic compute infrastructure, we implemented a GPU-accelerated Stratego simulator in CUDA C++ capable of sustaining approximately 10 million state updates per second on an Nvidia Hopper H100 GPU (H100). Rather than adopting the conventional approach of coupling this simulator to a separate CPU-based replay buffer—which would necessitate expensive host-device communication—we integrated replay buffer functionality directly into the GPU-resident simulator system. This system generates trajectories, stores them using compact representations, and reconstructs them entirely within device memory. This architectural design exploits the fact that for modern accelerators, recomputing large tensors from compact representations on the fly is substantially faster than retrieving precomputed tensors from CPU memory. We applied this same zero-retrieval approach—avoiding CPU-GPU data transfers—to enable fast search: the system ingests on-device samples from the belief model and materializes the corresponding simulator states directly on the GPU.

Other important facilitators of efficiency included the bfloat16 data type, which sped up self-play iterations by a roughly factor of 3 without negatively affecting performance; and exponential moving averages [30] of parameters, which reached indicative levels of performance more quickly and with lower variance than the parameters themselves, enabling more reliable hyperparameter tuning over shorter runs.

The training run for the final hyperparameters, which was launched at the end of May 2025, utilized PyTorch distributed data parallel [31] to train the setup and move networks on 16 H100s for 1 week and, subsequently, the belief network on 4 H100s for 4 days. Figure 4 shows metrics for this training run and searches performed on top of its final networks.

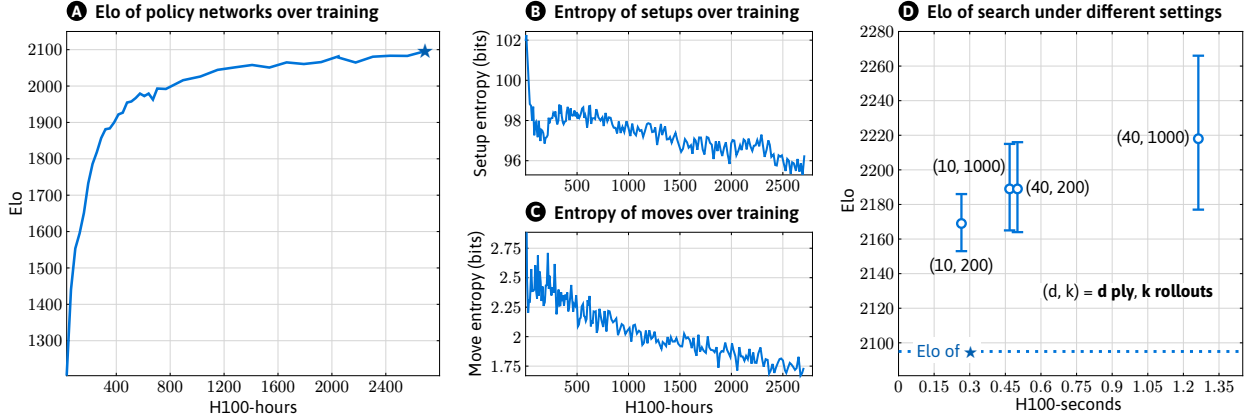


Figure 4: Performance and entropy of Ataraxos as function of compute. **A** Elo of the policy networks over the course of training, as measured by performance against a fixed reference opponent. Quantifying skill level in Stratego by Elo is useful but flawed, as randomization can compress margins (i.e., make margins between two players smaller than would be predicted by their respective margins against a common third player whose skill level lay between them), among other reasons. **B** The entropy of the setups and **C** the entropy the moves (in expectation over the self-play distribution of positions). Dynamic damping shepherds these entropies smoothly downward over training, preventing them from collapsing. **D** Elo of search on a single H100 under varying rollout counts and search depths with 95% confidence intervals. Deeper searches with more rollouts produce stronger performance but require more time per move. During the evaluation, Ataraxos used a 40-ply 1000-rollout search that averaged about 1.26 seconds per move—a pace of play faster than that of human players.

3 Evaluation

In July 2025, we evaluated Ataraxos (playing on a single H100) against Pim Niemeijer (hereafter Pim), a player whose accolades include: 4 world championships (the most of any active player and tied for the most of all time); 15 Dutch national championships (the most of all time); 2 online world championships (the most of all time); and over 600 weeks as the #1 ranked player (the most of all time). According to George Franka, the only player to compete in every world championship since 1997, Pim is “the best Stratego player ever.”

The evaluation consisted of a 20-game series, with performance measured by effective win rate (i.e., by counting draws as half wins). The large number of games was chosen both to reduce variance (due to the amount of randomization employed in Stratego, competent players win games against top players more often than in games like chess) and to give Pim an opportunity to find and exploit weaknesses in the strategy of Ataraxos—which Pim was made aware would not adapt to his play. To prevent fatigue and allow time for tactical preparation between games, the evaluation was spread over the course of 3 weeks. Pim was paid \$1,000 for participating in the evaluation, with an additional \$100 for each win and \$50 for each draw to align his monetary incentives with his performance.

Ataraxos won the series with 15 wins, 1 loss, and 4 draws (see Figure 5). This margin (an 85%

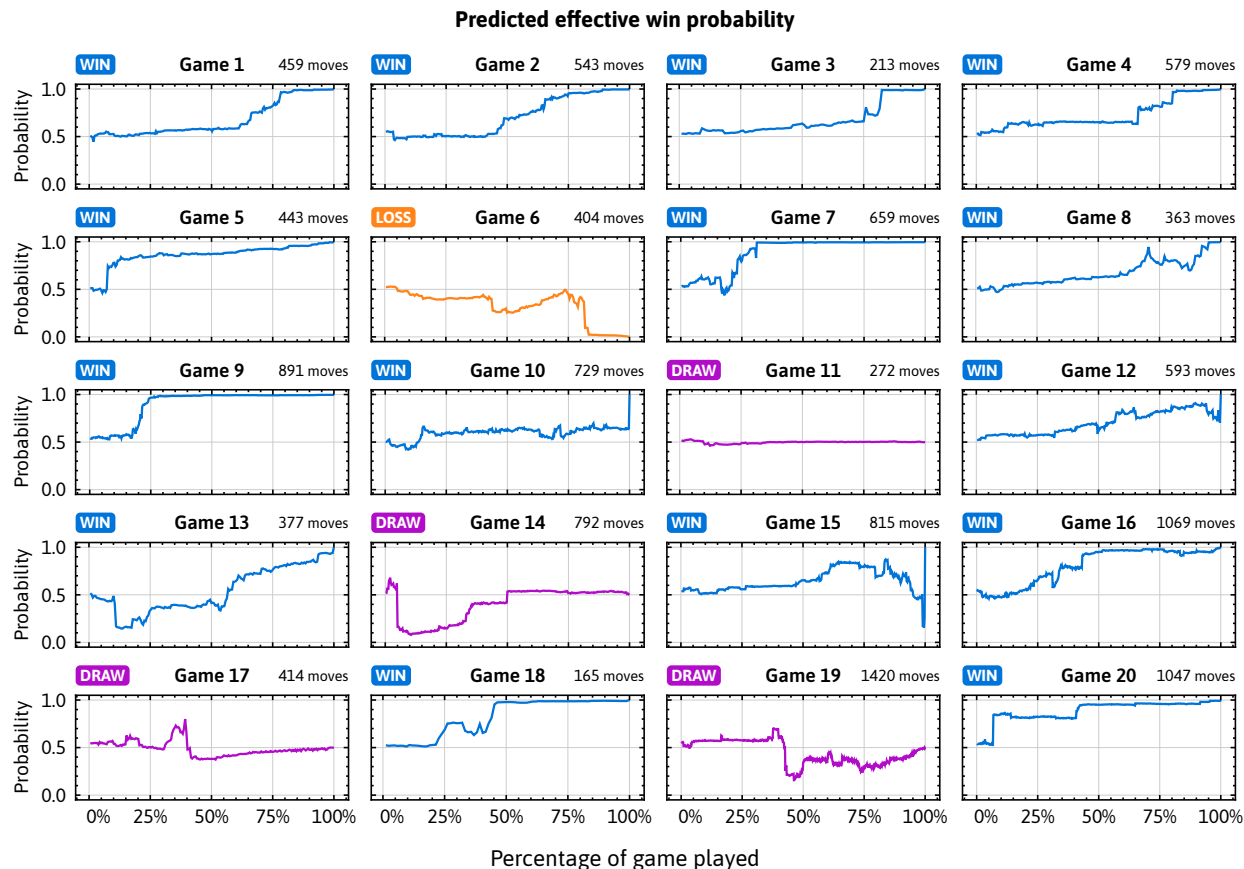


Figure 5: Effective win probability implied by the network value predictions over the games played against Pim Niemeijer. Effective win probability is the expected value of game with winning assigned value 1, drawing assigned value 1/2, and losing assigned value 0.

effective win rate) is unprecedentedly large for the highest level of human play, where, according to 3-time runner-up world champion Max Roelofs, margins are razor thin due to the level of risk that must inevitably be assumed during play. Ataraxos achieved such a margin despite the asymmetric structure of the evaluation—wherein Pim could adapt to Ataraxos over a large number of games, but Ataraxos could not adapt to Pim—which, according to 3-time world champion Vincent de Boer, constituted a large handicap.

Because of this adversarial adaptation, and more broadly because human strategies are not static from game to game, the outcomes of the games were far from independently and identically distributed. But under the assumption that they had been, the p -value associated with the alternative hypothesis that Ataraxos is more likely to win than lose would be less than 0.00026.

Following the evaluation against Pim, we demoed Ataraxos at the 2025 Stratego World Championship from August 1–3. During the demo, world championship attendees were given the opportunity to play against Ataraxos. Across 40 such games, Ataraxos recorded a vertiginously high 95% effective win rate (38 wins, 2 losses, 0 draws), confirming its dominance against a wide range of players and playing styles.

4 Conclusion

The combination of reinforcement learning and search has proven to be a powerful recipe for strategic decision making. But in the past, this recipe has had limited applicability to imperfect-information settings. The success of Ataraxos shows that reinforcement learning and search have now reached a point where large amounts of hidden information are no longer prohibitive. This puts practical AI within reach for many strategic decision-making problems for which fast, accurate simulators can be constructed.

5 Acknowledgments

We thank the site administrator of Strategus for allowing us to evaluate Ataraxos via Strategus. We thank Winner, Sébastien Crot, Axel Hangg, Rayan Manji, and Rein Halbersma for playtesting, feedback, and other assistance. We thank Hani Basilious, Shenglong Wang, and NYU’s high performance computing team for providing and facilitating the compute for the project. We thank Sara Werner for helping to facilitate the evaluation.

This work was supported by the Office of Naval Research grant N000142212121, initial funding from the NYU Department of Civil and Urban Engineering, the C2SMART Center, the National Science Foundation award CCF-2443068, the Office of Naval Research grant N000142512296, and an AI2050 Early Career Fellowship.

References

- [1] Murray Campbell, A. Joseph Hoane Jr., and Feng-hsiung Hsu. Deep blue. *Artificial Intelligence*, 134:57–83, 2002.
- [2] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan 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*, 362:1140–1144, 2018.
- [3] The LCZero Authors. LeelaChessZero. URL <https://github.com/LeelaChessZero/lc0>.
- [4] The Stockfish developers (see AUTHORS file). Stockfish. URL <https://github.com/official-stockfish/Stockfish>.
- [5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, L. Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354–359, 2017.
- [6] David J. Wu. Accelerating self-play learning in go, 2019. URL <http://arxiv.org/abs/1902.10565>.

- [7] Noam Brown and Tuomas Sandholm. Superhuman AI for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359:418–424, 2018.
- [8] Noam Brown and Tuomas Sandholm. Superhuman AI for multiplayer poker. *Science*, 365: 885–890, 2019.
- [9] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. In *Neural Information Processing Systems (NeurIPS)*, 2020.
- [10] Julien Perolat, Bart De Vylder, Daniel Hennes, Eugene Tarassov, Florian Strub, Vincent de Boer, Paul Muller, Jerome T. Connor, Neil Burch, Thomas Anthony, Stephen McAleer, Romuald Elie, Sarah H. Cen, Zhe Wang, Audrunas Gruslys, Aleksandra Malysheva, Mina Khan, Sherjil Ozair, Finbarr Timbers, Toby Pohlen, Tom Eccles, Mark Rowland, Marc Lantot, Jean-Baptiste Lespiau, Bilal Piot, Shayegan Omidshafiei, Edward Lockhart, Laurent Sifre, Nathalie Beauguerlange, Remi Munos, David Silver, Satinder Singh, Demis Hassabis, and Karl Tuyls. Mastering the game of Stratego with model-free multiagent reinforcement learning. *Science*, 378:990–996, 2022.
- [11] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356:508–513, 2017.
- [12] Noam Brown and Tuomas Sandholm. Safe and nested subgame solving for imperfect-information games. In *Neural Information Processing Systems (NIPS)*, 2017.
- [13] Noam Brown, Tuomas Sandholm, and Brandon Amos. Depth-limited solving for imperfect-information games. In *Neural Information Processing Systems (NeurIPS)*, 2018.
- [14] Samuel Sokota, Ryan D’Orazio, Chun Kai Ling, David J. Wu, J. Zico Kolter, and Noam Brown. Abstracting imperfect information away from two-player zero-sum games. In *International Conference on Machine Learning (ICML)*, 2023.
- [15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Neural Information Processing Systems (NIPS)*, 2017.
- [16] Daniel Monroe and Philip A. Chalmers. Mastering chess with a transformer model, 2024. URL <https://arxiv.org/abs/2409.12272>.
- [17] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. In *International Conference on Machine Learning (ICML)*, 2017.
- [18] Richard S. Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup. Horde: a scalable real-time architecture for learning knowledge

- from unsupervised sensorimotor interaction. In *Autonomous Agents and Multiagent Systems (AAMAS)*, 2011.
- [19] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
- [20] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations (ICLR)*, 2016.
- [21] Marco Francis Cusumano-Towner, David Hafner, Alexander Hertzberg, Brody Huval, Aleksei Petrenko, Eugene Vinitzky, Erik Wijmans, Taylor W. Killian, Stuart Bowers, Ozan Sener, Philipp Kraehenbuehl, and Vladlen Koltun. Robust autonomy emerges from self-play. In *International Conference on Machine Learning (ICML)*, 2025.
- [22] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojuan Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025.

- [23] Brian D. Ziebart, Andrew Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *National Conference on Artificial Intelligence (AAAI)*, 2008.
- [24] Samuel Sokota, Ryan D’Orazio, J Zico Kolter, Nicolas Loizou, Marc Lanctot, Ioannis Mitliagkas, Noam Brown, and Christian Kroer. A unified approach to reinforcement learning, quantal response equilibria, and two-player zero-sum games. In *International Conference on Learning Representations (ICLR)*, 2023.
- [25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL <https://arxiv.org/abs/1707.06347>.
- [26] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning (ICML)*, 2013.
- [27] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.
- [28] Samuel Sokota, Gabriele Farina, David J Wu, Hengyuan Hu, Kevin A. Wang, J Zico Kolter, and Noam Brown. The update-equivalence framework for decision-time planning. In *International Conference on Learning Representations (ICLR)*, 2024.
- [29] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.
- [30] B. T. Polyak and A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30:838–855, 1992.
- [31] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhersch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.
- [32] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL <https://arxiv.org/abs/1607.06450>.
- [33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

- [34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *International Conference on Computer Vision (ICCV)*, 2015.
- [35] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 2006.
- [36] Aaditya Ramdas, Peter Grünwald, Vladimir Vovk, and Glenn Shafer. Game-Theoretic Statistics and Safe Anytime-Valid Inference. *Statistical Science*, 38:576–601, 2023.
- [37] Larry Wasserman, Aaditya Ramdas, and Sivaraman Balakrishnan. Universal inference. *National Academy of Sciences*, 117:16880–16890, 2020.
- [38] R. Krichevsky and V. Trofimov. The performance of universal encoding. *IEEE Transactions on Information Theory*, 27:199–207, 1981.
- [39] Vincent de Boer. Invincible: A Stratego bot. Master’s thesis, Delft University of Technology, 2007.
- [40] Projekt Gravon. Stratego classic challenge rating/ranking 2022. <https://www.gravon.de/gravon/stratego/rating2022.jsp>, .
- [41] Projekt Gravon. Statistics for starship. <https://www.gravon.de/gravon/stratego/player0.jsp?nick=starship>, .
- [42] Google Cloud. Cloud tpu pricing. <https://cloud.google.com/tpu/pricing?hl=en>, 2025.
- [43] Voltage Park. Voltage park pricing. <https://www.voltagepark.com/pricing>, 2025.
- [44] Mingxing Xu, Wenrui Dai, Chunmiao Liu, Xing Gao, Weiyao Lin, Guo-Jun Qi, and Hongkai Xiong. Spatial-temporal transformer networks for traffic flow forecasting, 2021. URL <https://arxiv.org/abs/2001.02908>.
- [45] Brian Hu Zhang and Tuomas Sandholm. General search techniques without common knowledge for imperfect-information games, and application to superhuman fog of war chess, 2025. URL <https://arxiv.org/abs/2506.01242>.

A Rules of Stratego

Game Summary Stratego is a board wargame played on a 10-by-10 grid with 92 occupiable squares and 2 blocks of non-occupiable squares called lakes (see Figure 6). Each player begins with 40 pieces, arranged in secret on the first four rows of their side so that the identities are concealed from the other player. The game proceeds in alternating turns. On each turn, the acting player moves one piece. If the piece is moved onto a square occupied by a piece of the other player, a battle occurs, at which point both pieces are revealed and at least one of them is removed from the board. Victory is achieved by capturing the Flag of the other player or by leaving the other player with no legal moves; the game ends in a draw if neither player would possess legal moves.

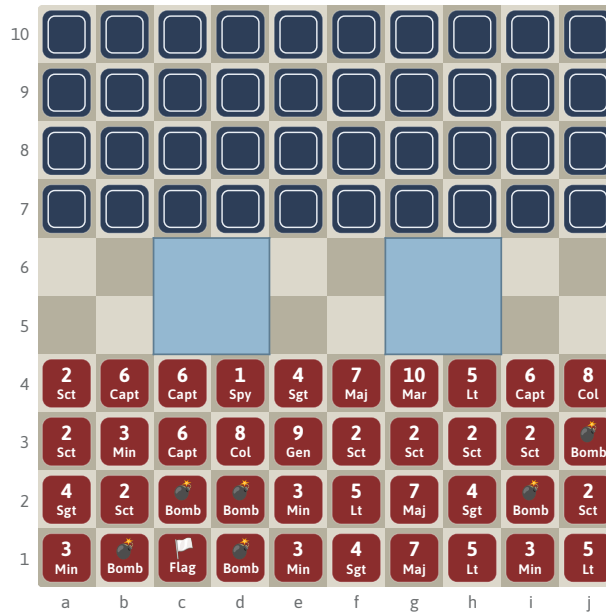


Figure 6: An example starting position. The light blue blocks in the middle of the board are called lakes, and cannot be occupied by the pieces of either player.

Piece details A description of the pieces is provided in Table 1. Movable pieces—aside from Scouts—can move one square in a cardinal direction (i.e., up, down, left, or right) to squares that are either empty or occupied by an opponent piece; Scouts can move any number of squares in a cardinal direction to squares that are either empty or occupied by an opponent piece, so long as the movement does not jump over a lake or an occupied square. When pieces engage in combat, the outcome is typically determined by rank, which means that the higher-ranking piece defeats the lower-ranking piece if their ranks differ and that both are defeated if their ranks are the same.

Additional rules for competitive play In competitive play, there are two additional rules. One is the two-square rule, which disallows a piece from crossing the same square boundary more than three consecutive turns of the owner of that piece in a row. The other is the continuous-chasing rule,

Table 1: Counts, movement rules, and combat rules for Stratego pieces.

Piece (rank)	Count	Movement	Combat
Flag	1	Immovable	Loses to any attacking piece.
Spy (1)	1	1 cardinal square	Defeats Marshal (10) when attacking, otherwise by rank.
Scout (2)	8	N cardinal squares	By rank.
Miner (3)	5	1 cardinal square	Defeats Bomb, otherwise by rank.
Sergeant (4)	4	1 cardinal square	By rank.
Lieutenant (5)	4	1 cardinal square	By rank.
Captain (6)	4	1 cardinal square	By rank.
Major (7)	3	1 cardinal square	By rank.
Colonel (8)	2	1 cardinal square	By rank.
General (9)	1	1 cardinal square	By rank.
Marshal (10)	1	1 cardinal square	Loses to Spy (1) when defending, otherwise by rank.
Bomb	6	Immovable	Loses to Miner (3), defeats any other attacking piece.

which sets limits on the ability of the pieces of one player to chase—in the sense of Table 2—those of the other.

Table 2: Definitions of threat, evade, chase, chasing.

Term	Definition
Threat	An action that moves a piece adjacent to a piece of the opponent.
Evade	An action that moves a piece that was threatened on the previous turn away from the piece that threatened it.
Chase	An unbroken sequence of alternating threats and evades.
Chasing	The act of making threats during a chase.

The continuous-chasing rule states that a player who is chasing may not make a threat that would result in a position that has already taken place during the chase, unless that threat would return the moved piece to the square it occupied prior to the previous turn of the chasing player.

Additional rule for online play There is also an additional rule implemented by Strategus, which is both the primary website for online competition and the website on which the evaluation against Pim Niemeijer was conducted. This rule is called the 200-move rule and states that the game ends in a draw if there is a sequence of 200 moves without a battle.

Time Controls Competitive Stratego also includes time controls. The evaluation against Pim Niemeijer was conducted under the default 15+3 Strategus time controls. The notation 15+3 means that each player starts with a 15 minute buffer and is allocated 3 free seconds for each move before their buffer starts to run down.

B GPU-Accelerated Stratego Simulator

To support reinforcement learning and search, we implemented an efficient GPU simulator for the game of Stratego. In this appendix, we describe the main architectural features of the simulator. We plan to open-source the simulator to enable future research from the community on this challenging benchmark.

Our simulator is written in CUDA C++, with Python bindings. At a high level, our simulator is designed with the following desiderata in mind:

- Avoid explicitly storing in memory quantities that are easily recomputed. For example, we eliminate the need for a traditional rollout buffer that stores large information states and legal action masks by implementing a simulator capable of traveling back in the history of a game and reconstructing quantities of interest on demand. This greatly reduced memory footprint and fragmentation, as discussed in Section B.1.
- Maximize simulation throughput, aiming for roughly 10 million board state updates per second, while also implementing anti-chasing rules.
- Minimize the number of dynamic memory allocations in the simulator. Our simulator allocates memory as needed at construction time, and manages memory directly rather than dynamically allocating and deallocating over time.
- Support search by enabling fast reset of board states to nonterminal states.
- Treating boards independently: once one game terminates, it is reset independently of whether the other games simulated in parallel have terminated or not. As a consequence, the boards gradually desynchronize, creating a distribution of training data that covers the different phases of the game.

B.1 Simulator and Rollout Buffer Design

Unlike typical RL infrastructure, we do not make a distinction between the rollout buffer and the simulator. Rather, we expose a single object, called `StrategoRolloutBuffer`, that serves both purposes. This object tracks a tunable number N of parallel games, and is responsible for two critical tasks:

- Support historical queries (for example, returning the legal action mask of a player at a given past state, or returning an encoding of the information state perceived by a player in the past);
- Support *stepping* the state, by receiving actions for each of the N parallel games and updating the states of the games accordingly. This can be achieved by calling the `ApplyActions` method, which expects as input a tensor of N actions.

Once a game terminates, a new game starts. A new game is created by sampling a new initial board for both players. The distribution from which the initial board is sampled can be customized. It is also possible to ask the simulator to reset terminated games to a specific *non-initial* game state; this feature is important to support search, as discussed in Section B.4.

The rollout buffer tracks game states in a *circular*, preallocated GPU-memory buffer of tunable length. Correspondingly, queries about past states can only be supported, provided that the past state is recent enough.

We found that this design, which integrates aspects of a traditional rollout buffer with a simulator, significantly reduces both memory fragmentation and usage compared to implementing a separate rollout buffer (e.g., using PyTorch). Indeed, a separate rollout buffer may need to make copies of legal action masks and information state tensors, and pack them into aggregate tensors allocated by a separate rollout buffer. Instead, our `StrategoRolloutBuffer` is able to reconstruct on-demand past information states and legal action masks. The training loop simply needs to remember at what historical time step the quantity needs to be computed, and then ask the backend to materialize the appropriate tensor to query the value and policy nets. Furthermore, this design is natural when considering that a Stratego simulator *needs* tracking history (at least to a nontrivial extent) to implement the anti-chasing rules described in Section A. Finally, by endowing the simulator with a notion of history, it becomes possible to implement efficient capturing of past states, as needed to efficiently implement search.

B.1.1 API Description

To index time into the `StrategoRolloutBuffer`, most methods exposed by the object take as input a *time step*. In the following table, we describe the main historical queries supported by the simulator. We defer a discussion about APIs for changing the resetting behavior upon game termination to Section B.4.

<i>Method</i>	<i>Input</i>	<i>Output</i>	<i>Description</i>
<code>CurrentStep</code>	(none)	64-bit integer	Returns the number of state updates performed by the simulator.
<code>ActingPlayer</code>	time step	0 or 1	The player acting at the current time step (0 for the red player, 1 for the blue player).
<code>ComputeLegalActionMask</code>	time step	(none)	Computes the legal action mask for the acting player at the given time step. The answer is written in a preallocated tensor kept by the rollout buffer.
<code>ComputeInfostateTensor</code>	time step	(none)	Computes the information state for the acting player at the given time step. The answer is written in a preallocated tensor kept by the rollout buffer.
<code>ComputeRewardP10</code>	time step	(none)	Computes the reward for the red player at the given time step. The reward is set to 0 for those boards that are not terminated at the given time step. The answer is written in a preallocated tensor kept by the rollout buffer.
<code>ComputeIsUnknownPiece</code>	time step	(none)	Computes a boolean tensor indicating which pieces are unknown to the acting player. The answer is written in a preallocated tensor kept by the rollout buffer.

ComputePieceTypeOnehot	time step	(none)	Computes a one-hot encoding of the piece types on the board from the perspective of the acting player. The answer is written in a pre-allocated tensor kept by the rollout buffer.
ComputeTwoSquareRuleApplies	time step	(none)	Computes whether the two-square repetition rule applies at the given time step. The answer is written in a preallocated tensor kept by the rollout buffer.
GetTerminatedSince	time step	Tensor	For each game, computes the number of steps since it terminated. For non-terminated games, this value is 0. Returns a non-owning tensor.
GetHasLegalMovement	time step	Tensor	Computes whether the acting player has at least one legal move. Returns a non-owning tensor.
GetFlagCaptured	time step	Tensor	Computes whether a player has captured the opponent's flag. Returns a non-owning tensor.
GetPlayedActions	time step	Tensor	Retrieves the action taken at the specified time step for each game. Returns a non-owning tensor.
GetNumMoves	time step	Tensor	Gets the total number of moves played in each game. Returns a non-owning tensor.
GetNumMovesSinceLastAttack	time step	Tensor	Gets the number of moves since the last attack for each game. Returns a non-owning tensor.
BoardStrs	time step	Vector of strings	Returns a string representation of the board state for each game at the current time step. Useful for debugging.
SampleRandomLegalAction	GPU-allocated tensor	(none)	Fills out the provided GPU-allocated tensor with a legal action drawn uniformly for each game at the current step.

B.2 Two-Square Rule

To efficiently implement the rule, we implemented a custom state machine whose state is tracked and updated by `StrategoRolloutBuffer`.

The state machine is implemented internally by tracking the last four positions occupied by the last-acting piece for each player. The update logic is implemented directly on the GPU. Special care needs to be used to properly account for Scouts, which have special movement abilities.

B.3 Continuous Chasing Rule

To properly implement the rule, the simulator requires access to the board history of each game. To quickly detect whether a move would violate the rule, we used a state machine design paired

with a fast diffing algorithm. Custom logic was implemented to make the rule compatible with supporting resetting terminated games to start from non-initial states (as needed to support search, cf. Section B.4). Indeed, in this case the continuous chasing rule needs to be tested against the history of board states that leads to the non-initial reset state, rather than the history of boards stored in the rollout buffer.

B.4 Reset Behavior and Support for Search

In order to support search, the simulator needs to “pin” the simulation of boards to start from a given state. This ability is implemented in our code by asking the `StrategoRolloutBuffer` to reset terminated boards not from an initial state, but rather from a custom non-initial state.

C Information State Representation

The representation of the information state for each player comprises two components:

- A representation of the current *board state*; and
- An encoding of the past K moves.

We describe each component in detail in the following subsections.

C.1 Representation of the Board State

We encode information about the state of the board and the history of the game as collection of channels, each of which is a 10×10 matrix of floating point numbers. The channels can be divided into different logical subgroups, as follows.

Basic piece properties The following channels expose basic information regarding the pieces on the board, including their visibility and whether they have moved in the past.

<i>Channel</i>	<i>Description</i>
0–11	Indicator of whether the piece occupying each square belongs to the player and has type T , for each of the 12 types $T \in \{\text{Spy, Scout, Miner, Sergeant, Lieutenant, Captain, Major, Colonel, General, Marshal, Flag, Bomb}\}$.
12–23	For each square, probability that it belongs to the opponent and has type T for each of the 12 types $T \in \{\text{Spy}, \dots, \text{Bomb}\}$, assuming the setup and moves were selected uniformly at random.
24–35	Same as channels 12–23, but from the point of the opponent player.
36	Indicator of what squares on the board are occupied by hidden pieces of the player
37	Indicator of what squares on the board are occupied by hidden pieces of the opponent
38	Indicator of what squares on the board are empty
39	Indicator of what squares on the board are occupied by pieces of the player that moved

40	Indicator of what squares on the board are occupied by pieces of the opponent that moved
41	Fraction of the maximum number of moves per game (Section D.1) that have been exhausted
42	Fraction of the maximum number of moves between attacks (Section D.1) that have been exhausted

Threats, evasions, missed opportunities to attack The following channels encode information regarding the historical behavior of pieces, including whether pieces threatened or evaded other pieces, and whether pieces opted not to attack other pieces when they had the opportunity.

For the purposes of the next planes, we recall that a *threat* is a movement of one's piece so as to make the piece adjacent to a piece of the opponent. The opponent's piece is considered *threatened*.

<i>Channel</i>	<i>Description</i>
43–52	Indicator of whether the piece occupying the square belongs to the player, and has ever threatened a piece of the opponent which, at the time of the threat, was a revealed piece of type $T \in \{\text{Spy}, \dots, \text{Marshal}\}$.
53	Indicator of whether the piece occupying the square belongs to the player, and has ever threatened a piece of the opponent which, at the time of the threat, was hidden.

An *evasion* is realized when a piece that was threatened in the immediately preceding move is moved away from the threatening piece, to an empty square.

<i>Channel</i>	<i>Description</i>
54–63	Indicator of whether the piece occupying the square belongs to the player, and has ever evaded a threat from an opponent's piece which, at the time of the threat, was a revealed piece of type $T \in \{\text{Spy}, \dots, \text{Marshal}\}$.
64	Indicator of whether the piece occupying the square belongs to the player, and has ever evaded a threat from an opponent's piece which, at the time of the threat, was hidden.

Finally, we define a piece to be *actively adjacent* to an opponent's piece when the player controlling the former has had the chance to attack the latter, but opted not to.

<i>Channel</i>	<i>Description</i>
----------------	--------------------

65–74	Indicator of whether the piece occupying the square belongs to the player, and has ever been actively adjacent to an opponent’s piece which, at the time of the adjacency, was a revealed piece of type $T \in \{\text{Spy}, \dots, \text{Marshal}\}$.
75	Indicator of whether the piece occupying the square belongs to the player, and has ever been actively adjacent to an opponent’s piece which, at the time of the adjacency, was hidden.

The same information regarding threats, evasions, and active adjacencies are then reported from the point of view of the opponent.

<i>Channel</i>	<i>Description</i>
76–85	Same as channels 43-52, but from the point of view of the opponent player.
86	Same as channel 53, but from the point of view of the opponent player.
87–96	Same as channels 54-63, but from the point of view of the opponent player.
97	Same as channel 64, but from the point of view of the opponent player.
98–107	Same as channels 65-74, but from the point of view of the opponent player.
108	Same as channel 75, but from the point of view of the opponent player.

Captured pieces The next group of channels encodes the starting positions of the pieces that have been captured.

<i>Channel</i>	<i>Description</i>
109–119	Indicator of whether the given square was occupied, at the start of the game, by a piece of the player of type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$ that has now been captured.
120–130	Indicator of whether the given square was occupied, at the start of the game, by a piece of the opponent of type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$ that has now been captured.

Causes of death The following channels encode the locations in which pieces died, as well as what interaction caused their death. The following causes of death are contemplated:

1. The piece attacked an opponent piece of strictly stronger rank, which was revealed at the time of the attack.
2. The piece attacked an opponent piece of equal rank, which was revealed at the time of the attack.
3. The piece attacked an unrevealed opponent piece, and lost the battle.

4. The piece was revealed, and was attacked by an opponent piece (revealed or not). It lost the attack because it was strictly weaker.
5. The piece was revealed, and was attacked by an opponent piece (revealed or not). The battle was a tie.
6. The piece was unrevealed, and was attacked by an opponent piece (revealed or not).

<i>Channel</i>	<i>Description</i>
131–140	Whether a piece of the player of type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$ died in this square, due to cause of death 1.
141–150	Whether a piece of the player of type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$ died in this square, due to cause of death 2.
\vdots	\vdots
181–190	Whether a piece of the player of type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$ died in this square, due to cause of death 6.
191–250	Same as channels 131–190, but for the opponent.

Protection moves The final group of channels refers to protection moves. When a player's piece A threatens a piece B of the opponent, the opponent might respond by moving one of their pieces, say C , to become adjacent to B . This way, if the player follows through with the aggression, the opponent can attack A with C . Moving C to become adjacent to B as a response to a threat from the A is a *protection* move. More precisely, we say that piece C protected piece B against piece A ; equivalently, piece B was protected by piece C against piece A . We extend the same terminology to the case in which B is an empty square.

<i>Channel</i>	<i>Description</i>
251–261	Whether the piece in the square belongs to the player, and has ever protected a piece of the player which, at the time of the protection, had revealed type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$.
262	Whether the piece in the square belongs to the player, and has ever protected an empty square.
263	Whether the piece in the square belongs to the player, and has ever protected a piece of the player which, at the time of the protection, was not revealed.

<i>Channel</i>	<i>Description</i>
----------------	--------------------

264–274	Whether the piece in the square belongs to the player, and has ever protected against a piece of the opponent which, at the time of the protection, had revealed type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$.
275	Padding plane (all zeros).
276	Whether the piece in the square belongs to the player, and has ever protected against a piece of the opponent which, at the time of the protection, was not revealed.

<i>Channel</i>	<i>Description</i>
277–287	Whether the piece in the square belongs to the player, and has ever been protected by a piece of the player which, at the time of the protection, had revealed type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$.
288	Padding plane (all zeros).
289	Whether the piece in the square belongs to the player, and has ever been protected by a piece of the player which, at the time of the protection, was not revealed.

<i>Channel</i>	<i>Description</i>
290–300	Whether the piece in the square belongs to the player, and has ever been protected against a piece of the opponent which, at the time of the protection, had revealed type $T \in \{\text{Spy}, \dots, \text{Marshal}, \text{Bomb}\}$.
301	Padding plane (all zeros).
302	Whether the piece in the square belongs to the player, and has ever been protected against a piece of the opponent which, at the time of the protection, was not revealed.

<i>Channel</i>	<i>Description</i>
303–354	Same as channels 251–315, but for the opponent.

Additional information In addition, the network has access to the following information.

<i>Channel</i>	<i>Description</i>
----------------	--------------------

355–455	Starting location of each piece on the board, one-hot encoded. Specifically, channel $355+k$ encodes in position $(i, j) \in \{1, \dots, 10\}^2$ whether the piece currently occupying square (i, j) started from square k , according to a row-major ordering of the squares.
---------	--

C.2 Representation of recent moves

In addition to the board state channels described above, we also include channels that encode the most recent 32 moves. Each move is encoded in a channel, for a total of 32 additional channels. For each move, a 10×10 matrix of floating point numbers encodes the source and destination square of the movement, with the convention that a $+1.0$ is written in the destination square and a -1.0 in the source square. All other squares are filled with zeros.

D Implementation Details

D.1 Game Rules

There are two main parameterized game settings, detailed in Table 17. One is the k for the k -move rule, which states that a draw is declared after k consecutive battleless moves. For training, we used a 100-move rule. We found that this outperformed training under a 200-move rule—despite that evaluation took place under a 200-move rule.¹ This may be because it was effective at discouraging dawdling, thereby increasing the quality of the generated data. The other game setting is the maximum game length before a draw is declared, which was set to 4,000. This was imposed for edge case safety and is not an actual rule of Stratego. In practice, we found that it virtually never triggered when paired with a 100-move rule.

Table 17: Game rules training hyperparameters.

Hyperparameter	Value
Number of consecutive battleless moves after which a draw is declared	100
Number of moves after which a draw is declared	4,000

D.2 Data Collection

For data collection, each GPU simulated 1,536 independent environments in parallel, playing out 202 moves (101 for each player) for each of these environments between training iterations. (Over 16 GPUs, this yielded about 5 million transitions per training iteration.) The initial boards of new

¹The representation of the proximity of the rule to triggering is fractional (see channel 42 in Section C.1); so n moves without a battle was represented as $n/100$ during training, but $n/200$ during testing.

games were sampled using a pool of pre-generated setups (sampled from the setup network), which contained 1,000 setups per player per GPU and was regenerated after each training iteration. These hyperparameters are listed in Table 18.

Table 18: Data collection hyperparameters.

Hyperparameter	Value
Number of parallel environments	1,536 per GPU
Generated setups per player	1,000 per GPU
Number of moves between training iterations	202

D.3 Setup Learning

Given a setup prefix (meaning a setup partially filled in under row-major order), the setup network produces three outputs:

1. A categorical estimate of the probabilities of the possible outcomes of the game (win, loss, draw), given the setup prefix.
2. A real-valued estimate of the conditional entropy of the setup, given the setup prefix.
3. A distribution over the next piece placement, given the setup prefix.

We trained these outputs on all setups for games that were finished during the last data collection period were included in the training data. Since most games span multiple reinforcement learning iterations, the training data was slightly off-policy.

Table 19: Setup learning notation.

Symbol	Meaning
$\bar{\sigma}$	Setup for which a game was completed.
σ	A prefix of $\bar{\sigma}$.
θ_t	Parameters that generated $\bar{\sigma}$.
θ	Current parameters.
$\mathcal{H}(\bar{\sigma} \mid \sigma; \theta_t)$	Conditional entropy of $\bar{\sigma}$ given σ under θ_t .
$h_{\theta_t}(\sigma)$	Predicted conditional entropy of setup prefix σ under θ_t .
$h_{\theta}(\sigma)$	Predicted conditional entropy of setup prefix σ under θ .
o	Outcome in $\{\text{win}=1, \text{loss}=-1, \text{draw}=0\}$ of the game started from $\bar{\sigma}$.
$v_{\theta}(o \mid \sigma)$	Predicted probability of outcome o given σ under θ .
$\mathbb{E}[v_{\theta_t}(\sigma)]$	Predicted expected outcome (with win=1, loss=-1, draw=0) given σ under θ_t .

Continued on next page

Table 19: Setup learning notation. (Continued)

Symbol	Meaning
α	Regularization temperature (coefficient for entropy maximization).
σ^+	Immediate successor to σ on the path to $\bar{\sigma}$.
$\pi_{\theta_t}(\sigma^+ \sigma)$	Probability of σ^+ given σ under θ_t .
$\pi_{\theta}(\sigma^+ \sigma)$	Probability of σ^+ given σ under θ .
$\pi_{\theta_t}(\sigma)$	Distribution over next piece type given σ under θ_t .
$\pi_{\theta}(\sigma)$	Distribution over next piece type given σ under θ .

Using the notation in Table 19, we used the conditional entropy prediction loss

$$\mathcal{L}_h = \left(\frac{\mathcal{H}(\bar{\sigma} | \sigma; \theta_t)}{10} - h_{\theta}(\sigma) \right)^2; \quad (1)$$

the value prediction loss

$$\mathcal{L}_v = -\log v_{\theta}(o | \sigma); \quad (2)$$

the estimated advantage

$$\delta = (o - \mathbb{E}[v_{\theta_t}(\sigma)]) + \alpha(\mathcal{H}(\bar{\sigma} | \sigma; \theta_t) - h_{\theta_t}(\sigma));$$

and the policy loss

$$\mathcal{L}_{\pi} = -\min(r\delta, \text{clip}(r, 0.8, 1.2)\delta) + 0.1\text{KL}(\pi_{\theta}(\sigma), \pi_{\theta_t}(\sigma)), \quad (3)$$

where

$$r = \frac{\pi_{\theta}(\sigma^+ | \sigma)}{\pi_{\theta_t}(\sigma^+ | \sigma)}.$$

We weighted these losses as shown below

$$\mathcal{L}_{\text{setup}} = \mathcal{L}_{\pi} + \frac{1}{2}\mathcal{L}_v + \mathcal{L}_h. \quad (4)$$

We clipped the gradient norm for this loss at 0.5. We performed 5 epochs over the setups associated to games that completed over the previous data collection in batches of 1,024, using Adam with a learning rate of 5×10^{-5} . We updated an exponential moving average of the parameters—which was used for evaluation—after completion of the training iteration with a smoothing parameter of 0.999.

The above hyperparameter choices are summarized in Table 20.

Table 20: Setup learning hyperparameters.

Hyperparameter	Value
Adam learning rate	5×10^{-5}

Continued on next page

Table 20: Setup learning hyperparameters. (Continued)

Hyperparameter	Value
Batch size	1,024 per GPU
Number of epochs per training iteration	5
Importance ratio clipping parameter	0.2
Conditional entropy prediction loss coefficient	1
Reverse KL to data collection policy loss coefficient	0.1
Value loss coefficient	0.5
Regularization temperature	$\frac{0.1}{(\text{Training iteration number})^{0.3}}$
Maximum gradient norm	0.5
Normalizing constant for conditional entropy prediction	1/10
Exponential moving average smoothing factor	0.999

D.4 Move Learning

Given a position, the move network produces two outputs:

1. A categorical estimate of the probabilities of the possible outcomes of the game (win, loss, draw), given the position.
2. A distribution over moves that are legal given the position.

We trained these outputs on some but not all of the moves made over the last 202 simulator steps. Specifically, we included a move in the training data if both of the following were true:

1. The absolute magnitude of the estimated advantage was in the 0.75 quantile or above across the data for which advantages were estimated.
2. The absolute magnitude of the estimated advantage was 0.01 or above.

Table 21: Move learning notation.

Symbol	Meaning
x	A position of the acting player, represented as described in Section C.
$\{x'\}$	The sequence of positions subsequent to x for the same player.
θ_t	The parameters that played the move at position x .
θ	The current parameters.

Continued on next page

Table 21: Move learning notation. (Continued)

Symbol	Meaning
$v_{\theta_t}(x)$	Predicted win, loss, draw probabilities at x under θ_t .
$v_{\theta}(x)$	Predicted win, loss, draw probabilities at x under θ .
$\mathbb{E}[v_{\theta_t}(x)]$	Predicted expected outcome (with win=1, loss=-1, draw=0) at x under θ_t .
o	The outcome of the game in which x occurred (if game has finished).
δ	λ -return-estimated advantage for move played at x .
ξ	λ -return-estimated probability of winning, losing, and drawing from x .
m	The move that the acting player played at x .
$\pi_{\theta_t}(m \mid x)$	The probability of m at x under θ_t .
$\pi_{\theta}(m \mid x)$	The probability of m at x under θ .
ρ	Magnet policy that selects piece to move uniformly and moves it uniformly.
α	Reverse KL to magnet policy ρ loss coefficient.
$\pi_{\theta}(x)$	Probability of each move at x under θ .
$\pi_{\theta_t}(x)$	Probability of each move at x under θ_t .
$\rho(x)$	Probability of each move at x under ρ .

Using the notation of Table 21, we estimated the advantage δ using the λ -return with $\lambda = 0.5$ over the values $\{\mathbb{E}[v_{\theta_t}(x')]\}$ (and o if the game finished) and baseline $\mathbb{E}[v_{\theta_t}(x)]$. We estimated outcome probabilities ξ using the λ -return with $\lambda = 0.8$ over the vector values $\{v_{\theta_t}(x')\}$ (and o , as a one-hot vector, if the game finished).

Given these estimates, we updated the move network using the value loss

$$\mathcal{L}_v = \text{cross-entropy}(\xi, v_{\theta}(x)), \quad (5)$$

and policy loss

$$\mathcal{L}_{\pi} = -\min(r\delta, \text{clip}(r, 0.8, 1.2)\delta) + 0.1\text{KL}(\pi_{\theta}(x), \pi_{\theta_t}(x)) + \alpha\text{KL}(\pi_{\theta}(x), \rho(x)). \quad (6)$$

where

$$r = \frac{\pi_{\theta}(m \mid x)}{\pi_{\theta_t}(m \mid x)}.$$

We weighted these losses evenly, as shown below

$$\mathcal{L}_{\text{move}} = \mathcal{L}_{\pi} + \mathcal{L}_v.$$

We trained on this loss in 202 batches of positions grouped by simulator step. If not for filtering by advantage magnitude, the size of these batches would have been 1,536 per GPU; with advantage

filtering, they were 1/4 or less of that size. We updated the network using Adam with the learning rate schedule specified in Table 22 and a maximum gradient norm of 0.267. We performed only one epoch over this training data. Thereafter, we updated an exponential moving average of the parameters—which was used for evaluation—with a smoothing parameter of 0.999.

Table 22: Move learning hyperparameters.

Hyperparameter	Value
Importance ratio clipping parameter	0.2
Advantage filtering quantile threshold	0.75
Advantage filtering magnitude threshold	0.01
Exponential moving average smoothing factor	0.999
Advantage estimation λ	0.5
Outcome estimation λ	0.8
Reverse KL to data collection policy loss coefficient	0.1
Adam learning rate	$\text{clip} \left(\frac{0.5}{(\text{Training iteration number})^{1.1}}, 5 \times 10^{-6}, 1 \times 10^{-4} \right)$
Reverse KL to magnet policy loss coefficient	$\frac{0.05}{(\text{Training iteration number})^{0.3}}$
Maximum gradient norm	0.267
Number of epochs per training iteration	1
Value loss coefficient	1
Batch size if not for filtering	1,536 per GPU

D.5 Belief Learning

Given a position, the belief network produces a distribution over configurations of opponent hidden pieces. We trained the belief network to minimize the log likelihood of the ground-truth opponent hidden pieces from a stream of self-play games of the final setup and move networks, given the associated positions—represented as described in Section C—as input.

D.6 Networks

Details about the networks are given in Table 23, Table 24, and Table 25. Pre-layernorm (i.e., layernorm [32] before the sublayer rather than after the residual [33]) was used for all of the networks. The belief architecture is detailed in Figure 7.

Table 23: Setup network information.

Hyperparameter	Value
Depth	4
Embedding dimension	512
Number of heads	8
Learned positional embedding initialization standard deviation	0.1
Feedforward dimension	2,048
Total parameters	12.6 million

Table 24: Move network information.

Hyperparameter	Value
Depth	8
Embedding dimension	384
Number of heads	8
Learned positional embedding initialization standard deviation	0.1
Feedforward factor	1,536
Total parameters	14.7 million

Table 25: Belief network information.

Hyperparameter	Value
Encoder depth	6
Number of decoder blocks	4
Number of heads	8
Embedding dimension	512
Dropout	0.2
Learned positional embedding initialization	Kaiming uniform [34]
Feedforward dimension	2,048
Total parameters	57.1 million

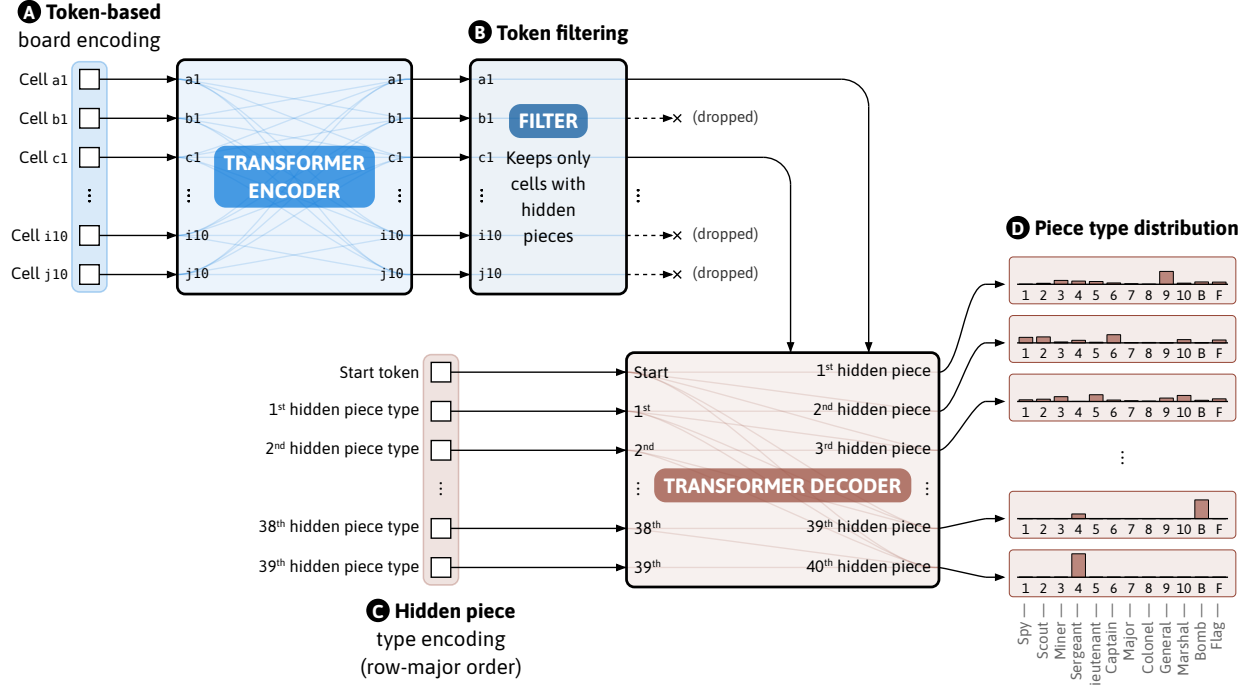


Figure 7: Diagram of belief architecture. **A** The input to the encoder is a tokenized representation of the occupiable squares with learned absolute positional embeddings, similar to the move network. **B** The output of the encoder is filtered to keep only tokens corresponding to squares occupied by hidden pieces of the opponent. Keys and values for these tokens are passed into transformer decoder layers. The inputs **C** and outputs **D** to the decoder are the types and predictions about the types of the opponent hidden pieces, respectively, in row-major order; the inputs use learned positional embeddings.

D.7 Search

The search proceeds in the following steps.

1. The search samples roughly

$$\frac{1,000}{\text{number of legal actions}}$$

possible configurations of the types of the hidden pieces of the opponent from the belief network, given the position x .

2. The search performs 1,000 rollouts of depth 40. There is at least one rollout for each combination of legal move and sampled opponent configuration. After the initial move, which is forced, the subsequent 39 moves are played according to the move network.
3. The search computes average values \hat{q} for each legal moves according to the value output of the move network at the positions reached by the rollouts starting from that move.
4. The search plays

$$M \sim \pi_{\text{search}}$$

where

$$\pi_{\text{search}} = \max_{\pi} \langle \hat{q}, \pi \rangle - \alpha \text{KL}(\pi, \rho) - \beta \text{KL}(\pi, \pi_{\theta}) \quad (7)$$

$$\propto \left[e^{\hat{q}} \rho^{\alpha} \pi_{\theta}^{\beta} \right]^{\frac{1}{\alpha+\beta}} \quad (8)$$

and where $\rho = \rho(x)$ is the magnet policy distribution over moves at position x , $\pi_{\theta} = \pi_{\theta}(x)$ is the move network distribution over moves at position x , α is the reverse KL coefficient to the magnet policy, and β is the reverse KL coefficient to the move network policy. The hyperparameters for this procedure are summarized in Table 26.

Table 26: Search hyperparameters.

Hyperparameter	Value
Reverse KL to magnet policy loss coefficient	0.002
Reverse KL to policy network loss coefficient	0.02
Number of rollouts	1,000
Rollout depth	40

E Training logs

We show loss values and other quantities for the setup network over the training run in Figure 8 and Figure 9. We show loss values and other quantities for the move network over the training run in Figure 10 and Figure 11. For both networks, dynamic damping guides the policy from taking large steps in a heavily regularized regime at the start of training to taking small steps in a weakly regularized regime by the end of training. For the setup network, this can be observed by the simultaneous reduction of 1) the entropy of the network and 2) both the KL and importance ratio clip rate between the network that is training and that which generated the setup. For the move network, this can be observed by the simultaneous 1) increase in KL divergence between the policy that is training and the magnet policy and decrease in entropy of the policy that is training and 2) the decrease in both KL divergence and importance ratio clip rate between the policy that is training and the policy that made the move.

Table 27 displays various count-based metrics for the training run.

Table 27: Counts from the training run of Ataraxos.

Metric	Value
Number of finished games	163×10^6
Number of environment steps	208×10^9
Number of gradient steps for move network	8.56×10^6
Number of gradient steps for setup network	99.5×10^3

Figure 12 shows other information about the training process and games, including training times, game outcomes, game lengths and move priority significance.

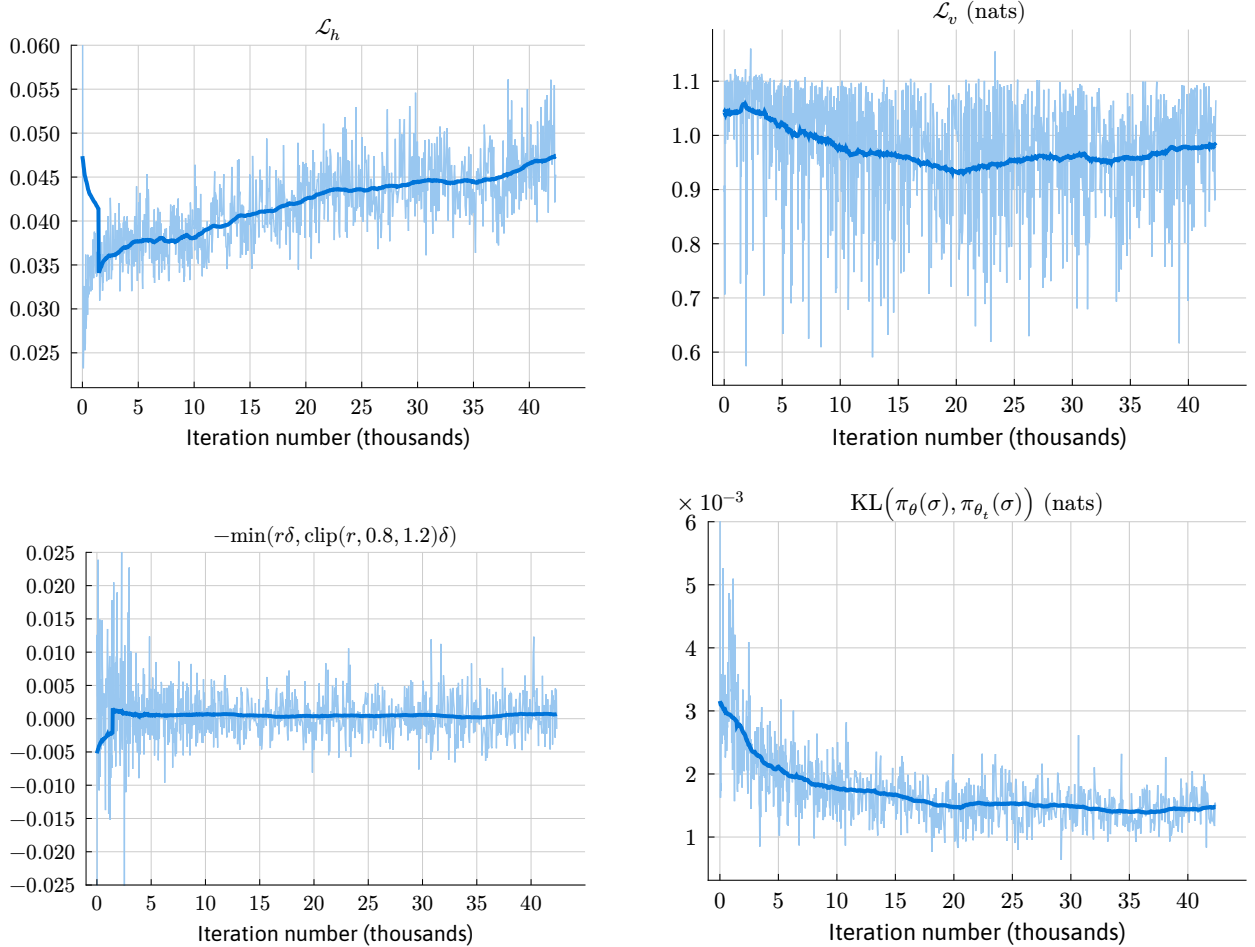


Figure 8: Loss values for setup learning over the training run.

F Other Experiments

We show reinforcement learning ablations in Figure 13 and the performance of search across varying hyperparameters in Table 28.

G Wealth Ratio Analysis of Evaluation

Given the non-i.i.d. nature of the evaluation games (due to both to the fact that humans do not play fixed strategies and especially because Pim Niemeijer was in a position to adversarially adapt to Ataraxos over the course of the series), it is reasonable to consider approaches alternative to

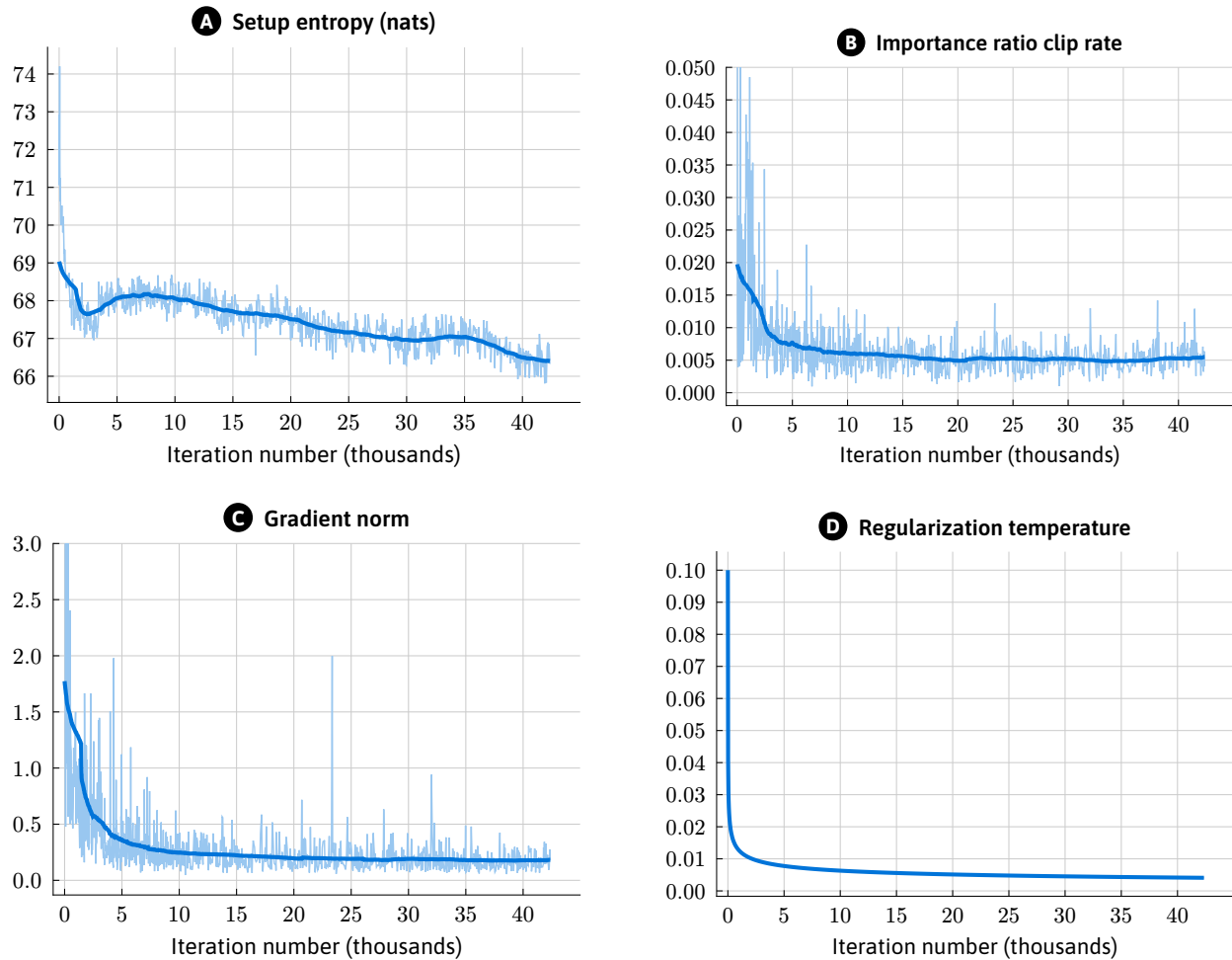


Figure 9: Other setup-related values over the training run. **(A)** Entropy of the setup distribution. **(B)** Proportion of data whose importance ratio is clipped. **(C)** Norm of the gradient of the loss. **(D)** Entropy maximization coefficient α .

statistical significance for quantifying the significance (used informally) of the results. We discuss one such alternative based on betting.

G.1 Betting Market

Consider a betting market on the sequence of game outcomes. The market works as follows:

1. The bettor commits wealth_0 units of capital. This committed capital will be “locked up” until the end of the sequence of games in the sense that the bettor may not withdraw it.
2. Prior to each game, the bettor bets the entirety of their committed capital $\text{wealth}_t = \pi_t(w)\text{wealth}_t + \pi_t(\ell)\text{wealth}_t$ across win and loss. The bettor may condition this allocation π_t on the outcomes of the previous games X_1, \dots, X_{t-1} .
3. After each game, the bettor’s committed capital is modified based on the realized outcome.

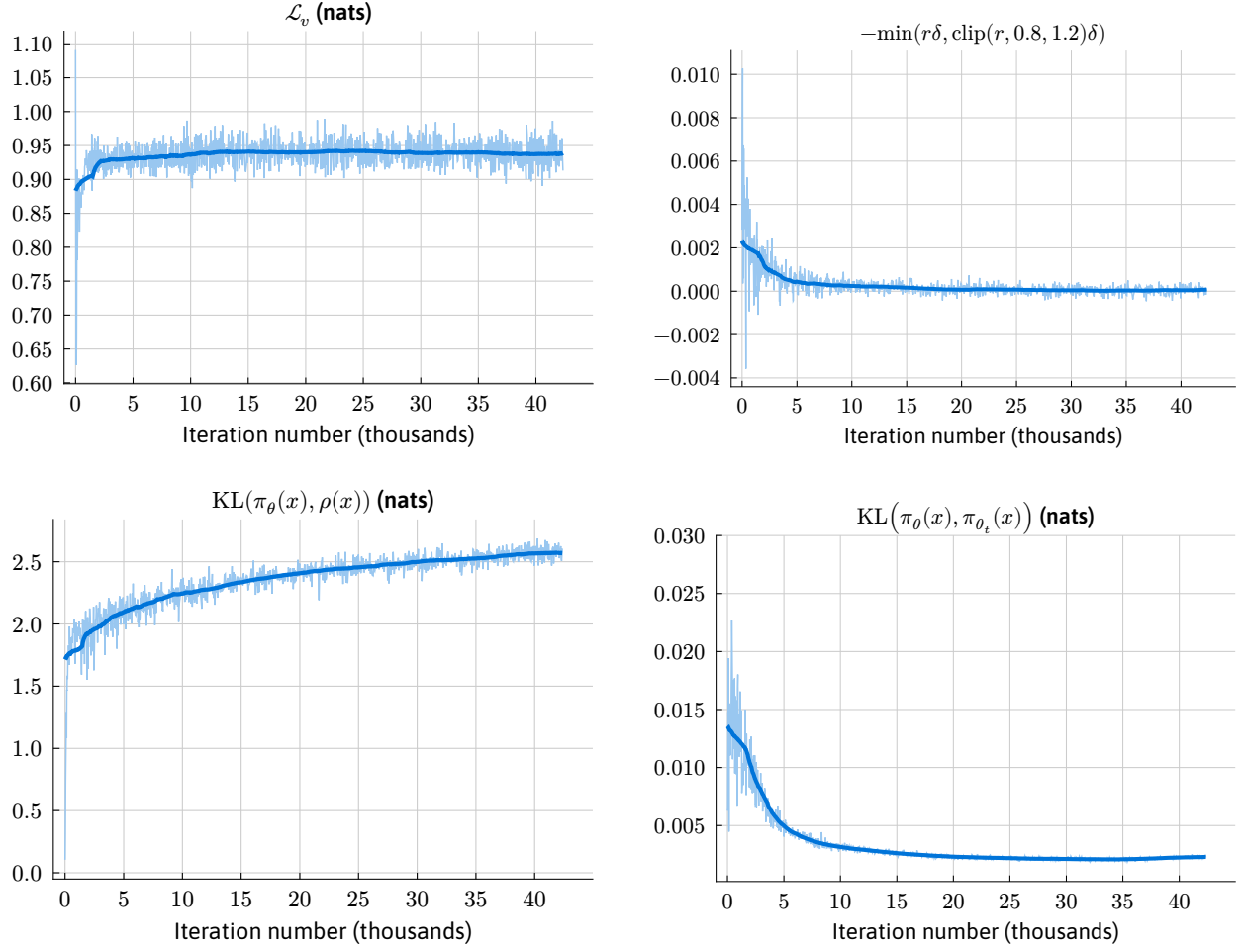


Figure 10: Loss values for move learning over the training run.

- If the game $X_t = d$ was a draw, the bettor's updated committed capital $\text{wealth}_{t+1} = \text{wealth}_t$ remains unchanged.
 - If the game $X_t \in \{w, \ell\}$ was a win or loss, the bettor's updated committed capital $\text{wealth}_{t+1} = 2\pi_t(X_t)\text{wealth}_t$ is twice the amount they wagered on the realized outcome.
4. At the end of the sequence of games, the bettor is returned wealth_N .

In this market, the bettor maximizes expected final wealth by allocating all of their committed capital to the more probable outcome at each step. However, this approach carries a high risk of going bust. A more conservative approach is maximizing expected log final wealth, which is achieved by allocating capital proportionally to outcome probability [35].

Thus, a natural measure of the performance of a bettor is the log wealth regret of that bettor (relative to the best-in-hindsight time-invariant bettor of some comparator class). This is ordinarily equivalent, as can be seen by exponentiation, to the wealth ratio between the bettor and the best-in-hindsight time-invariant comparator.

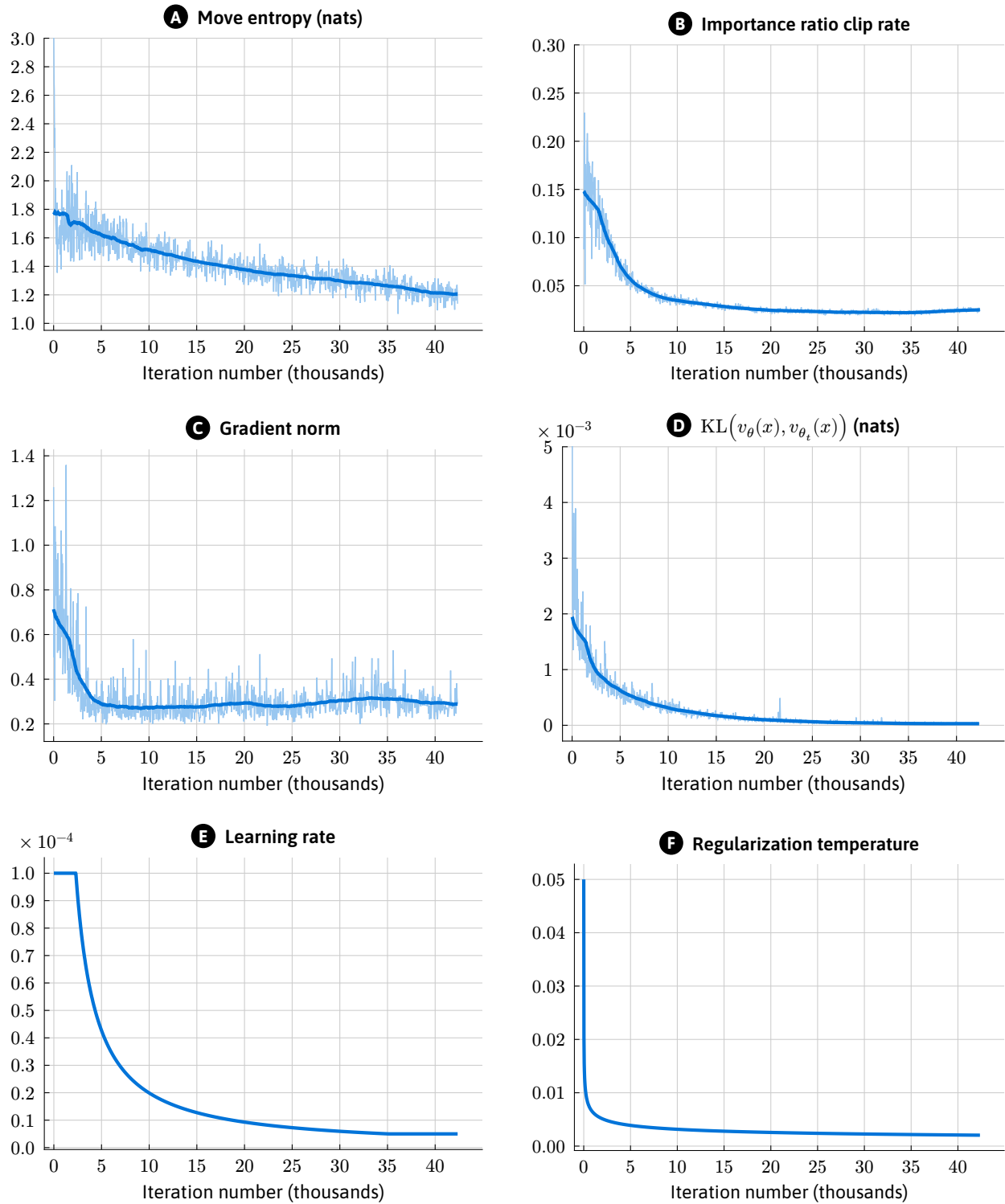


Figure 11: Other move-related values over the training run. **A** Entropy of moves over self-play distribution. **B** Proportion of data whose importance ratio is clipped. **C** Norm of the gradient of the loss. **D** Value function change. **E** Adam learning rate. **F** Coefficient to magnet policy α .

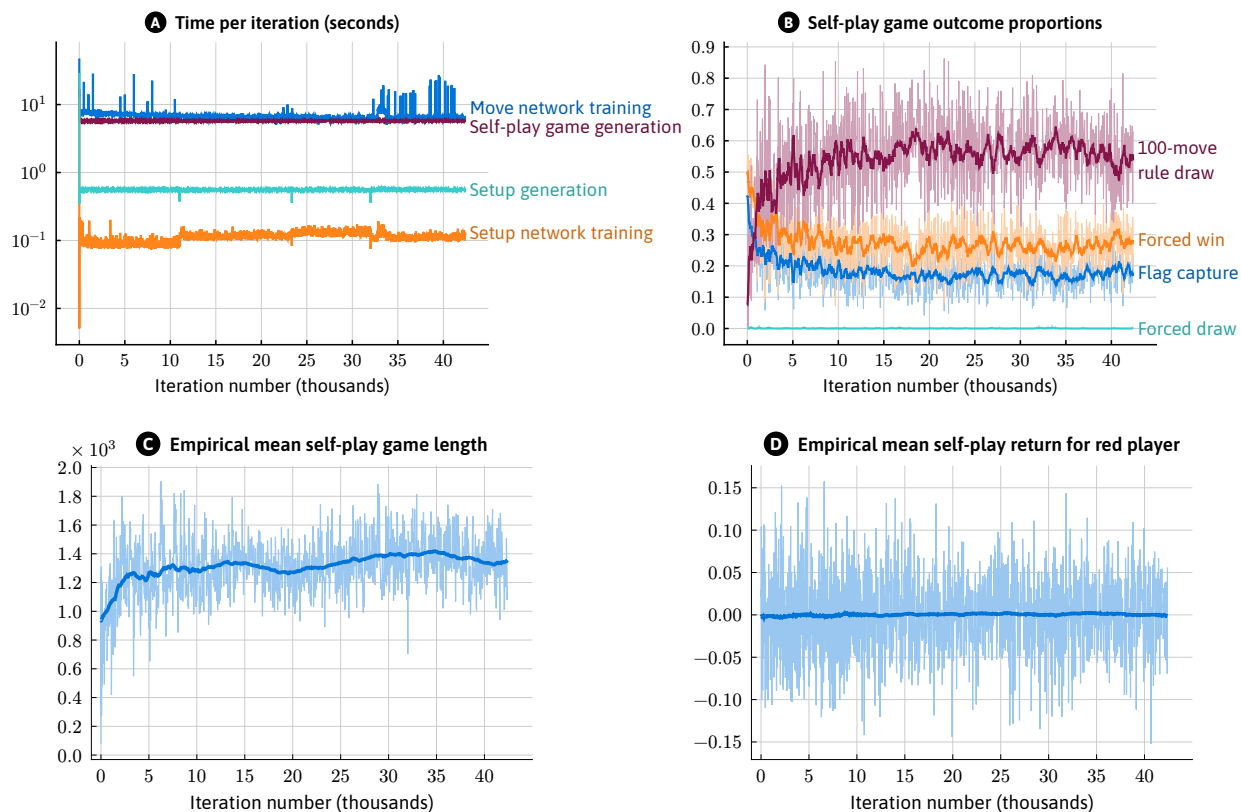


Figure 12: Other information about the training process and games. **A** Time expenditure of different components of the reinforcement learning loop. Iteration speed is bottlenecked by training and data collection for the move network. **B** Proportion over training of each of the four possible game endings: Flag capture, forced win (i.e., by virtue of the other player having no legal moves), forced draw (i.e., by virtue of neither player having legal moves), and a 100-move rule draw (i.e., by virtue of there having been 100 consecutive battleless moves). Ataraxos playing against itself draws a much higher proportion of games than humans playing against one another. **C** The average length of self-play games over training. Ataraxos playing against itself results in much longer games than humans playing against one another. **D** The average return (win=1, loss=-1, draw=0) for the red player (the first moving player) over training. The roughly equal returns of the red and blue players in the self-play of Ataraxos games supports the human belief that move priority is not strategically significant in Stratego.

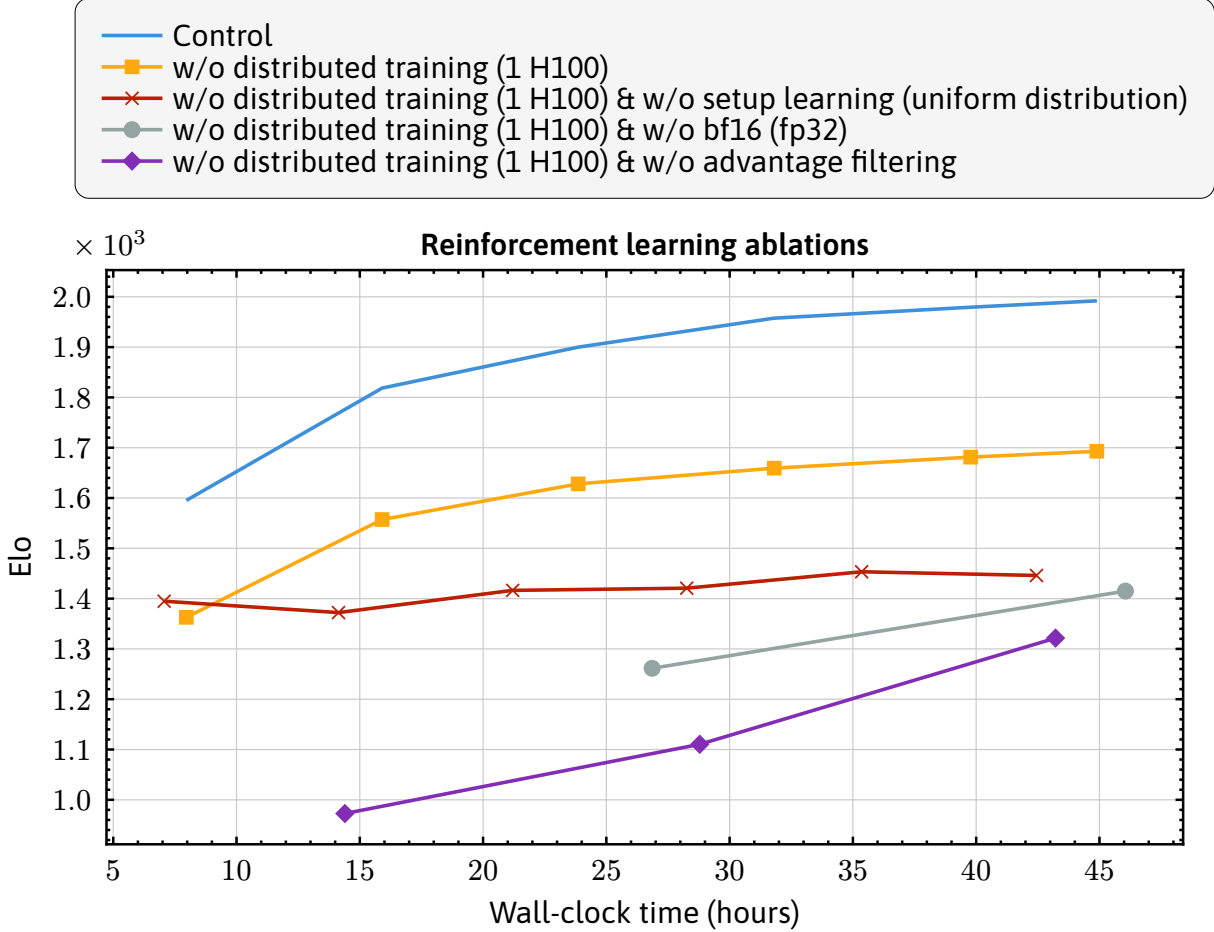


Figure 13: Single-seed ablations of the training run. Without distributed training (yellow), Ataraxos follows a similarly shaped trajectory of improvement over the course of scheduled annealing, but at a substantially lower Elo. When setup learning is also removed (and replaced by a fixed uniform distribution over setups), the shape of the Elo trajectory flattens—both because of the weakness of uniformly distributed setups and because of the bad assumptions the associated self-play distribution causes the move network to make about the pieces of its opponents. Removing bfloat16 and advantage filtering substantially slows iterations due to more expensive network calls and larger training datasets, respectively. Removing advantage filtering also causes a large increase in move entropy and large decrease in sample efficiency.

Table 28: Elo of search with varying hyperparameters. The first row shows directly sampling from the policy network (i.e., no search); the second shows the search that was used in the evaluation. Both Elo and time expended increase with the depth of the search and the number of rollouts. Reverse KL regularization strength has a large effect on performance. Removing the reverse KL term to the move network devastates performance—causing it to drop well below that of the move network—because the search overfits to idiosyncrasies of the move network that fail to generalize to other opponents.

Network KL coefficient	Magnet KL coefficient	Depth	Number of rollouts	H100 seconds per move	Elo (95% CI)
∞	N/A	N/A	N/A	~ 0.004	2095 (2091, 2100)
0.02	0.002	40	1000	~ 1.26	2218 (2177, 2266)
0.02	0.002	10	1000	~ 0.46	2189 (2165, 2215)
0.02	0.002	40	200	~ 0.50	2189 (2164, 2216)
0.02	0.002	10	200	~ 0.26	2169 (2153, 2186)
0.02	0	40	1000	~ 1.26	2174 (2138, 2215)
0.05	0.002	40	1000	~ 1.26	2160 (2124, 2201)
0	0.002	40	1000	~ 1.26	1733 (1701, 1761)

G.2 Quantifying Significance

We can use the wealth ratio as quantification the significance (used informally) of the performance of Ataraxos over the series by selecting a bettor constrained to bet in favor of Ataraxos and the comparator class of time-invariant bettors either betting evenly or against Ataraxos. This can wealth ratio can be interpreted as the inverse of an anytime-valid conservative p-value [36] under certain conditions [37], but is of independent interest as a quantification of significance (again used informally) even when these conditions are not met.

G.3 Bettor Specification

We use modified KT bettor [38]:

$$\pi_t(w) = \frac{\max(\frac{W_t + L_t}{2}, W_t) + 2/3}{W_t + L_t + 1},$$

where W_t is the number of cumulative wins up to game t and L_t is the number of cumulative losses up to game t . This bettor:

1. Allocates 2/3 of its capital on the first game being a win.
 2. Under worse than 50-50 performance, trends its allocation toward 50-50.
 3. Under better than 50-50 performance, trends its allocation toward the empirical probabilities.
- Since Ataraxos won more than half of the games, the best-in-hindsight time-invariant either betting against Ataraxos or betting evenly is the one that bets evenly (thereby maintaining its wealth).

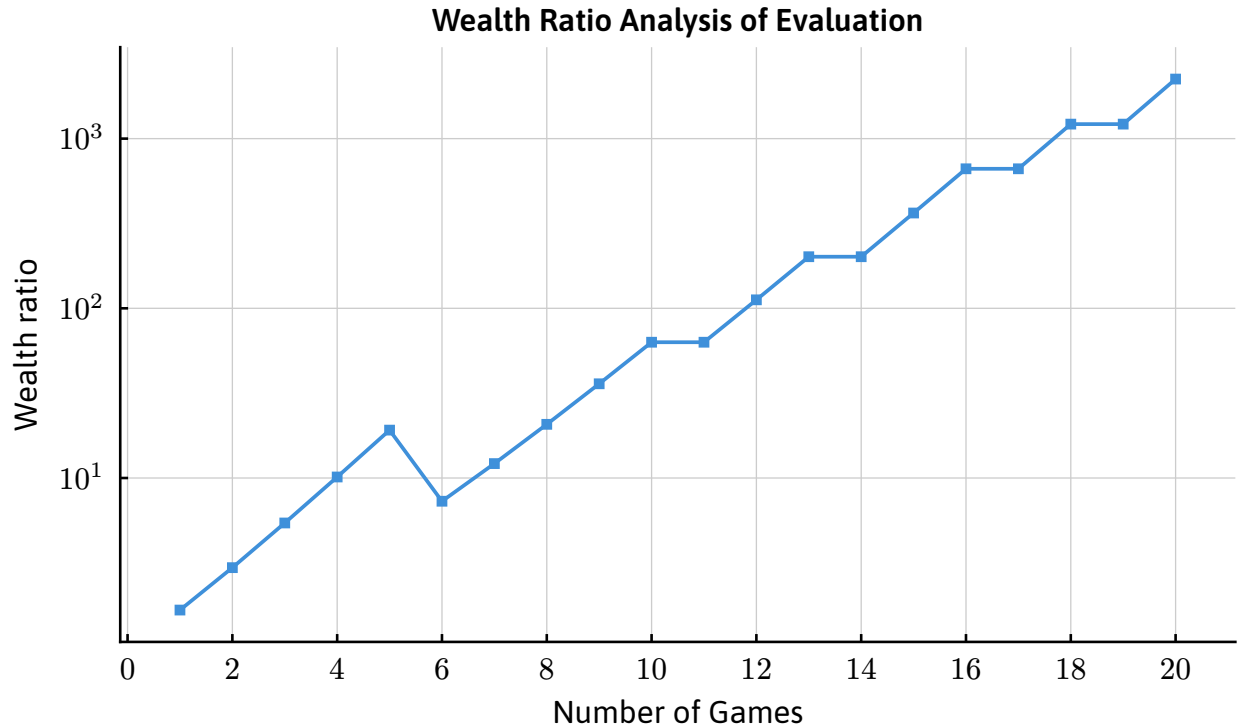


Figure 14: The ratio of wealth between a modified KT-bettor betting in favor of Ataraxos and the best-in-hindsight time-invariant bettor not betting in favor Ataraxos over the course of the series.

We show the resulting wealth ratio over the the series in Figure 14. By the end of the evaluation, the Ataraxos-favoring bettor accumulated more than 2000 times more wealth.

H Learned Setups

The probability distribution for each piece in a setup chosen by Ataraxos is given in Figure 15, assuming that the Flag is on the left side of the setup. The probabilities for the right side are symmetric, as Ataraxos performs a left-right orientation randomization to its setups after sampling from the setup network to enforce symmetry. Ataraxos uses a bombed in Flag (i.e., a Flag that is enclosed by Bombs, such as in Figure 6) about 2/3 of the time. The setups Ataraxos used in the series against Pim Niemeijer are shown in Figure 16 (games 1–10) and Figure 17 (games 11–20).

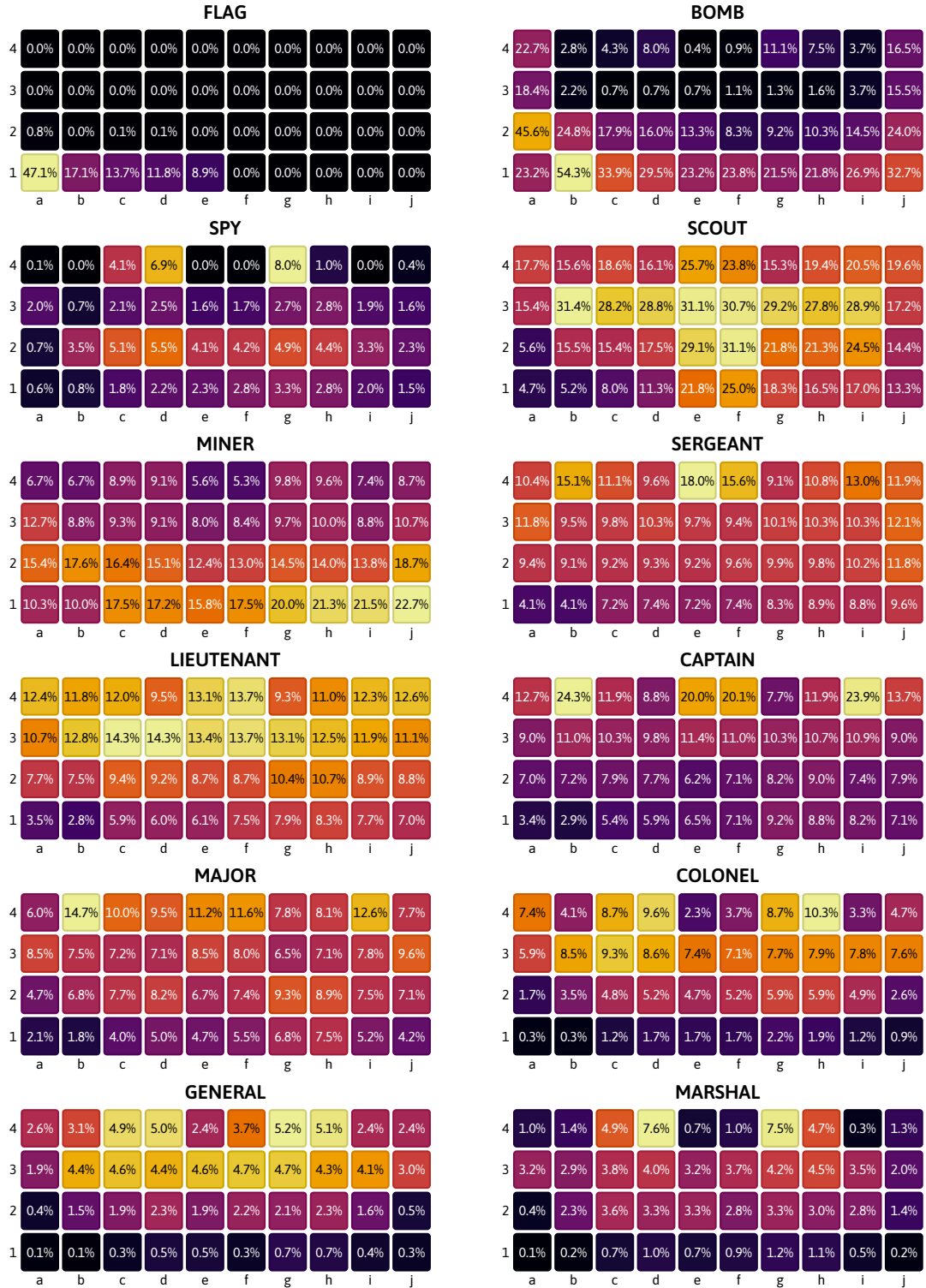


Figure 15: Estimated probabilities of pieces being different types in setups chosen by Ataraxos, assuming the Flag is on the left side of the setup.

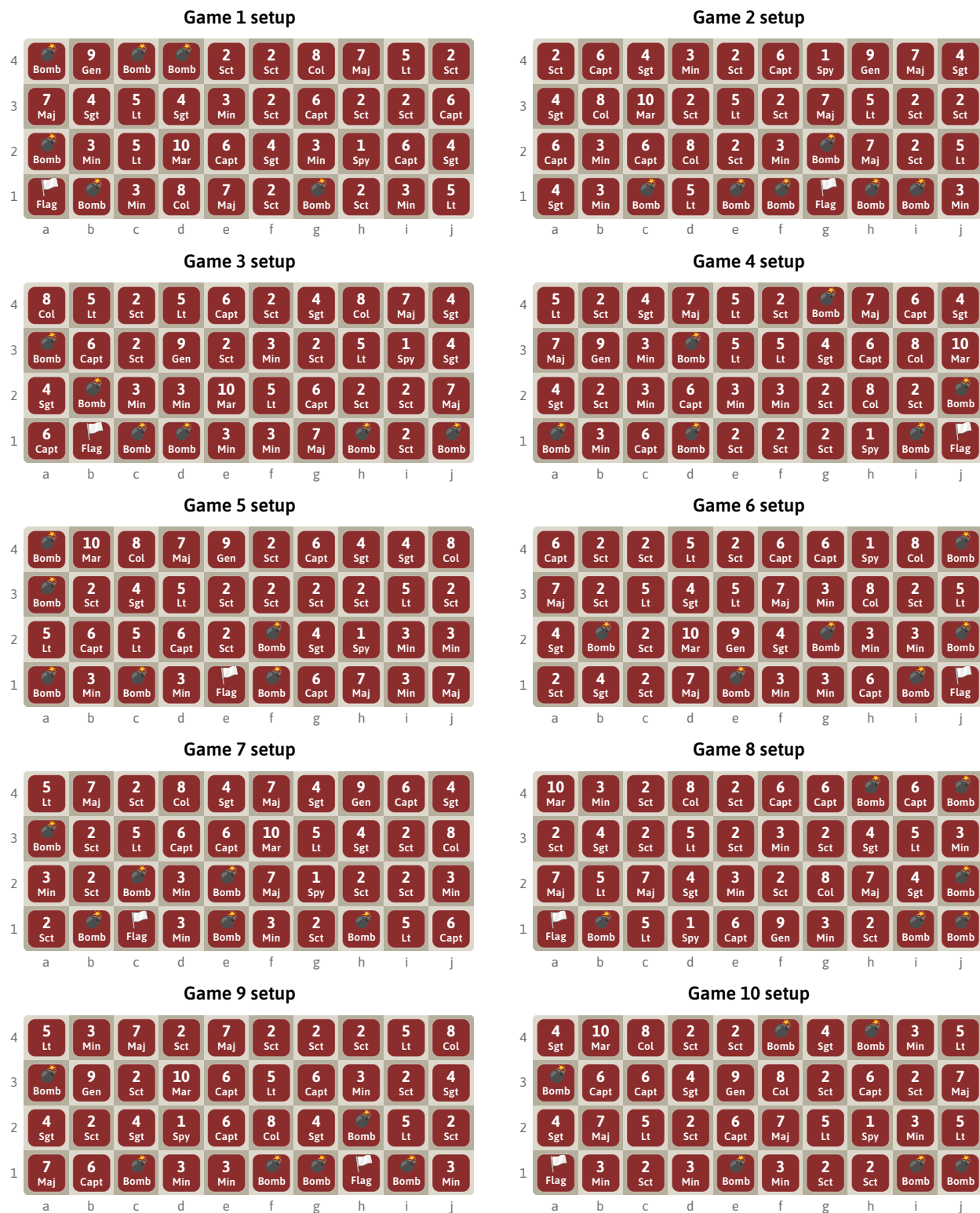


Figure 16: The setups Ataraxos played in games 1–10 in the series against Pim Niemeijer.

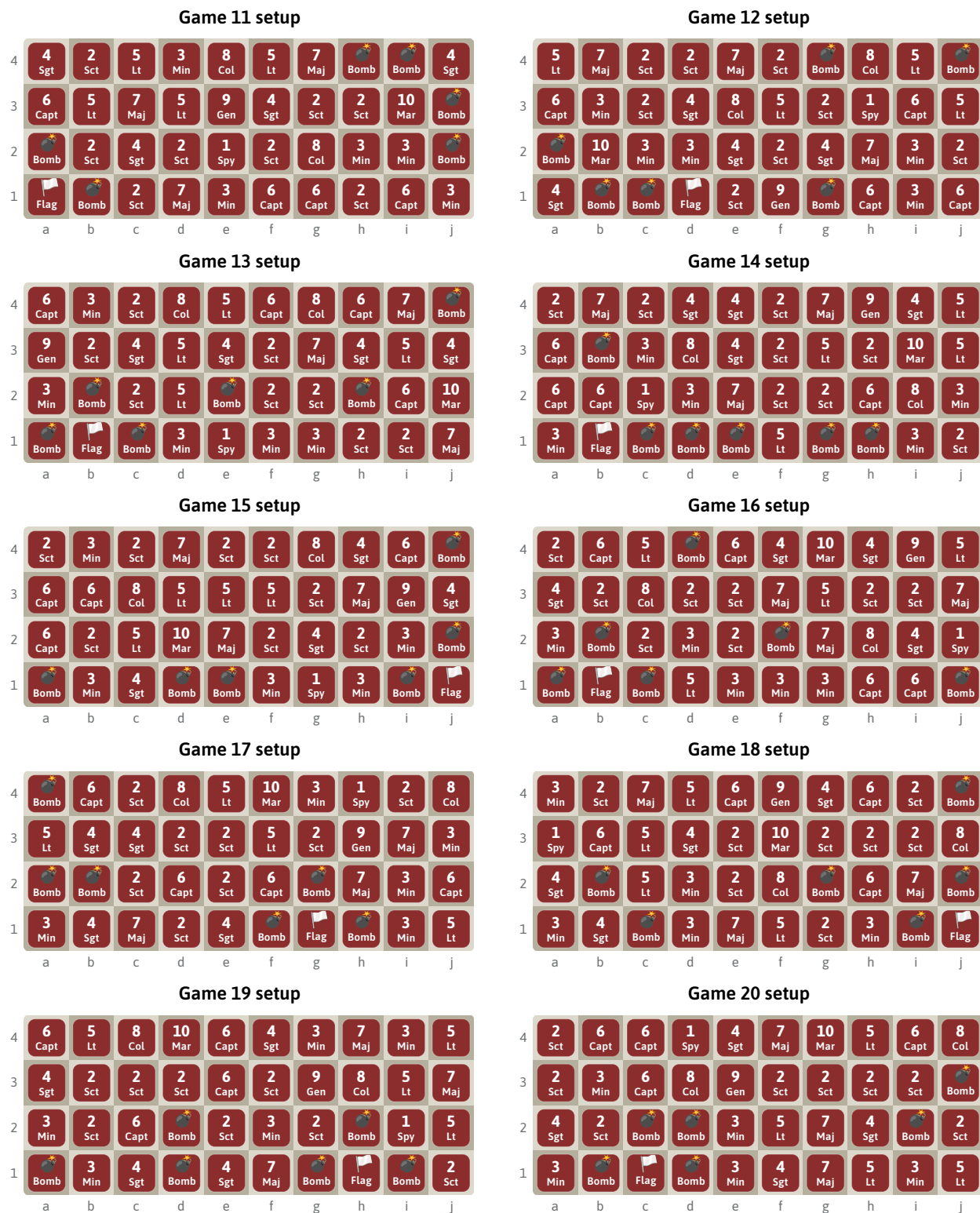


Figure 17: The setups Ataraxos played in games 11–20 in the series against Pim Niemeijer.

I Play Style

The play style of Ataraxos possesses differences from those of top human players (which are discussed, for example, in de Boer [39]), in terms of both setups and gameplay.

I.1 Setups

Players expressed that

- Ataraxos uses aggressive setups (i.e., ones in which high-value pieces are at or near the front), high Bombs (i.e., Bombs in the 3rd and 4th rows), and back corner Flags (which humans consider difficult to defend) more frequently than humans.
- Ataraxos generally uses setups that have less predictable structure than those of humans.

I.2 Gameplay

Players expressed that

- Ataraxos moves in a manner that makes its pieces hard to read (i.e., discern the types of) relative to the movement patterns of human players.
- Ataraxos is more willing than humans to accept a draw during the opening when progressing the game would have negative expected value (as happened in Game 11 against Pim Niemeijer).
- Ataraxos has a stronger preference than humans for preserving Scouts deep into the game.
- Ataraxos uses certain bluffs sparingly relative to humans, but uses other bluffs that humans consider altogether too risky to play.
- Ataraxos fights more bitterly than humans when it is behind, aggressively stalling and pestering to slow the progression of its opponent—tactics some humans consider rude.
- Ataraxos excels relative to humans at long-term positional play, punishing mistakes, defending, playing from an information deficit, transitioning from middle- to end-game, and negotiating favorable positions into wins and unfavorable positions into wins or draws.
- Ataraxos takes gambles that humans consider arrogant (in the sense of not respecting the opponent).
- Ataraxos feels preternaturally lucky, always seeming to have the pieces it needs in the right places, to have its gambles pay off, and to have its opponents do *as it wants them to do*.

J Contextualization vis-à-vis DeepNash

DeepNash [10] is an AI for Stratego that was developed by DeepMind.

J.1 Evaluation

DeepNash was evaluated on the website Gravon in April 2022, winning 42 of the 50 games counted by Perolat et al. [10]. However, by the time of the evaluation, the player base had largely moved away from Gravon (only 25 players are listed for the final ranking of 2022 [40]). As a result, the

opponents matched against DeepNash were far from the level of elite humans (let alone all-time great) [41]. This circumstance was exacerbated by the facts that 1) these opponents may not have been trying their hardest (one-off online games do not generally elicit the same effort level as competition play), 2) these opponents were not aware that they were playing against a bot (giving them no reason to look for anti-bot exploits), and 3) these opponents were not aware that DeepNash played a fixed strategy (meaning they were not aware that it was safe to exploit the same weakness across multiple games). Even under these conditions, Perolat et al. [10] report that DeepNash did not achieve the top ranking on the site.

We reached out to DeepMind to ask if they would allow an evaluation between DeepNash and Ataraxos, offering to build any infrastructure necessary for the evaluation. DeepMind responded that it would not be possible as the code for DeepNash is no longer functional.

J.2 Cost

Perolat et al. [10] state that DeepNash was trained on 1024 TPU nodes. To the recollection of the corresponding author with whom we spoke, this training took between 2 and 3 months and used TPU v3s. Under current pricing [42], such a training run would roughly cost between \$3,000,000 and \$4,500,000, depending on how much of the third month was used.

The reinforcement learning models and belief models of Ataraxos were trained on 16 H100s for one week and 4 H100s for 4 days, respectively. Such a run costs less than \$8,000 at current prices [43].

K Opportunities for Further Improvement

Learning Ataraxos accesses history through features rather than learning across time directly. For the belief model, we found that much stronger compute-normalized performance could be attained by interleaving spatial attention with either temporal attention [44] or recurrent models. However, for reinforcement learning, we did not observe an analogous out-of-the-box compute-normalized performance improvement due to the additional runtime and memory requirements of such architectures, as well as their interplay with advantage filtering. We believe such architectures could achieve stronger performance given a sufficiently large compute budget or provided with additional runtime, memory, and/or design optimizations.

Search The amount of improvement attainable by the search procedure of Ataraxos is ultimately bounded because it is mimicking a single update step. A more sophisticated search algorithm would be able to leverage arbitrary amounts of additional compute to continue to improve the policy. One possible route toward this end would be to incorporate innovations from knowledge-limited subgame solving [45].