Sample Efficient Game Strategy Through Active Demonstrations

Master's thesis in Computer science and engineering

Sigge Rajamäe

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Sample Efficient Game Strategy Through Active Demonstrations

Sigge Rajamäe

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Sample Efficient Game Strategy Through Active Demonstrations

Sigge Rajamäe

© Sigge Rajamäe, 2026.

Supervisor: Mattias Appelgren
Examiner: Kıvanç Tatar

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

Sample Efficient Game Strategy Through Active Demonstrations

Sigge Rajamäe
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

AlphaZero-style learning trains agents in game-playing purely through self-play, but requires millions of games before converging. In domains like robotics, imitation learning has been used to increase sample efficiency. The goal of this thesis is to incorporate imitation learning and expert feedback into AlphaZero-style learning. We extend Gumbel AlphaZero to query an expert for demonstrations during self-play, measuring uncertainty as the variance of returns backed up through search under the agent's selected action, and querying when this variance exceeds a sliding-window threshold. We propose two methods for using these demonstrations: corrective behavioral cloning, where the expert's intervention adjusts the search policy target; and action reranking, where a discriminator score nudges the search prior toward expert-like moves. We implement the system in Rust using the Burn framework, making it feasible to train on consumer hardware. We train three chess agents and evaluate them in a head-to-head tournament, where the demonstration-trained agents consistently beat a self-play baseline at matched sample counts. However, the improvement is small relative to the cost of expert queries. Analysis reveals that both methods improve the learned prior, but in high-uncertainty positions, the value estimate dominates action selection, leaving the improved prior with little influence exactly where guidance is most needed. The uncertainty measure itself proves useful: across all agents, positions with high return variance concentrate mistakes, supporting it as a query criterion for future work.

Keywords: Reinforcement learning, AlphaZero, imitation learning, active demonstrations, Monte Carlo tree search, chess, Rust.

Acknowledgements

Thank you Mattias for providing excellent supervision in this thesis work.

Sigge Rajamäe, Gothenburg, 2026-07-02

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Aim . . . 2
1.2 Contributions . . . 3

2 Related Work **5**

3 Theory and Background **9**
3.1 Problem Setting . . . 10
3.2 Game Trees . . . 10
3.3 Monte Carlo Tree Search . . . 11
3.4 Bandit-Based Action Selection . . . 12
3.5 Gumbel AlphaZero's Tree Search . . . 13
3.5.1 Learned Policy and Value Predictors . . . 13
3.5.2 Root Planning as Pure Exploration . . . 14
3.5.3 Internal Search as Policy Improvement . . . 15
3.5.4 Leaf Evaluation and Backup . . . 15
3.6 Learning from Self-Play . . . 16
3.7 Imitation Learning . . . 17
3.7.1 Behavioral Cloning . . . 17
3.7.2 Adversarial Imitation Learning . . . 17
3.8 Learning from Active Demonstrations . . . 18
3.9 Measuring Uncertainty . . . 19

4 Method **23**
4.1 Approach and Rationale . . . 23
4.2 Querying the Expert . . . 25
4.3 Corrective Behavioral Cloning . . . 26
4.4 Nudging the Prior . . . 27

5 Implementation **29**
5.1 The Training Process . . . 30
5.2 Optimizing Parallel Sample Production . . . 31

5.3 Reanalyze Demonstration Samples . . . 32
5.4 Network Architecture . . . 33
5.4.1 Policy-Value Network . . . 33
5.4.2 Discriminator Network . . . 33

6 Experiments **35**

7 Results and Discussion **39**
7.1 Overview of Runs . . . 39
7.2 Policy-Value Training . . . 40
7.3 Discriminator Training . . . 41
7.4 Query Behavior . . . 44
7.5 Analysis of Trained Agents . . . 45
7.6 Performance . . . 52

8 Ethical Considerations **55**
8.1 Licensing . . . 55
8.2 Disclosure of AI Tool Usage . . . 55

9 Conclusion **57**
9.1 Limitations . . . 59
9.2 Future Work . . . 61

Bibliography **63**

List of Figures

5.1 PRODUCERs send samples to the TRAINER. The TRAINER sends updated parameters to the INFERENCERs. 29
5.2 PRODUCERs exchange inference requests and responses with one INFERENCER per model. 30
5.3 Network architectures. 34

7.1 Policy (left) and value (right) loss over the 20,000 training steps for the Self-play, CBC, and AIL runs. Solid lines are exponential moving averages with smoothing factor $\alpha = 0.01$, and raw step values are shown faintly behind each smoothed curve. 41
7.2 Discriminator training in the AIL run. Each batch contains an equal number of demonstration samples (positives, label 1) and self-play samples (negatives, label 0). Solid lines are exponential moving averages with smoothing factor $\alpha = 0.01$, and raw step values are shown faintly behind each smoothed curve. 42
7.3 Per-episode mean uncertainty over training for the CBC and AIL runs. Each line is an exponential moving average over 2000 episodes. Raw per-episode values are shown faintly behind. 44
7.4 Kernel density estimates of relative query positions in the CBC and AIL runs. The first subfigure overlays both runs across all episodes. The other two split each run's episodes over training time into Early, Mid, and Late training groups as defined in the surrounding text. Legends show the average episode length in steps for each curve. . . . 45
7.5 Kernel density estimates of per-position uncertainty values for each of the four agents. 46
7.6 Scatter plot of per-position uncertainty and centipawn loss of the played move, overlaid for all four agents. Each point is one position from the fixed-game analysis. 47
7.7 Distribution of centipawn loss per uncertainty quantile bin for each agent, with positions split into 20 bins of 5 percentile points each and ordered by uncertainty. Each box reports the centipawn loss of the agent's played move within the bin. Whiskers extend to the standard 1.5 IQR. The solid horizontal line in each box is the median; the dashed line is the mean. 48

7.8 Distribution of centipawn loss per agent in three uncertainty spans. Top 5% covers positions at or above the agent's 95th uncertainty percentile. Bot 95% covers positions below the agent's 95th uncertainty percentile. Bot 50% covers positions at or below the agent's 50th uncertainty percentile. Each agent has six boxes, reporting the centipawn loss of the played move and the centipawn loss of the prior policy's argmax move in each span. Whiskers extend to the standard 1.5 IQR. The solid horizontal line in each box is the median. The dashed line is the mean. 52

List of Tables

6.1 Hyperparameters shared across all three runs. 35

6.2 Hyperparameters shared between the two runs that incorporate expert demonstrations. 36

7.1 Wall-clock time, self-play sample count, demonstration sample count, and realized query fraction (demonstrations divided by the total number of samples produced) for each run. 40

7.2 Pooled statistics for the discriminator score $d_\phi(s, a)$, the prior logit $h_\theta(s, a)$, and the root rank delta over the AIL run. The pooled mean is the typical value of each statistic over the run, computed as the average of the per-episode means weighted by episode step count. The pooled standard deviation is the typical spread of values within a single episode, computed as the square root of the same weighted average applied to the per-episode variances. It does not include variation between episode means. The average number of legal actions at the root across all episodes is 24.8. 43

7.3 Per-position uncertainty, prior policy entropy, and search policy entropy for each of the four agents, aggregated over all positions of the 1,000 fixed-game analysis games per agent. Mean and median of each distribution are reported. 46

7.4 Centipawn loss for each of the four agents, aggregated over all positions of the 1,000 fixed-game analysis games per agent. Search columns report the centipawn loss of the move played by the agent (emitted by the search). Prior columns report the centipawn loss of the action that would have been chosen by taking the argmax of the prior policy in the same position. Mean and median of each distribution are reported. 50

7.5 Centipawn loss restricted to three uncertainty spans. Search columns report the centipawn loss of the move played by the agent (emitted by the search). Prior columns report the centipawn loss of the action that would have been chosen by taking the argmax of the prior policy in the same position. Mean and median of each distribution are reported. 51

7.6 Head-to-head tournament results between Self-play and each of CBC, AIL-D, and AIL at three matched training checkpoints (steps 19,000, 19,500, and 20,000). In each matchup both agents are loaded from their checkpoint at the same step. Each matchup consists of 200 games, with each agent playing 100 as White and 100 as Black. Cells report the listed opponent's Wins, Draws, and Losses against Self-play as a percentage of the 200 games in each matchup. 53
7.7 Lichess Rapid Elo ratings of each deployed agent. 53

1

Introduction

Board games have for a long time served as challenges for Artificial Intelligence (AI) [1, 2, 3], due to being easy to model while allowing for great strategic depth. Good strategies often require thinking over many possible outcomes, and due to the sheer number of possible continuations in a game, many games cannot be mastered through memorization alone. Games are thus natural testbeds for methods that seek to learn sequential decision making.

Systems that learn to play board games are often designed to learn from scratch, given only the dynamics of the game and no prior knowledge of how to play it well. Strong strategies must instead be learned through experience. Methods of this kind can often be described as Reinforcement Learning (RL) [4], a paradigm of machine learning where agents learn through repeated interaction with environments, receiving feedback on decisions in the form of scalar reward signals and adjusting their decision-making with respect to observed rewards. How efficiently an agent learns this way depends on how informative this feedback is. When rewards are sparse and the number of possible situations is large, each interaction reveals little, and the agent may need a great many games to improve.

Take the game of chess: there is effectively one real reward signal, and that is the final outcome of the game. The outcome tells us how well we played against our opponent. From the perspective of one player there are three possible outcomes: a win (you were better than your opponent), a draw (you were equal to your opponent), or a loss (you were worse than your opponent). Although intermediate outcomes, such as being ahead in material or controlling the center of the board, are strongly correlated with winning, they do not conclusively determine the final outcome. Anecdotally, it is not uncommon to sacrifice material to gain better control of the board, or conversely, win material at the cost of a back-rank checkmate. Thus, any auxiliary rewards based on heuristics can only provide useful guidance; they do not describe ground truth. An agent trained only on game outcomes is not limited by what any heuristic can express, but learning from so sparse a signal requires playing many games, since a single game outcome says little about how good or bad a single move was, and the expected outcome of each position shifts as the agent's strategy changes.

The lineage of AlphaZero programs [5, 6, 7, 8] has shown that board games such as Go and chess can be mastered through self-play alone, with no human data or hand-crafted heuristics. Descendants of AlphaZero remain competitive, playing at a superhuman level, such as KataGo [9] in Go and Leela Chess Zero [10] in chess.

Reaching this level requires an enormous number of self-play games. AlphaZero, for instance, required an estimated 6.3 million games before surpassing the rating of the then-reigning world champion Magnus Carlsen [7]. [1] Leela Chess Zero reaches this scale through distributed, crowd-sourced self-play, with volunteers generating games on their own hardware, amounting to billions of games over the project's lifetime [10]. Learning new or niche games of high complexity using self-play alone is therefore not easily accomplished unless there is an abundance of computational resources. Hence, developing complementary methods that can facilitate more efficient learning is an appealing direction, and the goal of this thesis.

Prior work has made learning from self-play games more efficient [11, 10, 9]. This thesis instead complements learning from self-play with Imitation Learning (IL), a form of RL whose goal is to imitate an expert rather than to maximize the environment's reward [12, 13]. The expert may be a human or another program, and its demonstrations may be supplied upfront as a fixed dataset or actively during training. Because the expert already plays well, its demonstrations can carry information where the environment's reward is sparse, accelerating learning even when a simulator is available. Our approach is to involve an expert actively, letting the agent query it for a demonstration when it is uncertain about the outcome of its preferred move, so that fewer self-play games are needed. AlphaZero-derived methods have also been applied to problems in other domains [14, 15, 16], and if the methods we develop here prove effective, they may inspire similar techniques there.

1.1 Aim

The goal of this thesis is to investigate whether, given a limited self-play sample budget, AlphaZero-style agents can learn better chess strategies more efficiently when supplemented with active expert demonstrations during training.

We extend the Gumbel AlphaZero framework to incorporate active demonstrations, and we focus on two methods: First, we test an approach that corrects the policy targets (probability distributions over moves the agent is trained to predict) for samples in training steps that have received expert demonstrations. We refer to this method as corrective behavioral cloning, explained in Section 4.3 Corrective Behavioral Cloning. Then, we explore a complementary approach using adversarial imitation learning to guide the search towards expert-like moves by nudging the prior (the agent's probability distribution over moves before search), explained in Section 4.4 Nudging the Prior. For deciding when to query the expert, we use a novel approach described in Section 4.2 Querying the Expert, and in our experiments we will test the efficacy of this method.

The program is written in the Rust programming language [17] and uses the Burn deep learning framework [18] for all neural network related work, including loss computation and optimization. To our knowledge, no AlphaZero-style learning system written in Rust is publicly available, making this a major contribution to the Rust and machine learning ecosystems. The program supports both vanilla

[1] Estimated from the Elo learning curve and training step count reported in the paper.

AlphaZero and Gumbel AlphaZero, is optimized to run on consumer-grade hardware, and is designed to be extensible to any board game that falls under the problem formulation in Section 3.1 Problem Setting.

We formalize three research questions tied to our methods:

RQ1
: Does corrective behavioral cloning (Section 4.3 Corrective Behavioral Cloning) improve sample efficiency over the Gumbel AlphaZero baseline, and if so, to what degree?

RQ2
: Does discriminator-based prior nudging (Section 4.4 Nudging the Prior) improve sample efficiency over the Gumbel AlphaZero baseline, and if so, to what degree?

RQ3
: Does the uncertainty measure (Section 4.2 Querying the Expert) identify positions where expert demonstrations are likely to be informative?

1.2 Contributions

To our knowledge, this thesis is the first to explore active demonstrations as a vehicle for improving sample efficiency in AlphaZero-style learning for board games. More broadly, we are not aware of any prior work that combines AlphaZero-style learning with imitation learning from active demonstrations, in any domain. Prior imitation learning work that extends AlphaZero-style learning, such as EfficientImitate [19], relies on fixed demonstration datasets collected in advance. The methods proposed here instead query an expert during training, so that demonstrations are collected precisely where the agent is uncertain, and no dataset is required upfront. Exploring this direction is itself a contribution: the methods we develop are first attempts, and the findings, including where they fall short and why, establish a foundation for future work.

In this direction, we have explored two novel extensions that seek to complement learning in Gumbel AlphaZero to directly guide the search (the agent's tool for planning) towards continuations that would be preferred by the expert. Our approach has been centered on improving and complementing the prior, a prediction that hints to what continuations are worth exploring. To maximize the potential informativeness of demonstration requests, we devised a simple query mechanism that estimates the variance of returns under the selected action, and uses that to trigger a request whenever this estimate is greater than usual. Using the variance of returns as a proxy for uncertainty is not our contribution [20], nor is querying an expert with respect to a sliding window of observed uncertainty measures [21, 22], but combining both into a query criterion for complementing AlphaZero-style learning with active demonstrations is (Section 4.2 Querying the Expert). How to best make the prior more informed using these collected, expert demonstrations has been the biggest challenge of this work, and we explore two independent extensions in this direction:

1. **Corrective behavioral cloning** (Section 4.3 Corrective Behavioral Cloning), which directly trains the prior prediction to prefer actions demonstrated by

the expert, without disregarding the conclusions of the search.

2. **Discriminator-based prior nudging** (Section 4.4 Nudging the Prior), which trains a separate discriminator model to distinguish expert actions from the agent's, and uses its per-action estimate of expert-likeness to nudge the search toward expert-like continuations.

The implementation is itself a contribution. To our knowledge it is the first practical AlphaZero-style system written purely in Rust that is open source. It supports vanilla AlphaZero and Gumbel AlphaZero, runs on consumer hardware, includes environments for chess, Gomoku 8x8, Connect Four, and Tic-tac-toe, and is extensible to other board games. The repository also includes a companion program with a user interface for playing against the trained agents and for providing expert demonstrations during training, available at `https://github.com/siggerajamae/itl-public`.

As a lesser contribution, using our chosen set of hyperparameters (Chapter 6 Experiments), we saw in our analysis (Section 7.5 Analysis of Trained Agents) that, across all trained agents, when the variance of returns under the action preferred by the search is high, the move preferred by the search consistently yielded higher centipawn loss than the move preferred by the prior when both moves were analyzed by Stockfish [23]. As it might be the underlying reason why our extensions fell short, we want to highlight this finding as a fundamental challenge with Gumbel AlphaZero, indicating that methods that modify the search to take such variance into consideration, such as Weichart [20], seem a promising direction.

2

Related Work

This thesis develops methods for complementing self-play in AlphaZero-style learning with active learning from demonstrations. With this unique research direction, the works most closely related to ours are those that either seek to complement autonomous learning or enable learning using active learning from demonstrations, as well as those that bridge AlphaZero-style learning with efficient imitation learning. None of these works target both AlphaZero-style learning and active learning from demonstrations, and none of them address board games.

Active Deep Q-learning with Demonstration. Most directly related to this thesis is the work of [21], who propose the Active Reinforcement Learning with Demonstration (ARLD) framework, introducing the concept of active learning, with the goal of efficiently making use of expert guidance as a complement to the normally autonomous DQN learning process. Like our method, the agent queries an expert for demonstrations while interacting with its environment, and the aim is to learn more efficiently while asking the expert as few times as possible. ARLD argues that the expert need not be queried at every state the agent visits. Well-performing policies should agree on critical states and may differ only on less important ones, so selective querying can save expert effort. To decide when to query, ARLD uses the agent's uncertainty and queries when the current uncertainty is higher than most of the recent values in a sliding window. We query the expert the same way, as described in Section 4.2 Querying the Expert. ARLD demonstrates that this approach not only learns faster than passive demonstration methods with the same number of demonstrations, but reaches performance above the level of the expert it learns from, across four tasks, while being robust to imperfect experts and requiring little tuning of the query threshold.

Same as this thesis, their work complements an autonomous learning method with active learning from demonstrations. Different from our method is how they measure uncertainty, as they estimate it from the model, which requires changing the network architecture: either an ensemble of value heads (bootstrapped DQN [24]), where uncertainty is the disagreement among the heads' action distributions, or a noisy network, where uncertainty is the predicted variance of an action's value. We instead estimate uncertainty from the variance of the returns backed up during search, which needs no change to the model (covered in Section 3.9 Measuring Uncertainty). The methods also differ in what the expert provides when queried: in ARLD the expert takes over for several consecutive steps, whereas we ask for a single action.

ARLD adds a max-margin classification loss, inherited from DQfD, that trains the action-value estimate to score the expert action above the others by a margin. This is similar to our corrective behavioral cloning (covered in Section 4.3 Corrective Behavioral Cloning), which instead corrects the search policy target so that the expert action is at least as preferred as the action the search currently prefers. Finally, ARLD extends DQN and is evaluated on single-agent, discrete-action control tasks and Atari, whereas we extend Gumbel AlphaZero and train on deterministic, perfect-information, zero-sum, two-player games.

Planning for Sample Efficient Imitation Learning. The second work closely related to ours is EfficientImitate [19], a planning-based imitation learning method that learns from a fixed dataset of expert demonstrations, in the absence of environment rewards. It extends Sampled MuZero [25], which itself extends MuZero [26] to continuous action domains. MuZero is a descendant of AlphaZero that similarly uses Monte Carlo Tree Search (MCTS) for planning, but through a learned rather than programmed world model (true environment dynamics are unknown in planning). EfficientImitate extends this framework to pure imitation learning (with the explicit goal of imitating the expert). The work's central contribution is an efficient approach to imitation learning that combines Behavioral Cloning (BC) and Adversarial Imitation Learning (AIL). A separate BC policy is trained to model the expert's action preferences and is mixed directly into the search with the prior prediction, to always promote expert-like actions in exploration. Environment rewards are replaced with the score of a Generative Adversarial Imitation Learning (GAIL) discriminator, trained to distinguish between agent and expert transitions. EfficientImitate is evaluated on the DeepMind Control Suite on continuous-action locomotion tasks, where it achieves state-of-the-art sample efficiency and reaches near-expert performance from far fewer interactions (greater sample efficiency) than prior imitation learning methods evaluated on the same tasks.

Similar to this thesis, EfficientImitate extends a descendant system of AlphaZero. Their approach is important to our work, as we explore extensions that can independently be seen as reflecting the BC and AIL components of their unified approach, respectively. Unlike our work, EfficientImitate aims for pure imitation learning from a fixed dataset of demonstrations, whereas this thesis's aim is to complement Gumbel AlphaZero with demonstrations actively queried during training to reduce the number of self-play games needed by that system to learn complex board games. The way that the discriminator enters the search is different, as we use its score as a complement to the prior rather than letting it replace or complement environment rewards (outcomes in board games), as they do. The domains targeted by EfficientImitate are locomotion, whereas we extend Gumbel AlphaZero and train in deterministic, perfect-information, zero-sum, two-player board games.

Towards Safe and Efficient Reinforcement Learning for Surgical Robots Using Real-Time Human Supervision and Demonstration. Ou and Tavakoli [27] present a deep reinforcement learning framework based on Soft Actor-Critic (SAC) for surgical robot training with a dual goal of safety and learning efficiency. In surgical robotics, random exploration during training risks catastrophic failures such as damage to soft tissue, and rewards are often sparse, making autonomous learning

slow and unsafe. To address both concerns, a human expert monitors the training process and directly overwrites the agent's actions when it judges the situation to be dangerous or unproductive. Human and agent trajectories are collected and routed to separate replay buffers, and a GAIL discriminator is trained to distinguish human from agent state-action pairs in those buffers. Its output is added as an imitation loss to the policy update, regularizing the learned policy toward expert-like behavior, while the value estimate continues to be trained on environment rewards. The method is validated on two simulated surgical tasks, manipulating an endoscopic camera to keep track of a red cube moving in a 2D plane and grasping a piece of gauze and lifting it above a certain height. On both tasks it outperforms baseline methods that incorporate human intervention but not the discriminator component, achieving faster convergence and higher final return. On the latter task, where standard SAC without human guidance fails to learn entirely, the proposed method achieves a markedly higher success rate. The method also produces zero failures after 5,000 training steps, compared to up to 30 for the baselines.

Same as this thesis, their work complements autonomous reinforcement learning with active expert demonstrations during training rather than from a fixed dataset. Both also train a GAIL discriminator alongside the environment objective, using it to regularize behavior toward expert-likeness while preserving environment rewards in the value estimate. The most important difference lies in how the expert is involved. In their framework, a human monitors training and intervenes at will, whereas this thesis automates the query decision using the variance of returns under the selected action as an uncertainty measure. Their method extends SAC and is evaluated on continuous-action surgical robotics tasks, whereas we extend Gumbel AlphaZero and train on deterministic, perfect-information, zero-sum, two-player board games.

"Give Me an Example Like This": Episodic Active Reinforcement Learning from Demonstrations. EARLY (Episodic Active Reinforcement Learning from demonstration querY) [28] is a learning framework that complements SAC with expert demonstrations to promote faster convergence, focused on making demonstrations low-burden for the expert. It targets continuous-action navigation tasks where demonstrations are naturally provided as trajectories from an initial position to a goal. To make efficient use of expert demonstrations, the agent's uncertainty is measured over trajectories as the mean absolute TD-error, and when this measure is higher than usual, a request is made to the expert for a demonstration. Expert demonstrations are stored alongside agent trajectories and sampled equally to update SAC, treating them as additional off-policy experience rather than a separate imitation signal. In their evaluation, they compared EARLY with the methods of two prior works that also complement learning with expert demonstrations (DDPG-LfD and I-ARLD), and two pure imitation learning methods (BC and GAIL), and found that EARLY outperformed them on all three navigation tasks with 50% faster convergence. A pilot user study with 18 human participants further showed that their method successfully reduced the burden on human demonstrators, requiring significantly lower mental demand and less human time than DDPG-LfD and I-ARLD.

Same as this thesis, EARLY actively collects demonstrations during training rather than from a fixed dataset, and uses an uncertainty-based adaptive sliding window

threshold to decide when to query. The key differences lie in what is queried, how uncertainty is measured, and how demonstrations are incorporated. EARLY queries for a full episodic demonstration starting from a specific starting position, whereas we query for a single action at the current state. Its uncertainty measure is the mean absolute TD-error along a policy rollout, which measures how inconsistent the action-value estimates are across a trajectory. We instead measure uncertainty per transition from the variance of returns backed up during tree search for the selected action (Section 3.9 Measuring Uncertainty). Demonstrations in EARLY enter training as additional off-policy experience under the same SAC objective, with environment rewards preserved. In our thesis, demonstrated actions are used to correct policy training targets (Section 4.3 Corrective Behavioral Cloning) or to train a discriminator that complements the prior to steer exploration towards expert-like continuations (Section 4.4 Nudging the Prior). EARLY extends SAC and is evaluated on continuous-action navigation tasks, whereas we extend Gumbel AlphaZero and train on deterministic, perfect-information, zero-sum, two-player games.

3

Theory and Background

In this chapter we present the background needed to understand the methods proposed in this thesis. We deal with the challenge of developing Artificial Intelligence (AI) that can master board games, one of the challenges studied in Reinforcement Learning (RL) [4]. A common approach to it, and the one taken here, is AlphaZero-style learning [7], in which an agent learns to play from self-play alone, by repeatedly playing games against itself. Mastering complex games this way takes an enormous number of games, and prior work [11, 10, 9] has made it more efficient. This thesis instead complements self-play with active learning from demonstrations, in which an expert that already plays the game well is queried for guidance during training, so that strong play can be reached from fewer games. Here we first establish some standard vocabulary, then present a roadmap of the rest of the chapter.

The two-player board games studied in this thesis are modeled as Markov games [29]. The **agent** is the decision maker being trained, and the **environment** is what it interacts with. At each turn the agent observes the current **state** of the environment, chooses an **action**, and receives a scalar **reward** signal that it uses to adjust its future decisions. In the chess domain a state is synonymous with a position and an action with a move, and we use these terms interchangeably throughout the thesis. In a Markov game two players take turns acting in a shared environment, so the state a player faces on its next turn is determined by both its own action and the opponent's. It generalizes the Markov Decision Process (MDP) [30], the standard single-agent model, which is the special case when there is no opponent. We use Markov games because they keep the opponent separate from the environment rather than folding it into the environment's dynamics. We formalize the specific class of games we study, deterministic, perfect-information, zero-sum, and two-player, in Section 3.1 Problem Setting.

With these special properties, a way to reason about board game strategy is through **game trees** (Section 3.2 Game Trees). A complete game tree is a structure that from some position branches to every possible terminal position of that game. By assuming optimal play from the opponent, complete game trees let us recover optimal strategy through what is known as *minimax*. For games such as chess or Go, building complete game trees will be impractical for most positions, but planning into the future through *sparse look-ahead* trees *is* a practical approach, and one taken by AlphaZero. This can be done by estimating the promise of moves by iteratively building such trees through clever estimation techniques, and a concrete approach

is Monte Carlo Tree Search (Section 3.3 Monte Carlo Tree Search). Exploring the search tree is done according to a search strategy, and choosing which move to explore can be formulated as a *multi-armed bandit* problem (Section 3.4 Bandit-Based Action Selection).

The methods proposed in this thesis extend the later Gumbel AlphaZero variant (Section 3.5 Gumbel AlphaZero's Tree Search) [8]. It uses a modified search algorithm, and while otherwise similar, needs introduction as our second method (Section 4.4 Nudging the Prior) modifies its search strategy directly. For agent improvement, Gumbel AlphaZero learns entirely from self-play, repeatedly playing games against itself (Section 3.6 Learning from Self-Play).

The proposed methods complement this self-play with Imitation Learning (IL), which is learning with the explicit goal of imitating an expert by observing the expert's demonstrated behavior (Section 3.7 Imitation Learning). The two methods we propose fall under Behavioral Cloning (BC) (Section 3.7.1 Behavioral Cloning) and Adversarial Imitation Learning (AIL) (Section 3.7.2 Adversarial Imitation Learning), respectively, two common approaches within IL. In BC, the agent is trained by supervised learning to reproduce the expert's demonstrated actions. In AIL, a *discriminator* is trained to distinguish the expert's actions from the agent's, and the agent learns to make its action preferences indistinguishable from the expert's. In both, demonstrations are collected by querying the expert during training rather than from a fixed dataset. We call these **active demonstrations** (Section 3.8 Learning from Active Demonstrations), and a measure of the agent's uncertainty decides when to query (Section 3.9 Measuring Uncertainty).

3.1 Problem Setting

Following the line of AlphaZero work [5, 6, 7, 8], the class of games considered in this work are deterministic, perfect-information, zero-sum, two-player games. Such games can be described mathematically by a set of possible states $\mathcal{S}$ (positions), a function $\mathcal{A}(s)$ mapping each state to its legal actions (moves), a deterministic transition function $f(s, a)$ producing a successor state when action a is taken in state s, and a reward function $r_i(s)$ giving the reward at state s for player i. Because the games are zero-sum, the agents have diametrically opposed goals [29], which gives the property $r_1(s) = -r_2(s)$. Rewards are only emitted at terminal states, where $r_i(s)$ returns the outcome of the game for player i: $+1$ for a win, -1 for a loss, and 0 for a draw. The reward is also zero at non-terminal states. Each agent aims to maximize its own total reward, which, by the zero-sum property, is equivalent to minimizing the reward of the opponent.

3.2 Game Trees

One theoretical solution to playing these games optimally is through minimax search over complete game trees. Game trees are graphs where the root represents the current state (position), child nodes represent possible future states, edges represent

the moves leading to them, and leaf nodes represent terminal states, meaning all possible continuations from the root state until the end of the game. Minimax defines the optimal value $v^*(s)$ of a state s, from the perspective of the player to move, recursively as

$$v^*(s) = \begin{cases} r(s) & \text{if } s \text{ is terminal} \\ \max_{a \in \mathcal{A}(s)} -v^*(f(s,a)) & \text{otherwise,} \end{cases}$$

where $r(s)$ is the terminal reward from the perspective of the player to move at state s. The negation reflects that the next state is evaluated from the opponent's perspective, by the zero-sum property. The optimal move at any state is then $a^* = \arg\max_{a \in \mathcal{A}(s)} -v^*(f(s,a))$. Intuitively, if both agents play optimally, the value $v^*(s)$ tells you exactly what the outcome of the game will be from state s, for example, whether white is guaranteed to win, lose, or draw from a given chess position. For games, or positions within games, where it is possible to exhaustively compute $v^*(s)$, it is thus trivial to recover an optimal policy by simply selecting the action that promises the best outcome. However, for most games of practical interest, the game tree is far too large to search exhaustively. Chess alone has an estimated 10^{43} legal positions [1], meaning for most positions, the exact computation of $v^*(s)$ cannot be completed in any reasonable amount of time. When exhaustive search is not possible, we need an approach that lets us select good actions without it.

3.3 Monte Carlo Tree Search

We introduce the family of pure Monte Carlo Tree Search (MCTS) algorithms as historical background for the methods used by Gumbel AlphaZero, which underlie the extensions developed in this thesis. The basic idea builds on Monte Carlo evaluation [31]: performing a rollout from a state, meaning randomly selecting actions until a terminal state is reached, gives an unbiased estimate of the expected return under random play from that state. Pure MCTS uses this idea to guide an iterative tree search [32]. The search keeps a tree of statistics from past rollouts, and uses them to select the most promising leaf to try next. A random rollout is run from that leaf, and its outcome updates the statistics along the path taken, so later iterations make better selections. There are four steps to the algorithm:

1. **Selection.** Starting from the root, recursively select child nodes until a leaf node is reached. The selection strategy balances between exploiting moves that currently look good and exploring moves that have not been visited enough to evaluate reliably. Specific selection strategies are discussed in the next section.

2. **Expansion.** If the selected leaf is terminal, no expansion is needed. Otherwise, the selected leaf is expanded by adding successor states for one or more legal actions.

3. **Simulation.** In pure MCTS, a newly expanded non-terminal state is evaluated by simulating a random rollout until the game ends. If the selected leaf was terminal, a value representing the outcome is used directly.

4. **Backup.** Using either the terminal value or the rollout return, back up this value through the tree, updating the total return and visit count of every node along the path.

A useful intuition is that after every iteration the selection strategy has slightly more information to work with. As statistics accumulate, the selection strategy increasingly concentrates visits on moves that have consistently looked strong. Finally, after many repetitions, the action corresponding to the most visited child of the root is selected as the final move, on the grounds that a good selection strategy will have concentrated visits on the strongest move.

3.4 Bandit-Based Action Selection

What makes a selection strategy good? We want a selection strategy that produces useful tree statistics. Consider two extreme cases:

- **Uniform selection.** Selecting nodes uniformly wastes visits on nodes that are clearly bad. A node that has been visited many times and always returned a loss will always continue to do so, and uniform selection keeps visiting it anyway.
- **Greedy selection.** Selecting only the node that currently looks best will neglect moves that may be good in reality but appear bad due to early estimation errors. A move that is never selected will never accumulate better statistics, and so will remain underestimated indefinitely.

A good selection strategy sits between these two extremes: it concentrates visits on moves that look strong, while still visiting less-explored moves enough to be confident they are not being underestimated.

The problem of deciding which action to select during MCTS can be formulated as a Multi-Armed Bandit (MAB) problem with drifting distributions [33]. In this setting, a *gambler* repeatedly pulls one of several arms, each with its own reward distribution unknown to the gambler.

MABs typically fall into two categories: cumulative reward maximization (equivalently, cumulative regret minimization) and pure exploration. In cumulative reward maximization, the goal is to collect as much reward as possible over a number of trials. In pure exploration, the goal is instead to identify the arm with the highest mean reward, either under a fixed budget (a capped number of arm pulls) or with a specified confidence.

Both categories involve a trade-off in which arms are selected. In cumulative reward maximization, this is usually described as the exploration–exploitation trade-off: selecting arms whose current estimates look good can give high reward, while selecting uncertain arms can improve future decisions. In pure exploration, the same trade-off appears differently. The algorithm must decide how often to select promising arms to refine their estimates, and how often to select arms whose values are still uncertain. Selecting only the current best-looking arm may miss an arm that was underestimated

early, while selecting arms too uniformly may waste budget on arms that are already unlikely to be best.

This maps naturally to the selection strategy in MCTS. The legal actions at a node can be viewed as arms, and selecting an action during search gives information about the value of the subtree below that action. This formulation is useful because MAB problems have been studied extensively, and principled solutions exist.

3.5 Gumbel AlphaZero's Tree Search

We now turn to the search algorithm used in Gumbel AlphaZero, a variant of MCTS that forms the basis for the extensions developed in this thesis. It extends pure MCTS in two main ways. First, it replaces the implicit action selection of pure MCTS with two explicit selection strategies, one used at the root and one used inside the tree, both building on the ideas introduced in Section 3.4 Bandit-Based Action Selection. Second, it introduces prior knowledge into the search through two learned predictors: a policy **prior** that suggests which actions look promising before any search has been done, and a **value estimate** that scores leaf nodes without needing the random rollouts used in pure MCTS.

Given a state and a search budget (simulation budget), meaning a capped number of times nodes can be expanded, the algorithm returns an action to play in the real game together with an **improved policy**. The improved policy–also called the **search policy**, since it is a product of the search–is a probability distribution over the legal actions at the root, giving the probability of each action being the best to play. It is constructed from the prior knowledge and the statistics accumulated during the search, and is later used as a training target. Here, *improved* refers to the prior being improved by the search, not to a policy improved by a training update.

3.5.1 Learned Policy and Value Predictors

The policy and value predictors are the parts of the agent that are learned. For a state s, the value predictor returns $v_\theta(s)$, an estimate of how good s is for the player to move. The policy predictor returns a vector of policy logits $h_\theta(s)$, whose component $h_\theta(s, a)$ is the logit for action a. Applying a softmax over the legal actions gives the prior $p_\theta(s)$. The probability assigned to action a is written $p_\theta(a \mid s)$:

$$p_\theta(a \mid s) = \frac{\exp(h_\theta(s, a))}{\sum_{b \in \mathcal{A}(s)} \exp(h_\theta(s, b))}.$$

The prior and value estimate are two separate predictions that represent different aspects of the agent's experience. The prior reflects which actions look promising, and the value estimate reflects how good the state is for the player to move. In practice, both predictors are implemented by a single policy-value model, a neural network with two output heads (the policy head and the value head). The shared model parameters are denoted by θ, and these are the parameters optimized in Section 3.6 Learning from Self-Play. At the beginning of the search, action selection

is guided mainly by the prior. As the search continues, the algorithm gathers more evidence about which actions lead to strong positions. The search refines the agent's intuition into a more well thought-out strategy by planning through the search tree.

Gumbel AlphaZero uses two different action-selection methods during search. At the root node, the search has a fixed budget and must eventually return one action to play in the real game. For this root problem, Gumbel AlphaZero uses a variant of Sequential Halving [34]. Inside the search tree, action selection is instead treated as a policy-improvement problem.

3.5.2 Root Planning as Pure Exploration

At the root node, the search problem is a pure-exploration problem. The goal is to identify the strongest action after the search budget has been used. The general procedure in Sequential Halving is to start with a fixed number of candidate actions and halve the number of considered actions in each round. The simulation budget is split evenly between the rounds, and within each round it is split evenly between the remaining candidate actions. After a round, the candidate actions are ranked according to their current value estimates, and only the better half are considered in the next round. This continues until only one candidate action remains. [1] In Gumbel AlphaZero, this same general procedure is used, but the comparison between candidate actions combines current value estimates, policy logits, and Gumbel noise.

Gumbel noise refers to Gumbel variables, one for every legal action, drawn from the Gumbel$(0, 1)$ distribution. Let s be the root state, and let g be a vector of Gumbel variables sampled from Gumbel$(0, 1)$, with one value $g(a)$ for each legal action. The important property used by Gumbel AlphaZero is that $\arg\operatorname{top}(g + h_\theta(s), m)$ returns a set of m actions sampled from $p_\theta(s)$ without replacement. These are the initial candidate actions at the root. The Gumbel variables are made part of the scoring to introduce randomness, putting the agent in unfamiliar situations and teaching it to anticipate unexpected moves. Randomness is injected in this way, as opposed to random sampling using, e.g., fitness proportionate selection, because reusing the same g throughout the entire search is part of the policy-improvement guarantee.

The search then continues for $\lceil \log_2 m \rceil$ rounds. In each round, each remaining candidate action is evaluated the same number of times. One evaluation starts by following that candidate action from the root, and then selecting actions inside the search tree until a leaf node is reached. The leaf node is then evaluated, and the value is backed up along the selected path. For each visited node, including the root node, the visit count is incremented and the backed-up value is added to the node's total return. We write $N(s, a)$ for the visit count of action a at state s, and $W(s, a)$ for its total backed-up value. These statistics define the current action-value estimate

$$\hat{q}(s, a) = \frac{W(s, a)}{N(s, a)}$$

[1] In Gumbel AlphaZero, if the simulation budget runs out, the search can return early. In this case, the most promising candidate so far is selected. In our implementation, every round is given the same number of simulations, and all candidates within a single round are selected the same number of times.

for visited actions. After each round, the better half of the candidate actions are kept according to the score

$$g(a) + h_\theta(s, a) + \sigma(\hat{q}(s, a)). \tag{3.1}$$

Here, σ is a monotonically increasing transformation of this estimate, defined in Equation (3.3). The single action remaining after Sequential Halving is the one returned by the algorithm.

3.5.3 Internal Search as Policy Improvement

At internal nodes, action selection is interpreted as regularized policy optimization. This follows Grill et al. [35], who show that the search heuristics used by AlphaZero [7] approximate the solution of a specific regularized policy optimization problem, producing an improved policy.

For an internal node with state s, Gumbel AlphaZero uses the same general idea, constructing an improved policy $p'(s)$ as

$$p'(s) = \text{softmax}(h_\theta(s) + \sigma(\hat{q}(s))), \tag{3.2}$$

where $\hat{q}(s)$ denotes the vector of action-value estimates at the node with components $\hat{q}(s, a)$, and σ is a monotonically increasing transformation. Following [8], we use their concrete instantiation

$$\sigma(\hat{q}(s, a)) = (c_{\text{visit}} + \max_b N(s, b))\, c_{\text{scale}}\, \hat{q}(s, a), \tag{3.3}$$

where $N(s, b)$ is the visit count of action b at state s. We also adopt the values $c_{\text{visit}} = 50$ and $c_{\text{scale}} = 1$ recommended in the paper. $\sigma(\hat{q}(s))$ grows as the node receives more visits, and thus the influence of $\hat{q}(s)$ grows as the search continues.

Actions are then selected deterministically to minimize the squared error between $p'(s)$ and the normalized visit counts that would result from selecting each action. This makes the empirical visit distribution approach the improved policy.

The same construction, with the same σ, is applied at the root once the search budget has been used. The action-value estimates accumulated at the root are combined with the policy logits as in Equation (3.2) to form the improved policy returned by the search, which is later used as the policy target in Section 3.6 Learning from Self-Play.

3.5.4 Leaf Evaluation and Backup

After root and internal action selection, the search reaches a leaf node with state s. If the leaf node is terminal, the game outcome is used directly, as in pure MCTS. If the leaf node is non-terminal, pure MCTS would evaluate it by simulating a random rollout until a terminal state is reached. Gumbel AlphaZero instead uses the value estimate $v_\theta(s)$, introduced in Section 3.5.1 Learned Policy and Value Predictors, to estimate how good the leaf state is for the player to move under the agent's current strategy.

This is the same role value functions play when a search stops before reaching a terminal state. If the full game tree could be searched, the value of a state would be given by $v^*(s)$. Since the search stops at non-terminal leaf states, Gumbel AlphaZero instead uses the learned value estimate $v_\theta(s)$. This value is then backed up through the tree and contributes to the action-value estimates used by the search. Since the games considered here are zero-sum, the value changes sign when moving between players.

3.6 Learning from Self-Play

As hinted at in Section 3.5.1 Learned Policy and Value Predictors, learning in this framework means improving the predictors p_θ and v_θ. The predictions are given by a neural network, so improving them means updating the weights of the network to minimize a joint loss, given in Equation (3.4), through gradient descent. Gumbel AlphaZero follows the same general training loop as the AlphaZero line of work [5, 6, 7, 8]. Games of self-play are produced by running the tree search described in Section 3.5 Gumbel AlphaZero's Tree Search with the current model parameters. Each step of a played game is then turned into a training sample by associating the step with the outcome of its game.

A training sample contains three main parts: the game state s, the final game outcome z from the perspective of the player to move at s, and the improved policy $p'(s)$ produced by search at that state. The outcome z is represented as 1 for a win, -1 for a loss, and 0 for a draw. These samples are pushed to a replay buffer. The replay buffer is an ordered store of recent samples, typically with a fixed maximum size. It stores samples from a history of games, so training does not depend only on the most recent games. When the buffer is full, new samples replace old samples in first-in, first-out order.

Training the predictors is done jointly. After a number of games or samples have been produced, a training step is scheduled. In this training step, a mini-batch is drawn from the replay buffer. For sample i in the mini-batch, let s_i be the state, z_i the final outcome, and p'_i the improved policy produced by search. For each s_i, the model predicts a prior $p_\theta(s_i)$ and a value estimate $v_\theta(s_i)$. The network is trained to minimize a joint loss between these predictions and the training targets. Following the AlphaGo Zero formulation [6], we write this as

$$\mathcal{L}(\theta) = \text{mean}_i \left[(z_i - v_\theta(s_i))^2 - p_i'^\top \log p_\theta(s_i) \right] + c\|\theta\|^2, \tag{3.4}$$

where $c\|\theta\|^2$ is the L_2 regularization term.

The loss is backpropagated through the model, giving gradients for the shared parameters θ. These gradients are used to update the policy-value model. Future self-play games then use the updated model.

3.7 Imitation Learning

Part of the goal of this thesis is to improve the sample efficiency of Gumbel AlphaZero–which learns from self-play alone, with no expert data–by involving an expert actively in the learning process, so the agent learns both through self-play and by imitating the expert, who already plays the game well. The methods proposed to this end are forms of Imitation Learning (IL), and this section gives the theoretical background needed to motivate them.

IL is a paradigm in RL, where expert demonstrations are used to guide the learning process. The expert may be a human expert or another program. Expert demonstrations can be given either as trajectories, meaning full episodes or segments, or as individual state-action pairs. In this thesis, the basic unit of demonstration is a single state-action pair: given a state, the expert outputs an action. The two methods proposed in this thesis fall under behavioral cloning and adversarial imitation learning, respectively, which are two common approaches within IL.

3.7.1 Behavioral Cloning

Behavioral Cloning (BC) is one of the simplest forms of imitation learning. In its most basic form, BC is supervised learning. The learner is given a state and is trained to predict what an expert would do in that state.

The expert may provide a full policy, or the policy target may be constructed from a smaller demonstration signal. For example, if the expert only provides one action, that action can be used to construct a target for the learner. In discrete action spaces, this can be viewed as classification over actions. The method proposed in this thesis follows this idea, but constructs the target policy from both the expert action and the MCTS search policy. The details are given in Section 4.3 Corrective Behavioral Cloning.

BC is simple because it does not require a reward signal or a separate RL objective. It turns demonstrated behavior directly into policy targets. This is also its limitation. BC tells the agent what the expert did in the demonstrated state, but not why the action was good.

BC struggles with covariate shift because the policy is trained on states visited by the expert, but the learner may visit different states when it acts on its own [36]. In those states, the policy may be uncertain unless it has learned to generalize beyond the expert data. This depends on the model and training setup, but the demonstrations themselves do not directly support behavior in states the expert did not visit.

3.7.2 Adversarial Imitation Learning

A different approach to imitation is to model what distinguishes expert behavior from non-expert behavior, and to use this signal to guide the agent. This is the principle behind Adversarial Imitation Learning (AIL), which is rooted in Inverse Reinforcement Learning (IRL). The goal of IRL is to recover the reward function

that explains the expert's behavior [37], which can then complement [38] or replace the environment reward in an RL process. However, IRL recovers a reward function as an intermediate step, even though the goal is to obtain a policy, which makes it inherently expensive [39].

The canonical AIL method, GAIL [39], sidesteps the reward recovery step. GAIL trains a discriminator $D(s, a)$ as a binary classifier between expert and learner state-action samples. The demonstration samples are assumed to come from rollouts of the expert policy π_E, and the learner samples from rollouts of the current policy π. The discriminator outputs the probability that (s, a) came from the expert.

In the GAIL form, the discriminator is

$$D_\phi(s, a) = \sigma(d_\phi(s, a)), \tag{3.5}$$

where $d_\phi(s, a)$ is a learned score parameterized by ϕ. It is trained with binary cross entropy to classify demonstration samples against learner samples.

As shown by Ho and Ermon [39], training the agent against this discriminator is equivalent to matching the learner's occupancy measure to the expert's. In standard GAIL, the discriminator output is used to derive a reward signal in an RL update.

We adopt the GAIL discriminator form in this thesis, but our setup differs from standard GAIL in two ways. First, expert demonstrations are collected actively at states the agent reaches when uncertain (Section 4.2 Querying the Expert), so the positive samples come from the learner's state distribution rather than from rollouts of the expert policy. Second, we do not use the discriminator output as a replacement for the environment reward. Instead, in the method introduced in Section 4.4 Nudging the Prior, the score $d_\phi(s, a)$ is added to the learned prior inside search.

The score $d_\phi(s, a)$ has a useful meaning. If $d_\phi(s, a) = 0$, the discriminator is indifferent about whether the action came from the expert or the learner. A positive value indicates that the action looks more expert-like, and a negative value indicates that it looks more learner-like.

3.8 Learning from Active Demonstrations

Using expert demonstrations to guide an autonomous learner requires a source of demonstrations. A fixed expert-demonstration dataset only contains explicit expert behavior for the states it covers. How useful that guidance is depends on how similar the learner's visited states are to the expert's visited states in the dataset. The more similar they are, the more directly the demonstrations support the learner's decisions.

Supervised machine learning assumes that the distribution of states seen during training matches the distribution of states seen during evaluation. In IL, a model trained to recognize expert behavior in expert-visited states assumes that the learner will visit those same states when it acts. In large state-action spaces, this assumption

can easily fail [36]. Small deviations from expert play can lead to states that the demonstration dataset does not cover, and a model trained on expert-like states may give poor guidance there.

Learning from active demonstrations addresses this by collecting demonstrations during training instead of relying on a fixed dataset. The expert is queried at states the learner reaches, so demonstrations are collected on the learner's own state distribution. For BC, this directly addresses covariate shift because the policy receives targets in the states where it must act. For AIL, the discriminator must distinguish expert from learner actions at states the learner visits, which is easier when expert demonstrations come from those same states. Because the expert is queried on demand at a specific state the agent has reached, the natural unit of demonstration is a state-action pair.

Querying an expert for demonstrations bears a cost. When demonstrations are provided by a human expert, attention is a limited resource and cannot be spent on every state the learner visits. Given a limited query budget, the goal is to query when the expected information gain is high. In this thesis, uncertainty is used as a proxy for where a demonstration is likely to be useful. The next section introduces the uncertainty measure used for this purpose.

3.9 Measuring Uncertainty

Deciding when to query the expert for demonstrations is a question we would like to answer. A simple baseline is to query randomly, for example with some small probability. Random querying is not ideal because some positions, such as mate-in-one, have obvious best actions where a demonstration would waste expert attention. Wasteful queries are not restricted to any particular phase of the game. Ideally, we would want some measure to tell us that querying the expert in a given state will make the agent at least some amount smarter, but such a measure is non-trivial to come up with. Instead, we look for a measure of uncertainty that can serve as a proxy for when demonstrations are likely to be useful, on the assumption that guidance will be more informative in states where the agent's behavior is uncertain. The question is then what kind of uncertainty is useful for this purpose.

We first consider uncertainty measures derived directly from the value and policy estimates available after search. A neutral (near-zero) value estimate conveys a limited kind of uncertainty: it says that either player is about as likely to win. This does not mean that the position is a useful place to query the expert. A position can be easy and neutral at the same time. For example, at the start of a game, no player is typically favored to win. An agent can still learn strong opening moves and be confident in those moves, even though the value estimate remains neutral. Strong opening moves are an important part of good chess strategy, but specific opening positions recur much more often across games, since after only a few moves the game has had fewer ways to branch. Value estimates therefore cannot, on their own, serve as good uncertainty measures for deciding when to query the expert.

The entropy of the policy is also not a good measure. Consider two extreme

examples. First, in the opening of the game, many moves may be near equal in expected outcome. If a high-entropy search policy is treated as uncertainty, then early, low-stakes scenarios would receive many expert queries. Second, consider an endgame scenario where all legal moves lead to checkmate. Every move is then tied for best move, and the policy can have maximal entropy even though the actions are genuinely equally good. A high-entropy policy is therefore not a measure of useful uncertainty.

Previous work has proposed query criteria based on the disagreement between ensemble value and action-value predictors [21, 22]. One way to think about this is that if the predictor that tells the agent what will happen next is indecisive, then this is likely an unfamiliar situation that could benefit from expert guidance. Measuring disagreement can, for instance, be done through the variance of value estimates. The value function estimates how good a state is under the current policy. If this estimate has high variance, then the agent does not have a stable estimate of the outcome from that point onwards.

AlphaZero could be extended with multiple value heads, or with several independent models, so that disagreement at a node could be interpreted as epistemic uncertainty. But search gives another, planning-based notion of value variance. During search, each simulation backs up a value return along its path. These returns are outcome estimates at leaf nodes reached by search. For an action, the variance of these returns says something about how volatile the continuation under the search policy is. The search will sometimes explore bad continuations, and it will naturally find continuations with different estimated outcomes. The signal is therefore not that the returns differ at all, but that they vary more than usual through a node. If this happens, it can indicate two things: either the position is genuinely high-stakes, so different continuations lead to very different outcomes, or the current policy and value estimates are mismatched, so search often visits continuations where potential is lost. In the first case, expert guidance acts as a safety precaution. In the second case, it provides a useful learning signal.

Variance-aware MCTS maintains this estimate with a running second-moment statistic [20]. The update can be implemented with Welford's online variance algorithm [40]. This adds a running variance estimate to the usual visit-count and mean-value statistics stored in each search node.

One possible query score is the final variance estimate at the root node. This would measure how volatile the search from the current state appears overall. In Gumbel AlphaZero, this could be misinterpreted. Root planning is a pure-exploration procedure: within each round, all remaining candidate actions are evaluated the same number of times. The root variance can therefore be high in a high-stakes position even when the agent knows how to navigate it. In that case, the agent knows that the situation is dangerous and also knows which action to play, so querying the expert might be wasteful.

A better uncertainty measure might be the variance of returns below the agent's final preferred action. Unlike root planning, internal search matches the empirical visit distribution to the improved policy, where the improved policy allocates probability

mass toward the most promising actions. The action-value estimates at internal nodes therefore say more about what will happen if the agent plays as search currently intends. If this policy-improvement process gives a stable continuation, then the returns below the preferred action should be relatively consistent. High variance below the preferred action is therefore a stronger signal: it says that the agent is unsure about the outcome of playing what it currently believes is the best move. In this sense, selected-action variance acts as a guardrail for unsafe behavior, and should focus queries on what will happen next if the expert does not intervene. Focusing the uncertainty signal on the single action preferred by the agent also aligns with the noisy-network variant of [21], where uncertainty is the predictive variance of the selected action's Q-value rather than a node-level statistic.

4 Method

This chapter introduces the methods proposed in this thesis. In contrast to the theory chapter, which covered the established background that motivates the approach, the methods described here are those that we evaluate in Chapter 6 Experiments. Section 4.1 Approach and Rationale discusses the rationale of our method, why active learning from demonstration interests us as a seemingly practical approach, and the alternatives we considered. Section 4.2 Querying the Expert then describes the prerequisite of how we measure the agent's uncertainty, and how that measure gives a criterion for when to query the expert. With expert demonstrations, we incorporate them into the learning process through two extensions: Section 4.3 Corrective Behavioral Cloning describes the method we call corrective behavioral cloning, which directly trains the prior to prefer moves demonstrated by the expert, and Section 4.4 Nudging the Prior describes discriminator-based prior nudging, which uses a discriminator score to complement the prior in search, guiding it towards expert-like continuations.

4.1 Approach and Rationale

The extensions we explore should incorporate expert guidance into the otherwise autonomous learning process of Gumbel AlphaZero. We develop them to work well in the chosen problem setting (Section 3.1 Problem Setting) and to allow for human guidance. Board games are tasks performed by humans, and for such tasks we look for approaches that do not demand much more of the expert than its expertise in the task itself. A program might be able to systematically score decision making, but this is not natural for humans. For an expert, saying which move is best might be easy, while ranking moves against each other, or saying it will win with some likelihood, might not be, even for someone who strategizes well. The point is that if an expert knows how to perform a task that is easily demonstrable, then it should be able to answer what the best move is. When no game-playing program is available for some game, our method should work with human demonstrators. For these reasons, using demonstrations as the currency that helps agents learn from fewer games of self-play makes sense.

At minimum, this style of learning requires an implementation of the environment, in the sense formalized in Section 3.1 Problem Setting, an expert, and a means for that expert to provide demonstrations. The expert can be a strong program, or, where

none exists, a human who provides demonstrations through a graphical interface to the environment. For an accommodating program, then, supporting a new game just requires implementing the game's dynamics and such an interface. Where enough recorded games already exist, as for Go, a policy for use in search can instead be trained directly by supervised classification on that data, as AlphaGo did with 30 million recorded positions [5]. The alternative of learning from demonstrations that are not active, when no prerecorded data exists, as might be the case for a new or niche board game, is more demanding. The human would have to play games against themselves, or find others to play with, and record those games, and a portion of the recorded positions will carry little information if they already agree with the agent's strategy.

Programmed heuristics can also guide learning, and many strong chess programs have historically relied on handcrafted heuristics for all decision making, with Stockfish [23] as an example. But programming heuristics by hand can be difficult or unproductive, and the expert of a game is not necessarily a programmer. In an RL setting, heuristics can enter as auxiliary reward signals through potential-based reward shaping, which provably leaves the optimal policy unchanged [41]. The second method we propose (Section 4.4 Nudging the Prior) can be thought of as recovering something akin to heuristic guidance, since its discriminator does not care about the actual outcomes of games, only about expert-likeness. Through this lens, we imagine that this style of learning could offer what programmed heuristics would, but without the need for programming skills.

Another approach to consider is preference modeling, where the expert, instead of providing demonstrations, is queried with two clips or segments of agent trajectories to give their preferences on which is more expert-like [42]. Although this is a path worth considering, it seems unnatural for the domain of board games that we target. Especially if we target segments of high uncertainty, we expect these segments to often contain both good and bad moves, and then the expert must decide which segment features better play in general. If the expert is presented with two segments that both feature suboptimal play, then the expert's attention is wasted on preferring one of them, when they could have just demonstrated ideal play. Further, branching games to allow preferences, without playing them until the end, means we lose the outcomes that we normally get from all self-played games, meaning the value estimate can become more expensive to train relative to the increased amount of inference such an approach could require.

Part of our approach is also how we evaluate our extensions. How capable a chess-playing agent is can be most honestly measured by its win rate against a variety of other programs, since this reflects its general game-playing strength. Other work has analyzed how closely an agent's play follows established human openings [7], examined the agent's thinking on the network level and consulted a human grandmaster's assessment of its games, to see whether its decision making is motivated by concepts often central to human strategy in chess, such as material balance or king safety [43], or measured how well it predicts the specific moves and mistakes of human players at a given skill level [44]. But none of these evaluation methods directly tell us how strong the agents trained with our extensions are compared to a baseline

agent trained with only self-play, or against other agents, which is what we need to answer our research questions. Foreshadowing our results, our extensions fell short of meaningful performance gains, and such alternative methods of evaluation could help explain why, similar to how we use centipawn loss in Section 7.5 Analysis of Trained Agents.

4.2 Querying the Expert

After performing a search, the agent must decide whether to query the expert. If the expert is queried, then it returns an action a_E. The agent plays a_E and saves it for later, when training samples are constructed.

The measure of uncertainty used is the variance of returns discussed in Section 3.9 Measuring Uncertainty. The search computes this variance by keeping one additional running statistic for each action-value estimate. In addition to $N(s,a)$, $W(s,a)$, and $\hat{q}(s,a)$, it stores $M_2(s,a)$, the accumulated squared deviation of backed-up returns for action a at state s. Let s_{leaf} be the leaf reached by one simulation, and let $v_s(s_{\text{leaf}})$ denote the value estimate from the perspective of the player to act in s. This is the scalar backed up through (s,a). The statistic is updated using Welford's online variance update [40, 20]:

$$
\begin{aligned}
\hat{q}_{\text{old}}(s,a) &\leftarrow \hat{q}(s,a), \\
N(s,a) &\leftarrow N(s,a) + 1, \\
W(s,a) &\leftarrow W(s,a) + v_s(s_{\text{leaf}}), \\
\hat{q}(s,a) &\leftarrow \frac{W(s,a)}{N(s,a)}, \\
M_2(s,a) &\leftarrow M_2(s,a) + \left(v_s(s_{\text{leaf}}) - \hat{q}_{\text{old}}(s,a)\right)\left(v_s(s_{\text{leaf}}) - \hat{q}(s,a)\right).
\end{aligned}
$$

The empirical return variance for the action is then

$$
\operatorname{Var}(s,a) = \begin{cases} \dfrac{M_2(s,a)}{N(s,a)}, & N(s,a) > 1, \\ 0, & N(s,a) \leq 1. \end{cases}
$$

At the end of search, let a_A be the action selected by the agent. The uncertainty value u is $\operatorname{Var}(s, a_A)$.

To decide whether u is greater than usual, we use the adaptive query strategy proposed by [21]: the agent compares u with a sliding window of previously observed uncertainty values. Let $t \in [0,1]$ be the target fraction of searched positions that should receive demonstrations, and let w be the sliding window. Before inserting u into the window, the agent queries the expert if

$$
u > \text{percentile}_{1-t}(w).
$$

After the query decision, u is inserted into w. If the window is full, the oldest value is removed.

The hyperparameter t controls the expected query frequency through the percentile threshold. For example, $t = 0.05$ uses the 95th percentile of the recent window, so the agent expects to query roughly five percent of searched positions. We verify the realized query frequency empirically. A smaller window adapts more quickly when the agent's average uncertainty changes, reducing long bursts of over-querying or under-querying after a shift, but gives a noisier percentile estimate. A larger window gives a more stable threshold, but adapts more slowly when the uncertainty level changes.

If the threshold is exceeded, then the expert is queried. At this point the agent halts until it receives the demonstration. Thereafter, the agent takes the expert action and saves it for later, when training samples are constructed. After an episode has ended, two types of samples are created: self-play samples and demonstration samples. They only differ in that demonstration samples include a demonstration action.

4.3 Corrective Behavioral Cloning

The first proposed demonstration-learning extension is a form of *corrective* BC. We motivate it by a simple expectation: if the agent queries an expert in state s, and the expert returns action a_E, then future training should make the agent more likely to choose a_E in that state. A demonstration specifies behavior: in state s, the expert chose action a_E. Since the policy head is the part of the model that learns action preferences, a natural first attempt is to train it directly to prefer the demonstrated action.

One way to train the prior to prefer the demonstrated action would be to replace the search policy target with a one-hot target on the expert action. The search policy, however, is a probability distribution over moves, whereas a single demonstration provides only one move. For example, a target policy can agree with the expert and be made less informative by one-hot encoding at the same time. Consider a complete search tree that has explored every possible continuation and concluded that all moves are equally good, together with an expert that agrees with this conclusion. One-hot encoding the expert's move would still produce a worse target than the search policy. Instead, the demonstration is treated as a correction to the search policy: the expert action should be at least as preferred as the action currently preferred by search, but the rest of the search policy should otherwise be preserved.

Let $p'(s)$ be the improved policy produced by search for state s, and let a_E be the expert action. Define the action most preferred by search as

$$a_{\max} = \arg\max_{a \in \mathcal{A}(s)} p'(a \mid s),$$

If $a_E = a_{\max}$, the search policy is already consistent with the demonstration and is left unchanged:

$$p^{\star}(s) = p'(s).$$

Otherwise, a corrected target $p^{\star}(s)$ is constructed by assigning the expert action the same probability mass as the search-preferred action and rescaling the remaining

actions proportionally:

$$p^\star(a \mid s) = \begin{cases} \dfrac{p'(a_{\max} \mid s)}{1 + p'(a_{\max} \mid s) - p'(a_E \mid s)}, & \text{if } a = a_E, \\ \dfrac{p'(a \mid s)}{1 + p'(a_{\max} \mid s) - p'(a_E \mid s)}, & \text{if } a \neq a_E. \end{cases}$$

This is the smallest correction that makes the expert action tied with the action preferred by search while preserving the relative probabilities among all non-expert actions.

Demonstration samples are stored in a separate replay buffer. Each demonstration sample contains the same information as a regular self-play sample, together with the expert action a_E. During training, a fixed fraction of each mini-batch is drawn from this demonstration buffer. For these samples, a fresh search is run to produce the current search policy $p'(s)$, which is then replaced by the corrected target $p^\star(s)$. This avoids training on stale search policies while still reusing expensive expert demonstrations across multiple training updates. The value target is omitted for a similar reason: because demonstration samples are reused for far longer than self-play samples, the stored value target can also become stale, in the sense that it no longer represents the expected outcome under the agent's current policy.

Here we also want to highlight a broader limitation with both methods explored in this work: while demonstrations can say a lot about an optimal agent's behavior, they are limited in their ability to express the long-term potential of a state. We cannot blindly treat the states we end up in after taking the expert action as good, since we might have queried the expert from a losing position. The fact that an action was favored by the expert mainly tells us about a change in potential. If we imagine the expert's value estimate V_E, then we would not expect $-V_E(f(s, a_E))$ (negated to account for the change in perspective) to be much lower than $V_E(s)$; likewise, the agent's own value estimate after taking the expert action should not be lower than before. One way to use this signal would be a hinge loss that penalizes the network when $-V(f(s, a_E)) < V(s)$, aligning the agent's value changes with those of the imagined expert value estimate. We do not pursue this here. We instead see the guidance purely as a means of steering tree search toward promising positions.

4.4 Nudging the Prior

The second proposed extension uses the discriminator in Equation (3.5) to guide search more efficiently. Gumbel AlphaZero uses the raw policy logits in both the root score Equation (3.1) and the internal policy-improvement score Equation (3.2). The idea is to make these logits more informative through AIL by adding or subtracting energy depending on how expert-like an action appears.

For a healthy discriminator, we expect the score $d_\phi(s, a)$ to be positive for actions that look expert-like and negative for actions that do not. The score therefore promotes or discourages an action when added to the policy logit, effectively reranking actions

at each node. We can view this signal as a delayed prior, or as a reward; but if as a reward, only on that single edge: it complements the environment reward, without entering the value estimate. We investigate the behavior of this signal more closely in Chapter 6 Experiments.

We scale this effect by c_{disc}, which is set small enough that the discriminator can influence search without dominating the existing policy logits or value-based search term. If the child node reached by taking action a in state s has been expanded, then a discriminator score has been computed for (s, a), and we define the augmented policy logit as

$$h^\star(s, a) = h_\theta(s, a) + c_{\text{disc}} d_\phi(s, a). \tag{4.1}$$

If the child node has not been expanded, no discriminator score is available and $h^\star(s, a) = h_\theta(s, a)$. With this augmented logit, the root score Equation (3.1) becomes

$$g(a) + h^\star(s, a) + \sigma(\hat{q}(s, a)).$$

The internal policy-improvement score Equation (3.2) becomes

$$p'(s) = \text{softmax}(h^\star(s) + \sigma(\hat{q}(s))).$$

As the node receives more visits, the discriminator signal remains part of the prior logit, but its relative influence decreases with the rest of the prior as the action-value estimates become more important.

We do not compute $d_\phi(s, a)$ for all edges, only the ones leading to expanded nodes. Since the discriminator only sees transitions that were either preferred by the agent or the expert, it might provide uninformative guidance for transitions that would normally never be considered by the agent. We treat $d_\phi(s, a)$ as a second opinion, which can either encourage or discourage future visits to a node that was considered by the agent. Thereby, despite only being computed for a subset of edges, the discriminator can still affect the relative preferences in the search. If the correction discourages visits to one node, other nodes become better in comparison. This concentrates inference on edges where we expect the discriminator to be informative. If every expansion creates b new leaf nodes on average, then evaluating the discriminator only for edges leading to expanded nodes reduces the number of discriminator calls by roughly a factor of b.

Training the discriminator follows the same structure as Section 3.6 Learning from Self-Play. After a number of samples have been produced, a training step is scheduled. A mini-batch is drawn containing an equal number of self-play and demonstration samples. For sample i in the mini-batch, let s_i be the state, a_i the action taken, and $y_i \in \{0, 1\}$ the label, where $y_i = 1$ for demonstration samples and $y_i = 0$ for self-play samples. The discriminator parameters ϕ are trained by minimizing the binary cross-entropy loss

$$\mathcal{L}(\phi) = -\,\text{mean}_i \left[y_i \log D_\phi(s_i, a_i) + (1 - y_i) \log \left(1 - D_\phi(s_i, a_i)\right) \right], \tag{4.2}$$

where D_ϕ is as defined in Equation (3.5). The loss is backpropagated through the model, giving gradients for ϕ. Future searches then use the updated discriminator.

5 Implementation

This chapter presents the program implemented as part of this thesis. To feasibly conduct experiments on limited hardware, we need to make full use of the available hardware. AlphaZero-style learning requires a large number of training samples, generated by simulating many games. Each game consists of a sequence of steps, and at every step a tree search is performed. The tree search performs hundreds of node expansions, each requiring a prediction from the neural network. Batching these predictions is the most efficient way to use the GPU, which requires maintaining a constant stream of states (and actions) to evaluate, meaning asynchronous CPU-bound work. Rust was selected as a well-suited language for these purposes.

For comprehension, we introduce the three main units of the program. These are abstractions representing distinct responsibilities in the training process, and their relations are as follows. The PRODUCERs generate games of self-play and send training samples to the TRAINER. The TRAINER receives these samples, performs training updates, and sends the updated parameters to the INFERENCERs, as seen in Figure 5.1. The INFERENCER performs neural network inference, serving batched requests from the PRODUCERs, as seen in Figure 5.2.

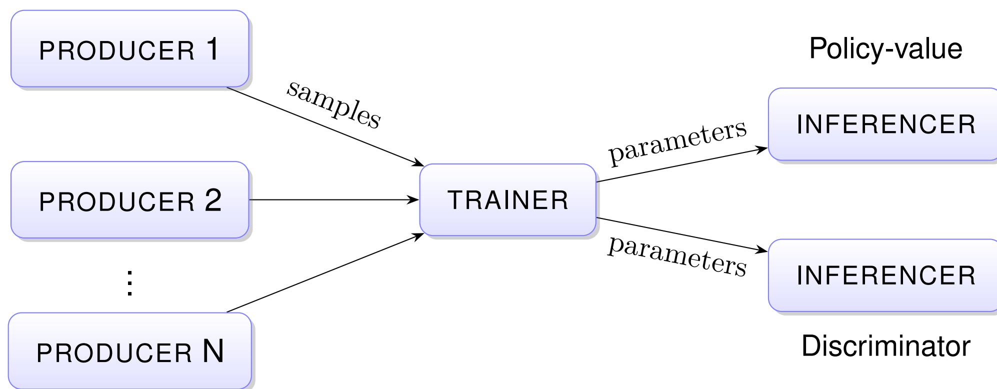


Figure 5.1: PRODUCERs send samples to the TRAINER. The TRAINER sends updated parameters to the INFERENCERs.

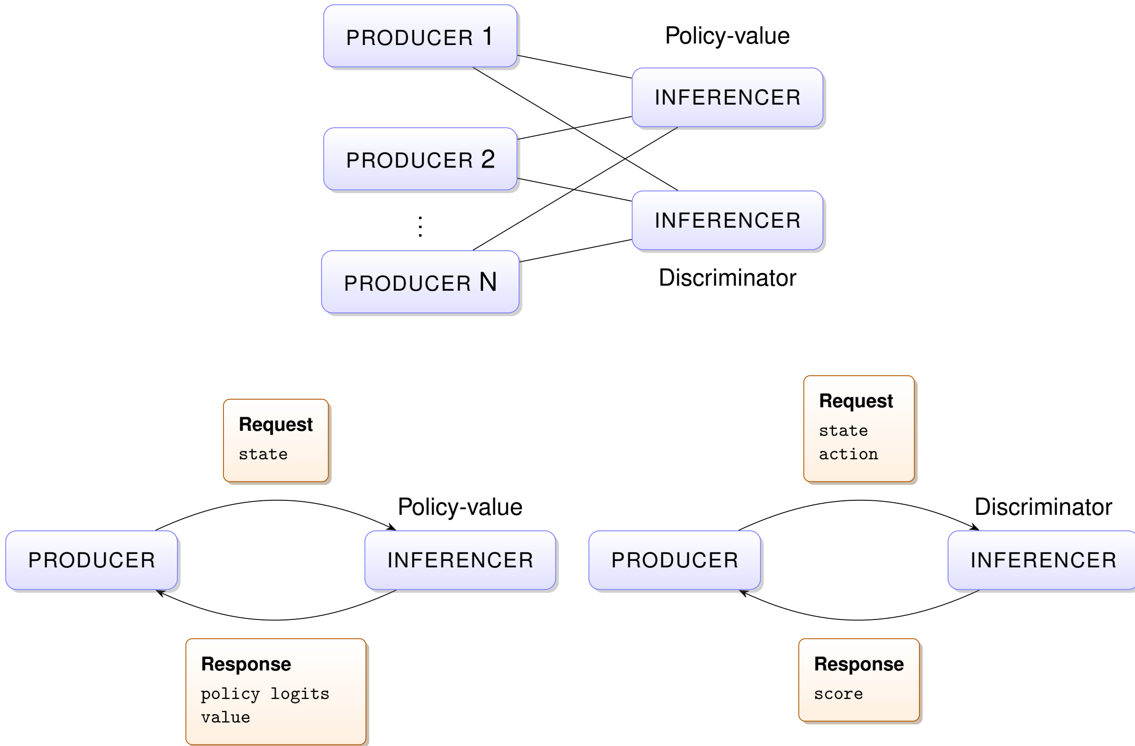


Figure 5.2: PRODUCERs exchange inference requests and responses with one INFERENCER per model.

5.1 The Training Process

Sample production happens in the PRODUCERs, and is done by repeatedly playing games from start to finish using the current version of the agent. One sample corresponds to one transition of the game, including the game's state, the action taken, and the game's final outcome from the perspective of the player to act. At the beginning of each episode, the current state s is initialized as the game's starting position. Until s is terminal, a loop runs as follows: The agent runs a tree search, returning the selected action a_A, the search policy $p'(s)$, and an uncertainty value u, as described in Section 3.5 Gumbel AlphaZero's Tree Search and Section 4.2 Querying the Expert. Each PRODUCER maintains a sliding window w of size n over recent uncertainty values, and $t \in [0, 1]$ is the target query fraction. If u exceeds $\text{percentile}_{1-t}(w)$, the expert is queried and returns an expert action a_E, which is played. Otherwise the agent plays a_A. u is pushed into w and s is updated to $f(s, a)$. After the episode ends, samples are constructed from the transitions, including the expert action a_E where available, and sent to the TRAINER.

The TRAINER receives samples from the PRODUCERs and routes them into two replay buffers: one storing self-play samples, and one storing demonstration samples (self-play samples with expert actions). The models being trained have their own *views* of each replay buffer, each with their own copies of received samples, capacity and sampling state. For example, demonstration samples used to train the discriminator model are drawn uniformly from its replay view, whereas demonstration samples

included in the policy-value training step are drawn with respect to the number of times a sample has already been used (as part of policy-value training steps). Keeping separate replay views makes this simple.

When enough new samples have been received since the last training step, a training step is triggered. The policy-value model and the discriminator can be trained on different schedules, though in our experiments both use the same schedule. Training steps run sequentially within the TRAINER. The training steps for each model are described below.

Training the policy-value model follows Section 3.6 Learning from Self-Play, except that demonstration samples are mixed into each mini-batch alongside self-play samples. A fixed number of self-play samples and a fixed number of demonstration samples are drawn from their respective replay views. The number of demonstration samples is configured with respect to the query frequency. For the experiments described in Chapter 6 Experiments, self-play samples are drawn using uniform sampling and demonstration samples using inverse usage count weighting to promote recent and underused samples. For demonstration samples, the policy target is constructed as described in Section 4.3 Corrective Behavioral Cloning, and the loss is computed as usual.

The discriminator is trained on an equal number of self-play and demonstration samples, both drawn uniformly from their respective replay views. One notable asymmetry is that demonstration samples are transitions where the agent had high uncertainty, while self-play samples are drawn from anywhere in the game. This violates the standard discriminator assumption that positive and negative samples share the same state distribution. In our experiments this appears to have only an insignificant effect, and is discussed further in Chapter 7 Results and Discussion.

After either model has been optimized as part of a training step, the updated model parameters are sent to the corresponding INFERENCER, and all future inference requests will use those parameters.

5.2 Optimizing Parallel Sample Production

Here we outline the steps taken to efficiently generate samples asynchronously. Training samples are produced by running games of self-play. To maximize the throughput of generated samples, we run several PRODUCERs concurrently. The ultimate bottleneck of simulating games is doing neural network inference, meaning forwarding states through the neural networks that output the policy, value, and discriminator predictions used in the search. By running many PRODUCERs simultaneously, we can synchronize via inference requests to a central INFERENCER, combining requests into batches that can be forwarded efficiently through the networks. What is special for our implementation is that PRODUCERs never synchronize with the TRAINER, meaning, model parameters can be updated at any point during an episode, even in the middle of an ongoing tree search. Since training is off-policy and tree search is slightly stochastic, we do not expect occasionally drifting model parameters to have a substantial impact on the quality of training data. We found that in practice,

this does not seem to prevent or destabilize learning. With that said, drifting might become more destructive if we increase the number of PRODUCERs beyond what is needed to maintain high GPU throughput. Thus, we set the number of PRODUCERs with a small margin to produce a constant stream of requests so that the INFERENCER never times out. This decision simplifies the design greatly, making the program easier to optimize. Some alternatives would be to either halt sample production when there is an ongoing training step, or have a history of model parameters and always let games or tree searches finalize with the parameters they started with.

Since PRODUCERs will constantly block to await inference responses, there will be frequent rescheduling. We therefore opt to run PRODUCERs as tasks on the Tokio asynchronous runtime [45]. This lets us scale the number of PRODUCERs well beyond the number of available cores with negligible context switching overhead.

Apart from increasing the number of PRODUCERs, we can keep the INFERENCERS busy by preparing multiple expansions within a single tree search before blocking to await responses. Since all root candidates are selected an equal number of times within each round, as described in Section 3.5 Gumbel AlphaZero's Tree Search, we expand leaves from all remaining candidates simultaneously, so each PRODUCER can send roughly $\frac{m\lceil\log_2 m\rceil}{m-1}$ expansion requests before blocking, where m is the initial number of sampled candidates. When there are only a few candidates left, we also use a multi-select strategy inspired by Leela Chess [10]: selections are made round-robin across candidates, and a virtual visit count is applied along each selected path to discourage future selections from reaching the same node without preventing them completely. Leela Chess uses multiple threads, one per selection, but we found this introduces unnecessary complexity since selections are cheap and opt for sequential selects before requesting inference. If a collision occurs and the same node is selected twice, multi-select breaks early and the pending requests are sent.

5.3 Reanalyze Demonstration Samples

In this work, expert demonstrations are treated as a scarce resource, and our methods should work where only a small fraction of training samples have received demonstrations. For demonstrations to be impactful on training, we seek to reuse them across many training steps. In Section 4.3 Corrective Behavioral Cloning, the search policy $p'(s)$ is corrected so that the expert action a_E is at least as preferred as the action most preferred by search, producing the corrected target $p^\star(s)$. This target depends on the agent's current policy.

Training on a search policy stored at collection time may therefore produce a weaker or misleading signal as the model improves. Since demonstration samples make up only a small fraction of each training batch, the cost of rerunning search for each reused sample is negligible, and we do so for all demonstration samples [11]. We do not reanalyze self-play samples, as producing a fresh self-play sample is as cheap as reanalyzing a stored one. Reanalysis searches run as tasks on the Tokio runtime [45], the same runtime as the PRODUCERs, so they naturally participate in the same inference batching infrastructure without any additional coordination.

5.4 Network Architecture

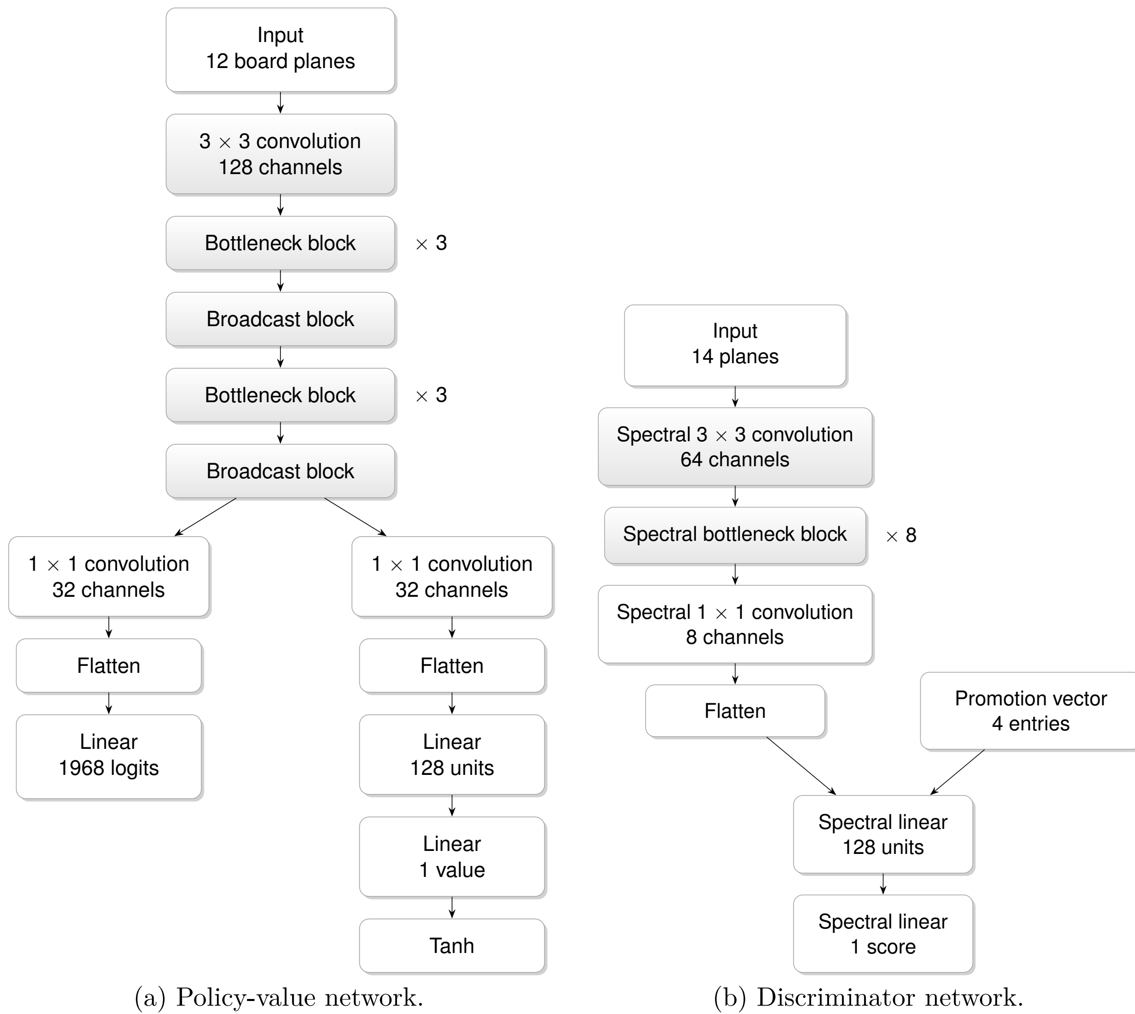
The chess agent uses two neural networks: the policy-value network used during search and training as described in Section 3.5 Gumbel AlphaZero's Tree Search and Section 3.6 Learning from Self-Play, and the discriminator network used by the nudging-the-prior method (Section 4.4 Nudging the Prior). Both are convolutional residual networks inspired by the network architecture used in Gumbel AlphaZero [8]. We use the same overall architecture, scaled down and with a simpler input plane representation, for efficiently conducting experiments where end performance is not as important as seeing the relative effects of the proposed methods.

5.4.1 Policy-Value Network

The policy-value network uses the Gumbel AlphaZero architecture [8], shown in Figure 5.3, scaled down to 128 hidden channels, 64 bottleneck channels, six bottleneck residual blocks, and two broadcast residual blocks. Gumbel AlphaZero uses 256 hidden channels, 128 bottleneck channels, and a broadcast residual block every eighth layer, while our smaller trunk places a broadcast block every fourth layer. The bottleneck and broadcast residual blocks were implemented in Burn [18] according to the source code given by Danihelka et al. [8]. The network input is a 12-plane representation of the current board state, with one plane per piece type per player, seen from the side to move. State information such as castling rights, en passant, repetitions, and the no-progress counter is not provided to the network, but remains available to the search tree through which actions are legal and which actions lead to terminal outcomes. The policy head produces 1968 logits for a flat action space. Each action is identified by the square moved from and the square moved to. Promotion moves are encoded separately for each piece the pawn can promote to.

5.4.2 Discriminator Network

The discriminator predicts whether a (state, action) pair came from the expert or from the agent (Section 4.4 Nudging the Prior), shown in Figure 5.3. Its input is the same 12 board planes used by the policy-value network, plus two planes encoding the move taken by the square moved from and the square moved to, and a four-element promotion vector that is concatenated with the flattened spatial features in the head. The network uses the same bottleneck-block structure as the policy-value network but is smaller, with 64 hidden channels, 32 bottleneck channels, and eight bottleneck residual blocks without broadcast blocks. The head reduces the spatial features to 8 channels, concatenates the promotion vector, applies a 128-unit linear layer, and outputs one score. All convolutional and linear layers use spectral normalization as proposed by Miyato et al. [46], and the activations are leaky ReLUs.



(a) Policy-value network.

(b) Discriminator network.

Figure 5.3: Network architectures.

6
Experiments

This chapter describes our training runs and the evaluations used to compare the resulting agents. The three runs let us compare a pure self-play baseline against two demonstration-based methods: corrective behavioral cloning (Section 4.3 Corrective Behavioral Cloning) and discriminator-based prior nudging (Section 4.4 Nudging the Prior). Training-time data collected alongside each run, together with per-agent analyses, helps explain the differences in agent strength.

All three runs share the same self-play and training setup, with the numeric hyperparameters summarized in Table 6.1. Note that the simulations per move listed here are a cap on the number of simulations, and the actual number of simulations can vary with the number of legal actions due to rounding that ensures all root candidates receive the same number of simulations. Each model is trained for 20,000 training steps, with one step taken every 1024 produced self-play samples, for a total budget of 20,000 $\times$ 1024 self-play samples per run. The policy-value network is optimized with AdamW. When sampling from the replay buffer for a training step, we prioritize samples by weighting them by the inverse of their use count (with add-one smoothing to handle unused samples). When new samples enter a full replay buffer, we evict the most-used samples first, with ties broken by age (older first). This is to promote recent, underused samples, and to ensure that older, stale samples are evicted first.

The two runs that incorporate expert demonstrations share the parameters summarized in Table 6.2. They use the adaptive query strategy described in Section 4.2 Querying the Expert, targeting approximately 5% of produced positions (self-play plus demonstrations) for expert demonstration; this is an exaggerated amount of expert supervision for the setting, chosen to make the effects of each method easier to observe. Demonstrations are provided by Stockfish [23]. We do not have a verified

Parameter	Value
Simulations per move	256
Sampled actions (root)	32
Learning rate	1e-3
Weight decay	1e-4
Batch size	2048
Self-play replay buffer size	100,000

Table 6.1: Hyperparameters shared across all three runs.

Parameter	Value
Target query fraction t	0.05
Query window size	500
Stockfish node limit	100,000
Demonstration replay buffer size	100,000

Table 6.2: Hyperparameters shared between the two runs that incorporate expert demonstrations.

Elo rating for Stockfish at this search limit, but for our purposes it is more than strong enough, and far stronger than the agents trained in this work. Demonstration samples are stored in a separate replay buffer, with the same inverse use-count sampling and most-used-first eviction as the self-play replay buffer.

The three runs differ as follows, and we refer to them as Self-play, CBC, and AIL respectively throughout the remainder of this thesis:

Self-Play
The baseline. This is the standard Gumbel AlphaZero-style training method, with no expert demonstrations. Each training batch is 2048 self-play samples drawn from the self-play replay buffer.

Corrective Behavioral Cloning
Each policy training batch consists of 1792 self-play samples and 256 demonstration samples drawn from the demonstration replay buffer, keeping the total batch size at 2048. Demonstration samples are reanalyzed at training time, and their policy targets are constructed as described in Section 4.3 Corrective Behavioral Cloning.

Nudging the Prior
The discriminator weight in the policy logits is $c_{\text{disc}} = 0.2$, and a discriminator network is trained alongside the policy-value network as described in Section 4.4 Nudging the Prior. Each policy training batch is the standard 2048 self-play samples; demonstration samples enter training only through the discriminator.

All three runs were trained on machines with RTX 5090 and RTX PRO 6000 graphics cards, rented through Vast.ai [47], a website where independent hardware owners list their machines for rent. Performance varied between machines.

The trained agent from the AIL run is evaluated in two modes: with the discriminator active during search, which we refer to as AIL-D, and without, which we refer to as AIL. Counting these as separate agents, our evaluations cover four agents in total: Self-play, CBC, AIL-D, and AIL.

For evaluation play we use a simulation budget of 2048 with 64 initial candidates. To compare the trained agents, each is deployed on Lichess [48], an online chess platform, through a bot account that accepts and sends match challenges to other similarly rated bots at a time control of 10+0 (10 minutes per side with no increment, classified as rapid on Lichess). The Lichess Elo of each agent is calibrated over 238 to 248 games against other bots. We also run local head-to-head tournaments between

Self-play and each of CBC, AIL-D, and AIL, at three matched training checkpoints (steps 19,000, 19,500, and 20,000). In each matchup both agents are loaded from their checkpoint at the same step, and the matchup consists of 200 games, with each agent playing 100 as White and 100 as Black. Together, these give a rough comparison of the agents' relative strengths, and of their strengths relative to other bots on Lichess.

We also conduct a fixed-game analysis to study each agent's per-position behavior. Each agent plays 1,000 self-play games with the evaluation settings, and for each position we record the agent's prior policy, search policy, uncertainty, and chosen action. Each position is then evaluated by Stockfish at depth 9. We use a fixed depth rather than a node limit to give consistent estimates regardless of the branching at each position. A node limit acts as more of a computational ceiling, where the depth reached can vary by position. Here we treat Stockfish as the expert again, which selects the best move a_E. For each position we score three moves with Stockfish. These are a_E, the action a_A taken by the agent (emitted by the search), and the prior argmax $a_p = \arg\max_a p_\theta(a \mid s)$. Stockfish returns either a centipawn score or a mate-in-N score. A centipawn is one hundredth of a pawn, so 100 centipawns corresponds to losing one pawn's worth of value. We map mate-in-N scores to $+1000$ centipawns for forced wins and -1000 for forced losses, and we clamp raw centipawn scores to $[-1000, 1000]$. Let $v_{\mathrm{SF}}(a)$ denote the Stockfish score of action a at the position. The centipawn loss of a is $\mathrm{clamp}(v_{\mathrm{SF}}(a_E) - v_{\mathrm{SF}}(a), 0, 1000)$. The lower clamp ensures that CPL is never negative, since Stockfish at a fixed depth does not always find the absolute best move and a can occasionally score higher than a_E. The search CPL is the CPL of a_A. The prior CPL is the CPL of a_p.

The following Results chapter analyzes the data collected during these experiments. We compare wall-clock times and sample counts in Section 7.1 Overview of Runs, the policy-value loss curves in Section 7.2 Policy-Value Training, the discriminator training and its in-search behavior in Section 7.3 Discriminator Training, and the adaptive querying behavior in Section 7.4 Query Behavior. The fixed-game analysis is covered in Section 7.5 Analysis of Trained Agents, where we look at per-position uncertainty, prior and search policy entropy, and centipawn losses of the played move and the prior argmax across uncertainty bands. Strength comparisons are reported in Section 7.6 Performance.

7

Results and Discussion

This chapter presents the results of our experiments and discusses their implications for the research questions posed in Section 1.1 Aim. The data presented and discussed comes from four sources: data collected during the three training runs, Lichess Elo ratings obtained by deploying each final agent as a bot, local head-to-head tournaments between the final agents, and Stockfish-annotated analysis of each final agent's play over a fixed number of games. We combine these sources as needed in each subsection. The strength comparisons across agents are presented at the end of this chapter in Section 7.6 Performance.

7.1 Overview of Runs

Table 7.1 summarizes each run's wall-clock time, the self-play and demonstration sample counts, and the realized query fraction. The realized query fraction is the demonstration count divided by the total number of samples produced over the run (self-play samples plus demonstration samples), and can be compared against the target query fraction $t = 0.05$ from Section 4.2 Querying the Expert.

The first thing we see here is the drastic difference in training time between Self-play and the other runs. Inference is inherently more difficult to optimize when many PRODUCERs are also waiting for Stockfish demonstrations, which is bound by CPU throughput. The AIL run trained the additional discriminator model, needing more compute and additional synchronization between INFERENCERs, further slowing down that run.

To maintain high GPU throughput, one solution is to simply scale up the number of PRODUCERs, but this has the additional side effect of increasing the frequency of training steps during episodes. Whether this had a positive effect, negative effect, or no effect at all is not something we analyzed specifically, but the agents do not seem to learn less efficiently as a result of more producers. For the Self-play, CBC, and AIL runs we set the number of PRODUCERs to 120, 180, and 240, respectively. In other words, training updates during episodes were about twice as common in the AIL run as in the Self-play run. We selected these as safe numbers of PRODUCERs for each run, which should not obfuscate the results shown here.

Another thing that stands out is how the realized query fraction exceeds the target query fraction, and this is simply explained by how the uncertainty grew for each of

Run	Wall-clock time	Self-play samples	Demonstration samples	Realized query fraction
Self-play	11h 25m	20,480,000	n/a	n/a
CBC	19h 34m	20,480,000	1,206,000	0.056
AIL	21h 53m	20,480,000	1,154,129	0.053

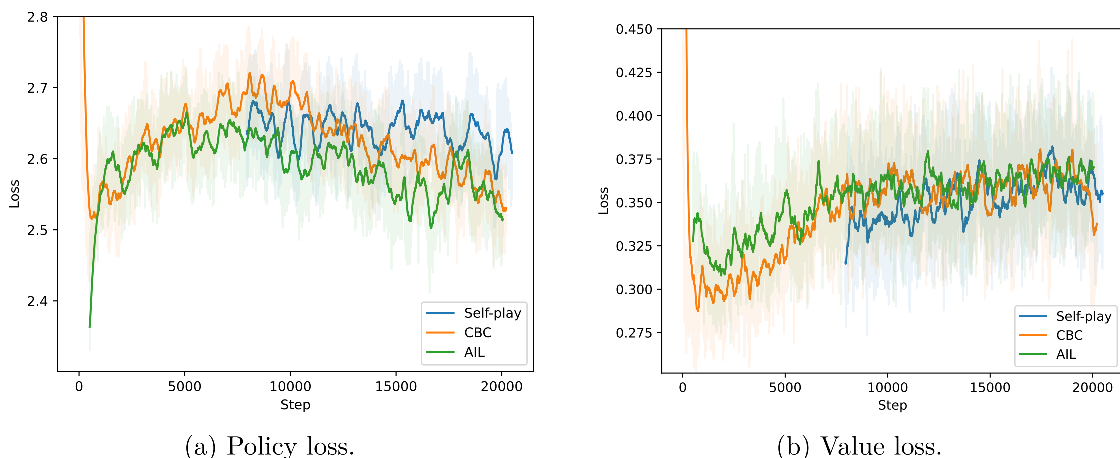
Table 7.1: Wall-clock time, self-play sample count, demonstration sample count, and realized query fraction (demonstrations divided by the total number of samples produced) for each run.

the runs with demonstrations, seen in Figure 7.3. Since the realized query fraction depends on the trend of uncertainties, this is expected. Had the mean uncertainty converged to roughly stable or decreasing, then we would expect to see a realized query fraction closer to our target.

7.2 Policy-Value Training

Figure 7.1 shows the policy and value loss over the 20,000 training steps for all three runs, as logged at each training step of the policy-value network. Due to restarts of the training program at step 7,936 in the Self-play run and step 502 in the AIL run, the loss data for those runs is truncated to start at those respective steps. The CBC run was not restarted.

First, the loss is fairly noisy, as shown by the faint, raw loss in both plots. This is expected from AlphaZero-style learning, as the value is constantly adapting to how the agent's strategy evolves, and the prior is itself adapting to the expected outcome, represented by the value estimate. It is difficult to make any conclusive statements about what is shown here. It seems that the Self-play run had slightly more policy loss towards the end, suggesting that the prior and search policies may be in closer agreement for the runs that use demonstrations. For the AIL agent, this could be expected if the discriminator score tends to dominate the search, producing lower-variance policy targets that are easier to predict accurately. For the CBC agent, this seems like a natural consequence of demonstration samples being reused across many training updates, giving the model more exposure to them. For the value loss, there does not appear to be any clear separation between the agents.



(a) Policy loss.

(b) Value loss.

Figure 7.1: Policy (left) and value (right) loss over the 20,000 training steps for the Self-play, CBC, and AIL runs. Solid lines are exponential moving averages with smoothing factor $\alpha = 0.01$, and raw step values are shown faintly behind each smoothed curve.

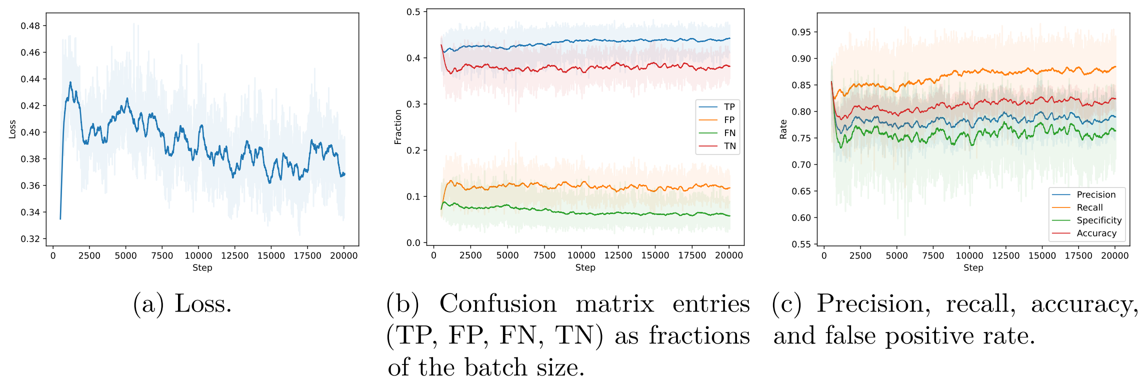
7.3 Discriminator Training

Only the AIL run trains a discriminator network. Figure 7.2 shows the binary cross-entropy training loss, the per-batch confusion matrix entries as fractions of the batch size, and the per-batch precision, recall, accuracy, and false positive rate, as logged at each of the 20,000 discriminator training steps.

Here we can make some observations about the discriminator's health. First, the training loss is noisy, similar to what we saw for the policy-value models. This is again expected since the discriminator is constantly adapting to a moving target, the agent's action preferences. Figure 7.2b shows what the discriminator is generally good at: it is good at identifying expert moves, with few false identifications, and we do not see any downward trend in its ability to identify agent moves. Over the entire training run, the trends are quite flat, with the number of true positives increasing, as the number of false negatives decreases. This is reflected by Figure 7.2c, where recall, TP/(TP + FN), increases over training, so the discriminator catches more expert moves as the training goes on. This might be explained by demonstration samples being reused many more times than self-play samples.

In addition to the per-training-step data, the AIL run also logs per-episode statistics that aggregate statistics computed inside the Gumbel AlphaZero search trees (Section 3.5 Gumbel AlphaZero's Tree Search) built for each move of each episode. Every move is decided by one such search. For each tree we record the discriminator score $d_\phi(s, a)$ appearing in the augmented logit Equation (4.1) at every node expansion, where s is the parent state and a is the action leading to the expanded node, and the change in action ordering at the root induced by adding the discriminator's contribution to the policy logits as described in Section 4.4 Nudging the Prior.

Per-tree quantities are first aggregated within an episode by taking a flat mean,



(a) Loss.

(b) Confusion matrix entries (TP, FP, FN, TN) as fractions of the batch size.

(c) Precision, recall, accuracy, and false positive rate.

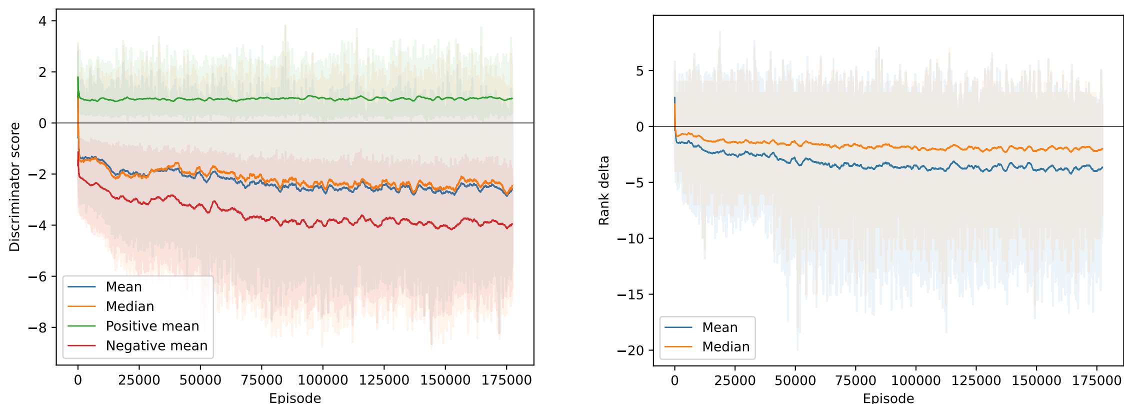
Figure 7.2: Discriminator training in the AIL run. Each batch contains an equal number of demonstration samples (positives, label 1) and self-play samples (negatives, label 0). Solid lines are exponential moving averages with smoothing factor $\alpha = 0.01$, and raw step values are shown faintly behind each smoothed curve.

median, or standard deviation over the combined values from every move's tree, producing one statistic per episode. The figures and table that follow then aggregate these per-episode statistics across episodes weighted by episode step count, so that longer episodes contribute proportionally more. Aggregating over episodes was done to keep storage manageable. In hindsight, these statistics would likely have been more representative had they been per-search and per-simulation instead.

Figure 7.3a reports per-episode statistics of the raw discriminator scores. The four statistics are the overall mean, the median, the mean of only the positive scores, and the mean of only the negative scores. Figure 7.3b shows the per-episode root rank delta. For each root action in a search, the rank delta is the change in that action's rank between the ordering by prior logits $h_\theta(s, a)$ alone and the ordering by the augmented logits $h^\star(s, a)$ from Equation (4.1), so the discriminator's scale factor c_{disc} is included in the comparison. A positive value means the discriminator's contribution promotes the action and a negative value means it demotes it.

We include these statistics mainly to show the general effect of the discriminator score, which is that it punishes exploration of non-expert-like actions. Many of the actions explored in the search tree are not expert-like and therefore receive a negative score, corresponding to a demotion in that action's rank. It is important to note that the distribution of actions seen in the search tree is unlikely to closely match the distribution of actions taken, as many actions are evaluated in the tree for the sake of exploration, and are thus likely to be non-expert-like, corresponding to the negative mean that we see.

Table 7.2 reports pooled statistics across all episodes of the AIL run. The discriminator score and root rank delta correspond to the quantities plotted in Figure 7.3a and Figure 7.3b. The prior logit $h_\theta(s, a)$ at the same expansions is included as context. The tabulated discriminator score is the raw $d_\phi(s, a)$, so the corresponding contribution to the augmented logit is c_{disc} times this value. The value $c_{\text{disc}} = 0.2$ was set before the run by briefly checking discriminator scores against prior logits and



(a) Per-episode discriminator score statistics over the AIL run. The figure shows the overall mean and median of the per-episode scores, the mean of only the positive scores, and the mean of only the negative scores, with the negative mean plotted below zero. Each line is an episode-step-weighted rolling mean over 2000 episodes. Raw per-episode values are shown faintly behind.

(b) Per-episode root rank delta over the AIL run, the change in a root action's rank between the ordering by prior logits and the ordering by the augmented logits in Equation (4.1). The figure shows the mean and median over all root actions across all moves in each episode, smoothed by an episode-step-weighted rolling mean over 2000 episodes. Raw per-episode values are shown faintly behind.

picking a scale at which the discriminator could meaningfully, but not destructively, shift the ordering of root actions. The pooled statistics here are consistent with that choice: given an average of 24.8 legal moves at the root, the discriminator demotes a move by 3.25 ranks on average with a standard deviation of 6.42.

Statistic	Pooled mean	Pooled std
Discriminator score	-2.3064	2.7431
Prior logit	-1.3046	0.8999
Root rank delta	-3.2526	6.4168

Table 7.2: Pooled statistics for the discriminator score $d_\phi(s, a)$, the prior logit $h_\theta(s, a)$, and the root rank delta over the AIL run. The pooled mean is the typical value of each statistic over the run, computed as the average of the per-episode means weighted by episode step count. The pooled standard deviation is the typical spread of values within a single episode, computed as the square root of the same weighted average applied to the per-episode variances. It does not include variation between episode means. The average number of legal actions at the root across all episodes is 24.8.

7.4 Query Behavior

This subsection covers when the adaptive query strategy (Section 4.2 Querying the Expert) queried the demonstrator during the CBC and AIL runs. For each episode played during training we logged the indices of queried steps and the per-move uncertainty values.

Figure 7.3 plots the per-episode mean uncertainty over training. The per-move uncertainty is the variance estimate from Section 3.9 Measuring Uncertainty. The per-episode value averages over the moves of the episode. Here we see that the uncertainty grows quickly near the start, when the model transitions from its uniformly initialized weights, and then remains fairly consistent throughout training.

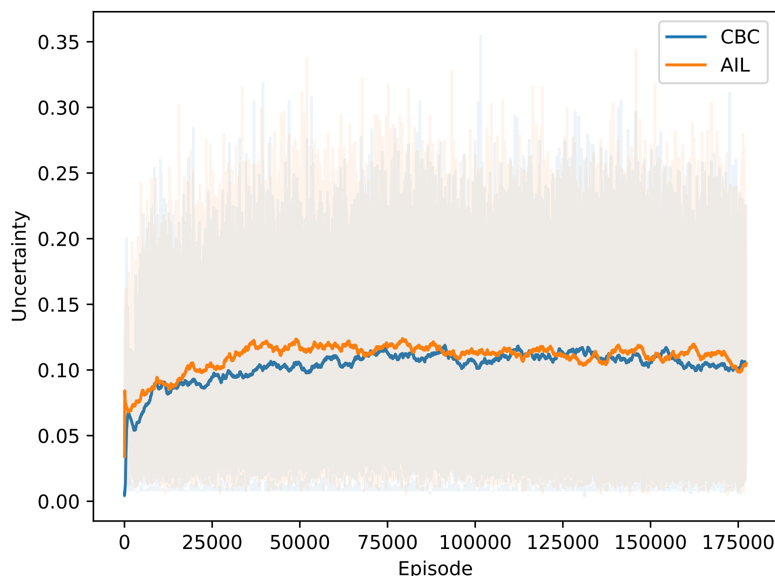
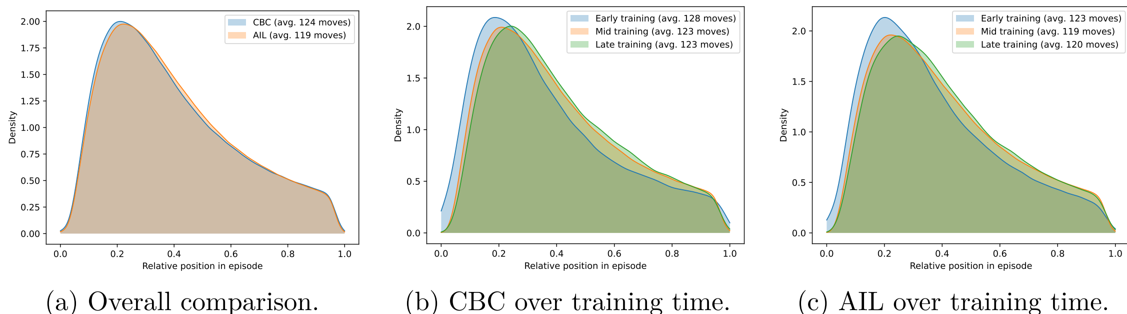


Figure 7.3: Per-episode mean uncertainty over training for the CBC and AIL runs. Each line is an exponential moving average over 2000 episodes. Raw per-episode values are shown faintly behind.

Figure 7.4 shows the distribution of relative query positions in the CBC and AIL runs. The relative position of a queried step is the step at which the query occurred divided by the total number of steps in the episode, so it ranges from zero at the opening of the game to one at the endgame. The first subfigure overlays the kernel density estimates for CBC and AIL across all episodes. The other two split each run's episodes over training time into three groups and plot one density estimate per group. The Early training group is the first 10% of episodes. The Mid training group is the middle 80%. The Late training group is the last 10%. Together they show how the within-episode query distribution changed over the course of training. What is noteworthy here is that the query distribution does appear to drift slightly forward as training goes on, but the difference between the Late training and Mid training distributions is smaller, suggesting that the agent quickly becomes more certain in the opening but learns the middlegame more slowly. The shape of the tail indicates that the agent is more certain about the outcomes of games in the endgame

than in the middlegame, possibly because games are volatile in the middlegame and already appear decided by the endgame. In the overall comparison we see that the query distribution is roughly the same for both agents trained with demonstrations, meaning the methods do not seem to differ in where return variance is relatively high.



(a) Overall comparison. (b) CBC over training time. (c) AIL over training time.

Figure 7.4: Kernel density estimates of relative query positions in the CBC and AIL runs. The first subfigure overlays both runs across all episodes. The other two split each run's episodes over training time into Early, Mid, and Late training groups as defined in the surrounding text. Legends show the average episode length in steps for each curve.

7.5 Analysis of Trained Agents

This subsection analyzes the behavior of each trained agent using data from the fixed-game analysis described in Chapter 6 Experiments. Recall that each agent played 1,000 self-play games with the evaluation settings, and that for each position we recorded the prior policy, search policy, uncertainty, and chosen action, and scored a_E, a_A, and a_p with Stockfish to obtain the per-position centipawn losses.

Figure 7.5 shows the distribution of per-position uncertainty values for each agent. Each curve is a kernel density estimate fitted to the uncertainty values from all positions of all games played by that agent. Here we see that the uncertainty distribution of the Self-play agent is slightly flatter, showing a longer tail with more spread-out uncertainty. The agents trained with demonstrations appear to have more uncertainty concentrated around zero, meaning that they more often show near-zero uncertainty.

Table 7.3 summarizes per-position uncertainty and the entropy of the prior and search policies for each agent, aggregated over all positions from the fixed-game analysis. The Self-play agent shows higher uncertainty and higher prior entropy but lower search entropy than the agents trained with demonstrations. The higher prior entropy may reflect the absence of expert input during training. The Self-play prior trains only on improved policies from search. CBC additionally trains on a corrected target for demonstration samples, which raises the probability assigned to the expert action. AIL adds the discriminator score to the prior logits during search, which biases the improved policy toward expert-like actions before the prior trains

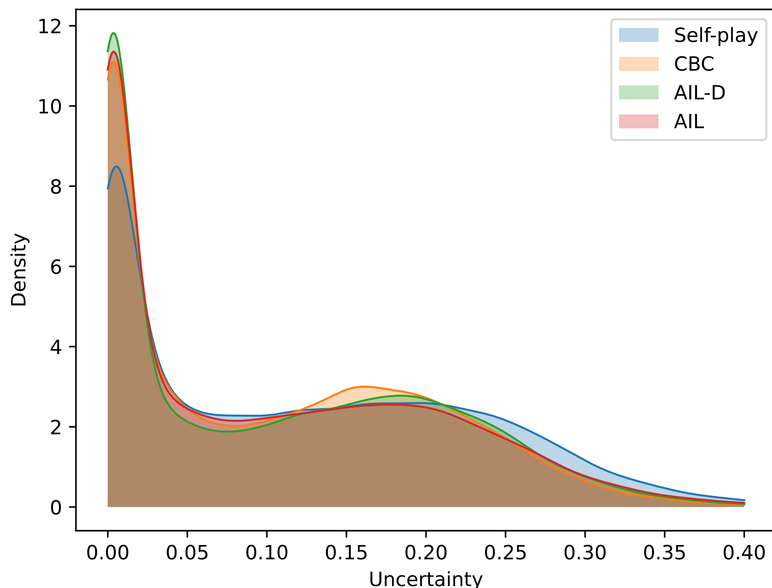


Figure 7.5: Kernel density estimates of per-position uncertainty values for each of the four agents.

Agent	Uncertainty mean	Uncertainty median	Prior entropy mean	Prior entropy median	Search entropy mean	Search entropy median
Self-play	0.122	0.107	2.790	3.098	0.256	9.5e-07
CBC	0.103	0.085	2.596	2.923	0.334	1.7e-05
AIL-D	0.104	0.082	2.667	3.039	0.348	9.35e-05
AIL	0.104	0.078	2.683	3.049	0.363	6.93e-05

Table 7.3: Per-position uncertainty, prior policy entropy, and search policy entropy for each of the four agents, aggregated over all positions of the 1,000 fixed-game analysis games per agent. Mean and median of each distribution are reported.

against it. Either mechanism could reduce the prior entropy when averaged over positions. The higher uncertainty is a higher variance of returns at the played action, which may indicate that the Self-play value function gives less informed estimates. If the demonstrations strongly influence the agent's play, the agent's trajectories become more regularized, since the expert's contributions constrain the agent to a narrower set of positions. With less variation in the trajectories the value head sees during training, the value function can become more confident, which would lower the variance of returns at the played action. The lower search entropy may also be related to the same variance. Higher variance below a root action means returns under that action span a wider range, and these returns propagate up to the root as the search accumulates visits. With high variance, $\hat{q}$ at the root may settle at larger magnitudes across actions. Since $\sigma(\hat{q})$ scales linearly with $\hat{q}$, larger $\hat{q}$ values let $\sigma(\hat{q})$ contribute more to the improved policy softmax, producing a sharper distribution.

Figure 7.6 shows per-position uncertainty plotted against centipawn loss of the played move, overlaid for all four agents. There appears to be a correlation between uncertainty and CPL. We see high density near the bottom-left corner, meaning that many low-uncertainty positions also have low CPL. As we move toward higher-uncertainty positions on the right, the CPL spreads out and shifts upward. This

suggests that our uncertainty measure tracks high-stakes positions that often result in bad outcomes.

Figure 7.7 illustrates this further by splitting each agent's positions into 20 uncertainty quantile bins of 5 percentile points each and showing the distribution of centipawn loss within each bin. Here the relationship is clearer: if we want to most reliably identify high-stakes positions, then targeting positions with high variance of returns under the selected action is a valid approach, and this is consistent across all trained agents.

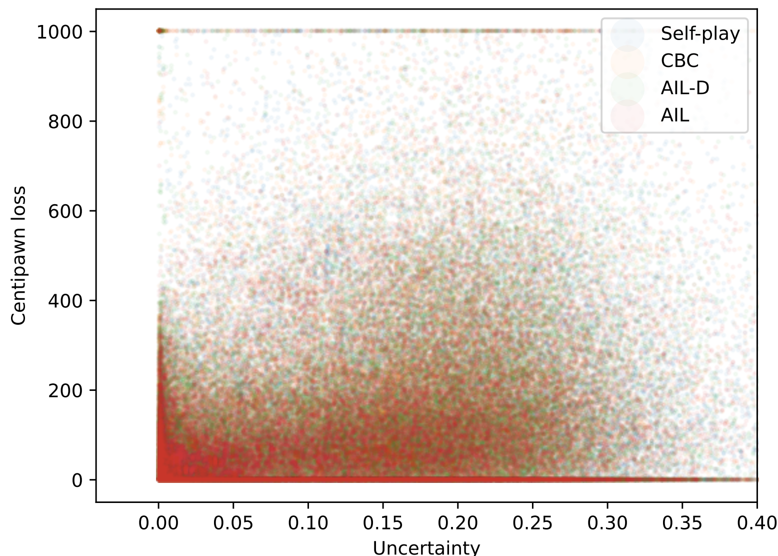
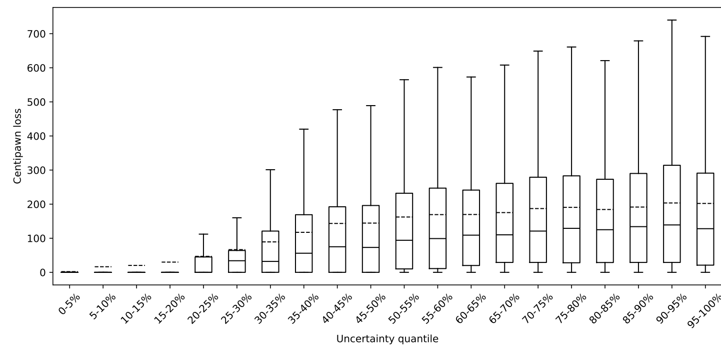
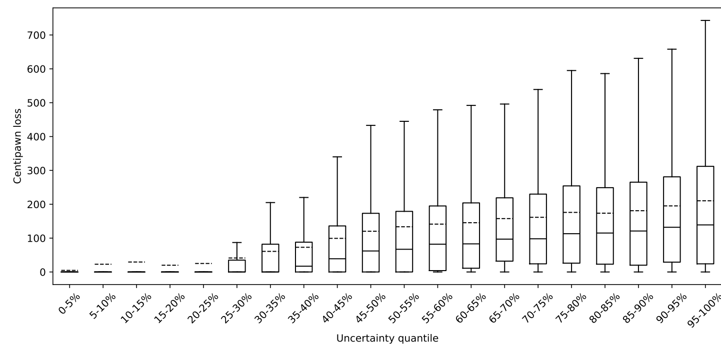


Figure 7.6: Scatter plot of per-position uncertainty and centipawn loss of the played move, overlaid for all four agents. Each point is one position from the fixed-game analysis.

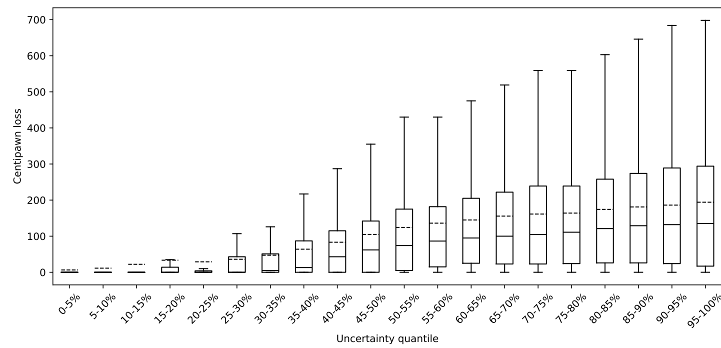
Table 7.4 reports the centipawn loss of each agent's played move and of the prior policy's argmax move, aggregated over all positions. Table 7.5a, Table 7.5b, and Table 7.5c restrict the same comparison to positions at or above each agent's 95th uncertainty percentile, below the 95th, and at or below the 50th, respectively. These data show that for many positions the action from the search policy gives higher CPL (according to Stockfish) than the argmax of the prior. This seems contradictory at first, and it is not immediately clear why. This effect seems to be amplified at positions with high uncertainty. When positions are ranked by uncertainty per agent, in those at or below the 50th uncertainty percentile the search action has lower mean CPL than the prior argmax across all four agents. This means that when the variance of returns under the chosen action is low, the search consistently produces actions that are better than the prior argmax. When it is high, we see the opposite. This demonstrates the importance of the value function in Gumbel AlphaZero-style learning. If the value function is not confident, the search itself can hurt the quality of the chosen action. It is also worth noting that the Gumbel noise at the root, which is necessary for exploration, can shift the search away from the prior argmax in high-stakes positions, leading to a worse chosen action even when the prior was already correct.



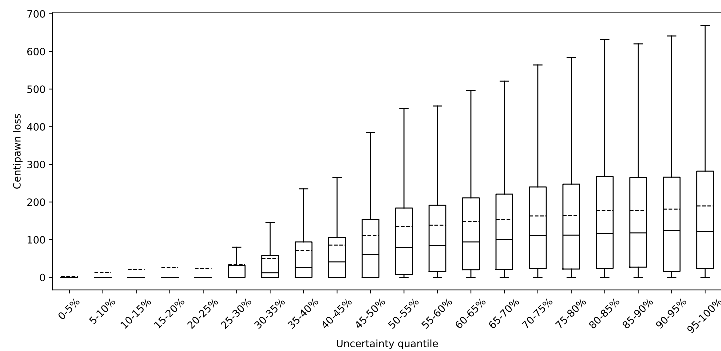
(a) Self-play.



(b) CBC.



(c) AIL-D.



(d) AIL.

Figure 7.7: Distribution of centipawn loss per uncertainty quantile bin for each agent, with positions split into 20 bins of 5 percentile points each and ordered by uncertainty. Each box reports the centipawn loss of the agent's played move within the bin. Whiskers extend to the standard 1.5 IQR. The solid horizontal line in each box is the median; the dashed line is the mean.

Figure 7.8 shows the centipawn loss distribution in each uncertainty span for both the agent's played move and the prior policy's argmax move. Two contrasting trends stand out. In the bottom 95% and bottom 50% bands the prior and search columns of the demonstration-trained agents both sit lower than those of Self-play, so in positions where the agents are already confident our methods do help. In the bottom 50% band the search also sits clearly below the prior argmax for all four agents, so within the low-uncertainty majority of positions the search is constructive regardless of demonstration training. The top 5% band tells the opposite story. Across all four agents the search action does not improve on the prior argmax: the medians clearly favor the prior, and the means favor it for three of the four agents. The demonstration-trained agents fail in the same way as Self-play, and nothing inherent to the proposed methods improves the quality of the search when uncertainty is already high.

We see this as the main flaw of the proposed methods. Both methods steer the prior toward expert-like actions, but in uncertain positions the search is dominated by the value-based term in Equation (3.2) rather than by the prior. The prior loses influence exactly where it would be most useful, and nothing in either method tells the agent *you are uncertain now, rely on what the expert has shown you rather than what you think*. This is a contradiction in the design of our methods: we set out to guide the agent through uncertain positions by working through the prior, yet in exactly those positions the value function, not the prior, dominates action selection. The demonstration-trained agents do appear marginally more confident on average, since Figure 7.5 shows them concentrating more uncertainty near zero, which we attribute to slightly more confident value functions on their guided trajectories. This average-case effect does not extend into the high-uncertainty tail where it would matter for action selection.

Addressing this directly would be to compensate for an uncertain value estimate with explicit expert guidance, either by regularizing the existing value function or by training an additional one. EfficientImitate [19], whose setting is pure imitation learning, trains a discriminator and uses its score to construct an AIL reward $R_E(s,a) = -\log(1 - D_\phi(s,a))$. The reward is then used inside the MCTS tree, where rewards along each search path and the value at the leaf combine into the bootstrapped return for that path. The same reward is the immediate reward in the n-step value targets used to train the value function, so their value function ends up approximating the expected cumulative R_E from each state. Adapting this to our AIL method, which showed the most promising results of the two methods that learned from demonstrations, one could train a separate value function V_E that approximates the expected cumulative R_E from a state:

$$V_E(s) \approx \mathbb{E}\left[\sum_t \gamma^t R_E(s_t, a_t)\right],$$

where γ is potential discounting, and let it bias the search toward states where V_E is high, gated against within-tree, within-episode, or across-episode uncertainty statistics, for example.

In EfficientImitate the demonstrations are full expert trajectories, so the discriminator

Agent	Search mean ↓	Search median ↓	Prior mean ↓	Prior median ↓
Self-play	125.6	48.0	121.1	27.0
CBC	108.6	**30.0**	103.7	**11.0**
AIL-D	**103.0**	33.0	**100.4**	**11.0**
AIL	103.3	33.0	101.4	**11.0**

Table 7.4: Centipawn loss for each of the four agents, aggregated over all positions of the 1,000 fixed-game analysis games per agent. Search columns report the centipawn loss of the move played by the agent (emitted by the search). Prior columns report the centipawn loss of the action that would have been chosen by taking the argmax of the prior policy in the same position. Mean and median of each distribution are reported.

matches both the state and action distributions of the expert, and with full expert trajectories $V_E(s)$ would have the clean interpretation we want: the expected expert-likeness of trajectories from s. In our setting demonstrations are collected at states the agent visits when uncertain, so the discriminator sees expert actions on the agent's state distribution. V_E would therefore say, from this point onwards, how are the action preferences of the agent expected to match the expert's. This potentially carries an unwanted side-effect: if $R_E(s, a)$ is typically more confident for positions with high variance of returns under the selected action, then V_E could inherit this as well, meaning that the expert-likeness potential given by V_E is higher for continuations with high uncertainty. If V_E has an unregulated influence over the search, then the search might prefer uncertain continuations rather than steer the agent through and away from them. This is not a problem in the same way for the AIL method we chose, since the discriminator only affects the decision at a single node. If the discriminator is less confident at that edge, we simply get a less confident answer there; the signal does not accumulate into a trajectory-level preference. Doing this cleanly under active demonstrations seems non-trivial, which is why we settled for the methods that appeared safer, possibly less effective. We leave this as a potential direction for future work.

Table 7.5: Centipawn loss restricted to three uncertainty spans. Search columns report the centipawn loss of the move played by the agent (emitted by the search). Prior columns report the centipawn loss of the action that would have been chosen by taking the argmax of the prior policy in the same position. Mean and median of each distribution are reported.

(a) Positions at or above each agent's 95th uncertainty percentile.

Agent	Search mean ↓	Search median ↓	Prior mean ↓	Prior median ↓
Self-play	202.0	128.0	194.7	104.0
CBC	210.2	139.0	**183.3**	**85.0**
AIL-D	194.1	135.0	189.0	101.0
AIL	**189.6**	**122.0**	194.9	102.0

(b) Positions below each agent's 95th uncertainty percentile.

Agent	Search mean ↓	Search median ↓	Prior mean ↓	Prior median ↓
Self-play	121.6	44.0	117.2	24.0
CBC	103.2	**24.0**	99.5	**7.0**
AIL-D	**98.2**	29.0	**95.7**	8.0
AIL	98.8	28.0	96.5	9.0

(c) Positions at or below each agent's 50th uncertainty percentile.

Agent	Search mean ↓	Search median ↓	Prior mean ↓	Prior median ↓
Self-play	67.6	0.0	78.4	0.0
CBC	49.7	0.0	59.0	0.0
AIL-D	43.8	0.0	56.3	0.0
AIL	**43.7**	0.0	**55.8**	0.0

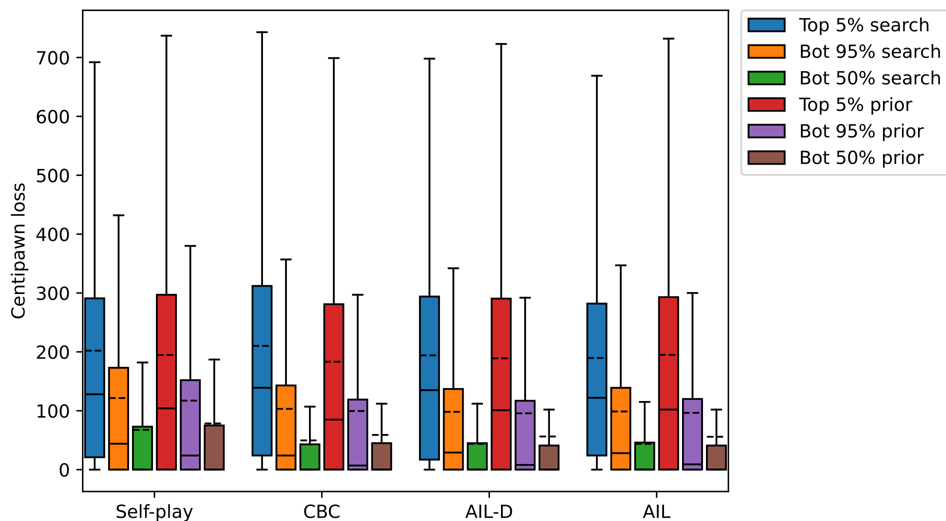


Figure 7.8: Distribution of centipawn loss per agent in three uncertainty spans. Top 5% covers positions at or above the agent's 95th uncertainty percentile. Bot 95% covers positions below the agent's 95th uncertainty percentile. Bot 50% covers positions at or below the agent's 50th uncertainty percentile. Each agent has six boxes, reporting the centipawn loss of the played move and the centipawn loss of the prior policy's argmax move in each span. Whiskers extend to the standard 1.5 IQR. The solid horizontal line in each box is the median. The dashed line is the mean.

7.6 Performance

We compare agent strength through two evaluations: local head-to-head tournaments between Self-play and each of the other agents, and Lichess Elo ratings obtained by deploying each agent as a bot account.

Table 7.6 shows the head-to-head tournament results. All three demonstration-trained agents beat Self-play with a clear margin. The demonstration-trained agents lose between 21.5% and 33.5% of games across the three matchups and three checkpoints, and very few games end in draws (at most 4 out of 200 in any single matchup). AIL-D loses the fewest games (21.5% to 23.5%), followed by AIL (28.5% to 31.5%) and CBC (29.5% to 33.5%), which sit close to each other. The ranking is stable across all three checkpoints, so we treat it as more than checkpoint-level noise.

The gap between AIL-D and AIL is something to highlight as well. AIL-D and AIL share the same trained model from the AIL run, and differ only in whether the discriminator score is added to the prior logits during search at evaluation time. At each checkpoint AIL-D loses between 5.5 and 8 percentage points fewer games than AIL, so the discriminator's contribution to the augmented logits, present in AIL-D and removed in AIL, continues to nudge search toward better moves at evaluation time, beyond whatever influence it had on the prior through training targets.

Read against research questions RQ1 and RQ2, the answer is at best weakly positive. Each matchup pairs the demonstration-trained agent against Self-play at the matched self-play sample budget (Table 7.1), so the consistent win advantage means that at

the same number of self-play samples the demonstration-trained agents are stronger. However, the demonstration runs also received roughly 1.2 million demonstration samples on top of that self-play budget, so this comparison does not isolate whether the demonstrations themselves drove the gap, or whether the same number of additional self-play samples for the baseline would have produced a comparable improvement. Even taken at face value, the gap is small relative to the supervision cost. The 1.2 million expert queries form an exaggerated budget, chosen to make the effects of each method easy to observe, and the resulting gap falls short of the meaningful efficiency gain we had hoped for. This is consistent with the analysis in Section 7.5 Analysis of Trained Agents: the demonstration methods improve play in positions where the agents are already confident, but cannot compensate for an uncertain value estimate in the high-uncertainty positions where the search struggles the most. Future methods that address this deficit more directly seem like the more promising path.

Table 7.7 reports the Lichess Elo rating of each deployed agent. For the AIL run we evaluate its performance with the discriminator (AIL-D), as it outperformed itself without the discriminator in the head-to-head tournaments (Table 7.6) and can be seen as part of the agent itself. The Lichess rating gives a rating relative to other bots and should not be confused with a rating relative to human players. On that within-pool scale the agents land close together, with AIL-D slightly above the others, in line with the local tournament results. Against Lichess's own Rapid rating distribution, the agents sit above 30% to 32% of human Rapid players, but we do not take this comparison at face value, for the reason stated above. Although the agents have clearly learned and play with apparent strategy, they are not at a level anywhere near the expert (Stockfish at 100,000 nodes) in playing strength.

Table 7.6: Head-to-head tournament results between Self-play and each of CBC, AIL-D, and AIL at three matched training checkpoints (steps 19,000, 19,500, and 20,000). In each matchup both agents are loaded from their checkpoint at the same step. Each matchup consists of 200 games, with each agent playing 100 as White and 100 as Black. Cells report the listed opponent's Wins, Draws, and Losses against Self-play as a percentage of the 200 games in each matchup.

	Step 19,000			Step 19,500			Step 20,000		
Opponent	Wins ↑	Draws	Losses ↓	Wins ↑	Draws	Losses ↓	Wins ↑	Draws	Losses ↓
CBC	70.0%	0.5%	29.5%	65.5%	1.0%	33.5%	66.5%	1.5%	32.0%
AIL-D	**78.0%**	0.5%	**21.5%**	**75.0%**	2.0%	**23.0%**	**75.0%**	1.5%	**23.5%**
AIL	71.0%	0.5%	28.5%	70.5%	1.0%	28.5%	67.5%	1.0%	31.5%

Agent	Username	Rating ↑	Deviation	Games
Self-play	sigraj-selfplay	1195	46.00	238
CBC	sigraj-bc	1187	45.54	248
AIL-D	sigraj-ail	**1198**	45.02	240

Table 7.7: Lichess Rapid Elo ratings of each deployed agent.

8

Ethical Considerations

This research involves no human participants and no personal or sensitive data. The expert used for demonstrations is Stockfish, a chess engine. No ethics approval was required. The training experiments were conducted on rented hardware, on machines with RTX 5090 and RTX PRO 6000 graphics cards. Around 340 GPU-hours were spent in total. The three main runs are summarized in Table 7.1, with the remainder spent on exploratory experiments, debugging, and failed experiments. The implementation has been made publicly available to support reproducibility. The methods explored in this thesis are applied to board games, a low-risk domain. If applied to more consequential domains such as robotics or autonomous systems, additional ethical considerations regarding safety and oversight would apply.

8.1 Licensing

The implementation made publicly available [49] is released under GPL-3.0 to be compatible with Stockfish [23] and Shakmaty [50], both released under GPL-3.0. Burn [18], a significant dependency of this implementation, is licensed under MIT and Apache-2.0, both compatible with GPL-3.0. All other dependencies have licenses that are compatible with GPL-3.0. No external datasets were used since all training data was generated either through self-play or provided by the demonstrator (Stockfish in the reported experiments).

8.2 Disclosure of AI Tool Usage

AI tools have been used extensively throughout this thesis. The primary tools were Claude [51] by Anthropic (Sonnet and Opus models, used both through the web interface and locally through Claude Code) and ChatGPT [52] by OpenAI (used both through the web interface and locally through Codex). For the written report, AI was used for refinement of text, including resolving ambiguous wording, correcting grammar and spelling, and improving the structure and flow of chapters. During the research phase, AI was used to brainstorm ideas, discuss potential research directions, find relevant papers through deep research modes, and summarize papers to efficiently identify the most relevant sources. AI was also used to discuss, summarize, and visualize experimental results. For the implementation, AI was used to write boilerplate code, suggest and restructure the codebase during large

structural changes, verify correctness of code, discover bugs and optimization issues, and generate plots. The author takes full responsibility for all statements and facts in this thesis, and all AI-generated output was reviewed and verified before inclusion.

9

Conclusion

This thesis extended Gumbel AlphaZero with two active-demonstration methods, corrective behavioral cloning and discriminator-based prior nudging, together with a query criterion based on the empirical variance of returns under the action selected by search. The implementation is itself a contribution. To our knowledge it is the first practical AlphaZero-style system written purely in Rust that is open source. It supports vanilla AlphaZero and Gumbel AlphaZero, runs on consumer hardware, includes environments for chess, Gomoku 8x8, Connect Four, and Tic-tac-toe, and is extensible to other board games. The repository also includes a companion program with a user interface for playing against the trained agents and for providing expert demonstrations during training. A snapshot of the implementation is available at `https://github.com/siggerajamae/itl-public`.

Restating the research questions tied to our methods:

RQ1
: Does corrective behavioral cloning (Section 4.3 Corrective Behavioral Cloning) improve sample efficiency over the Gumbel AlphaZero baseline, and if so, to what degree?

RQ2
: Does discriminator-based prior nudging (Section 4.4 Nudging the Prior) improve sample efficiency over the Gumbel AlphaZero baseline, and if so, to what degree?

RQ3
: Does the uncertainty measure (Section 4.2 Querying the Expert) identify positions where expert demonstrations are likely to be informative?

In the head-to-head tournament (Table 7.6) we see how the agents trained using our extensions compare against the agent trained solely through self-play. We refer to each specific combination of trained agent and evaluation settings as a *configuration*. The agent trained with corrective behavioral cloning (the CBC configuration) wins between 65.5 and 70.0% of games against Self-play. The agent trained with discriminator-based prior nudging was evaluated in two configurations, with the discriminator present (AIL-D) and without it (AIL). The former wins between 75.0 and 78.0% of games against Self-play, and the latter between 67.5 and 71.0%. Of these configurations, Self-play, CBC, and AIL-D were also evaluated against the pool of bots on Lichess (Table 7.7), where each received an Elo rating. Here we saw no meaningful separation between the evaluated configurations, as

their ratings fell within each other's deviations. With these combined measures of performance we can make some observations. First, neither of the proposed extensions appears decisively better at reducing the number of self-play games needed to learn, as all three demonstration-trained configurations performed well against Self-play. Second, versus the pool of Lichess bots, the demonstration-trained configurations did not appear stronger than Self-play, putting into question whether these methods have any significant effect on performance when matched against bots other than ones trained in our program.

With these results, we take into consideration the cost of supplementing active demonstrations. In the experiments we found that the challenge of optimizing hardware utilization became more difficult with awaited expert demonstrations preventing games from continuing, making the number of inference requests infrequent, and specifically for discriminator-based prior nudging, the additional overhead of discriminator inference. For the experimental settings used (Table 7.1), chosen to make runs comparable, CBC required 19h 34m and AIL required 21h 53m against 11h 25m for Self-play, increases of roughly 70% and 90%. Also, the number of supplemented demonstrations, targeting a fraction of 5% (roughly 1.2M realized) of encountered positions for demonstration, is an exaggerated amount. Whether a different target fraction of demonstrations would have produced similar results cannot be concluded, nor what the effect would have been had the baseline instead received an equivalent number of additional self-play samples, as demonstrations were provided on top of the existing self-play budget.

For RQ1, from the results above and our analysis (Section 7.5 Analysis of Trained Agents), it seems that extending learning with corrective behavioral cloning had some effect, but not to a degree that suggests it is a practical extension that reduces the number of self-play games needed in Gumbel AlphaZero. What we can say is that this extension does not appear destructive to learning, and may be a safe direction for future work, as part of an extension that also complements the value estimate.

Similarly, for RQ2, it seems discriminator-based prior nudging had some effect on performance, but not to a meaningful degree. In the head-to-head tournament (Table 7.6) we saw that the AIL agent performed better when evaluated with its discriminator (AIL-D). Since these configurations share the same trained agent and differ only in whether the discriminator score augments the prior logits at evaluation time, this shows that the discriminator continues to steer the search toward better moves, beyond whatever effect it had on the prior during training. We did not systematically tune the influence of the discriminator score, controlled by c_{disc}, and the method might have performed better had we done so, or had we made the influence adapt to the uncertainty in the search tree. Like corrective behavioral cloning, discriminator-based prior nudging does not appear destructive to learning, and a similar extension that also makes a deliberate effort to complement the self-learned value estimate could be an interesting direction for future work.

We attribute the shortfall of both extensions to the same underlying cause, which our analysis (Section 7.5 Analysis of Trained Agents) points to. Both proposed methods steer the learned prior toward expert-like actions. In the high-uncertainty positions

where the search needs guidance the most, action selection comes to rely on the value estimate rather than on the prior. The prior loses influence in exactly the positions where the demonstrations were collected, so neither method is effective there. We view this as the primary reason our methods did not produce the efficiency gain we had hoped for.

For RQ3, the variance-of-returns uncertainty measure correlates clearly with centipawn loss in the fixed-game analysis. Positions with high variance under the selected action concentrate the agents' mistakes, so the measure does identify high-stakes positions in a useful way. The correlation is not a clean separation. High uncertainty does not always imply a mistake, and not every mistake comes from a high-uncertainty position. The relationship is still consistent across all four evaluated configurations and supports return variance as a query criterion in future work.

To summarize, both extensions steer the learned prior and the search toward expert-like moves, without taking any extra measures to ensure that the value estimate also reflects expert behavior. In the broader goal of making learning new or niche games practical on limited hardware with fewer self-play games, these methods have not been successful. Demonstrations are demanding, the extensions make training more difficult to optimize, and the sacrifice is too large relative to the gains. Demonstrations should be treated as a limited resource, and these methods would be impractical if the goal is a method that works with human demonstrators, as they would be demanding for a human to supply for this small benefit. We evaluated using an exaggerated number of demonstrations from Stockfish at 100,000 nodes, a program far stronger than any of the trained agents, so the shortfall is not due to weak expert guidance. Whether the effect would have been the same using a human demonstrator or with fewer demonstrations cannot be concluded, but we expect those gains to be less significant. We cannot conclude whether the extensions would yield similar gains in other domains, as we only evaluated them on chess. Given the limited scope of experiments and that we cannot rule out that the extensions would have performed better had we systematically tuned all hyperparameters, we cannot conclude that these particular extensions could not yield more meaningful results under different conditions, nor can we disregard the broader line of research on methods that complement self-play with active learning from demonstrations.

9.1 Limitations

Here we summarize the limitations covered throughout this thesis. The rationale behind these extensions was that active demonstrations directly give us a direction for improving the prior, as that is what encodes learned action preferences, but they say nothing further about the outcomes of games (Section 4.3 Corrective Behavioral Cloning). For that reason, both our methods seek to complement the prior in two distinct ways, steering it toward expert-like actions, but without taking any extra measures to ensure that the value estimate also reflects expert behavior. In the high-uncertainty positions where the search needs guidance the most, action selection comes to rely on the value estimate rather than on the prior (Section 7.5 Analysis of Trained Agents). The prior loses influence in exactly the positions where the

demonstrations were collected, so neither method is effective there.

The scope of our experiments is limited. AlphaZero-style training requires a large number of self-play games, and the compute available for this thesis was limited (Chapter 6 Experiments). We therefore trained each agent only once, so our results do not account for the variability that comes from the randomness in sample production and model optimization. We only evaluated the extensions on chess, so we cannot conclude whether they would yield similar gains in other domains. We did not systematically tune all hyperparameters, including the discriminator weight c_{disc}, and the extensions might have performed better had we done so. The networks were also scaled down and given a simpler input plane representation that encodes only the positions of the pieces and not information such as castling rights, en passant, repetitions, and the no-progress counter, though this information remains available to the search (Section 5.4 Network Architecture). We made these simplifications for efficiently conducting experiments where end performance is not as important as seeing the relative effects of the proposed methods.

We used an exaggerated amount of expert supervision, roughly five percent of produced positions, chosen to make the effects of each method easier to observe (Chapter 6 Experiments). We therefore cannot conclude whether a smaller target fraction of demonstrations or a human demonstrator in place of Stockfish would have produced the same effect. This is the practical setting the thesis targets, where the agent learns from fewer self-play games with a limited number of demonstrations that may come from a human (Section 4.1 Approach and Rationale).

The training runs in our experiments (Chapter 6 Experiments) differed for practical reasons. They used a different number of producers, 120 for Self-play, 180 for CBC, and 240 for AIL, affecting the frequency of model updates during games (Section 7.1 Overview of Runs). We did not isolate the exact effect the number of concurrent producers had on learning, although in preliminary experiments it seemed insignificant. Because producers never synchronize with the trainer, model updates can even happen during a search, which likewise did not prevent learning (Chapter 5 Implementation). The runs were also trained on separately rented machines with different hardware, which affects the wall-clock comparison between them.

Our evaluation methods were limited in scope, to measure the performance gains provided by the developed extensions. The local head-to-head tournament evaluated the extensions against a baseline trained under the same conditions in the same program, and the Lichess ratings gave a broader measure of performance against other bots (Section 7.6 Performance). Both the tournament and the Lichess ratings should provide limited evidence, since the training runs were not repeated, as noted above. Further, since the Lichess Elo ratings are incomparable to human Elo ratings, we cannot provide evidence for how the agents perform against human players. Together they let us conclude whether the extensions improved performance, but not why they did or did not. For this reason we separately analyzed each agent on self-played games (Section 7.5 Analysis of Trained Agents), which gave deeper insight, but we could have expanded this analysis with other evaluation techniques (Section 4.1 Approach and Rationale), such as how closely an agent follows established human

openings, whether its decisions relate to concepts central to human chess strategy, or how well it predicts the moves of human players, which might have helped explain why our extensions fell short of meaningful performance gains.

9.2 Future Work

We close by summarizing the directions for future work that we deem most promising. As indicated by our analysis (Section 7.5 Analysis of Trained Agents), the quality of value estimates plays an important role in the search, as the quality of moves preferred by the search degrades for continuations where the value estimate seems uncertain. The extensions developed here only complement the prior, so future work that develops similar extensions should not neglect the importance of the value estimate, and instead target it directly. We discuss the idea of a potential-based hinge loss in Section 4.3 Corrective Behavioral Cloning, and, in Section 7.5 Analysis of Trained Agents, that of a separate value function trained to predict the expert-like potential of continuations from a position.

Another direction follows from our uncertainty measure. Our analysis (Section 7.5 Analysis of Trained Agents) shows that the variance of returns under the selected action identifies high-stakes positions. These are exactly the positions where the extensions are not effective, because there the prior, which carries the expert's influence, gives way to the value estimate. Future work could make the influence of what was learned from the expert adapt to this uncertainty, growing stronger where the agent is more uncertain. The influence of the discriminator score, for instance, could be made to depend on the uncertainty in the search tree rather than on a fixed weight (Section 4.4 Nudging the Prior). Other measures of uncertainty could also be explored, such as extending the network with multiple value heads (Section 3.9 Measuring Uncertainty).

Future work could also broaden this line of research to target other domains, or learning in the absence of environment rewards. The extensions could be applied to other board games, or to other categories of domains such as the ones targeted by EfficientImitate [19]. For learning in the absence of environment rewards, EfficientImitate performs pure imitation learning from a fixed dataset of demonstrations, and a version that instead queries them actively during training, as this thesis does, could be explored.

Bibliography

[1] C. E. Shannon, "Xxii. programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950. DOI: 10.1080/14786445008521796 eprint: https://doi.org/10.1080/14786445008521796. [Online]. Available: https://doi.org/10.1080/14786445008521796

[2] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM J. Res. Dev.*, vol. 3, no. 3, pp. 210–229, Jul. 1959, ISSN: 0018-8646. DOI: 10.1147/rd.33.0210 [Online]. Available: https://doi.org/10.1147/rd.33.0210

[3] G. Tesauro, "Td-gammon, a self-teaching backgammon program, achieves master-level play," *Neural Comput.*, vol. 6, no. 2, pp. 215–219, Mar. 1994, ISSN: 0899-7667. DOI: 10.1162/neco.1994.6.2.215 [Online]. Available: https://doi.org/10.1162/neco.1994.6.2.215

[4] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction* (Adaptive computation and machine learning series), en, Second edition. Cambridge, Massachusetts: The MIT Press, 2018, ISBN: 978-0-262-03924-6.

[5] D. Silver et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016. [Online]. Available: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html

[6] D. Silver et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017, ISSN: 1476-4687. DOI: 10.1038/nature24270 [Online]. Available: https://doi.org/10.1038/nature24270

[7] D. Silver et al., "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. DOI: 10.1126/science.aar6404 eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aar6404

[8] I. Danihelka, A. Guez, J. Schrittwieser, and D. Silver, "Policy improvement by planning with gumbel," in *International Conference on Learning Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=bERaNdoegnO

[9] D. J. Wu, "Accelerating self-play learning in go," *CoRR*, vol. abs/1902.10565, 2019. arXiv: 1902.10565. [Online]. Available: http://arxiv.org/abs/1902.10565

[10] The LCZero Authors, *LeelaChessZero*, 2026. [Online]. Available: https://github.com/LeelaChessZero/lc0

[11] J. Schrittwieser, T. Hubert, A. Mandhane, M. Barekatain, I. Antonoglou, and D. Silver, "Online and offline reinforcement learning by planning with a learned model," *CoRR*, vol. abs/2104.06294, 2021. arXiv: 2104.06294. [Online]. Available: https://arxiv.org/abs/2104.06294

[12] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2008.10.024 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889008001772

[13] M. Zare, P. M. Kebria, A. Khosravi, and S. Nahavandi, "A survey of imitation learning: Algorithms, recent developments, and challenges," *IEEE Transactions on Cybernetics*, vol. 54, no. 12, pp. 7173–7186, 2024. DOI: 10.1109/TCYB.2024.3395626

[14] A. Fawzi et al., "Discovering faster matrix multiplication algorithms with reinforcement learning," *Nature*, vol. 610, pp. 47–53, Oct. 2022. DOI: 10.1038/s41586-022-05172-4

[15] D. J. Mankowitz et al., "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7964, pp. 257–263, Jun. 2023, ISSN: 1476-4687. DOI: 10.1038/s41586-023-06004-9 [Online]. Available: https://doi.org/10.1038/s41586-023-06004-9

[16] M. H. S. Segler, M. Preuss, and M. P. Waller, "Planning chemical syntheses with deep neural networks and symbolic ai," *Nature*, vol. 555, no. 7698, pp. 604–610, Mar. 2018, ISSN: 1476-4687. DOI: 10.1038/nature25978 [Online]. Available: https://doi.org/10.1038/nature25978

[17] The Rust Project Developers, *The Rust programming language*, 2026. [Online]. Available: https://www.rust-lang.org

[18] N. Simard, L. Fortier-Dubois, D. Tadjibaev, G. Lagrange, and Burn Framework Contributors, *Burn*, version 0.21.0, May 2026. [Online]. Available: https://github.com/tracel-ai/burn

[19] Z.-H. Yin, W. Ye, Q. Chen, and Y. Gao, *Planning for sample efficient imitation learning*, 2022. arXiv: 2210.09598 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2210.09598

[20] M. Weichart, *Variance-aware prior-based tree policies for monte carlo tree search*, 2026. arXiv: 2512.21648 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2512.21648

[21] S.-A. Chen, V. Tangkaratt, H.-T. Lin, and M. Sugiyama, "Active deep q-learning with demonstration," *Machine Learning*, vol. 109, no. 9, pp. 1699–1725, Sep. 2020, ISSN: 1573-0565. DOI: 10.1007/s10994-019-05849-4 [Online]. Available: https://doi.org/10.1007/s10994-019-05849-4

[22] F. L. Da Silva, P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "Uncertainty-aware action advising for deep reinforcement learning agents," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, pp. 5792–5799, Apr. 2020. DOI: 10.1609/aaai.v34i04.6036 [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/6036

[23] The Stockfish developers, *Stockfish*, 2026. [Online]. Available: https://stockfishchess.org/

[24] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, "Deep exploration via bootstrapped dqn," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29, Curran Associates, Inc., 2016. [Online]. Available: `https://proceedings.neurips.cc/paper_files/paper/2016/file/8d8818c8e140c64c743113f563cf750f-Paper.pdf`

[25] T. Hubert, J. Schrittwieser, I. Antonoglou, M. Barekatain, S. Schmitt, and D. Silver, "Learning and planning in complex action spaces," *CoRR*, vol. abs/2104.06303, 2021. arXiv: `2104.06303`. [Online]. Available: `https://arxiv.org/abs/2104.06303`

[26] J. Schrittwieser et al., "Mastering atari, go, chess and shogi by planning with a learned model," *CoRR*, vol. abs/1911.08265, 2019. arXiv: `1911.08265`. [Online]. Available: `http://arxiv.org/abs/1911.08265`

[27] Y. Ou and M. Tavakoli, "Towards safe and efficient reinforcement learning for surgical robots using real-time human supervision and demonstration," in *2023 International Symposium on Medical Robotics (ISMR)*, 2023, pp. 1–7. DOI: `10.1109/ISMR57123.2023.10130214`

[28] M. Hou, K. Hindriks, A. E. Eiben, and K. Baraka, *"give me an example like this": Episodic active reinforcement learning from demonstrations*, 2024. arXiv: `2406.03069 [cs.AI]`. [Online]. Available: `https://arxiv.org/abs/2406.03069`

[29] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine Learning Proceedings 1994*, W. W. Cohen and H. Hirsh, Eds., San Francisco (CA): Morgan Kaufmann, 1994, pp. 157–163, ISBN: 978-1-55860-335-6. DOI: `https://doi.org/10.1016/B978-1-55860-335-6.50027-1` [Online]. Available: `https://www.sciencedirect.com/science/article/pii/B9781558603356500271`

[30] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957, ISSN: 00959057, 19435274. Accessed: May 3, 2026. [Online]. Available: `http://www.jstor.org/stable/24900506`

[31] B. Abramson, "Expected-outcome: A general model of static evaluation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 2, pp. 182–193, 1990. DOI: `10.1109/34.44404`

[32] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and Games*, H. J. van den Herik, P. Ciancarini, and H. H. L. M. (Donkers, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83, ISBN: 978-3-540-75538-8.

[33] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293, ISBN: 978-3-540-46056-5.

[34] Z. Karnin, T. Koren, and O. Somekh, "Almost optimal exploration in multi-armed bandits," in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., ser. Proceedings of Machine Learning Research, vol. 28, Atlanta, Georgia, USA: PMLR, Jun. 2013, pp. 1238–

1246. [Online]. Available: https://proceedings.mlr.press/v28/karnin13.html

[35] J. Grill et al., "Monte-carlo tree search as regularized policy optimization," *CoRR*, vol. abs/2007.12509, 2020. arXiv: 2007.12509. [Online]. Available: https://arxiv.org/abs/2007.12509

[36] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, G. Gordon, D. Dunson, and M. Dudík, Eds., ser. Proceedings of Machine Learning Research, vol. 15, Fort Lauderdale, FL, USA: PMLR, Nov. 2011, pp. 627–635. [Online]. Available: https://proceedings.mlr.press/v15/ross11a.html

[37] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Machine Learning, Proceedings of the Twenty-first International Conference (ICML 2004), Banff, Alberta, Canada, July 4-8, 2004*, C. E. Brodley, Ed., ser. ACM International Conference Proceeding Series, ACM, 2004. DOI: 10.1145/1015330.1015430 [Online]. Available: https://doi.org/10.1145/1015330.1015430

[38] B. Kang, Z. Jie, and J. Feng, "Policy optimization with demonstrations," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, Oct. 2018, pp. 2469–2478. [Online]. Available: https://proceedings.mlr.press/v80/kang18a.html

[39] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29, Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2016/file/cc7e2b878868cbae992d1fb743995d8f-Paper.pdf

[40] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962. DOI: 10.1080/00401706.1962.10490022 eprint: https://www.tandfonline.com/doi/pdf/10.1080/00401706.1962.10490022. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/00401706.1962.10490022

[41] A. Y. Ng, D. Harada, and S. J. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proceedings of the Sixteenth International Conference on Machine Learning*, ser. ICML '99, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 278–287, ISBN: 1558606122.

[42] P. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, *Deep reinforcement learning from human preferences*, 2023. arXiv: 1706.03741 [stat.ML]. [Online]. Available: https://arxiv.org/abs/1706.03741

[43] T. McGrath et al., "Acquisition of chess knowledge in alphazero," *Proceedings of the National Academy of Sciences*, vol. 119, no. 47, e2206625119, 2022. DOI: 10.1073/pnas.2206625119 eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2206625119. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.2206625119

[44] R. McIlroy-Young, S. Sen, J. Kleinberg, and A. Anderson, “Aligning superhuman ai with human behavior: Chess as a model system,” in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD ’20, Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 1677–1687, ISBN: 9781450379984. DOI: `10.1145/3394486.3403219` [Online]. Available: `https://doi.org/10.1145/3394486.3403219`

[45] Tokio Contributors, *Tokio: An asynchronous Rust runtime*, 2026. [Online]. Available: `https://tokio.rs`

[46] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, *Spectral normalization for generative adversarial networks*, 2018. arXiv: `1802.05957 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/1802.05957`

[47] Vast.ai, *About vast.ai*, Accessed: 2026-06-18, 2026. [Online]. Available: `https://vast.ai/about`

[48] The Lichess contributors, *Lichess*, 2026. [Online]. Available: `https://lichess.org/`

[49] S. Rajamäe, *Itl-public*, 2026. [Online]. Available: `https://github.com/siggerajamae/itl-public`

[50] N. Fiekas, *Shakmaty*, 2026. [Online]. Available: `https://github.com/niklasf/shakmaty`

[51] Anthropic, *Claude*, Accessed: 2026-06-03, 2025. [Online]. Available: `https://www.anthropic.com/claude`

[52] OpenAI, *ChatGPT*, Accessed: 2026-06-03, 2022. [Online]. Available: `https://openai.com/index/chatgpt/`