UNIVERSITY OF CALIFORNIA,
IRVINE

Sequential Decision Making in Single-Agent and Multi-Agent Domains

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Stephen McAleer

Dissertation Committee:
Distinguished Professor Pierre Baldi, Chair
Assistant Professor Roy Fox
Assistant Professor Ioannis Panageas
Associate Professor Sameer Singh
Professor Steven Barwick

2022

DEDICATION

To Leo, Ann Marie, Mom, Dad, Tim, and Jacqueline.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

Chapter 1 is based on McAleer et al. [72] and Agostinelli et al. [1]. Both papers were co-first-authored with Forest Agostinelli and Alexander Shmakov, and co-authored with Pierre Baldi. McAleer et al. [72] was published in the International Conference on Learning Representations (ICLR) and Agostinelli et al. [1] was published in Nature Machine Intelligence.

Chapter 2 is based on McAleer et al. [74] published in Advances in Neural Information Processing Systems (NeurIPS). This was co-first-authored with J.B. Lanier and co-authored with Roy Fox and Pierre Baldi. During this work I was supported in part by grant NSF 1839429 to Pierre Baldi.

Chapter 3 is based on McAleer et al. [71] published in Advances in Neural Information Processing Systems (NeurIPS). This paper was co-authored with J.B. Lanier, Kevin Wang, Pierre Baldi, and Roy Fox.

I would like to sincerely thank my advisor Pierre Baldi for his mentorship and support. I am truly grateful for all he has done for me.

I would also like to especially thank Roy Fox for his close collaboration and valuable mentorship.

Thank you to my thesis committee Pierre Baldi, Roy Fox, Ioannis Panageas, Sameer Singh, and Steve Barwick.

Thank you to all my coauthors and collaborators, particularly J.B. Lanier for his close teamwork over the past few years.

I would also like to thank the MAPS: Machine Learning and Physical Sciences National Science Foundation/UCI Graduate Training Program for funding, training, and support during my PhD.

Thank you also to Yuzo Kanomata for computing support and to Janet Ko for administrative support.

Finally, thank you to Laurent Orseau, Marc Lanctot, Karl Tuyls, Julien Perolat, Michael Bowling, Bart De Vylder, Thomas Anthony, Luke Marris, Daniel Hennes, Paul Muller, Siqi Liu, Romuald Elie, Sombdeb Majumdar, Shauharda Khadka, and many others at DeepMind and Intel for all your help and collaboration during my internships.

VITA

Stephen McAleer

EDUCATION

Doctor of Philosophy in Computer Science **2022**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2021**
University of California, Irvine *Irvine, California*

Bachelor of Science in Mathematics and Economics **2017**
Arizona State University *Tempe, Arizona*

RESEARCH EXPERIENCE

Postdoctoral Fellow **2022–Present**
Carnegie Mellon University *Pittsburgh, Pennsylvania*

Graduate Research Assistant **2017–2022**
University of California, Irvine *Irvine, California*

Research Scientist Intern **2021–2021**
DeepMind *Remote*

Research Scientist Intern **2019–2019**
Intel AI *San Diego, California*

SELECTED PUBLICATIONS

XDO: A Double Oracle Algorithm for Extensive Form Games. **2021**
Neural Information Processing Systems (NeurIPS)

Pipeline PSRO: A Scalable Approach for Finding Approximate Nash Equilibria in Large Games. **2020**
Neural Information Processing Systems (NeurIPS)

Solving the Rubik's Cube with Approximate Policy Iteration. **2019**
International Conference on Learned Representations (ICLR)

ABSTRACT OF THE DISSERTATION

Sequential Decision Making in Single-Agent and Multi-Agent Domains

By

Stephen McAleer

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Distinguished Professor Pierre Baldi, Chair

In this dissertation I outline three contributions. The first is in the area of single-agent planning. In this section I describe my work on combining deep reinforcement learning with search to solve hard planning problems such as the Rubik's cube. In the next two sections I focus on contributions in the area of two-player zero-sum games. Both contributions are improvements to an existing double oracle algorithm called Policy Space Response Oracles (PSRO). The first improves PSRO by parallelizing PSRO while maintaining convergence guarantees. The resulting method, called Pipeline PSRO (P2SRO) achieves state-of-the-art performance on Barrage Stratego. The second contribution, called Extensive-Form Double Oracle (XDO) improves PSRO by extending it to extensive-form games. As a result, XDO is able to converge to a Nash equilibrium in a number of iterations linear in the number of infostates of the game, as opposed to a potentially exponential number of iterations necessary to insure convergence for PSRO. A neural version of XDO achieves state-of-the-art performance on a continuous-action game.

Chapter 1

Introduction

A major branch of of machine learning studies the problem of learning optimal sequential decision making under uncertainty. Most research in this branch studies the problem from the perspective of a single decision maker, using techniques such as search, optimal control, and reinforcement learning. In sequential decision making, actions taken at one time step affect the state of the environment at future timesteps, so optimal policies must consider both the short-term and long-term effects of actions. Learning approximately optimal policies from data provides further challenges, as exploration and exploitation need to be balanced. In this thesis we present one such project that combines deep reinforcement learning with search to solve planning problems.

Even more complicated than the single-agent domain is the multi-agent domain. In this problem setting, agents must learn to make optimal sequential decisions in the presence of other agents that are also potentially learning. This complicates the problem because now from the perspective of a single agent, the environment is not stationary. Thankfully, in certain domains such as two-player zero-sum games, we can combine game-theoretic concepts with techniques from single-agent sequential decision making to learn optimal

policies. In particular, in this thesis we combine the game-theoretic double oracle algorithm with deep reinforcement learning to learn an approximate Nash equilibrium, which is optimal in two-player zero-sum games. Below is a summary of each section of the thesis.

1.1 Solving the Rubik's Cube with Deep Reinforcement Learning and Search

The Rubik's Cube is a prototypical combinatorial puzzle that has a large state space with a single goal state. The goal state is unlikely to be accessed using sequences of randomly generated moves, posing unique challenges for machine learning. We solve the Rubik's Cube with DeepCubeA, a deep reinforcement learning approach that learns how to solve increasingly difficult states in reverse from the goal state without any specific domain knowledge. DeepCubeA solves 100% of all test configurations, finding a shortest path to the goal state 60.3% of the time. DeepCubeA generalizes to other combinatorial puzzles and is able to solve the 15-puzzle, 24-puzzle, 35-puzzle, 48-puzzle, Lights Out, and Sokoban. For all puzzles in which a shortest path solution can be computed tractably, DeepCubeA finds a shortest path for the majority of test configurations.

1.2 Pipeline PSRO: A Scalable Approach for Finding Approximate Nash Equilibria in Large Games

Finding approximate Nash equilibria in zero-sum imperfect-information games is challenging when the number of information states is large. Policy Space Response Oracles (PSRO) is a deep reinforcement learning algorithm grounded in game theory that is guaranteed to converge to an approximate Nash equilibrium. However, PSRO requires training a reinforcement

learning policy at each iteration, making it too slow for large games. We show through counterexamples and experiments that DCH and Rectified PSRO, two existing approaches to scaling up PSRO, fail to converge even in small games. We introduce Pipeline PSRO (P2SRO), the first scalable PSRO-based method for finding approximate Nash equilibria in large zero-sum imperfect-information games. P2SRO is able to parallelize PSRO with convergence guarantees by maintaining a hierarchical pipeline of reinforcement learning workers, each training against the policies generated by lower levels in the hierarchy. We show that unlike existing methods, P2SRO converges to an approximate Nash equilibrium, and does so faster as the number of parallel workers increases, across a variety of imperfect information games. We also introduce an open-source environment for Barrage Stratego, a variant of Stratego with an approximate game tree complexity of 10^{50}. P2SRO is able to achieve state-of-the-art performance on Barrage Stratego and beats all existing bots. Experiment code is available at `https://github.com/JBLanier/pipeline-psro`.

1.3 XDO: A Double Oracle Algorithm For Extensive-Form Games

Policy Space Response Oracles (PSRO) is a reinforcement learning (RL) algorithm for two-player zero-sum games that has been empirically shown to find approximate Nash equilibria in large games. Although PSRO is guaranteed to converge to an approximate Nash equilibrium and can handle continuous actions, it may take an exponential number of iterations as the number of information states (infostates) grows. We propose Extensive-Form Double Oracle (XDO), an extensive-form double oracle algorithm for two-player zero-sum games that is guaranteed to converge to an approximate Nash equilibrium *linearly* in the number of infostates. Unlike PSRO, which mixes best responses at the root of the game, XDO mixes best responses at every infostate. We also introduce Neural XDO (NXDO), where the

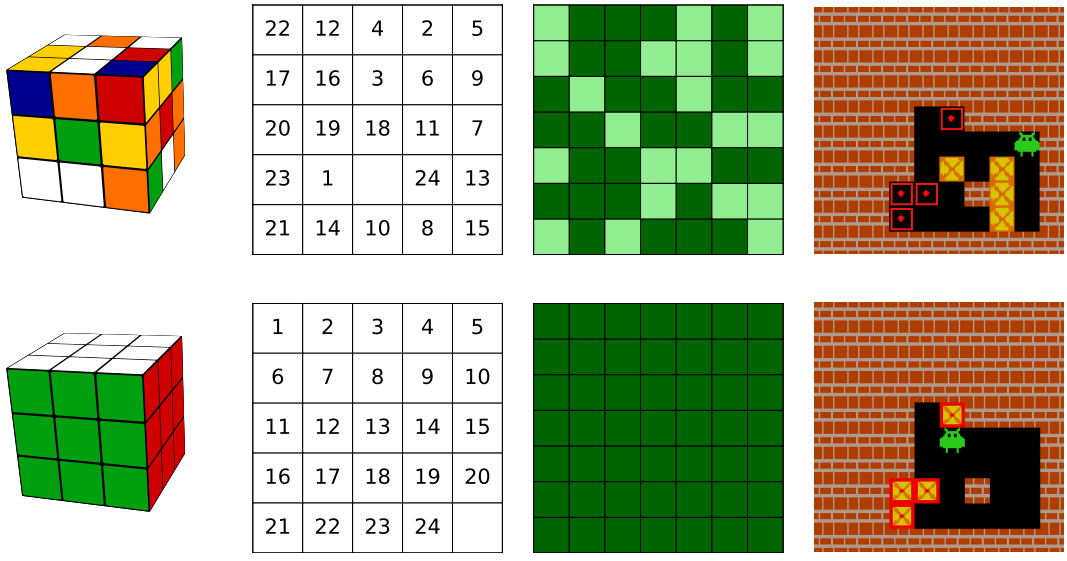
best response is learned through deep RL. In tabular experiments on Leduc poker, we find that XDO achieves an approximate Nash equilibrium in a number of iterations an order of magnitude smaller than PSRO. Experiments on a modified Leduc poker game and Oshi-Zumo show that tabular XDO achieves a lower exploitability than CFR with the same amount of computation. We also find that NXDO outperforms PSRO and NFSP on a sequential multidimensional continuous-action game. NXDO is the first deep RL method that can find an approximate Nash equilibrium in high-dimensional continuous-action sequential games. Experiment code is available at `https://github.com/indylab/nxdo`.

Chapter 2

Solving the Rubik's Cube with Deep Reinforcement Learning and Search

2.1 Introduction

The Rubik's Cube is a classic combinatorial puzzle that poses unique and interesting challenges for artificial intelligence and machine learning. Although the state space is exceptionally large (4.3×10^{19} different states), there is only one goal state. Furthermore, the Rubik's Cube is a single-player game and a sequence of random moves, no matter how long, is unlikely to end in the goal state. Developing machine learning algorithms to deal with this property of the Rubik's Cube might provide insights into learning to solve planning problems with large state spaces. While machine learning methods have previously been applied to the Rubik's Cube, these methods have either failed to reliably solve the cube[68, 99, 23, 49] or have had to rely on specific domain knowledge[54, 7]. Outside of machine learning methods, methods based on pattern databases have been effective at solving puzzles such as the Rubik's Cube, 15-puzzle, and 24-puzzle [55, 57]. However, pattern databases are very memory intensive because they



(a) Rubik's Cube (b) 24-puzzle (c) Lights Out (7x7) (d) Sokoban

Figure 2.1: A visualization of a scrambled state (top) and the goal state (bottom) for four of the puzzles investigated in this paper.

store the number of moves required to reach all possible states for multiple given subgoals. Generally speaking, the larger the subgoals, the more effective the heuristic. However, using larger subgoals may come at the expense of an exponential increase in memory usage. For example, for the 35-puzzle, a pattern database that divides the puzzle into 7 subgoals of size 5 would require 0.37 GB of memory while a pattern database that divides the puzzle into 5 subgoals of size 7 would require 321.70 GB of memory.

More broadly, a major goal in artificial intelligence is to create algorithms that are able to learn how to master various environments without relying on domain-specific human knowledge. The classical 3x3x3 Rubik's Cube is only one representative of a larger family of possible combinatorial puzzles, broadly sharing the characteristics described above, including: (1) cubes with longer edges or in higher dimension (e.g. 4x4x4 or 2x2x2x2); (2) sliding tile puzzles (e.g. the 15-puzzle, 24-puzzle, and 35-puzzle); (3) Lights Out; as well as (4) Sokoban. As the size and dimensions are increased, the complexity of the underlying combinatorial problems rapidly increases. For instance, while finding an optimal solution to the 15-puzzle takes less than a second on a modern day desktop, finding an optimal solution to the 24-puzzle

can take days, and finding an optimal solution to the 35-puzzle is generally intractable[30]. Not only are the aforementioned puzzles relevant as mathematical games, but they can also be used to test planning algorithms [14] and to assess how well a machine learning approach may generalize to different environments. Thus, a general algorithm that can be used to solve combinatorial puzzles would be a significant contribution to the field of automated planning [35]. Furthermore, since the operation of the Rubik's Cube and other combinatorial puzzles are deeply rooted in group theory, these puzzles also raise broader questions about the application of machine learning methods to complex symbolic systems, including mathematics. In short, for all these reasons, the Rubik's Cube poses interesting challenges for machine learning.

To address these challenges, we develop DeepCubeA which combines deep learning [94, 38] with classical reinforcement learning [103] (approximate value iteration[11, 85, 13]) and path finding methods (weighted A* search[42, 84]). DeepCubeA is able to solve combinatorial puzzles such as the Rubik's Cube, 15-puzzle, 24-puzzle, 35-Puzzle, 48-puzzle, Lights Out, and Sokoban (see Figure 2.1). DeepCubeA works by using approximate value iteration to train a deep neural network (DNN) to approximate a function that outputs the cost to reach the goal (also known as the cost-to-go function). Since random play is unlikely to end in the goal state, DeepCubeA trains on states obtained by starting from the goal state and randomly taking moves in reverse. After training, the learned cost-to-go function is used as a heuristic to solve the puzzles using weighted A* search [42, 84, 29].

DeepCubeA builds upon DeepCube [73], a deep reinforcement learning algorithm that solves the Rubik's Cube using a policy and value combined with Monte Carlo tree search (MCTS). MCTS combined with a policy and value function is also used by AlphaZero which learns to beat the best existing programs in chess, Go, and shogi [98]. In practice, we find that, for combinatorial puzzles, MCTS has relatively long runtimes and often produces solutions many moves longer than the length of a shortest path. In contrast, DeepCubeA finds a shortest

path to the goal in the majority of cases, not only for the Rubik's Cube, but also for the 15-puzzle, 24-puzzle, and Lights Out.

2.2 Methods

2.2.1 The Rubik's Cube

The 3x3x3 Rubik's Cube consists of smaller cubes called cubelets. These are classified by their sticker count: center, edge, and corner cubelets have 1, 2, and 3 stickers, respectively. The Rubik's Cube has 26 cubelets with 54 stickers in total. The stickers have colors and there are six colors, one per face. In the solved state, all stickers on each face of the cube are the same color. Since the set of stickers on each cubelet is unique (i.e. there is only one cubelet with white, red, and green stickers), the 54 stickers themselves can be uniquely identified in any legal configuration of the Rubik's Cube. The representation given to the DNN encodes the color of each sticker at each location using a one-hot encoding. Since there are 6 possible colors and 54 stickers in total, this results in a state representation of size 324.

Moves are represented using face notation: a move is a letter stating which face to rotate. *F*, *B*, *L*, *R*, *U*, and *D* correspond to turning the *front*, *back*, *left*, *right*, *up*, and *down* faces, respectively. Each face name is in reference to a fixed front face. A clockwise rotation is represented with a single letter, whereas a letter followed by an apostrophe represents a counter-clockwise rotation. For example: *R* rotates the right face by 90° clockwise, while *R'* rotates it by 90° counter-clockwise.

The Rubik's Cube state space has 4.3×10^{19} possible states. Any valid Rubik's Cube state can be optimally solved with at most 26 moves in the quarter-turn metric, or 20 moves in the half-turn metric [89, 87]. The quarter-turn metric treats 180 degree rotations as two

moves, whereas the half-turn metric treats 180 degree rotations as one move. We use the quarter-turn metric.

2.2.2 Additional Combinatorial Puzzles

Sliding Puzzles

Another combinatorial puzzle we use to test DeepCubeA is the n piece sliding puzzle. In the n-puzzle, n square sliding tiles, numbered from 1 to n, are positioned in a square of length $\sqrt{n+1}$, with one empty tile position. Thus, the 15-puzzle consists of 15 tiles in a 4x4 grid, the 24-puzzle consists of 24 tiles in a 5x5 grid, the 35-puzzle consists of 35 tiles in a 6x6 grid, and the 48-puzzle consists of 48 tiles in a 7x7 grid. Moves are made by swapping the empty position with any tile that is horizontally or vertically adjacent to it. For both puzzles, the representation given to the neural network uses one-hot encoding to specify which piece (tile or blank position) is in each position. For example, the dimension of the input to the neural network for the 15-puzzle would be $16 * 16 = 256$. The 15-puzzle has $16!/2 \approx 1.0 \times 10^{13}$ possible states, the 24-puzzle has $25!/2 \approx 7.7 \times 10^{24}$ possible states, the 35-puzzle has $36!/2 \approx 1.8 \times 10^{41}$ possible states, and the 48-puzzle has $49!/2 \approx 3.0 \times 10^{62}$ possible states. Any valid 15-puzzle configuration can be solved with at most 80 moves [24, 56]. The largest minimal number of moves required to solve the 24-puzzle, 35-puzzle, and 48-puzzle is not known.

Lights Out

Lights Out contains N^2 lights on an N by N board. The lights can either be on or off. The representation given to the DNN is a vector of size N^2. Each element is 1 if the corresponding light is on and 0 if the corresponding light is off.

Sokoban

The Sokoban environment we use is a 10 by 10 grid which contains four boxes that an agent needs to push on to four targets. In addition to the agent, boxes, and targets, Sokoban also contains walls. The representation given to the DNN contains four binary vectors of size 10^2 that represent the position on the agent, boxes, targets, and walls. Since boxes can only be pushed, not pulled, some actions are irreversible. For example, a box pushed into a corner can no longer be moved, creating a sampling problem because some states are unreachable when starting from the goal state. To address this, for each training state, we start from the goal state and allow boxes to be pulled instead of pushed.

2.2.3 Deep Approximate Value Iteration

Value iteration [85] is a dynamic programming algorithm [11, 13] that iteratively improves a cost-to-go function J. In traditional value iteration, J takes the form of a lookup table where the cost-to-go $J(s)$ is stored in a table for all possible states s. Value iteration loops through each state s and updates $J(s)$ until convergence:

$$J(s) \leftarrow min_a \sum_{s'} P^a(s, s')(g^a(s, s') + \gamma J(s')) \tag{2.1}$$

Here $P^a(s, s')$ is the *transition matrix* representing the probability of transitioning from state s to state s' by taking action a; $g^a(s, s')$ is the *cost* associated with transitioning from state s to s' by taking action a; and γ is the *discount factor*. In principle, this update equation can also be applied to the puzzles investigated in this paper. However, since these puzzles are deterministic, the transition function is a degenerate probability mass function for each action, simplifying Equation 2.1. Furthermore, because we wish to assign equal importance

Algorithm 1 Deep Approximate Value Iteration (DAVI)

Input:
    B: Batch size
    M: Training iterations
    K: Maximum number of scrambles
    C: How often to check for convergence
    ϵ: Error threshold
Output:
    θ: Trained neural network parameters

$\theta \leftarrow initialize\_parameters()$
$\theta_e \leftarrow \theta$

for $m = 1$ to M **do**
    $X \leftarrow get\_scrambled\_states(B, K)$
    for $x_i \in X$ **do**
        $y_i \leftarrow \min_a(g^a(x_i, A(x_i, a)) + j_{\theta_e}(A(x_i, a)))$
    $\theta, loss \leftarrow train(j_\theta, X, \mathbf{y})$
    if (M mod C = 0) and ($loss < \epsilon$) **then**
        $\theta_e \leftarrow \theta$
Return θ

to all costs, $\gamma = 1$. Therefore, we can update $J(s)$ using the following equation:

$$J'(s) = min_a(g^a(s, A(s, a)) + J(A(s, a))) \tag{2.2}$$

However, given the size of the state space of the Rubik's Cube, maintaining a table to store the cost-to-go of each state is not feasible. Therefore, we resort to *approximate value iteration* [13]. Instead of representing the cost-to-go function as a lookup table, we approximate the cost-to-go function using a parameterized function j_θ, with parameters θ. This function is implemented using a deep neural network (DNN). Therefore, we call the resulting algorithm *deep approximate value iteration* (DAVI).

In order to train the DNN, we have two sets of parameters: the parameters being trained θ,

and the parameters used to obtain an improved estimate of the cost-to-go θ_e. The output of $j_{\theta_e}(s)$ is set to 0 if s is the goal state. The DNN is trained to minimize the mean squared error between its estimation of the cost-to-go and the estimation obtained from Equation 2.1. Every C iterations, the algorithms checks if the error falls below a certain threshold ϵ; if so, then θ_e is set to θ. The entire DAVI process is shown in Algorithm **??**. While we tried updating θ_e at each iteration, we found that the performance saturated after a certain point and sometimes became unstable. Updating θ_e only after the error falls below a threshold ϵ yields better, more stable, performance.

Training Set State Distribution

In order for learning to occur, we must train on a state distribution that allows information to propagate from the goal state to all the other states seen during training. Our approach for achieving this is simple: each training state x_i is obtained by randomly scrambling the goal state k_i times, where k_i is uniformly distributed between 1 and K. During training, the cost-to-go function first improves for states that are only one move away from the goal state. The cost-to-go function then improves for states further away as the reward signal is propagated from the goal state to other states through the cost-to-go function. This can be seen as a simplified version of prioritized sweeping [78]. Working backward from the goal state is a well-known technique in AI, and has been used in means-end analysis [81] and STRIPS planning [31]. In future work we will explore different ways of generating a training set distribution.

2.2.4 Batch Weighted A* Search

A* search [42] is a heuristic-based search algorithm that finds a path between a starting node x_s and a goal node x_g. A* search maintains a set, OPEN, from which it iteratively removes

and expands the node with the lowest cost. The cost of each node x is determined by the function $f(x) = g(x) + h(x)$, where $g(x)$ is the path cost, which is the distance between x_s and x, and $h(x)$ is the heuristic function, which estimates the distance between x and x_g. After a node is expanded, that node is then added to another set, CLOSED, and its children that are not already in CLOSED are added to OPEN. The algorithm starts with only the starting node in OPEN and terminates when the goal node is removed from OPEN.

In this application, each node corresponds to a state of the Rubik's Cube and the goal node corresponds to the goal state shown in Figure 2.1. The path cost of every child of a node x is set to $g(x) + 1$. The path cost of x_s is 0. The heuristic function $h(x)$ is obtained from the learned cost-to-go function.

A variant of A* search, called weighted A* search [29], trades potentially longer solutions for potentially less memory usage. In this case, the function $f(x)$ is modified to $f(x) = \lambda\, g(x) + h(x)$, with the weight $\lambda \in [0, 1]$. While decreasing the weight λ will not necessarily decrease the number of nodes generated [108], in practice our experiments show that decreasing λ generally reduces the number of nodes generated and increases the length of the solutions found. In our implementation, if we encounter a node x that is already in CLOSED, and if x has a lower path cost than the node that is already in CLOSED, we remove that node from CLOSED and add x to OPEN.

The most time consuming aspect of the algorithm is the computation of the heuristic $h(x)$. The heuristic of many nodes can be computed in parallel across multiple GPUs by expanding the N best nodes from OPEN at each iteration. Our experiments show that larger values of N generally lead to shorter solutions and evaluate more nodes per second than searches with smaller N. We call the combination of A* search with a path-cost weight λ and a batch size of N *batch weighted A* search* (BWAS).

To satisfy the theoretical bounds on how much the length of a solution will deviate from

the length of an optimal solution, the heuristic used in weighted A* search must be admissible. That is to say that the heuristic can never overestimate the cost to reach the goal. While DeepCubeA's value function is not admissible, we empirically evaluate by how much DeepCubeA overestimates the cost to reach the goal. To do this, we obtain the length of a shortest path to the goal for 100,000 Rubik's cube states scrambled between 1 and 30 times. We then evaluate those same states with DeepCubeA's heuristic function j_θ. We find that DeepCubeA's heuristic function does not overestimate the cost to reach the goal 66.8% of the time and 97.4% of the time it does not overestimate it by more than 1. The average overestimation of the cost is 0.24.

2.2.5 Neural Network Architecture

The first two hidden layers of the DNNs have size 5,000 and 1,000 respectively, with full connectivity. This is then followed by 4 residual blocks [44], where each residual block has two hidden layers of size 1,000. Finally, the output layer consists of a single linear unit representing the cost-to-go estimate. We used batch normalization [47] and rectified linear activation functions [37] in all hidden layers. The DNN was trained with a batch size of 10,000, optimized with ADAM [51], and did not use any regularization. The maximum number of random moves applied to any training state K was set to 30. The error threshold ϵ was set to 0.05. We checked if the loss fell below the error threshold every 5,000 iterations. Training was carried out for 1 million iterations on two NVIDIA Titan V GPUs, with six other GPUs used in parallel for data generation. In total, the DNN saw 10 billion examples during training. Training was completed in 36 hours. When solving scrambled cubes from the test set, we use 4 NVIDIA X Pascal GPUs in parallel to compute the cost-to-go estimate. For the 15-puzzle, 24-puzzle, and Lights Out we set K to 500. For the 35-puzzle, 48-puzzle, and Sokoban, we set K to $1,000$. For the 24-puzzle, we use 6 residual blocks instead of 4.

2.2.6 Hyperparameter Selection for Batch Weighted A* Search

To choose the hyperparameters of BWAS, we did a grid search over λ and N. Values of λ were 0.0, 0.2, 0.4, 0.6, 0.8, and 1.0 while values of N were 1, 100, 1,000, and 10,000. The grid search was performed on 100 cubes that were generated separately from the test set. The GPU machines available to us had 64GB of RAM. Hyperparameter configurations that reached this limit were stopped early and thus not included in the results. Extended Figure **??** shows how λ and N affect performance in terms of average solution length, average number of nodes generated, average solve time, and average number of nodes generated per second. The figure shows that as λ increases, the average solution length decreases, however, the time to find a solution typically increases as well. The results also show that larger values of N lead to shorter solution lengths, but generally also require more time to find a solution; however, the number of nodes generated per second also increases due to the parallelism provided by the GPUs. Since $\lambda = 0.6$ and $N = 10,000$ resulted in the shortest solution lengths, we use these hyperparameters for the Rubik's Cube. For the 15-puzzle, 24-puzzle, and 35-puzzle we use $\lambda = 0.8$ and $N = 20,000$. For the 48-puzzle we use $\lambda = 0.6$ and $N = 20,000$. We increased N from $10,000$ to $20,000$ because we saw a reduction in solution length. For Lights Out we use $\lambda = 0.2$ and $N = 1,000$. For Sokoban we use $\lambda = 0.8$ and $N = 1$.

2.2.7 Pattern Databases

Pattern databases (PDBs)[27] are used to obtain a heuristic using lookup tables. Each lookup table contains the number of moves required to solve all possible combinations of a certain subgoal. For example, we can obtain a lookup table by enumerating all possible combinations of the edge cubelets on the Rubik's cube using a breadth-first search. These tables are then combined through either a max operator or a sum operator (depending on independence between subgoals)[55, 57] to produce a lower bound on the number of steps required to solve

the problem. Features from different pattern databases can be combined with neural networks for improved performance [90].

For the Rubik's Cube, we implemented the pattern database that Korf uses to find optimal solutions to the Rubik's Cube [55]. For the 15-puzzle, 24-puzzle, and 35-puzzle, we implement the pattern databases described in Felner et. al's work on additive pattern databases[30]. To the best of our knowledge, no one has created a pattern database for the 48-puzzle. We create our own by dividing the puzzle into 9 subgoals of size 5 and one subgoal of size 3. For all the n-puzzles, we also save the mirror of each PDB to improve the heuristic. Each PDB is saved in a "flat" representation; meaning that we use a database of size $(n+1)^k$ where n is the prefix of the n-puzzle and k is the size of the subgoal. Though this uses more memory, this is done to increase the speed of the table lookup operation[30].

2.2.8 Web Server

We have created a web server, located at http://deepcube.igb.uci.edu/, to allow anyone to use DeepCubeA to solve the Rubik's Cube. In the interest of speed, the hyperparameters for BWAS are set to $\lambda = 0.2$ and $N = 100$ in the server. The user can initiate a request to scramble the cube randomly, or use the keyboard keys to scramble the cube as he wishes. The user can then use the "solve" button to have DeepCubeA compute and post a solution, and execute the corresponding moves.

2.3 Results

To test the approach, we generate a test set of 1,000 states by randomly scrambling the goal state between 1,000 and 10,000 times. Additionally, we test the performance of DeepCubeA on the three known states that are the furthest possible distance away from the goal (26 moves)[87]. In order to assess how often DeepCubeA finds a shortest path to the goal, we need to compare our results to a shortest path solver. We can obtain a shortest path solver by using iterative deepening A* search (IDA*)[53] with an admissible heuristic computed from a pattern database. Initially, we used the pattern database described in Korf's work on finding optimal solutions to the Rubik's Cube[55]; however, this solver only solves a few states a day. Therefore, we use the optimal solver that was used to find the maximum of the minimum number of moves required to solve the Rubik's Cube from any given state (so-called "God's number") [88, 89]. This human-engineered solver relies on large pattern databases [27] (requiring 182GB of memory) and sophisticated knowledge of group theory to find a shortest path to the goal state. Comparisons between DeepCubeA and shortest path solvers are shown in Extended Table 2.1.

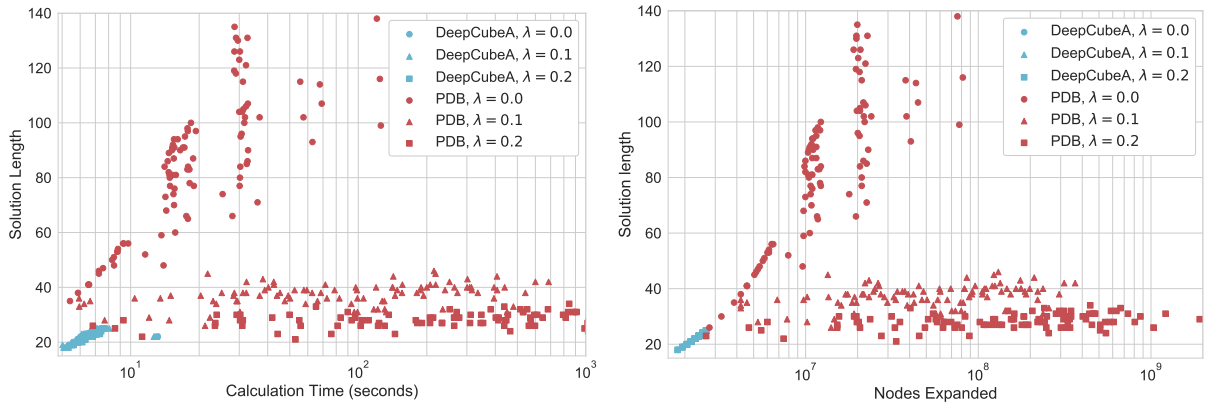
The DNN architecture consists of two fully connected hidden layers, followed by 4 residual blocks [44], followed by a linear output unit which represents the cost-to-go estimate. The hyperparameters of BWAS were chosen by doing a grid search over λ and N on data generated separately from the test set (see Methods for more details). When performing BWAS, the heuristic function is computed in parallel across four NVIDIA Titan V GPUs.

2.3.1 Performance

DeepCubeA finds a solution to 100% of all test states. DeepCubeA finds a shortest path to the goal 60.3% of the time. Aside from the optimal solutions, 36.4% of the solutions are

Puzzle	Solver	Len	% Opt	Nodes	Secs	Nodes/Sec
Rubik's Cube	PDBs[55]	-	-	-	-	-
	PDBs$^+$[88]	20.67	100.0%	2.05E+06	2.20	1.79E+06
	DeepCubeA	21.50	60.3%	6.62E+06	24.22	2.90E+05
Rubik's Cube$_h$	PDBs[55]	-	-	-	-	-
	PDBs$^+$[88]	26.00	100.0%	2.41E+10	13,561.27	1.78E+06
	DeepCubeA	26.00	100.0%	5.33E+06	18.77	2.96E+05
15-Puzzle	PDBs[30]	52.02	100.0%	3.22E+04	0.002	1.45E+07
	DeepCubeA	52.03	99.4%	3.85E+06	10.28	3.93E+05
15-Puzzle$_h$	PDBs[30]	80.00	100.0%	1.53E+07	0.997	1.56E+07
	DeepCubeA	82.82	17.65%	2.76E+07	69.36	3.98E+05
24-Puzzle	PDBs[30]	89.41	100.0%	8.19E+10	4,239.54	1.91E+07
	DeepCubeA	89.49	96.98%	6.44E+06	19.33	3.34E+05
35-Puzzle	PDBs[30]	-	-	-	-	-
	DeepCubeA	124.64	-	9.26E+06	28.45	3.25E+05
48-Puzzle	PDBs	-	-	-	-	-
	DeepCubeA	253.35	-	1.96E+07	74.46	2.63E+05
Lights Out	DeepCubeA	24.26	100.0%	1.14E+06	3.27	3.51E+05
Sokoban	LevinTS[83]	39.80	-	6.60E+03	-	-
	LevinTS[83] (*)	39.50	-	5.03E+03	-	-
	LAMA[83]	51.60	-	3.15E+03	-	-
	DeepCubeA	32.88	-	1.05E+03	2.35	5.60E+01

Extended Table 2.1: Comparison of DeepCubeA with optimal solvers based on pattern databases (PDBs) along the dimension of solution length, percentage of optimal solutions, number of nodes generated, time taken to solve the problem (in seconds), and number of nodes generated per second. The datasets with an "h" subscript represent the dataset containing the states that are furthest away from the goal state. PDBs$^+$ refers to Rokiki's pattern database combined with knowledge of group theory[88, 89]. For Sokoban, we compare nodes expanded instead of nodes generated to allow for direct comparison to previous work. DeepCubeA often finds a shortest path to the goal, sometimes doing so much faster than the optimal solvers.

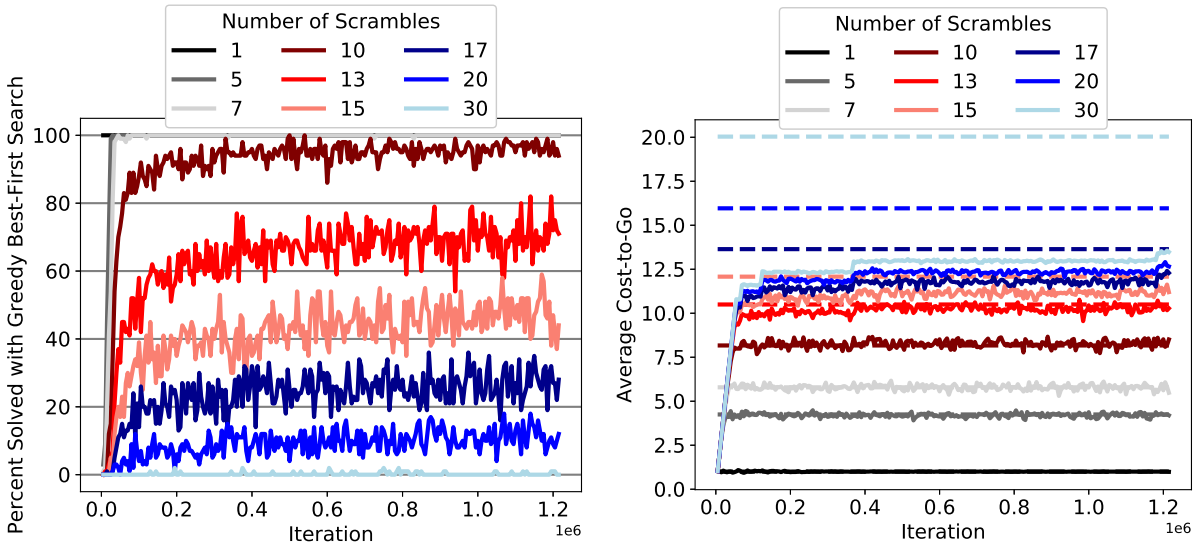


Extended Figure 2.1: The performance of DeepCubeA vs pattern databases (PDBs)[55] when solving the Rubik's cube with BWAS. N = 10,000 and λ is either 0.0, 0.1, or 0.2. Each dot represents the result on a single state. DeepCubeA is both faster and produces shorter solutions.

only two moves longer than the optimal solution, while the remaining 3.3% are four moves longer than the optimal solution. For the three states that are furthest away from the goal, DeepCubeA finds a shortest path to the goal for all three states (see Table 2.2). We would like to note that comparisons between DeepCubeA and shortest path solvers are not a direct comparison to pattern databases because, in this instance, pattern databases are being used to guarantee an optimal solution.

For a more direct comparison to pattern databases, we use Korf's pattern database heuristic for BWAS and compare the results to that of DeepCubeA. We perform BWAS with $N = 10,000$ and $\lambda = 0.0, 0.1, 0.2$. We compute the pattern database heuristic in parallel across 32 CPUs. Note that at $\lambda = 0.3$ BWAS runs out of memory when using pattern databases. Figure 2.1 shows that performing BWAS with DeepCubeA's learned heuristic consistently produces shorter solutions, expands fewer nodes, and is overall much faster than Korf's pattern database heuristic.

In addition to performance when doing BWAS, we also compare the memory footprint and speed of pattern databases and DeepCubeA. In terms of memory, for pattern databases, it is necessary to load lookup tables into memory. For DeepCubeA, it is necessary to load



Extended Figure 2.2: The performance of DeepCubeA. The plots show that DeepCubeA first learns how to solve cubes closer to the goal and then learns to solve increasingly difficult cubes. The dashed lines represent the true average cost-to-go.

the DNN into memory. Table 2.3 shows that DeepCubeA uses significantly less memory than pattern databases. In terms of speed, we measure how quickly pattern databases and DeepCubeA can compute a heuristic for a single state. We averaged performance across 1,000 states. Since DeepCubeA uses neural networks, which benefit from GPUs and batch processing, we measure the speed of DeepCubeA on a single CPU and a single GPU, both when doing sequential processing of the states and batch processing of the states. Table 2.4 shows that pattern databases almost always orders of magnitude faster than DeepCubeA. However, in the case of using a single GPU and batch processing, while pattern databases are still faster than DeepCubeA, DeepCubeA is on the same order of speed as pattern databases.

During training we monitor how well the DNN is able to solve the Rubik's cube using greedy best-first search; we also monitor how well the DNN is able to estimate the optimal cost-to-go function. How these performance metrics change as a function of training iteration is shown in Figure 2.2. The results show that DeepCubeA first learns to solve states closer to the goal before it learns to solve states further away from the goal. Cost-to-go estimation is less accurate for states further away from the goal; however, the cost-to-go function still

Puzzle	Solver	Len	% Opt	Nodes	Secs	Nodes/Sec
Rubik's Cube$_h$	PDBs[55]	-	-	-	-	-
	PDBs$^+$[88]	**26.00**	**100.0%**	2.41E+10	13,561.27	**1.78E+06**
	DeepCubeA	**26.00**	**100.0%**	**5.33E+06**	**18.77**	2.96E+05

Extended Table 2.2: Comparison of DeepCubeA with optimal solvers based on pattern databases (PDBs) along the dimension of solution length, percentage of optimal solutions, number of nodes generated, time taken to solve the problem (in seconds), and number of nodes generated per second on the Rubik's Cube states that are furthest away from the goal. PDBs$^+$ refers to Rokiki's optimal solver that uses pattern database combined with knowledge of group theory[88, 89]. DeepCubeA finds a shortest path to the goal and does so much faster than the optimal solver.

	RC	15-p	24-p	35-p	48-p	LightsOut	Sokoban
PDBs	4.67	8.51	1.86	0.64	4.86	-	-
PDBs$^+$	182.00	-	-	-		-	-
DeepCubeA	**0.06**	**0.06**	**0.08**	**0.08**	**0.10**	**0.05**	**0.06**

Extended Table 2.3: Comparison of the size (in GB) of the lookup tables for pattern databases (PDBs) and the size of the DNN used by DeepCubeA. The RC column corresponds to the Rubik's Cube and columns with a "-p" suffix correspond to n-puzzles. PDBs$^+$ refers to Rokiki's pattern database combined with knowledge of group theory[88, 89]. The table shows that DeepCubeA always uses memory that is orders of magnitude less than PDBs.

correctly orders the states according to difficulty. In addition, we found that DeepCubeA frequently used the conjugate patterns of moves of the form aba^{-1} in its solutions and often found symmetric solutions to symmetric states. An example of this is shown in Figure 2.3 (see Methods for more details).

2.3.2 Generalization to Other Combinatorial Puzzles

The Rubik's Cube is only one combinatorial puzzle among many others. To demonstrate the ability of DeepCubeA to generalize to other puzzles, we applied DeepCubeA to four popular sliding tile puzzles: the 15-puzzle, the 24-puzzle, 35-puzzle, and 48-puzzle. Additionally, we applied DeepCubeA to Lights Out and Sokoban. Sokoban posed a unique challenge for DeepCubeA because actions taken in its environment are not always reversible.

	RC	**15-p**	**24-p**	**35-p**	**48-p**	**LightsOut**	**Sokoban**
PDBs	**2E-06**	**1E-06**	**2E-06**	**3E-06**	**4E-06**	-	-
PDBs$^+$		-	-	-	-	-	-
DeepCubeA (GPU-B)	6E-06	6E-06	7E-06	8E-06	9E-06	**7E-06**	**6E-06**
DeepCubeA (GPU)	3E-03	3E-03	3E-03	2E-03	3E-03	4E-03	3E-03
DeepCubeA (CPU-B)	7E-04	6E-04	9E-04	9E-04	1E-03	1E-03	7E-04
DeepCubeA (CPU)	6E-03	6E-03	8E-03	8E-03	1E-02	2E-01	6E-03

Extended Table 2.4: Comparison of the speed (in seconds) of the lookup tables for pattern databases (PDBs) and the speed of the DNN used by DeepCubeA when computing the heuristic for a single state. Results were averaged over 1,000 states. DeepCubeA was timed on a single CPU and on a single GPU when doing sequential processing of the states and batch processing of the states (batch processing is denoted by the "-B" suffix). The RC column corresponds to the Rubik's Cube and columns with a "-p" suffix correspond to n-puzzles. PDBs$^+$ refers to Rokiki's pattern database combined with knowledge of group theory[88, 89]. The table shows that PDBs are always faster than DeepCubeA's heuristic; however, when computing DeepCubeA's heuristic on a GPU with batch processing, the speed is on the same order of magnitude as PDBs.

Sliding Tile Puzzles

The 15-puzzle has 1.0×10^{13} possible combinations, the 24-puzzle has 7.7×10^{24} possible combinations, the 35-puzzle has 1.8×10^{41} possible combinations, and the 48-puzzle has 3.0×10^{62} possible combinations. The objective is to move the puzzle into its goal configuration shown in Figure 2.1. For these sliding tile puzzles, we generated a test set of 500 states randomly scrambled between 1,000 and 10,000 times. The same DNN architecture and hyperparameters that are used for the Rubik's Cube are also used for the n-puzzles with the exception of the addition of two more residual layers. We implemented an optimal solver using additive pattern databases[30]. DeepCubeA not only solved every test puzzle, but also found a shortest path to the goal 99.4% of the time for the 15-puzzle and 96.98% of the time for the 24-puzzle. We also test on the 17 states that are furthest away from the goal for the 15-puzzle (these states are not known for the 24-puzzle)[52]. Solutions produced by DeepCubeA are, on average, 2.8 moves longer than the length of a shortest path and DeepCubeA finds a shortest path to the goal for 17.6% of these states. For the 24-puzzle, on average, pattern databases take 4,239 seconds and DeepCubeA takes 19.3 seconds, over 200

Puzzle	Solver	Len	% Opt	Nodes	Secs	Nodes/Sec
24-Puzzle	PDBs[30]	**89.41**	**100.0%**	8.19E+10	4,239.54	**1.91E+07**
	DeepCubeA	89.49	96.98%	**6.44E+06**	**19.33**	3.34E+05
35-Puzzle	PDBs[30]	-	-	-	-	-
	DeepCubeA	**124.64**	-	**9.26E+06**	**28.45**	**3.25E+05**

Extended Table 2.5: Comparison of DeepCubeA with optimal solvers based on pattern databases (PDBs) along the dimension of solution length, percentage of optimal solutions, number of nodes generated, time taken to solve the problem (in seconds), and number of nodes generated per second for the 24-puzzle and 35-puzzle. For the 24-puzzle, DeepCubeA finds a shortest path to the goal the overwhelming majority of the time and does so much faster than PDBs. For the 35-puzzle, no tractable optimal solver exists.

times faster. Moreover, in the worst case we observed that the longest time needed to solve the 24-puzzle is 5 days for pattern databases and two minutes for DeepCubeA. The average solution length for the 124.76 for the 35-puzzle and 253.53 for the 48-puzzle; however, we do not know how many of them are optimal due to the optimal solver being intractably slow for the 35-puzzle and 48-puzzle. The performance of DeepCubeA on the 24-puzzle and 35-puzzle are summarized in Table 2.5.

The shortest path solver for the 35-puzzle and 48-puzzle was intractably slow; however, we compare DeepCubeA to pattern databases method using BWAS. The results show that, compared to pattern databases, DeepCubeA produces shorter solutions, generates fewer nodes, and is, on average, faster for two out the three assignments of λ.

Lights Out

Lights Out is a grid-based puzzle consisting of an N by N board of lights that may be either active or inactive. The goal is to convert all active lights to inactive from a random starting position as seen in Figure 2.1. Pressing any light in the grid will switch the state of that light and its immediate horizontal and vertical neighbors. At any given state, a player may click on any of the N^2 lights. However, one difference of Lights Out compared to the other

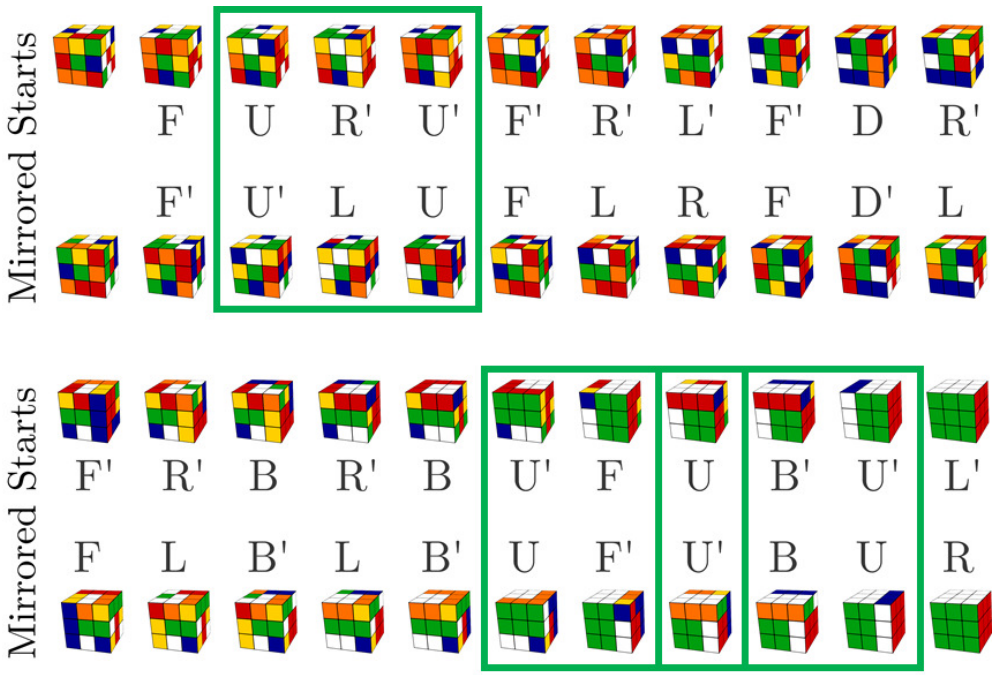
environments is that the moves are commutative. We tested DeepCubeA on the 7 by 7 Lights Out puzzle. A theorem by Scherphuis[92] shows that, for 7 by 7 Lights Out, any solution that does not contain any duplicate moves is the optimal solution. Using this theorem, we found that DeepCubeA found a shortest path to the goal for all test cases.

Sokoban

Sokoban [28] is a planning problem that requires an agent to move boxes onto target locations. Note that training states are generated by pulling instead of pushing (see Methods for more details). Boxes can only be pushed, not pulled. To test our method on Sokoban, we train on the 900,000 training examples and test on the 1,000 testing examples used by previous research on single-agent policy tree search applied to Sokoban[8]. DeepCubeA successfully solves 100% of all test examples. We compare solution length and number of nodes expanded to this same previous research[83]. Although the goals of the aforementioned paper are slightly different; DeepCubeA finds shorter paths than previously reported methods and also expands, at least, 3 times fewer nodes (see Extended Table 2.1).

2.3.3 Conjugate Patterns and Symmetric States

Since the operation of the Rubik's Cube is deeply rooted in group theory, solutions produced by an algorithm that learns how to solve this puzzle should contain group theory properties. In particular, conjugate patterns of moves of the form aba^{-1} should appear relatively often when solving the Rubik's Cube. These patterns are necessary for manipulating specific cubelets while not affecting the position of other cubelets. Using a sliding window, we gathered all triplets in all solutions to the Rubik's Cube and found that aba^{-1} accounted for 13.11% of all triplets (significantly above random) while aba accounted for 8.86%, aab accounted for 4.96%, and abb accounted for 4.92%. To put this into perspective, for the optimal solver, aba^{-1}, aba,



Extended Figure 2.3: An example of symmetric solutions that DeepCubeA finds to symmetric states. Conjugate triplets are indicated by the green boxes. Note that the last two conjugate triplets are overlapping.

aab, and *abb* accounted for 9.15%, 9.63%, 5.30%, and 5.35% of all triplets, respectively.

In addition, we found that DeepCubeA often found symmetric solutions to symmetric states. One can produce a symmetric state for the Rubik's Cube by mirroring the cube from left to right, as shown in Figure 2.3. The optimal solutions for two symmetric states have the same length; furthermore, one can use the mirrored solution of one state to solve the other. To see if this property was present in DeepCubeA, we created mirrored states of the Rubik's Cube test set and solved them using DeepCubeA. The results showed that 58.30% of the solutions to the mirrored test set were symmetric to those of the original test set. Of the solutions that were not symmetric, 69.54% had the same solution length as the solution length obtained on the original test set. To put this into perspective, for the handmade optimal solver, the results showed that 74.50% of the solutions to the mirrored test set were symmetric to those of the original test set.

2.4 Discussion

DeepCubeA is able to solve planning problems with large state spaces and few goal states by learning a cost-to-go function, parameterized by a deep neural network, which is then used as a heuristic function for weighted A* search. The cost-to-go function is learned by using approximate value iteration on states generated by starting from the goal state and taking moves in reverse. DeepCubeA's success on solving the seven problems investigated in this paper suggests that DeepCubeA can be readily applied to new problems given an input representation, a state transition model, a goal state, and a reverse state transition model.

For the puzzles investigated in this paper, an effective cost-to-go function could be trained on states obtained by randomly taking moves in reverse; however, there are other puzzles, such as peg solitaire, in which randomly taking moves in reverse is most likely to generate states relatively close to the goal state while very rarely generating states relatively far from the goal state. This could cause the cost-to-go function to be ineffective when presented with a starting state relatively far from the goal state. Future research may be able to improve upon DeepCubeA by starting from the goal state and using the learned cost-to-go function to search for difficult states.

With a heuristic function that never overestimates the cost of a shortest path (i.e. an admissible heuristic function), weighed A* search comes with known bounds on how much the length of a solution can deviate from the length of an optimal solution. While DeepCubeA's heuristic function is not guaranteed to be admissible, and thus does not satisfy the requirement for these theoretical bounds, DeepCubeA nevertheless finds a shortest path to the goal in the majority of cases (see Methods for more details).

The generality of the core algorithm suggests that it may have applications beyond combinatorial puzzles, as problems with large state spaces and few goal states are not rare in planning, robotics, and the natural sciences.

Chapter 3

Pipeline PSRO: A Scalable Approach for Finding Approximate Nash Equilibria in Large Games

3.1 Introduction

A long-standing goal in artificial intelligence and algorithmic game theory has been to develop a general algorithm which is capable of finding approximate Nash equilibria in large imperfect-information two-player zero-sum games. AlphaStar [107] and OpenAI Five [12] were able to demonstrate that variants of self-play reinforcement learning are capable of achieving expert-level performance in large imperfect-information video games. However, these methods are not principled from a game-theoretic point of view and are not guaranteed to converge to an approximate Nash equilibrium. Policy Space Response Oracles (PSRO) [60] is a game-theoretic reinforcement learning algorithm based on the Double Oracle algorithm and is guaranteed to converge to an approximate Nash equilibrium.

PSRO is a general, principled method for finding approximate Nash equilibria, but it may not scale to large games because it is a sequential algorithm that uses reinforcement learning to train a full best response at every iteration. Two existing approaches parallelize PSRO: Deep Cognitive Hierarchies (DCH) [60] and Rectified PSRO [9], but both have counterexamples on which they fail to converge to an approximate Nash equilibrium, and as we show in our experiments, neither reliably converges in random normal form games.

Although DCH approximates PSRO, it has two main limitations. First, DCH needs the same number of parallel workers as the number of best response iterations that PSRO takes. For large games, this requires a very large number of parallel reinforcement learning workers. This also requires guessing how many iterations the algorithm will need before training starts. Second, DCH keeps training policies even after they have plateaued. This introduces variance by allowing the best responses of early levels to change each iteration, causing a ripple effect of instability. We find that, in random normal form games, DCH rarely converges to an approximate Nash equilibrium even with a large number of parallel workers, unless their learning rate is carefully annealed.

Rectified PSRO is a variant of PSRO in which each learner only plays against other learners that it already beats. We prove by counterexample that Rectified PSRO is not guaranteed to converge to a Nash equilibrium. We also show that Rectified PSRO rarely converges in random normal form games.

In this paper we introduce Pipeline PSRO (P2SRO), the first scalable PSRO-based method for finding approximate Nash equilibria in large zero-sum imperfect-information games. P2SRO is able to scale up PSRO with convergence guarantees by maintaining a hierarchical pipeline of reinforcement learning workers, each training against the policies generated by lower levels in the hierarchy. P2SRO has two classes of policies: fixed and active. Active policies are trained in parallel while fixed policies are not trained anymore. Each parallel reinforcement learning worker trains an active policy in a hierarchical pipeline, training against the meta

Nash equilibrium of both the fixed policies and the active policies on lower levels in the pipeline. Once the performance increase of the lowest-level active worker in the pipeline does not improve past a given threshold in a given amount of time, the policy becomes fixed, and a new active policy is added to the pipeline. P2SRO is guaranteed to converge to an approximate Nash equilibrium. Unlike Rectified PSRO and DCH, P2SRO converges to an approximate Nash equilibrium across a variety of imperfect information games such as Leduc poker and random normal form games.

We also introduce an open-source environment for Barrage Stratego, a variant of Stratego. Barrage Stratego is a large two-player zero sum imperfect information board game with an approximate game tree complexity of 10^{50}. We demonstrate that P2SRO is able to achieve state-of-the-art performance on Barrage Stratego, beating all existing bots.

To summarize, in this paper we provide the following contributions:

- We develop a method for parallelizing PSRO which is guaranteed to converge to an approximate Nash equilibrium, and show that this method outperforms existing methods on random normal form games and Leduc poker.
- We present theory analyzing the performance of PSRO as well as a counterexample where Rectified PSRO does not converge to an approximate Nash equilibrium.
- We introduce an open-source environment for Stratego and Barrage Stratego, and demonstrate state-of-the-art performance of P2SRO on Barrage Stratego.

3.2 Background and Related Work

A two-player normal-form game is a tuple (Π, U), where $\Pi = (\Pi_1, \Pi_2)$ is the set of policies (or strategies), one for each player, and $U : \Pi \to \mathbb{R}^2$ is a payoff table of utilities for each joint

policy played by all players. For the game to be zero-sum, for any pair of policies $\pi \in \Pi$, the payoff $u_i(\pi)$ to player i must be the negative of the payoff $u_{-i}(\pi)$ to the other player, denoted $-i$. Players try to maximize their own expected utility by sampling from a distribution over the policies $\sigma_i \in \Sigma_i = \Delta(\Pi_i)$. The set of best responses to a mixed policy σ_i is defined as the set of policies that maximally exploit the mixed policy: $(\sigma_i) = \arg\min_{\sigma'_{-i} \in \Sigma_{-i}} u_i(\sigma'_{-i}, \sigma_i)$, where $u_i(\sigma) = \mathrm{E}_{\pi \sim \sigma}[u_i(\pi)]$. The exploitability of a pair of mixed policies σ is defined as: $\textsc{Exploitability}(\sigma) = \frac{1}{2}(u_2(\sigma_1, (\sigma_1)) + u_1((\sigma_2), \sigma_2)) \geqslant 0$. A pair of mixed policies $\sigma = (\sigma_1, \sigma_2)$ is a Nash equilibrium if $\textsc{Exploitability}(\sigma) = 0$. An approximate Nash equilibrium at a given level of precision ϵ is a pair of mixed policies σ such that $\textsc{Exploitability}(\sigma) \leqslant \epsilon$ [96].

In small normal-form games, Nash equilibria can be found via linear programming [82]. However, this quickly becomes infeasible when the size of the game increases. In large normal-form games, no-regret algorithms such as fictitious play, replicator dynamics, and regret matching can asymptotically find approximate Nash equilibria [34, 105, 110]. Extensive form games extend normal-form games and allow for sequences of actions. Examples of perfect-information extensive form games include chess and Go, and examples of imperfect-information extensive form games include poker and Stratego.

In perfect information extensive-form games, algorithms based on minimax tree search have had success on games such as checkers, chess and Go [97]. Extensive-form fictitious play (XFP) [46] and counterfactual regret minimization (CFR) [110] extend fictitious play and regret matching, respectively, to extensive form games. In large imperfect information games such as heads up no-limit Texas Hold 'em, counterfactual regret minimization has been used on an abstracted version of the game to beat top humans [20]. However, this is not a general method because finding abstractions requires expert domain knowledge and cannot be easily done for different games. For very large imperfect information games such as Barrage Stratego, it is not clear how to use abstractions and CFR. Deep CFR [22] is a general method that trains a neural network on a buffer of counterfactual values. However, Deep CFR uses

external sampling, which may be impractical for games with a large branching factor such as Stratego and Barrage Stratego. DREAM [102] and ARMAC [39] are model-free regret-based deep learning approaches. Current Barrage Stratego bots are based on imperfect information tree search and are unable to beat even intermediate-level human players [91, 50].

Recently, deep reinforcement learning has proven effective on high-dimensional sequential decision making problems such as Atari games and robotics [65]. AlphaStar [107] beat top humans at Starcraft using self-play and population-based reinforcement learning. Similarly, OpenAI Five [12] beat top humans at Dota using self play reinforcement learning. Similar population-based methods have achieved human-level performance on Capture the Flag [48]. However, these algorithms are not guaranteed to converge to an approximate Nash equilibrium. Neural Fictitious Self Play (NFSP) [45] approximates extensive-form fictitious play by progressively training a best response against an average of all past policies using reinforcement learning. The average policy is represented by a neural network and is trained via supervised learning using a replay buffer of past best response actions. This replay buffer may become prohibitively large in complex games.

3.2.1 Policy Space Response Oracles

The Double Oracle algorithm [77] is an algorithm for finding a Nash equilibrium in normal form games. The algorithm works by keeping a population of policies $\Pi^t \subset \Pi$ at time t. Each iteration a Nash equilibrium $\sigma^{*,t}$ is computed for the game restricted to policies in Π^t. Then, a best response to this Nash equilibrium for each player ($\sigma^{*,t}_{-i}$) is computed and added to the population $\Pi^{t+1}_i = \Pi^t_i \cup \{(\sigma^{*,t}_{-i})\}$ for $i \in \{1, 2\}$.

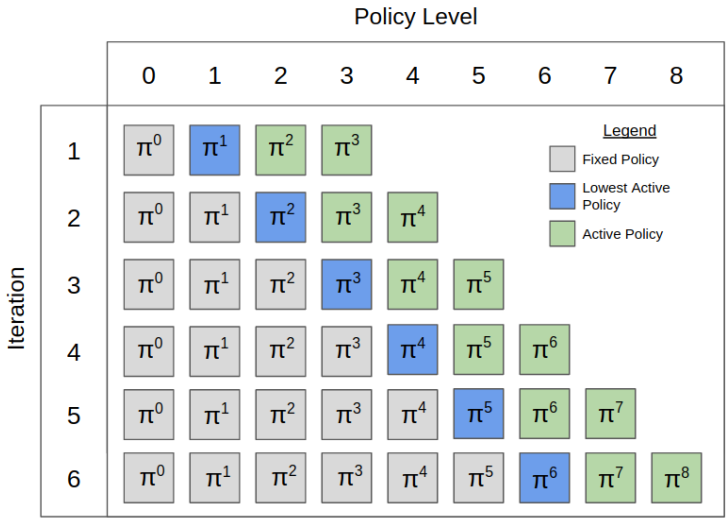
Policy Space Response Oracles (PSRO) approximates the Double Oracle algorithm. The meta Nash equilibrium is computed on the empirical game matrix U^{Π}, given by having each policy in the population Π play each other policy and tracking average utility in a payoff

matrix. In each iteration, an approximate best response to the current meta Nash equilibrium over the policies is computed via any reinforcement learning algorithm. In this work we use a discrete-action version of Soft Actor Critic (SAC), described in Section 3.3.1.

One issue with PSRO is that it is based on a normal-form algorithm, and the number of pure strategies in a normal form representation of an extensive-form game is exponential in the number of information sets. In practice, however, PSRO is able to achieve good performance in large games, possibly because large sections of the game tree correspond to weak actions, so only a subset of pure strategies need be enumerated for satisfactory performance. Another issue with PSRO is that it is a sequential algorithm, requiring a full best response computation in every iteration. This paper addresses the latter problem by parallelizing PSRO while maintaining the same convergence guarantees.

DCH [60] parallelizes PSRO by training multiple reinforcement learning agents, each against the meta Nash equilibrium of agents below it in the hierarchy. A problem with DCH is that one needs to set the number of workers equal to the number of policies in the final population beforehand. For large games such as Barrage Stratego, this might require hundreds of parallel workers. Also, in practice, DCH fails to converge in small random normal form games even with an exact best-response oracle and a learning rate of 1, because early levels may change their best response occasionally due to randomness in estimation of the meta Nash equilibrium. In our experiments and in the DCH experiments in Lanctot et al. [60], DCH is unable to achieve low exploitability on Leduc poker.

Another existing parallel PSRO algorithm is Rectified PSRO [9]. Rectified PSRO assigns each learner to play against the policies that it currently beats. However, we prove that Rectified PSRO does not converge to a Nash equilibrium in all symmetric zero-sum games. In our experiments, Rectified PSRO rarely converges to an approximate Nash equilibrium in random normal form games.



Extended Figure 3.1: Pipeline PSRO. The lowest-level active policy π^j (blue) plays against the meta Nash equilibrium $\sigma^{*,j}$ of the lower-level fixed policies in Π^f (gray). Each additional active policy (green) plays against the meta Nash equilibrium of the fixed and training policies in levels below it. Once the lowest active policy plateaus, it becomes fixed, a new active policy is added, and the next active policy becomes the lowest active policy. In the first iteration, the fixed population consists of a single random policy.

3.3 Pipeline Policy Space Response Oracles (P2SRO)

Pipeline PSRO (P2SRO; Algorithm 2) is able to scale up PSRO with convergence guarantees by maintaining a hierarchical pipeline of reinforcement learning policies, each training against the policies in the lower levels of the hierarchy (Figure 3.1). P2SRO has two classes of policies: fixed and active. The set of fixed policies are denoted by Π^f and do not train anymore, but remain in the fixed population. The parallel reinforcement learning workers train the active policies, denoted Π^a in a hierarchical pipeline, training against the meta Nash equilibrium distribution of both the fixed policies and the active policies in levels below them in the pipeline. The entire population Π consists of the union of Π^f and Π^a. For each policy π_i^j in the active policies Π_i^a, to compute the distribution of policies to train against, a meta Nash equilibrium $\sigma_{-i}^{*,j}$ is periodically computed on policies lower than π_i^j: $\Pi_{-i}^f \cup \{\pi_{-i}^k \in \Pi_{-i}^a | k < j\}$ and π_i^j trains against this distribution.

The performance of a policy π^j is given by the average performance during training $\mathbb{E}_{\pi_1 \sim \sigma_1^{*,j}}[u_2(\pi_1, \pi_2^j)] + \mathbb{E}_{\pi_2 \sim \sigma_2^{*,j}}[u_1(\pi_1^j, \pi_2)]$ against the meta Nash equilibrium distribution $\sigma^{*,j}$. Once the performance of the lowest-level active policy π^j in the pipeline does not improve past a given threshold in a given amount of time, we say that the policy's performance plateaus, and π^j becomes fixed and is added to the fixed population Π^f. Once π^j is added to the fixed population Π^f, then π^{j+1} becomes the new lowest active policy. A new policy is initialized and added as the highest-level policy in the active policies Π^a. Because the lowest-level policy only trains against the previous fixed policies Π^f, P2SRO maintains the same convergence guarantees as PSRO. Unlike PSRO, however, each policy in the pipeline above the lowest-level policy is able to get a head start by pre-training against the moving target of the meta Nash equilibrium of the policies below it. Unlike Rectified PSRO and DCH, P2SRO converges to an approximate Nash equilibrium across a variety of imperfect information games such as Leduc Poker and random normal form games.

In our experiments we model the non-symmetric games of Leduc poker and Barrage Stratego as symmetric games by training one policy that can observe which player it is at the start of the game and play as either the first or the second player. We find that in practice it is more efficient to only train one population than to train two different populations, especially in larger games, such as Barrage Stratego.

3.3.1 Implementation Details

For the meta Nash equilibrium solver we use fictitious play [34]. Fictitious play is a simple method for finding an approximate Nash equilibrium in normal form games. Every iteration, a best response to the average strategy of the population is added to the population. The average strategy converges to an approximate Nash equilibrium. For the approximate best response oracle, we use a discrete version of Soft Actor Critic (SAC) [40, 26]. We modify the

Algorithm 2 Pipeline Policy-Space Response Oracles (P2SRO)

Input: Initial policy sets for all players Π^f
Compute expected utilities for empirical payoff matrix U^{Π} for each joint $\pi \in \Pi$
Compute meta-Nash equilibrium $\sigma^{*,j}$ over fixed policies (Π^f)
for many episodes **do**
    for all $\pi^j \in \Pi^a$ in parallel **do**
        for player $i \in \{1, 2\}$ **do**
            Sample $\pi_{-i} \sim \sigma_{-i}^{*,j}$
            Train π_i^j against π_{-i}
        if π^j plateaus and π^j is the lowest active policy **then**
            $\Pi^f = \Pi^f \cup \{\pi^j\}$
            Initialize new active policy at a higher level than all existing active policies
            Compute missing entries in U^{Π} from Π
            Compute meta Nash equilibrium for each active policy
        Periodically compute meta Nash equilibrium for each active policy
Output current meta Nash equilibrium on whole population σ^*

version used in RLlib [66, 79] to account for discrete actions.

3.3.2 Analysis

PSRO is guaranteed to converge to an approximate Nash equilibrium and doesn't need a large replay buffer, unlike NFSP and Deep CFR. In the worst case, all policies in the original game must be added before PSRO reaches an approximate Nash equilibrium. Empirically, on random normal form games, PSRO performs better than selecting pure strategies at random without replacement. This implies that in each iteration, PSRO is more likely than random to add a pure strategy that is part of the support of the Nash equilibrium of the full game, suggesting the conjecture that PSRO has faster convergence rate than random strategy selection. The following theorem indirectly supports this conjecture.

Theorem 1. *Let σ be a Nash equilibrium of a symmetric normal form game (Π, U) and let Π^e be the set of pure strategies in its support. Let $\Pi' \subset \Pi$ be a population that does not cover $\Pi^e \nsubseteq \Pi'$, and let σ' be the meta Nash equilibrium of the original game restricted to strategies in Π'. Then there exists a pure strategy $\pi \in \Pi^e \backslash \Pi'$ such that π does not lose to σ'.*

Proof. σ' is a meta Nash equilibrium, implying $\sigma'^{\intercal} G \sigma' = 0$, where G is the payoff matrix for the row player. In fact, each policy π in the support Π'^{e} of σ' has $1_{\pi}^{\intercal} G \sigma' = 0$, where 1_{π} is the one-hot encoding of π in Π.

Consider the sets $\Pi^{+} = \{\pi : \sigma(\pi) > \sigma'(\pi)\} = \Pi^{e} \backslash \Pi'$ and $\Pi^{-} = \{\pi : \sigma(\pi) < \sigma'(\pi)\} \subseteq \Pi'^{e}$. Note the assumption that Π^{+} is not empty. If each $\pi \in \Pi^{+}$ had $1_{\pi}^{\intercal} G \sigma' < 0$, we would have

$$\begin{aligned} \sigma^{\intercal} G \sigma' = (\sigma - \sigma')^{\intercal} G \sigma' &= \sum_{\pi \in \Pi^{+}} (\sigma(\pi) - \sigma'(\pi)) 1_{\pi}^{\intercal} G \sigma' + \sum_{\pi \in \Pi^{-}} (\sigma(\pi) - \sigma'(\pi)) 1_{\pi}^{\intercal} G \sigma' \\ &= \sum_{\pi \in \Pi^{+}} (\sigma(\pi) - \sigma'(\pi)) 1_{\pi}^{\intercal} G \sigma' < 0, \end{aligned}$$

in contradiction to σ being a Nash equilibrium. We conclude that there must exist $\pi \in \Pi^{+}$ with $1_{\pi}^{\intercal} G \sigma' \geqslant 0$. □

Ideally, PSRO would be able to add a member of $\Pi^{e} \backslash \Pi'$ to the current population Π' at each iteration. However, the best response to the current meta Nash equilibrium σ' is generally not a member of Π^{e}. Theorem 1 shows that for an *approximate* best response algorithm with a weaker guarantee of not losing to σ', it is possible that a member of $\Pi^{e} \backslash \Pi'$ is added at each iteration.

Even assuming that a policy in the Nash equilibrium support is added at each iteration, the convergence of PSRO to an approximate Nash equilibrium can be slow because each policy is trained sequentially by a reinforcement learning algorithm. DCH, Rectified PSRO, and P2SRO are methods of speeding up PSRO through parallelization. In large games, many of the basic skills (such as extracting features from the board) may need to be relearned when starting each iteration from scratch. DCH and P2SRO are able to speed up PSRO by pre-training each level on the moving target of the meta Nash equilibrium of lower-level policies before those policies converge. This speedup would be linear with the number of parallel workers if each policy could train on the fixed final meta Nash equilibrium of the

policies below it. Since it trains instead on a moving target, we expect the speedup to be sub-linear in the number of workers.

DCH is an approximation of PSRO that is not guaranteed to converge to an approximate Nash equilibrium if the number of levels is not equal to the number of pure strategies in the game, and is in fact guaranteed *not* to converge to an approximate Nash equilibrium if the number of levels cannot support it.

Another parallel PSRO algorithm, Rectified PSRO, is not guaranteed to converge to an approximate Nash equilibrium.

Proposition 1. *Rectified PSRO with an oracle best response does not converge to a Nash equilibrium in all symmetric two-player, zero-sum normal form games.*

Proof. Consider the following symmetric two-player zero-sum normal form game:

$$\begin{bmatrix} 0 & -1 & 1 & -\frac{2}{5} \\ 1 & 0 & -1 & -\frac{2}{5} \\ -1 & 1 & 0 & -\frac{2}{5} \\ \frac{2}{5} & \frac{2}{5} & \frac{2}{5} & 0 \end{bmatrix}$$

This game is based on Rock–Paper–Scissors, with an extra strategy added that beats all other strategies and is the pure Nash equilibrium of the game. Suppose the population of Rectified PSRO starts as the pure Rock strategy.

- Iteration 1: Rock ties with itself, so a best response to Rock (Paper) is added to the population.
- Iteration 2: The meta Nash equilibrium over Rock and Paper has all mass on Paper. The new strategy that gets added is the best response to Paper (Scissors).

- Iteration 3: The meta Nash equilibrium over Rock, Paper, and Scissors equally weights each of them. Now, for each of the three strategies, Rectified PSRO adds a best response to the meta-Nash-weighted combination of strategies that it beats or ties. Since Rock beats or ties Rock and Scissors, a best response to a $50-50$ combination of Rock and Scissors is Rock, with an expected utility of $\frac{1}{2}$. Similarly, for Paper, since Paper beats or ties Paper and Rock, a best response to a $50-50$ combination of Paper and Rock is Paper. For Scissors, the best response for an equal mix of Scissors and Paper is Scissors. So in this iteration no strategy is added to the population and the algorithm terminates.

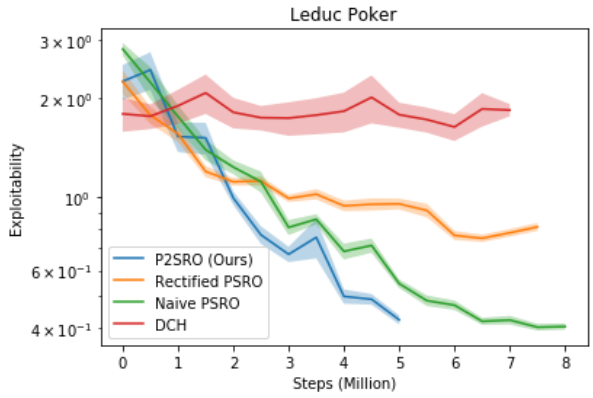
We see that the algorithm terminates without expanding the fourth strategy. The meta Nash equilibrium of the first three strategies that Rectified PSRO finds are not a Nash equilibrium of the full game, and are exploited by the fourth strategy, which is guaranteed to get a utility of $\frac{2}{5}$ against any mixture of them. □

The pattern of the counterexample presented here is possible to occur in large games, which suggests that Rectified PSRO may not be an effective algorithm for finding an approximate Nash equilibrium in large games. Prior work has found that Rectified PSRO does not converge to an approximate Nash equilibrium in Kuhn Poker [80].

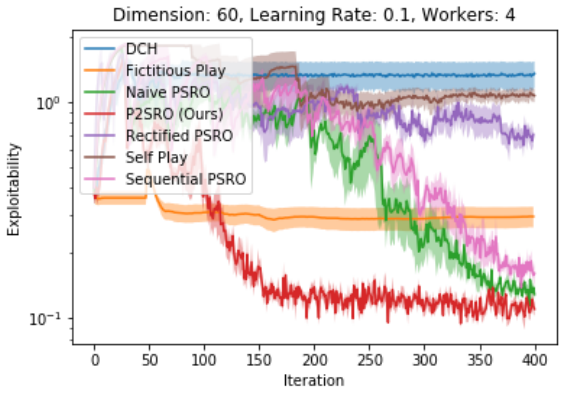
Proposition 2. *P2SRO with an oracle best response converges to a Nash equilibrium in all two-player, zero-sum normal form games.*

Proof. Since only the lowest active policy can be submitted to the fixed policies, this policy is an oracle best response to the meta Nash distribution of the fixed policies, making P2SRO with an oracle best response equivalent to the Double Oracle algorithm. □

Unlike DCH which becomes unstable when early levels change, P2SRO is able to avoid this problem because early levels become fixed once they plateau. While DCH only approximates



(a) Leduc poker



(b) Random Symmetric Normal Form Games

Extended Figure 3.2: Exploitability of Algorithms on Leduc poker and Random Symmetric Normal Form Games

PSRO, P2SRO has equivalent guarantees to PSRO because the lowest active policy always trains against a fixed meta Nash equilibrium before plateauing and becoming fixed itself. This fixed meta Nash distribution that it trains against is in principle the same as the one that PSRO would train against. The only difference between P2SRO and PSRO is that the extra workers in P2SRO are able to get a head-start by pre-training on lower level policies while those are still training. Therefore, P2SRO inherits the convergence guarantees from PSRO while scaling up when multiple processors are available.

3.4 Results

We compare P2SRO with DCH, Rectified PSRO, and a naive way of parallelizing PSRO that we term Naive PSRO. Naive PSRO is a way of parallelizing PSRO where each additional worker trains against the same meta Nash equilibrium of the fixed policies. Naive PSRO is beneficial when randomness in the reinforcement learning algorithm leads to a diversity of trained policies, and in our experiments it performs only slightly better than PSRO. Additionally, in random normal form game experiments, we include the original, non-parallel PSRO algorithm, termed sequential PSRO, and non-parallelized self-play, where a single

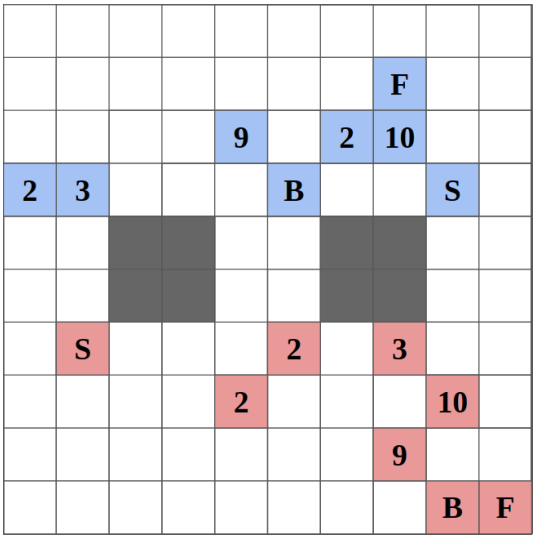
policy trains against the latest policy in the population.

We find that DCH fails to reliably converge to an approximate Nash equilibrium across random symmetric normal form games and small poker games. We believe this is because early levels can randomly change even after they have plateaued, causing instability in higher levels. In our experiments, we analyze the behavior of DCH with a learning rate of 1 in random normal form games. We hypothesized that DCH with a learning rate of 1 would be equivalent to the double oracle algorithm and converge to an approximate Nash. However, we found that the best response to a fixed set of lower levels can be different in each iteration due to randomness in calculating a meta Nash equilibrium. This causes a ripple effect of instability through the higher levels. We find that DCH almost never converges to an approximate Nash equilibrium in random normal form games.

Although not introduced in the original paper, we find that DCH converges to an approximate Nash equilibrium with an annealed learning rate. An annealed learning rate allows early levels to not continually change, so the variance of all of the levels can tend to zero. Reinforcement learning algorithms have been found to empirically converge to approximate Nash equilibria with annealed learning rates [100, 16]. We find that DCH with an annealed learning rate does converge to an approximate Nash equilibrium, but it can converge slowly depending on the rate of annealing. Furthermore, annealing the learning rate can be difficult to tune with deep reinforcement learning, and can slow down training considerably.

3.4.1 Random Symmetric Normal Form Games

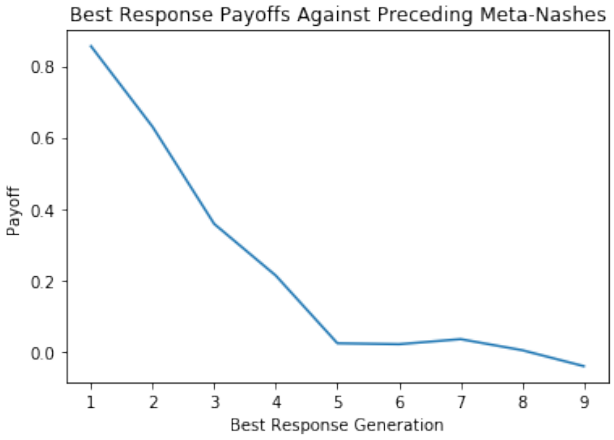
For each experiment, we generate a random symmetric zero-sum normal form game of dimension n by generating a random antisymmetric matrix P. Each element in the upper triangle is distributed uniformly: $\forall i < j \leqslant n$, $a_{i,j} \sim \textsc{Uniform}(-1, 1)$. Every element in the lower triangle is set to be the negative of its diagonal counterpart: $\forall j < i \leqslant n$, $a_{i,j} = -a_{j,i}$.



Extended Figure 3.3: Valid Barrage Stratego Setup (note that the piece values are not visible to the other player)

The diagonal elements are equal to zero: $a_{i,i} = 0$. The matrix defines the utility of two pure strategies to the row player. A strategy $\pi \in \Delta^n$ is a distribution over the n pure strategies of the game given by the rows (or equivalently, columns) of the matrix. In these experiments we can easily compute an exact best response to a strategy and do not use reinforcement learning to update each strategy. Instead, as a strategy π "trains" against another strategy $\hat{\pi}$, it is updated by a learning rate r multiplied by the best response to that strategy: $\pi' = r(\hat{\pi}) + (1 - r)\pi$.

Figure 3.2 show results for each algorithm on random symmetric normal form games of dimension 60, about the same dimension of the normal form of Kuhn poker. We run each algorithm on five different random symmetric normal form games. We report the mean exploitability over time of these algorithms and add error bars corresponding to the standard error of the mean. P2SRO reaches an approximate Nash equilibrium much faster than the other algorithms. Additional experiments on different dimension games and different learning rates are included in the supplementary material. In each experiment, P2SRO converges to an approximate Nash equilibrium much faster than the other algorithms.



Extended Figure 3.4: Barrage Best Response Payoffs Over Time

3.4.2 Leduc Poker

Leduc poker is played with a deck of six cards of two suits with three cards each. Each player bets one chip as an ante, then each player is dealt one card. After, there is a a betting round and then another card is dealt face up, followed by a second betting round. If a player's card is the same rank as the public card, they win. Otherwise, the player whose card has the higher rank wins. We run the following parallel PSRO algorithms on Leduc: P2SRO, DCH, Rectified PSRO, and Naive PSRO. We run each algorithm for three random seeds with three workers each. Results are shown in Figure 3.2. We find that P2SRO is much faster than the other algorithms, reaching 0.4 exploitability almost twice as soon as Naive PSRO. DCH and Rectified PSRO never reach a low exploitability.

3.4.3 Barrage Stratego

Barrage is a smaller variant of the board game Stratego that is played competitively by humans. The board consists of a ten-by-ten grid with two two-by-two barriers in the middle (see image for details). Each player has eight pieces, consisting of one Marshal, one General, one Miner, two Scouts, one Spy, one Bomb, and one Flag. Crucially, each player only knows

Name	P2SRO Win Rate vs. Bot
Asmodeus	81%
Celsius	70%
Vixen	69%
Celsius1.1	65%
All Bots Average	**71%**

Extended Table 3.1: Barrage P2SRO Results vs. Existing Bots

the identity of their own pieces. At the beginning of the game, each player is allowed to place these pieces anywhere on the first four rows closest to them.

The Marshal, General, Spy, and Miner may move only one step to any adjacent space but not diagonally. Bomb and Flag pieces cannot be moved. The Scout may move in a straight line like a rook in chess. A player can attack by moving a piece onto a square occupied by an opposing piece. Both players then reveal their piece's rank and the weaker piece gets removed. If the pieces are of equal rank then both get removed. The Marshal has higher rank than all other pieces, the General has higher rank than all other beside the Marshal, the Miner has higher rank than the Scout, Spy, Flag, and Bomb, the Scout has higher rank than the Spy and Flag, and the Spy has higher rank than the Flag and the Marshal when it attacks the Marshal. Bombs cannot attack but when another piece besides the Miner attacks a Bomb, the Bomb has higher rank. The player who captures his/her opponent's Flag or prevents the other player from moving any piece wins.

We find that the approximate exploitability of the meta-Nash equilibrium of the population decreases over time as measured by the performance of each new best response. This is shown in Figure 3, where the payoff is 1 for winning and -1 for losing. We compare to all existing bots that are able to play Barrage Stratego. These bots include: Vixen, Asmodeus, and Celsius. Other bots such as Probe and Master of the Flag exist, but can only play Stratego and not Barrage Stratego. We show results of P2SRO against the bots in Table 1. We find that P2SRO is able to beat these existing bots by 71% on average after $820,000$ episodes, and has a win rate

of over 65% against each bot. We introduce an open-source environment for Stratego, Barrage Stratego, and smaller Stratego games at `https://github.com/JBLanier/stratego_env`.

Chapter 4

XDO: A Double Oracle Algorithm For Extensive-Form Games

4.1 Introduction

Policy Space Response Oracles (PSRO) [60] is a reinforcement learning (RL) method for finding approximate Nash equilibria (NE) in large two-player zero-sum games. Methods based on PSRO have recently achieved state-of-the-art performance on large imperfect-information two-player zero-sum games such as Starcraft [107] and Stratego [74]. One major benifit of PSRO versus other deep RL methods for two-player zero-sum games is that it is naturally compatible with games that have continuous actions. The only other deep RL method compatible with continuous actions, self play, is not guaranteed to converge to a Nash equilibrium even in small games like Rock Paper Scissors. Despite the empirical success of PSRO, in the worst case, PSRO may need to expand all pure strategies in the normal form of the game, which grows exponentially in the number of information states (infostates). The reason for this is that PSRO is based on the Double Oracle algorithm for normal-form

games [77], and a mixture of normal-form pure strategies is an inefficient representation of extensive-form policies.

In this work, we propose a new double oracle algorithm, Extensive-Form Double Oracle (XDO), that is designed for extensive-form (sequential) games. Like PSRO, XDO keeps a population of pure strategies. At every iteration, XDO creates a restricted game by only considering actions that are chosen by at least one strategy in the population. This restricted game is then approximately solved via an extensive-form game solver, such as Counterfactual Regret Minimization (CFR) [110] or Fictitious Play (FP) [17], to find a meta-NE, which is extended to the full game by taking arbitrary actions at infostates not encountered in the restricted game. Next, a best response (BR) to the restricted game meta-NE is computed and added to the population. XDO can be viewed as a version of PSRO where the restricted game allows mixing population strategies not only at the root of the game, but at every infostate.

XDO is guaranteed to converge to an approximate NE in a number of iterations that is linear in the number of infostates, while PSRO may require a number of iterations exponential in the number of infostates. Furthermore, on a worst-case family of games for the lower bound on the number of PSRO iterations, we show that XDO converges in a number of iterations that does not grow with the number of infostates, and grows only linearly with the number of actions at each infostate.

We also introduce a neural version of XDO, called Neural XDO (NXDO). NXDO can be used in games that are large enough to benefit from the generalization over infostates induced by neural-network strategies. NXDO learns approximate BRs through any deep reinforcement learning algorithm. The restricted game consists of meta-actions, each selecting a population policy to play the next action. This restricted game is then solved through any neural extensive-form game solver, such as NFSP [45] or Deep CFR [22]. In our experiments, we use PPO [95] or DDQN [106] for the approximate BR and NFSP as the restricted game

solver. Although convergence guarantees may not apply in such cases, like PSRO, NXDO is compatible with continuous action spaces.

In games with a large number of actions, NXDO and PSRO effectively prune the game tree and outperform methods such as Deep CFR and NFSP, which cannot be applied at all with continuous actions. Additionally, because PSRO might require an exponential number of pure strategies, NXDO outperforms PSRO on games that require mixing over multiple timesteps. To demonstrate the effectiveness of our approach on these types of games, we run experiments on two sets of environments. The first, m-Clone Leduc, is similar to Leduc poker but with every call, fold, and bet action duplicated m times. The second, the Loss Game, is a sequential continuous-action multidimensional optimization game in which agents simultaneously adjust parameters to maximize or minimize a complex loss function. We show that tabular XDO greatly outperforms PSRO, CFR, and XFP [46] on m-Clone Leduc. We also show that NXDO outperforms both PSRO and NFSP on m-Clone Leduc and on the continuous-action Loss Game, where NFSP is provided a binned discrete action space.

To summarize, our contributions are as follows:

- We present a tabular extensive-form double oracle algorithm, XDO, that terminates in a linear number of iterations in the number of infostates.
- We present a neural version of XDO, NXDO, that outperforms PSRO and NFSP on both modified Leduc poker and sequential continuous-action games. NXDO is the first method that can find an approximate NE in high-dimensional continuous-action sequential games.

4.2 Background

4.2.1 Extensive-Form Games

We consider partially-observable stochastic games [41] which correspond to perfect-recall extensive-form games (from here on referred to as extensive-form games). An extensive-form game progresses through a sequence of player actions, and has a **world state** $w \in \mathcal{W}$ at each step. In an N-player game, $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_N$ is the space of joint actions for the players. $\mathcal{A}_i(w)$ denotes the set of legal actions for player $i \in \mathcal{N} = \{1, \ldots, N\}$ at world state w and $a = (a_1, \ldots, a_N) \in \mathcal{A}$ denotes a joint action. At each world state, after the players choose a joint action, a transition function $\mathcal{T}(w, a) \in \Delta^{\mathcal{W}}$ determines the probability distribution of the next world state w'. Upon transition from world state w to w' via joint action a, player i makes an **observation** $o_i = \mathcal{O}_i(w, a, w')$. In each world state w, player i receives a reward $\mathcal{R}_i(w)$.

A **history** is a sequence of actions and world states, denoted $h = (w^0, a^0, w^1, a^1, \ldots, w^t)$, where w^0 is the known initial world state of the game. $\mathcal{R}_i(h)$ and $\mathcal{A}_i(h)$ are, respectively, the reward and set of legal actions for player i in the last world state of a history h. An **infostate** for player i, denoted by s_i, is a sequence of that player's observations and actions up until that time $s_i(h) = (a_i^0, o_i^1, a_i^1, \ldots, o_i^t)$. Define the set of all infostates for player i to be $\mathcal{I}_i$. The set of histories that correspond to an infostate s_i is denoted $\mathcal{H}(s_i) = \{h : s_i(h) = s_i\}$, and it is assumed that they all share the same set of legal actions $\mathcal{A}_i(s_i(h)) = \mathcal{A}_i(h)$.

A player's **policy** π_i is a function mapping from an infostate to a probability distribution over actions. A **policy profile** π is a tuple $(\pi_1, \ldots, \pi_N)$. All players other than i are denoted $-i$, and their policies are jointly denoted π_{-i}. A policy for a history h is denoted $\pi_i(h) = \pi_i(s_i(h))$ and $\pi(h)$ is the corresponding policy profile. We also define the transition function $\mathcal{T}(h, a_i, \pi_{-i}) \in \Delta^{\mathcal{W}}$ as a function drawing actions for $-i$ from π_{-i} to form $a = (a_i, a_{-i})$

and to then sample the next world state w' from $\mathcal{T}(w, a)$, where w is the last world state in h.

The **expected value (EV)** $v_i^\pi(h)$ for player i is the expected sum of future rewards for player i in history h, when all players play policy profile π. The EV for an infostate s_i is denoted $v_i^\pi(s_i)$ and the EV for the entire game is denoted $v_i(\pi)$. A **two-player zero-sum** game has $v_1(\pi) + v_2(\pi) = 0$ for all policy profiles π. The EV for an action in an infostate is denoted $v_i^\pi(s_i, a_i)$. A **Nash equilibrium (NE)** is a policy profile such that, if all players played their NE policy, no player could achieve higher EV by deviating from it. Formally, π^* is a NE if $v_i(\pi^*) = \max_{\pi_i} v_i(\pi_i, \pi^*_{-i})$ for each player i.

The **exploitability** $e(\pi)$ of a policy profile π is defined as $e(\pi) = \sum_{i \in \mathcal{N}} \max_{\pi'_i} v_i(\pi'_i, \pi_{-i})$. A **best response (BR)** policy $\mathbb{BR}_i(\pi_{-i})$ for player i to a policy π_{-i} is a policy that maximally exploits π_{-i}: $\mathbb{BR}_i(\pi_{-i}) = \arg\max_{\pi_i} v_i(\pi_i, \pi_{-i})$. An **$\epsilon$-best response ($\epsilon$-BR)** policy $\mathbb{BR}_i^\epsilon(\pi_{-i})$ for player i to a policy π_{-i} is a policy that is at most ϵ worse for player i than the best response: $v_i(\mathbb{BR}_i^\epsilon(\pi_{-i}), \pi_{-i}) \geqslant v_i(\mathbb{BR}_i(\pi_{-i}), \pi_{-i}) - \epsilon$. An **$\epsilon$-Nash equilibrium ($\epsilon$-NE)** is a policy profile π in which, for each player i, π_i is an ϵ-BR to π_{-i}.

A **normal-form game** is a single-step extensive-form game. An extensive-form game induces a normal-form game in which the legal actions for player i are its deterministic policies $\times_{s_i \in \mathcal{I}_i} \mathcal{A}_i(s_i)$. These deterministic policies are called **pure strategies** of the normal-form game. Since each deterministic policy specifies one action at every infostate, there are an exponential number of pure strategies in the number of infostates. A **mixed strategy** is a distribution over a player's pure strategies. Two policies π_i^1 and π_i^2 for player i are said to be **realization-equivalent** if for any fixed strategy profile of the other player, both π_i^1 and π_i^2, define the same probability distribution over the states of the game.

Theorem 2 (Kuhn's Theorem [58])**.** *Any mixed strategy in the normal form of a game is realization equivalent to a policy in the extensive form of that game, and vice versa.*

4.3 Related Work

There has been much recent work on non-game-theoretic multi-agent RL [32, 69, 86, 10]. Most of this work focuses on games with more than two players such as multi-agent cooperative games or mixed competitive-cooperative scenarios. In cooperative environments, self-play has empirically been shown to find an approximate NE [69, 70], but can be brittle when cooperating with agents it hasn't trained with [60]. Self-play reinforcement learning has achieved expert level performance on video games [107, 12, 48], but is not guaranteed to converge to an approximate NE.

Extensive-form fictitious play (XFP) [46] and counterfactual regret minimization (CFR) [110] extend Fictitious Play (FP) [17] and regret matching [43], respectively, to extensive-form games. Deep CFR [22, 101, 64] is a general method that trains a neural network on a buffer of counterfactual values. However, Deep CFR uses external sampling, which may be impractical for games with a large branching factor, such as Stratego and Barrage Stratego. DREAM [102] and ARMAC [39] are model-free regret-based deep learning approaches. DREAM and ARMAC have achieved good results in poker games, but since they are based on MCCFR, like Deep CFR, they will not scale to games with continuous actions.

Our work is related to pruning approaches [18, 21]. These methods start with all actions and sequentially remove actions that have low expected value. XDO instead starts with no actions and sequentially adds actions. Our work is also related to methods that automatically find abstractions [19, 25].

Close to our work, [15] develop a sequence-form double oracle (SDO) algorithm. The SDO algorithm iteratively adds sequence-form BRs to a population and then computes a meta-Nash on a restricted sequence-form game where only sequences in the population are allowed. In contrast, XDO iteratively adds extensive-form BRs to a population and then computes a meta-Nash on a restricted extensive form game where only actions in the population are

allowed. DO, SDO, and XDO are fundamentally different because they operate on the normal form, sequence form, and extensive form, respectively. We give a detailed description of the difference between XDO and SDO in the supplementary materials.

4.3.1 Neural Fictitious Self Play (NFSP)

Neural Fictitious Self Play (NFSP) [45] approximates XFP by progressively training a best response against an average of all past policies using reinforcement learning. The average policy is represented by a neural network and is trained via supervised learning using a replay buffer of past best response actions. Each episode, both players either play from their best response policy with probability $\eta = 0.1$ or with their average policy with probability $1 - \eta$. This experience is then added to the best response circular replay buffer and is used to train the best response for both players with off-policy DQN. If a player plays with their best response policy, the data is also added to the average policy reservoir replay buffer and is used to train the average policy via supervised learning.

4.3.2 Policy Space Response Oracles (PSRO)

The Double Oracle algorithm [77] is an algorithm for finding a NE in normal-form games. The algorithm works by keeping a population of policies Π^t at time t. Each iteration a meta-Nash Equilibrium (meta-NE) $\pi^{*,t}$ is computed for the game restricted to policies in Π^t. Then, a best response to this meta-NE for each player $\mathbb{BR}_i(\pi^{*,t}_{-i})$ is computed and added to the population $\Pi_i^{t+1} = \Pi_i^t \cup \{\mathbb{BR}_i(\pi^{*,t}_{-i})\}$ for $i \in \{1, 2\}$.

Policy Space Response Oracles (PSRO) [60] approximates the Double Oracle algorithm. The meta-NE is computed on the empirical game matrix U^{Π}, given by having each policy in the population Π play each other policy and tracking average utility in a payoff matrix. In each

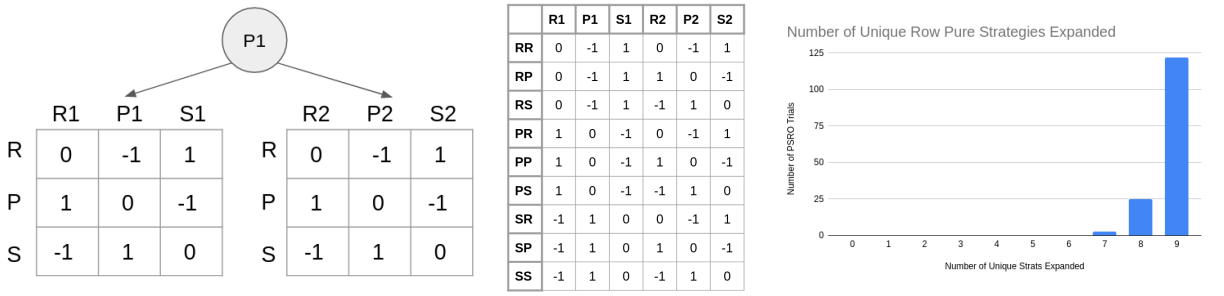
iteration, an approximate best response to the current meta-NE over the policies is computed via any reinforcement learning algorithm. Pipeline PSRO parallelizes PSRO with convergence guarantees [74].

PSRO Hard Instance

A primary issue with PSRO is that it is based on a normal-form algorithm, and the number of pure strategies in a normal-form representation of an extensive-form game is exponential in the number of infostates. In contrast, our approach implements the double oracle algorithm directly in the extensive-form game, overcoming this problem and terminating in a linear number of iterations in the number of infostates. The following example helps illustrate this point.

Consider the game in Figure 4.1. In this game, first, player 1 chooses which Rock Paper Scissors (RPS) game both players play. After player 1 chooses the RPS game, both players know which RPS game they are playing. Then both players simultaneously play an action in that RPS game. There are 6 pure strategies for player 1, denoted R1, P1, S1, R2, P2, S2. But there are 9 pure strategies for player 2. A pure strategy for player 2 specifies what move they play at each infostate. If player 2 played Rock in the first infostate and Paper in the second, that pure strategy is denoted RP. Note that if we generalize this game by including more RPS games and more actions in each game, the number of pure strategies for player 2 will be $|A|^{|\mathcal{I}|}$, where $|A|$ is the number of actions and $|\mathcal{I}|$ is the number of RPS games.

We conduct an experiment where we run PSRO with oracle BRs on this game with a random starting population each time. We find that PSRO expands all 9 row (player 2) pure strategies the majority of the time, expanding all 9 strategies in 122 out of 150 trials. These results are shown in Figure 4.1c. We also find that the column player (player 1) expands all 6 pure strategies in all 150 trials.



(a) (b) (c)

Extended Figure 4.1: PSRO hard instance. (a) Player 1 first chooses which RPS game both players play. Both players know which RPS game they are playing. Then both players simultaneously make their move. (b) The normal form game. Player 2 has 9 pure strategies. (c) The proportion of PSRO trials that expanded each possible number of pure strategies for player 2. In the majority of trials, PSRO had to expand all possible pure strategies.

4.4 Extensive-Form Double Oracle (XDO)

We propose Extensive-Form Double Oracle (XDO), a double-oracle (DO) algorithm designed for two-player zero-sum extensive-form games (Algorithm 3). As in other DO algorithms, XDO maintains a population of pure strategies, and in each iteration computes a meta-NE of this population. Then the algorithm finds a best response (BR) to the meta-NE and adds it to the population.

In XDO, the population induces a different restricted game, and therefore a different population meta-NE, than in PSRO [60]. In PSRO, a restricted normal-form game is induced by the empirical payoff matrix of population strategies. In XDO, a restricted extensive-form game is induced through a transformation on the original base extensive-form game that restricts the allowed actions at each infostate to only those suggested by any strategy in the population.

XDO uses a tabular method such as CFR [110] or XFP [46] to solve the restricted game. The algorithm terminates after an iteration in which neither of the players finds a BR that outperforms the meta-NE. When this happens, the meta-NE policies are approximate BRs to each other in the original game as well, and the meta-NE is therefore an approximate NE

of the original game.

Importantly, at each but the final iteration of XDO, at least one player adds some new action at some non-terminal infostate, because a BR cannot outperform the meta-NE with only restricted-game actions. The number of iterations that XDO takes to terminate is therefore at most the number of infostates, including terminal ones. In contrast, the best known guarantee for the number of iterations that PSRO takes to terminate (Proposition 4) is exponential in the number of infostates, because PSRO may need to add all pure strategies to the population. Moreover, computing the meta-NE in PSRO may become intractable in later iterations as the population size increases, while in XDO it is bounded by the unrestricted game.

Formally, XDO keeps a population of pure strategies Π^t at time t. Each iteration, a restricted extensive-form game is created and a NE to the restricted game is computed. The restricted game is created by taking the original game and restricting the actions at every infostate s_i to be only the actions where there exists a policy in the population Π^t that chooses that action at that infostate:

$$\mathcal{A}_i^r(s_i) = \{a \in \mathcal{A}_i(s_i) : \exists \pi_i \in \Pi_i^t \text{ s.t. } \pi_i(s_i, a) = 1\}. \tag{4.1}$$

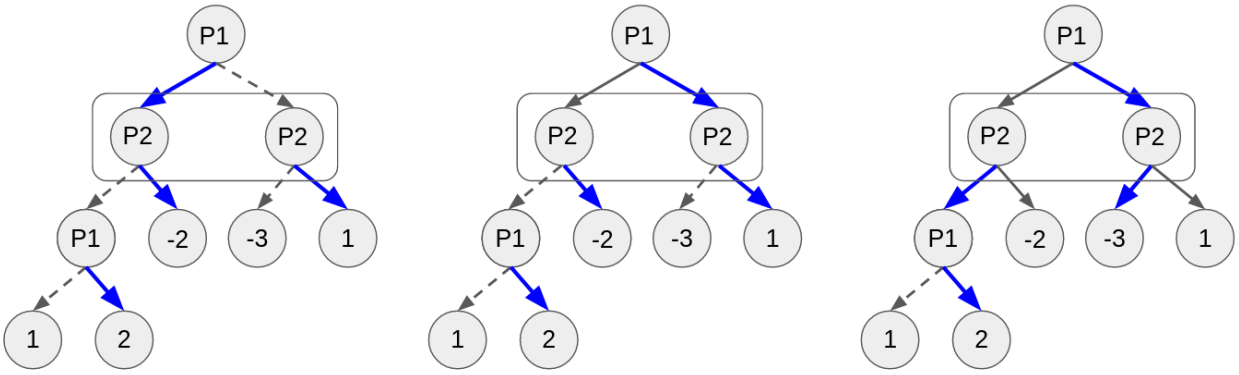
An ϵ-NE policy π^{r*} is then computed in this restricted game via a tabular method such as CFR and is extended to the full game by defining arbitrary actions on infostates not encountered in the restricted game. Next, BRs to this restricted game meta-NE $\mathbb{BR}_1(\pi_2^{r*})$ and $\mathbb{BR}_2(\pi_1^{r*})$ are computed via an oracle. These BRs are then added to the population of policies: $\Pi_i^{t+1} = \Pi_i^t \cup \mathbb{BR}_i(\pi_{-i}^{r*})$ for $i \in \{1, 2\}$.

The algorithm terminates when neither player benefits more than ϵ from deviating from the meta-NE to the BR, indicating that the meta-NE is an ϵ-NE also in the original game (Proposition 3).

Algorithm 3 XDO

Input: initial population Π^0
repeat
    Define restricted game for Π^t via eq. (4.1)
    Get ϵ-NE policy π^{r*} of restricted game
    Find $\mathbb{BR}_i(\pi^{r*}_{-i})$ for $i \in \{1, 2\}$
    if $v_i(\mathbb{BR}_i(\pi^{r*}_{-i}), \pi^{r*}_{-i}) \leqslant v_i(\pi^{r*}) + \epsilon$ for both i **then**
        Terminate
    $\Pi^{t+1}_i = \Pi^t_i \cup \mathbb{BR}_i(\pi^{r*}_{-i})$ for $i \in \{1, 2\}$

To illustrate how XDO works, we demonstrate a simple game in Figure 4.2. The algorithm starts with empty populations. At the first iteration (left diagram), player 1 adds a BR that plays Left at the first infostate (the root) and Right at the second one. Player 2 simultaneously adds a BR that plays Right at their single infostate. The restricted game now consists of only these added actions. At the second iteration (middle diagram), player 1 adds a BR that plays Right at both infostates, and player 2's BR still plays Right. The restricted game now includes both actions for the root infostate, but only Right is in the meta-NE. Next, in the third iteration (right diagram), player 1 keeps the same BR, while player 2's BR plays Left. In the meta-NE of this final restricted game, player 1 plays Left and Right with equal probability at the first infostate, and player 2 plays Left with probability 0.37 and Right with probability 0.63. Since the BRs to this meta-NE do not add any new actions, XDO terminates, and the meta-NE is the NE for the full game. Note that in this example, most actions are needed to find a NE. In games like this, it would be faster to simply solve the original game from the beginning. However, certain games, such as those in our experiments, have Nash equilibria that only need to mix over a small subset of actions [93], in which case solving the XDO restricted game will be much faster than solving the original game.



Extended Figure 4.2: Three iterations of XDO (left to right). In these extensive-form game diagrams, player 1 (P1) plays at the root, then P2 plays without knowing P1's action, and if both played Left P1 plays another action. P1's reward is number at the reached leaf. Actions in the restricted game are solid, vs. dashed outside the restricted game. Meta-NE actions are blue, vs. black not in the meta-NE.

4.4.1 Theoretical Considerations

In this section, we present a theoretical analysis of XDO and compare it with PSRO. Our first proposition states that, when XDO terminates, the final meta-NE of the restricted game is an approximate NE of the full game.

Proposition 3. *In XDO with an ϵ_1-BR oracle, let π^{r*} be the final ϵ_2-NE in the restricted game. Then π^{r*} is an $(\epsilon_1 + \epsilon_2)$-NE in the full game.*

Proof. For each $i \in \{1, 2\}$, let $\mathbb{BR}_i^{\epsilon_1}(\pi_{-i}^{r*})$ be player i's ϵ_1-BR to π_{-i}^{r*} obtained in the last iteration. By the termination condition

$$v_i(\pi^{r*}) \geqslant v_i(\mathbb{BR}_i^{\epsilon_1}(\pi_{-i}^{r*}), \pi_{-i}^{r*}) - \epsilon_2 \tag{4.2}$$

$$\geqslant \max_{\pi_i'} v_i(\pi_i', \pi_{-i}^{r*}) - \epsilon_1 - \epsilon_2, \tag{4.3}$$

where the last inequality follows from $\mathbb{BR}_i^{\epsilon_1}(\pi_{-i}^{r*})$ being an ϵ_1-best response to π_{-i}^{r*}. □

The next two propositions show an exponential gap in the known guarantees for the number

of iterations in which PSRO and XDO terminate. If each non-terminal infostate allows A different actions, PSRO is guaranteed to terminate in $\sum_i A^{|\bar{\mathcal{I}}_i|}$ iterations, where $\bar{\mathcal{I}}_i$ is the set of non-terminal infostates for player i, while XDO is guaranteed to terminate in $\sum_i |\mathcal{I}_i|$ iterations.

Proposition 4. *Normal-form DO terminates in at most* $\sum_i \prod_{s_i \in \mathcal{I}_i} |\mathcal{A}_i(s_i)|$ *iterations.*

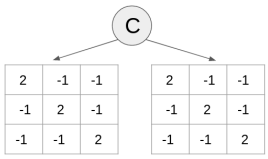
Proof. In each iteration of DO, at least one player adds a new normal-form pure strategy to the population. The space of pure strategies for player i has size $\prod_{s_i \in \mathcal{I}_i} |\mathcal{A}_i(s_i)|$, because each normal-form pure strategy specifies an action at each infostate for that player. □

Proposition 5. *XDO terminates in at most* $\sum_i |\mathcal{I}_i|$ *iterations.*

Proof. Consider an infostate $s'_i = (a_i^0, o_i^1, \ldots, a_i^t, o_i^{t+1})$ for player i as *covered* in the restricted game if any of player i's population policies chooses action a_i^t in infostate $s_i = (a_i^0, o_i^1, \ldots, a_i^{t-1}, o_i^t)$. At each but the final iteration, at least one player i has $v_i(\mathbb{BR}_i(\pi_{-i}^{r*}), \pi_{-i}^{r*}) > v_i(\pi^{r*}) + \epsilon$. Since π_i^{r*} is an ϵ-BR to π_{-i}^{r*} in the restricted game, the BR $\mathbb{BR}_i(\pi_{-i}^{r*})$ must be choosing at least some action a_i at some non-terminal infostate s_i that was not previously chosen by any population policy. Adding this action to the restricted game covers at least one previously uncovered infostate: all infostates $s'_i = (s_i, a_i, o_i)$, for any observation o_i. All infostates will therefore be covered in at most $\sum_i |\mathcal{I}_i|$ iterations, at which point the next iteration must terminate. □

Tightness of the guarantees. The guarantees in Proposition 4 and Proposition 5 are tight in the sense that they are achieved in some games, but more nuanced analysis is required to identify easier cases where these bounds overestimate the complexity of the algorithms. Both PSRO and XDO often outperform these guarantees and terminate in fewer iterations. A case in which PSRO expands all pure normal-form strategies of an extensive-form game is described in the supplementary materials.

XDO can add multiple actions in each iteration. In practice, XDO often outperforms the guarantee of Proposition 5 because it adds multiple actions in each iteration. Here we present and analyze a family of games in which XDO terminates in asymptotically fewer iterations than suggested by the bound in Proposition 5.



Extended Figure 4.3: A 2-GMP game with $n = 3$ actions. The chance node selects uniformly at random which generalized matching pennies game is played. Both players know which stage game they play.

In a generalized matching pennies (GMP) game, both players simultaneously choose one of n actions. The payoff to player 1 is $n - 1$ if the actions match, or -1 if they are different. In a k-GMP game (Figure 4.3), a chance node first selects an index j between 1 and k, and then the players play the j'th of k identical GMP games. The following proposition provides a tighter performance bound for XDO in this case, $2n$ iterations instead of $\sum_i |\mathcal{I}_i| = 2k(n+1)$ (there are kn terminal infostates for each player). For PSRO on k-GMP, no tighter bound than the $2n^k$ indicated by Proposition 4 is known.

Proposition 6. *In k-GMP with n actions, XDO terminates in $2n$ iterations.*

Proof. In a given iteration, consider the restricted game for a single GMP game. If player 2 is allowed an action that player 1 is not, such an action will be player 2's NE, and player 1's BR will add that action. If player 2 is not allowed an action unavailable to player 1, player 2's BR will be a new action unavailable to player 1, if one exists. Thus at least one of the players add a new action in every GMP game in parallel, until both players add all actions. □

Size of the restricted game. The number of iterations in each algorithm does not provide the full picture of their performance, since iterations can require vastly different computation times. Intuitively, the restricted game in XDO is much larger than in PSRO when both algorithms have the same population size, because XDO induces an extensive-form restricted

game with all discovered actions, while PSRO induces a normal-form restricted game with population policies as actions. However, as both algorithms progress, the XDO restricted game is bounded in size by the original game, while PSRO can induce a game with exponentially many actions.

XDO for sparse-support policies. XDO is useful when the policies in the population do not cover the full original game, because when they do, finding the restricted game meta-NE is as hard as solving the original game. The motivation behind XDO is that, in games where the NE policies are supported by few actions in most infostates, XDO has the potential to quickly find these actions and terminate without expanding the full game.

To analyze this behavior, consider the m-clone GMP game, in which there are mn actions partitioned into n equal classes. The actions of the two players are considered a match (with payoff $n-1$ to player 1) if they belong to the same class. In (k, m)-clone GMP, a chance node selects among k identical m-clone GMP games. The following proposition states that in (k, m)-clone GMP with n classes, XDO terminates after adding at most $2n$ actions for each player, instead of the full game of kmn actions.

Proposition 7. *In (k, m)-clone GMP with n classes, XDO adds at most $2n$ actions for each player.*

Proof. The proof repeats that of Proposition 6, but considering classes instead of actions, because it does not matter which member of a class is added. Once at least one member of each class is added to the restricted game, the meta-NE has full-game exploitability 0, and XDO terminates. In iterations where a BR for a player does not add a new class, it may add a new action member of an existing class. In total, $2n$ actions may be added for each player. □

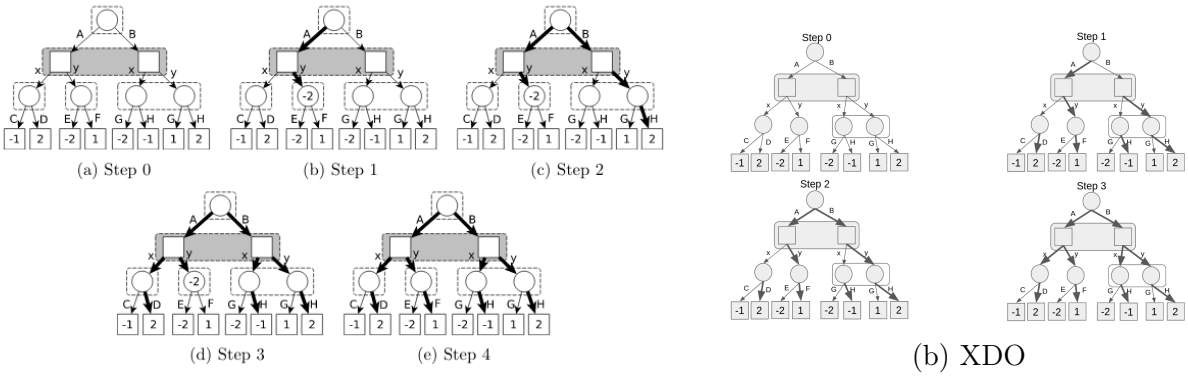
PSRO lower bound. Similarly to XDO, PSRO can also outperform the guarantee of Proposition 4 in certain cases. Generically, however, the linear upper bound on XDO established by Proposition 5, $\sum_i |\mathcal{I}_i|$, is also a *lower bound* on the normal-form population size of pure strategies that is needed to support a NE in PSRO. To show this, consider a perturbed k-GMP game, in which the payoffs in each GMP game are slightly modified to induce k distinct NE. The following proposition establishes a linear lower bound for PSRO in perturbed k-GMP games.

Proposition 8. *There exist perturbed k-GMP games with n actions in which PSRO cannot terminate in fewer than $k(n-1)+1$ iterations.*

Proof. For each policy $\pi_2 \in (\Delta(n))^k$ for player 2, consider the perturbed k-GMP game that gives player 1 payoff $\frac{1}{\pi_2(j,a)}$ for matching action a in stage game j. In the NE for this game, player 2 has policy π_2. This implies that the set of policies that are NE of any perturbed k-GMP game has positive $k(n-1)$-dimensional volume.

Consider the space of stochastic policies that can be spanned by mixing a specific population of at most $k(n-1)$ pure strategies. The dimension of this space is at most $k(n-1)-1$. When we consider the union of all such spaces for the finitely many possible populations of this size, this set has zero $k(n-1)$-dimensional volume.

It follows that there exists a perturbed k-GMP game G, and a neighborhood around player 2's policy in the NE of G, such that no policy in that neighborhood is spanned by any population of $k(n-1)$ pure strategies. For sufficiently small ϵ, no ϵ-NE for G can be found until PSRO adds at least $k(n-1)+1$ pure strategies to its population. □



(a) SDO [15]

(b) XDO

Extended Figure 4.4

4.4.2 Comparison to SDO

To illustrate the difference between the two algorithms, we provide an example run of XDO (Figure 4.4b) and SDO (Figure 4.4a) on the same game that is presented as an example in section 4.1.3 in the SDO paper [15].

Like in that work, we represent actions that are in the restricted game by bold arrows. This example demonstrates how SDO creates smaller restricted games than XDO because it only considers infostates that can be reached by compatible sequences. In tabular games, SDO results in a cautious approach that only considers a small subset of infostates in order to prevent adding suboptimal actions to the restricted game. However, as we describe below, this will cause obstacles when trying to scale SDO to large games.

Extensive-form pure strategies specify an action at every infostate. In this example we will refer to extensive-form pure strategies by concatenating the actions the strategy takes in every infostate. For example, the pure strategy for the circle player in step 1 in our diagram corresponds to ADFH. Sequence-form pure strategies specify a sequence of actions that must be internally consistent. For example, $\{\varnothing, A, AD\}$ is a valid sequence-form pure strategy but $\{\varnothing, B, AD\}$ is not.

In step 0, both SDO and XDO start with an empty game tree. Let's assume that the default strategy for both algorithms is uniform random.

In step 1, both SDO and XDO add the same best responses to the default strategy for both players. However, SDO adds actual sequences of actions, in particular $\{\varnothing, A, AD\}$ for the circle player and $\{\varnothing, y\}$ for the box player. Since the AD sequence of actions for the circle player is not compatible with the y action for the square player, the restricted game is the game with A as the available action for circle and y as the available action for square. But since neither AE nor AF are in the sequence population, the restricted game in SDO at this step terminates in a temporary leaf at that point.

In contrast, XDO adds full extensive form strategies, in this case adding ADFH for the circle player and y for the square player. Now the restricted game for XDO is as in step 1 in the diagram. Since XDO adds full extensive form strategies, there is no need to create a temporary leaf node.

After only one iteration, SDO and XDO result in a different restricted game. This is because SDO adds sequences of actions (best responses in sequence form) to the population while XDO adds a pure strategy defined at every infostate (best responses in extensive form). The SDO restricted game is much smaller than the XDO restricted game because SDO only considers infostates that can be reached by compatible sequences in the population. There are three more steps for SDO, which are described in their paper. XDO has only two more steps, which are described in the figure.

There currently exists one main algorithm that can scale up the double oracle approach to large games through deep reinforcement learning, which is PSRO. We propose another, Neural Extensive-Form Double Oracle (NXDO). Similarly extending SDO to large games via neural networks is one possible direction for future work.

Algorithm 4 NXDO

Input: initial population Π^0
repeat
    Define restricted game for Π^t via eq. (4.4)
    Get ϵ-NE policy π^{r*} of restricted game via NFSP
    Find $\mathbb{BR}_i(\pi^{r*}_{-i})$ for $i \in \{1, 2\}$ via DRL
    $\Pi^{t+1}_i = \Pi^t_i \cup \mathbb{BR}_i(\pi^{r*}_{-i})$ for $i \in \{1, 2\}$

4.5 Neural Extensive-Form Double Oracle (NXDO)

Neural Extensive-Form Double Oracle (NXDO) extends XDO to large games through deep reinforcement learning (DRL). Instead of using an oracle best response, NXDO instead uses approximate best responses that are trained via any DRL algorithm, such as PPO [95] or DDQN [106]. Instead of representing the restricted game explicitly as the set of allowed actions in every infostate, to create its restricted game, NXDO replaces the original game action space with a discrete set of meta-actions, each corresponding to a population policy to which the actual action choice is delegated.

Formally, NXDO (Algorithm 4) keeps a population of DRL policies Π^t at time t. Each iteration, a restricted extensive-form game is created and a meta-NE to the restricted game is computed. The restricted game has meta-actions at every infostate that pick one policy from the population

$$\forall s_i \in \mathcal{I}_i \quad \mathcal{A}^r_i(s_i) = \{1, 2, ..., |\Pi^t_i|\}. \tag{4.4}$$

While the action space differs, the restricted game states, observations, and histories remain the same as in the original game. After each player selects a meta-action that indicates a population policy, an action is sampled from that population policy and used for the world state transition. With $\pi^1_i, \ldots, \pi^{|\Pi_i|}_i$ the population policies for player i, the transition function

in the restricted game satisfies

$$\mathcal{T}^r(h, a^r, w') = \sum_a \prod_i \pi_i^{a_i^r}(s_i(h), a_i)\mathcal{T}(h, a, w'). \tag{4.5}$$

With the restricted game thus defined, an ϵ-meta-NE π^{r*} is computed in this restricted game via a DRL method for finding NE, such as NFSP [45] or DREAM [102]. Approximate BRs $\mathbb{BR}_1(\pi_2^{r*})$ and $\mathbb{BR}_2(\pi_1^{r*})$ to this meta-NE are computed via a DRL algorithm, such as PPO or DDQN. These BRs are then added to the population of policies: $\Pi_i^{t+1} = \Pi_i^t \cup \mathbb{BR}_i(\pi_{-i}^{r*})$ for $i \in \{1, 2\}$. Provided that the DRL best responses are sufficiently close to oracle best responses and the inner-loop solver finds a sufficiently close approximate NE of the restricted game, NXDO inherits the same convergence properties as XDO. In practice, contemporary DRL methods lack any guarantee of providing approximate NE or BRs. Nevertheless, we show experimentally that exploitability can decrease through execution of NXDO faster than it does for PSRO and NFSP.

Because the original game action space has no influence on the NXDO restricted game action space, NXDO, like PSRO, is compatible with extremely large and continuous action spaces, provided that the deep RL BRs can operate in such an action space well. We demonstrate this capability in our Loss Game experiments, in which NXDO and PSRO use continuous-action PPO BRs. While no convergence guarantees are known in continuous-action games, NXDO and PSRO empirically produce meta-Nash strategies that are hard to exploit in our experiments.

A drawback of meta-actions that delegate actions to population policies is that, as in PSRO, the number of meta-actions grows linearly with the number of iterations. This can eventually make the restricted game harder to solve than the original game. In our experiments, however, NXDO achieves significant improvements in exploitability within a very small number of iterations, such that the issue of action delegation does not become an obstacle. In discrete

action space games where it is tractable, we also consider a variant, NXDO-VA, where the restricted game is explicitly calculated and defined with valid and invalid original-game actions in the same way as with tabular XDO, using equation (4.1). Because its restricted game action space size is at most equal to that of the original game, NXDO-VA does not suffer from the aforementioned drawback.

4.6 Experiments

4.6.1 Game Descriptions

***m*-Clone Leduc poker:** m-Clone Leduc poker is similar to Leduc poker but with every action duplicated m times, such that instead of a single call, fold, and bet action there are m identical call, fold, and bet actions. As the number of cloned actions increases, we expect the performance of methods based on CFR and FP such as DREAM and NFSP to deteriorate, while the performance of XDO and PSRO remains largely unchanged because the size of the restricted games scale with the size of the meta-game population rather than the size of the action space.

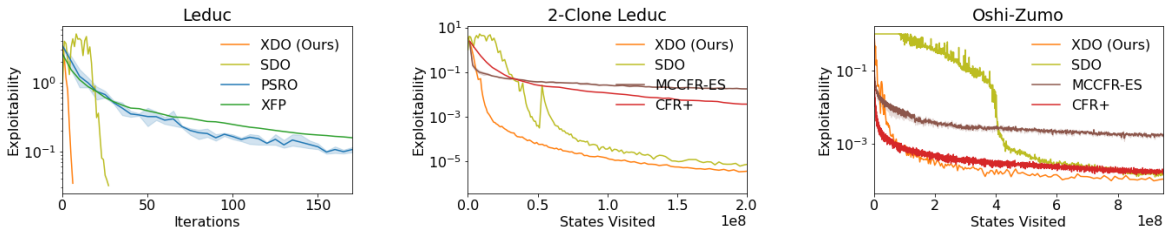
Oshi-Zumo Oshi-Zumo is a zero-sum two-player sequential game where both players try to move a token to the other player's side of the board. Both players start with an allotment of coins and each step simultaneously bet an amount of coins. The player who bets the higher amount gets to move the token toward the opponent. A player wins if they are able to move the token off the board on their opponent's side or if the token is on the opponent's side when either no more coins remain or the time horizon is reached. Reward is 1 for wining, -1 for losing, and 0 for a tie, which can occur when the token is in the middle of the board when the game ends. We consider a variant with 4 coins, and a board of 3 spaces with a

time horizon of 6.

Loss Game: The Loss Game is a zero-sum two-player sequential continuous action game in which agents simultaneously apply bounded adjustments to a real valued vector of parameters $\boldsymbol{x} = [x_1, \ldots, x_d]$ in order to optimize a fixed loss function's scalar output $L : \mathbb{R}^d \mapsto \mathbb{R}$. One agent aims to maximize this output while to other aims to minimize it. Each timestep, agents observe the current vector of parameter values along with the function's scalar output value and provide a bounded continuous action describing an adjustment vector to add to the current parameters. The sum of both agents' adjustments is applied and player 1 is rewarded with the value of the function's output while player 2 is provided with the negative of this value. The game lasts 10 steps and the parameters' adjusted values are preserved after each step.

We consider a 2D action space variation with the loss function $L(x_1, x_2) = \sum_j \sin(x_j)$ and a 16D action space variation with $L(x_1, \ldots, x_{16}) = \sin(\sum_j x_j) + \sum_j \sin(x_j)/16$. These functions were chosen to demonstrate games in which it is advantageous to mix between multiple strategies in many infostates. The 2 and 16 dimensional action spaces where chosen to represent continuous spaces that, respectively, can and cannot be directly binned into a tractable amount of discrete actions. We test the binned discrete action version of the 2D continuous action space Loss Game with both 10 and 100 bins in each of the two dimensions , totalling 100 and 10,000 actions respectively.

For the tabular experiments, we use XDO with an oracle best response (BR) and CFR$^+$ [104] for the inner-loop meta-NE solver. We compare XDO with PSRO [60] and XFP [46], which use oracle BRs as well. We also compare with CFR$^+$, and for both CFR$^+$ and XFP we follow the implementations in OpenSpiel [62]. We compare to SDO, using the same oracle BR and meta-NE solver as XDO. Since CFR$^+$, XFP, SDO, and XDO are deterministic, we do not plot error bars for these algorithms. For the neural experiments, we use NXDO with



(a) (b) (c)

Extended Figure 4.5: (a) Exploitability in Leduc poker of XDO vs. PSRO, SDO, and XFP with oracle BRs throughout their iterations; (b) Exploitability in 2-Clone Leduc poker as a function of the number of game states visited by XDO, SDO, CFR$^+$, and MCCFR with external sampling (ES); (c) Exploitability in Oshi-Zumo as a function of the number of game states visited by XDO, SDO, CFR$^+$, and MCCFR-ES

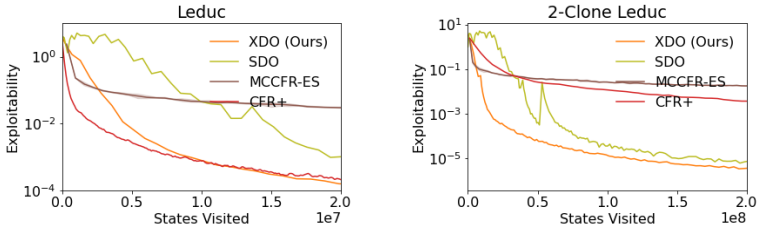
NFSP [45] as the meta-NE solver and PSRO with FP [17] as the meta-NE solver. NXDO and PSRO share the same BR configuration, using DDQN [106] for discrete action spaces and PPO [95] for continuous action spaces. We compare these algorithms on m-Clone Leduc poker, Oshi-Zumo, and the Loss Game, described in the supplementary materials.

4.6.2 Tabular Experiments with XDO

Finding a normal-form meta-NE can be much less efficient and more exploitable than finding an extensive-form meta-NE. This means that PSRO will often require many more pure strategies to achieve a similar level of exploitability to XDO. Figure 4.5a summarizes the results of running XDO, SDO, XFP, and PSRO with an oracle BR on Leduc poker. Even after 150 iterations, PSRO remains significantly more exploitable than XDO is at 7 iterations. XDO achieves exploitability of 0.1 in over 20x fewer iterations than PSRO. In large games where calculating many approximate BRs via reinforcement learning is expensive, requiring vastly more iterations can render PSRO infeasible. XFP performs similarly to PSRO in Leduc, as its average policy is equivalent to uniformly mixing population strategies at the root of the game. SDO takes more iterations than XDO to achieve low exploitability, because it adds fewer actions to the restricted game per iteration.

We compare XDO with SDO, CFR$^+$, and MCCFR [59] in 2-Clone Leduc poker. In Figure 4.5b, we plot the exploitability of these algorithms as a function of the number of game states visited by the algorithms. Since XFP and PSRO only use BR oracles, we do not include them in this analysis. Since CFR$^+$ updates every infostate every iteration, as we increase the number of cloned actions, the performance of CFR$^+$ will deteriorate. In contrast, XDO will tend to not add cloned actions, which allows the inner-loop CFR$^+$ to expand fewer infostates. These results for XDO are with a deterministic best response oracle. We found that if XDO randomly chose a best response instead, then XDO would still outperform CFR$^+$, but not by as much. The results in Figure 4.5c suggest that on Oshi-Zumo XDO outperforms SDO, MCCFR, and, to a lesser degree, CFR$^+$. We do not include XDO with MCCFR as the inner loop solver because it did not perform as well as XDO with CFR$^+$.

Empirical Analysis of XDO Restricted Game Sizes

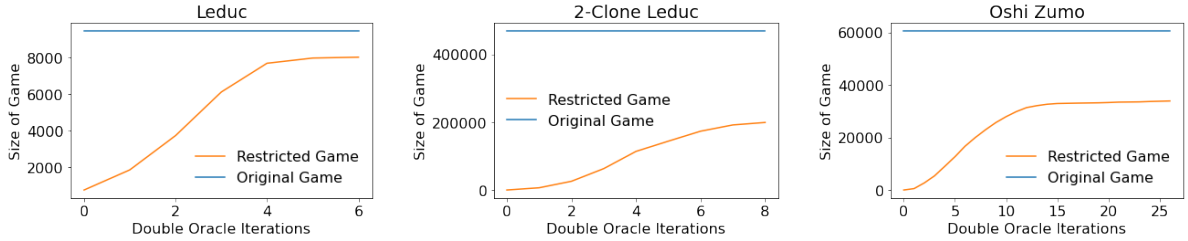


(a) (b)

Extended Figure 4.6: XDO exploitability compared to CFR$^+$, SDO and MCCFR as a function of states visited in (a) Leduc poker and (b) 2-Clone Leduc Poker.

XDO outperforms CFR$^+$ in games where Nash equilibria require mixing over a small fraction of actions. For example, XDO performs similarly to CFR$^+$ in standard Leduc poker, but XDO significantly outperforms CFR$^+$ in 2-Clone Leduc poker. XDO scales better in this transition because a restricted game sufficient to achieve a low exploitability can be induced with only a portion of the game's actions. Such a restricted game is much smaller in terms of total game states and thus requires less computation to solve. The number of states in

2-Clone Leduc poker is roughly 50 times that of standard Leduc poker, but when moving from standard to 2-Clone Leduc poker, the number of states in the XDO restricted game increases by a much smaller proportion.

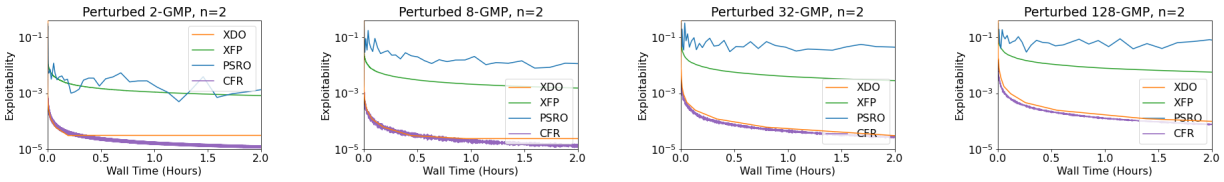


(a) (b) (c)

Extended Figure 4.7: The size, in states, of the restricted game induced by the BR population in XDO in various games.

In our XDO experiments, the final size (in states) of the restricted game in Leduc Poker is roughly 85% of the size of the full game. The final size of the restricted game in 2-Clone Leduc Poker is only approximately 43% that of the full game, so XDO performs proportionally less computation to solve the restricted game meta-NE than it would take to directly solve the full-game NE with all actions considered. XDO also outperforms CFR in Oshi Zumo, where the final size of the restricted game is only half that of the full game.

Empirical Tests on Perturbed k-GMP



Extended Figure 4.8: Exploitability vs wall time hours with XDO, XFP, PSRO, and CFR in Perturbed k-GMP as the number of subgames rises. XDO and CFR scale well even when there are many subgames while PSRO becomes unable to reach a low exploitability.

We run XDO, XFP, PSRO (DO) and CFR on the Perturbed k-GMP games. In perturbed k-GMP, a chance node randomly selects one of the k stage games and the outcome is observed by both players. A static perturbation uniformly sampled between $(-1.0, 1.0)$ is added to every GMP payoff where both players select the same action to create a unique equilibrium solution for each of the k subgames. We see that as k is increased from 2 to 128, XDO and CFR exploitability increases but remains relatively low, but PSRO performance relative to computation time dramatically decreases. For this experiment, we use CFR as the meta-NE solver for XDO.

Tabular Experiment Details

For calculating tabular BRs, we use an oracle implementation supplied by the OpenSpiel framework [61].

For tabular PSRO, we use linear programming to solve each meta-NE. To calculate the empirical payoff matrix, we play 100 games per policy combination.

For CFR$^+$, CFR, MCCFR, and XFP, we use the default implementations and settings provided by OpenSpiel.

For tabular XDO, we repeatedly improve the ϵ-NE (e.g. perform CFR iterations) for the restricted game until both of the following conditions are met: *a)* the exploitability of the meta-NE in the restricted game is less than ϵ, and *b)* the exploitability of the meta-NE in the restricted game is less than the exploitability of the meta-NE in the full game. We initialize ϵ to be 0.35, and each XDO iteration, we set ϵ to $0.98 * \epsilon$.

This has two benefits: First, we guarantee that all BRs added to the population are useful, in that they contain at least some out-of-restricted-game action, and thus expand the restricted game. Second, if the restricted game already contains the actions necessary for a NE in the

full game, we simply continue to improve the meta-NE ad infinitum, rather than add a new BR and restart the meta-NE solver.

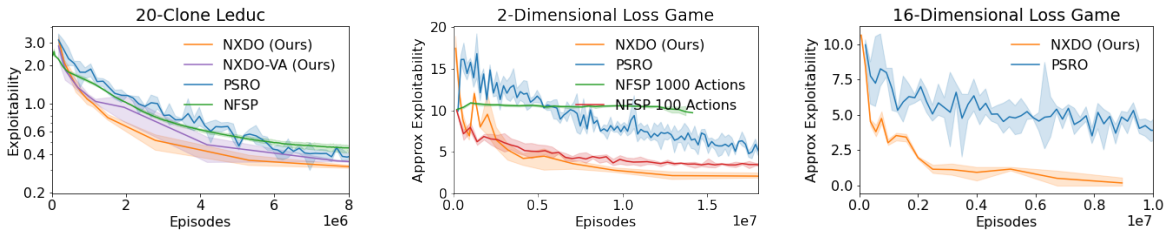
For SDO, we use CFR+ as an inner-loop solver, instead of a linear program as used in the SDO paper [15]. The default pure strategy we use is to choose the first possible action. To find best-response sequences, we compute a tabular BR in the full game.

Unless otherwise stated, all non-deterministic tabular experiments were run with 3 seeds each.

4.6.3 Neural Experiments with NXDO

In Figure 4.9a, we compare the exploitability of NXDO and NXDO-VA with PSRO and NFSP on 20-Clone Leduc poker. DDQN is used to train NXDO and PSRO BRs. Similar to the tabular experiments, we find that NXDO outperforms both methods. However, we find that the margin by which NXDO outperforms these methods is smaller than in tabular experiments. This can be attributed to the large proportion of experience required by NFSP to solve the restricted game relative to experience spent learning BRs. Training details and an analysis on the proportion of experience spent in the inner vs outer loop of NXDO are included in supplementary materials.

We also test NXDO and PSRO on the 2D and 16D continuous-action Loss Game, shown in Figures 4.9b and 4.9c respectively. PPO is used to train NXDO and PSRO BRs. In the 2D action space game, we compare to NFSP with a binned discrete action space. Because calculating exact exploitability is intractable for the Loss Game, for each algorithm checkpoint, we measure approximate exploitability by training a continuous-action PPO best response against it until saturation and measuring the BR's final mean reward. NXDO outperforms NFSP in the 2D game while operating in a continuous action space and outperforms PSRO

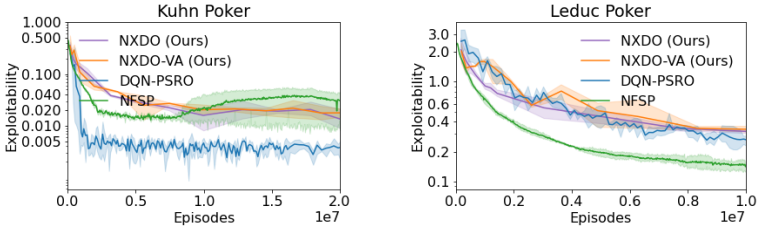


(a) (b) (c)

Extended Figure 4.9: (a) Exploitability in 20-Clone Leduc poker of NXDO, NXDO-VA, PSRO, and NFSP as a function of episodes gathered; (b and c) Approximate exploitability as a function of episodes gathered in the continuous-action Loss Game of NXDO, PSRO, and NFSP (with binned discrete actions).

in the 2D and 16D game which we conjecture is due to more effective use of population strategies in its restricted game.

Neural Methods on Kuhn and Leduc Poker



(a) (b)

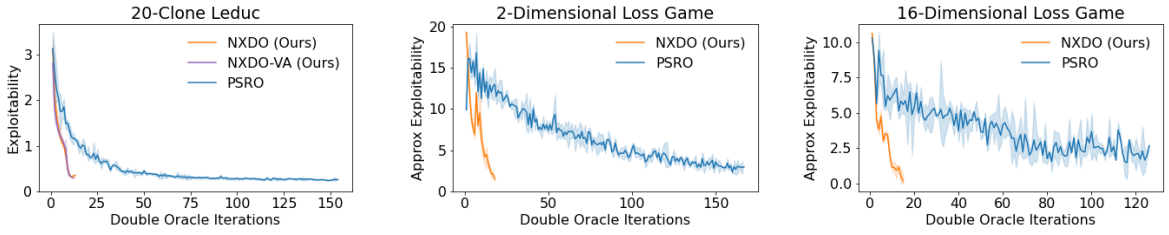
Extended Figure 4.10: NXDO, NXDO-VA, PSRO, and NFSP exploitability vs episodes collected on Kuhn and Leduc Poker

We test NXDO, NXDO-VA, PSRO, and NFSP on Kuhn and Leduc Poker using the same hyperparameters as used with 20-Clone Leduc. In these smaller games, NXDO performs less competitively because the time spent computing the extensive-form meta-NE strategies is large compared to the amount of time needed to train nearly all pure strategies for Kuhn and Leduc. NFSP also reaches an initial low exploitability quickly in part due to the small action-space sizes relative to other games tested.

Additional Loss Game Experiments

We analyze the effect of different stopping condition schedules for the NFSP meta-NE solver when training NXDO on the 16-Dimensional Loss Game. In figure 4.12c we compare a variaty of warm start amounts (the number of initial iterations in which zero time is spent training the meta-NE) and the coefficient at which we increase the number of episodes spent training any meta-NE thereafter. The default parameters are a warm start of 5 and a coefficient of 1.5, thus we spend 5 fixed NXDO iterations with a randomly initialized meta-NE solution, then train our sixth iteration meta-NE for the starting amount of 1e6 episodes, and then train each subsequent meta-NE for 1.5x the number of episodes in the previous iteration. Ablations with other reasonable values considered show that the fine details of such a schedule have only minor effects on performance in the Loss Game.

4.6.4 Costs and Benefits of Extensive-Form Restricted Games

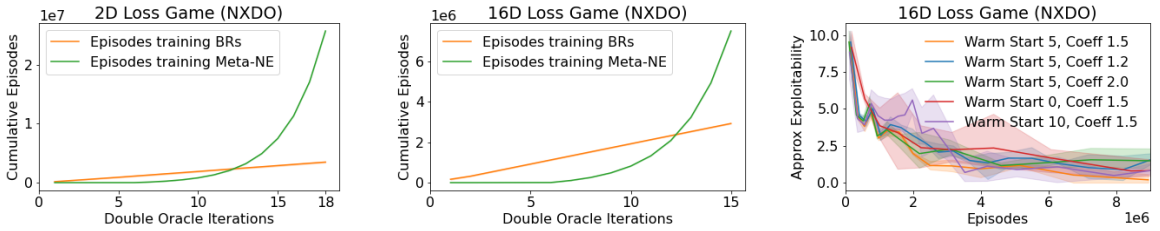


(a) (b) (c)

Extended Figure 4.11: NXDO exploitability in 20-Clone Leduc (a) and approximate exploitability in the Loss Game (b and c) as a function of Double Oracle iterations in NXDO and PSRO.

Shown in figure 4.11, we compare the exploitability of NXDO against that of PSRO in terms of Double Oracle iterations. NXDO acheives a low exploitability in significantly less iterations due to mixing population strategies at every infostate in the game rather than just the at root like is done in PSRO. When both are limited to a small population of behavioral strategies,

the extensive-form restricted game allows for much more expressive power in solving for a meta-NE than a normal-form restricted game does.



(a) (b) (c)

Extended Figure 4.12: (a and b) In NXDO tests on the Loss Game, the cumulative amount of experience in episodes to either train BRs or calculate meta-NE with NFSP as a function of Double Oracle iterations. After the warm start phase, each NXDO iteration, we multiply the amount of episodes we spend training NFSP by a coefficient of 1.5. (c) NXDO ablations on the 16-Dimensional Loss Game. We vary the number of warm start iterations in which we spend zero time training the NFSP meta-NE solver and the coefficient by which we multiply the episodes spent training NFSP each iteration after the warm start.

This additional expressive power in the restricted game does come at an increased computational cost which needs to be taken into account. NXDO is most applicable when the need to mix population strategies at many infostates outweighs the extra computation needed to compute the extensive-form restricted game. Figures 4.12a and 4.12b show the cumulative episodes spent training either BRs or meta-NE vs NXDO iterations. The time spent training a meta-NE is gradually increased each iteration to trade quickly adding BRs for solving the meta-NE at a high accuracy.

4.7 Conclusion

PSRO and NXDO are the only existing game-theoretic deep RL methods that can work on large continuous-action games. In games where the NE must mix in many infostates, but only a small fraction of all actions are in the support of the NE at each infostate, we expect XDO and NXDO to outperform PSRO, because PSRO may require a superlinear, or even

exponential number of pure strategies. We also expect XDO and NXDO to outperform CFR and NFSP, respectively, on discrete-action games where the NE only needs to mix over a small fraction of actions. This is because CFR and NFSP scale poorly with the number of actions in the game, but XDO and NXDO tend to discover a set of relevant actions and ignore actions that are dominated or redundant. We hypothesize that games with these properties are prevalent across a number of domains such as large board games, video games, and robotics applications.

4.8 Computational Costs

Experiments were performed on a shared local computer with 128 CPU-cores, 2 RTX 3090 GPUs, and 256GB of RAM. Due to small network sizes, most neural experiments were performed without GPU acceleration. Neural experiments on 20-Clone Poker and the Loss Game used 8 to 16 cores each and took between 2 and 4 days to complete using 10 to 40GB of RAM. Tabular XDO experiments were run for up to 1 day, using a single core each and between 1 to 10GB of RAM.

4.9 Experiment Code

Code for tabular experiments can be found at `https://github.com/indylab/tabular_xdo`

Code for neural experiments can be found at `https://github.com/indylab/nxdo`

Chapter 5

Conclusion

In this thesis we introduce three projects related to sequential decision making. The first combines deep reinforcement learning with search to solve single-agent planning problems. The second and third projects combine deep reinforcement learning with the game-theoretic double oracle algorithm to efficiently find approximate Nash equilibria in large games. In particular, P2SRO parallelizes PSRO while maintaining convergence guarantees and achieves state-of-the-art performance on Barrage Stratego, and NXDO uses a novel extensive-form double oracle algorithm to achieve state-of-the-art performance on a continuous-action game.

Progress in the field of optimal sequential decision making will have many real-world applications. In particular, with more advanced algorithms we will one day be able to automate many economic activities such as physical labor with robotics and work done on computers with algorithms. As one example of future related work, we are currently applying DeepCubeA to optimize multi-agent path-finding for warehouse robots [2, 3]. In future work I am interested in making reinforcement learning methods more efficient and enabling them to generalize over a wide range of tasks [109, 67, 63].

Two-player zero-sum games have long served as testbeds for artificial intelligence algorithms.

Advances in algorithms have achieved landmark results against humans in checkers, chess, go, and most recently, poker. Additionally, algorithms for two-player zero-sum games have many applications in areas as diverse as cybersecurity, negotiation, robotics, and business strategy. However, there still does not exist a general method that can reliably find Nash equilibria in very large games with potentially continuous action spaces. Such an algorithm would be a major contribution in this field and is what I will be focusing on in future research.

I am also interested in moving beyond two-player games to games with large numbers of players. For example, the internet has enabled many new platforms that match supply and demand through markets. But the design of modern markets is challenging because a designer needs to learn agent preferences from large datasets while considering the strategic behavior of those agents. Learning how to make optimal decisions in the presence of other agents will have many applications in creating markets where humans can better compete and collaborate with each other and with AI algorithms [76]. I have also done work on independent reinforcement learning in Markov Potential Games [33] and plan to continue this line of research.

Finally, I have done work on applications of deep learning in science [4, 36, 6, 5, 75]. I believe there will continue to be an incredible opportunity to use deep learning to make predictions from large data sets in science and I am excited to continue to work in this direction.

Bibliography

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

[2] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, Roy Fox, Marco Valtorta, Biplav Srivastava, and Pierre Baldi. Obtaining approximately admissible heuristic functions through deep reinforcement learning and a* search. 2021.

[3] Forest Agostinelli, Alexander Shmakov, Stephen McAleer, Roy Fox, and Pierre Baldi. A* search without expansions: Learning heuristic functions with deep q-networks. *arXiv preprint arXiv:2102.04518*, 2021.

[4] A Anker, P Baldi, SW Barwick, D Bergman, H Bernhoff, DZ Besson, N Bingefors, O Botner, P Chen, Y Chen, et al. White paper: Arianna-200 high energy neutrino telescope. *arXiv preprint arXiv:2004.09841*, 2020.

[5] A Anker, P Baldi, SW Barwick, J Beise, DZ Besson, S Bouma, M Cataldo, P Chen, G Gaswint, C Glaser, et al. Improving sensitivity of the arianna detector by rejecting thermal noise with deep learning. *arXiv preprint arXiv:2112.01031*, 2021.

[6] A Anker, P Baldi, SW Barwick, J Beise, DZ Besson, S Bouma, M Cataldo, P Chen, G Gaswint, C Glaser, et al. Measuring the polarization reconstruction resolution of the arianna neutrino detector with cosmic rays. *arXiv preprint arXiv:2112.01501*, 2021.

[7] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C Holte. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16-17):2075–2098, 2011.

[8] Karol Gregor Rishabh Kabra Sebastien Racaniere Theophane Weber David Raposo Adam Santoro Laurent Orseau Tom Eccles Greg Wayne David Silver Timothy Lillicrap Victor Valdes Arthur Guez, Mehdi Mirza. An investigation of model-free planning: boxoban levels. https://github.com/deepmind/boxoban-levels/, 2018.

[9] David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech Czarnecki, Julien Perolat, Max Jaderberg, and Thore Graepel. Open-ended learning in symmetric zero-sum games. In *International Conference on Machine Learning*, pages 434–443, 2019.

[10] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. *arXiv preprint arXiv:1710.03748*, 2017.

[11] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[12] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[13] Dimitri P Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996. ISBN 1-886529-10-8.

[14] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[15] Branislav Bosansky, Christopher Kiekintveld, Viliam Lisy, and Michal Pechoucek. An exact double-oracle algorithm for zero-sum extensive-form games with imperfect information. *Journal of Artificial Intelligence Research*, 51:829–866, 2014.

[16] Michael Bowling and Manuela Veloso. Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136(2):215–250, 2002.

[17] George W. Brown. Iterative solution of games by fictitious play. *Activity analysis of production and allocation*, pages 374–376, 1951.

[18] Noam Brown and Tuomas Sandholm. Regret-based pruning in extensive-form games. In *NIPS*, pages 1972–1980, 2015.

[19] Noam Brown and Tuomas Sandholm. Simultaneous abstraction and equilibrium finding in games. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

[20] Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

[21] Noam Brown, Christian Kroer, and Tuomas Sandholm. Dynamic thresholding and pruning for regret minimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[22] Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. In *International Conference on Machine Learning*, pages 793–802, 2019.

[23] Robert Brunetto and Otakar Trunda. Deep heuristic-learning in the rubik's cube domain: an experimental evaluation. 2017.

[24] Adrian Brüngger, Ambros Marzetta, Komei Fukuda, and Jurg Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 90:45–63, 1999.

[25] Jiří Čermák, Branislav Bošansky, and Viliam Lisy. An algorithm for constructing and solving imperfect recall abstractions of large extensive-form games. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 936–942, 2017.

[26] Petros Christodoulou. Soft actor-critic for discrete action settings. *arXiv preprint arXiv:1910.07207*, 2019.

[27] Joseph C Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[28] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[29] Rüdiger Ebendt and Rolf Drechsler. Weighted a search–unifying view and application. *Artificial Intelligence*, 173(14):1310–1342, 2009.

[30] Ariel Felner, Richard E Korf, and Sarit Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[31] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[32] Jakob Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[33] Roy Fox, Stephen McAleer, Will Overman, and Ioannis Panageas. Independent natural policy gradient always converges in markov potential games. *arXiv preprint arXiv:2110.10614*, 2021.

[34] Drew Fudenberg, Fudenberg Drew, David K Levine, and David K Levine. *The theory of learning in games*. The MIT Press, 1998.

[35] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[36] Christian Glaser, Stephen McAleer, Pierre Baldi, and Steven Barwick. Deep learning reconstruction of the neutrino energy with a shallow askaryan detector. *PoS (ICRC2021)*, 1051, 2021.

[37] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[38] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[39] Audrūnas Gruslys, Marc Lanctot, Rémi Munos, Finbarr Timbers, Martin Schmid, Julien Perolat, Dustin Morrill, Vinicius Zambaldi, Jean-Baptiste Lespiau, John Schultz, et al. The advantage regret-matching actor-critic. *arXiv preprint arXiv:2008.12234*, 2020.

[40] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870, 2018.

[41] Eric A Hansen, Daniel S Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In *AAAI*, volume 4, pages 709–715, 2004.

[42] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[43] Sergiu Hart and Andreu Mas-Colell. A simple adaptive procedure leading to correlated equilibrium. *Econometrica*, 68(5):1127–1150, 2000.

[44] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[45] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

[46] Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *International Conference on Machine Learning*, pages 805–813, 2015.

[47] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[48] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[49] Colin G Johnson. Solving the Rubik's cube with learned guidance functions. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 2082–2089. IEEE, 2018.

[50] Sven Jug and Maarten Schadd. The 3rd stratego computer world championship. *Icga Journal*, 32(4):233, 2009.

[51] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[52] Herbert Kociemba. 15-puzzle optimal solver. `http://kociemba.org/themen/fifteen/fifteensolver.html`.

[53] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[54] Richard E Korf. Macro-operators: A weak method for learning. *Artificial intelligence*, 26(1):35–77, 1985.

[55] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, AAAI'97/IAAI'97, pages 700–705. AAAI Press, 1997. ISBN 0-262-51095-2. URL `http://dl.acm.org/citation.cfm?id=1867406.1867515`.

[56] Richard E Korf. Linear-time disk-based implicit graph search. *Journal of the ACM (JACM)*, 55(6):26, 2008.

[57] Richard E Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial intelligence*, 134(1-2):9–22, 2002.

[58] Harold William Kuhn and Albert William Tucker. *Contributions to the Theory of Games*, volume 2. Princeton University Press, 1953.

[59] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael H Bowling. Monte carlo sampling for regret minimization in extensive games. In *NIPS*, pages 1078–1086, 2009.

[60] Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4190–4203, 2017.

[61] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. OpenSpiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019. URL `http://arxiv.org/abs/1908.09453`.

[62] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019.

[63] John B Lanier, Stephen McAleer, and Pierre Baldi. Curiosity-driven multi-criteria hindsight experience replay. *arXiv preprint arXiv:1906.03710*, 2019.

[64] Hui Li, Kailiang Hu, Zhibang Ge, Tao Jiang, Yuan Qi, and Le Song. Double neural counterfactual regret minimization. *arXiv preprint arXiv:1812.10607*, 2018.

[65] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[66] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062, 2018.

[67] Litian Liang, Yaosheng Xu, Stephen McAleer, Dailin Hu, Alexander Ihler, Pieter Abbeel, and Roy Fox. Temporal-difference value estimation via uncertainty-guided soft updates. *arXiv preprint arXiv:2110.14818*, 2021.

[68] Peter Lichodzijewski and Malcolm Heywood. The rubik cube and gp temporal sequence learning: an initial study. In *Genetic Programming Theory and Practice VIII*, pages 35–54. Springer, 2011.

[69] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, pages 6379–6390, 2017.

[70] Somdeb Majumdar, Shauharda Khadka, Santiago Miret, Stephen Mcaleer, and Kagan Tumer. Evolutionary reinforcement learning for sample-efficient multiagent coordination. In *International Conference on Machine Learning*, pages 6651–6660. PMLR, 2020.

[71] Stephen McAleer, John Lanier, Pierre Baldi, and Roy Fox. Xdo: A double oracle algorithm for extensive-form games. *Advances in Neural Information Processing Systems*, 34.

[72] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with approximate policy iteration. In *International Conference on Learning Representations*, 2018.

[73] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with approximate policy iteration. *International Conference on Learning Representations (ICLR)*, 2019.

[74] Stephen McAleer, JB Lanier, Kevin Wang, Roy Fox, and Pierre Baldi. Pipeline psro: A scalable approach for finding approximate nash equilibria in large games. *Advances in Neural Information Processing Systems*, 33:20238–20248, 2020.

[75] Stephen McAleer, Alexander Fast, Yuntian Xue, Magdalene J Seiler, William C Tang, Mihaela Balu, Pierre Baldi, and Andrew W Browne. Deep learning–assisted multiphoton microscopy to reduce light exposure and expedite imaging in tissues with high and low light sensitivity. *Translational vision science & technology*, 10(12):30–30, 2021.

[76] Stephen McAleer, John Lanier, Michael Dennis, Pierre Baldi, and Roy Fox. Improving social welfare while preserving autonomy via a pareto mediator. *arXiv preprint arXiv:2106.03927*, 2021.

[77] H Brendan McMahan, Geoffrey J Gordon, and Avrim Blum. Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 536–543, 2003.

[78] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, Oct 1993. ISSN 1573-0565. doi: 10.1007/BF00993104. URL `https://doi.org/10.1007/BF00993104`.

[79] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[80] Paul Muller, Shayegan Omidshafiei, Mark Rowland, Karl Tuyls, Julien Perolat, Siqi Liu, Daniel Hennes, Luke Marris, Marc Lanctot, Edward Hughes, et al. A generalized training approach for multiagent learning. *International Conference on Learning Representations (ICLR)*, 2020.

[81] Allen Newell and Herbert Alexander Simon. GPS, a program that simulates human thought. Technical report, RAND CORP SANTA MONICA CALIF, 1961.

[82] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[83] Laurent Orseau, Levi Lelis, Tor Lattimore, and Théophane Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems*, pages 3201–3211, 2018.

[84] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1 (3-4):193–204, 1970.

[85] Martin L Puterman and Moon Chirl Shin. Modified policy iteration algorithms for discounted markov decision problems. *Management Science*, 24(11):1127–1137, 1978.

[86] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485*, 2018.

[87] Tomas Rokicki. God's number is 26 in the quarter-turn metric. `http://www.cube20.org/qtm/`, Aug 2014.

[88] Tomas Rokicki. cube20. `https://github.com/rokicki/cube20src`, 2016.

[89] Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. The diameter of the rubik's cube group is twenty. *SIAM Review*, 56(4):645–670, 2014.

[90] Mehdi Samadi, Ariel Felner, and Jonathan Schaeffer. Learning from multiple heuristics. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, 2008.

[91] Maarten Schadd and Mark Winands. Quiescence search for stratego. In *Proceedings of the 21st Benelux Conference on Artificial Intelligence. Eindhoven, the Netherlands*, 2009.

[92] Jaap Scherphuis. The mathematics of Lights Out. `https://www.jaapsch.net/puzzles/lomath.htm`, 2015.

[93] Martin Schmid, Matej Moravcik, and Milan Hladik. Bounding the support size in extensive form games with imperfect information. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

[94] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[95] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[96] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[97] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 10 2017. ISSN 0028-0836. doi: 10.1038/nature24270. URL `http:https://doi.org/10.1038/nature24270`.

[98] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[99] Robert J Smith, Stephen Kelly, and Malcolm I Heywood. Discovering rubik's cube subgroups using coevolutionary gp: A five twist experiment. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 789–796. ACM, 2016.

[100] Sriram Srinivasan, Marc Lanctot, Vinicius Zambaldi, Julien Pérolat, Karl Tuyls, Rémi Munos, and Michael Bowling. Actor-critic policy optimization in partially observable multiagent environments. In *Advances in neural information processing systems*, pages 3422–3435, 2018.

[101] Eric Steinberger. Single deep counterfactual regret minimization. *arXiv preprint arXiv:1901.07621*, 2019.

[102] Eric Steinberger, Adam Lerer, and Noam Brown. Dream: Deep regret minimization with advantage baselines and model-free learning. *arXiv preprint arXiv:2006.10410*, 2020.

[103] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[104] Oskari Tammelin. Solving large imperfect information games using CFR+. *CoRR*, abs/1407.5042, 2014. URL `http://arxiv.org/abs/1407.5042`.

[105] Peter D Taylor and Leo B Jonker. Evolutionary stable strategies and game dynamics. *Mathematical biosciences*, 40(1-2):145–156, 1978.

[106] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[107] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[108] Christopher Makoto Wilt and Wheeler Ruml. When does weighted a* fail? In *SOCS*, pages 137–144, 2012.

[109] Yaosheng Xu, Dailin Hu, Litian Liang, Stephen McAleer, Pieter Abbeel, and Roy Fox. Target entropy annealing for discrete soft actor-critic. *arXiv preprint arXiv:2112.02852*, 2021.

[110] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Advances in neural information processing systems*, pages 1729–1736, 2008.