Toward Scalable Reinforcement Learning via Massive Batching

Sotetsu Koyamada

A dissertation submitted for the degree of

Doctor of Informatics

Kyoto University

2024

Supervised by:
Prof. Shin Ishii
Examination committee:
Prof. Shin Ishii (Chair)
Prof. Hidetoshi Shimodaira
Prof. Jun Morimoto

Abstract

Reinforcement learning (RL) has demonstrated performance comparable to or even surpassing human capabilities in various domains. However, RL is notorious for requiring a huge number of samples for learning. This dissertation is motivated by the observation that scalable algorithms, which efficiently leverage computation, are essential for the progress of artificial intelligence (AI). It explores scalable RL through straightforward vectorization of sample generation on modern accelerators such as GPUs. We verify the applicability and effectiveness of vectorized sample generation in RL. This dissertation primarily focuses on classic game environments due to their historical importance as benchmarks in AI development and the challenges posed by their complex branching structures, which seem *incompatible* with batched RL.

We demonstrate that batching is effective even in discrete sequential game environments, which are often considered *unsuitable* for this technique. We found that even in such classic game environments, simulators running on GPUs can be at least 10 times faster than existing implementations that use threading or multiprocessing on CPUs. Next, we demonstrate the effectiveness of our approach in bridge bidding AI benchmarks. Specifically, our approach surpasses existing benchmarks in bridge bidding through a simple combination of existing techniques, indicating the potential of batching approaches in RL. Finally, we consider a potential disadvantage of batching in RL. The delayed feedback in batched RL can potentially lead to performance degradation compared to the sequential approach, as agents have the reduced adaptability. However, we show that a pure exploration algorithm, Sequential Halving (SH),

does not degrade performance under realistic conditions. We also empirically demonstrate the effectiveness of batched SH when combined with the Monte Carlo tree search algorithm. Overall, this dissertation verifies that the application range of fully vectorized RL on accelerators is wider than expected and demonstrates its effectiveness with appropriate algorithms.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Prof. Shin Ishii, for his invaluable guidance and support throughout my research. There were times when I struggled with writing my dissertation, but his unwavering trust in me enabled me to complete it. I am equally grateful to Prof. Hidetoshi Shimodaira and Prof. Jun Morimoto for their invaluable guidance and insightful feedback during the review of my dissertation. Their thoughtful suggestions have greatly contributed to improving the quality of this work. I am deeply thankful to my collaborators, Soichiro Nishimori, Keigo Habara, Shinri Okano, Yu Murata, Haruka Kita, and Nao Goto, for their dedicated contributions. Without their assistance, I would not have been able to accomplish such extensive development. I am grateful to Dr. Ken Nakae, who occasionally encouraged me during the challenging moments of my PhD journey. I am thankful to Yuki Shiojiri, whose cheerful words and casual conversations often lifted my spirits and lightened my mood during this journey. Lastly, I extend my heartfelt appreciation to my wife, Aika, for her unwavering support and encouragement. Her constant support gave me the strength to persevere, and without her, I might have given up on completing my doctoral dissertation.

Sotetsu Koyamada

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
ASH	Advance-first Sequential Halving
BSH	Breadth-first Sequential Halving
DDS	Double Dummy Solver
FSP	Fictitious Self-Play
GPU	Graphics Processing Unit
IMP	International Match Point
JIT	Just-In-Time
LHS	Left-hand Side
LLM	Large Language Model
MCTS	Monte Carlo Tree Search
MIMD	Multiple Instruction, Multiple Data
MLP	Multi-Layer Perceptron
NN	Neural Network
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
RHS	Right-hand Side
SE	Standard Error
SH	Sequential Halving
SIMD	Single Instruction, Multiple Data
SL	Supervised Learning
SOTA	State-of-the-Art
TPU	Tensor Processing Unit

Related Publications

This dissertation is based on the following publications:

- S. Koyamada, K. Habara, N. Goto, S. Okano, S. Nishimori, and S. Ishii.
  Mjx: A framework for Mahjong AI research.
  In *IEEE Conference on Games (CoG)*, 2022.

- S. Koyamada, S. Okano, S. Nishimori, Y. Murata, K. Habara, H. Kita, and S. Ishii.
  Pgx: Hardware-Accelerated Parallel Game Simulators for Reinforcement Learning.
  In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

- S. Koyamada, S. Nishimori, and S. Ishii.
  A Batch Sequential Halving Algorithm without Performance Degradation.
  Reinforcement Learning Journal (RLJ), 2024.

- H. Kita*, S. Koyamada*, Y. Yamaguchi, and S. Ishii.
  A Simple, Solid, and Reproducible Baseline for Bridge Bidding AI.
  In *IEEE Conference on Games (CoG)*, 2024.

*Equal contributions.

Chapter 1

Introduction

We start this dissertation with a quote from Richard S. Sutton, who is known as the father of reinforcement learning (RL):

> *"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin." — Sutton [2019], "The Bitter Lesson."*

The innovation that best embodies his words may be the appearance of AlexNet in the ImageNet classification [Krizhevsky et al., 2012]. This model dramatically improved the accuracy of image recognition by using large-scale learning with GPUs. It surpassed human-crafted feature-based methods such as SIFT and opened a new era of image recognition by *deep learning* [Goodfellow et al., 2016]. The "scaling law" [Kaplan et al., 2020] in large language models (LLMs) also supports Sutton's words. LLMs built using the Transformer architecture [Vaswani et al., 2017], such as GPT-3 [Brown et al., 2020] and GPT-4 [OpenAI, 2023], seem to scale with model size, data size, and the amount of computation. Given his words, this dissertation is about *leveraging computation* in Artificial Intelligence (AI) research.

Specifically, our focus is on RL [Sutton and Barto, 2018]. Unlike supervised learning (SL), RL agents can generate training samples by themselves through interaction with the environment, the target world where the agent learns to maximize the expected

future return. RL has the potential to realize AI with capabilities that essentially outperforms humans, rather than mere imitation of humans. Current RL algorithms have demonstrated their potential through remarkable successes such as AlphaGo [Silver et al., 2016], the first AI who defeated a top-human Go player, the discovery of faster matrix multiplication algorithms [Fawzi et al., 2022] and sorting algorithms [Mankowitz et al., 2023], and Tokamak control [Degrave et al., 2022]. However, they often require a vast number of samples and are recognized as slow and inefficient compared to humans. To alleviate this problem, two approaches are considered:

- (1) Improve *sample efficiency* (i.e., performance gain per sample). This approach has been well studied in model-based RL — in which the model of the environment is learned to reduce interaction with the real environment (see Sutton and Barto [2018]).

- (2) Improve *sample generation speed* (i.e., samples per time). This is not an elegant approach, but the idea is that learning progresses fast as long as a large number of samples can be generated, even if the sample efficiency is poor.

In this dissertation, we focus on the *less elegant* approach (2) and aim to *just leverage computation* to improve the sample generation speed of RL.

To leverage computation in RL, let us revisit the success of AlexNet [Krizhevsky et al., 2012]. What was the mechanism to leverage computation behind the success of AlexNet? One of the most critical elements is *batching* the training and utilizing GPUs for efficient computation. GPUs were originally developed for graphics processing, but they have been used for various computations as GPGPU (General-Purpose computing on Graphics Processing Units). Nowadays, training deep learning models on GPUs is a standard practice. Given the revolution in AlexNet by batching on GPU accelerators, in this dissertation, we focus on leveraging the computation in RL by (massive) batching the sample generation process, fully utilizing the accelerator hardware.

Recent studies [Makoviychuk et al., 2021, Freeman et al., 2021, Hessel et al., 2021]

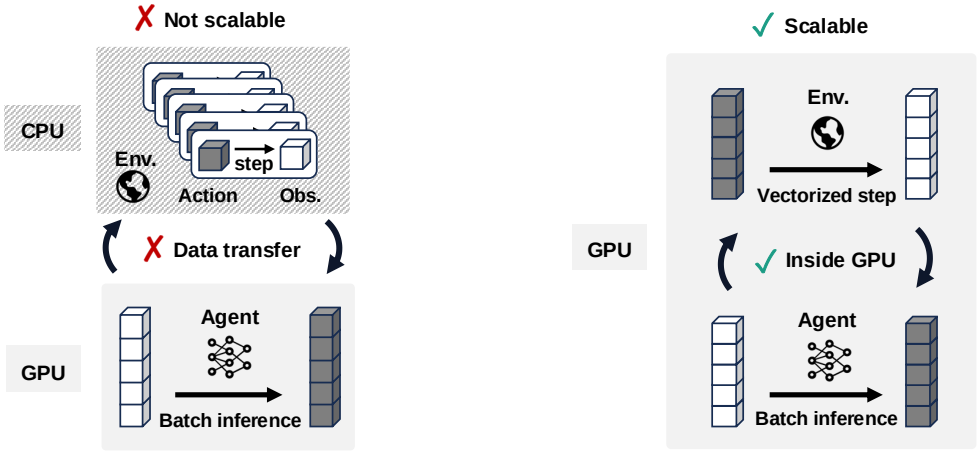


Figure 1.1: Comparison of conventional sample generation scheme in RL (Left) and fully vectorized sample generation on a GPU accelerator (Right).

proposed to generate sample sequences entirely on GPU accelerators (Figure 1.1). This approach has the following advantages:

- As the environment step, which is usually performed on the CPU, is also performed on the GPU, data transfer between the CPU and GPU is unnecessary, improving efficiency.

- By executing vectorized environment steps on the GPU, it is possible to scale to a large batch size, improving the scalability.

These approaches have been validated to be effective in tasks with continuous action spaces using physics simulators [Makoviychuk et al., 2021, Freeman et al., 2021] and simple environments such as classical control [Lange, 2022]. However, vectorized environment steps may not be applicable depending on the nature of the environment. In particular, vectorized environment steps are not suitable for environments with many control structures such as conditional branching.

We consider the applicability and effectiveness of RL architectures using vectorized environment steps on the accelerators. Therefore, we chose (classic) game environments as benchmarks (Figure 1.2). There are several reasons for this choice. First,

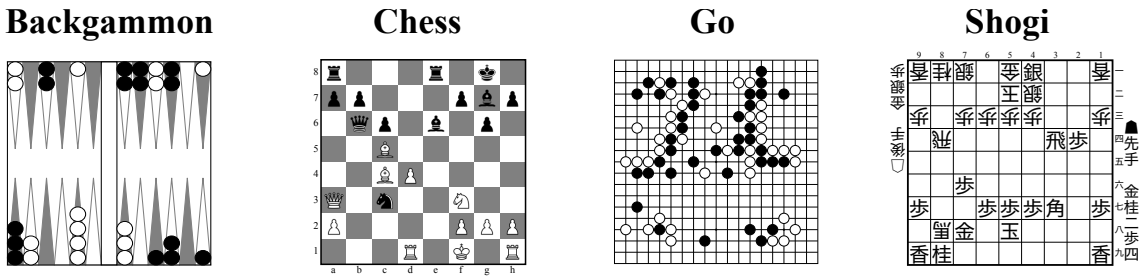


Figure 1.2: Classic game environment examples.

(classic) games have been considered as important milestones in AI research [Tesauro, 1995, Mnih et al., 2015, Silver et al., 2016, 2017, 2018, Brown and Sandholm, 2018, 2019, Berner et al., 2019, Vinyals et al., 2019, Li et al., 2020a]. Second, such games with large discrete state and action spaces seem to be *unsuitable* with such vectorized environment steps due to the sequential nature of the environments with complex branching structures. If we can demonstrate that the batch environment step approach is effective even in environments that seem to be incompatible with, we can expand the range of applicability of this approach.

Our contributions in this dissertation are as follows:

- We demonstrated that the sample sequence generation with vectorized environment step is more efficient than existing appoach (e.g., multi-threading or multi-processing) in classic game environments by a large margin. These games include Chess, Shogi, and Go, which are considered as important benchmarks in AI research and also have a large discrete state and action spaces. Their environment steps include complex branching structures, which are not suitable for vectorized environment steps. However, we showed that the vectorized approach on accelerators is effective even in such game environments, expanding the range of applicability of this approach than previously thought (Chapter 3).

- As a demonstration of RL with vectorized sequence generation in GPU accelerators, we achieved performance that significantly outperforms existing benchmarks in bridge bidding AI. Our method is a combination of simple existing methods,

except that it uses the vectorized sequence generation. This demonstration shows the actual effectiveness of this approach (Chapter 4)

- We also discuss a potential disadvantage of massive batching approach. We point out that batching may degrade performance compared to sequential processing due to delayed feedback. For example, comparing (1) the case of performing 100K sequential environment steps, observing a reward each step and adapting the policy sequentially, and (2) the case of performing 20 vectorized environment steps with 5K batch size each. The total number of steps is the same in both cases, but in the latter case, the agent has only 20 opportunities to observe the reward and update the policy, which may lead to the inferior performance. We consider this problem in the context of the pure exploration problem of the stochastic bandit problem, which is the simplest class of RL. We found that a natural variant of the popular Sequential Halving (SH) algorithm [Karnin et al., 2013] does not degrade performance under realistic assumptions — the arm selection is exactly the same in the two cases mentioned above. We also experimentally verified that SH is robust to batching in more realistic conditions, such as applying it to Monte Carlo tree search (Chapter 5).

Chapter 2

Preliminary

In this chapter, we explain the minimum background knowledge necessary to understand this dissertation.

2.1 Reinforcement Learning (RL)

Reinforcement learning (RL) is a framework for solving decision-making problems [Sutton and Barto, 2018]. In this dissertation, we consider RL problems with discrete state space $\mathcal{S}$ and action space $\mathcal{A}$. Each episode starts with an initial state $s_0 \in \mathcal{S}$ given by $s_0 \sim p_0(s_0)$, where p_0 is the (unknown) initial state distribution. The agent and the environment interact as follows until the state reaches a terminal state.

- Given the current state s_t, the agent selects an action $a_t \in \mathcal{A}$ according to the policy $\pi(a_t|s_t)$.
- Given the action a_t from agent, the environment returns a reward by the (unknown) reward function: $r(s_t, a_t) : \mathcal{S} \times \mathcal{A} \to [R_{\min}, R_{\max}]$, where $R_{\min}$ and $R_{\max}$ are the given minimum and maximum reward, respectively.
- Also, the environment returns the next state $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$, where p is the (unknown) state transition kernel.

We assume that this interaction reaches a terminal state in finite time, and the episode ends. Note that we assume the Markov property, which means that the current state s_t contains enough information for the next state transition and for decision-making.

Value and objective. We define the value of a state $s \in \mathcal{S}$ under the policy π as the expected cumulative rewards given the state:

$$v^\pi(s) \coloneqq \mathbb{E}_{p,\pi}\left[\sum_{t\geq 0} r(s_t, a_t) \,\middle|\, s_0 = s\right].$$

Here, we omitted the discount factor $\gamma < 1$ for simplicity. In this dissertation, we focus only on the finite-horizon problem, assuming that the episode ends in finite time. Thus, note that the cumulative rewards is bounded even without introducing a discount factor. The agent's goal is to find a policy π that maximizes the expected cumulative rewards

$$J(\pi) \coloneqq \mathbb{E}_{p_0}\left[v^\pi(s_0)\right].$$

This problem can be solved by approximated dynamic programming when the number of states is small (see Sutton and Barto [2018]). However, real-world problems have a large number of states and such solutions are infeasible.

2.2 Deep RL

In RL, it has been common to represent the policy and value to be learned by function approximators when the state space is large. Neural Networks (NNs) has been a popular choice as function approximators since 1990s [Tesauro, 1995], and the success of deep learning in ImageNet [Krizhevsky et al., 2012] and its application to the Atari 2600 benchmark [Mnih et al., 2015] makes *deep* RL accelerated. The further potential of deep RL has been demonstrated by (but not limited to) AlphaGo [Silver et al.,

2016], which first defeated top professional Go human players, and application to the LLMs [OpenAI, 2023].

To grasp the outline of deep RL and make it easier to understand the motivation of this dissertation, we provide a high-level overview of the PPO algorithm [Schulman et al., 2017], the most popular mode-free policy optimization algorithm in deep RL to date (Algorithm 1). While PPO is popular as shown by the application to the LLMs [OpenAI, 2023], its implementation details are known to be complex [Huang et al., 2022a]. Since the subject of this dissertation is not related to the implementation details of PPO, we do not provide a detailed explanation. For implementation details, please refer to the original paper [Schulman et al., 2017] and the *37* implementation details of PPO [Huang et al., 2022a].

Algorithm 1 High-level overview of PPO [Schulman et al., 2017]

1: **initialize** N vectorized environments, initial policy θ, and batch size B.
2: **for** iteration $= 1, \ldots$ **do**
3: Generate N rollouts for T timesteps from the N environments using policy θ.
4: Update policy θ **for** K epochs and **for** $\lfloor \frac{NT}{B} \rfloor$ minibatches in each epoch.

High-level overview of PPO (Algorithm 1). As other deep RL algorithms, PPO represents the policy and value function by neural networks, whose *torso* is shared by them. We denote the parameters of these neural networks by θ. The torso is connected to *policy head* and *value head* which output the policy $\pi_\theta(a_t|s_t)$ and value $v_\theta(s_t)$, respectively. Given the current parameter θ, PPO generates N rollouts for T timesteps by interacting with the N parallel environments. Then, PPO scans the generated samples for K epochs as minibatches. For each minibatch, PPO computes the loss $\mathcal{L}$ and updates the parameters θ to minimize the loss using the stochastic optimization method, Adam [Kingma and Ba, 2015]. Here, $\mathcal{L}$ is a linear combination of the surrogate loss $\mathcal{L}_{\text{policy}}$, value loss $\mathcal{L}_{\text{value}}$, and entropy loss $\mathcal{L}_{\text{entropy}}$. The surrogate loss $\mathcal{L}_{\text{policy}}$ aims to maximize the expected *benefit* of an action (advantage) by validating multi-epoch updates through the use of a clipped policy ratio. The value loss $\mathcal{L}_{\text{value}}$

is the clipped mean squared error between the prediction and the target value. The entropy loss $\mathcal{L}_{\text{entropy}}$ is the negative entropy of the policy, which encourages exploration by maintaining the policy's stochasticity. Note that the training (gradient updates) of neural networks are typically performed in batches on accelerators such as GPUs or TPUs.

2.3 Deep RL architectures

As shown in the high-level overview of PPO, roughly speaking, deep RL repeats the following two steps: (1) generating sample sequences by the agent-environment interaction and (2) learning the agent's neural network by gradient updates. The bottleneck in deep RL is typically not the gradient updates but the sample generation, especially the parallel environment steps. For example, Weng et al. [2022b] reported that environment steps are the bottleneck in the PPO training on Atari with naive parallelization. In addition, deep RL is known to be sample-inefficient, requiring a large number of samples for learning. Therefore, previous studies have focused on parallelizing environment execution efficiently [Weng et al., 2022b, Makoviychuk et al., 2021, Freeman et al., 2021]. In this section, we categorises the parallelizing architectures in deep RL into two approaches based on how to parallelize the environment execution:

- **MIMD** (Multiple Instruction, Multiple Data), which gives different instructions to parallel environments, enabling flexible computation.
- **SIMD** (Single Instruction, Multiple Data), which shares the same instruction to parallel environments, thus, is limited in flexibility but enables scalable computation.

We use the terms MIMD and SIMD from Flynn's taxonomy in computer science. These concepts are particularly important in computer architecture [Hennessy and Patterson, 2017]. We borrow these concepts to classify parallelization in the current deep RL architectures into two categories.

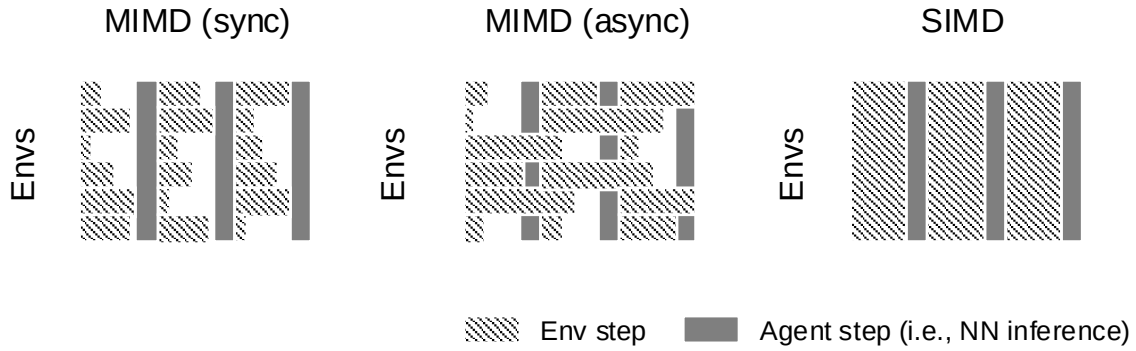


Figure 2.1: MIMD and SIMD parallelization in environment execution. The number of environments is six for all cases. The batch size of NN inference (i.e., agent step) is six for MIMD (sync) and SIMD, and four for MIMD (async). MIMD (async) requires that the number of environments is larger the batch size of training for efficient CPU utilization.

2.3.1 MIMD parallelization in RL

In the parallel execution of the environments by MIMD, each environment independently executes the environment step when it receives an action from the agent. Typically, each environment step is executed by a thread or a process; C++ threading or Python multiprocessing is a common choice. This environment step has a variation in execution time for each thread or process. On the other hand, the agent step with NN is performed in batches on a GPU or TPU accelerator, except for a few exceptions such as A3C [Mnih et al., 2016]. Note that this batch NN inference is executed by SIMD. A key observation here is that, unlike the agent step with NN, the execution time of the environment step varies for each thread or process, leading to a potentially low CPU utilization. Therefore, in addition to the naive synchronous implementation that waits for all environment steps to finish, asynchronous implementations have been proposed (Figure 2.1). In asynchronous implementations, as environment steps are executed on the CPUs, and the agent steps (NN inferences) are executed on the GPUs, some studies proposed to execute them on different machines [Espeholt et al., 2018, 2020], while others execute them on the same machine [Weng et al., 2022b].

A MIMD parallelization example. As an example of parallelization in RL by MIMD in the current computer environment, we consider parallelization in the game of Mahjong, one of the complex game environments. Mahjong requires determining the winning hand based on the combination of 14 hand tiles, making the environment logic complex. The environment step written in a scripting language like Python is significantly slower than that in a performance-oriented language like C++, making the environment step on the CPU a bottleneck. On the other hand, Python is the *lingua franca* of machine learning. Therefore, it is common to implement the environment step in a fast language like C++ and call it from Python. For example, the Mahjong environment written mainly in C++[1](Mjx) is about 100x faster than Mjai, the environment written in the Ruby scripting language[2] (Table 2.1). In addition, the MIMD (async) implementation is typically a distributed RL algorithm such as IMPALA [Espeholt et al., 2018] and SEED [Espeholt et al., 2020]. To support such distributed RL algorithms, a distributed environment such as a server and client is required, and an implementation using gRPC[3], for example, is common as in Mjx. Such a distributed environment eliminates the bottleneck of the environment step with (much) larger number of environments than the batch size. The Mahjong AI Suphx [Li et al., 2020a] uses such a distributed system and achieve better performance than 99.99% of ranked human players on the online leaderboard. However, the distributed system introduces significant complexity in the implementation.

Table 2.1: Runtime in Mahjong environments. Players are pass agent.

	C++ (Mjx)	Ruby (Mjai)
sec/game	0.0874	12.4131

[1]Mjx. `https://github.com/mjx-project/mjx`
[2]Mjai. `https://github.com/gimite/mjai`
[3]`https://grpc.io/`

2.3.2 SIMD parallelization in RL

In SIMD-based parallel environment execution, all parallel environments work with a sequence of the same instructions. Therefore, unlike MIMD, there is no variation in the execution time of the environment step. One can think of this as aligning the steps with the slowest environment step among all environments. On the other hand, the accelerators that can efficiently process SIMD instructions[4], such as GPUs, may be able to execute the environment step efficiently. SIMD is supposed to be limited in flexibility [Weng et al., 2022b, Hessel et al., 2021] but enables scalable computation if the environment step is compatible with SIMD instructions. In particular, in environments that use physics simulators, SIMD parallelization has been shown to be effective because matrix operations are central [Makoviychuk et al., 2021, Freeman et al., 2021]. In this dissertation, we focus on parallelizing RL in the SIMD approach, which we refer to as *massive batching*, and discuss its applicability, effectiveness, and limitations.

A SIMD parallelization example. As an example of SIMD programming, we show SIMD programming in JAX [Bradbury et al., 2018]. JAX is a framework in Python, the *lingua franca* of AI research in recent years. JAX provides an API compatible with NumPy [Harris et al., 2020] and the JAX code can be JIT (Just-In-Time)-compiled and optimized using XLA[5]. In JAX, SIMD programming is performed using pure functions. A function is pure function if it has no side effects and it guarantees the same output for the same input. To perform efficient parallel processing, it is important that individual calculations do not interfere with each other. Pure functions always return the same result for the same input and do not depend on external states, so they can avoid conflicts and synchronization problems caused by parallelization. Figure 2.2 shows an example of SIMD programming in JAX. JAX can automatically vectorize pure functions using `vmap`. This feature allows us to achieve SIMD parallelization by describing the environment transition in RL as a pure function.

[4]Note that modern GPU architectures adopt SIMT/SIMD architecture rather than SIMD.
[5]`https://github.com/openxla/xla`

```
import jax
# state : vectorized (e.g., 1024 Go boards)
# action: vectorized (e.g., 1024-dimensional vector)
state = jax.vmap(env.step)(state, action)
```

Figure 2.2: SIMD programming example in JAX. JAX can automatically vectorize the function using `vmap` if the `step` function is a pure function. JIT-compilation is also needed for efficient execution.

2.3.3 Limitation in SIMD parallelization

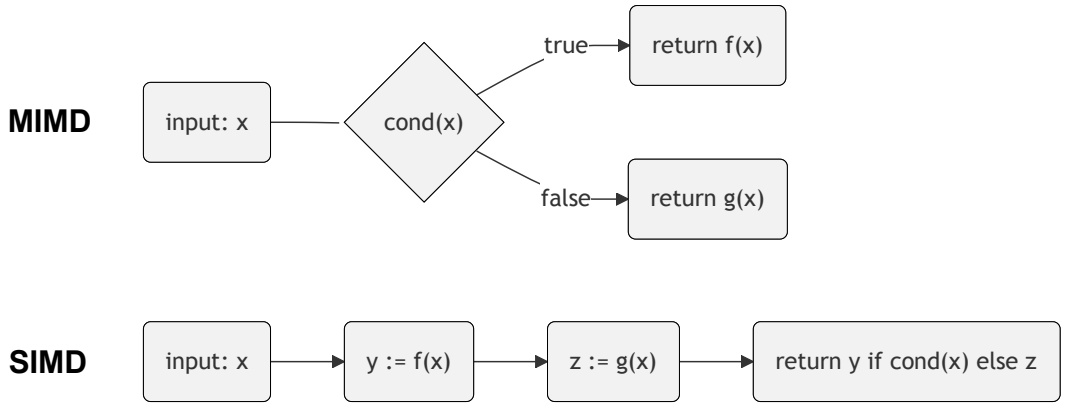


Figure 2.3: A example of branching in SIMD parallelization.

SIMD-based parallel environment execution requires that the environment step can be efficiently executed with SIMD instructions. Examples of suitable processing include matrix operations in physics simulators. On the other hand, in environments with brahches, parallelized environments become heterogeneous, making SIMD parallelization inefficient. This is because SIMD needs to evaluate all branches of the dynamically branching process (Figure 2.3); otherwise, it cannot process all data with a single operation. Game environments are a prominent example of environments that include such branches. For example, the game of Go is a popular benchmark for AI development but it has a large branching factor (Figure 2.4): Given the current state of the board and the action, the processing branches depending on whether the action is a pass or not. If the pass is consecutive, the process calculates the scores. If the action is not a pass, the process further branches depending on whether the action is

a suicide move or not and whether the opponent's stones are captured or not. Note that some of these procedures involve dynamic processing. For example, determining whether stones are captured requires tracing connected stones. These observations and the fact that SIMD programming with conditional branching (basically) needs to evaluate all branches demonstrate that complex state space environments like the game of Go are not suitable for SIMD-based approaches. In Chapter 3, however, we demonstrate that even in such environments that seem *unsuitable* for SIMD approaches, the SIMD approach is more efficient than the existing tuned MIMD approach in modern computational frameworks.

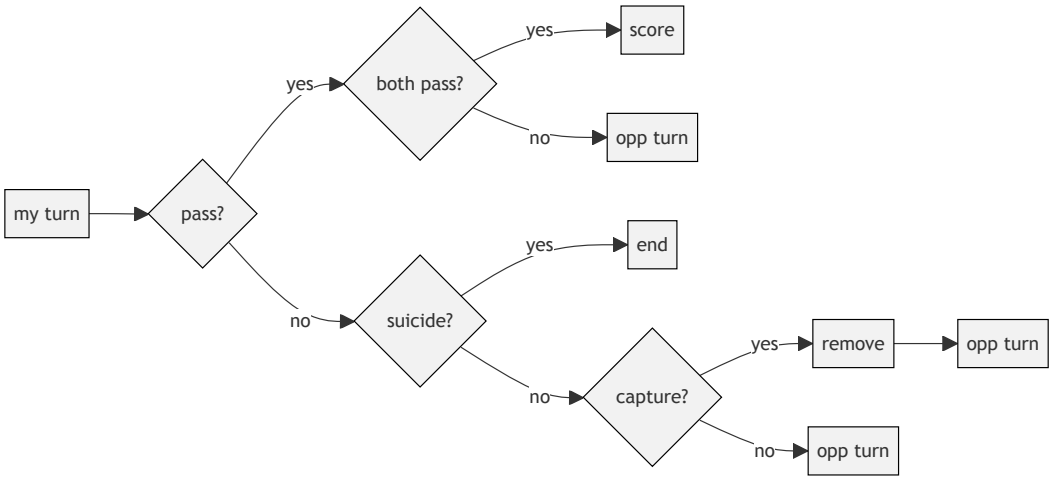


Figure 2.4: A simplified diagram of the environment step in the game of Go. For example, the calculation of the score and the determination of whether stones are killed depend dynamically on the state of the surrounding positions of the placed stone and may require scanning the entire board at worst, making it unsuitable for SIMD parallelization.

2.4 Multi-agent RL

As some of the environments we consider in this dissertation are multi-agent RL (MARL) problems, we describe the formulation of problems in MARL. There are various ways to formulate multi-agent environments, but since all the environments we consider in this dissertation are turn-based game environments like Chess and Go, we can for-

mulate them simply as follows. Each state contains information about which agent should act in that state, and the selected agent chooses an action based on this indicator. The observation function and reward function are defined independently for each agent, and each agent learns a policy that maximizes the expected cumulative reward independently. Note that this is essentially equivalent to the definition of Agent Environment Cycle (AEC) games [Terry et al., 2021]. Naively, for a given agent, we can treat the problem as a normal RL problem by fixing the policies of other agents and incorporating them into the environment.

Chapter 3

Pgx: Massively Vectorized Game Simulators on Accelerators

In this chapter, we demonstrate the effectiveness of massively vectorized environment steps on GPU accelerators in classic game environments, which are seemingly *unsuitable* for batching due to their complex branching structures. We believe that this shows that the application range of massively batching approaches is wider than generally thought.

3.1 Introduction

Developing algorithms for solving challenging games is a standard artificial intelligence (AI) research benchmark. Especially building AI, which can defeat skilled professional players in complex games like chess, shogi, and Go, has been a vital milestone. Though reinforcement learning (RL) algorithms, which combine deep learning and tree search, are successful in obtaining such high-level strategies [Silver et al., 2016, 2017, 2018], complex games like chess and Go are still in the interest of AI research for developing RL and search algorithms in discrete state environments.

On the other hand, studying algorithms for solving large state-space games such as Go requires a huge sample size. MuZero [Schrittwieser et al., 2020], employing a

learned model, has been successful in the domain of game AI and can significantly reduce the number of interactions with the real environment. However, this does not render research on approaches without a learned model like AlphaZero unnecessary. For example, it is mentioned that AlphaZero's learning is faster than MuZero's in the case of chess [Danihelka et al., 2022].

Thus, in RL research, a fast simulator of the environment, which achieves high throughput is often required. Simulators that possess practical speed and performance are often implemented in C++. In the machine learning community, however, Python serves as a *lingua franca*. Therefore, libraries like OpenSpiel [Lanctot et al., 2019] wrap the core C++ implementation and provide a Python API. However, this approach presents several challenges. Efficient parallelization of the environments is important for generating a large number of samples, but efficient parallel environments utilizing C++ threading, such as EnvPool [Weng et al., 2022b], may not always be accessible from Python. In fact, to our knowledge, there are no libraries publicly available in Python that allow the use of efficient parallel environments for important game AI benchmarks like chess and Go from the identical Python API. Also, the RL algorithm often runs on accelerators (such as GPUs and TPUs), whereas simulation runs on CPUs, which makes additional data transfer costs between CPUs and accelerators.

In *continuous* state space environments, Brax [Freeman et al., 2021] and Isaac Gym [Makoviychuk et al., 2021] demonstrate that environments that work on accelerators can dramatically improve the simulation throughput and RL training speed, resolving the data transfer and parallelization problems. Brax, written in JAX [Bradbury et al., 2018], a Python library, provides *hardware-accelerated* environments that leverage JAX's auto-vectorization, parallelization over accelerators, and Just-In-Time (JIT) compilation optimized for individual hardware platforms.

In this chapter, we offer the hardware-accelerated environments of complex, *discrete* state domains like chess, shogi and Go. Specifically, we introduce Pgx, a suite of efficient game simulators developed in JAX. Thanks to JAX's auto-vectorization

and parallelization across multiple accelerators, Pgx can achieve high throughput on GPUs. As of writing this paper, to our best knowledge, there is no other broad game environment library written in JAX. Highlighted features of Pgx include:

- **Fast simulation**: Pgx provides high-performance simulators written in JAX that run fast on GPU, similar to Brax. We demonstrated that Pgx is 10-100x faster than existing Python libraries such as PettingZoo [Terry et al., 2021] and OpenSpiel [Lanctot et al., 2019] on a DGX-A100 workstation (see Figure 3.2).
- **Diverse set of games**: Pgx offers over 20 games, ranging from perfect information games like chess to imperfect information games like bridge (see Table 3.1). Pgx also offers miniature versions of game environments (e.g., miniature chess) to facilitate research cycles.
- **Baseline models**: Evaluating agents in multi-agent games is relative, requiring baseline opponents for evaluation. Since it is not always easy to have appropriate baselines available, Pgx provides its own baseline models. In Section 3.6, we demonstrate the availability of them with AlphaZero training.

Pgx is open-sourced and freely available at `https://github.com/sotetsuk/pgx`[1].

3.2 How to vectorize game environments

As mentioned in the previous chapter, game environments are not necessarily suitable for SIMD approaches like continuous action space tasks handled by Isaac Gym/Brax. Here, we explain how to efficiently vectorize such seemingly non-vectorizable game environments. The key idea is to impose constraints on the program. Not all programs can be easily vectorized, and programs that can be vectorized are not necessarily fast.

Constraints for automatic vectorization. All of our game environment state transitions are stateless and implemented as *pure functions*. A pure function always

[1]See Section A.1 for the license.

returns the same output for the same input, does not depend on external state, and has no side effects (e.g., changes to global variables or I/O operations). To efficiently parallelize, it is important that individual calculations do not interfere with each other. Pure functions can avoid concurrency and synchronization issues due to parallelization because they always return the same result for the same input and do not depend on external state. JAX can easily and efficiently vectorize such pure function implementations using `vmap`. JAX provides a set of functions to help implement such pure functions, including control structures like `if` and `for` based on the functional programming paradigm, which uses pure functions.

Constraints for fast vectorized simulation. While pure function implementations are essential for efficient vectorization, they alone are not sufficient. Here, we describe the heuristic optimizations used in the implementation of Pgx game environments, designed to run efficiently on accelerators.

1. *Consider utilizing mathematical tricks.* For example, in Go, mathematical shortcuts can be applied to avoid searching when detecting suicide moves or captures.

2. *Consider rewriting in the form of matrix operations, even if this introduces some redundant calculations.* For example, in Chess, the available moves for each piece are represented as a board. While this representation is somewhat redundant, it facilitates efficient computation on accelerators.

3. *Precompute and store static information as matrices or lookup tables.* In the Chess example, the potential moves for each piece, and the relationship between discrete actions and board positions, are stored in static precomputed matrices. Similarly, static lookup tables can be implemented using methods like cuckoo hashing.

4. *Prefer* `vmap` *over sequential* `for` *loops or* `map` *when there are no dependencies between iterations.* For processes where sequential dependencies are not present,

```python
import jax
import pgx

env = pgx.make("go_19x19")
init_fn = jax.jit(jax.vmap(env.init))
step_fn = jax.jit(jax.vmap(env.step))

batch_size = 1024
# pseudo-random number generator keys determine the first players
rng_keys = jax.random.split(jax.random.PRNGKey(9999), batch_size)

state = init_fn(rng_keys)  # vectorized initial states (1024, ...)
while not state.terminated.all():
  action = model(state.current_player, state.observation, state.legal_action_mask)
  state = step_fn(state, action)  # state.rewards with shape (1024, 2)
```

Figure 3.1: Basic usage of Pgx. The *init* function generates the initial *state* object. The *state* object has an attribute *current player* that indicates the agent which acts next. In this case, since we are using a batch size of 1024 for a 2-player game, *current player* is a binary vector whose size is 1024. Note that *current player* is independent of the colors (i.e., first player or second player). Here, *current player* is determined randomly using a pseudo-random number generator. The *step* function takes the previous *state* and an *action* vector, whose size is 1024, as input and returns the next *state*. The *observation* of the *current player* can be accessed through *state*. In the case of Go, for example, each *observation* has a shape of $1024 \times 19 \times 19 \times 17$. The available actions at the current *state* can be obtained through the boolean vector *legal action mask*. Here, it has a shape of 1024×362. For more detailed API description and usage, refer to the Pgx documentation at `https://sotetsuk.github.io/pgx/`.

such as enumerating legal moves in Chess and checking for suicide moves, it is preferable to vectorize these operations using `vmap` instead of processing them sequentially in a loop or with *map*.

In Appendix, we provide the complete source code implementations of Go and Chess based on these constraints (Section A.2).

3.3 Pgx overview

In this section, we will provide an overview of the basic usage of Pgx, along with the fundamental principles behind the design of the Pgx API. Additionally, we will explain the overview of the game environments currently offered by Pgx as of this publication.

3.3.1 Pgx API design

Pgx does take inspiration from existing APIs, but it provides its own custom API for game environments. Specifically, two existing Python RL libraries highly inspired Pgx API:

- **Brax** [Freeman et al., 2021], a physics engine written in JAX, providing continuous RL tasks, and
- **PettingZoo** [Terry et al., 2021], a multi-agent RL environment library available through Gym-like API [Brockman et al., 2016].

Figure 3.1 describes an example usage of Pgx API. The main difference from the Brax API comes from that the environments targeted by Pgx are multi-agent environments. Therefore, in Pgx environments, the next agent to act is specified as the *current player*, as in the PettingZoo API. On the other hand, the significant difference from the PettingZoo API stems from the fact that Pgx is a vectorization-oriented library. Therefore, Pgx does not use an *agent iterator* like PettingZoo and does not allow the number of agents to change. Concrete code examples comparing the Pgx API with the Brax and PettingZoo APIs are provided in Section A.3.

While we do not focus on the design of the API, the Pgx API is sufficiently generic. At present, all game environments implemented in Pgx can be converted to the PettingZoo API and called from the PettingZoo API through the Pgx API (see Section A.3). This fact demonstrates the practical generality of the Pgx API. However, there is a limitation of the Pgx API in that it cannot handle environments where the number of agents dynamically changes. This limitation arises from the fact that the Pgx API is specialized for efficient vectorized simulation.

3.3.2 Available games in Pgx

The Pgx framework offers a diverse range of games, as summarized in Table 3.1. While Pgx primarily emphasizes multi-agent board games, it also includes some single-agent

Table 3.1: Available games in Pgx.

Env Name	# Players	Obs. shape	# Actions	Tag
2048	1	$4 \times 4 \times 31$	4	perfect info. (w/ chance)
Animal shogi	2	$4 \times 3 \times 194$	132	perfect info.
Backgammon	2	34	156	perfect info. (w/ chance)
Bridge bidding	4	480	38	imperfect info.
Chess	2	$8 \times 8 \times 119$	4672	perfect info.
Connect Four	2	$6 \times 7 \times 2$	7	perfect info.
Gardner chess	2	$5 \times 5 \times 115$	1225	perfect info.
Go 9x9	2	$9 \times 9 \times 17$	82	perfect info.
Go 19x19	2	$19 \times 19 \times 17$	362	perfect info.
Hex	2	$11 \times 11 \times 4$	122	perfect info.
Kuhn poker	2	7	2	imperfect info.
Leduc hold'em	2	34	3	imperfect info.
MinAtar Asterix	1	$10 \times 10 \times 4$	5	Atari-like
MinAtar Breakout	1	$10 \times 10 \times 4$	3	Atari-like
MinAtar Freeway	1	$10 \times 10 \times 7$	3	Atari-like
MinAtar Seaquest	1	$10 \times 10 \times 10$	6	Atari-like
MinAtar Space Invaders	1	$10 \times 10 \times 6$	4	Atari-like
Othello	2	$8 \times 8 \times 2$	65	perfect info.
Shogi	2	$9 \times 9 \times 119$	2187	perfect info.
Sparrow mahjong	3	11×15	11	imperfect info.
Tic-tac-toe	2	$3 \times 3 \times 2$	9	perfect info.

environments and Atari-like environments to assist board RL research. Thus, as we will describe below, games implemented in Pgx span various categories, including two-player perfect information games, games with chance events, imperfect information games, and Atari-like games.

Two-player perfect information games Pgx provides two-player perfect information games that range from simple games like *Tic-tac-toe* and *Connect Four* to complex strategic games like *chess*, *shogi*, and *Go 19x19*. While these traditional board games offer rich gameplay and strategic depth, they can be computationally demanding for many RL researchers. To address this, Pgx also includes smaller versions of shogi and chess: *Animal shogi* and *Gardner chess*. Although they have smaller board sizes compared to their original counterparts, these games are not mere toy environments. They retain enough complexity to provide engaging gameplay experiences for humans. Animal shogi, in particular, was specifically designed for children, while Gardner chess has a history of active play in Italy [Mhalla and Prost, 2013]. Additionally, Pgx offers other medium-sized two-player perfect information games such as *Go 9x9*, *Hex* and *Othello*.

Games with (stochastic) chance events Pgx supports perfect information games with chance events, including *backgammon* and *2048*, which are popular benchmarks for RL algorithms in stochastic state transitions [Antonoglou et al., 2022, Paster et al., 2022]. These games introduce elements of randomness and uncertainty, adding a layer of complexity and decision-making under uncertainty to the gameplay.

Imperfect information games In the realm of imperfect information games, Pgx provides several environments. These include *Kuhn poker*, *Leduc hold'em*, *Sparrow mahjong* (a miniature version of mahjong), and *bridge bidding*. These games involve hidden information, requiring agents to reason and strategize based on imperfect knowledge of the current game state.

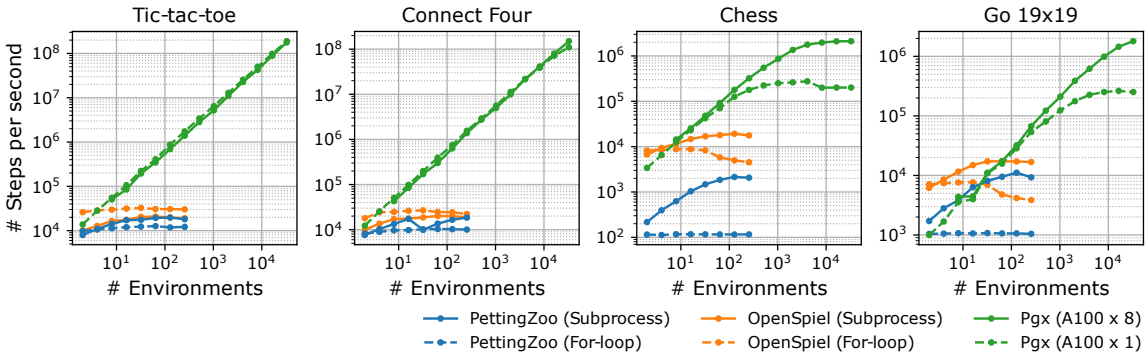


Figure 3.2: Simulation throughput of PettingZoo, OpenSpiel, and Pgx. Policies are random without learning processes. Error bars are not visible in this scale.

Atari-like games While the primary focus of Pgx is on board games, to make Pgx all-inclusive, Pgx also implements *all* five environments from the MinAtar game suite: *Asterix*, *Breakout*, *Freeway*, *Seaquest*, and *Space Invaders* [Young and Tian, 2019]. These Atari-like environments offer a more visually oriented and dynamic gameplay experience compared to traditional board games. Researchers often utilize MinAtar to conduct comprehensive ablation studies on RL methods in environments with visual inputs [Ceron and Castro, 2021, Gogianu et al., 2021]. Of these games, Freeway and Seaquest are highlighted as significant benchmarks for assessing the exploration capabilities of RL algorithms [Ceron and Castro, 2021]. However, as of this writing, gymnax has not incorporated Seaquest into its suite.

For detailed descriptions of each environment, please refer to `https://sotetsuk.github.io/pgx`.

3.4 Performance benchmarking: simulation throughput

Pgx excels in efficient and scalable simulation on accelerators thanks to JAX's auto-vectorization, parallelization over accelerators, and JIT-compilation. In this section, we validate it through experiments on an NVIDIA DGX-A100 workstation.

3.4.1 Comparison to existing Python libraries

Experiment setup. We compare the pure simulation throughput of Pgx with existing popular RL libraries available in Python: PettingZoo [Terry et al., 2021] and OpenSpiel [Lanctot et al., 2019]. For the evaluation, we specifically selected Tic-tac-toe, Connect Four, chess, and Go, as these games are included in all three libraries. To our knowledge, neither PettingZoo nor OpenSpiel provides official parallelized environments from the Python API. Therefore, we prepared two implementations for each library:

1. **For-loop** (*DummyVecEnv*): sequentially executes and does not parallelize actually.

2. **Subprocess** (*SubprocVecEnv*): parallelize using the *multiprocessing* module in Python.

We modified and used *SubprocVecEnv* provided by Tianshou [Weng et al., 2022a] for all competitor libraries. We performed all experiments on an NVIDIA DGX-A100 workstation with 256 cores; Pgx simulations used a single A100 GPU or eight A100 GPUs. We used random policies to evaluate the pure performance of simulators without agent learning. In all implementations, the environment automatically resets to the initial state upon reaching termination. The Pgx version used here was v0.8.0.

Results. Figure 3.2 shows the results. We found that Pgx achieves at least 10x faster throughput than other existing Python libraries when the number of vectorized environments (i.e., batch size) is large enough (e.g., 1024) on a single A100 GPU. Furthermore, when utilizing eight A100 GPUs, Pgx achieves throughput of approximately 100x higher. This trend was identical from the simplest environment, Tic-tac-toe, to complex environments such as 19x19 Go.

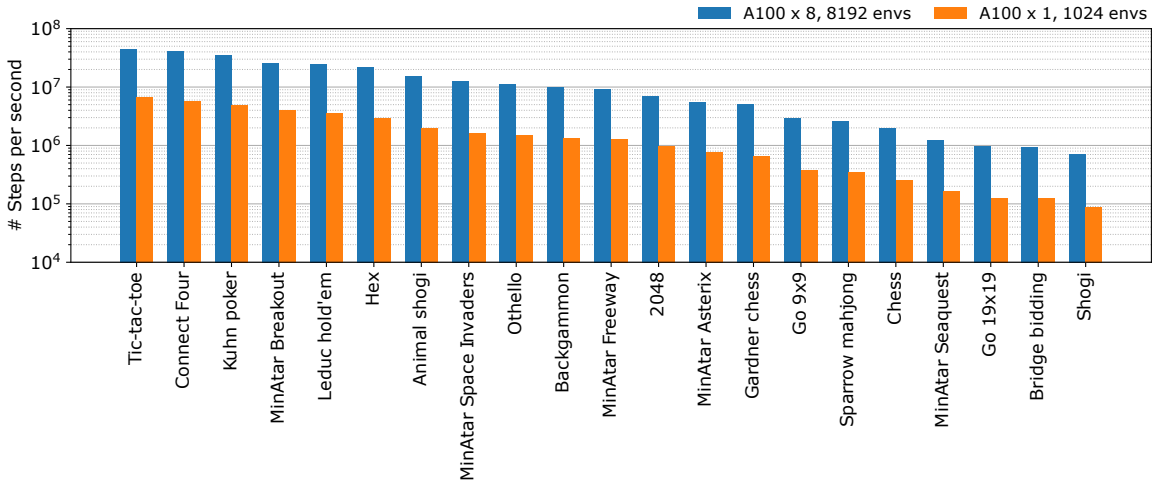


Figure 3.3: Simulation throughput of all Pgx environments. Error bars are not visible in this scale.

3.4.2 Throughput of other Pgx environments

The throughput of each environment is influenced by several factors, including the complexity and nature of the game, as well as the quality of its implementation. For instance, OpenSpiel demonstrates throughput of the same order for chess and Go 19x19. In contrast, PettingZoo's chess implementation exhibits a throughput approximately 10x slower than its Go 19x19 counterpart. This suggests that there might be room for optimization in PettingZoo's chess implementation regarding execution speed. To ensure that *all* Pgx environments achieve reasonable throughput and scalability like the four environments shown in Figure 3.2, we measured the sample throughput of all other Pgx environments on a DGX-A100 workstation, following the same approach as in the previous section. The number of vectorized environments was 1024 for a single A100 GPU and 8192 for eight A100 GPUs. The results are shown in Figure 3.3. From these results, we can see that even in the slowest environment, Pgx achieves a throughput of approximately 10^5 samples/second with a single A100 GPU. This demonstrates the efficiency of Pgx, considering that achieving such throughput with other Python libraries, as shown in the previous section, is challenging even in simpler environments like Tic-tac-toe. Furthermore, we observe a significant improvement in throughput when using eight A100 GPUs compared to a single A100 GPU in all environments.

The throughput increased by an average of 7.4x across all environments (at least 6.6x in the MinAtar Breakout environment). This highlights the excellent parallelization performance of Pgx across multiple accelerators.

3.5 PPO training example

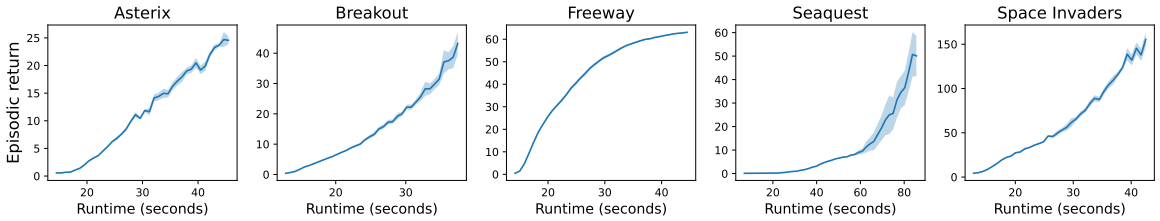


Figure 3.4: PPO training in MinAtar suite using Pgx. Each model is trained up to 20M frames on a single A100 GPU. The shaded area represents the standard error of five runs.

In Section 3.6, we conducted learning experiments with AlphaZero for two-player, perfect information games such as 9x9 Go. Here, we present the results of RL training using the Proximal Policy Optimization (PPO) algorithm [Schulman et al., 2017] in the MinAtar [Young and Tian, 2019] environments as an example of model-free RL with Pgx.

MinAtar is a miniature version of Atari, specifically used by researchers to perform detailed ablation studies on RL methods using visual inputs (like Atari), without requiring large computational resources [Ceron and Castro, 2021, Gogianu et al., 2021]. MinAtar includes five games: Asterix, Breakout, Freeway, Seaquest, and Space Invader. It may be worth noting that Freeway and Seaquest are highlighted as possible useful benchmarks for exploration methods [Ceron and Castro, 2021]. Pgx has re-implemented all five environments in JAX. The versions of *pgx* and *pgx-minatar* used here were v0.9.0 and v0.2.1, respectively.

PPO implementation details. The implementation of PPO was based on the code available at `https://github.com/luchris429/purejaxrl` [Lu et al., 2022], with some

modifications. As Pgx focuses on achieving high-speed training through vectorized simulation, we utilized a large batch size of 4096, similar to the training example in Brax [Freeman et al., 2021]. We fine-tuned hyperparameters using the Asterix environment in the preliminary experiments (with a different seed used in the main experiments), and we maintained the same hyperparameters across all five games. The hyperparameters are listed in Table 3.2. Given the large batch size, the training was conducted up to 20M frames, which is longer compared to previous studies. However, it should be noted that the execution time is remarkably short, as discussed later.

Table 3.2: Hyperparameters in PPO training.

Hyperparameter	Value
Rollout batch size (i.e., number of vectorized environments)	4096
Rollout length	128
Training minibatch size	4096
Number of epochs	3
Optimizer	Adam
Learning rate	0.0003
Gradient clipping max norm	0.5
Discount factor (γ)	0.99
GAE lambda (λ)	0.95
Clipping parameter (ϵ)	0.2
Value function coefficient	0.5
Entropy coefficient	0.01

Results. Figure 3.4 displays the learning results of PPO in the five MinAtar games. We conducted five learning runs with different seeds for each game. At each point of the learning process, we conducted 100 evaluation runs using the learned (stochastic) policy at that time and plotted the average score. The shaded area illustrates the standard error of the mean scores over the five runs. The runtime includes the RL training time and JIT-compilation time of JAX code. In all games, PPO with Pgx achieves sufficiently high scores in less than a minute, except for Seaquest, which takes more than 80 seconds to elapse 20M frames. These results demonstrate the effectiveness of vectorized execution in other game types than two-player, perfect information games

implemented in Pgx.

3.6 AlphaZero on Pgx environments

In this section, we exhibit the effectiveness of RL training on accelerators using the Pgx environments, with a specific focus on two-player perfect information games such as Go. For demonstrations of RL training in other game types, please refer to Section 3.5. We begin by providing a brief overview of the Gumbel AlphaZero. Next, we describe the Pgx environments where we apply the Gumbel AlphaZero. Subsequently, we explain the experimental setup and discuss the selection of baseline models, which serve as anchor opponents for evaluation purposes. Finally, we present the results.

AlphaZero [Silver et al., 2018] and Gumbel AlphaZero [Danihelka et al., 2022]. AlphaZero is an RL algorithm that leverages a combination of NNs and Monte Carlo Tree Search (MCTS). It has achieved state-of-the-art performance in chess, shogi, and Go, employing a unified approach. Through self-play, AlphaZero integrates MCTS with the current parameters of NNs, continually updating them through training on the samples generated during self-play. Gumbel AlphaZero is an adaptation of AlphaZero that removes several heuristics present in AlphaZero, enabling it to function even with a reduced number of simulations. In particular, Gumbel AlphaZero addressed the issue in the original AlphaZero where utilizing Dirichlet noise at the root node during tree search did not guarantee policy improvement. To overcome this limitation, Gumbel AlphaZero introduced an enhancement by utilizing the Gumbel-Top-k trick to perform sampling actions without replacement. They also proposed a MuZero version of the algorithm but we focus on the evaluation of AlphaZero in this paper. They released the Mctx library, which includes a JAX implementation of Gumbel AlphaZero. We utilized this library in our experiments. From now on, unless otherwise specified, when referring to "AlphaZero," it refers to Gumbel AlphaZero, not the original AlphaZero.

Environments. While Pgx enables fast simulations on accelerators, large-scale environments such as chess, shogi, and 19x19 Go pose significant challenges for researchers in terms of computational resources for learning. To address this, Pgx provides several small-scale environments, including miniature versions of shogi (Animal shogi) and chess (Gardner chess), as well as Hex (11x11) and Othello (8x8). In addition to these environments, we included the 9x9 Go environment to create a set of five environments for training AlphaZero. Here, we describe these environments briefly:

- **Animal shogi** is a 4x3 miniature version of shogi designed originally for children. Like shogi, players can reuse captured pieces. The small board size allows researchers to conduct research in an environment where planning ability is important with minimal computational resources.

- **Gardner chess** is a 5x5 variant of minichess that uses the leftmost five columns of the standard chessboard. It has a history of active play by human players in Italy [Mhalla and Prost, 2013].

- **Go 9x9** maintains the essential aspects of full-sized Go while being the smallest playable board size in the game. The advantage of 9x9 Go is that several full-sized Go AI models are also capable of playing on the 9x9 board, allowing us to use 9x9 Go as a reliable benchmark (e.g., Danihelka et al. [2022]).

- **Hex** is a game played on an 11x11 board where two players take turns placing stones, and the player who forms a connected path from one side of the board to the other with their stones wins. Its rules are simple, making it relatively easy to interpret for researchers.

- **Othello**, also known as Reversi, is played on an 8x8 board. Players take turns placing stones and flip the opponent's discs that are sandwiched. The game concludes when neither player can make a valid move, and the player with the most discs on the board wins.

For more detailed information about each environment, please refer to `https://sotetsuk.github.io/pgx`.

Training setup. We trained the models using the same network architecture and hyperparameters across all five environments. The network architecture is 6 ResNet blocks with policy head and value head, following the structure outlined in the original AlphaZero study [Silver et al., 2018] basically but with a smaller model size. During the self-play, 32 simulations were performed at each position for policy improvement. In each iteration, we generated data for 256 steps with a self-play batch size of 1024 (i.e., the number of vectorized environments). We then divided this data into mini-batches of size 4096 for gradient estimation and parameter updates. We performed training for 400 iterations ($\approx$ 105M frames) in each environment. The choice of accelerator varied across the environments, but for example, in the case of 9x9 Go, we trained the model using a single A100 GPU, and the training process took approximately 8 hours.

Evaluation and baseline model selection. We trained agents using the AlphaZero algorithm on the five environments described above and performed evaluations. However, in multi-agent games, the performance of the trained agents is relative, so we need a reference agent for comparison. However, finding a suitable baseline model for any environment is difficult, or even if it exists, it may not be computationally efficient. Therefore, for researchers and practitioners using Pgx, we created our own baseline models. It is important to note that our baseline models are not designed to be state-of-the-art or oracle models, but rather serve the purpose of examining the learning process within the Pgx environment. In the given learning setup, we selected the 200-iteration ($\approx$ 52M frames) model for 9x9 Go and the 100-iteration ($\approx$ 26M frames) model for other environments, considering their lower complexity compared to 9x9 Go. To evaluate the agents, we estimated the Elo rating through their pairwise matches. We used the *BayesElo* program to calculate the Elo rating[2][Coulom, 2008].

[2] `https://www.remi-coulom.fr/Bayesian-Elo`

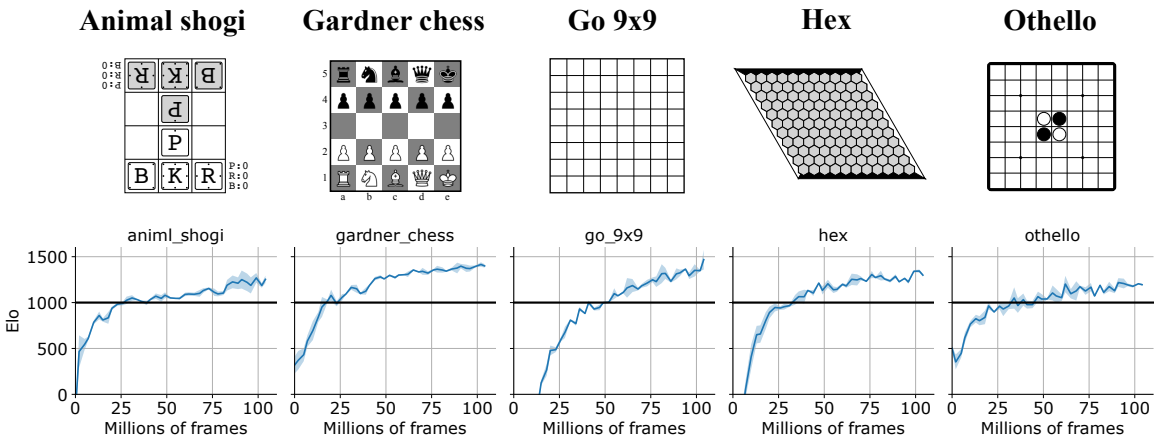


Figure 3.5: AlphaZero training results using Pgx. Black line represents the Elo rating of baseline models provided by Pgx (1000 Elo). The shaded area represents the standard errors of two runs.

We adjusted the Elo rating to ensure that our baseline models had 1000 Elo. During the evaluation matches, the agents conducted 32 simulations for each move like during the training.

Results. Figure 3.5 presents the learning results of AlphaZero in the five environments. We can observe that the agents successfully learn in all five environments starting from a random policy with the same network architecture and hyperparameters. Furthermore, for the baseline model in 9x9 Go, we evaluated its performance by playing against Pachi [Baudiš and Gailly, 2012] with 10K simulations per move, which was used as a baseline opponent in prior study [Danihelka et al., 2022]. The baseline model conducted 800 simulations for each move. Our baseline model outperformed Pachi with a record of 62 wins and 38 losses out of 100 matches, confirming its reasonable strength as a baseline model. Although no comparisons were made with other AIs in environments other than 9x9 Go, we trained them using the same network architecture and hyperparameters as in 9x9 Go. Given that the baseline model obtained in 9x9 Go exhibited reasonable strength throughout the learning process, we suppose that the baseline models in other environments, which were trained with exactly the same settings, have also learned reasonably. Therefore, we believe that researchers can

accelerate their research cycles using the five environments and baseline models presented here, instead of relying on full-scale chess, shogi, and 19x19 Go, while exploring RL algorithms such as AlphaZero.

Details. Here, we describe the details of AlphaZero training. In the AlphaZero training, we used the same hyperparameters for all games. Table 3.3 shows the hyperparameters used in the training. The network architecture is the same as the original AlphaZero [Silver et al., 2018] except for the following points:

- We used ResNet v2 [He et al., 2016] instead of v1.
- We used a smaller network (as shown in Table 3.3).

The GPUs used for training vary depending on the game. Table 3.4 shows the GPUs and runtime for each game. The version of Pgx used in the training is v0.8.0.

Table 3.3: Hyperparameters in AlphaZero training.

Hyperparameter	Value
Number of residual blocks	6
Number of channels of conv. layer	128
Self-play batch size (i.e., number of vectorized environments)	1024
Self-play length in each iteration	256
Number of simulations per move	32
Training minibatch size	4096
Optimizer	Adam
Learning rate	0.001
Completed Q-values *value scale*	0.1
Completed Q-values *rescale values*	False

Table 3.4: GPUs details in AlphaZero training.

Environment	GPUs	Runtime (hours)
Animal shogi	A4000 x 1	6.2
Gardner chess	A4000 x 4	14.3
Go 9x9	A100 x 1	8.6
Hex	A4000 x 1	17.6
Othello	A4000 x 1	11.4

3.7 Training scalability to multiple accelerators

In Section 3.4, we demonstrated a significant improvement in the pure throughput of Pgx when increasing the number of accelerators using a random agent. Here, we will show that increasing the number of cores also improves the learning speed in the AlphaZero training on Pgx environments.

Experiment setup. To demonstrate the improvement in learning speed by increasing the number of cores in AlphaZero training with Pgx, we conducted experiments in the 9x9 Go environment. In the experiments of Section 3.6, we performed training using a single A100 GPU in the 9x9 Go environment. Here, we conducted the exact same number of training frames but with an increased number of GPUs and batch size during self-play. In the experiment using a single A100 GPU, the batch size during self-play was set to 1024. However, in training with eight A100 GPUs, we increased the *self-play* batch size to 8192, which is eight times larger. It is important to note that the *training* batch size, learning rate, and the other hyperparameters were kept the same as in the single GPU case, ensuring that these hyperparameters did not affect the learning speed. Similar to Section 3.6, we used the baseline model as an anchor and adjusted Elo ratings so that the rating of the baseline model is 1000. Furthermore, in this setup, we want to mention that the time spent on self-play was dominant (more than 90%) compared to the time spent on training (gradient calculation and parameter updates).

Results. Figure 3.6 shows the learning curves for the 9x9 Go environment using one A100 GPU and eight A100 GPUs. The shaded regions represent the standard error of runs with two different seeds. Based on the figure, it is evident that when both models are trained with the same number of training frames, the model trained with eight A100 GPUs achieves the same level of performance approximately four times faster than the model trained with a single GPU. This experimental outcome highlights the

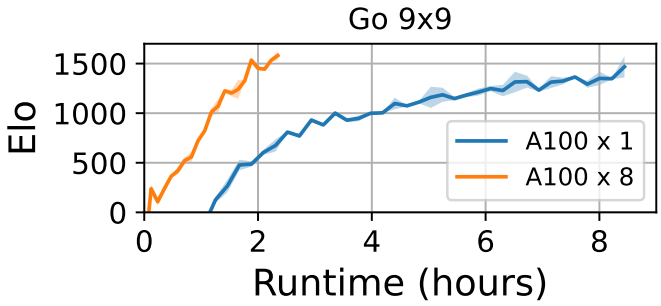


Figure 3.6: Multi-GPU AlphaZero training in 9x9 Go.

fact that Pgx not only enhances throughput in random play but also accelerates the learning process when training RL algorithms such as AlphaZero. These results feature the practical utility of Pgx in the field of RL, providing researchers and practitioners with a powerful tool for efficient experimentation utilizing multiple accelerators.

3.8 Related work

Games in AI research. An early study that combined neural networks (NNs) with RL to build world-class agents in a complex board game was TD-Gammon [Tesauro, 1995]. After the breakthrough of deep learning [Krizhevsky et al., 2012], RL agents combined with NNs performed well in the video game domain [Mnih et al., 2015] and large state fully-observable board games, including chess, shogi, and Go [Silver et al., 2016, 2017, 2018]. RL agents with NNs also performed well in large-scale, partially observable games like mahjong [Li et al., 2020a]. However, these RL agents in complex board games require a huge number of self-play samples.

Games as RL environment. Game AI studies often have to pay high engineering costs, and there are a variety of libraries behind the democratization of game AI research. Arcade learning environment (ALE) made using Atari 2600 games as RL environments possible [Bellemare et al., 2013]. Several RL environment libraries provide classic board game suits [Lanctot et al., 2019, Terry et al., 2021, Zha et al., 2020]. Pgx

Table 3.5: Environments implemented in JAX. Sorted by the initial submission date to arXiv.

Library	Environments	arXiv sub.
Brax [Freeman et al., 2021]	Continuous control	2021/6/24
gymnax [Lange, 2022]	Classic control, bsuite, MinAtar	
Pgx (ours)	Board Games	2023/05/29
Jumanji [Bonnet et al., 2024]	Combinatorial optimization	2023/06/16
Waymax [Gulino et al., 2023]	Autonomous driving	2023/10/12
JaxMARL [Rutherford et al., 2024]	Multi-agent RL	2023/11/16
XLand-MiniGrid [Nikulin et al., 2024]	Meta RL, Mini-grid	2023/12/19
Craftax [Matthews et al., 2024]	Open-ended RL	2024/02/26
TORAX [Citrin et al., 2024]	Tokamak Transport	2024/06/10
NAVIX [Pignatelli et al., 2024]	Mini-grid	2024/07/28

aims to implement (classic) board game environments with high throughput utilizing GPU acceleration.

Hardware-accelerated RL environments. While hardware acceleration is a more specific approach compared to methods that run on CPUs, such as EnvPool [Weng et al., 2022a], it has a major advantage of its ability to leverage accelerators for parallel execution, enabling high-speed simulations. Also, NN training is often performed on GPU accelerators, and there is an advantage that there is no data transfer cost between CPU and GPU accelerators. There is a wide range of environments available through various open-source software. In particular, JAX-based environments have gained popularity due to their high scalability over accelerators. We summarize these environments, including Pgx, in Table 3.5. Pgx complements these environments by offering a (classic) *board game* suite for (multi-agent) RL research. Other prominent hardware-accelerated environments include, but are not limited to, Isaac Gym [Makoviychuk et al., 2021] for continuous control and CuLE [Dalton and Frosio, 2020], a GPU-based Atari emulator.

Algorithms and architectures that can leverage Pgx. The Anakin architecture [Hessel et al., 2021] is an RL architecture that enables efficient utilization of accelerators and fast learning under the constraint that both the algorithm and en-

vironment are written as pure JAX functions. The architecture is capable of scaling up to (potentially) thousands of TPU cores with a simple configuration change. Since all Pgx environments are implemented using pure JAX functions, the Anakin architecture is applicable to any Pgx environment. Gumbel AlphaZero [Danihelka et al., 2022] improves the performance of AlphaZero when the number of simulations is small by employing the Gumbel-Top-k trick for search. They provide JAX-based Gumbel AlphaZero implementation[3][Babuschkin et al., 2020], which allows batch planning on accelerators. In Section 3.6, we use this implementation to show the example of Pgx usage in AlphaZero training.

3.9 Limitations and future work

Limitations. There are several limitations users should take care of regarding Pgx, including:

- **Lack of support for Atari:** One of the limitations of Pgx is that it does not support Atari, which is an important benchmark in RL research. This limitation arises from the difficulty of implementing the Atari emulator and re-implementing the dynamics of each game in JAX.
- **Pgx API limitation:** While the games currently implemented in Pgx can be exported to the PettingZoo API, which is regarded as a general API for multi-agent games, Pgx API is not well-suited for handling certain types of games. These game types include those with a varying number of agents and those that involve chance players (nature players) such as poker.
- **JAX lock-in:** Although Pgx provides a convenient way to implement fast algorithms in Python without directly working with C++, it has a reliance on JAX, which may require users to be familiar with JAX. This can make it less straightforward to utilize other frameworks like PyTorch [Paszke et al., 2019].

[3] `https://github.com/google-deepmind/mctx`

Future work. Our future work for Pgx includes the following:

- **Expansion of baseline algorithms and models:** Currently, we are unable to provide learning examples or models for large-scale games like chess, shogi, and Go 19x19, making it an important area for future work. We plan to expand the availability of strong models through proprietary training and connect with other strong AI systems to enhance the baselines.
- **Diversification of game types:** The current game collection in Pgx is biased towards (two-player) perfect information games. We plan to implement games with imperfect information, such as Texas hold'em and mahjong, to broaden the range of supported game types.
- **Verification on TPUs:** While we validated Pgx performance on an NVIDIA DGX-A100 workstation, it is important to conduct verification using Google TPUs as well. This will provide valuable insights into the performance and scalability of Pgx on different hardware architectures.
- **Human-vs-agent UI:** Developing a user interface that enables human-versus-agent gameplay is important future work. This will allow researchers with domain knowledge to conduct high-quality evaluations and experiments, fostering improved research and assessment.

By addressing these areas in our future work, we aim to enhance the capabilities and applicability of Pgx in the field of RL research and game AI for both researchers and practitioners.

3.10 Conclusion

We proposed Pgx, a library of hardware-accelerated game simulators that operate efficiently on accelerators, implemented in JAX. Pgx achieves 10-100x higher throughput compared to other libraries available in Python and demonstrates its ability to

scale and train using multiple accelerators in the context of AlphaZero training. By providing smaller game environments, along with their baselines, Pgx facilitates the development and research of RL algorithms and planning algorithms that can operate at faster speeds. We anticipate that Pgx will contribute to advancing the field in terms of developing efficient RL algorithms and planning algorithms in these accelerated environments.

Chapter 4

A Demonstration: Bridge Bidding AI

In this chapter, we show that we can achieve the state-of-the-art (SOTA) performance in the bridge bidding AI benchmark using Pgx, developed in Chapter 3. The training recipe we present here is a simple combination of existing techniques, except that it uses the massively vectorized environment, which demonstrates the potential effectiveness of massively batching approaches.

4.1 Introduction

Throughout the history of artificial intelligence (AI) research, games have played central roles as benchmarks for measuring progress. AIs have now achieved or even surpassed the skill levels of human experts in a variety of classic games. Notable examples include backgammon [Tesauro, 1995], chess [Silver et al., 2018], Go [Silver et al., 2016, 2017, 2018], poker [Brown and Sandholm, 2018, 2019], mahjong [Li et al., 2020a], and Atari 2600 [Mnih et al., 2015].

Contract bridge joins the ranks of these classic games as a significant benchmark for AI [Ginsberg, 1999, Ventos et al., 2017, Yeh et al., 2018, Rong et al., 2019, Tian et al., 2020, Lockhart et al., 2020]. It presents complex sets of challenges due to its

multi-agent nature, the imperfect information available to players, and the need for both cooperation within teams and competition against the opposing team. Bridge is somewhat akin to the game of Hanabi [Bard et al., 2020], where information sharing is essential, though bridge also incorporates the competitive element of playing against another team like DouDiZhu [Zha et al., 2021]. Despite extensive research efforts, to our best knowledge, no AI has yet been demonstrated to consistently outperform top human players in bridge.

The game of bridge is structured around two main phases: bidding and playing. The bidding phase, in particular, is critical to success in the game [Yeh et al., 2018] and is the focus of our study. Our contributions to this area are twofold:

- We have discovered that a straightforward integration of existing techniques can achieve state-of-the-art (SOTA) performance in the bidding phase, specifically in tests against WBridge5[1]. This program is a multiple-time winner of the World Computer-Bridge Championship (2005, 2007, 2008, and 2016-2018) and serves as the standard benchmark for bridge AI research.

- To foster further advancements in the field, we have made our code and trained models open-source. This allows our work to be easily reproduced and verified by others, offering a new baseline for future research in bridge AI, beyond the traditional evaluations using WBridge5.

4.2 Background: bridge overview

Here, we provide a simplified overview of the game's flow rather than detailing all its rules. Bridge is a card game for four players, divided into two teams. Each player receives 13 cards from a standard 52-card deck, and these cards are kept secret from the other players. The game unfolds in two main stages: the bidding phase and the playing phase.

[1] `http://www.wbridge5.com/`

- **Bidding phase.** In this auction-style stage, players predict how many tricks (sets of four cards, one from each player) their team can win, using bids as a form of communication to signal their hand's strength and potential to their partner. Additionally, they select a suit to serve as trump, which can override other suits to win tricks. They make bids to set a "contract," which outlines the number of tricks the team aims to win and identifies the "declarer" (the player who made the bid that established the final contract).

- **Playing phase.** Players take turns playing one card at a time, with the highest card of the led suit or trump winning the trick. This process repeats for all 13 tricks.

The team's score depends on meeting or exceeding their contract in tricks won, with penalties for falling short. Effective communication and strategy are key, as players must signal their hand's potential to their partner through their bids to form a winning contract.

4.3 Related work

While advancements like those by Jack[2], WBridge5, and in the work of Ginsberg [1999] have seen AI reach human-level performance in the *playing* phase, the *bidding* phase remains a more formidable challenge [Yeh et al., 2018]. This complexity has guided much of the recent focus towards improving AI performance in the bidding aspect of bridge: Yeh et al. [2018] pioneered the application of neural networks to bridge bidding, albeit under some simplified conditions such as a restricted number of bids and opponents that always pass. Rong et al. [2019] developed a neural network-based bidding system free from these constraints. Their approach included both a policy network for decision-making and an estimation network to predict unseen hands, initially trained on data from human experts and later refined through reinforcement

[2] `https://www.jackbridge.com/eindex.htm`

learning (RL) and self-play. Gong et al. [2019] were the first to claim the creation of a strong bidding system developed without relying on human game data, achieving significant improvements over WBridge5. They utilized the A3C algorithm [Mnih et al., 2016] for training their policy-value network entirely through self-play. Tian et al. [2020] introduced a joint policy search (JPS) algorithm tailored for cooperative games, offering theoretical assurances that JPS-derived policies would at least match the performance of baseline strategies in purely cooperative settings. Despite these guarantees not strictly applying to bridge, their application of JPS led to enhanced bidding strategies. Lockhart et al. [2020] focused on developing AI policies capable of cooperating with human players, achieving SOTA results against WBridge5 through the use of search techniques and policy iteration on a pretrained model. To the best of our knowledge, their work represents the current benchmark in AI performance for bridge bidding.

4.4 Methods: training recipe

This section outlines the training process for our bridge bidding model, which involves two main stages:

- Initially, we pretrain the neural network using supervised learning (**SL**). Further information is given in Section 4.4.2.
- Next, we enhance the model using the Proximal Policy Optimization (PPO) algorithm [Schulman et al., 2017], a popular reinforcement learning (**RL**) method, combined with fictitious self-play (**FSP**) [Heinrich et al., 2015]. Details are provided in Section 4.4.3.

4.4.1 Network architecture and input features

Our model processes a 480-dimensional binary input vector, consistent with standards set by OpenSpiel [Lanctot et al., 2019] and Pgx [Koyamada et al., 2023]. The input fea-

Table 4.1: Input features in bridge bidding.

Feature	Size
Vulnerability	4
Pass before the opening bid	4
For each bid, who made it? (35 4-dim one-hot vector)	140
For each double, who made it? (35 4-dim one-hot vector)	140
For each redouble, who made it? (35 4-dim one-hot vector)	140
Current player's hand	52
Total	480

tures are detailed in Table 4.1. The network architecture comprises a 4-layer multi-layer perceptron (MLP), each layer containing 1024 neurons and employing ReLU activation functions [Glorot et al., 2011], following the design of Lockhart et al. [Lockhart et al., 2020]. Outputs include a policy head for 38 actions (35 bids, pass, double, redouble) and a value head.

4.4.2 Model pretraining by Supervised Learning (SL)

Initial training utilizes a dataset from OpenSpiel[3], also employed by Lockhart et al. [Lockhart et al., 2020]. This dataset, generated with WBridge5 but based on the SAYC bidding system[4], a simple bidding system different from WBridge5's own system. It includes 1M boards for training and 10K for evaluation, with 12.8M state-action pairs for training and 110K for evaluation. We used Adam [Kingma and Ba, 2015] with a learning rate of 1.0×10^{-4} and a batch size of 128, running the training over 40 epochs.

4.4.3 Reinforcement Learning (RL)

For model enhancement, we applied the PPO algorithm [Schulman et al., 2017], effective in cooperative multi-agent settings [Yu et al., 2022], and includes A2C as a special case [Huang et al., 2022b]. To alleviate policy cycling common in self-play, we incorporated FSP [Heinrich et al., 2015], which samples the opponent uniformly from

[3] `https://console.cloud.google.com/storage/browser/openspiel-data/bridge`
[4] `https://web2.acbl.org/documentlibrary/play/SP3%20(bk)%20single%20pages.pdf`

the checkpoints.

Reward function. Non-zero rewards are assigned only at the end of each game. The reward z is calculated by $z = \text{score}/7600$, where the score is derived from the double dummy solver (DDS)[5], a standard approximator for the playing phase, and 7600 represents the maximum absolute score.

DDS dataset. To bypass real-time DDS calculations during RL, we used a pre-computed DDS dataset from Pgx [Koyamada et al., 2023], containing 12.5M boards for training and 100K for evaluation.

Invalid action masking. This technique, aimed at preventing the agent from selecting illegal actions, has been widely adopted in AI research; including well-known implementations like Suphx [Li et al., 2020a], OpenAI Five [Berner et al., 2019], and AlphaStar [Vinyals et al., 2019], among others. For detailed insights, see [Huang and Ontañón, 2022].

Other details. Our PPO implementation is a fork of PureJaxRL[6] [Lu et al., 2022]. After conducting preliminary tests without using the test DDS data, we established the following hyperparameters: 8192 vectorized environments, a rollout length of 32, GAE λ of 0.95, a discount factor of 1.0, a clip ratio of 0.2, a value loss coefficient of 0.5, an entropy coefficient of 1.0×10^{-3}, a batch size of 1024, using Adam, with a learning rate of 1.0×10^{-6}. We trained the model for 10^4 PPO update steps, in which each step has 10 epochs over rollout data.

4.5 Results

4.5.1 Performance against WBridge5

To assess our model's effectiveness, trained as described in Section 4.4, we tested it against WBridge5, the leading benchmark in computer bridge. We utilized WBridge5

[5] `https://github.com/dds-bridge/dds`
[6] `https://github.com/luchris429/purejaxrl`

Table 4.2: Performance against WBridge5.

Paper	IMPs/board (±SE)	# games
Rong et al. [2019]	+0.25 (±N/A)	64
Gong et al. [2019]	+0.41 (±0.27)	64
Tian et al. [2020]	+0.63 (±0.22)	1K
Lockhart et al. [2020]	+0.85 (±0.05)	10K
Qiu et al. [2024]	+0.98 (±0.05)	10K
Ours	+**1.24** (±0.19)	1K

at its highest difficulty setting and with its native bidding system, which differs from the SAYC system used during our SL pretraining phase. The evaluation comprised 1K games, conducted over a day, reflecting the significant time needed because WBridge5 operates with a GUI and includes a playing phase.

The outcomes, detailed in Table 4.2, also compare our model's performance with that reported in prior studies. Our approach achieved an average of +1.24 International Match Points (IMPs)[7] per board against WBridge5 across these games, surpassing the previous SOTA performance of +0.85 IMPs/board by Lockhart et al. [Lockhart et al., 2020]. This improvement of 0.39 IMPs/board is significant in the context of computer bridge competitiveness [Ventos et al., 2017]. We also note that a concurrent study by Qiu et al. [2024] achieved +0.98 IMPs/board around the same time as our study[8].

4.5.2 Ablation study

Our method combines **SL** pretraining with **RL** model improvement through **FSP**. To dissect the contribution of each component, we tested variations of our model lacking one of these elements against WBridge5, with findings summarized in Figure 4.1. We used a learning rate 10 times larger for the model from scratch (i.e., w/o SL), as we found that it performs better than the original learning rate in those settings. We also trained the model from scratch with twice the number of steps to compensate for the

[7]Law 78B: `https://web2.acbl.org/documentlibrary/play/laws-of-duplicate-bridge.pdf`.

[8]Our study [Kita et al., 2024] was published in Aug. 2024, while Qiu et al. [2024] was published in Sep. 2024

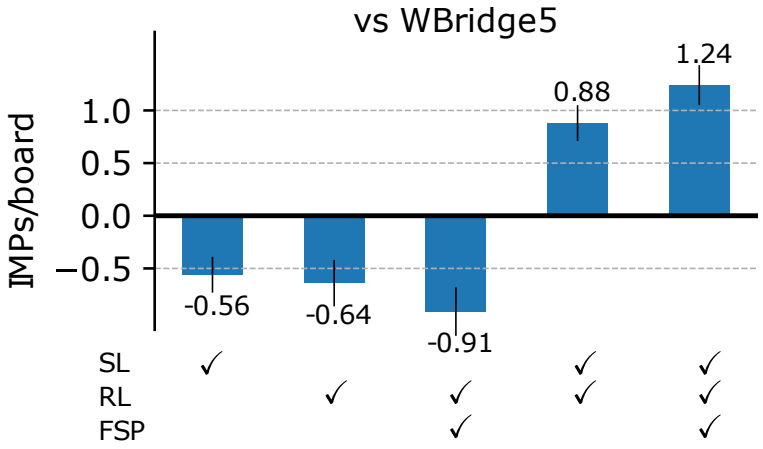


Figure 4.1: Ablation of each training component.

lack of SL pretraining.

Key observations include:

1. Removing SL pretraining drastically reduces performance, rendering the model unable to surpass the WBridge5 baseline.

2. Integrating FSP enhances results post-SL pretraining but is ineffective on its own.

The first insight challenges Gong et al. [2019]'s assertion that a model can outperform WBridge5 without SL pretraining, a claim we could not replicate despite extensive hyperparameter testing. We leave further exploration of this discrepancy for future work. We can offer a plausible explanation for the second observation. Starting from scratch, facing a random (or nearly random) opponent policy might slow the learning process. It is important to note that the bidding system used to create the dataset for SL pretraining differs from WBridge5's system. Therefore, the model enhanced with FSP is not just learning to outperform a version that mimics WBridge5.

To verify the mitigation of policy cycling by FSP, we organized a round-robin tournament among training checkpoints. Figure 4.2 shows the results. Unlike standard self-play, where some later-stage models might struggle against earlier ones, FSP consistently demonstrated the ability to outperform its predecessors, indicating its value in stable training.

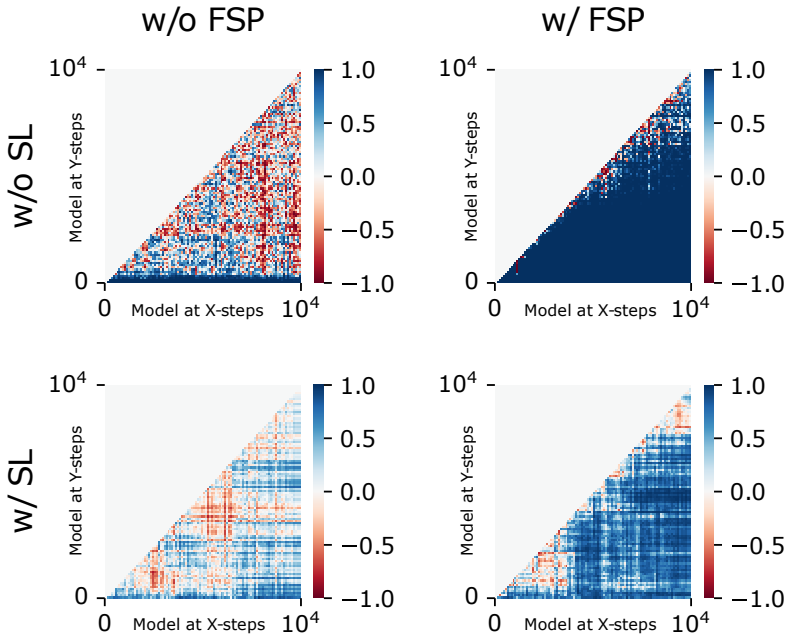


Figure 4.2: Comparison of usual self-play (w/o FSP) and FSP. Each item represents the IMPs/board scaled by tanh of the model at the X-steps against the model at the Y-steps (X is greater than Y).

4.5.3 Multi-GPU training

As we have demonstrated in Chapter 3, extending RL training to multi-GPU with hardware-accelerated environments is straightforward. We tested how long it takes to train a model with the same performance as the best model trained in Section 4.5.1 on NVIDIA A100 8 GPUs. As a result, we found that it would take less than an hour to beat the model presented in Section 4.5.1 from the pretrained model. Here, we changed the learning rate from 1.0×10^{-6} to 1.0×10^{-5} and used larger DDS datasets (100M).

4.6 Open-source software and models

Our straightforward approach, as detailed in Section 4.4, has demonstrated SOTA performance against the most recognized benchmark in computer bridge. While effective, this method is not specifically optimized for bridge's unique aspects, indicating potential areas for enhancement. To encourage continued advancement in bridge AI research,

we made our code, dataset, and trained models public as open-source resources.[9]

This new baseline aims to overcome certain limitations associated with the current WBridge5 benchmark:

1. **Slow WBridge5 evaluation.** Primarily designed for human interaction, WBridge5's evaluation process, which relies on GUI operations and includes a playing phase, is notably time-consuming and resource-intensive. This was highlighted by Rong et al. [2019], who manually tested their model against WBridge5.

2. **Potential weakness of WBridge5.** As evidenced in Table 4.2, recent advancements have significantly outperformed WBridge5, raising questions about the benchmark's current competitiveness. Moreover, fairness in evaluation is a concern since WBridge5 does not incorporate DDS strategies, although recent studies trained their models with DDS datasets.

By addressing these issues, our baseline not only offers a more efficient and equitable framework for assessment but also enhances the diversity of bidding systems under consideration.

4.7 Limitations, future work, and conclusion

Our study demonstrates that straightforward integration of existing techniques can outperform WBridge5, a leading benchmark in computer bridge bidding systems. However, our approach relies on SL pretraining to surpass WBridge5, contrasting with Gong et al. [2019], who claimed to achieve superior results without SL, using only RL from scratch. Exploring the reasons behind this discrepancy presents a valuable opportunity for future research.

Additionally, our methodology, while effective, is not specifically designed with the unique aspects of bridge in mind. This suggests there may be room for further

[9] Our code with multi-gpu training and models are available at: `https://github.com/sotetsuk/brl`. DDS dataset is available at: `https://huggingface.co/datasets/sotetsuk/dds_dataset`.

optimization and refinement tailored to bridge's strategic complexities.

Despite these limitations, we are confident our work lays a solid foundation for subsequent studies in bridge AI. By providing our code and models as open-source resources, we aim to facilitate the development of more advanced AI systems capable of exceeding human expertise in bridge.

Chapter 5

A Potential Disadvantage of Massive Batching

In this chapter, we discuss the potential disadvantage of massive batching. While batching improves computational efficiency, the performance may degrade compared to the sequential case due to delayed feedback. We consider the pure exploration problem in stochastic bandit tasks, which is a simple form of RL to investigate this potential disadvantage. We show that one of the most popular algorithms, Sequential Halving (SH; Karnin et al. [2013]), does not actually degrade performance under realistic conditions. Finally, we combine batched SH with Monte Carlo tree search (MCTS) and empirically demonstrated its effectiveness in 9x9 Go.

5.1 Introduction

In this chapter, we consider the pure exploration problem in the field of stochastic multi-armed bandits, which aims to identify the best arm within a given budget [Audibert et al., 2010]. Specifically, we concentrate on the *fixed-size batch pulls* setting, where we pull a fixed number of arms simultaneously. Batch computation plays a central role in improving computational efficiency, especially in large-scale bandit applications where reward computation can be expensive. For instance, consider applying this

to tree search algorithms like Monte Carlo tree search [Tolpin and Shimony, 2012]. The reward computation here typically involves the value network evaluation [Silver et al., 2016, 2017], which can be computationally expensive. By leveraging batch computation and hardware accelerators (e.g., GPUs), we can significantly reduce the computational cost of the reward computation. However, while batch computation enhances computational efficiency, its performance (e.g., simple regret) may not match that of sequential computation with the same total budget, due to delayed feedback reducing adaptability. Therefore, the objective here is to develop a pure exploration algorithm that maintains its performance regardless of the batch size.

We focus on the *Sequential Halving* (SH) algorithm [Karnin et al., 2013], a popular and well-analyzed pure exploration algorithm. Due to its simplicity, efficiency, and lack of task-dependent hyperparameters, SH finds practical applications in, but not limited to, hyperparameter tuning [Jamieson and Talwalkar, 2016], recommendation systems [Aziz et al., 2022], and state-of-the-art AlphaZero [Silver et al., 2018] and MuZero [Schrittwieser et al., 2020] family [Danihelka et al., 2022]. In this chapter, we aim to extend SH to a batched version that matches the original SH algorithm's performance, even with large batch sizes. To date, Jun et al. [2016] introduced a simple batched extension of SH and reported that it performed well in their experiments. However, the theoretical properties of batched SH have not yet been well-studied in the setting of fixed-size batch pulls.

We consider two simple and natural batched variants of SH (Section 5.3): *Breadth-first Sequential Halving* (BSH) and *Advance-first Sequential Halving* (ASH). We introduce BSH as an intermediate step to understanding ASH, which is our main focus. Our main contribution is providing a theoretical guarantee for ASH (Section 5.4), showing that *it is algorithmically equivalent to SH as long as the batch budget is not extremely small* — For example, in a 32-armed stochastic bandit problem, ASH can match SH's choice with 100K sequential pulls using just 20 batch pulls, each of size 5K. This means that ASH can achieve the same performance as SH with significantly

fewer pulls when the batch size is reasonably large. Moreover, one can understand the theoretical properties of ASH using the theoretical properties of SH, which have been well-studied [Karnin et al., 2013, Zhao et al., 2023]. In our experiments, we validate our claim by comparing the behavior of ASH and SH (Section 5.5.1) and analyze the behavior of ASH with the extremely small batch budget as well (Section 5.5.2).

5.2 Preliminary

Pure Exploration Problem. Consider a pure exploration problem involving n arms and a budget T. We define a reward matrix $\mathcal{R} \in [0,1]^{n \times T}$, where each element $\mathcal{R}_{i,j} \in [0,1]$ represents the reward of the j-th pull of arm $i \in [n] \coloneqq \{1, \ldots, n\}$, with j being counted independently for each arm. Each element in the i-th row is an i.i.d. sample from an unknown reward distribution of i-th arm with mean μ_i. Without loss of generality, we assume that $1 \geq \mu_1 \geq \mu_2 \geq \ldots \geq \mu_n \geq 0$. In the standard sequential setting, a pure exploration algorithm sequentially observes T elements from $\mathcal{R}$ by pulling arms one by one for T times. The algorithm then selects one arm as the best arm candidate. Note that we only consider deterministic pure exploration algorithms in this chapter. Such an algorithm can be characterized by a mapping $\pi : [0,1]^{n \times T} \to [n]$ that takes $\mathcal{R}$ as input and outputs the selected arm a_T. The natural performance measure in pure exploration is the *simple regret*, defined as $\mathbb{E}_{\mathcal{R}}[\mu_1 - \mu_{a_T}]$ [Bubeck et al., 2009], which compares the performance of the selected arm a_T with the best arm 1.

Sequential Halving (SH; Karnin et al. [2013]) is a sequential elimination algorithm designed for the pure exploration problem. It begins by initializing the set of best arm candidates as $\mathcal{S}_0 \coloneqq [n]$. In each of the $\lceil \log_2 n \rceil$ rounds, the algorithm halves the set of candidates (i.e., $|\mathcal{S}_{r+1}| = \lceil |\mathcal{S}_r|/2 \rceil$) until it narrows down the candidates to a single arm in $\mathcal{S}_{\lceil \log_2 n \rceil}$. During each round $r \in \{0, \ldots, \lceil \log_2 n \rceil - 1\}$, the arms in the active arm set $\mathcal{S}_r$ are pulled equally $J_r \coloneqq \left\lfloor \frac{T}{|\mathcal{S}_r| \lceil \log_2 n \rceil} \right\rfloor$ times, and the total budget

Algorithm 2 SH: Sequential Halving [Karnin et al., 2013]

input number of arms: n, budget: T
initialize best arm candidates $\mathcal{S}_0 \coloneqq [n]$
for round $r = 0, \ldots, \lceil \log_2 n \rceil - 1$ **do**
    pull each arm $a \in S_r$ for $J_r = \left\lfloor \frac{T}{|\mathcal{S}_r| \lceil \log_2 n \rceil} \right\rfloor$ times
    $\mathcal{S}_{r+1} \leftarrow$ top-$\lceil |\mathcal{S}_r|/2 \rceil$ arms in $\mathcal{S}_r$ w.r.t. the empirical rewards
return the only arm in $\mathcal{S}_{\lceil \log_2 n \rceil}$

consumed for round r is $T_r \coloneqq J_r \times |\mathcal{S}_r|$. The SH algorithm is described in Algorithm 2. We denote the mapping induced by the SH algorithm as π_{SH}. It has been shown that the simple regret of SH satisfies $\mathbb{E}_{\mathcal{R}}[\mu_1 - \mu_{a_T}] \leq \tilde{\mathcal{O}}(\sqrt{n/T})$, where $\tilde{\mathcal{O}}(\cdot)$ ignores the logarithmic factors of n [Zhao et al., 2023]. Note that the consumed budget $\sum_{r<\lceil \log_2 n \rceil} T_r$ might be less than T. We assume that the remaining budget is consumed equally by the last two arms in the final round.

5.3 Batch SH algorithms

We consider the fixed-size batch pulls setting, where we simultaneously pull b arms for B times, with b being the fixed batch size and B being the batch budget [Jun et al., 2016]. The standard sequential case corresponds to $b = 1$ and $B = T$. Our interest is to compare the performance of the batch SH algorithms with a large batch size b and a small batch budget B to that of the standard SH algorithm when pulling sequentially T times. Therefore, we compare the performance of the batch SH algorithms under the assumption that $T = b \times B$ holds, so that the total budget is the same in both the sequential and batch settings. In this section, we first reconstruct the SH algorithm so that it can be easily extended to the batched setting (Section 5.3.1). Then, we consider *Breadth-first Sequential Halving* (BSH), one of the simplest batched extensions of SH, as an intermediate step (Section 5.3.2). Finally, we introduce *Advance-first Sequential Halving* (ASH) as a further extension (Section 5.3.3).

Algorithm 3 SH [Karnin et al., 2013] implementation with target pulls $L^{\mathbf{B}}/L^{\mathbf{A}}$

input number of arms: n, budget: T
initialize empirical mean $\bar{\mu}_a \coloneqq 0$ and arm pulls $N_a \coloneqq 0$ for all $a \in [n]$
for $t = 0, \dots, T-1$ **do**
    let $\mathcal{A}_t$ be $\{a \in [n] \mid N_a = L_t\}$ ▷ L_t is either $L_t^{\mathbf{B}}$ (5.1) or $L_t^{\mathbf{A}}$ (5.2)
    pull arm $a_t \coloneqq \mathrm{argmax}_{a \in \mathcal{A}_t} \bar{\mu}_a$
    update $\bar{\mu}_{a_t}$ and $N_{a_t} \leftarrow N_{a_t} + 1$
return $\mathrm{argmax}_{a \in [n]}(N_a, \bar{\mu}_a)$

Algorithm 4 *Breadth-first* target pulls $L^{\mathbf{B}}$

input number of arms: n, budget: T
initialize empty $L^{\mathbf{B}}$, $K \coloneqq n$, $J \coloneqq 0$
for $r = 0, \dots \lceil \log_2 n \rceil - 1$ **do**
    for ▷ $j = 0, \dots, J_r - 1$ **do**
        for ▶ $k = 0, \dots, K-1$ **do**
            append $J + j$ to $L^{\mathbf{B}}$
    $K \leftarrow \lceil K/2 \rceil$ and $J \leftarrow J + J_r$
return $L^{\mathbf{B}}$ ▷ `(0,0,0,...)`

Algorithm 5 *Advance-first* target pulls $L^{\mathbf{A}}$

input number of arms: n, budget: T
initialize empty $L^{\mathbf{A}}$, $K \coloneqq n$, $J \coloneqq 0$
for $r = 0, \dots \lceil \log_2 n \rceil - 1$ **do**
    for ▶ $k = 0, \dots, K-1$ **do**
        for ▷ $j = 0, \dots, J_r - 1$ **do**
            append $J + j$ to $L^{\mathbf{A}}$
    $K \leftarrow \lceil K/2 \rceil$ and $J \leftarrow J + J_r$
return $L^{\mathbf{A}}$ ▷ `(0,1,2,...)`

5.3.1 SH implementation with target pulls

Since BSH/ASH is a natural batched extension of SH, we first reconstruct the implementation of the SH algorithm as Algorithm 3 so that it can be easily extended to BSH/ASH. Note that, here, the operation $\mathrm{argmax}_{x \in \mathcal{X}}(\ell_x, m_x)$ selects the element $x \in \mathcal{X}$ that maximizes ℓ_x first. If multiple elements achieve this maximum, it then selects among these the one that maximizes m_x. At the t-th arm pull, SH selects the arm a_t that has the highest empirical reward $\bar{\mu}_a$ among the candidates $\mathcal{A}_t$:

$$a_t \coloneqq \mathrm{argmax}_{a \in \mathcal{A}_t} \bar{\mu}_a,$$

where $\mathcal{A}_t \coloneqq \{a \in [n] \mid N_a = L_t\}$ are the candidates at the t-th arm pull, N_a is the total number of pulls of arm a, and L_t is the number of *target pulls* at t, defined as

either **breadth-first** manner

$$L_t^{\mathbf{B}} \coloneqq \underbrace{\sum_{r'<r(t)} J_{r'}}_{\text{pulls before } r(t)} + \underbrace{\left\lfloor \frac{t - \sum_{r'<r(t)} T_{r'}}{|\mathcal{S}_{r(t)}|} \right\rfloor}_{\text{pulls in } r(t)}, \tag{5.1}$$

or **advance-first** manner

$$L_t^{\mathbf{A}} \coloneqq \underbrace{\sum_{r'<r(t)} J_{r'}}_{\text{pulls before } r(t)} + \underbrace{\left(\left(t - \sum_{r'<r(i)} T_{r'} \right) \bmod J_{r(t)} \right)}_{\text{pulls in } r(t)}, \tag{5.2}$$

where $r(t)$ is the round of the t-th arm pull. This $L_t^{\mathbf{B}}/L_t^{\mathbf{A}}$ represents the cumulative number of pulls of the arm selected at the t-th pull before the t-th arm pull. We omitted the dependency on n and T for simplicity. The definition of $L_t^{\mathbf{B}}/L_t^{\mathbf{A}}$ is somewhat complicated, and it may be straightforward to write down the algorithm that constructs $L^{\mathbf{B}} \coloneqq (L_0^{\mathbf{B}}, \ldots, L_T^{\mathbf{B}})$ and $L^{\mathbf{A}} \coloneqq (L_0^{\mathbf{A}}, \ldots, L_T^{\mathbf{A}})$ as shown in Algorithm 4 and Algorithm 5, respectively. Note that the choice between $L^{\mathbf{B}}$ and $L^{\mathbf{A}}$ is arbitrary and does not affect the behavior of SH — as long as the arm pull is sequential (not batched). Python code for this SH implementation is available in Section B.1. Note that using target pulls to implement SH is natural and not new. For example, Mctx[1] [Babuschkin et al., 2020] has a similar implementation.

5.3.2 BSH: Breadth-first Sequential Halving

Now, we extend SH to BSH, in which we select arms so that the number of pulls of each arm becomes as equal as possible using $L^{\mathbf{B}}$. Note that $L^{\mathbf{B}}$ uses $T = b \times B$ as the scheduled total budget. When pulling arms in a batch, we need to consider not only the number of pulls of the arms but also the number of scheduled pulls in the current batch. Therefore, we introduce *virtual arm pulls* M_a, the number of scheduled pulls of arm a in the current batch. For each batch pull, we sequentially select b arms with

[1] `https://github.com/google-deepmind/mctx`

Algorithm 6 BSH: Breadth-first Sequential Halving

input number of arms: n, batch size: b, batch budget: B
initialize counter $t \coloneqq 0$, empirical mean $\bar{\mu}_a \coloneqq 0$, and arm pulls $N_a \coloneqq 0$ for all $a \in [n]$
for B times **do**
    initialize empty batch $\mathcal{B}$ and virtual arm pulls $M_a = 0$ for all $a \in [n]$
    for b times **do**
        let $\mathcal{A}_t$ be $\{a \in [n] \mid N_a + M_a = L_t^{\mathbf{B}}\}$
        push $a_t \coloneqq \mathrm{argmax}_{a \in \mathcal{A}_t} \bar{\mu}_a$ to $\mathcal{B}$
        update $t \leftarrow t + 1$ and $M_{a_t} \leftarrow M_{a_t} + 1$
    batch pull arms in $\mathcal{B}$
    update $\bar{\mu}_a$ and $N_a \leftarrow N_a + M_a$ for all $a \in \mathcal{B}$
return $\mathrm{argmax}_{a \in [n]}(N_a, \bar{\mu}_a)$

Algorithm 7 ASH: Advance-first Sequential Halving

input number of arms: n, batch size: b, batch budget: B
initialize counter $t \coloneqq 0$, empirical mean $\bar{\mu}_a \coloneqq 0$, and arm pulls $N_a \coloneqq 0$ for all $a \in [n]$
for B times **do**
    initialize empty batch $\mathcal{B}$ and virtual arm pulls $M_a = 0$ for all $a \in [n]$
    for b times **do**
        let $\mathcal{A}_t$ be $\{a \in [n] \mid N_a + M_a = L_t^{\mathbf{A}}\}$ ▷ BSH uses $L_t^{\mathbf{B}}$ instead
        push $a_t \coloneqq \mathrm{argmax}_{a \in \mathcal{A}_t}(N_a, \bar{\mu}_a)$ to $\mathcal{B}$ ▷ BSH uses $\mathrm{argmax}_{a \in \mathcal{A}_t} \bar{\mu}_a$ instead
        update $t \leftarrow t + 1$ and $M_{a_t} \leftarrow M_{a_t} + 1$
    batch pull arms in $\mathcal{B}$
    update $\bar{\mu}_a$ and $N_a \leftarrow N_a + M_a$ for all $a \in \mathcal{B}$
return $\mathrm{argmax}_{a \in [n]}(N_a, \bar{\mu}_a)$

the highest empirical rewards from the candidates $\{a \in [n] \mid N_a + M_a = L_t^{\mathbf{B}}\}$ and pull them as a batch. The BSH algorithm is described in Algorithm 6. BSH is similar to a batched extension of SH introduced in Jun et al. [2016] in the sense that it selects arms so that the number of pulls of each arm becomes as equal as possible.

5.3.3 ASH: Advance-first Sequential Halving

We further extend SH to ASH in a manner similar to BSH. The ASH algorithm is described in Algorithm 7. Figure 5.1 shows the pictorial representation of BSH and ASH. Python code for this ASH implementation is available in Section B.1. The differences between BSH and ASH are that:

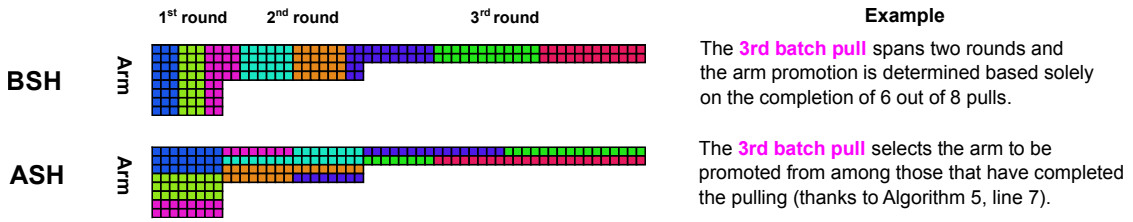


Figure 5.1: Pictorial representation of *breadth-first* SH (BSH; Section 5.3.2) and *advance-first* SH (ASH; Section 5.3.3) for an 8-armed bandit problem. Batch size b is 24 and batch budget B is 8. The same color indicates the same batch pull — For example, in the first batch pull (blue), BSH pulls each of the 8 arms 3 times, while ASH pulls 3 arms 8 times each. BSH selects arms so that the number of pulls of each active arm becomes as equal as possible, while ASH selects arms so that once an arm is selected, it is pulled until the budget for the arm in the round is exhausted.

1. ASH selects arms in *advance-first* manner using $L^{\mathbf{A}}$ instead of $L^{\mathbf{B}}$ (line 6), and

2. ASH considers not only the empirical rewards $\bar{\mu}_a$ but also the number of actual pulls N_a when selecting arms in a batch (line 7).

The second difference ensures that, when the batch spans two rounds, the arm to be promoted is selected from the arms that have completed pulling (e.g., see the 3rd batch pull in Figure 5.1). Note that this second modification is not useful for BSH. Let $\pi_{\mathrm{ASH}} : [0,1]^{n \times T} \to [n]$ be the mapping induced by the ASH algorithm. In Section 5.4, we will show that ASH is algorithmically equivalent to SH with the same total budget $T = b \times B$ — π_{ASH} is identical to π_{SH}.

5.4 Algorithmic equivalence of SH and ASH

This section presents a theoretical guarantee for the ASH algorithm.

Theorem 1 *Given a stochastic bandit problem with $n \geq 2$ arms, let $b \geq 2$ be the batch size and B be the batch budget satisfying $B \geq \max\{4, \frac{n}{b}\}\lceil \log_2 n \rceil$. Then, the ASH algorithm (Algorithm 7) is algorithmically equivalent to the SH algorithm (Algorithm 3) with the same total budget $T = b \times B$ — the mapping π_{ASH} is identical to π_{SH}.*

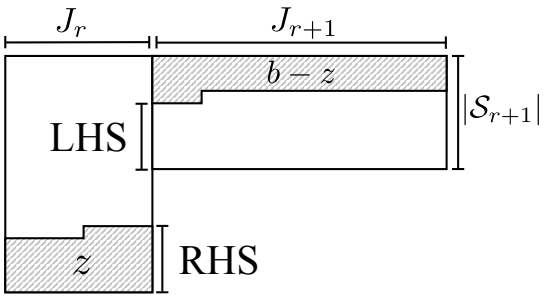


Figure 5.2: Visualization of inequality (5.3).

Proof sketch A key observation is that ASH and SH differ only when a batch pull spans two rounds, like the 3rd batch pull in Figure 5.1. In this case, ASH may promote an incorrect arm to the next round that would not have been promoted in SH. We can prove that such *incorrect promotion* does not occur under the condition $B \geq \max\{4, \frac{n}{b}\}\lceil \log_2 n \rceil$. This is done by demonstrating that the inequality (5.3) holds for any $z < b$, the number of pulls for the current round r in the batch. Figure 5.2 illustrates (5.3).

Proof. The condition $B \geq \max\{4, \frac{n}{b}\}\lceil \log_2 n \rceil$ is divided into two separate conditions:

$$B \geq \frac{n}{b}\lceil \log_2 n \rceil, \tag{C1}$$

and

$$B \geq 4\lceil \log_2 n \rceil. \tag{C2}$$

We focus on the scenario where a batch pull spans two rounds. In this case, let $z < b$ be the number of pulls that consume the budget for round r, and $b - z$ be the number of pulls that consume the budget for round $r + 1$. The following proposition is demonstrated: $\forall n \geq 2, \forall b \geq 2$, $\forall r < \lceil \log_2 n \rceil - 1$, $\forall z < b$, if (C1) and (C2) hold, then

$$|\mathcal{S}_{r+1}| - \left\lceil \frac{b-z}{J_{r+1}} \right\rceil \geq \left\lceil \frac{z}{J_r} \right\rceil. \tag{5.3}$$

The left-hand side (LHS) of (5.3) represents the number of arms promoting to the subsequent round post-batch pull, whereas the right-hand side (RHS) quantifies the arms pending completion of their pulls at the batch pull juncture. This inequality, if satisfied, ensures that, even when a batch spans two rounds, arms supposed to advance to the next round in SH are not left behind in ASH, i.e., no incorrect promotion occurs. Considering the scenario where $z = b - 1$ suffices, as it represents the worst-case condition. Let $x := |S_r| \geq 3$ for the given $r < \lceil \log_2 n \rceil - 1$. Two cases are considered. **Case 1:** when $n \leq 4b$. Given that $J_r = \left\lfloor \frac{b \times B}{x \lceil \log_2 n \rceil} \right\rfloor \geq \lfloor 4b/x \rfloor$ as derived from (C2), it is sufficient to show

$$\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{b-1}{\lfloor 4b/x \rfloor} \right\rceil \tag{5.4}$$

in $x \in [3, 4b]$. This assertion is directly supported by Lemma 1. **Case 2:** when $4b < n$. Given that $J_r = \left\lfloor \frac{b \times B}{x \lceil \log_2 n \rceil} \right\rfloor \geq \lfloor n/x \rfloor$ as derived from (C1), it is sufficient to show $\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{n/4-1}{\lfloor n/x \rfloor} \right\rceil$ in $x \in [3, n]$. This conclusion follows by the same reasoning applied in Case 1. □

Lemma 1 *For any integer $b \geq 2$, the inequality $\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{b-1}{\lfloor 4b/x \rfloor} \right\rceil$ holds for all integers $x \in [3, 4b]$.*

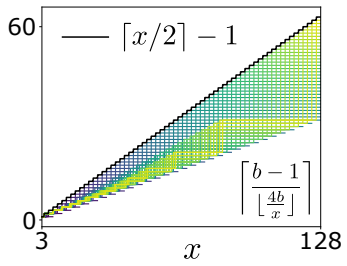


Figure 5.3: Visualization of Lemma 1.

The proof of Lemma 1 is in Section B.2. Here, we provide the visualization of (5.4) in Fig. 5.3 to intuitively show that Lemma 1 holds. Each colored line represents the RHS for different $b \leq 32$. One can see that the LHS is always greater than the RHS for any $x \in [3, 4b]$.

Remark 1 The condition (C1) is common to both SH and ASH — SH implicitly assumes $T \geq n\lceil \log_2 n \rceil$ as the minimum condition to execute. This is because we need to pull each arm at least once in the first round (i.e., $J_1 \geq 1$). With the same argument, the batch budget B must satisfy (C1). On the other hand, (C2) is specific to ASH and is required to ensure the equivalence. As we discuss in the Section 5.4.1, we argue that this additional (C2) is not practically problematic.

Remark 2 Note that the condition (C2) is tight; Theorem 1 does not hold even if $B \geq \alpha\lceil \log_2 n \rceil$ for any positive value $\alpha < 4$.

Proof. We aim to demonstrate the existence of a value x such that $\left\lceil \frac{x}{2} \right\rceil - 1 - \left\lceil \frac{b-1}{\lfloor \alpha b/x \rfloor} \right\rceil < 0$ when $n \leq \alpha b$. Consider the case when $x = 4$. In this scenario, the LHS of the inequality can be rewritten as $1 - \left\lceil \frac{b-1}{\lfloor \alpha b/4 \rfloor} \right\rceil \leq 1 - \frac{b-1}{\lfloor \alpha b/4 \rfloor} \leq 1 - \frac{4}{\alpha}\frac{b-1}{b} \to 1 - \frac{4}{\alpha}$ as $b \to \infty$. As $\alpha < 4$, it follows that LHS < 0 for sufficiently large values of b. □

Remark 3 When b is sufficiently large, the minimum B that satisfies both (C1) and (C2) is $4\lceil \log_2 n \rceil$. Theorem 1 implies that for arbitrarily large target budget T, ASH can achieve the same performance as SH by increasing the batch size b without increasing the batch budget B from $4\lceil \log_2 n \rceil$ — ASH guarantees its scalability in batch computation.

Remark 4 Theorem 1 allows us to understand the properties of ASH based on existing theoretical research on SH, such as the simple regret bound [Zhao et al., 2023].

5.4.1 Discussion on the conditions

To show that SH and ASH are algorithmically equivalent, we used an additional condition (C2) of $\mathcal{O}(\log n)$. However, we argue that this condition is not practically problematic because the condition (C1), the minimum condition required to execute (unbatched) SH, is dominant ($\mathcal{O}(n \log n)$). This condition (C1) is dominant over (C2) as shown in Figure 5.4. We can see that the condition (C2) only affects the algorithm

when the batch size is sufficiently larger than the number of arms ($b \gg n$). This is a reasonable result, meaning that we cannot guarantee the equivalent behavior to SH with an extremely small batch budget, such as $B = 1$. On the other hand, if the user secures the minimum budget $B = 4\lceil \log_2 n \rceil$ that depends only on the number of arms n and increases only logarithmically, regardless of the batch size b, they can increase the batch size arbitrarily and achieve the same result as when SH is executed sequentially with the same total budget, with high computational efficiency.

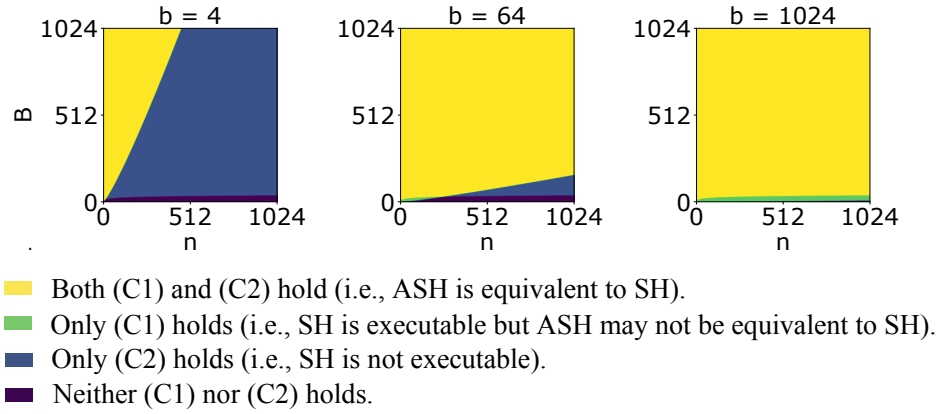


Figure 5.4: Visualization of conditions (C1) and (C2).

5.5 Empirical validation

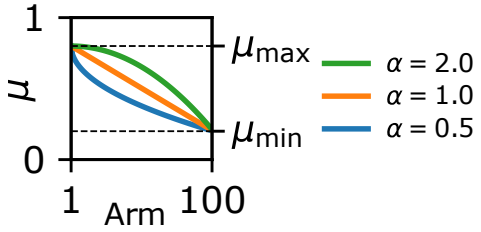


Figure 5.5: Polynomial(α) bandit problem instances.

We conducted experiments to empirically demonstrate that ASH maintains its performance for large batch size b, in comparison to its sequential counterpart SH. To evaluate this, we utilized a polynomial family parameterized by α as a representative bandit problem instance, where the reward gap $\Delta_a := \mu_1 - \mu_a$ follows a polynomial distribution with parameter α: $\Delta_a \propto (a/n)^\alpha$ [Jamieson et al., 2013, Zhao et al., 2023].

This choice is motivated by the observation that real-world applications exhibit polynomially distributed reward gaps, as mentioned in [Zhao et al., 2023]. In our study, we considered three different values of α (0.5, 1.0, and 2.0) to capture various reward distributions (see Figure 5.5). Additionally, we characterized each bandit problem instance by specifying the minimum and maximum rewards, denoted as $\mu_{\min}$ and $\mu_{\max}$ respectively. Hence, we denote a bandit problem instance as $\mathcal{T}(n, \alpha, \mu_{\min}, \mu_{\max})$.

We also implemented a simple batched extension of SH introduced by Jun et al. [2016] as a baseline for comparison. We refer to this algorithm as Jun+16. The implementation of Jun+16 is described in Algorithm 8. Jun et al. [2016] did not provide a theoretical guarantee for Jun+16, but it has shown performance comparable to or better than their proposed algorithm in their experiments.

Algorithm 8 Batched Sequential Halving introduced in Jun et al. [2016]

input number of arms: n, batch budget: B, batch size: b
initialize best arm candidates $\mathcal{S}_0 \coloneqq [n]$
for round $r = 0, \dots, \lceil \log_2 n \rceil - 1$ **do**
    for $\left\lfloor B / \lceil \log_2 n \rceil \right\rfloor$ times **do**
        select batch actions $\mathcal{B}$ so that the number of pulls of each arm in $\mathcal{S}_r$ is as equal as possible
        pull arms $\mathcal{B}$ in the batch
    $\mathcal{S}_{r+1} \leftarrow$ top-$\lceil |\mathcal{S}_r|/2 \rceil$ arms in $\mathcal{S}_r$ w.r.t. the empirical rewards
return the only arm in $S_{\lceil \log_2 n \rceil}$

5.5.1 Large batch budget scenario: $B \geq 4\lceil \log_2 n \rceil$

First, we empirically confirm that, as we claimed in Section 5.4, ASH is indeed equivalent to SH under the condition (C2). We generated 10K instances of bandit problems and applied ASH and SH to each instance with 100 different seeds. We randomly sampled n from $\{2, \dots, 1024\}$, α from $\{0.5, 1.0, 2.0\}$, and $\mu_{\min}$ and $\mu_{\max}$ from $\{0.1, 0.2, \dots, 0.9\}$. For each instance $\mathcal{T}(n, \alpha, \mu_{\min}, \mu_{\max})$, we randomly sampled the batch budget $B \leq 10\lceil \log_2 n \rceil$ and the batch size $b \leq 5n$ so that the condition (C1) and (C2) are satisfied. *As a result, we confirmed that the selected arms of ASH and SH*

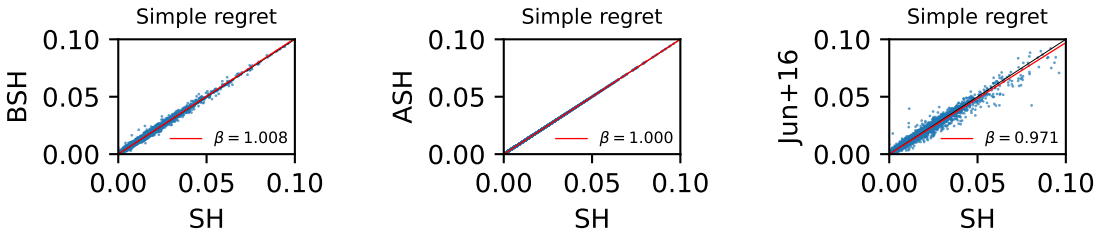


Figure 5.6: Single regret of BSH, ASH, and Jun+16 against SH when $B \geq 4\lceil \log_2 n \rceil$.

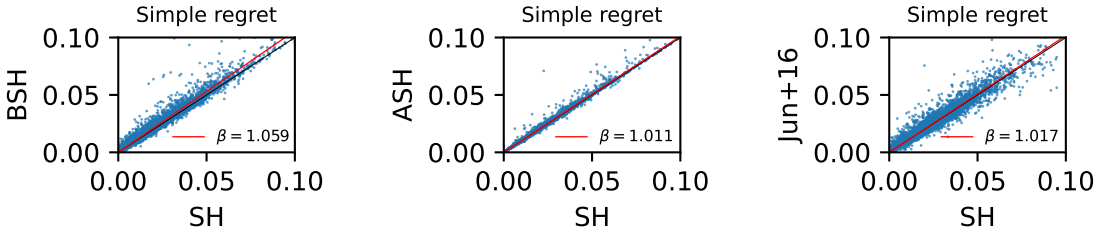


Figure 5.7: Single regret of BSH, ASH, and Jun+16 against SH when $B < 4\lceil \log_2 n \rceil$.

are identical in all 10K instances and 100 seeds for each instance. We also conducted the same experiment for BSH and Jun+16. We plotted the simple regret of BSH, ASH, and Jun+16 against SH in Figure 5.6. There are 10K instances, and each point represents the average simple regret of 100 seeds for each instance. To compare the performance, we fitted a linear regression model to the simple regret of BSH, ASH, and Jun+16 against SH as $y = \beta x$, where y is the simple regret of BSH, ASH, or Jun+16, x is the simple regret of SH. The slope β is estimated by the least squares method. The estimated slope β is 1.008 for BSH, 1.000 for ASH, and 0.971 for Jun+16, which indicates that the simple regret of ASH, BSH, and Jun+16 is comparable to SH on average.

5.5.2 Small batch budget scenario: $B < 4\lceil \log_2 n \rceil$

Next, we examined the performances of BSH, ASH, and Jun+16 against SH when the additional condition (C2) is not satisfied, i.e., when the batch budget is extremely small $B < 4\lceil \log_2 n \rceil$ and thus Theorem 1 does not hold. We conducted the same experiment as in Section 5.5.1 except the batch budget $B < 4\lceil \log_2 n \rceil$. We sampled B

so that B is larger than the number of rounds. The results are shown in Figure 5.7. The slope β is estimated as 1.059 for BSH, 1.011 for ASH, and 1.017 for Jun+16. All the estimated slopes are worse than when $B \geq 4\lceil \log_2 n \rceil$. However, the estimated slopes are still close to 1, which indicates that while we do not have a theoretical guarantee, the performance of BSH, ASH, and Jun+16 is comparable to SH on average.

5.6 Application to Monte Carlo tree search

We have theoretically and empirically shown that SH is robust against performance degradation due to batch computation. This allows us to enjoy the computational efficiency improvement by batch computation while suppressing performance degradation. In this section, we consider applying this property of SH to more practical applications.

Monte Carlo tree search (MCTS) is a representative planning algorithm, as exemplified by its application to the AlphaZero family [Silver et al., 2016, 2017, 2018, Schrittwieser et al., 2020]. On the other hand, the MCTS algorithm is inherently sequential — It has been pointed out that it is not suitable for parallel computation [Liu et al., 2020, Hafner et al., 2021]. Here, we review the MCTS algorithm and the most popular method for parallelizing MCTS, *virtual loss* [Chaslot et al., 2008, Segal, 2010], and experimentally examine the performance of the batch MCTS algorithm using SH.

Monte Carlo tree search (MCTS). Here, we focus on MCTS using policy net and value net as used in the AlphaZero family. There are various variations of MCTS, but basically, it is an algorithm that grows a tree from the root node corresponding to the current state (e.g., board) as follows (see Silver et al. [2017]):

1. **Selection.** Traverse the tree based on a selection criterion at each node from the root node to reach the leaf node.

2. **Expand and evaluate.** When reaching the leaf node, apply the action selected based on the prior from the state corresponding to the leaf node (e.g., board) and

transition to the next state. Then, add the state after the transition as a new leaf node to the tree and calculate the prior and value of the state corresponding to the node by policy and value net.

3. **Backup.** Backpropagate the leaf value from the new leaf node to the root. In two-player zero-sum games like Go, the value is multiplied by -1.0 when passed to the parent node.

As a convention, this cycle is called one *simulation*, and this is repeated for the number of simulations (i.e., budget). Finally, it is common to select the action with the most visits at the root node.

Virtual loss. A typical bottleneck in MCTS is the *expand and evaluate* step, which involves the NN inference and environment state transition. This step can be easily batched by batch NN inference and batch state transition (by Pgx). Therefore, the problem is how to select the leaf nodes to expand in batch. *Virtual loss* [Chaslot et al., 2008, Segal, 2010] is a technique to avoid selecting the same leaf node and makes it possible to parallelize (or vectorize) the *expand and evaluate* step. In virtual loss, a batch size of leaf nodes is sequentially selected. Given the current search tree, a leaf node is selected in the usual *selection* step, and the value is temporarily set as if it lost at the leaf node, and then backed up. This makes it less likely to select the same leaf node. This is repeated until the batch size of leaf nodes is selected. The *virtual loss* is canceled out during the *backup* step. Virtual loss is a popular choice, but it has been pointed out that it has problems such as excessive diversity [Liu et al., 2020].

Combining SH into MCTS. Here, we consider applying SH by treating the action selection at the root node as a pure exploration problem of the multi-armed bandit problem. Using SH for the action selection at the root node of MCTS is not new [Cazenave, 2014, Danihelka et al., 2022], but we aim to demonstrate that using SH for the action selection at the root node is particularly effective in batch computa-

tion. Note that while we treat the action selection at the root node as a pure exploration problem, the values of the child nodes are updated as the tree grows. Therefore, it is necessary to be aware that the theoretical properties of ASH, as shown in Section 5.4, do not hold because the i.i.d. assumption of the arm reward distribution does not hold.

Experimental setup. We implemented two batch MCTS algorithms:

- **virtual loss**: A batch version of the MCTS algorithm used in MuZero [Schrittwieser et al., 2020] with virtual loss applied.
- **virtual loss + SH**: A batch MCTS algorithm with SH at the root node in addition to the virtual loss.

The only difference between the two algorithms is the action selection at the root node. We implemented the algorithms by forking the Mctx library [2]. The SH implementation also followed the Mctx implementation. The score function used by SH is the same as the Gumbel-MuZero [Danihelka et al., 2022]. We conducted experiments using the 9x9 Go environment and the baseline model developed in Chapter 3. First, we prepared an evaluation opponent by applying the Mctx's MuZero version MCTS to the baseline model with a 100 simulation budget. We evaluated the two batch MCTS algorithms by playing against the evaluation opponent with a fixed total budget of 1024 and varying the batch size and batch budget. To guarantee the diversity of evaluations, we used the sampling from the policy net for the first 20 steps of each game and then used the MCTS algorithms for playing. Each algorithm played 512 games for each batch size and batch budget setting, and we compared the average performance.

Results. The results are shown in Figure 5.8. As expected, when the batch size b is 1 and the batch budget B is 1024, i.e., when MCTS is executed sequentially without batch computation, both algorithms significantly outperform the evaluation opponent because the budget is sufficiently larger than the opponent's budget ($T = 100$). On

[2] `https://github.com/google-deepmind/mctx`

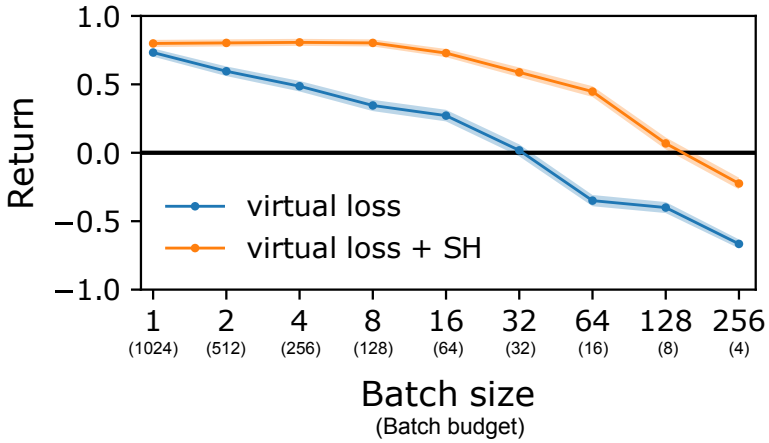


Figure 5.8: Performance of the batch MCTS algorithms w/ and w/o SH against the (unbatched) MCTS opponent with $T = 100$. The total budget of batch MCTS algorithms is fixed to $T = 1024$. The batch size b (and the corresponding batch budget B) is varied from 1 to 256 (and 1024 to 4). The shaded area represents the standard error of the mean over 512 games.

the other hand, as the batch size b increases, the performance of both *virtual loss* and *virtual loss + SH* decreases. The performance degradation of *virtual loss + SH* is less severe than that of *virtual loss*. In this experimental setting, *virtual loss + SH* achieves comparable performance to the opponent (with a simulation budget of 100) with only 8 batch budgets.

5.7 Related work

Sequential Halving. Among the algorithms for the pure exploration problem in multi-armed bandits [Audibert et al., 2010], Sequential Halving (SH; Karnin et al. [2013]) is one of the most popular algorithms. The theoretical properties of SH have been well studied [Karnin et al., 2013, Zhao et al., 2023]. Due to its simplicity, SH has been widely used for these (but is not limited to) applications: In the context of *tree-search* algorithms, as the root node selection of Monte Carlo tree search can be regarded as a pure exploration problem [Tolpin and Shimony, 2012], Danihelka et al. [2022] incorporated SH into the root node selection and significantly reduced the num-

ber of simulations to improve the performance during AlphaZero/MuZero training. From the min-max search perspective, some studies recursively applied SH to the internal nodes of the search tree [Cazenave, 2014, Pepels et al., 2014]. SH is also used for *hyperparameter optimization*; Jamieson and Talwalkar [2016] formalized the hyperparameter optimization problem in machine learning as a *non-stochastic* multi-armed bandit problem, where the reward signal is not from stochastic stationary distributions but from deterministic function changing over training steps. Li et al. [2018, 2020b] applied SH to hyperparameter optimization in asynchronous parallel settings, which is similar to our batch setting. Their asynchronous approach may have *incorrect promotions* to the next rounds but is more efficient than the synchronous approach. Aziz et al. [2022] applied SH to *recommendation systems*, which identify appealing podcasts for users.

Batched bandit algorithms. Batched bandit algorithms have been studied in various contexts [Perchet et al., 2016, Gao et al., 2019, Esfandiari et al., 2021, Jin et al., 2021a,b, Kalkanli and Ozgur, 2021, Karbasi et al., 2021, Provodin et al., 2022]. Among the batched bandit studies for the pure exploration problem [Agarwal et al., 2017, Grover et al., 2018, Jun et al., 2016], Jun et al. [2016] is the most relevant to our work as they also consider the *fixed-size batch pulls* setting. To the best of our knowledge, the first batched SH variant with a fixed batch size b was introduced by Jun et al. [2016] as a baseline algorithm in their study (Jun+16). It is similar to BSH and it pulls arms so that the number of pulls of the arms is as equal as possible (breadth-first manner). They reported that Jun+16 experimentally performs comparably to or better than their proposed method but did not provide a theoretical guarantee for Jun+16. Our ASH is different from their batch variant in that ASH pulls arms in an advance-first manner instead of a breadth-first manner.

5.8 Limitation

Our batched variants of SH assume that the reward distributions of the arms are from i.i.d. distributions. This property is essential to allow batch pulls. One limitation is that it may be difficult to apply our algorithms to bandit problems where the reward distribution is non-stationary. For example, Jamieson and Talwalkar [2016] applied SH to hyperparameter tuning, where rewards are time-series losses during model training. We cannot apply our batched variants to this problem because we cannot observe "future losses" in a batch.

5.9 Conclusion

In this paper, we proposed ASH as a simple and natural extension of the SH algorithm. We theoretically showed that ASH is algorithmically equivalent to SH as long as the batch budget is not excessively small. This allows ASH to inherit the well-studied theoretical properties of SH, including the simple regret bound. Our experimental results confirmed this claim and demonstrated that ASH and other batched variants of SH, like Jun+16, perform comparably to SH in terms of simple regret. These findings suggest that we can utilize simple batched variants of SH for efficient evaluation of arms with large batch sizes while avoiding performance degradation compared to the sequential execution of SH. By providing a practical solution for efficient arm evaluation, our study opens up new possibilities for applications that require large budgets. Overall, our work highlights the batch robust nature of SH and its potential for large-scale bandit problems.

Chapter 6

Conclusion, Limitations, and Future Directions

6.1 Conclusion

In this dissertation, we explored the potential of RL with massively batched sample generation on accelerators using SIMD programming. Our research is motivated by the R. Sutton's observation — *leveraging computation are ultimately the most effective* in AI research [Sutton, 2019].

- In Chapter 3, we found that massive batching with SIMD programming is effective even in environments with complex branching like (classic) game environments that seem to be incompatible with SIMD programming. Also, we demonstrated this by implementing a game environment suite, Pgx, that allows us to implement fully vectorized RL on GPUs. We showed that not only sample generation but also end-to-end RL algorithms like PPO and AlphaZero can run efficiently on GPUs.

- In Chapter 4, we demonstrated that a simple combination of existing methods implemented using Pgx can achieve the state-of-the-art (SOTA) performance in the bridge bidding AI benchmark. This demonstrates the effectiveness of this

approach and Pgx.

- In Chapter 5, we discussed the potential disadvantage of massive batching. While batching improves computational efficiency, performance may degrade compared to the sequential case due to delayed feedback. However, we showed that the Sequential Halving (SH) algorithm, one of the most popular pure exploration algorithms, does not degrade performance under realistic conditions (i.e., the batch budget should not be extremely small). We also empirically demonstrated that batched SH when combined with the Monte Carlo tree search (MCTS) algorithm is more robust against large batch size in 9x9 Go.

Overall, through this dissertation, we empirically showed that the application range of massively vectorized RL environments on accelerators is wider than thought and demonstrated its effectiveness.

6.2 Limitations

We have demonstrated the effectiveness of massive batching to some extent and that the range of its application was broader than thought. However, we have several limitations that have not been verified in this dissertation.

- Our approach, massively vectorized environment step on accelerators, has some specific classes of problems where such an approach is impossible. Concrete examples include video game environments like StarCraft II [Vinyals et al., 2019], environments with another binary execution, or real-world robotics problems.

- We cannot ignore the cost of human pay for implementing environments. Implementing environments that run on GPUs has some constraints, making it more difficult than normal programming. It is a significant burden if humans need to implement an environment for each problem they want to solve, especially when

the number of problems to be solved is large or the environment implementation is complicated and difficult.

- The theoretical robustness against massive batching of SH shown in Chapter 5 requires the reward distribution to be stationary. This does not hold in more interesting applications, such as applying SH to root node selection in MCTS [Danihelka et al., 2022] or hyperparameter search [Jamieson and Talwalkar, 2016].

6.3 Future directions

Our findings in this dissertation and their limitations suggest several directions for future research. They include but are not limited to the following:

- **Automation of implementation.** As we mentioned in the limitations, the cost of implementation is one of the problems. As the cost of obtaining a normal implementation that runs on CPUs is relatively low, given the recent advances in large language models (LLMs), an approach such as automatic translation to an implementation that runs on GPUs is also a promising direction.

- **Expansion of the applicable range.** In the concurrent and subsequent studies, the effectiveness of vectorized RL on accelerators has been demonstrated in other domains than (classic) games [Bonnet et al., 2024, Nikulin et al., 2024, Rutherford et al., 2024, Matthews et al., 2024, Gulino et al., 2023], but there are still areas where it is unclear whether it is applicable. It is important to investigate problems in such areas and expand the border.

- **Combination with model-based RL.** As we mentioned in the limitations, there are environments where vectorized environment step on accelerators is clearly inapplicable. However, an intermediate approach combined with model-based RL may expand the applicable range. For example, combining a vectorized

simulator that handles relatively low-dimensional spaces with a model that handles high-dimensional spaces like image inputs may expand the applicable range.

- **Algorithms that exploit the property of massively vectorized environments.** We believe that the fact that RL may be executed on massively parallelized environments will also affect the design of algorithms. In particular, planning algorithms like MCTS are known to have sequential nature [Hafner et al., 2021], and there is room for improvement in this area.

Compared to the complexity of previous parallelized RL architectures, parallelization using massively vectorized environments is straightforward and simple. It significantly improves the implementation and experimental experience for RL researchers and practitioners. We have demonstrated the effectiveness of massively vectorized environments and identified algorithms that are well-suited for such environments (such as SH), as well as those that are not. Based on these findings, future RL research should assume that massively vectorized environments are the default and focus on designing algorithms, such as SH, that can fully exploit the advantages of these environments.

Appendix A

Appendix of Chapter 3

A.1 License

The source code of Pgx is available at `https://github.com/sotetsuk/pgx` under the Apache-2.0 license. However, since the original source code of the MinAtar game suite is released under the GPL 3.0 license, a separate extension for Pgx called *pgx-minatar* is provided at `https://github.com/sotetsuk/pgx-minatar`, which adheres to the GPL 3.0 license.

A.2 Example implementation of Go and Chess

We provide examples of Go and Chess implementations in JAX. These implementations are self-contained and single-file without dependencies other than JAX. Note that these implementations have much fewer lines of code than the C++ threading implementation that we compare with. The Go implementation is about 200 lines, and the chess implementation is about 350 lines. In contrast, the OpenSpiel Go and Chess implementations are composed of several thousand lines of code.

Go

```python
from typing import NamedTuple
import jax
from jax import Array, lax
from jax import numpy as jnp

ZOBRIST_BOARD = jax.random.randint(jax.random.PRNGKey(12345), (3, 19 * 19, 2), 0, 2**31 - 1, jnp.uint32)

class State(NamedTuple):
  step_count: Array = jnp.int32(0)
  # ids of representative stone (smallest) in the connected stones
  board: Array = jnp.zeros(19 * 19, dtype=jnp.int32)  # b > 0, w < 0, empty = 0
  board_history: Array = jnp.full((8, 19 * 19), 2, dtype=jnp.int32)  # for obs
  num_captured: Array = jnp.zeros(2, dtype=jnp.int32)  # (b, w)
  consecutive_pass_count: Array = jnp.int32(0)
  ko: Array = jnp.int32(-1)  # by SSK
  is_psk: Array = jnp.bool_(False)
  hash_history: Array = jnp.zeros((19 * 19 * 2, 2), dtype=jnp.uint32)

  @property
  def color(self) -> Array:
    return self.step_count % 2

class Game:
  def __init__(
    self, size: int = 19, komi: float = 7.5, history_length: int = 8, max_termination_steps: int | None = None
  ):
    self.size = size
    self.komi = komi
    self.history_length = history_length
    self.max_termination_steps = size * size * 2 if max_termination_steps is None else max_termination_steps

  def init(self) -> State:
    return State(
      board=jnp.zeros(self.size**2, dtype=jnp.int32),
      board_history=jnp.full((self.history_length, self.size**2), 2, dtype=jnp.int32),
      hash_history=jnp.zeros((self.max_termination_steps, 2), dtype=jnp.uint32),
    )

  def step(self, state: State, action: Array) -> State:
    state = state._replace(ko=jnp.int32(-1))
    # update state
    state = lax.cond(
      (action < self.size * self.size),
      lambda: _apply_action(state, action, self.size),
      lambda: _apply_pass(state),
    )
    # update board history
    board_history = jnp.roll(state.board_history, self.size**2)
    board_history = board_history.at[0].set(jnp.clip(state.board, -1, 1).astype(jnp.int32))
    state = state._replace(board_history=board_history)
    # check PSK
    hash_ = _compute_hash(state)
    state = state._replace(hash_history=state.hash_history.at[state.step_count].set(hash_))
    state = state._replace(is_psk=_is_psk(state))
    # increment turns
    state = state._replace(step_count=state.step_count + 1)
    return state

  def observe(self, state: State, color: Array | None = None) -> Array:
    if color is None:
      color = state.color
    my_sign, _ = _signs(color)

    def _make(i):
      c = jnp.int32([1, -1])[i % 2] * my_sign
      return state.board_history[i // 2] == c

    log = jax.vmap(_make)(jnp.arange(self.history_length * 2))
    color = jnp.full_like(log[0], color)  # b = 0, w = 1
    return jnp.vstack([log, color]).transpose().reshape((self.size, self.size, -1))

  def legal_action_mask(self, state: State) -> Array:
    # some logic is inspired by OpenSpiel's Go implementation
    is_empty = state.board == 0
    my_sign, opp_sign = _signs(state.color)
```

```
      num_pseudo, idx_sum, idx_squared_sum = _count(state, self.size)
      chain_ix = jnp.abs(state.board) - 1
      in_atari = (idx_sum[chain_ix] ** 2) == idx_squared_sum[chain_ix] * num_pseudo[chain_ix]
      has_liberty = (state.board * my_sign > 0) & ~in_atari
      can_kill = (state.board * opp_sign > 0) & in_atari

      def is_adj_ok(xy):
        adj_ixs = _adj_ixs(xy, self.size)
        on_board = adj_ixs != -1
        return (on_board & (is_empty[adj_ixs] | can_kill[adj_ixs] | has_liberty[adj_ixs])).any()

      mask = is_empty & jax.vmap(is_adj_ok)(jnp.arange(self.size**2))
      mask = lax.select(state.ko == -1, mask, mask.at[state.ko].set(False))
      return jnp.append(mask, True)  # pass is always legal

    def is_terminal(self, state: State) -> Array:
      two_consecutive_pass = state.consecutive_pass_count >= 2
      timeover = self.max_termination_steps <= state.step_count
      return two_consecutive_pass | state.is_psk | timeover

    def rewards(self, state: State) -> Array:
      scores = _count_scores(state, self.size)
      is_black_win = scores[0] - self.komi > scores[1]
      rewards = lax.select(is_black_win, jnp.float32([1, -1]), jnp.float32([-1, 1]))
      to_play = state.color
      rewards = lax.select(state.is_psk, jnp.float32([-1, -1]).at[to_play].set(1.0), rewards)
      rewards = lax.select(self.is_terminal(state), rewards, jnp.zeros(2, dtype=jnp.float32))
      return rewards

  def _apply_pass(state: State) -> State:
    return state._replace(consecutive_pass_count=state.consecutive_pass_count + 1)

  def _apply_action(state: State, action, size) -> State:
    state = state._replace(consecutive_pass_count=0)
    my_sign, opp_sign = _signs(state.color)

    # remove killed stones
    adj_ixs = _adj_ixs(action, size)
    adj_ids = state.board[adj_ixs]
    num_pseudo, idx_sum, idx_squared_sum = _count(state, size)
    chain_ix = jnp.abs(adj_ids) - 1
    is_atari = (idx_sum[chain_ix] ** 2) == idx_squared_sum[chain_ix] * num_pseudo[chain_ix]
    single_liberty = (idx_squared_sum[chain_ix] // idx_sum[chain_ix]) - 1
    is_killed = (adj_ixs != -1) & (adj_ids * opp_sign > 0) & is_atari & (single_liberty == action)
    surrounded_stones = (state.board[:, None] == adj_ids) & (is_killed[None, :])
    num_captured = jnp.count_nonzero(surrounded_stones)
    ko_ix = jnp.nonzero(is_killed, size=1)[0][0]
    ko_may_occur = ((adj_ixs == -1) | (state.board[adj_ixs] * opp_sign > 0)).all()
    state = state._replace(
      board=jnp.where(surrounded_stones.any(axis=-1), 0, state.board),
      num_captured=state.num_captured.at[state.color].add(num_captured),
      ko=lax.select(ko_may_occur & (num_captured == 1), adj_ixs[ko_ix], -1),
    )

    # set stone
    state = state._replace(board=state.board.at[action].set((action + 1) * my_sign))

    # merge adjacent chains
    is_my_chain = state.board[adj_ixs] * my_sign > 0
    should_merge = (adj_ixs != -1) & is_my_chain
    new_id = state.board[action]
    tgt_ids = state.board[adj_ixs]
    smallest_id = jnp.min(jnp.where(should_merge, jnp.abs(tgt_ids), 9999))
    smallest_id = jnp.minimum(jnp.abs(new_id), smallest_id) * my_sign
    mask = (state.board == new_id) | (should_merge[None, :] & (state.board[:, None] == tgt_ids[None, :])).any(axis=-1)
    state = state._replace(board=jnp.where(mask, smallest_id, state.board))

    return state

  def _count(state: State, size):
    board = jnp.abs(state.board)
    is_empty = board == 0
    idx_sum = jnp.where(is_empty, jnp.arange(1, size**2 + 1), 0)
    idx_squared_sum = jnp.where(is_empty, jnp.arange(1, size**2 + 1) ** 2, 0)

    def _count_neighbor(xy):
      adj_ixs = _adj_ixs(xy, size)
      on_board = adj_ixs != -1
      return (
```

```
        jnp.where(on_board, is_empty[adj_ixs], 0).sum(),
        jnp.where(on_board, idx_sum[adj_ixs], 0).sum(),
        jnp.where(on_board, idx_squared_sum[adj_ixs], 0).sum(),
      )

    idx = jnp.arange(size**2)
    num_pseudo, idx_sum, idx_squared_sum = jax.vmap(_count_neighbor)(idx)

    def count_all(x):
      return (
        jnp.where(board == x + 1, num_pseudo, 0).sum(),
        jnp.where(board == x + 1, idx_sum, 0).sum(),
        jnp.where(board == x + 1, idx_squared_sum, 0).sum(),
      )

    return jax.vmap(count_all)(idx)

  def _signs(color):
    return jnp.int32([[1, -1], [-1, 1]])[color]  # (my_sign, opp_sign)

  def _adj_ixs(xy, size):
    dx, dy = jnp.int32([-1, +1, 0, 0]), jnp.int32([0, 0, -1, +1])
    xs, ys = xy // size + dx, xy % size + dy
    on_board = (0 <= xs) & (xs < size) & (0 <= ys) & (ys < size)
    return jnp.where(on_board, xs * size + ys, -1)  # -1 if out of board

  def _compute_hash(state: State):
    board = jnp.clip(state.board, -1, 1)
    to_reduce = ZOBRIST_BOARD[board, jnp.arange(board.shape[-1])]
    return lax.reduce(to_reduce, 0, lax.bitwise_xor, (0,))

  def _is_psk(state: State):
    not_passed = state.consecutive_pass_count == 0
    curr_hash = state.hash_history[state.step_count]
    has_same_hash = (curr_hash == state.hash_history).all(axis=-1).sum() > 1
    return not_passed & has_same_hash

  def _count_scores(state: State, size):
    def calc_point(c):
      return _count_ji(state, c, size) + jnp.count_nonzero(state.board * c > 0)

    return jax.vmap(calc_point)(jnp.int32([1, -1]))

  def _count_ji(state: State, color: int, size: int):
    board = jnp.clip(state.board * color, -1, 1)  # my stone: 1, opp stone: -1
    adj_mat = jax.vmap(_adj_ixs, in_axes=(0, None))(jnp.arange(size**2), size)  # (size**2, 4)

    def fill_opp(x):
      b, _ = x
      # true if empty and adjacent to opponent's stone
      mask = (b == 0) & ((adj_mat != -1) & (b[adj_mat] == -1)).any(axis=1)
      return jnp.where(mask, -1, b), mask.any()

    board, _ = lax.while_loop(lambda x: x[1], fill_opp, (board, True))
    return (board == 0).sum()
```

Implementation details. To efficiently detect a single *liberty* in a *string* (see Tromp [1995] for definitions), we introduce the concept of *pseudo liberty*. Let $a_1, \ldots, a_n$ represent the positions of the n *pseudo liberties* in the string. Unlike liberties, *pseudo liberty* allows duplicates. According to the Cauchy–Schwarz inequality: $\left(\sum_{i=1}^{n} a_i\right)^2 = \left(\sum_{i=1}^{n} a_i \cdot 1\right)^2 \leq \left(\sum_{i=1}^{n} a_i^2\right)\left(\sum_{i=1}^{n} 1^2\right) = n \sum_{i=1}^{n} a_i^2$. Equality holds if and only if $a_1 = a_2 = \ldots = a_n$, meaning there is a single distinct *pseudo liberty* a. This position can be determined by $a = \left(\sum_{i=1}^{n} a_i^2\right) / \left(\sum_{i=1}^{n} a_i\right)$. Note that the concept of *pseudo liberty* is not new and has already been applied in OpenSpiel [Lanctot et al., 2019]. We used this concept to efficiently detect a single liberty with accelerators like GPUs.

Chess

```python
from typing import NamedTuple
import jax
import jax.numpy as jnp
import numpy as np
from jax import Array, lax

EMPTY, PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING = tuple(range(7))  # opponent: -1 * piece
MAX_TERMINATION_STEPS = 512  # from AlphaZero paper

# ************* precomputed values *************

# index: a1: 0, a2: 1, ..., h8: 63
INIT_BOARD = jnp.int32([
  4, 1, 0, 0, 0, 0, -1, -4,
  2, 1, 0, 0, 0, 0, -1, -2,
  3, 1, 0, 0, 0, 0, -1, -3,
  5, 1, 0, 0, 0, 0, -1, -5,
  6, 1, 0, 0, 0, 0, -1, -6,
  3, 1, 0, 0, 0, 0, -1, -3,
  2, 1, 0, 0, 0, 0, -1, -2,
  4, 1, 0, 0, 0, 0, -1, -4,
])

# Action: AlphaZero style label (4672 = 64 x 73)
# * [0:9]  underpromotions
#     plane // 3 == 0: rook, 1: bishop, 2: knight
#     plane  % 3 == 0: up  , 1: right,  2: left
# * [9:73] normal moves (queen:56 + knight:8)
FROM_PLANE = -np.ones((64, 73), dtype=np.int32)
TO_PLANE = -np.ones((64, 64), dtype=np.int32)  # ignores underpromotion
zeros, seq, rseq = [0] * 7, list(range(1, 8)), list(range(-7, 0))
# down, up, left, right, down-left, down-right, up-right, up-left, knight, and knight
dr = rseq[::] + seq[::] + zeros[::] + zeros[::] + rseq[::] + seq[::] + seq[::-1] + rseq[::-1]
dc = zeros[::] + zeros[::] + rseq[::] + seq[::] + rseq[::] + seq[::] + rseq[::] + seq[::]
dr += [-1, +1, -2, +2, -1, +1, -2, +2]
dc += [-2, -2, -1, -1, +2, +2, +1, +1]
for from_ in range(64):
  for plane in range(73):
    if plane < 9:  # underpromotion
      to = from_ + [+1, +9, -7][plane % 3] if from_ % 8 == 6 else -1
      if 0 <= to < 64:
        FROM_PLANE[from_, plane] = to
    else:  # normal moves
      r = from_ % 8 + dr[plane - 9]
      c = from_ // 8 + dc[plane - 9]
      if 0 <= r < 8 and 0 <= c < 8:
        to = c * 8 + r
        FROM_PLANE[from_, plane] = to
        TO_PLANE[from_, to] = plane

INIT_LEGAL_ACTION_MASK = np.zeros(64 * 73, dtype=np.bool_)
ixs = [89, 90, 652, 656, 673, 674, 1257, 1258, 1841, 1842, 2425, 2426, 3009, 3010, 3572, 3576, 3593, 3594, 4177, 4178]
INIT_LEGAL_ACTION_MASK[ixs] = True

LEGAL_DEST = -np.ones((7, 64, 27), np.int32)  # LEGAL_DEST[0, :, :] == -1
CAN_MOVE = np.zeros((7, 64, 64), dtype=np.bool_)
for from_ in range(64):
  legal_dest = {p: [] for p in range(7)}
  for to in range(64):
    if from_ == to:
      continue
    r0, c0, r1, c1 = from_ % 8, from_ // 8, to % 8, to // 8
    if (r1 - r0 == 1 and abs(c1 - c0) <= 1) or ((r0, r1) == (1, 3) and abs(c1 - c0) == 0):
      legal_dest[PAWN].append(to)
    if (abs(r1 - r0) == 1 and abs(c1 - c0) == 2) or (abs(r1 - r0) == 2 and abs(c1 - c0) == 1):
      legal_dest[KNIGHT].append(to)
    if abs(r1 - r0) == abs(c1 - c0):
      legal_dest[BISHOP].append(to)
    if abs(r1 - r0) == 0 or abs(c1 - c0) == 0:
      legal_dest[ROOK].append(to)
    if (abs(r1 - r0) == 0 or abs(c1 - c0) == 0) or (abs(r1 - r0) == abs(c1 - c0)):
      legal_dest[QUEEN].append(to)
    if from_ != to and abs(r1 - r0) <= 1 and abs(c1 - c0) <= 1:
      legal_dest[KING].append(to)
  for p in range(1, 7):
```

```
      LEGAL_DEST[p, from_, : len(legal_dest[p])] = legal_dest[p]
      CAN_MOVE[p, from_, legal_dest[p]] = True

LEGAL_DEST_ANY = -np.ones((64, 35), np.int32)
for from_ in range(64):
  legal_dest_any = [x for x in list(LEGAL_DEST[5, from_]) + list(LEGAL_DEST[2, from_]) if x >= 0]
  LEGAL_DEST_ANY[from_, : len(legal_dest_any)] = legal_dest_any

BETWEEN = -np.ones((64, 64, 6), dtype=np.int32)
for from_ in range(64):
  for to in range(64):
    r0, c0, r1, c1 = from_ % 8, from_ // 8, to % 8, to // 8
    if not (abs(r1 - r0) == 0 or abs(c1 - c0) == 0 or abs(r1 - r0) == abs(c1 - c0)):
      continue
    dr, dc = max(min(r1 - r0, 1), -1), max(min(c1 - c0, 1), -1)
    for i in range(6):
      r, c = r0 + dr * (i + 1), c0 + dc * (i + 1)
      if r == r1 and c == c1:
        break
      BETWEEN[from_, to, i] = c * 8 + r

FROM_PLANE, TO_PLANE, INIT_LEGAL_ACTION_MASK, LEGAL_DEST, LEGAL_DEST_ANY, CAN_MOVE, BETWEEN = (
  jnp.array(x) for x in (FROM_PLANE, TO_PLANE, INIT_LEGAL_ACTION_MASK, LEGAL_DEST, LEGAL_DEST_ANY, CAN_MOVE, BETWEEN)
)

keys = jax.random.split(jax.random.PRNGKey(12345), 4)
ZOBRIST_BOARD = jax.random.randint(keys[0], shape=(64, 13, 2), minval=0, maxval=2**31 - 1, dtype=jnp.uint32)
ZOBRIST_SIDE = jax.random.randint(keys[1], shape=(2,), minval=0, maxval=2**31 - 1, dtype=jnp.uint32)
ZOBRIST_CASTLING = jax.random.randint(keys[2], shape=(4, 2), minval=0, maxval=2**31 - 1, dtype=jnp.uint32)
ZOBRIST_EN_PASSANT = jax.random.randint(keys[3], shape=(65, 2), minval=0, maxval=2**31 - 1, dtype=jnp.uint32)
INIT_ZOBRIST_HASH = jnp.uint32([1455170221, 1478960862])

# ***********************************************

class State(NamedTuple):
  color: Array = jnp.int32(0)  # w: 0, b: 1
  board: Array = INIT_BOARD  # (64,)
  castling_rights: Array = jnp.ones([2, 2], dtype=jnp.bool_)  # my queen, my king, opp queen, opp king
  en_passant: Array = jnp.int32(-1)
  halfmove_count: Array = jnp.int32(0)  # number of moves since the last piece capture or pawn move
  fullmove_count: Array = jnp.int32(1)  # increase every black move
  hash_history: Array = jnp.zeros((MAX_TERMINATION_STEPS + 1, 2), dtype=jnp.uint32).at[0].set(INIT_ZOBRIST_HASH)
  board_history: Array = jnp.zeros((8, 64), dtype=jnp.int32).at[0, :].set(INIT_BOARD)
  legal_action_mask: Array = INIT_LEGAL_ACTION_MASK
  step_count: Array = jnp.int32(0)

class Action(NamedTuple):
  from_: Array = jnp.int32(-1)
  to: Array = jnp.int32(-1)
  underpromotion: Array = jnp.int32(-1)  # 0: rook, 1: bishop, 2: knight

  @staticmethod
  def _from_label(label: Array):
    from_, plane = label // 73, label % 73
    underpromotion = lax.select(plane >= 9, -1, plane // 3)
    return Action(from_=from_, to=FROM_PLANE[from_, plane], underpromotion=underpromotion)

  def _to_label(self):
    return self.from_ * 73 + TO_PLANE[self.from_, self.to]

class Game:
  def init(self) -> State:
    return State()

  def step(self, state: State, action: Array) -> State:
    state = _apply_move(state, Action._from_label(action))
    state = _flip(state)
    state = _update_history(state)
    state = state._replace(legal_action_mask=_legal_action_mask(state))
    state = state._replace(step_count=state.step_count + 1)
    return state

  def observe(self, state: State, color: Array | None = None) -> Array:
    if color is None:
      color = state.color
    ones = jnp.ones((1, 8, 8), dtype=jnp.float32)

    def make(i):
      board = jnp.rot90(state.board_history[i].reshape((8, 8)), k=1)
```

```
155
156         def piece_feat(p):
157           return (board == p).astype(jnp.float32)
158
159         my_pieces = jax.vmap(piece_feat)(jnp.arange(1, 7))
160         opp_pieces = jax.vmap(piece_feat)(-jnp.arange(1, 7))
161
162         h = state.hash_history[i, :]
163         rep = (state.hash_history == h).all(axis=1).sum() - 1
164         rep = lax.select((h == 0).all(), 0, rep)
165         rep0 = ones * (rep == 0)
166         rep1 = ones * (rep >= 1)
167         return jnp.vstack([my_pieces, opp_pieces, rep0, rep1])
168
169       return jnp.vstack(
170         [
171           jax.vmap(make)(jnp.arange(8)).reshape(-1, 8, 8),  # board feature
172           color * ones,  # color
173           (state.step_count / MAX_TERMINATION_STEPS) * ones,  # total move count
174           state.castling_rights.flatten()[:, None, None] * ones,  # (my queen, my king, opp queen, opp king)
175           (state.halfmove_count.astype(jnp.float32) / 100.0) * ones,  # no progress count
176         ]
177       ).transpose((1, 2, 0))
178
179     def legal_action_mask(self, state: State) -> Array:
180       return state.legal_action_mask
181
182     def is_terminal(self, state: State) -> Array:
183       terminated = ~state.legal_action_mask.any()
184       terminated |= state.halfmove_count >= 100
185       terminated |= has_insufficient_pieces(state)
186       rep = (state.hash_history == _zobrist_hash(state)).all(axis=1).sum() - 1
187       terminated |= rep >= 2
188       terminated |= MAX_TERMINATION_STEPS <= state.step_count
189       return terminated
190
191     def rewards(self, state: State) -> Array:
192       is_checkmate = (~state.legal_action_mask.any()) & _is_checked(state)
193       return lax.select(
194         is_checkmate,
195         jnp.ones(2, dtype=jnp.float32).at[state.color].set(-1),
196         jnp.zeros(2, dtype=jnp.float32),
197       )
198
199   def _update_history(state: State):
200     board_history = jnp.roll(state.board_history, 64)
201     board_history = board_history.at[0].set(state.board)
202     hash_hist = jnp.roll(state.hash_history, 2)
203     hash_hist = hash_hist.at[0].set(_zobrist_hash(state))
204     return state._replace(board_history=board_history, hash_history=hash_hist)
205
206   def has_insufficient_pieces(state: State):
207     # uses the same condition as OpenSpiel
208     num_pieces = (state.board != EMPTY).sum()
209     num_pawn_rook_queen = ((jnp.abs(state.board) >= ROOK) | (jnp.abs(state.board) == PAWN)).sum() - 2  # two kings
210     num_bishop = (jnp.abs(state.board) == BISHOP).sum()
211     coords = jnp.arange(64).reshape((8, 8))
212     black_coords = jnp.hstack((coords[::2, ::2].ravel(), coords[1::2, 1::2].ravel()))
213     num_bishop_on_black = (jnp.abs(state.board[black_coords]) == BISHOP).sum()
214     is_insufficient = False
215     # king vs king
216     is_insufficient |= num_pieces <= 2
217     # king vs king + (knight or bishop)
218     is_insufficient |= (num_pieces == 3) & (num_pawn_rook_queen == 0)
219     # king + bishop* vs king + bishop* (bishops are on same color tile)
220     is_bishop_all_on_black = num_bishop_on_black == num_bishop
221     is_bishop_all_on_white = num_bishop_on_black == 0
222     is_insufficient |= (num_pieces == num_bishop + 2) & (is_bishop_all_on_black | is_bishop_all_on_white)
223
224     return is_insufficient
225
226   def _apply_move(state: State, a: Action) -> State:
227     piece = state.board[a.from_]
228     # en passant
229     is_en_passant = (state.en_passant >= 0) & (piece == PAWN) & (state.en_passant == a.to)
230     removed_pawn_pos = a.to - 1
231     state = state._replace(
232       board=state.board.at[removed_pawn_pos].set(lax.select(is_en_passant, EMPTY, state.board[removed_pawn_pos]))
233     )
```

```
    is_en_passant = (piece == PAWN) & (jnp.abs(a.to - a.from_) == 2)
    state = state._replace(en_passant=lax.select(is_en_passant, (a.to + a.from_) // 2, -1))
    # update counters
    captured = (state.board[a.to] < 0) | is_en_passant
    state = state._replace(
      halfmove_count=lax.select(captured | (piece == PAWN), 0, state.halfmove_count + 1),
      fullmove_count=state.fullmove_count + jnp.int32(state.color == 1),
    )
    # castling
    board = state.board
    is_queen_side_castling = (piece == KING) & (a.from_ == 32) & (a.to == 16)
    board = lax.select(is_queen_side_castling, board.at[0].set(EMPTY).at[24].set(ROOK), board)
    is_king_side_castling = (piece == KING) & (a.from_ == 32) & (a.to == 48)
    board = lax.select(is_king_side_castling, board.at[56].set(EMPTY).at[40].set(ROOK), board)
    state = state._replace(board=board)
    # update castling rights
    cond = jnp.bool_([[(a.from_ != 32) & (a.from_ != 0), (a.from_ != 32) & (a.from_ != 56)], [a.to != 7, a.to != 63]])
    state = state._replace(castling_rights=state.castling_rights & cond)
    # promotion to queen
    piece = lax.select((piece == PAWN) & (a.from_ % 8 == 6) & (a.underpromotion < 0), QUEEN, piece)
    # underpromotion
    piece = lax.select(a.underpromotion < 0, piece, jnp.int32([ROOK, BISHOP, KNIGHT])[a.underpromotion])
    # actually move
    state = state._replace(board=state.board.at[a.from_].set(EMPTY).at[a.to].set(piece))
    return state

  def _flip_pos(x: Array):  # e.g., 37 <-> 34, -1 <-> -1
    return lax.select(x == -1, x, (x // 8) * 8 + (7 - (x % 8)))

  def _flip(state: State) -> State:
    return state._replace(
      board=-jnp.flip(state.board.reshape(8, 8), axis=1).flatten(),
      color=(state.color + 1) % 2,
      en_passant=_flip_pos(state.en_passant),
      castling_rights=state.castling_rights[::-1],
      board_history=-jnp.flip(state.board_history.reshape(-1, 8, 8), axis=-1).reshape(-1, 64),
    )

  def _legal_action_mask(state: State) -> Array:
    def legal_normal_moves(from_):
      piece = state.board[from_]

      def legal_label(to):
        ok = (from_ >= 0) & (piece > 0) & (to >= 0) & (state.board[to] <= 0)
        between_ixs = BETWEEN[from_, to]
        ok &= CAN_MOVE[piece, from_, to] & ((between_ixs < 0) | (state.board[between_ixs] == EMPTY)).all()
        c0, c1 = from_ // 8, to // 8
        pawn_should = ((c1 == c0) & (state.board[to] == EMPTY)) | ((c1 != c0) & (state.board[to] < 0))
        ok &= (piece != PAWN) | pawn_should
        return lax.select(ok, Action(from_=from_, to=to)._to_label(), -1)

      return jax.vmap(legal_label)(LEGAL_DEST[piece, from_])

    def legal_en_passants():
      to = state.en_passant

      def legal_labels(from_):
        ok = (from_ >= 0) & (from_ < 64) & (to >= 0) & (state.board[from_] == PAWN) & (state.board[to - 1] == -PAWN)
        a = Action(from_=from_, to=to)
        return lax.select(ok, a._to_label(), -1)

      return jax.vmap(legal_labels)(jnp.int32([to - 9, to + 7]))

    def is_not_checked(label):
      a = Action._from_label(label)
      return ~_is_checked(_apply_move(state, a))

    def legal_underpromotions(mask):
      def legal_labels(label):
        a = Action._from_label(label)
        ok = (state.board[a.from_] == PAWN) & (a.to >= 0)
        ok &= mask[Action(from_=a.from_, to=a.to)._to_label()]
        return lax.select(ok, label, -1)

      labels = jnp.int32([from_ * 73 + i for i in range(9) for from_ in [6, 14, 22, 30, 38, 46, 54, 62]])
      return jax.vmap(legal_labels)(labels)

    # normal move and en passant
    possible_piece_positions = jnp.nonzero(state.board > 0, size=16, fill_value=-1)[0]
```

```
    a1 = jax.vmap(legal_normal_moves)(possible_piece_positions).flatten()
    a2 = legal_en_passants()
    actions = jnp.hstack((a1, a2))  # include -1
    actions = jnp.where(jax.vmap(is_not_checked)(actions), actions, -1)
    mask = jnp.zeros(64 * 73 + 1, dtype=jnp.bool_)  # +1 for sentinel
    mask = mask.at[actions].set(True)

    # castling
    b = state.board
    can_castle_queen_side = state.castling_rights[0, 0]
    can_castle_queen_side &= (b[0] == ROOK) & (b[8] == EMPTY) & (b[16] == EMPTY) & (b[24] == EMPTY) & (b[32] == KING)
    can_castle_king_side = state.castling_rights[0, 1]
    can_castle_king_side &= (b[32] == KING) & (b[40] == EMPTY) & (b[48] == EMPTY) & (b[56] == ROOK)
    not_checked = ~jax.vmap(_is_attacked, in_axes=(None, 0))(state, jnp.int32([16, 24, 32, 40, 48]))
    mask = mask.at[2364].set(mask[2364] | (can_castle_queen_side & not_checked[:3].all()))
    mask = mask.at[2367].set(mask[2367] | (can_castle_king_side & not_checked[2:].all()))

    # set underpromotions
    actions = legal_underpromotions(mask)
    mask = mask.at[actions].set(True)

    return mask[:-1]

  def _is_attacked(state: State, pos: Array):
    def can_move(to):
      ok = (to >= 0) & (state.board[to] < 0)  # should be opponent's
      piece = jnp.abs(state.board[to])
      between_ixs = BETWEEN[pos, to]
      ok &= CAN_MOVE[piece, pos, to] & ((between_ixs < 0) | (state.board[between_ixs] == EMPTY)).all()
      ok &= ~((piece == PAWN) & (to // 8 == pos // 8))  # should move diagonally to capture
      return ok

    return jax.vmap(can_move)(LEGAL_DEST_ANY[pos, :]).any()

  def _is_checked(state: State):
    king_pos = jnp.argmin(jnp.abs(state.board - KING))
    return _is_attacked(state, king_pos)

  def _zobrist_hash(state: State) -> Array:
    hash_ = lax.select(state.color == 0, ZOBRIST_SIDE, jnp.zeros_like(ZOBRIST_SIDE))
    to_reduce = ZOBRIST_BOARD[jnp.arange(64), state.board + 6]  # 0, ..., 12 (w:pawn, ..., b:king)
    hash_ ^= lax.reduce(to_reduce, 0, lax.bitwise_xor, (0,))
    to_reduce = jnp.where(state.castling_rights.reshape(-1, 1), ZOBRIST_CASTLING, 0)
    hash_ ^= lax.reduce(to_reduce, 0, lax.bitwise_xor, (0,))
    hash_ ^= ZOBRIST_EN_PASSANT[state.en_passant]
    return hash_
```

A.3 Comparison to Brax and PettingZoo APIs

The Pgx API draws inspiration from the Brax and PettingZoo APIs, and we illustrate this with actual code examples. Figure A.1 and Figure A.2 present the code examples for the Pgx and Brax APIs, respectively. While both APIs share similarities, the Pgx API handles multi-agent environments and the *current player* vector specifies which agents are to act. Similar to Pgx, the PettingZoo API (Figure A.3) designates the next agent to act using the *agent iterator* concept. However, since Pgx is a library centered on environment vectorization, we prefer a vectorized *current player* to a flexible but dynamic iterator.

To prove the practical generality of the Pgx API, we provide a demonstration illustrating the conversion from Pgx API to PettingZoo API, available at `https://github.com/sotetsuk/pgx/blob/main/colab/pgx2pettingzoo.ipynb`.

```python
import jax
import pgx

env = pgx.make("go_19x19")
init_fn = jax.jit(jax.vmap(env.init))
step_fn = jax.jit(jax.vmap(env.step))

batch_size = 1024
rng_keys = jax.random.split(jax.random.PRNGKey(9999), batch_size)

state = init_fn(rng_keys)
while not state.terminated.all():
  action = model(state.current_player, state.observation, state.legal_action_mask)
  state = step_fn(state, action)
```

Figure A.1: Example usage of Pgx API.

```python
import jax
from brax import envs

env = envs.get_environment("ant")
reset_fn = jax.jit(jax.vmap(env.reset))
step_fn = jax.jit(jax.vmap(env.step))

batch_size = 1024
rng_keys = jax.random.split(jax.random.PRNGKey(9999), batch_size)

state = reset_fn(rng_keys)
while not state.done.all():
  action = model(state)
  state = step_fn(state, action)
```

Figure A.2: Example usage of Brax API.

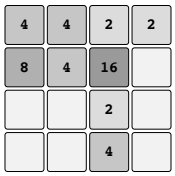
```python
from pettingzoo.classic import go_v5

env = go_v5.env()
env.reset()
for agent in env.agent_iter():
  observation, reward, terminated, truncated, info = env.last()
  action = model(agent, observation["observation"], observation["action_mask"])
  env.step(action)
env.close()
```

Figure A.3: Example usage of PettingZoo API.

A.4 Game explanations

This section describes the short summary of each game implemented in Pgx (as of v1.4.0), except the MinAtar suite. See Young and Tian [2019] for the description of the MinAtar suite. For the full description of each game, please refer to the Pgx documentation from `https://github.com/sotetsuk/pgx`.

2048



The game of 2048 [Cirulli, 2014] is a single-player perfect information game with chance events on a 4x4 board. The rules are simple, but to play well, agents need planning ability in stochastic dynamics.

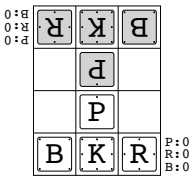
Rules. The objective is to create larger-numbered tiles by merging tiles. The player can take four actions: left, up, right, or down. All tiles move in the chosen direction, and when tiles with the same number collide, they merge, forming a new tile with twice the value (e.g., $4 + 4 = 8$). When a new tile is created, the new tile number is rewarded. After the tiles have moved, a new tile whose value is either two or four is randomly placed in empty positions. The probability of a two appearing is 0.9, and the probability of a four appearing is 0.1. The game ends when no legal move exists.

Observation. Shape is $4 \times 4 \times 31$. The observation design follows Antonoglou et al. [2022]. Each plane represents a 4×4 board, and each tile number is encoded in a 31-bit binary representation.

Action. There are 4 actions: left (0), up (1), right (2), or down (3).

Reward. The sum of merged tiles.

Animal shogi



Animal shogi is a miniature version of the traditional Japanese board game, shogi, played on a small 4x3 board. It is a two-player perfect information game. Originally developed for children, it possesses sufficient complexity for human play, making it more than just a toy environment.

Rules. Two players take turns moving their pieces, aiming to achieve *checkmate* – a situation where the king is attacked and cannot make a legal move to escape the threat. The pieces used in Animal shogi are *Lion* (**K**ing), *Giraffe* (**R**ook), *Elephant* (**B**ishop), and *Chick* (**P**awn). The available directions for movement are indicated by circular dots for each piece. Note that each piece can move only one square at a time, even if the piece is a Giraffe (Rook). The Chick (Pawn) can be *promoted* to a *Hen* (**G**old) by entering the opponent's territory (the farthest rank). Similar to ordinary shogi, captured opponent pieces can be reused by dropping them on the board. A player can win by checkmating the opponent's Lion (King) or by having their Lion (King) enter the opponent's territory (*Try rule*). If the same position occurs three times, it results in a repetition draw.

Observation. Shape is $4 \times 3 \times 194$. The observation design follows AlphaZero [Silver et al., 2018]. Each plane represents a 4×3 board and encodes the position-dependent features in Table A.1. P1 represents the current player, and P2 represents the opponent player. As in AlphaZero, the last 8-step history is encoded ($24 \times 8 = 192$ planes in

Table A.1: Animal shogi position-dependent features.

Feature	# Planes
P1 piece	5
P2 piece	5
P1 prisoner piece count	6
P2 prisoner piece count	6
Repetitions	2
Total	24

total). Also, Table A.2 shows the position-independent features. Each plane has the same values for all positions.

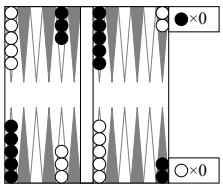
Table A.2: Animal shogi position-independent features.

Feature	# Planes
Color	1
Elapsed timesteps (normalized to $[0, 1]$)	1

Action. Action consists of (1) the source position of the piece to move and the direction to move ($12 \times 8 = 96$), and (2) the position of the piece to drop and the type of piece to drop ($12 \times 3 = 36$), for a total of 132 discrete actions.

Reward. Each player gets $+1$ (win), -1 (lose), or 0 (draw).

Backgammon



Backgammon is a two-player game with perfect information that also incorporates chance events. It serves as an important benchmark for RL in stochastic environ-

ments [Tesauro, 1995, Antonoglou et al., 2022]. To excel, agents require a high planning capability within stochastic environments.

Rules. Players, represented by white and black colors, aim to move their set of 15 *checkers* across a board consisting of 24 *points*. The objective is to be the first to *bear off* all their checkers, moving in the opposite direction. Each turn involves rolling two dice, determining the number of points a player can move their checkers. If both dice show the same number, the player can make four moves of that number. The game has several constraints on legal actions:

- Checkers cannot bear off until all of them have reached the *home board* (an area one-quarter of the distance from the goal).
- A point with two or more opponent's checkers stacked on it is *blocked*, and the player cannot move their checkers onto that point.
- By moving a checker to a point where the opponent has only one checker, the player can *hit* the opponent's checker and send it to the central bar. A player with checkers on the bar must first move those checkers before moving any other checkers.

The game ends when one of the players has borne off all their checkers. Victory rewards differ:

- A *gammon win* (2 points) is when the opponent has not borne off any checkers.
- A *backgammon win* (3 points) is a gammon win where the opponent has checkers left on the bar or within the winner's home board.
- All other victories are termed *single wins* (1 point).

Observation. Shape is 34. Table A.3 shows the backgammon features. The first 28 features are the same as in Antonoglou et al. [2022]. The last 6 features encode the number of playable die number.

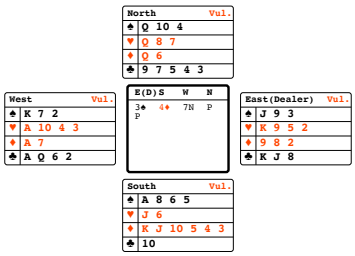
Table A.3: Backgammon features.

Feature	Size
Checkers on points	24
Checkers on bar	2
Checkers borne off	2
Number of available moves for each die number	6
Total	34

Action. There are $26 \times 6 = 156$ discrete actions. Action design in Pgx follows Antonoglou et al. [2022]. Each action consists of 26 *source* positions and *die* number. The first source position is a *no-op* when there is no movable checker, the second source is the bar, and the remaining 24 sources represent each point on board.

Reward. Each player gets the game payoff as a reward.

Bridge bidding



Contract bridge is an imperfect information game played by four people in teams of two. Cooperation within the team is required to win.

Game flow. In the contract bridge, after each player is dealt 13 cards, there are two phases: (1) *bidding* and (2) *playing*.

- **Bidding**: Each player bids in an auction format to determine the *contract*, the target number of *tricks* their team will try to achieve in the next playing phase.
- **Playing**: Each player plays a card, and the player who plays the "strongest" card wins the trick. This process is repeated 13 times, aiming to maximize the

number of tricks won by the team and achieve the target number of tricks for their team or prevent the opponent team from achieving their target number of tricks.

Bidding is considered more challenging than playing and is believed to have a significant impact on the outcome of the game. Therefore, previous studies [Rong et al., 2019, Tian et al., 2020, Lockhart et al., 2020] have often focused only on the bidding part by replacing the playing part results using a double dummy solver[1]. Pgx also follows this setting.

Rules of bidding. The four players take turns to act. The available actions for each player are as follows: (1) *Bid* "higher" than the previous bid, (2) *Pass*, (3) *Double* the opponent's last bid, or (4) *Redouble* in response to the opponent team's double. There are 35 possible bid combinations, including the target trick number (1-7) and the suit (club, diamond, heart, spade, no trump). If three consecutive passes occur after someone's last bid, the bidding phase ends, and the bidding team, target number of tricks, and *trump* suit are determined (if no one bids and there are four consecutive passes, the game ends in a draw). The rewards are based on the achievement of the target, the magnitude of the target trick number, and whether a double or redouble is present.

Observation. Shape is 480. Table A.4 shows the bridge bidding features. Observation design follows Lockhart et al. [2020] and Lanctot et al. [2019].

Action. There are 38 discrete actions: pass, double, redouble, and 35 bids.

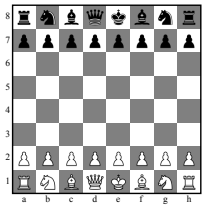
Reward. Each player gets the game payoff as a reward.

[1] `https://github.com/dds-bridge/dds`

Table A.4: Bridge bidding features.

Feature	Size
Vulnerability	4
Pass before the opening bid	4
Bidding history ($35 \times 4 \times 3$)	420
Current player's hand	52
Total	480

Chess



Chess is a two-player perfect information game on an 8x8 board.

Rules. Two players take turns moving their pieces, aiming to achieve *checkmate* – a situation where the king is attacked and cannot make a legal move to escape the threat. Each piece has its own specific movement rules. While capturing the opponent's pieces is allowed, unlike in shogi, the captured pieces cannot be reused. In addition to checkmate, there are other terminal conditions for a draw, such as a threefold repetition of the same position or a *stalemate* (when the king is not in check but has no legal moves). There are also special moves called *pawn promotion*, *en passant*, and *castling*.

Observation. Shape is $8 \times 8 \times 119$. The observation design follows AlphaZero [Silver et al., 2018]. Each plane represents an 8×8 board and encodes the position-dependent features (Table A.5). As in AlphaZero, the last 8-step history is encoded ($14 \times 8 = 112$ planes in total). Also, Table A.6 shows the 7 position-independent feature planes.

Action. There are $64 \times 73 = 4672$ discrete actions. Action design also follows AlphaZero [Silver et al., 2018]. Each action consists of 64 *source* positions and 73 moves,

Table A.5: Chess position-dependent features.

Feature	# Planes
P1 piece	6
P2 piece	6
Repetitions	2
Total	14

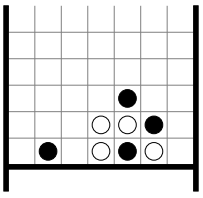
Table A.6: Chess position-independent features.

Feature	# Planes
Color	1
Total move count	1
P1 castling	2
P2 castling	2
No progress count	1
Total	7

where 56 moves are queen moves, 8 moves are knight moves, and 9 moves are underpromotions.

Reward. Each player gets +1 (win), −1 (lose), or 0 (draw).

Connect Four



Connect Four is a two-player perfect information game played on a 7x6 board.

Rules. Players take turns dropping discs into any of the seven columns. The objective is to create a line of four of their own discs either vertically, horizontally, or diagonally.

The player who achieves this first is declared the winner. If all the spaces on the board are filled, and neither player has managed to create a line of four discs, the game ends in a draw.

Observation. Shape is $6 \times 7 \times 2$. Observation consists of two 6×7 feature planes (Table A.7).

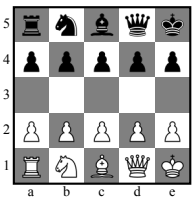
Table A.7: Connect Four features.

Feature	# Planes
P1 discs	1
P2 discs	1

Action. There are 7 discrete actions. Each action represents the column index into which the player drops the token.

Reward. Each player gets $+1$ (win), -1 (lose), or 0 (draw).

Gardner chess



Gardner chess is a two-player perfect information game played on a 5x5 board. It uses the pieces corresponding to the leftmost 5 columns of standard chess.

Rules. The rules are the same as in regular chess, except that *double pawn moves*, *en passant*, and *castling* are not allowed.

Table A.8: Gardner chess position-dependent features.

Feature	# Planes
P1 piece	6
P2 piece	6
Repetitions	2
Total	14

Observation. Shape is $5 \times 5 \times 115$. The observation design follows AlphaZero [Silver et al., 2018]. Each plane represents a 5×5 board and encodes position-dependent features (Table A.8). As in AlphaZero, the last 8-step history is encoded ($14 \times 8 = 112$ planes in total). Also, Table A.9 shows position-independent feature planes.

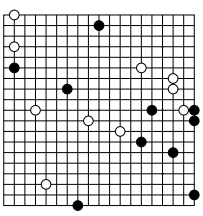
Table A.9: Gardner chess position-independent features.

Feature	# Planes
Color	1
Total move count	1
No progress count	1
Total	3

Action. There are $25 \times 49 = 1225$ discrete actions. Action design also follows AlphaZero [Silver et al., 2018]. Each action consists of 25 *source* positions and 49 moves, where 32 moves are queen moves, 8 moves are knight moves, and 9 moves are underpromotions.

Reward. Each player gets $+1$ (win), -1 (lose), or 0 (draw).

Go



Go is a two-player perfect information game played on a 19x19 board. The essential strategic aspects of Go can be preserved even on smaller boards like 9x9. There are variations of the Go rules, such as Chinese rules and Japanese rules. In computer Go, the Tromp-Taylor rules [Tromp, 1995] are commonly used, and Pgx also follows them. To address the inherent advantage of the first player (black), it is common to add a scoring adjustment called *komi* (e.g., 6.5) to the final score of the second player (white). This improves fairness and helps to avoid a draw. Pgx uses a *komi* of 6.5 by default.

Observation. Shape is $9 \times 9 \times 17$ (or $19 \times 19 \times 17$). The observation design follows the AlphaGo Zero observation design [Silver et al., 2017]. Each plane represents a 9×9 (or 19×19) board and encodes the position-dependent features (Table A.10). As

Table A.10: Go features.

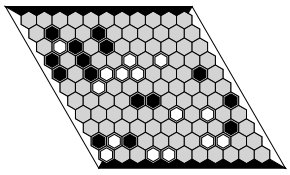
# Planes	Description
P1 stones	1
P2 stones	1

is the case with AlphaGo Zero, the last 8-step history is encoded ($2 \times 8 = 16$ planes in total). An additional plane encodes the color of the current player. This is necessary for the agent to know the *komi* information.

Action. There are 82 (9x9) or 362 (19x19) discrete actions. Each action represents the position on the board to place a stone. The last action represents pass.

Reward. Each player gets $+1$ (win), -1 (lose), or 0 (draw).

Hex



Hex is a two-player perfect information game played on an 11x11 board.

Rules. Players take turns placing stones on a board, aiming to connect one side of the board to the opposite side with their stones. A draw does not occur because both players cannot simultaneously connect their sides of the board. To balance the first-player advantage, the *swap rule* is implemented. This rule allows the second player, instead of placing a stone, to replace the color of the first player's stone with their own color at the mirrored position.

Observation. Shape is $11 \times 11 \times 4$. Each plane represents an 11×11 board and encodes position-dependent features (Table A.11). The last two planes encode the

Table A.11: Hex position-dependent features.

Feature	# Planes
P1 stones	1
P2 stones	1

color of the current player and whether the swap is a legal action (Table A.12). Color information is necessary for the agent to know the side to connect.

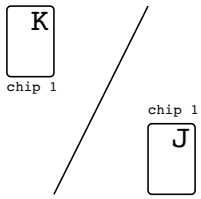
Table A.12: Hex position-independent features.

Feature	# Planes
Color	1
Swap	1

Action. There are 122 discrete actions. The first 121 actions represent placing a stone on each cell of the board. The final action 121 is the swap action available only at the second turn.

Reward. Each player gets $+1$ (win) or -1 (lose).

Kuhn poker



Kuhn poker is a two-player imperfect information game designed for research purposes [Kuhn, 1950].

Rules. The deck has three cards: Jack, Queen, and King. Each player is dealt one card, and the remaining card is unused. Players have two actions available: *bet* and *pass*. The following scenarios can occur, with player A being the first to play and player B being the second:

- bet (A) - bet (B): *Showdown*, and the winner takes +2.
- bet (A) - pass (B): A takes +1.
- pass (A) - pass (B): Showdown, and the winner takes +1.
- pass (A) - bet (B) - bet (A): Showdown, and the winner takes +2.
- pass (A) - bet (B) - pass (A): B takes +1.

As Kuhn poker is a zero-sum game, the loser of the game receives the negative of the winner's payoff.

Observation. Shape is 7. The observation consists of a binary vector of size 7 (Table A.13).

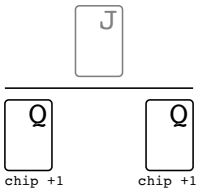
Action. There are 2 discrete actions: bet (0) and pass (1).

Reward. Each player gets +2, +1, −1, or −2, depending on the game payoff.

Table A.13: Kuhn poker features.

Feature	Size
P1 hand	3
P1 chip	2
P2 chip	2
Total	7

Leduc hold'em



Leduc hold'em is a two-player imperfect information game designed for research purposes [Southey et al., 2005].

Rules. The deck consists of six cards: two Jacks, two Queens, and two Kings. Each player starts with a bet of 1 chip. The game consists of two rounds. In the first round, each player is dealt one private card, and in the second round, one public card is revealed. In each round, players have the option to *call*, *raise*, or *fold*. If either player folds, the hand ends, and the opponent takes the *pot*. If both players call, the game proceeds to the next round. In the second round, it advances to *showdown*, where the winner is determined by the strength of their cards. Each player can raise 2 chips in the first round and 4 chips in the second round. In each round, each player is allowed to raise only once (a total of 2 raises per round). Therefore, the maximum number of chips that can be bet for each player is $1 + 2 \times 2 + 4 \times 2 = 13$.

Observation. The observation consists of a binary vector of size 34 (Table A.14).

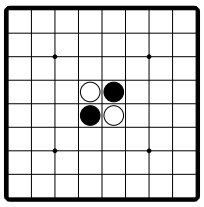
Action. There are 3 discrete actions: call (0), raise (1), or fold (2).

Reward. Each player gets the game payoff as a reward.

Table A.14: Leduc hold'em features.

Feature	Size
P1 hand	3
Public cards	3
P1 chip	14
P2 chip	14
Total	34

Othello



Othello is a two-player perfect information game played on an 8x8 board.

Rules. Players take turns placing discs. The player with more discs at the end wins. Players can place a disc in an empty position where it can sandwich the opponent's discs between their own discs, and the sandwiched discs are flipped to their own color. If a player has no valid move to make, they must pass. The game ends when neither player can make a legal move.

Observation. Shape is $8 \times 8 \times 2$. Each plane represents an 8×8 board and encodes the features in Table A.15.

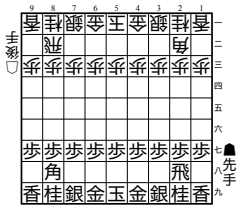
Table A.15: Othello features.

Feature	# Planes
P1 discs	1
P2 discs	1

Action. There are 65 discrete actions. The first 64 actions represent placing a disc on each square of the board. The last action represents pass.

Reward. Each player gets +1 (win), −1 (lose), or 0 (draw).

Shogi



Shogi is a two-player perfect information game on a 9x9 board.

Rules. Two players take turns moving their pieces, aiming to achieve *checkmate* – a situation where the king is attacked and cannot make a legal move to escape the threat. Each piece has its own specific movement rules. Captured pieces can be reused by dropping them to the board (instead of moving a piece) on the player's turn. The game ends when a player achieves checkmate or when the game reaches a draw by four-fold repetition. Unlike chess, shogi does not have a draw by *stalemate* (when the king is not in check but has no legal moves). Not only pawns but also other pieces (except Gold and King) can *promote* by entering the opponent's territory (1-3 rows).

Observation. Shape is $9 \times 9 \times 119$. The observation design follows the *dlshogi* observation design [Yamaoka, 2017]. Each plane represents a 9×9 board and encodes the position-dependent features in Table A.16. Also, the observation has the position-independent feature planes (Table A.17).

Action. There are $81 \times 27 = 2187$ distinct actions. The action design also follows the *dlshogi* action design [Yamaoka, 2017]. Each action consists of 81 *destination* to which the piece moves and 27 *directions* from which the piece moves. The direction is one of 10 moves (8 King moves and 2 Knight moves), 10 moves with promotion, or 7 drops.

Table A.16: Shogi position-dependent features.

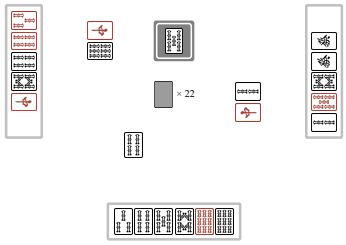
# Planes	Description
P1 piece	14
Attacked by P1 piece	14
Attacked by N or more P1 pieces ($N = 1, 2, 3$)	3
P2 piece	14
Attacked by P2 piece	14
Attacked by N or more P2 pieces ($N = 1, 2, 3$)	3
Total	62

Table A.17: Shogi position-independent features.

# Planes	Description
P1 hand	28
P2 hand	28
P1 king is checked	1
Total	57

Reward. Each player gets +1 (win), −1 (lose), or 0 (draw).

Sparrow mahjong



Sparrow mahjong is an imperfect information game played with 44 tiles. It is a simplified version of Japanese mahjong. Sparrow mahjong is designed for human players who are not familiar with the rules of full-size Japanese mahjong but requires the same essential skills as Japanese mahjong. It can be played with 2 to 6 players, but Pgx uses a 3-player version.

Rules. Sparrow mahjong has 11 types of tiles, and each tile has 4 copies, for a total of 44 tiles. The 11 types of tiles are *bamboo* 1-9 (1s-9s) and *red dragon* (rd), *green dragon* (gd). Each player starts with 5 tiles in their *hand*, and the game proceeds by each player drawing a tile from the deck and discarding one tile from their hand to a public *river*. The objective is to "accomplish" a hand of 6 tiles faster than other players and with a higher score. There are two ways to win the game: *ron* (winning by using a tile discarded by another player) and *tsumo* (winning by drawing a tile from the deck). A hand is "accomplished" when the hand consists of either (1) one *chow* (a sequence of three tiles) and one *pung* (a set of three identical tiles), (2) two *chows*, or (3) two *pungs*. For example, "2s 3s 4s 6s 6s 6s" (one *chow* and one *pung*), "1s 2s 3s 7s 8s 9s" (two *chows*), and "1s 1s 1s rd rd rd" (two *pungs*) are accomplished hands. The accomplished hand is scored according to its difficulty. There is a minimum score required to win, and it is key to infer the opponent's hand. There is a unique rule in Japanese mahjong called *furiten*: a player cannot win by *ron* with a tile discarded previously by themselves. The game ends when a player wins or when the deck is empty.

Observation. Shape is 11×15. The observation consists of 15 feature planes, where each plane represents 11 tile types (Table A.18). P1 represents the current player, while P2 and P3 represent the opponents.

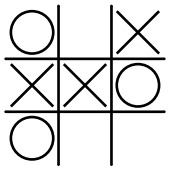
Table A.18: Sparrow mahjong features.

Feature	# Planes
P1 hand	4
Red dora in P1 hand	1
Dora	1
All discarded tiles by P1	1
All discarded tiles by P2	1
All discarded tiles by P3	1
Discarded tiles in the last 3 steps by P2	3
Discarded tiles in the last 3 steps by P3	3
Total	15

Action. There are 11 discrete actions, each representing a tile to discard.

Reward. Each player gets the game payoff normalized to $[-1, 1]$.

Tic-tac-toe



Tic-tac-toe is a two-player perfect information game played on a 3x3 board.

Rules. Players take turns, with one marking X and the other marking O. The objective is for a player to place their mark in a vertical, horizontal, or diagonal line. The player who achieves this first is the winner. If all nine squares are filled and neither player has made a line, the game ends with a draw.

Observation. Shape is $3 \times 3 \times 2$. Each plane represents a 3×3 board and encodes the features in Table A.19.

Table A.19: Tic-tac-toe features.

Feature	# Planes
Marked by P1	1
Marked by P2	1

Action. There are 9 discrete actions, each representing a square index to place a mark.

Reward. Each player gets $+1$ (win), -1 (lose), or 0 (draw).

Appendix B

Appendix of Chapter 5

B.1 Python implementation of SH and ASH

For the sake of reproducibility and a better understanding, we provide Python code for the Sequential Halving (SH) algorithm using advance-first target pulls and the Advance-first Sequential Halving (ASH) algorithm in Figure B.1.

```python
from math import log2, ceil, floor
import numpy as np

def sh(bandit: BanditProblem, n: int, T: int) -> int:
  L = _get_target_pulls(n, T)                  # L: target pulls
  N = np.append(np.zeros(n, dtype=int), -1e9)  # N: pull counts
  R = np.append(np.zeros(n, dtype=float), 0.)  # R: avg rewards
  for i in range(T):
    a = np.argmax(np.where(N == L[i], R, -np.inf))
    r = bandit.pull(a)
    R[a] = (R[a] * N[a] + r) / (N[a] + 1)
    N[a] += 1
  return int(np.argmax(np.where(N >= max(N), R, -np.inf)))

def ash(bandit: BanditProblem, n: int, B: int, b: int = 1) -> int:
  L = _get_target_pulls(n, b * B)              # L: target pulls
  N = np.append(np.zeros(n, dtype=int), -1e9)  # N: pull counts
  R = np.append(np.zeros(n, dtype=float), 0.)  # R: avg rewards
  for i in range(B):
    batch = []
    M = np.zeros_like(N)                       # M: virtual pull counts
    for j in range(b):
      t = i * b + j
      N_max = np.max(np.where(N + M == L[t], N, -np.inf))
      a = np.argmax(np.where((N + M == L[t]) & (N == N_max), R, -np.inf))
      batch.append(a)
      M[a] += 1
    rewards = bandit.batch_pull(batch)
    for a, r in zip(batch, rewards):
      R[a] = (R[a] * N[a] + r) / (N[a] + 1)
      N[a] += 1
  return int(np.argmax(np.where(N >= max(N), R, -np.inf)))

def _get_target_pulls(n: int, T: int) -> list[int]:
  target_pulls = []
  num_rounds = ceil(log2(n))
  num_active_arms = n
  cum_pulls = 0
  for r in range(num_rounds):
    J = floor(T / (num_active_arms * num_rounds))
    if r == num_rounds - 1:
      remaining_pulls = T - len(target_pulls)
      J = remaining_pulls // 2
    for _ in range(num_active_arms):
      for i in range(J):
        target_pulls.append(cum_pulls + i)
    cum_pulls += J
    num_active_arms = ceil(num_active_arms / 2)  # halving
  return target_pulls + [int(-1e9)] * (T - len(target_pulls))
```

Figure B.1: Python implementation of SH (Algorithm 3) and ASH (Algorithm 7).

B.2 Proof of Lemma 1

Lemma 1 *For any integer $b \geq 2$, the inequality*

$$\left\lceil \frac{x}{2} \right\rceil - 1 \geq \left\lceil \frac{b-1}{\lfloor 4b/x \rfloor} \right\rceil \tag{B.1}$$

holds for all integers $x \in [3, 4b]$.

Proof. This proof demonstrates that for any integer $b \geq 2$ and $x \in [3, 4b]$, the inequality (B.1) is satisfied. Given $z \geq c \implies z \geq \lceil c \rceil$ for any integer z and real number c, it suffices to demonstrate that

$$\left\lceil \frac{x}{2} \right\rceil - 1 \geq \frac{b-1}{\lfloor 4b/x \rfloor} \iff \left\lceil \frac{x}{2} \right\rceil - 1 - \frac{b-1}{\lfloor 4b/x \rfloor} \geq 0.$$

Given that $\left\lfloor \frac{4b}{x} \right\rfloor > 0$, it follows that

$$\left(\left\lceil \frac{x}{2} \right\rceil - 1 \right) \left\lfloor \frac{4b}{x} \right\rfloor - (b-1) \geq 0, \tag{B.2}$$

for any integer $b \geq 2$ and $x \in [3, 4b]$. Two cases are considered:

Case 1: x is even. Suppose $x = 2y$, with $y \in [2, 2b]$. We aim to show that

$$(y-1) \left\lfloor \frac{2b}{y} \right\rfloor - (b-1) \geq 0. \tag{B.3}$$

Two sub-cases are considered:

1. For $y \in [b+1, 2b]$, as $\left\lfloor \frac{2b}{y} \right\rfloor = 1$, LHS $= (y-1) - (b-1) \geq 0$.

2. For $y \in [2, b]$, as $\lfloor c \rfloor > c-1$ for any real number c, we have LHS $> (y-1)\left(\frac{2b}{y} - 1\right) - (b-1) = -\frac{(y-2)(y-b)}{y}$. As $y > 0$ and $-(y-2)(y-b) \geq 0$ in $y \in [2, b]$, we have LHS ≥ 0.

Consequently, it has been established that for even values of x, the inequality (B.3) is upheld.

Case 2: x is odd. Suppose $x = 2y + 1$, with $y \in [1, 2b - 1]$. We aim to show that

$$y \left\lfloor \frac{4b}{2y+1} \right\rfloor - (b-1) \geq 0. \tag{B.4}$$

Two sub-cases are considered:

1. For $y \in [b, 2b-1]$, as $\left\lfloor \frac{4b}{2y+1} \right\rfloor = 1$, LHS $= y - (b-1) \geq 0$.

2. For $y \in [1, b-1]$, as $\lfloor c \rfloor > c - 1$ for any real number c, we have LHS $> y\left(\frac{4b}{2y+1} - 1\right) - (b-1) = \frac{2by-b-2y^2+y+1}{2y+1} = \frac{-2y(y-(b+\frac{1}{2}))-(b-1)}{2y+1} \geq 0$. As $2y + 1 > 0$ and $-2y(y - (b + \frac{1}{2})) - (b-1) \geq 0$ in $y \in [1, b-1]$, we have LHS ≥ 0.

Similarly, it has been demonstrated that for odd values of x, the inequality (B.4) is upheld.

Therefore, through the analysis of these two cases, it is proven that for any integer $b \geq 2$ and $x \in [3, 4b]$, the inequality (B.2) is satisfied, thereby confirming the validity of (B.1). □

Bibliography

In this bibliography, the names of journals and conference proceedings will be abbreviated as follows:

Full Name	Abbreviation
AAAI Conference on Artificial Intelligence	AAAI
ACM Conference on Recommender Systems	RecSys
Advances in Computer Games	ACG
Advances in Neural Information Processing Systems	NIPS/NeurIPS
Annals of Statistics	Ann. Statist.
Communications of the ACM	Commun. ACM
Conference on Learning Theory	COLT
Conference on Uncertainty in Artificial Intelligence	UAI
European Conference on Computer Vision	ECCV
IEEE/CAA Journal of Automatica Sinica	IEEE/CAA JAS
IEEE Conference on Games	IEEE CoG
IEEE International Conference on Data Mining	ICDM
IEEE International Conference on Tools for Artificial Intelligence	IEEE ICTAI
IEEE Transactions on Computational Intelligence and AI in Games	IEEE T-CIAIG
IEEE Transactions on Games	IEEE ToG
International Conference on Artificial Intelligence and Statistics	AISTATS
International Conference on Autonomous Agents and MultiAgent Systems	AAMAS
International Conference on Computers and Games	CG
International Conference on Learning Representations	ICLR
International Conference on Machine Learning	ICML
International Joint Conference on Artificial Intelligence	IJCAI
Journal of Machine Learning Research	JMLR
Machine Learning and Systems	MLSys
Reinforcement Learning Journal	RLJ
Workshop on Computer Games	CGW

Arpit Agarwal, Shivani Agarwal, Sepehr Assadi, and Sanjeev Khanna. Learning with Limited Rounds of Adaptivity: Coin Tossing, Multi-Armed Bandits, and Ranking from Pairwise Comparisons. In *COLT*, 2017.

https://proceedings.mlr.press/v65/agarwal17c.html.

Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K Hubert, and David Silver. Planning in Stochastic Environments with a Learned Model. In *ICLR*, 2022.

https://openreview.net/forum?id=X6D9bAHhBQ1.

Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. Best Arm Identification in Multi-armed Bandits. In *COLT*, 2010.

https://www.learningtheory.org/colt2010/papers/59Audibert.pdf.

Maryam Aziz, Jesse Anderton, Kevin Jamieson, Alice Wang, Hugues Bouchard, and Javed Aslam. Identifying new podcasts with high general appeal using a pure exploration infinitely-armed bandit strategy. In *RecSys*, 2022.

https://dl.acm.org/doi/abs/10.1145/3523227.3546766.

Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, John Quan, George Papamakarios, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Luyu Wang, Wojciech Stokowiec, and Fabio Viola. The DeepMind JAX Ecosystem. https://github.com/google-deepmind, 2020.

https://github.com/google-deepmind.

Nolan Bard, Jakob N Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, et al. The Hanabi challenge: A new frontier for AI research. *Artificial Intelligence*, 280:103216, 2020.

https://www.sciencedirect.com/science/article/pii/S0004370219300116.

Petr Baudiš and Jean-loup Gailly. PACHI: State of the Art Open Source Go Program.

In *ACG*, 2012.

https://link.springer.com/chapter/10.1007/978-3-642-31866-5_3.

M G Bellemare, Y Naddaf, J Veness, and M Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *JAIR*, 47:253–279, 2013.

https://www.jair.org/index.php/jair/article/view/10819.

Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *arXiv:1912.06680*, 2019.

https://doi.org/10.48550/arXiv.1912.06680.

Clément Bonnet, Daniel Luo, Donal Byrne, Shikha Surana, Vincent Coyette, Paul Duckworth, Laurence I. Midgley, Tristan Kalloniatis, Sasha Abramowitz, Cemlyn N. Waters, Andries P. Smit, Nathan Grinsztajn, Ulrich A. Mbou Sob, Omayma Mahjoub, Elshadai Tegegn, Mohamed A. Mimouni, Raphael Boige, Ruan de Kock, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. Jumanji: a Diverse Suite of Scalable Reinforcement Learning Environments in JAX. *ICLR*, 2024.

https://openreview.net/forum?id=C4CxQmp9wc.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. https://github.com/google/jax, 2018.

https://github.com/google/jax.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman,

Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540*, 2016.

https://doi.org/10.48550/arXiv.1606.01540.

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

https://www.science.org/doi/full/10.1126/science.aao1733.

Noam Brown and Tuomas Sandholm. Superhuman AI for multiplayer poker. *Science*, 365(6456):885–890, 2019.

https://www.science.org/doi/full/10.1126/science.aay2400.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *arXiv:2005.14165*, 2020.

https://doi.org/10.48550/arXiv.2005.14165.

Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure Exploration in Multi-armed Bandits Problems. In *ALT*, 2009.

https://link.springer.com/chapter/10.1007/978-3-642-04414-4_7.

Tristan Cazenave. Sequential Halving Applied to Trees. *IEEE T-CIAIG*, 7(1):102–105, 2014.

https://ieeexplore.ieee.org/abstract/document/6799192.

Johan Samir Obando Ceron and Pablo Samuel Castro. Revisiting Rainbow: Promoting more Insightful and Inclusive Deep Reinforcement Learning Research. In *ICML*, 2021.

https://proceedings.mlr.press/v139/ceron21a.html.

Guillaume MJ B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel Monte-Carlo Tree Search. In *CG*, 2008.

https://doi.org/10.1007/978-3-540-87608-3_6.

Gabriele Cirulli. 2048. https://gabrielecirulli.github.io/2048/, 2014.

https://gabrielecirulli.github.io/2048/.

Jonathan Citrin, Ian Goodfellow, Akhil Raju, Jeremy Chen, Jonas Degrave, Craig Donner, Federico Felici, Philippe Hamel, Andrea Huber, Dmitry Nikulin, et al. TORAX: A Fast and Differentiable Tokamak Transport Simulator in JAX. *arXiv:2406.06718*, 2024.

https://doi.org/10.48550/arXiv.2406.06718.

Rémi Coulom. Whole-History Rating: A Bayesian Rating System for Players of Time-Varying Strength. In *CG*, 2008.

https://link.springer.com/chapter/10.1007/978-3-540-87608-3_11.

Steven Dalton and Iuri Frosio. Accelerating Reinforcement Learning through GPU Atari Emulation. In *NeurIPS*, 2020.

https://proceedings.neurips.cc/paper/2020/hash/e4d78a6b4d93e1d79241f7b282fa3413-Abstract.html.

Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with Gumbel. In *ICLR*, 2022.

https://openreview.net/forum?id=bERaNdoegnO.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

Hossein Esfandiari, Amin Karbasi, Abbas Mehrabian, and Vahab Mirrokni. Regret Bounds for Batched Bandits. In *AAAI*, 2021.

https://ojs.aaai.org/index.php/AAAI/article/view/16901.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. IMPALA: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *ICML*, 2018.

https://proceedings.mlr.press/v80/espeholt18a.html.

Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. SEED RL: Scalable and efficient deep-rl with accelerated central inference. In *ICLR*, 2020.

https://openreview.net/forum?id=rkgvXlrKwH.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

https://www.nature.com/articles/s41586-022-05172-4.

Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. In *NeurIPS Track on Datasets and Benchmarks*, 2021.

https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/hash/d1f491a404d6854880943e5c3cd9ca25-Abstract-round1.html.

Zijun Gao, Yanjun Han, Zhimei Ren, and Zhengqing Zhou. Batched Multi-armed Bandits Problem. In *NeurIPS*, 2019.

https://papers.nips.cc/paper_files/paper/2019/hash/20f07591c6fcb220ffe637cda29bb3f6-Abstract.html.

Matthew L Ginsberg. GIB: Steps Toward an Expert-Level Bridge-Playing Program. In *IJCAI*, 1999.

https://www.ijcai.org/Proceedings/99-1/Papers/084.pdf.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural

Networks. In *AISTATS*, 2011.

https://proceedings.mlr.press/v15/glorot11a.html.

Florin Gogianu, Tudor Berariu, Mihaela C Rosca, Claudia Clopath, Lucian Busoniu, and Razvan Pascanu. Spectral Normalisation for Deep Reinforcement Learning: An Optimisation Perspective. In *ICML*, 2021.

https://proceedings.mlr.press/v139/gogianu21a.html.

Qucheng Gong, Yu Jiang, and Yuandong Tian. Simple is better: Training an end-to-end contract bridge bidding agent without human knowledge. In *Real-world Sequential Decision Making Workshop at ICML*, 2019.

https://realworld-sdm.github.io/paper/42.pdf.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. 2016.

https://www.deeplearningbook.org/.

Aditya Grover, Todor Markov, Peter Attia, Norman Jin, Nicolas Perkins, Bryan Cheong, Michael Chen, Zi Yang, Stephen Harris, William Chueh, and Stefano Ermon. Best arm identification in multi-armed bandits with delayed feedback. In *AISTATS*, 2018.

https://proceedings.mlr.press/v84/grover18b.html.

Cole Gulino, Justin Fu, Wenjie Luo, George Tucker, Eli Bronstein, Yiren Lu, Jean Harb, Xinlei Pan, Yan Wang, Xiangyu Chen, John Co-Reyes, Rishabh Agarwal, Rebecca Roelofs, Yao Lu, Nico Montali, Paul Mougin, Zoey Yang, Brandyn White, Aleksandra Faust, Rowan McAllister, Dragomir Anguelov, and Benjamin Sapp. Waymax: An Accelerated, Data-Driven Simulator for Large-Scale Autonomous Driving Research. In *NeurIPS*, 2023.

https://papers.nips.cc/paper_files/paper/2023/hash/1838feeb71c4b4ea524d0df2f7074245-Abstract-Datasets_and_Benchmarks.html.

Danijar Hafner, Timothy P Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering

Atari with Discrete World Models. In *International Conference on Learning Representations*, 2021.

https://openreview.net/forum?id=0oabwyZbOu.

Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

https://www.nature.com/articles/s41586-020-2649-2.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity Mappings in Deep Residual Networks. In *ECCV*, 2016.

https://link.springer.com/chapter/10.1007/978-3-319-46493-0_38.

Johannes Heinrich, Marc Lanctot, and David Silver. Fictitious self-play in extensive-form games. In *ICML*, 2015.

https://proceedings.mlr.press/v37/heinrich15.html.

John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. 6th edition, 2017.

https://www.elsevier.com/books-and-journals/book-companion/9780128119051.

Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable Reinforcement Learning. *arXiv:2104.06272*, 2021.

https://doi.org/10.48550/arXiv.2104.06272.

Shengyi Huang and Santiago Ontañón. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. In *FLAIRS*, 2022.

https://doi.org/10.32473/flairs.v35i.130584.

Shengyi Huang, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. The 37 Implementation Details of Proximal Policy Optimization. In

ICLR Blog Track, 2022a.

https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

Shengyi Huang, Anssi Kanervisto, Antonin Raffin, Weixun Wang, Santiago Ontañón, and Rousslan Fernand Julien Dossa. A2C is a special case of PPO. *arXiv:2205.09123*, 2022b.

https://doi.org/10.48550/arXiv.2205.09123.

Kevin Jamieson and Ameet Talwalkar. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *AISTATS*, 2016.

https://proceedings.mlr.press/v51/jamieson16.html.

Kevin Jamieson, Matthew Malloy, Robert Nowak, and Sebastien Bubeck. On Finding the Largest Mean Among Many. *arXiv:1306.3917*, 2013.

https://doi.org/10.48550/arXiv.1306.3917.

Tianyuan Jin, Jing Tang, Pan Xu, Keke Huang, Xiaokui Xiao, and Quanquan Gu. Almost Optimal Anytime Algorithm for Batched Multi-Armed Bandits. In *ICML*, 2021a.

https://proceedings.mlr.press/v139/jin21c.html.

Tianyuan Jin, Pan Xu, Xiaokui Xiao, and Quanquan Gu. Double Explore-then-Commit: Asymptotic Optimality and Beyond. In *COLT*, 2021b.

https://proceedings.mlr.press/v134/jin21a.html.

Kwang-Sung Jun, Kevin Jamieson, Robert Nowak, and Xiaojin Zhu. Top Arm Identification in Multi-Armed Bandits with Batch Arm Pulls. In *AISTATS*, 2016.

https://proceedings.mlr.press/v51/jun16.html.

Cem Kalkanli and Ayfer Ozgur. Batched Thompson Sampling. In *NeurIPS*, 2021.

https://proceedings.neurips.cc/paper/2021/hash/fb647ca6672b0930e9d00dc384d8b16f-Abstract.html.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv:2001.08361*, 2020.

https://doi.org/10.48550/arXiv.2001.08361.

Amin Karbasi, Vahab Mirrokni, and Mohammad Shadravan. Parallelizing Thompson Sampling. In *NeurIPS*, 2021.

https://proceedings.neurips.cc/paper_files/paper/2021/hash/56f0b515214a7ec9f08a4bbf9a56f7ba-Abstract.html.

Zohar Karnin, Tomer Koren, and Oren Somekh. Almost Optimal Exploration in Multi-Armed Bandits. In *ICML*, 2013.

https://proceedings.mlr.press/v28/karnin13.html.

Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR*, 2015.

https://doi.org/10.48550/arXiv.1412.6980.

Haruka Kita, Sotetsu Koyamada, Yotaro Yamaguchi, and Shin Ishii. A Simple, Solid, and Reproducible Baseline for Bridge Bidding AI. *IEEE CoG*, 2024.

https://doi.org/10.1109/CoG60054.2024.10645547.

Sotetsu Koyamada, Keigo Habara, Nao Goto, Shinri Okano, Soichiro Nishimori, and Shin Ishii. Mjx: A framework for Mahjong AI research. In *IEEE CoG*, 2022.

https://doi.org/10.1109/CoG51982.2022.9893712.

Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-Accelerated Parallel Game Simulators for Reinforcement Learning. In *NeurIPS*, 2023.

https://papers.nips.cc/paper_files/paper/2023/hash/8f153093758af93861a74a1305dfdc18-Abstract-Datasets_and_Benchmarks.html.

Sotetsu Koyamada, Soichiro Nishimori, and Shin Ishii. A Batch Sequential Halving Algorithm without Performance Degradation. *RLJ*, 5:2218–2232, 2024.

https://rlj.cs.umass.edu/2024/papers/Paper314.html.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

Harold W Kuhn. A simplified two-person poker. *Contributions to the Theory of Games*, 1:97–103, 1950.

https://doi.org/10.1515/9781400881727-010.

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. *arXiv:1908.09453*, 2019.

https://doi.org/10.48550/arXiv.1908.09453.

Robert Tjarko Lange. gymnax: A JAX-based Reinforcement Learning Environment Library. http://github.com/RobertTLange/gymnax, 2022.

http://github.com/RobertTLange/gymnax.

Junjie Li, Sotetsu Koyamada, Qiwei Ye, Guoqing Liu, Chao Wang, Ruihan Yang, Li Zhao, Tao Qin, Tie-Yan Liu, and Hsiao-Wuen Hon. Suphx: Mastering Mahjong with Deep Reinforcement Learning. *arXiv:2003.13590*, 2020a.

https://doi.org/10.48550/arXiv.2003.13590.

Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A System for Massively

Parallel Hyperparameter Tuning. In *MLSys*, 2020b.

https://proceedings.mlsys.org/paper_files/paper/2020/hash/a06f20b349c6cf09a6b171c71b88bbfc-Abstract.html.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *JMLR*, 18(185):1–52, 2018.

https://www.jmlr.org/papers/v18/16-558.html.

Anji Liu, Jianshu Chen, Mingze Yu, Yu Zhai, Xuewen Zhou, and Ji Liu. Watch the unobserved: A simple approach to parallelizing monte carlo tree search. In *ICLR*, 2020.

https://openreview.net/forum?id=BJlQtJSKDB.

Edward Lockhart, Neil Burch, Nolan Bard, Sebastian Borgeaud, Tom Eccles, Lucas Smaira, and Ray Smith. Human-Agent Cooperation in Bridge Bidding. In *Cooperative AI Workshops at NeurIPS*, 2020.

https://doi.org/10.48550/arXiv.2011.14124.

Chris Lu, Jakub Kuba, Alistair Letcher, Luke Metz, Christian Schroeder de Witt, and Jakob Foerster. Discovered Policy Optimisation. In *NeurIPS*, 2022.

https://openreview.net/forum?id=vsQ4gufZ-Ve.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac Gym: High Performance GPU-Based Physics Simulation For Robot Learning. In *NeurIPS Track on Datasets and Benchmarks*, 2021.

https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/hash/28dd2c7955ce926456240b2ff0100bde-Abstract-round2.html.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al.

Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618 (7964):257–263, 2023.

https://www.nature.com/articles/s41586-023-06004-9.

Michael Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Jackson, Samuel Coward, and Jakob Foerster. Craftax: A Lightning-Fast Benchmark for Open-Ended Reinforcement Learning. In *ICML*, 2024.

https://proceedings.mlr.press/v235/matthews24a.html.

Mehdi Mhalla and Frédéric Prost. Gardner's Minichess Variant is Solved. *ICGA Journal*, 36(4):215–221, 2013.

https://doi.org/10.3233/ICG-2013-36404.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

https://www.nature.com/articles/nature14236.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

https://proceedings.mlr.press/v48/mniha16.html.

Alexander Nikulin, Vladislav Kurenkov, Ilya Zisman, Viacheslav Sinii, Artem Agarkov, and Sergey Kolesnikov. XLand-MiniGrid: Scalable Meta-Reinforcement Learning Environments in JAX. In *NeurIPS*, 2024.

https://doi.org/10.48550/arXiv.2312.12044.

OpenAI. GPT-4 technical report. *arXiv:2303.08774*, 2023.

https://doi.org/10.48550/arXiv.2303.08774.

Keiran Paster, Sheila McIlraith, and Jimmy Ba. You Can't Count on Luck: Why Decision Transformers and RvS Fail in Stochastic Environments. In *NeurIPS*, 2022.

https://openreview.net/forum?id=atb3yifRtX.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.

https://papers.nips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Tom Pepels, Tristan Cazenave, Mark HM Winands, and Marc Lanctot. Minimizing Simple and Cumulative Regret in Monte-Carlo Tree Search. In *CGW*, 2014.

https://link.springer.com/chapter/10.1007/978-3-319-14923-3_1.

Vianney Perchet, Philippe Rigollet, Sylvain Chassang, and Erik Snowberg. Batched bandit problems. *Ann. Statist.*, 44(2):660 – 681, 2016.

https://doi.org/10.1214/15-AOS1381.

Eduardo Pignatelli, Jarek Liesen, Robert Tjarko Lange, Chris Lu, Pablo Samuel Castro, and Laura Toni. NAVIX: Scaling MiniGrid Environments with JAX. *arXiv:2407.19396*, 2024.

https://doi.org/10.48550/arXiv.2407.19396.

D. Provodin, P. Gajane, M. Pechenizkiy, and M. Kaptein. The Impact of Batch Learning in Stochastic Linear Bandits. In *ICDM*, 2022.

https://doi.ieeecomputersociety.org/10.1109/ICDM54844.2022.00146.

Zizhang Qiu, Shouguang Wang, Dan You, and MengChu Zhou. Bridge Bidding via Deep Reinforcement Learning and Belief Monte Carlo Search. *IEEE/CAA JAS*, 11 (10):2111–2122, 2024.

https://doi.org/10.1109/JAS.2024.124488.

Jiang Rong, Tao Qin, and Bo An. Competitive Bridge Bidding with Deep Neural Networks. In *AAMAS*, 2019.

https://dl.acm.org/doi/abs/10.5555/3306127.3331669.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garðar Ingvarsson, Timon Willi, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, et al. JaxMARL: Multi-Agent RL Environments and Algorithms in JAX. In *NeurIPS*, 2024.

https://doi.org/10.48550/arXiv.2311.10090.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

https://www.nature.com/articles/s41586-020-03051-4.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*, 2017.

https://doi.org/10.48550/arXiv.1707.06347.

Richard B Segal. On the Scalability of Parallel UCT. In *CG*, 2010.

https://doi.org/10.1007/978-3-642-17928-0_4.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

https://www.nature.com/articles/nature16961.

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, 2017.

https://www.nature.com/articles/nature24270.

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

https://www.science.org/doi/10.1126/science.aar6404.

Finnegan Southey, Michael P Bowling, Bryce Larson, Carmelo Piccione, Neil Burch, Darse Billings, and Chris Rayner. Bayes' Bluff: Opponent Modelling in Poker. In *UAI*, 2005.

https://doi.org/10.48550/arXiv.1207.1411.

Richard Sutton. The Bitter Lesson, 2019.

http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

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

http://incompleteideas.net/book/the-book-2nd.html.

Justin K Terry, Benjamin Black, Mario Jayakumar, Ananth Hari, Luis Santos, Clemens Dieffendahl, Niall L Williams, Yashas Lokesh, Ryan Sullivan, Caroline Horsch, and Praveen Ravi. PettingZoo: Gym for Multi-Agent Reinforcement Learning. In *NeurIPS*, 2021.

https://openreview.net/forum?id=fLnsj7fpbPI.

Gerald Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38

(3):58–68, 1995.
https://dl.acm.org/doi/10.1145/203330.203343.

Yuandong Tian, Qucheng Gong, and Yu Jiang. Joint Policy Search for Multi-agent Collaboration with Imperfect Information. In *NeurIPS*, 2020.
https://proceedings.neurips.cc//paper/2020/hash/e64f346817ce0c93d7166546ac8ce683-Abstract.html.

David Tolpin and Solomon Shimony. MCTS Based on Simple Regret. In *AAAI*, 2012.
https://ojs.aaai.org/index.php/AAAI/article/view/8126.

John Tromp. The game of Go. https://tromp.github.io/go.html, 1995.
https://tromp.github.io/go.html.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. *NIPS*, 30, 2017.
https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Veronique Ventos, Yves Costel, Olivier Teytaud, and Solène Thépaut Ventos. Boosting a Bridge Artificial Intelligence. In *IEEE ICTAI*, 2017.
https://ieeexplore.ieee.org/abstract/document/8372096.

Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*,

575(7782):350–354, 2019.

https://www.nature.com/articles/s41586-019-1724-z.

Jiayi Weng, Huayu Chen, Dong Yan, Kaichao You, Alexis Duburcq, Minghao Zhang, Yi Su, Hang Su, and Jun Zhu. Tianshou: A Highly Modularized Deep Reinforcement Learning Library. *JMLR*, 23(267):1–6, 2022a.

https://www.jmlr.org/papers/v23/21-1127.html.

Jiayi Weng, Min Lin, Shengyi Huang, Bo Liu, Denys Makoviichuk, Viktor Makoviychuk, Zichen Liu, Yufan Song, Ting Luo, Yukun Jiang, et al. EnvPool: A Highly Parallel Reinforcement Learning Environment Execution Engine. In *NeurIPS*, 2022b.

https://openreview.net/forum?id=BubxnHpuMbG.

Tadao Yamaoka. dlshogi. https://github.com/TadaoYamaoka/DeepLearningShogi, 2017.

https://github.com/TadaoYamaoka/DeepLearningShogi.

Chih-Kuan Yeh, Cheng-Yu Hsieh, and Hsuan-Tien Lin. Automatic bridge bidding using deep reinforcement learning. *IEEE ToG*, 10(4):365–377, 2018.

https://ieeexplore.ieee.org/document/8438937.

Kenny Young and Tian Tian. MinAtar: An Atari-Inspired Testbed for Thorough and Reproducible Reinforcement Learning Experiments. *arXiv:1903.03176*, 2019.

https://doi.org/10.48550/arXiv.1903.03176.

Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and YI WU. The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games. In *NeurIPS*, 2022.

https://papers.nips.cc/paper_files/paper/2022/hash/9c1535a02f0ce079433344e14d910597-Abstract-Datasets_and_Benchmarks.html.

Daochen Zha, Kwei-Herng Lai, Songyi Huang, Yuanpu Cao, Keerthana Reddy, Juan Vargas, Alex Nguyen, Ruzhe Wei, Junyu Guo, and Xia Hu. RLCard: A Platform

for Reinforcement Learning in Card Games. In *IJCAI*, 2020.

https://www.ijcai.org/proceedings/2020/764.

Daochen Zha, Jingru Xie, Wenye Ma, Sheng Zhang, Xiangru Lian, Xia Hu, and Ji Liu. DouZero: Mastering DouDizhu with Self-Play Deep Deinforcement Learning. In *ICML*, 2021.

https://proceedings.mlr.press/v139/zha21a.html.

Yao Zhao, Connor Stephens, Csaba Szepesvari, and Kwang-Sung Jun. Revisiting Simple Regret: Fast Rates for Returning a Good Arm. In *ICML*, 2023.

https://proceedings.mlr.press/v202/zhao23g.html.