MooZi: A High-Performance Game-playing System that Plans with a Learned Model

by

Zeyi Wang

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department or Computing Science
University of Alberta

Abstract

The intent of this thesis is to develop a high-performance open-source system that plans with a learned model and to understand the algorithm through extensive analysis. We formulate the problem of maximizing accumulated rewards in Markov Decision Processes, and we frame playing games as such problems. We develop the *MooZi* system to solve these problems. *MooZi* includes (1) a MuZero-based algorithm that plans with a learned model (2) a distributed architecture that trains and evaluates the algorithm efficiently, and (3) a collection of tools to visualize and understand the algorithm. We empirically show that *MooZi* outperforms PPO and AC in MinAtar environments. We also show that *MooZi* learns to play the two-players board game *Breakthrough*. We use our tools to analyze the learned model by visualizing search trees and learned representation. We make *MooZi* publicly available to accelerate future research.

Preface

This dissertation is original, unpublished, independent work by the author, Zeyi Wang.

This thesis is dedicated to my dear parents.

Acknowledgements

I would like to thank my supervisor, Professor Martin Müller, for providing valuable guidance, editing oceans of mistakes in my thesis, and patiently enduring my hectic working schedule. Further, I would like to thank Dr. Chao Gao and Dr. Ting-Han Wei for providing feedback during the meetings. Thanks also to Jiuqi Wang for contributing to the project by implementing a multi-layer perceptron network.

Contents

Abstract iii

Contents xi

List of Tables xiii

List of Figures xv

List of Symbols xvii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Contributions of this Thesis . . . 2

2 Literature Review 3
- 2.1 Planning and Search . . . 3
- 2.2 Monte Carlo Methods . . . 4
- 2.3 Monte Carlo Tree Search (MCTS) . . . 4
  - 2.3.1 Selection . . . 5
  - 2.3.2 Expansion . . . 6
  - 2.3.3 Evaluation . . . 6
  - 2.3.4 Backpropagation . . . 7
  - 2.3.5 MCTS Iteration and Move Selection . . . 7
- 2.4 AlphaGo . . . 7
- 2.5 AlphaGo Zero . . . 9
- 2.6 AlphaZero . . . 9
- 2.7 *MuZero* . . . 10
  - 2.7.1 MuZero Reanalyze . . . 12
- 2.8 Atari Game Playing . . . 12
  - 2.8.1 The Atari Learning Environment . . . 12
  - 2.8.2 Deep Q-Networks . . . 13

2.8.3 Double Q Learning . . . 13
2.8.4 Experience Replay . . . 14
2.8.5 Network Architectures . . . 15
2.8.6 Scalar Transformation . . . 15
2.8.7 MinAtar . . . 16
2.8.8 Consistency Loss . . . 16
2.9 Deep Reinforcement Learning Systems . . . 17
2.9.1 Mnih et al.'s Asynchronous Methods Framework . . . 17
2.9.2 The IMPALA Architecture . . . 17
2.9.3 The SEED Architecture . . . 18
2.9.4 The Acme Framework . . . 19
2.9.5 Ray and RLlib . . . 19
2.9.6 JAX and Podracer Architecture . . . 20

3 Problem Definition **23**
3.1 Markov Decision Process and Agent-Environment Interface . . . 23
3.2 Policies and Value Functions . . . 24
3.3 Partially Observable Markov Decision Process . . . 24
3.4 Game Playing . . . 25

4 Method **27**
4.1 Design Philosophy . . . 27
4.1.1 Use of Pure Functions . . . 27
4.1.2 Training Efficiency . . . 29
4.1.3 Understanding the Method is Important . . . 30
4.2 Architecture Overview . . . 30
4.3 The *MooZi* System Components . . . 31
4.3.1 Environment Bridges . . . 31
4.3.2 Vectorized Environment . . . 32
4.3.3 Action Space Augmentation . . . 34
4.3.4 History Stacking . . . 34
4.3.5 The *MooZi* Neural Network . . . 36
4.3.6 Planner . . . 38
4.3.7 Training Target Generation . . . 39
4.3.8 Loss Computation . . . 40
4.3.9 Updating Neural Network Parameters . . . 42
4.3.10 *MooZi* Reanalyze . . . 43

4.3.11 Training Worker . . . 43
4.3.12 Testing Worker . . . 43
4.3.13 Reanalyze Worker . . . 44
4.3.14 Replay Buffer . . . 44
4.3.15 Parameter Server . . . 45
4.4 The *MooZi* System in Action . . . 45
4.5 Logging and Visualization . . . 45

5 Experiments **49**
5.1 Experiment Setup . . . 49
5.1.1 Basic Configuration . . . 49
5.1.2 Neural Network Configurations . . . 49
Residual Block . . . 50
The Representation Function Network . . . 50
The Prediction Function Network . . . 50
The Dynamics Function Network . . . 51
The Projection Function Network . . . 52
Network Training . . . 52
5.1.3 Planner Configurations . . . 52
5.1.4 Driver Configuration . . . 53
5.2 *MooZi* vs PPO in *MinAtar* Environments . . . 53
5.3 Sticky Actions in MinAtar . . . 56
5.4 Testing Strength when Scaling the Search Budget in *Asterix* . . . 58
5.5 Improving Sample Efficiency with Reanalyze in *Asterix* . . . 59
5.6 Analysis of Planning in *Space Invaders* . . . 61
5.7 Learning through Self-play in *Breakthrough* . . . 62
5.8 Analysis of Planning in *Breakthrough* . . . 63
5.9 Visualizing the Hidden Space of the Learned Representation . . . 66

6 Conclusion **69**

7 Future Work **71**

List of Tables

5.1 Planner Configuration 52

List of Figures

2.1 **The Monte Carlo Tree Search Framework, from Chaslot et al. [9].** . . . 5
2.2 **Illustration of *MuZero* planning, acting, and training with a learned model.** . 10
2.3 **IMPALA Architecture, from Espeholt et al. [14].**. . . . 18
2.4 **The SEED Architecture, from Espeholt et al. [15].** . . . 19
2.5 **Example of a distributed asynchronous agent with Acme, from Hoffman et al. [27].** . . . 20
2.6 **RLlib Abstraction Layers, from Liang et al. [39].** . . . 21
2.7 **Sebulba architecture, from Hessel et al. [26].** . . . 22

3.1 **The Agent-Environment Interface, from Sutton and Barto [62].** . . . 23

4.1 **Computation graph of the simple dense layer in Algorithm 1.** . . . 29
4.2 **The *MooZi* Architecture.** . . . 31
4.3 **An example of history stacking.** . . . 37
4.4 **Self-consistency Loss Computation.** . . . 42
4.5 **Timeline of training steps.** . . . 45
4.6 ***MooZi Tensorboard* dashboard.** . . . 47
4.7 ***MooZi* produces trajectories in *Asterix* as GIFs with annotations, presented here as tiled images of every four frames.** . . . 47

5.1 ***MinAtar* Environments, from Young and Tian [67].** . . . 54
5.2 **Evaluation of *MooZi* in *MinAtar* games.** . . . 55
5.3 **Evaluation of *MooZi* vs PPO vs AC in two variants of *Breakout*.** . . . 57
5.4 **Agent strength with different number of simulations.** . . . 58
5.5 ***MooZi* with reanalyze workers in *Asterix*.** . . . 60
5.6 ***MooZi* acting in *Space Invaders* environment by planning with a learned model.** . . . 61
5.7 **The board game *Breakthrough*.** . . . 62
5.8 **Evaluation of *MooZi* training in *Breakthrough*.** . . . 63
5.9 ***MooZi* planning with a learned model in *Breakthrough*.** . . . 64

5.10 ***MooZi* planning with a learned model that fully captures environment dynamics.** 65
5.11 **The hidden space of the learned representation visualized through *t-distributed stochastic neighbor embedding (t-SNE).*** 67

List of Symbols

Symbol	Description	Section
s	state	3.1
a	action	3.1
r	reward	3.1
t	timestep	3.1
T	terminal timestep	3.1
γ	discount	3.1
o	partially observable environment frame	3.3
G	return	3.2
G^N	N-step return	3.2
V	value function	3.2
Q	state-action value function	3.2
δ	TD-error or value diff	4.3.14
$\mathcal{A}^e$	environment action space	4.3.3
$\mathcal{A}^a$	agent action space	4.3.3
$\mathcal{S}$	state space	3.1
$\mathcal{O}$	observation space	3.1
$\mathcal{T}_t$	step sample	4.3.7
$\mathcal{T}$	trajectory sample	4.3.7
$\mathcal{L}$	loss function	4.3.8
$\mathbf{x}$	hidden state	4.3.5
h	representation function	4.3.5
g	dynamics function	4.3.5
f	prediction function	4.3.5
ϱ	projection function	4.3.5
v	value prediction	4.3.5
v^*	value after search	4.3.6
$\hat{r}$	reward prediction	4.3.5
$\mathbf{p}$	policy prediction	4.3.5

$\mathbf{p}^*$	policy after search	4.3.6
Z	support of the scalar transformation	2.8.6, 4.3.5
B	batch size	4.3.4
H	height	4.3.1
W	width	4.3.1
C_e	environment channels	4.3.1
C_h	history channels	4.3.4
K	number of unrolled steps	4.3.7
L	history length	4.3.7
N	bootstrap length for N-step return	4.3.7

1 Introduction

Deep Learning (DL) is a branch of **Artificial Intelligence (AI)** that emphasizes the use of neural networks to fit any arbitrary function represented by a dataset. The training of a neural network is done as follows: compute a loss function from a batch of data, back-propagate gradients with respect to the loss, and update weights and biases based on the gradients. Deep learning techniques have been widely adopted in many domains, including computer vision, natural language processing, and robotics.

Reinforcement Learning (RL) is a branch of AI that emphasizes solving decision making problems with delayed rewards through trial and error. RL had most success in the domain of **game playing** [60], in which the algorithm is represented as an **agent** and interacts with game environments, such as board games and Atari games. An extension to game playing is **general game playing (GGP)** [19], with its goal of designing a single agent that can play many different games without having much prior knowledge of the games.

Deep Reinforcement Learning (DRL) is a popular research area that combines DL and RL to solve decision making problems. In a DRL system, the RL techniques lay out the structure of the algorithm such as the use of the **agent-environment interface**, a value function, a reward signal, etc., while the DL techniques are used to approximate value functions and learn representations [62].

Planning refers to any computational process that analyzes generated actions and their consequences in an environment [62]. In RL terms, planning means the use of a model to improve a policy. In board games where perfect models are accessible, planning with these models yields great performance. The most significant achievement of planning with a perfect model is AlphaGo beating human champions in Computer Go [60]. However, how to plan in games where no perfect models are available remains a challenging problem to researchers.

A **distributed system** utilizes concurrency through multiple processes or computer nodes to complete tasks. DRL systems for solving large problems are both data and compute intensive. Utilizing concurrency to increase efficiency and throughput for these DRL systems is sometimes necessary. Building a distributed system to achieve such concurrency is a common practice in industry, but requires significant engineering effort.

1.1 Motivation

Schrittwieser et al. developed MuZero [56], an algorithm that plans with a learned model (reviewed in Section 2.7). This algorithm achieved state-of-the-art performance in both Atari games and board games. However, the source code of the algorithm is not publicly available, and the pseudo-code provided with the paper isn't sufficient to reproduce the full algorithm. Moreover, MuZero requires much more computation than other RL algorithms, and an inefficient implementation will drastically slow down experimentation. The algorithm learns a model using a neural network, and such a model, like other applications using neural networks, is impossible to understand with a casual glance of the learned weights. We need a publicly available efficient implementation of an algorithm that plans with a learned model and tools that help us understand the learned model. This helps researchers understand how the algorithm and facilitates future research.

1.2 Contributions of this Thesis

In this thesis we present the project ***MooZi***, a system that plays games by planning with a learned model. This project includes:

- A collection of environment bridges that connect the system to common RL environments such as *MinAtar* [66], Atari [5], and *OpenSpiel* [38].
- Neural networks that learn a representation and can be used for planning.
- A MCTS based planner that uses the learned model to perform planning.
- A concurrent computing system that efficiently trains the model.
- Empirical studies and analysis of the system using environments from *MinAtar* and *OpenSpiel*.

The source code of the project can be found on GitHub at `https://github.com/uduse/moozi`.

2 Literature Review

2.1 Planning and Search

Many AI problems can be reduced to a search problem [65, p.39]. Such search problems can be solved by determining the best plan, path, model, function, and so on, based on some metrics of interest. Therefore, search has played a vital role in AI research since its dawn. The terms **planning** and **search** are widely used across different domains. Here we adopt the definition by Sutton and Barto [62].

Planning refers to any process by which the agent updates the action selection policy $\pi(a \mid s)$ or the value function $V_\pi(s)$. We will focus on the case of improving the policy in our discussion. We view the planning process as an operator $\mathcal{I}$ that takes the policy π as input and outputs an improved policy $\mathcal{I}\pi$.

Planning methods can be categorized based on the target state s they aim to improve. If the method improves the policy for arbitrary states, we call it **background planning**. That is, for any timestep t and a set of states $S' \subset \mathcal{S}$:

$$\pi(a \mid s) \leftarrow \mathcal{I}\pi(a \mid s), \ \ \forall s \in \mathcal{S}'$$

Typical background planning methods include **dynamic programming** and **Dyna-Q** [62]. In the case of dynamic programming, a full sweep of the state space is performed and all states are updated. In the case of Dyna, a subset of the state space is selected for update.

Another type of planning focuses on improving the policy of the current state s_t. We call this **decision-time planning**. That is, for any timestep t:

$$\pi(a \mid s) \leftarrow \mathcal{I}\pi(a \mid s), s = s_t$$

Algorithms such as AlphaGo use both types of planning when they use self-play for training. For decision-time planing, a tree search is performed at the root node and updates the policy of the current state. For background planning, a neural network uses past experience to train and update policy for all states.

An early example of the use of search as a planning method is the **A*** algorithm. In 1968, Hart, Nilsson, and Raphael designed this algorithm for finding a shortest path from a start vertex to a target vertex [21]. Although A* works quite well for many problems, especially in early game AI, it falls short in cases where the assumptions of A* do not hold. For example, A* requires a heuristic, and an optimal solution under stochastic environments. It is computationally infeasible on large problems. To address this problem, Korf framed the problem of **Real-Time Heuristic Search**, where the agent has to make a decision in each timestep with bounded computation, and developed the **Real-Time-A*** algorithm as a modified version of A* with bounded computation per step [37]. Tree-based search algorithms such as **MiniMax** and **Alpha-Beta Pruning** were developed to play and solve two-player games [34]. Monte Carlo techniques are designed to handle complex environments.

2.2 Monte Carlo Methods

In 1873, Joseph Jagger observed the bias in roulette wheels at the Monte Carlo Casino. He studied the bias by recording the results of roulette wheels and won over 2 million francs over several days by betting on the most favorably biased wheel [46]. Therefore, **Monte Carlo (MC)** methods gained their name as a class of algorithms based on random sampling.

MC methods are used in many domains but in this thesis we will primarily focus on its usage in search. In a game where terminal states are usually unreachable by the limited search depth, evaluation has to be performed on the leaf nodes that represent intermediate game states. One way of obtaining an evaluation on a state is by applying a heuristic function. Heuristic functions used this way are usually hand-crafted by human based on expert knowledge, and hence are prone to human error. The other way of evaluating the state is to perform a rollout from that state to a terminal state by selecting actions by some randomized policy. This evaluation process is called **random rollout** or **Monte Carlo rollout**.

2.3 Monte Carlo Tree Search (MCTS)

Kocsis and Szepesvári developed the **Upper Confidence Bounds applied to Trees (UCT)** method as an extension of the **Upper Confidence Bound (UCB)** algorithm employed in multi-armed bandit problems [36]. Rémi Coulom developed the general idea of **Monte Carlo Tree Search** that combines Monte Carlo rollouts with tree search [10] for his Go program CrazyStone. Shortly afterwards, Gelly et al. implemented another Go program MoGo that uses the UCT selection formula [18]. MCTS was generalized by Chaslot et al. as a framework for game AI [9]. This framework requires less domain knowledge than classic approaches to game

AI while often giving better results. The core idea of this framework is to gradually build the search tree by iteratively applying four steps: **selection**, **expansion**, **evaluation**, and **backpropagation**. The search tree built in this way emphasizes search of more promising moves and game states based on collected statistics in rollouts. More promising states are visited more often, have more children, have deeper subtrees, and rollout results are aggregated to yield more accurate values. Here we detail the four steps in the MCTS framework by Chaslot et al. (see Figure 2.1).

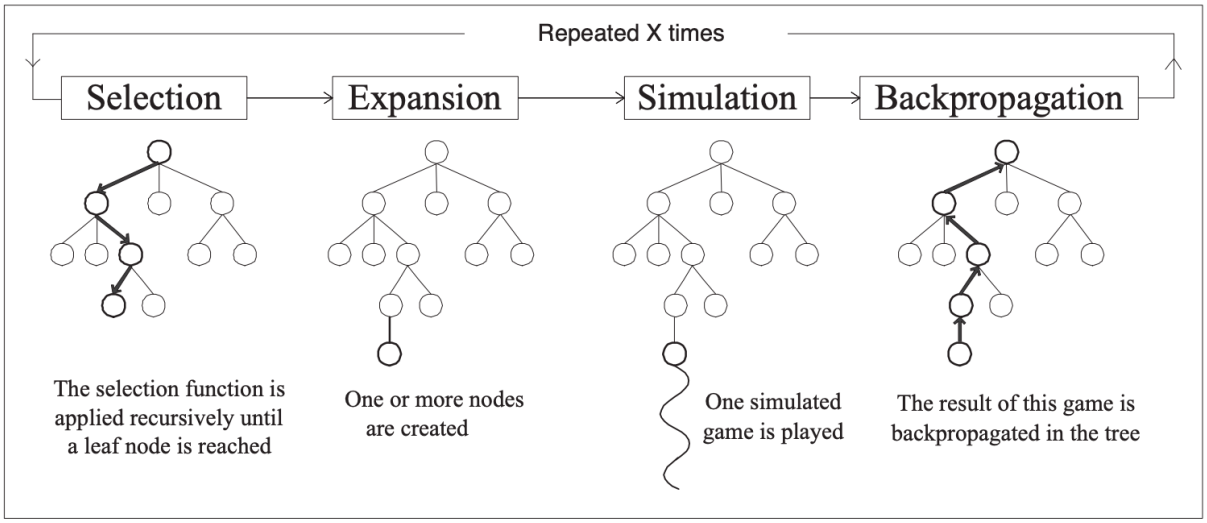

Figure 2.1: The Monte Carlo Tree Search Framework, from Chaslot et al. [9].

2.3.1 Selection

The selection process starts at the root node and repeats until a leaf node in the current tree is reached. At each level of the tree, a child node is selected based on a selection formula such as UCT or PUCT. A selection formula usually has two parts: the exploitation part based on the evaluation function E, and the exploration bonus function B. For actions $(s, a), a \in \mathcal{A}$ of a parent state s , the selection $I(s)$ is defined as

$$I(s) \doteq \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[E(s, a) + B(s, a) \right] \tag{2.1}$$

The evaluation function E can be based on the value of the child, the accumulated reward of the child, or the prior selection probability based on the policy $\pi(a \mid s)$. The exploration bonus function B is usually based on the visit count of the child and the parent. The more visits a child

has, the smaller the exploration bonus becomes. For example, the UCT algorithm uses

$$E(s,a) = \frac{V(s)}{N(s,a)}$$
$$B(s,a) = \sqrt{\frac{2 * \log(\sum_{b \in \mathcal{A}} N(s,b))}{N(s,a)}}$$

where $V(s)$ is the value of the node, and $N(s,a)$ is the visit count of the edge. Gelly et al. used this selection rule in their implementation of MoGo, the first computer Go program that uses UCT [18]. Rosin developed the PUCB and the PUCT algorithm that utilize a predictor $P(s,a)$ that estimates the prior probability of the action a being selected from state s. This approach was later used in AlphaGo (Section 2.5 and [52]).

2.3.2 Expansion

The selected leaf node is expanded by adding one or more children. Each child represents a successor game state reached by playing the associated legal move.

2.3.3 Evaluation

The expanded node is evaluated, either by playing a game with a rollout policy, or by using an evaluation function, or by using a blend of both approaches. Many MCTS algorithms use a randomized policy as the rollout policy, and the game result as the evaluation. Early work on evaluation functions focused on hand-crafted or machine learned heuristic functions based on expert knowledge. Recently, evaluation functions use deep neural networks specifically trained for the given problem (Section 2.4 gives an example).

2.3.4 Backpropagation

After the expanded nodes are evaluated, the nodes on the path from the expanded nodes back to the root are updated. The statistics updated usually include visit count, estimated value and accumulated reward of the nodes.

2.3.5 MCTS Iteration and Move Selection

The four MCTS steps are repeated until the budget runs out. The budget is usually a limited number of simulations or a period of time. After the search, the agent acts by selecting the action associated with the most promising child of the root node. This could be the most visited

child, the child with the greatest value, or the child with the greatest lower confidence bound value [53, 64].

2.4 AlphaGo

In 2017, Silver et al. developed **AlphaGo**, the first Go program that beat a human Go champion on even terms [60]. AlphaGo was trained with a machine learning pipeline with multiple stages. For the first stage of training, a supervised learning policy (or SL policy) is trained to predict expert moves using a neural network. This SL policy p is parametrized by weights σ, denoted p_σ. The input of the policy network is a representation of the board state, denoted s. The network outputs a probability distribution over all legal moves a through the last softmax layer. During the training of the network, randomly sampled expert moves are used as training targets. The weights σ are then updated through gradient ascent to maximize the probability of matching the human expert move:

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a \mid s)}{\partial \sigma}$$

For the second stage of training, the supervised policy p_σ is used as the starting point for training with reinforcement learning. This reinforcement learning trained policy (or RL policy) is parametrized by weights ρ and is initialized $p_\rho = p_\sigma$. Training data is generated in form of self-play games using p_ρ as the rollout policy. For each game, the game outcome $z_t = \pm r(s_T)$, where s_T is the terminal state, $z_T = +1$ for winning, $z_T = -1$ for losing from the perspective of the current player. Weights ρ are updated using gradient ascent to maximize the expected outcome using the update formula:

$$\Delta\rho \propto \frac{\partial \log p_\rho\left(a_t \mid s_t\right)}{\partial \rho} z_t$$

Finally, a value function is trained to evaluate board positions. This value function is modeled with a neural network with weights θ, denoted V_θ. Given a state s, $V_\theta(s)$ predicts the outcome of the game if both players act according to the policy p_ρ. This neural network is trained with stochastic gradient descent to minimize the mean squared error (MSE) between the predicted value $V_\theta(s)$ and the outcome z.

$$\Delta\theta \propto \frac{\partial V_\theta(s)}{\partial \theta}\left(z - V_\theta(s)\right)$$

AlphaGo combines the policy network p_ρ and the value network V_θ with MCTS for acting. AlphaGo uses a MCTS variant called PUCT similar to that described in Section 2.3. In the search tree, each edge (s, a) stores an action value $Q(s, a)$, a visit count $N(s, a)$, and a prior probability

$P(s,a)$. At each time step, the search starts at the root node and simulates until the budget runs out. In the selection phase of each simulation, an action is selected for each traversed node using the base formula in Equation 2.1. In AlphaGo, the exploitation score of the selection formula is the estimated average value of the next state after taking the action $Q(s,a)$. In AlphaGo's PUCT formula, the exploration bonus of edge (s,a) is based on the prior probability P and decays as its visit count N grows. As before, the action taken at time t maximizes the sum of the exploitation score and the exploration bonus

$$\begin{aligned} I(s) &= \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[E(s,a) + B(s,a) \right] \\ E(s,a) &= Q\left(s,a\right) \\ B(s,a) &\propto \frac{P(s,a)}{1+N(s,a)} \end{aligned}$$

AlphaGo evaluates a leaf node state s_L by blending both the value network estimation $V_\theta(s_L)$ and the game result z_L obtained by the rollout policy p_π. The mixing parameter $\lambda \in [0,1]$ is used to balance these two types of evaluations into the final evaluation $V(s_L)$

$$V\left(s_L\right) = (1-\lambda) V_\theta\left(s_L\right) + \lambda z_L \ .$$

2.5 AlphaGo Zero

AlphaGo Zero is a successor of AlphaGo that beat AlphaGo by 100-0 in 100 games [60]. In contrast, AlphaGo Zero learns to play Go from *tabula rasa*. This means it learns solely by reinforcement learning from self-play, starting from random play, without supervised learning from human expert data.

Central to AlphaGo Zero is a deep neural network f_θ with parameters θ. Given a state s as an input, the network outputs both move probabilities $\boldsymbol{p}$ and value estimation v

$$(\mathbf{p}, v) = f_\theta(s)$$

To generate self-play games $s_1, ..., s_T$, MCTS is performed at each state s using the latest neural network f_θ. To select a move for a parent node p in the search tree, a variant of the PUCT

algorithm is used:

$$\begin{aligned} I(s) &= \text{argmax}_{a \in \mathcal{A}} \left(E(s,a) + B(s,a) \right) \\ E(s,a) &= Q(s,a) \\ B(s,a) &\propto P(s,a) \frac{\sqrt{\sum_{b \in \mathcal{A}} N(s,b)}}{1 + N(s,a)} \end{aligned}$$

.

Self-play games are processed into training targets to update the network parameters θ through gradient descent on the loss function l

$$\mathcal{L}(\theta) = (z - v)^2 - \boldsymbol{\pi}^{\mathrm{T}} \log \boldsymbol{p} + c\|\theta\|^2$$

Here $(z - v)^2$ is the mean squared error of the prediction value, $-\boldsymbol{\pi}^{\mathrm{T}} \log \boldsymbol{p}$ is the cross-entropy loss of the move probabilities, and $c\|\theta\|^2$ is a L_2 weight regularization. Many other components of this system are similar to those in AlphaGo.

2.6 AlphaZero

AlphaZero reduces game specific knowledge of AlphaGo Zero even further so that the same algorithm can be also applied to Shogi and chess [59]. One generalization is that in AlphaZero the game result is no longer either winning or losing ($z \in \{-1, +1\}$), but can also be a draw ($z \in \{-1, 0, +1\}$).

2.7 *MuZero*

In 2020, Schrittwieser et al. developed **MuZero**, an even more general algorithm that learns to play Atari, Go, chess and Shogi at superhuman level. Compared to AlphaGo and AlphaZero, MuZero has no access to a perfect model of the game. MuZero plans with a neural network that also learns the game dynamics through experience. Therefore, MuZero can be applied to games where the perfect model is either not known or is infeasible to compute with. Figure 2.2 illustrates how MuZero plans, acts, and trains with a learned model.

MuZero defines three main functions. The **representation function** h encodes a history of observations $o_1, o_2, \ldots, o_t$ and actions $a_1, a_2, \ldots, a_{t-1}$ into a hidden state $\mathbf{x}_t^0$. This hidden state representation is learned, and is the main conceptual change from AlphaZero. The **dynamics function** g implements action execution in the representation. Given a hidden state $\mathbf{x}^k$ and action a^k, produces an immediate reward r^k and the next hidden state $\mathbf{x}^{k+1}$. The **prediction**

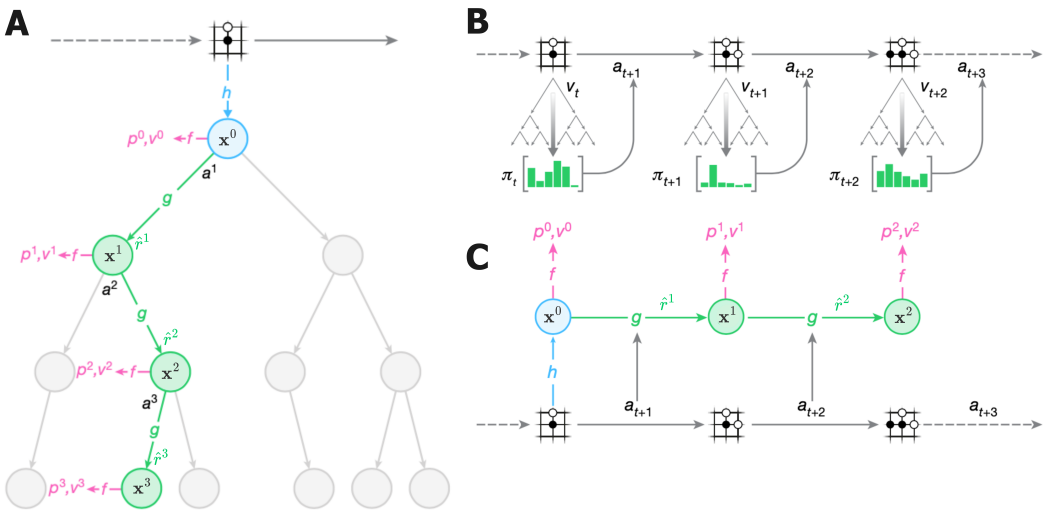

Figure 2.2: Illustration of *MuZero* planning, acting, and training with a learned model. *A*: MuZero plans with MCTS and the learned model. The *representation function* h is used to construct the root node. The *prediction function* f is used to estimate node values and action priors. The *dynamics function* g is used to estimate state transitions and transition rewards. *B*: MuZero acts in the environment by sampling an action proportional to the action visit counts at the root. *C*: MuZero trains its three functions by aligning the unrolled model outputs and collected statistics in trajectories.

function f corresponds to the one network in AlphaZero. Given a hidden state $\mathbf{x}^k$, it produces a probability distribution p^k of actions and a value v^k associated to that hidden state. The three functions f, g, h are approximated jointly in a neural network with weights θ

$$\mathbf{x}_t^0 = h_\theta(o_1, \ldots, o_t, a_1, \ldots, a_{t-1}) \tag{2.2}$$

$$(\mathbf{x}^{k+1}, \hat{r}^{k+1}) = g_\theta(\mathbf{x}^k, a^k) \tag{2.3}$$

$$(v^k, \boldsymbol{p}^k) = f_\theta(\mathbf{x}^k) \tag{2.4}$$

The superscripts of $\mathbf{x}, a, v$ denote the depth of such values in the search tree, and depth 0 is at the search tree's root. Equivalently, the superscripts also mean the number of recurrent inferences (through the dynamics function g) the algorithm performs to obtain that value.

MuZero plans with a search method based on the MCTS framework (discussed in Section 2.3). Due to the lack of access to a perfect model, MuZero's MCTS differs from a standard one in numerous ways. The nodes are no longer perfect representations of the board states. Instead, each node is associated with a hidden state $\mathbf{x}$ as a learned representation of the board state. The transition is no longer made by the perfect model but by the dynamics function g. Moreover, since the dynamics function also predicts a reward, edges created through inferencing with the

dynamics function also contribute to the Q-value estimation.

To act in the environment, MuZero plans following the MCTS framework described in Section 2.3. At each timestep t, $\mathbf{x}_t^0$ is created using Equation 2.2. A variant of PUCT is used to select an action during the search:

$$\begin{aligned} I(s) &= \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left(E(s,a) + B(s,a) \right) \\ E(s,a) &= Q(s,a) \\ B(s,a) &\propto P(s,a) \frac{\sqrt{\sum_{b \in \mathcal{A}} N(s,b)}}{1 + N(s,a)} \left[c_1 + \log \left(\frac{\sum_{b \in \mathcal{A}} N(s,b) + c_2 + 1}{c_2} \right) \right] \quad . \end{aligned}$$

Here c_1 and c_2 are two constants that adjust the exploration bonus. The selected edge $(\mathbf{x}^k, a^k)$ at depth k is expanded using Equation 2.3 and evaluated using Equation 2.4. At the end of the simulation, the statistics of the nodes along the search path are updated. We denote the updated prior action probabilities $\mathbf{p}^*$, and the updated value estimation with v^*. Since the node transitions are approximated by the neural network, the search is performed over hypothetical trajectories without using a perfect model. Finally, the action a^0 of the most visited edge $(\mathbf{x}^0, a^0)$ of the root node is selected as the action to take in the environment.

Experience generated is stored in a replay buffer and processed into training targets. The three functions of the model are trained jointly using the loss function

$$\mathcal{L}_t(\theta) = \underbrace{\sum_{k=0}^{K} \mathcal{L}^p \left(p_{t+k}^*, p_t^k \right)}_{(1)} + \underbrace{\sum_{k=0}^{K} \mathcal{L}^v \left(z_{t+k}, v_t^* \right)}_{(2)} + \underbrace{\sum_{k=1}^{K} \mathcal{L}^{\mathrm{r}} \left(r_{t+k}, \hat{r}^k \right)}_{(3)} + \underbrace{c \|\theta\|^2}_{(4)} \tag{2.5}$$

where K is the rollout depth, (1) is the loss of the predicted prior move probabilities and move probabilities improved by the search, (2) is the loss of the predicted value and experienced N-step return, (3) is the loss of the predicted reward and the experienced reward, and finally (4) is the L_2 regularization.

2.7.1 MuZero Reanalyze

Schrittwieser et al. also developed **MuZero Reanalyze**, a sample efficient variant of MuZero [56]. This method generates training targets in addition to those generated through game play through re-executing search on old games using the latest parameters. **MuZero Unplugged** and **Efficient Zero** also use similar mechanisms to generate new data by updating search statistics of old data [57]. In Efficient Zero, experiments use a reanalyze ratio of 0.99, which means only 1% of the training data are generated through interacting with the environment, and the other

99% are generated by re-running search on old trajectories. In our project, we also implement a reanalyze worker to perform this task (see Sections 4.3.10 and 4.3.13).

2.8 Atari Game Playing

2.8.1 The Atari Learning Environment

The Atari 2600 gaming console was developed by *Atari, Inc.* and was released in 1977. Over 30 million copies of the console sold over its 15 years on the market [4]. The most popular game, PacMan, sold over 8 million copies and was the all-time best-selling video game back then. **Stella** is a multi-platform Atari 2600 emulator released under the GNU General Public License (GPL) [61]. Stella was ported to popular operating systems such as Linux, MacOS, and Windows, providing Atari 2600 experiences to users without physical copies of the equipment. In 2013, Bellemare et al. introduced the **Arcade Learning Environment (ALE)** and the library has been publicly available since [5]. ALE provides interfaces of over a hundred of Atari game environments using Stella as the backend. Each ALE environment has specifications on its visual representation, action space, and reward signals. ALE environments are suitable for controlled machine learning research, because data are well-represented and evaluation metrics are clearly defined. Moreover, ALE environments are diverse in their characteristics: while some environments require more mechanical mastery of the agent, others require more long-term planning. This makes solving multiple ALE environments using the same algorithm a good general game playing problem.

2.8.2 Deep Q-Networks

Mnih et al. pioneered the study of using deep neural networks to learn in ALE environments [44]. They developed the algorithm **Deep Q-Networks (DQN)** that learned to play seven of the Atari games and reached human-level performance. The DQN agent has a neural network that approximates the Q function, parametrized by weights θ, denoted Q_θ. Experiences are generated through interacting with the environment by taking the action that maximizes the immediate Q-value

$$\pi(a_t \mid (o_{t-L+1}, \ldots, o_t)) = \operatorname*{argmax}_a Q_\theta(o_{t-L+1}, \ldots, o_t, a)$$

where L is the length of history, and o_t is the "frame", a partial observation of the game state at timestep t (also see Section 4.3.4). Generated experience is stored in an experience replay buffer implemented as a FIFO queue. For each training step, a batch of uniformly sampled experience

is drawn from the experience replay, and the loss is computed using

$$\mathcal{L}(\theta) \propto \mathbb{E}_{\pi}\left[r + \gamma \max_{a'} Q_{\theta'}(s', a') - Q_{\theta}(s, a)\right] \quad .$$

The network parameters θ' are updated less frequently than θ.

2.8.3 Double Q Learning

Hasselt analyzed the overestimation problem of Q values in Q-learning and developed **double Q learning**, where a double Q update replaces the traditional Q update [22]. Double Q learning reduces the overestimation problem by introducing an additional Q estimator and updating the two estimators using each other:

$$Q^A(s, a) \leftarrow Q^A(s, a) + \alpha\left(r + \gamma Q^B\left(s', \operatorname*{argmax}_{a'} Q^A(s', a')\right) - Q^A(s, a)\right)$$
$$Q^B(s, a) \leftarrow Q^B(s, a) + \alpha\left(r + \gamma Q^A\left(s', \operatorname*{argmax}_{a'} Q^B(s', a')\right) - Q^B(s, a)\right)$$

where Q^A and Q^B are two different Q estimators updated alternately. Hasselt, Guez, and Silver applied double Q learning to DQN [23]. Similar to the double Q update above, a double Q update for neural networks is formulated as

$$\mathcal{L}(\theta^A) \propto \mathbb{E}_{\pi}\left[r + \gamma Q_{\theta^B}\left(s', \operatorname*{argmax}_{a'} Q_{\theta^A}(s', a')\right) - Q_{\theta^A}(s, a)\right]$$
$$\mathcal{L}(\theta^B) \propto \mathbb{E}_{\pi}\left[r + \gamma Q_{\theta^A}\left(s', \operatorname*{argmax}_{a'} Q_{\theta^B}(s', a')\right) - Q_{\theta^B}(s, a)\right] \quad .$$

Here Q_{θ^A} and Q_{θ^B} are two sets of parameters of the same neural network architecture.

2.8.4 Experience Replay

Schaul et al. studied the role of experience replay in DQN and developed the **prioritized experience replay** method [55]. In the original DQN, all samples were drawn from the experience replay uniformly. In prioritized experience replay however, samples are drawn according to a distribution based on their calculated priority

$$P(i) = \frac{p_i^{\alpha}}{\sum_k p_k^{\alpha}}$$

where $P(i)$ is the probability of the i-th sample being drawn, α is a constant, and p_i is the priority of the sample. Schaul et al. developed two approaches to compute priorities of samples. In **proportional sampling**, the priority p of sample i is calculated by

$$p_i = |\delta_i| + \epsilon$$

where δ_i is the temporal-difference error of the sample, and ϵ is a small constant to give all samples a non-zero probability to be drawn. In **rank-based sampling**, the same temporal differences are calculated, but the final priority is computed based on the rank of the error,

$$\begin{aligned} \text{score}(i) &= |\delta_i| + \epsilon \\ p_i &= \frac{1}{\text{rank}(\text{score}(i))} \end{aligned}$$

Horgan et al. followed up by implementing a distributed version of prioritized experience replay [28]. Kapturowski et al. investigated the challenges of using experience replay for RNN-based agents and developed **Recurrent Replay Distributed DQN** [32].

2.8.5 Network Architectures

Wang et al. studied an alternative neural network architecture for ALE learning [63]. A **Dueling Q-network** retains the input and output specifications of the Q-network used in DQN and structurally represents the learning of the advantage function $A(s, a)$ defined as

$$A(s, a) \doteq Q(s, a) - V(s)$$

The Q-network has three parts: θ, the shared trunk of the network; Λ, the advantage head; and Υ, the value head. The network approximates the value function internally through the shared trunk and the value head, denoted $V_{\theta,\Upsilon}$, and the advantage function, denoted $A_{\theta,\Lambda}$. The values computed by the two heads are combined to form the Q-value as follows:

$$Q_{\theta,\Upsilon,\Lambda}(s, a) = V_{\theta,\Upsilon}(s) + \left(A_{\theta,\Lambda}(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A_{\theta,\Lambda}(s, a') \right) .$$

Similar to DQN, the dueling Q-network is trained through fitting to empirical data generated by interacting with the environment. Experiments show that this architecture encourages the network to learn to differentiate between the values of states and the values of state-action pairs, and leads to better performance of the agent.

2.8.6 Scalar Transformation

Pohlen et al. introduced enhancements to achieve more stable training in Atari games [50]. We focus on discussing the **transformed Bellman Operator** since both MuZero and MooZi use it. For different Atari games, reward signals can vary drastically both in density and scale. This leads to high variance in training targets during training of the algorithms, causing algorithms to have difficulty converging. DQN clips the reward signal to a range of $[-1, 1]$ to reduce such variance [44]. However, this clipping discards the scale of rewards and consequently changes the set of optimal policies. The transformed Bellman Operator was developed to address this problem. The Q update of the new operator is as follows

$$Q(s,a) \leftarrow Q(s,a) + \alpha\phi\left(r + \gamma\phi^{-1}\left(\max_{a' \in \mathcal{A}} Q\left(s', a'\right)\right) - \phi^{-1}Q(s,a)\right)$$

where ϕ is an invertible transformation that contracts. One example of such a transformation is

$$\phi(x) \doteq \operatorname{sign}(x)\left(\sqrt{|x|+1} - 1\right) + \varepsilon x$$
$$\phi^{-1}(x) \doteq \operatorname{sign}(x)\left(\left(\frac{\sqrt{1 + 4\varepsilon(|x| + 1 + \varepsilon)} - 1}{2\varepsilon}\right)^2 - 1\right)$$

Both *MuZero* and *MooZi* use this specific ϕ definition to transform both values and rewards (see Section 4.3.5).

2.8.7 MinAtar

MinAtar, developed by Young and Tian, is an open-source project that offers RL environments inspired by ALE [67]. *MinAtar* offers five environments that pose similar challenges to ALE environments: learning representation from raw pixels, and learning behaviors that associate actions and delayed rewards. *MinAtar* environments are implemented in pure Python, have simpler environment dynamics, and are visually less rich than ALE environments, while retaining some flavor of the original games. This makes *MinAtar* environments perfect for university research.

2.8.8 Consistency Loss

One interesting characteristic of Atari-like games is that the environment frames are usually temporally consistent. For example, given the position of the player avatar for the last few frames, it is not difficult for a human to guess the position of the avatar in the next frame. To take

advantage of this property, one common approach is to enforce temporal consistency in the loss function. De Vries et al. visualized the latent space of a learned model of MuZero in a 3D space, in which a hidden state is a point in the space [12]. As MuZero applies recurrent inferences to a hidden state, the transitions can be traced as a 1-D path in the 3D space. The consistency loss they developed creates a smoother path in the 3D space and improves performance. Ye et al. developed a project-then-predict structure similar to a Siamese network to enforce consistency [66, 35].

2.9 Deep Reinforcement Learning Systems

Deep reinforcement learning systems involve irregular computation patterns and complicated hardware interactions between CPUs and AI accelerators. Designing such systems efficiently is a great challenge. Decisions the designer has to make include but are not limited to: (1) Where and how to generate experience? (2) Where and how to store generated experience? (3) Where to store the model and copies of it? (4) Where to carry out the gradient computation? (5) How to orchestrate processes for stable training? Here we briefly review several popular deep reinforcement learning system designs that utilize parallelization to achieve faster and more efficient training.

2.9.1 Mnih et al.'s Asynchronous Methods Framework

Mnih et al. developed asynchronous variants for four popular RL algorithms with a parallelization structure that uses actor-learner processes [43]. Each actor-learner process holds a local copy of the model, generates experience locally using the model, and accumulates gradients locally. Once in a while, all local gradients are aggregated to update the global model. Delaying and aggregating updates to neural network parameters reduces gradient variance among processes and achieves a more stable learning. Among the asynchronous algorithm variants, **Asynchronous Advantage Actor Critic (A3C)** had the best performance and achieved the state-of-the-art at the time of publication using only half the training time.

2.9.2 The IMPALA Architecture

Espeholt et al. developed **IMPALA**, a scalable distributed deep reinforcement learning agent [14]. IMPALA deploys two types of computation workers: *actor* and *learner*. A actor holds a copy of the neural network parameters and the environment. It performs model inferences locally to interact with its environments and generates experiences. Generated experiences are saved in local storage and subsequently pushed into the learner's local storage. The learner

holds the master copy of the neural network parameters. Once the learner receives enough experiences from the actors, it samples experiences from its local queue and performs batched forward pass and back-propagation steps using its model. Figure 2.3 shows two variants of this structure.

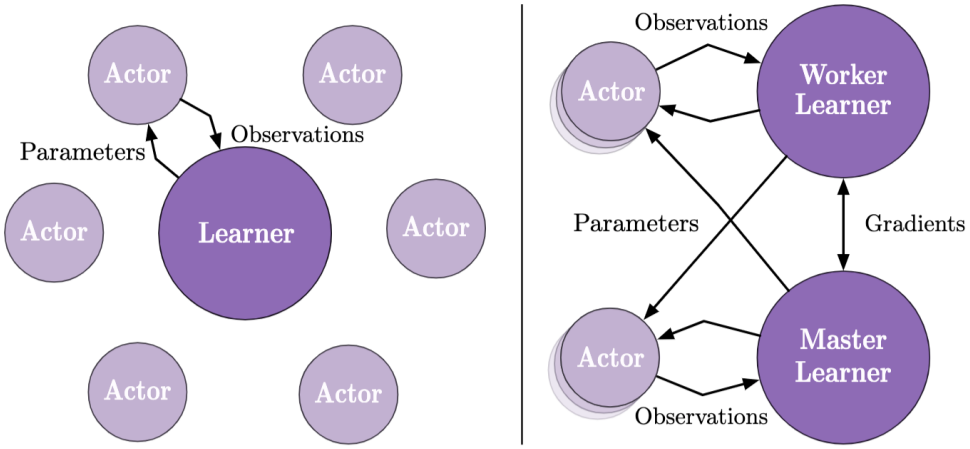

Figure 2.3: IMPALA Architecture, from Espeholt et al. [14].. *Left*: a single learner computes all gradients; *Right*: multiple worker learners compute gradients and one master learner collects and aggregates gradients.

2.9.3 The SEED Architecture

Espeholt et al. developed the **Scalable, Efficient Deep-RL (SEED)** architecture to effectively utilize accelerators using a centralized inference server [15]. Similar to IMPALA, SEED also uses two main types of workers: actors and learners. However, in SEED, actors do not hold copies of the model. Instead, SEED actors interact with their environments through querying the learner. The learner not only computes gradients and stores trajectories as in IMPALA, but also has a batching layer that batches actor queries and efficiently performs batched inference with the model. Since actors no longer need to pull neural network parameters from the learner, the IO overhead from serializing and messaging parameters is eliminated. Moreover, since the learner batches queries from all actors, the IO overhead from moving inputs and outputs to accelerators (GPUs or TPUs) is also reduced, increasing the overall inferencing throughput. One downside of the SEED architecture is that actors have to wait for a response from the learner to take an action, and thus have a higher latency for taking a step. Figure 2.4 illustrates a distributed SEED agent.

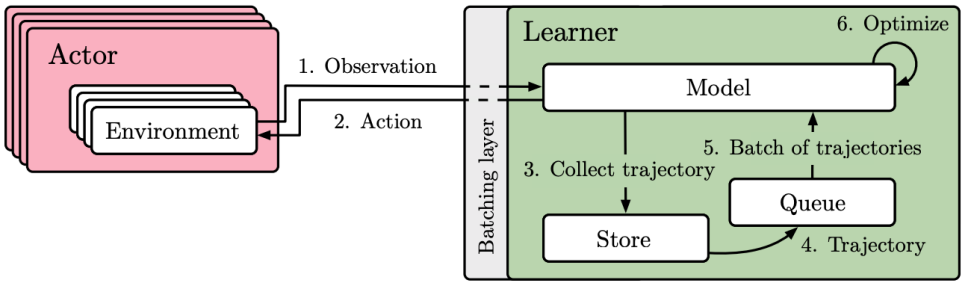

Figure 2.4: The SEED Architecture, from Espeholt et al. [15]. All inferences are computed on the learner and actors act through querying the learner.

2.9.4 The Acme Framework

Hoffman et al. developed the **Acme** research framework [27]. Acme is similar to IMPALA: processes that interact with the environment are actors, and processes that collect experience and update gradients are learners. Additionally, Acme has a *dataset* component, which is synonymous to the replay buffer used in DQN. This component uses **Reverb**, a high-performance library developed by Cassirer et al. for storing and sampling collected experiences [8]. Figure 2.5 illustrates a distributed asynchronous agent in Acme.

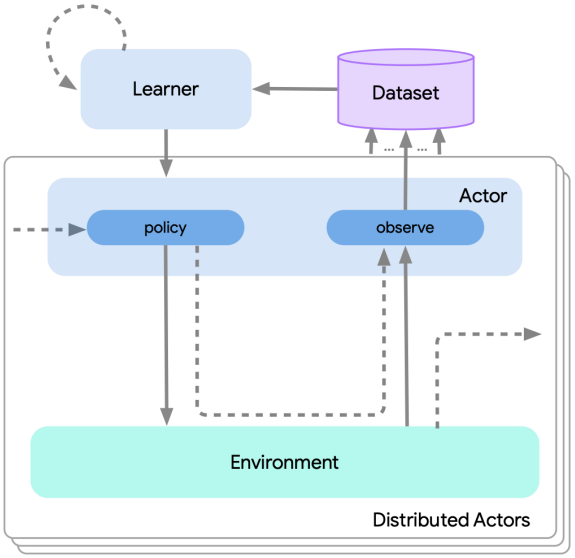

Figure 2.5: Example of a distributed asynchronous agent with Acme, from Hoffman et al. [27].

2.9.5 Ray and RLlib

Moritz et al. designed and implemented **Ray**, a framework for scalable distributed computing [47]. Ray enables both task-level and actor-level parallelization through a unified interface. **Ray Core** was designed with AI applications in mind and has powerful primitives for building distributed AI systems. For example, Ray uses shared memory to store inputs and outputs of tasks, allowing zero-copy data sharing among tasks. This is useful for DRL systems in which generated experiences are stored and sampled in separate processes. Liang et al. developed **RLlib**, an industrial-grade deep reinforcement learning library. RLlib is built on top of Ray Core and provides abstractions for a broad range of DRL systems. Figure 2.6 illustrates RLlib's abstraction layers. As of the writing of this thesis, RLlib implements 24 popular DRL algorithms using its abstractions. One major difference between RLlib agents and other DRL agents is that RLlib deploys a hierarchical control over the worker processes. Our project uses Ray Core to implement its worker processes and deploys a hierarchical control paradigm similar to RLlib (see Section 4).

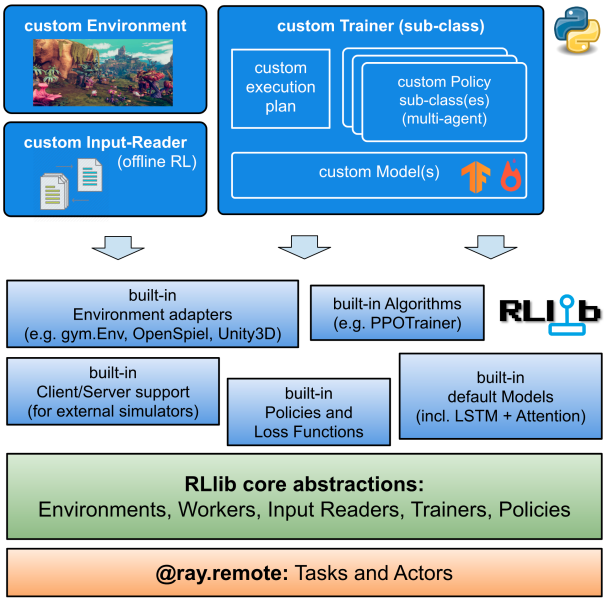

Figure 2.6: RLlib Abstraction Layers, from Liang et al. [39].

2.9.6 JAX and Podracer Architecture

Frostig, Johnson, and Leary designed **JAX**, a just-in-time (JIT) compiler that compiles computations expressed in Python code into high-performance accelerator code [16]. JAX is compatible with **Autograd**, so computation procedures expressed and compiled with JAX can

be automatically differentiated. JAX also supports control flow, allowing more sophisticated logic to be expressed while taking advantage of accelerators. Our project uses JAX for both neural networks and search. As a result, we are able to compile the entire policy in rollout workers, including history stacking, planning, and neural networks inferencing, into a single optimized program that can be hardware-accelerated. Hessel et al. designed two paradigms to efficiently use JAX for DRL systems [26]. In the **Anakin** architecture, the environment is implemented with JAX and the entire agent-environment loop is compiled using JAX and computed with accelerators. **Gymnax**, developed by Robert Tjarko Lange, provides environment implementations in native JAX, and is compatible with the Anakin architecture [51]. However, pure JAX implemented environments are not always feasible, especially when environments involve external services, such as Stella or Unity in their backend. Alternatively, in the **Sebulba** architecture [26], environments run on CPUs, but policies could be compiled and computed on accelerators. Generated experiences in both architectures can be used to compute gradients directly on accelerators. Figure 2.7 illustrates the Sebulba architecture.

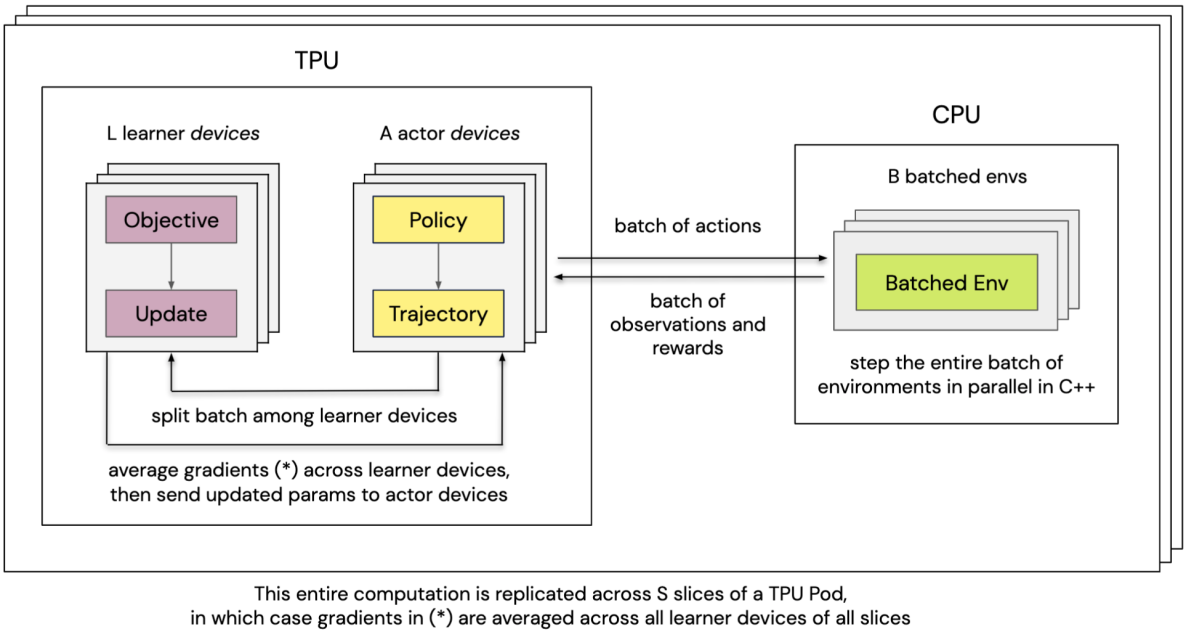

Figure 2.7: Sebulba architecture, from Hessel et al. [26]. The environments runs on CPUs. Inferences and gradient computations are compiled, optimized and executed on TPUs.

3 Problem Definition

3.1 Markov Decision Process and Agent-Environment Interface

A RL problem is usually represented as a **Markov Decision Process (MDP)**. MDP is defined as a four-tuple $(\mathcal{S}, \mathcal{A}, R, P)$. $\mathcal{S}$ is a set of states that forms the **state space**. $\mathcal{A}$ is a set of actions that forms the **action space**; $P(s'|s,a) \doteq \Pr[s_{t+1} = s' \mid s_t = s, a_t = a]$ is the **transition probability function**. $R(s,a,s')$ is the **reward function**. We use the **agent-environment interface** (as in Figure 3.1) to solve a problem formulated as an MDP. The MDP is represented as the **environment**. The decision maker that interacts with the environment is called the **agent**. At each time step t, the agent starts at state $s_t \in \mathcal{S}$, takes an action $a_t \in \mathcal{A}$, transitions to state $s_{t+1} \in \mathcal{S}$ based on the transition probability function $P(s_{t+1} \mid s_t, a_t)$ and receives a reward $R(s_t, a_t, s_{t+1})$. These interactions yield a sequence of actions, states, and rewards $s_0, a_0, r_1, s_1, a_1, r_2, \ldots$. We call this sequence a **trajectory**. When a trajectory ends at a terminal state s_T at time $t = T$, this sequence is completed and we called it an **episode**. Figure 3.1 illustrates the interaction between the agent and the environment.

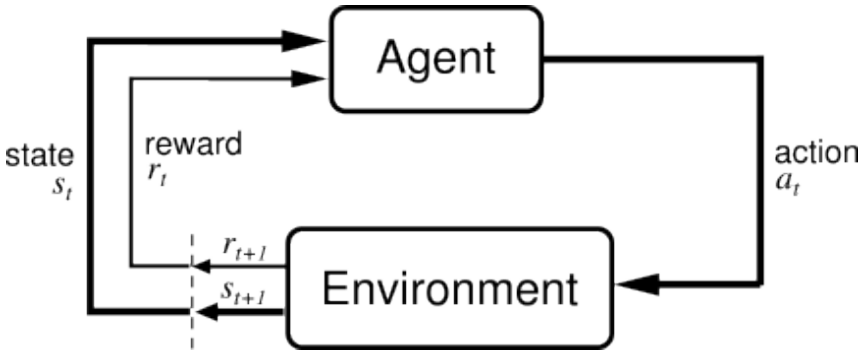

Figure 3.1: The Agent-Environment Interface, from Sutton and Barto [62].

3.2 Policies and Value Functions

At each state s, the agent takes an action based on a **policy** $\pi(a \mid s)$. This policy represents the conditional probability of the agent taking an action given a state, $\pi(a \mid s) \doteq Pr[a_t = a \mid s_t = s]$.

The objective of the agent is to maximize the expected discounted sum of rewards from the current state s_t following the policy π

$$\text{maximize} \quad \mathbb{E}_\pi \left[G_t \mid s_t = s \right], \quad \forall s \in \mathcal{S} \tag{3.1}$$

$$G_t \doteq \sum_{k=0}^{T} \gamma^k r_{t+k+1} \quad . \tag{3.2}$$

Here γ is the discount factor to favor short-term rewards. G is the discounted sum of rewards, or, equivalently, the discounted **return**. We represent the maximization target above as the **value function** V

$$V_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid s_t = s \right] \quad .$$

The value function indicates how good a state is when following the policy π. Similarly, we define the **state-action value function**

$$Q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid s_t = s, a_t = a \right]$$

that indicates how good a state and action pair is. We define the N-step return as a proxy of the true return, bootstrapped from a value function of a future state

$$G_t^N \doteq \sum_{k=0}^{N-1} \gamma^k r_{t+k+1} + \gamma^N V(s_{t+N}) \quad .$$

3.3 Partially Observable Markov Decision Process

A generalization of MDP is a Partially Observable Markov Decision Process (POMDP) [3]. In addition to the four-tuples of MDP, POMDP also defines Ω, a set of observations o that forms the **observation space**; and $O(o \mid s, a) \doteq \Pr[o_t \mid s_t = s, a_t = a]$, the conditional probability of observing o_t given the last taken action a_t and state s_t. In an agent-environment interface with a POMDP represented environment, the true environment state s_t at each timestep is hidden from the agent and the agent only receives a partial observation o_t.

3.4 Game Playing

We can represent board games and video games as POMDPs and solve them by developing an agent. Many board games, such as Go and chess, are fully observable and we treat them

as the special case where $o_t = s_t$. Video games, however, are partially observable since frames rendered on the screen do not contain all information of the program's running memory. In Go, chess, and Shogi, the only reward is given from the last timestep based on the game result, and the reward is one if $\{-1, 0, +1\}$. In Atari games, environments produce intermediate rewards based on game progression, and the scale and density of the rewards varies from game to game. In all cases, the goal of the agent is to maximize the expected return as described in Equation 3.1.

4 Method

4.1 Design Philosophy

4.1.1 Use of Pure Functions

One of the most notable differences of the *MooZi* implementation compared to other implementations is the use of pure functions. In *MooZi*, we separate the storage of data and the handling of data whenever possible, especially for the parts with heavy computations. We use **JAX** and **Haiku** [25, 31] to implement neural network related modules (see Section 2.9.6). These libraries separate the **specification** and the **parameters** of a neural network. The **specification** of a neural network is a pure function that is internally represented by a fixed computation graph. The **parameters** of a neural network includes all learned network weights that can be used with the specification to perform a forward pass. For example, consider a simple neural network with a single dense layer that computes

$$\mathbf{y} = \tanh\left(\mathbf{A}\mathbf{x} + \mathbf{b}\right)$$

Here $\mathbf{x}$ is the input vector of shape $(n, 1)$, $\mathbf{y}$ is the output vector of shape $(m, 1)$, $\mathbf{A}$ are the learned weights of shape (m, n), and b is the learned bias of shape $(m, 1)$. The parameters are all the weights in $\mathbf{A}$ and b. We demonstrate how to build this simple network using JAX and Haiku in Algorithm 1. We visualize the computation graph in Figure 4.1. Using these pure functions separates the *algorithm* of the agent and the *state* of the agent both conceptually and in implementation. The *algorithm* can be abstracted into a computation graph that can be compiled and optimized using a specialized compiler, such as XLA [1], for hardware acceleration (see Section 2.9.6). The *state* part of the agent can be efficiently handled by tools specialized in data manipulations and transfer such as Ray (see Section 2.9.5). This way, our system efficiently performs inferences on accelerators (such as GPUs and TPUs) and transfers data on CPUs.

4.1.2 Training Efficiency

In Section 2.9 we reviewed common DRL systems for which their developers stated training efficiency as the highest priority in their system design. We also designed our system to be

```python
import haiku as hk
import jax
import jax.numpy as jnp

m = 3
n = 2

# specify the computations to be performed
class Model(hk.Module):
  def __call__(self, x):
    A = hk.get_parameter('A', shape=(m, n), init=jnp.zeros)
    b = hk.get_parameter('b', shape=(m, 1), init=jnp.zeros)

    return jax.nn.tanh(A @ x + b)

# haiku transforms the object-oriented model into a functional one
model = hk.without_apply_rng(hk.transform(lambda x: Model()(x)))

# construct a concrete input
x = jnp.ones((n, 1))

# initialize the parameters
params = model.init(jax.random.PRNGKey(0), x)

# perform the forward pass
y = model.apply(params, x)
```

Algorithm 1: A simple dense layer implemented in JAX and Haiku. The *model* in the code is the specification of the neural network. The *params* in the code is the parameters of the neural network. Only *params* contains concrete data.

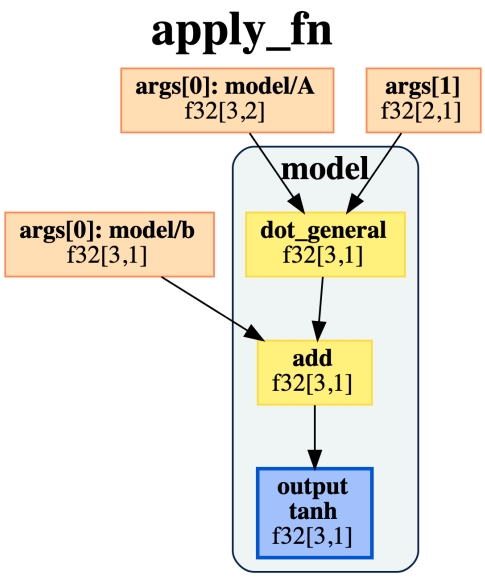

Figure 4.1: Computation graph of the simple dense layer in Algorithm 1. This computation graph show no concrete data, but the data types, shapes, and operators of the layer (*f32* stands for *single-precision float*). To complete a forward pass, we need both concrete neural network parameters *args[0]* ($\mathbf{A}, \mathbf{b}$) and a concrete input value *args[1]* ($\mathbf{x}$).

efficient and scalable. Here we describe key features of our system which improve its efficiency. The first one is system parallelization. The computation throughput of a single process is simply not enough for DRL systems. In the published results of MuZero by [56], the agent generated over 20 billion environment frames for training. For a non-parallelized system, consider Gymnax's efficient *MinAtar* implementation where each environment step takes about 1 millisecond [51]. A single process would take more than 200 days just to step the environment, without considering the model training cost. As a result, we have to build a distributed system to increase total throughput through parallelism.

The second key feature of our system is the environment transition speed. In Atari games, especially Atari games in ALE (2.8.1), taking one environment step invokes a full-fledged Atari emulator in the backend, which is much more time consuming than neural network inference. Board games, especially those implemented in high performance languages, are much faster. We use *MinAtar* (reviewed in Section 2.8.7) for simpler variants of Atari games, and *OpenSpiel* for efficient implementations of board games to reduce the time spent on environment transitions [38, 67].

The third key feature of *MooZi* is efficient acting with interdependent inferences. DRL systems such as IMPALA assume that the policy output can be computed by a single forward pass of a neural network. However, MuZero's policy requires multiple inferences per action

taken and these inferences are dependent of each other based on the tree search. Our system, utilizing JAX and MCTX, handles acting with multiple inferences per action efficiently by implementing the entire process of acting in native JAX.

4.1.3 Understanding the Method is Important

Machine learning algorithms, especially those involving neural networks, have interpretability issues and sometimes can only be used as "black boxes" [40]. We believe that having a system that we can understand is much more useful for future research than having a system that "just works". Therefore, our project studies the behavior of the system through extensive logging and visualization utilities. We will show we use these tools to understand the learned model in Section 4.5 and Chapter 5.

4.2 Architecture Overview

In *MooZi*, we use the **Ray** library designed by Moritz et al. [47] for orchestrating distributed processes. We also adopt the terminology used by Ray. In a distributed system with **centralized control**, a single **driver** process is responsible for operating all other processes. Other processes are either **tasks** or **actors**. **Tasks** are stateless functions that take inputs and return outputs. **Actors** are stateful objects that can perform multiple tasks. In the RL literature, **actor** is also a commonly used term for describing the process that holds a copy of the network weights and interacts with an environment [15, 14]. Even though *MooZi* does not adopt this concept of a RL actor, we will use the terms **Ray task** and **Ray actor** to avoid confusion. In contrast to distributed systems with **distributed control**, ray tasks and actors are reactive and do not have busy loops. The driver controls when a ray task or actor is activated, what data is used as input, and where the output goes. The driver orchestrates the data and control flow of the entire system. Ray tasks and actors merely respond to instructions, process input and return output on command. We illustrate *MooZi*'s architecture in Figure 4.2.

4.3 The *MooZi* System Components

4.3.1 Environment Bridges

Environment bridges unify environments which are defined in different libraries to a shared interface used by *MooZi*. In software engineering terms, environment bridges follow the **bridge design pattern** [6]. In our project we implement environment bridges for three types of

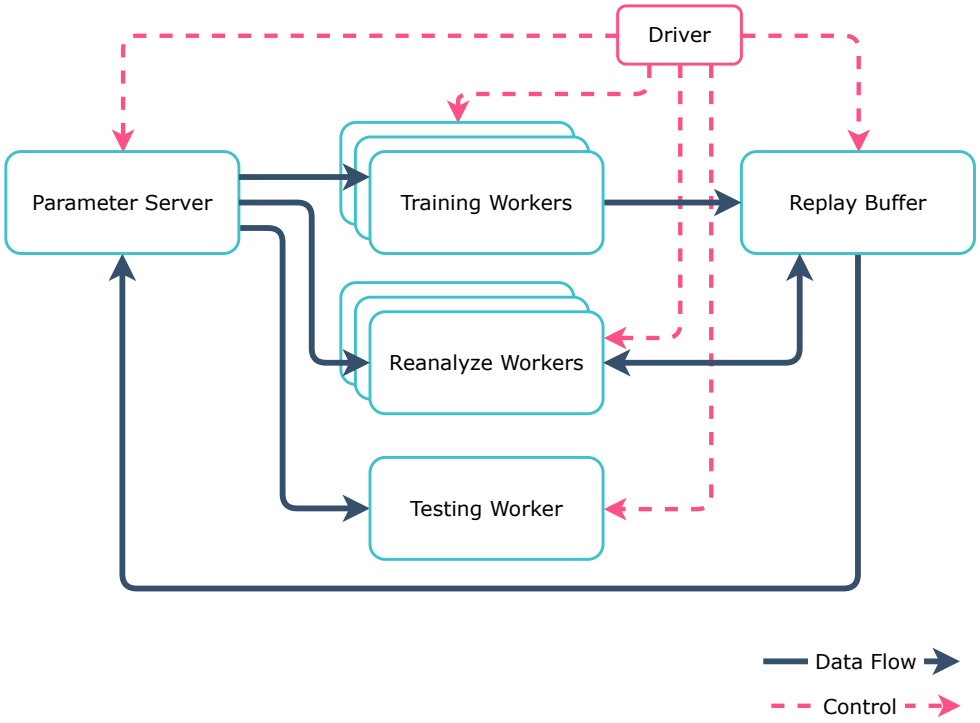

Figure 4.2: The *MooZi* Architecture. The *Driver* is the entry point of the program (details in Section 4.4). The *Parameter Server* stores the latest copy of the network weights and performs batched updates to them (see Section 4.3.15). The *Replay Buffer* stores generated trajectories and processes these trajectories into training targets (see Section 4.3.14). A *Training Worker* is a Ray actor responsible for generating experiences by interacting with the environment (see Section 4.3.11). A *Testing Worker* is a Ray actor responsible for evaluating the system by interacting with the environment (see Section 4.3.12). A *Reanalyze Worker* is a Ray actor that updates search statistics for history trajectories (see Section 4.3.13).

environments that are commonly used in RL research: *OpenAI Gym*, *OpenSpiel*, and *MinAtar* [7, 38, 67].

For all these wrapped environments, our bridges produce a flat structure for each timestep that has the following inputs and outputs:

- Inputs:

  b_t^{last}: A boolean indicating the episode end.

  a_t: An integer encoding of the action taken.

- Outputs:

  o_t: An 3-dimensional array representing the observation of the current timestep as an image in the shape (H, W, C_e). H is the height, W is the width, and C_e is the number of channels.

  b_t^{first}: A boolean indicating the episode start.

  b_t^{last}: A boolean indicating the episode end.

  b^{player_t}: A boolean indicating the current player. A boolean is sufficient because *MooZi* currently only supports environments with at most two players.

  r_t: A float indicating the reward of taking the given action.

  $m_t^{A^a}$: A bit mask indicating legal action indices. Valid action indices have a 1-bit and invalid actions indices have a 0-bit for non-terminal states (also see Section 4.3.3).

All environments are generalized to continuous tasks by passing an additional input b_t^{last} to the environment stepping argument. For an episodic task, the environment is reset internally when b_t^{last} is *True*. The policy still executes for the last environment step, but the resulting action is discarded. For a continuous task, the environment always steps with the latest action and never sets b^{last} to *true*. Algorithm 2 demonstrates the unified main loop interface. We also implement a mock environment [45] using the same interface. A mock environment is initialized with a **trajectory sample** $\mathcal{T}$, and simulates the environment by outputting step samples one at a time. An agent can interact with this mock environment as if it were a real environment. However, the actions taken by the agent do not affect the state transitions since they are predetermined by the given trajectory. This mock environment is used by the reanalyze workers in Section 4.3.13.

4.3.2 Vectorized Environment

A vectorized environment supervisor stacks multiple individual environments to form a single vectorized environment. The environment takes inputs and produces outputs similar to an

```python
# interact with the environment with a policy indefinitely
def main_loop(env, policy):
    action = 0
    reset = True
    while True:
        result = step(env, action, reset)
        action = policy(result.observation)
        reset = result.is_last


def step(env, action, reset):
    if env.type == "episodic":
        if reset:
            return env.reset()
        else:
            return env.step(action)
    elif env.type == "continuous":
        return env.step(action)
```

Algorithm 2: Unified Main Loop Interface. Both *episodic* environments and *continuous* environments are handled with the same main loop.

individual environment, but with an additional batch dimension. For example, an individual environment produces a single frame of shape (H, W, C), while the vectorized environment produces a batched frame of shape (B, H, W, C). Previously scalar outputs such as reward are also stacked into vectors of size B. Since environment bridges generalize episodic tasks as continuous tasks, we do not need special handling for the first and the last timesteps in the vectorized environment and its main loop looks exactly like that in Algorithm 2. Using vectorized environments increases the communication bandwidth between the environment and the agent and facilitates designing a vectorized agent that processes batched inputs and returns batched actions in one call.

The mock environment described in Section 4.3.1 is less trivial to vectorize. Each mock environment has to be initialized with a trajectory sample. To vectorize B mock environments, at least B trajectories have to be tracked at the same time. These B trajectories usually have different length and therefore terminate at different timesteps. Once one of the mocked trajectories reaches its termination, another trajectory has to fill the slot. We create a trajectory buffer to address this problem. When a new trajectory is needed by one of the mocked environments, the buffer replenishes it, so the vectorized mocked environment can process batched interactions at full capacity like a regular vectorized environment until the trajectory buffer runs out of trajectories. An external process has to refill the buffer once in a while. The driver pulls the latest trajectories from the replay buffer and supplies the mock environment's trajectory buffer (also see Section 4.3.13).

4.3.3 Action Space Augmentation

We augment the action space by adding a dummy action a^{dummy} indexed at 0. This dummy action is used to construct history observations when the horizon extends beyond the current timestep. For example, if the history horizon is 3, we need the last three frames and actions to construct the input observation to the policy. However, if the current timestep is 0, the agent hasn't taken any actions yet. We use zeroed frames with the same shape as history frames, and the dummy action as history actions. Moreover, *MooZi*'s planner (see Section 4.3.6) does not have access to a perfect model, and it does not know when a node represents a terminal state. Node expansions do not stop at terminal states and the tree search can simulate multiple steps beyond the end. Search performed in these invalid subtrees not only wastes precious search budget, but also back-propagates value and reward estimates that are not learned from generated experience. We address this issue by letting the model learn a policy that always takes the dummy action beyond a terminal state. This learned dummy action acts as a switch that, once taken, treats all nodes in its subtree as absorbing states and edges that have zero value and reward respectively. This discourages the planner from searching in invalid regions and improves search performance for near-end game states. To formally differentiate these two types of action spaces, we denote the original environment action space $\mathcal{A}^e$ and the augmented action space $\mathcal{A}^a$, with

$$
\begin{aligned}
&\mathcal{A}^a = \mathcal{A}^e \cup \{a^{\text{dummy}}\} \\
&a_i = a^{\text{dummy}} \quad \forall i < 0 && \text{(before the first timestep)} \\
&a_i = a^{\text{dummy}} \quad \forall i \geq T && \text{(after the last timestep)}
\end{aligned}
$$

Notice that the environment terminates at timestep T so the last effective action taken by the agent is a_{T-1}.

4.3.4 History Stacking

In fully observable environments, the state s_t at timestep t observed by the agent entails sufficient information about the future state distribution. However, for partially observable environments, this does not hold. The optimal policy might not be representable by a policy $\pi(a \mid o_t)$ that only takes into account the most recent partial observation o_t. Most Atari games are such partially observable environments. In DQN, Mnih et al. alleviated this problem by augmenting the inputs of the policy network from a single frame observation to a stacked history of four frames so that the policy network had a signature of $\pi(a \mid o_{t-3}, o_{t-2}, o_{t-1}, o_t)$

(Section 2.8.2, [44]). AlphaZero and MuZero use not only a stacked history of environment frames, but also a history of past actions. *MooZi* uses the last L environment frames and taken actions, so the signature of the learned model through the policy head of the prediction function is $\mathbf{p} = f(a \mid o_{t-L+1}, \ldots, o_t, a_{t-L}, \ldots, a_{t-1})$. The greater L is, the better the stacked observation represents a full state. In a deterministic environment with a fixed starting state, the stacked history represents a full environment state when $L = \infty$. On the other hand, $L = 1$ is sufficient for fully-observable perfect information environments. We represent all partial observations as images with height H, width W and environment channels C_e. In ALE, the height and width are the resolution of the screen frame, and the channels are RGB values. In MinAtar, the height and the width are also the resolution of the screen frame but the channels are layers of game entities such as enemies or bullets. In board games, the height and the width are the width of the board, and the channels are layers of game entities such as white pawn, black knight, and empty tiles.

The exact process of creating the model input by stacking history frames and actions is as follows:

1. Prepare L saved environment frames of shape (L, H, W, C_e).
2. Stack the L dimension with the environment channels dimension C_e, resulting in shape $(H, W, L * C_e)$
3. Prepare saved L past actions of shape (L), encoded as integers.
4. One-hot encode the actions as shape $(L, |\mathcal{A}^a|)$.
5. Normalize the action planes by the number of actions $|\mathcal{A}^a|$. The shape remains the same.
6. Stack the L axis with the action axis, now shape $(L * |\mathcal{A}^a|)$.
7. Tile action planes $(L * |\mathcal{A}^a|)$ along the H and W dimensions, now shape $(H, W, L * |\mathcal{A}^a|)$
8. Stack the environment planes and action planes, now shape $(H, W, L * (C_e + |\mathcal{A}^a|))$
9. The history is now represented as an image with a height of H, width of W, and $L * (C_e + |\mathcal{A}^a|)$ channels

To process batched inputs from vectorized environments described in Section 4.3.2, all operations above are performed with an additional batch dimension B, yielding the final output with the shape $(B, H, W, L * (C_e + |\mathcal{A}^a|))$. We denote the channels of the final stacked history $C_h = L * (C_e + |\mathcal{A}^a|)$, where the subscript h means the channel dimension for the representation function h. Figure 4.3 illustrates this process with an example. We process history as images this way to utilize neural network architectures that were originally developed to work with images such as ResNet [24].

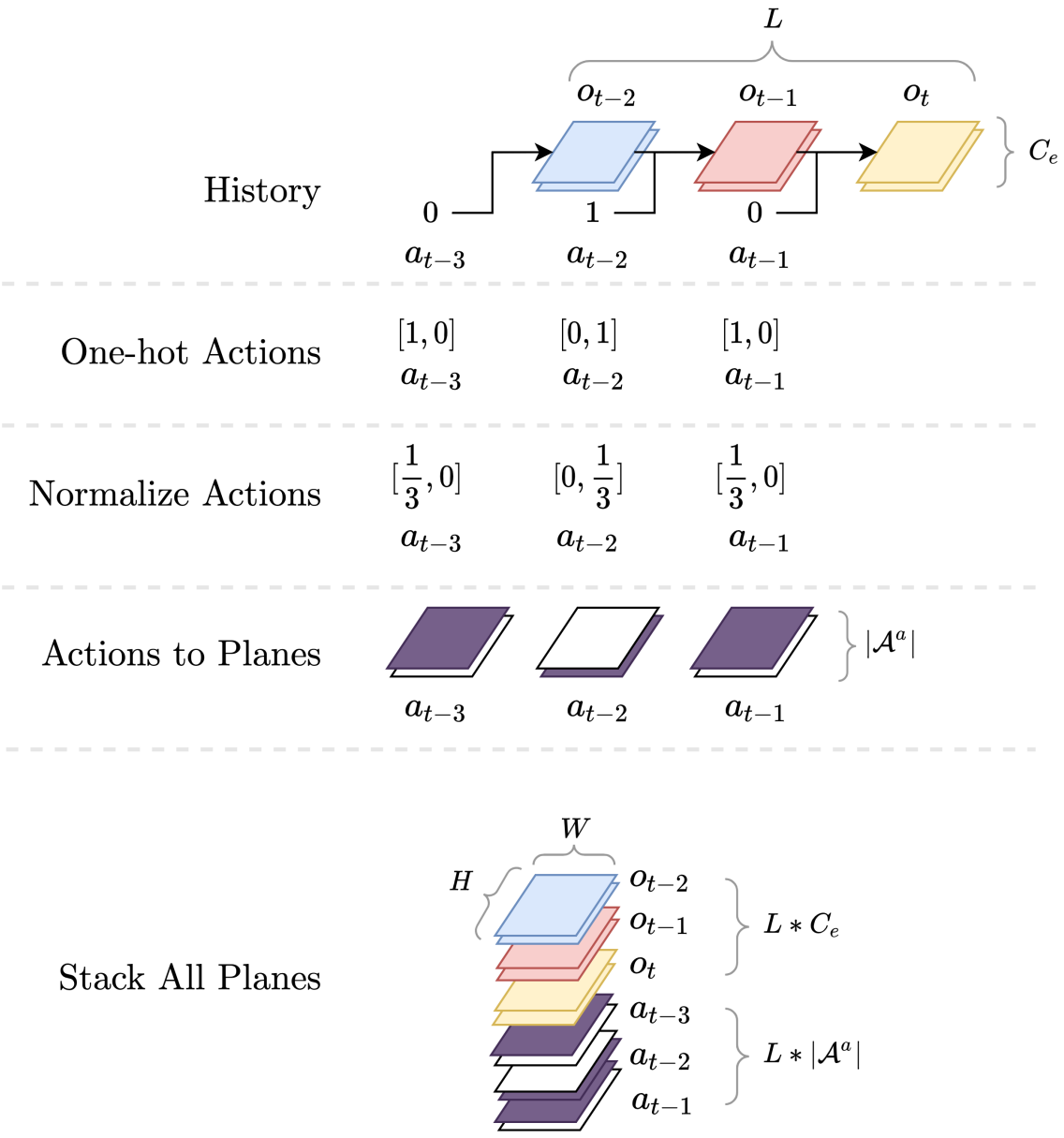

Figure 4.3: An example of history stacking. *History*: Partial observations and actions from the last 3 timesteps ($L = 3$). Actions are integers and observations are images with 2 channels each. *One-hot Actions*: One-hot encodes L history actions into vectors. *Normalize Actions*: Divide the resulting one-hot encoded actions by the size of the action space. *Actions to Planes*: One-hot encodes actions into feature planes that have the same resolution (i.e., same width and height) as the observations, $|\mathcal{A}^a| = 2$. *Stack Planes*: Stack all planes together, creating an image with 12 channels and the same resolution as the observations.

4.3.5 The *MooZi* Neural Network

MooZi uses the JAX and Haiku libraries to build the neural network [25, 16, 31]. We also consulted other open-source projects that use neural networks to play games [13, 66, 64]. *MooZi* implements two neural network variants, one is based on multilayer-perceptrons (contribution of Jiuqi Wang) and the other one is based on residual blocks [24]. These two implementations share the same interface and can be used interchangeably.

Similar to MuZero described in Section 2.7, the model has a representation function h, a dynamics function g, and a prediction function f. Additionally, *MooZi* has a **projection function** ϱ for training with the self-consistency loss (see Section 4.3.8). The learned model is used to construct the root node of a tree search using the representation function h and the prediction function f. We call this process the **initial inference**. The learned model is also used to create tree edges and child nodes using the dynamics function g and the prediction function f. We call this process the **recurrent inference**. The initial and recurrent inference terminology is also used in MuZero's public pseudo-code and MCTX's source code [59, 30]. For convenience, the initial and the recurrent inference both produce a tuple $(\mathbf{x}, v, \hat{r}, \mathbf{p})$, where $\mathbf{x}$ is the hidden state, v is the value prediction, $\hat{r}$ is the reward prediction, and $\mathbf{p}$ is the policy prediction. The reward prediction $\hat{r}$ is set to 0 for the initial inference. During training, v and $\hat{r}$ are logits with size $|Z|$. During acting, the logits of v and $\hat{r}$ are first converted to softmax distributions, then converted to scalars using the transformation Φ (see Section 2.8.6).

We apply the invertible transformation ϕ described in Section 2.8.6 to both the scalar reward targets and scalar value targets to create categorical representations with the same support size. Scalars are first transformed using ϕ, then converted to a linear combination of the nearest two integers in the support. For example, for scalar $\phi(x) = 1.3$, the nearest two integers in the support are 1 and 2, and the linear combination is $\phi(x) = 1 * 0.7 + 2 * 0.3$, which means that the target of this scalar is 0.7 for category 1, and 0.3 for category 2. Operator Φ applies ϕ then categorizes the resulting value into a support Z. Using the same example $\phi(x) = 1.3$, assume the support is $Z = [-2, -1, 0, 1, 2]$, $|Z| = 5$, then $\Phi(x) = [0, 0, 0, 0.7, 0.3]$, and $\Phi(x) \cdot Z = \phi(x) = 1.3$. For training, the value head and the reward head first produce estimations as logits of size $|Z|$. These logits are aligned with the scalar targets to produce categorization loss as described in Section 4.3.8. For acting, the neural network additionally applies the softmax function to the logits to generate a distribution over the support. The linear combination of the distribution and their corresponding integer values are computed and fed through the inverse transformation ϕ^{-1} to produce scalar values. This means from the perspective of the planner (see Section 4.3.6), the scalar estimations made by the model are in the same shape and scale as those produced by the environment.

4.3.6 Planner

The planner component $\mathcal{P}$ takes a stacked history as its input (see Section 4.3.4), performs a search, collects search statistics, and outputs a action and search statistics

$$a_t, v_t^*, \mathbf{p}_t^* = \mathcal{P}(o_{t-L+1}, \ldots, o_t, a_{t-L}, \ldots, a_{t-1}) \ .$$

Here v_t^* is the search-updated value estimate of the root, $\mathbf{p}_t^*$ is the search-updated action visits at the root, and a_t is the action to take. Training workers (see Section 4.3.11) use the full planner. Testing workers only use the action output from the planner. The planner used this way is equivalent to a policy π. Reanalyze workers only use the output statistics from the planner. The planner uses the MuZero variant of MCTS described in Section 2.3 and Section 2.7, implemented with the help from **MCTX** by Ivo Danihelka [30, 31, 11]. The planner uses a boolean flag b^{player} from the environment bridge output (described in Section 4.3.1) to indicate the current player. In a single-player environment, this boolean flag does not affect the search. In a two-player environment, the planner performs a *logical NOT* that flips the player along with the dynamics function g for each node expansion. As described in Section 4.3.5, the model is trained from the perspective of the first player. The planner re-orients the value and reward predictions $v, \hat{r}$ of each node based on the player of that node. *MooZi* assumes the two-player game is zero-sum, so this re-orientation is a simple sign flip. Once the values are oriented from the perspective of the current player of a node, the planner searches in a Negamax fashion [54] by selecting nodes that maximizes the negation of Q-values of the edges. We use a discount factor $\gamma = -1$ as an implementation trick to achieve this Negamax search in the planner. The planner also applies the legal actions mask m^{A_a} on root prior policy so only legal actions will be chosen. At the last timestep T of any environment, the only legal action will be the dummy action and the planner will be forced to take it.

4.3.7 Training Target Generation

At each timestep t, the environment provides a tuple of data as described in Section 4.3.1. The agent interacts with the environment by performing a tree search and taking action a_t. The search statistics of the tree search are also saved, including the updated value estimate of the root action $\hat{v}_t$, and the updated action probability distribution $\hat{p}_t$. This completes one **step sample** $\mathcal{T}_t$ for timestep t, which is a tuple $(o_t, a_t, b_t^{\text{first}}, b_t^{\text{last}}, b^{\text{player}}, r_t, m_t^{A_a}, \hat{v}_t, \hat{p}_t)$. Once an episode concludes ($b_T^{\text{last}} = 1$), all recorded step samples are gathered and stacked together. This yields a final trajectory sample $\mathcal{T}$ that has a similar shape to a step sample but with an extra batch dimension of size T. For example, T observations are stacked from shape (H, W, C_e) to shape (T, H, W, C_e). The training workers described in Section 4.3.11 generate

trajectories this way. The reanalyze workers generate trajectories with the same signature, but through statistics updates using a vectorized mock environment (see Section 4.3.10 and Section 4.3.2).

Each trajectory sample with T step samples is processed into T training targets. We define K as the number of unrolled steps for training. The larger the K, the deeper the search tree we train the model to align with real trajectories. Training targets are computed with the minimum information necessary for the loss function computation (see Section 4.3.8) so that the precomputed training targets take up the least memory. We create a training target at timestep i as follows:

- Observations $o_{i-L+1}, \ldots, o_{i+1}$ where L is the history stacking size. The first L observations are used to create policy inputs in Section 4.3.4, and the pair of observations o_i, o_{i+i} is used to compute the self-consistency loss described in Section 4.3.8.
- Actions $a_{i-L}, \ldots, a_{i+K-1}$. Similarly, The first L actions are used for policy input and the pair of actions at (a_{i-1}, a_i) are used for self-consistency loss. The actions $a_i, \ldots, a_{i+K-1}$ are used to unroll the model during the training for K steps.
- Rewards $r_{i+1}, \ldots, r_{i+K}$ are the targets of the reward head of the dynamics function.
- Action probabilities $\mathbf{p}_i^*, \ldots, \mathbf{p}_{i+K}^*$ from the statistics of $K+1$ searches.
- Root values $v_i^*, \ldots, v_{i+K}^*$, similarly, from the statistics of $K+1$ searches.
- N-step returns $G_i^N, \ldots, G_{i+K}^N$. Each N-step return is computed based on the formula

$$G_t^N = \sum_{i=0}^{N-1} \gamma^i r_{t+i+1} + \gamma^N v_{t+N}^* \ .$$

- The current player b^{player}.
- The importance sampling ratio $\rho = 1$. This is a placeholder value for future override based on replay buffer sampling weights (see Section 4.3.14).

In both single-player and two-player environments, we train the neural network from the perspective of the first player. This means *MooZi* only supports zero-sum two-player environments where the value of the second player can be obtained by taking the negative of the value of the first player.

4.3.8 Loss Computation

Our loss function is similar to the one in MuZero (see Section 2.7), but with additional self-consistency loss term, terminal action loss, and value loss coefficient:

$$\begin{aligned}\mathcal{L}_t(\theta) = \Bigg[& \underbrace{\mathcal{L}^p(\mathbf{p}_t^*, \mathbf{p}_t^0) + \frac{1}{K}\sum_{k=1}^{K} \mathcal{L}^p\left(\mathbf{p}_{t+k}^*, \mathbf{p}_t^k\right)}_{(1)} \\ & + \underbrace{c^v\left(\mathcal{L}^v(G_t^N, v_t^*) + \frac{1}{K}\sum_{k=1}^{K} \mathcal{L}^v\left(G_{t+k}^N, v_{t+k}^*\right)\right)}_{(2)} \\ & + \underbrace{\sum_{k=1}^{K} \mathcal{L}^r\left(\hat{r}_t^k, r_{t+k}\right)}_{(3)} + \underbrace{c^s \mathcal{L}_t^s(\mathbf{x}_t^1, \mathbf{x}_{t+1}^0)}_{(4)} \\ & + \underbrace{c^{L_2}\|\theta\|^2}_{(5)} \Bigg] \cdot \rho\end{aligned}$$

To compute these terms used in the loss function, we use the history observations $o_{t-L+1}, \ldots, o_t$ and history actions $a_{t-L}, \ldots, a_{t-1}$ to reconstruct the stacked frames as the input of the initial inference (see Section 4.3.4). We apply the initial inference to obtain $\mathbf{p}_t^0, v_t^0, \mathbf{x}_t^0$. We apply K consecutive recurrent inferences using actions $a_t, \ldots, a_{t+K-1}$ to obtain $\mathbf{p}_t^1, \ldots, \mathbf{p}_t^K, v_t^1, \ldots, v_t^K, \mathbf{x}_t^1, \ldots, \mathbf{x}_t^K$. The policy loss (1) is the standard categorization loss using cross-entropy

$$\mathcal{L}^p(\mathbf{p}, \mathbf{q}) = -\sum_{p \in \mathbf{p}, q \in \mathbf{q}} p \log q \ .$$

The policy targets $\mathbf{p}_{t+i}^* (i = 0, 1, \ldots, K)$ are action visit counts at the root of $K+1$ searches performed in the game (see Section 4.3.7). To compute the value loss (2) and the reward loss (3), we apply the scalar transformation Φ (see Section 2.8.6) that converts scalar values to categorizations, and use the same cross-entropy categorization loss

$$\mathcal{L}^v(p, q) = \mathcal{L}^r(p, q) = -\sum_{p \in \Phi(p), q \in \Phi(q)} p \log q$$

MooZi also trains with a self-consistency loss similar to that described by Ye et al. and de Vries et al. [66, 12]. To compute the self-consistency loss (4), we reconstruct the initial inference for the next timestep $o_{t-L+2}, \ldots, o_{t+1}, a_{t-L+1}, \ldots, a_t$, and compute the cosine distance between the projected one-step hidden state $\varrho(\mathbf{x}_t^1)$ of timestep t and the initial hidden state $\mathbf{x}_{t+1}^0$ of the next timestep $t+1$. Formally,

$$\text{cosine distance}\,(\mathbf{a}, \mathbf{b}) = 1 - \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

$$\mathcal{L}^s(\mathbf{x_t^1}, \mathbf{x}_{t+1}^0) = 1 - \frac{\varrho(\mathbf{x}_t^1) \cdot \mathbf{x}_{t+1}^0}{\|\varrho(\mathbf{x}_t^1)\|\|\mathbf{x}_{t+1}^0|\|}$$

Figure 4.4 illustrates the intuition behind this loss. Part (5) of the loss function is a standard

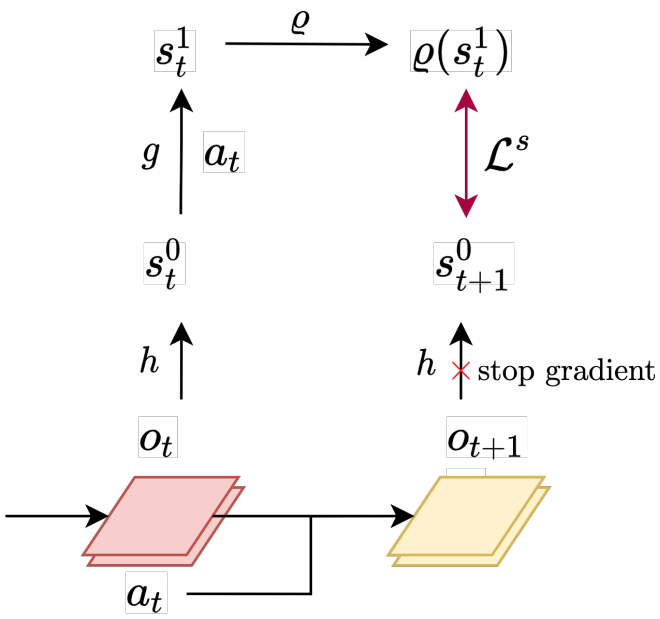

Figure 4.4: Self-consistency Loss Computation. The hidden state $\mathbf{x}_t^1$ after projection should be similar to the hidden state $\mathbf{x}_{t+1}^0$. We assume the next timestep has more information, so we stop gradient from $\mathbf{x^0}_{t+1}$ to push the representation of the previous timestep towards the next timestep.

L_2 regularization loss to reduce network overfitting, with coefficient c^{L_2} to control the strength of this regularization. The overall loss of a training target is scaled by its importance sampling ratio ρ based on the probability of its being drawn from the replay buffer (see Section 4.3.14). We also use the gradient scaling described by Schrittwieser et al. that halves the gradient at the beginning of each dynamics function call [56]. The constants in the loss functions depend on system configuration, and the constants we used for experiments can be found in Chapter 5.

4.3.9 Updating Neural Network Parameters

We use a standard **Adam** optimizer developed by Kingma and Ba [33]. We also clip the gradient as described by Pascanu, Mikolov, and Bengio [49]. The dynamics function g in our learned model is essentially an RNN, so we expect this gradient clipping trick to have a similar effect in our model. **Optax**, developed by Matteo Hessel et al., is a library for gradient manipulations implemented in JAX [42]. We use Optax's implementation for both the Adam optimizer and the gradient clipper. Moreover, we also use a target network that was used in DQN to stabilize training [44].

4.3.10 *MooZi* Reanalyze

In Section 2.7.1, we reviewed **MuZero Reanalyze**. In our project, we also implement a reanalyze worker process that re-runs search on old trajectories with the latest neural network parameters. Given a trajectory sample $\mathcal{T}$, for each timestep t in the trajectory, the reanalyze process works as follows

- Use observations $(o_{t-T+1}, \ldots, o_t)$ and actions $(a_{t-T}, \ldots, a_{t-1})$ to reconstruct the planner input.
- Feed the planner $\mathcal{P}$ with the reconstructed input, obtaining the update action $\tilde{a_t}$, the updated policy target at the root $\tilde{\mathbf{p}}_t^*$, and the updated value target at the root $\tilde{v}_t^*$.
- Discard the updated action $\tilde{a_t}$ since the action that got executed in the environment has to be the old action a_t to keep the trajectory consistent.
- Replace the old policy target $\mathbf{p}_t^*$ with the updated policy target $\tilde{\mathbf{p}}_t^*$.
- Replace the old value target v_t^* with the updated policy target $\tilde{v}_t^*$.

Once the entire trajectory $\mathcal{T}$ is processed, we obtain an updated trajectory $\tilde{\mathcal{T}}$ in which only the value targets and policy targets are replaced. The updated trajectories are processed into training targets by the replay buffer, and used in training the same way as normally collected trajectories.

4.3.11 Training Worker

The main goal of **training workers** is to generate trajectories by interacting with environments for training purposes. For each worker, a vectorized environment is created as described in Section 4.3.2, a history stacker is created as described in Section 4.3.4, and a planner was created using MCTS configurations as described in Section 4.3.6. Each worker also has a copy of the

parameters similar to that in IMPALA (see Section 2.9.2 and [14]). Step samples and trajectory samples are collected as the planners create actions and the vectorized environments execute the actions. Each worker is allocated one CPU and a fraction of a GPU (usually 10% to 20% of a GPU) so neural network inferences can be done on GPU. Collected trajectory samples are returned to the driver for further management.

4.3.12 Testing Worker

The main goal of **testing workers** is to evaluate the strength of the agent by interacting with environments. These workers are similar to training workers and they hold the same type of data. The differences are: testing workers only use a single environment, have less GPU allocation, and are only run once every n training steps, where n is a configurable number (see configuration in Section 5.1.4).

4.3.13 Reanalyze Worker

The main goal of **reanalyze workers** is to update search statistics using the reanalyze process described in Section 4.3.10, and push the updated trajectories to the replay buffer.

4.3.14 Replay Buffer

The **replay buffer** processes trajectories into training targets and samples trajectories or training targets. Since most training targets are expected to be sampled more than once, the replay buffer precomputes the training targets for all received trajectory samples in the replay buffer with the process described in Section 4.3.7. The replay buffer also computes the value difference δ for each target, which is the difference between the predicted value from the search, and the bootstrapped N-step return

$$\delta_i = |v_i^* - G_i^N|$$

We implemented three modes of sampling: **uniform**, **proportional**, and **rank-based**. In uniform sampling, every training target has equal probability of being drawn. Proportional and rank-based sampling follow the same formula described by Schaul et al. [55]. However, instead of one-step temporal difference error, we use the δ error we described above. For each training target i, the replay buffer also computes the importance sampling ratio $\rho(i)$ based on the probability $P(i)$ of it being drawn

$$\rho_i = \frac{1}{N \cdot P(i)}$$

where N is the number of samples in the buffer. Since the probabilities of targets depends on other targets as well, the importance sampling ratio of targets is not static, and has to be recomputed each time a batch is sampled from the replay buffer.

4.3.15 Parameter Server

The parameter server holds the central copy of the neural network parameters, and updates the parameters. Once a batch of training targets is received by the parameter server, the loss and gradients are computed as described in Section 4.3.8, and the parameters are updated.

4.4 The *MooZi* System in Action

In *MooZi*, Algorithm 3 is the driver that manages all others processes and dataflow. The driver starts by initializing all rollout workers, a parameter server, and a replay buffer. At the beginning of a training step, the driver performs lightweight tasks of all processes such as synchronizing parameters and tally statistics. During the training step, all processes perform their heavyweight tasks such as generating trajectories or updating parameters. Rollout workers interact with environments, the parameter server computes gradients, and the replay buffer processes trajectories into training targets. The method calls made by the driver do not block. They schedule call events and return immediately rather than waiting for the methods to finish. The immediate return values of the calls are *promises* managed by Ray [17]. Actors execute their scheduled method calls sequentially once their concrete inputs are ready. Figure 4.5 illustrates how tasks are executed in parallel over time.

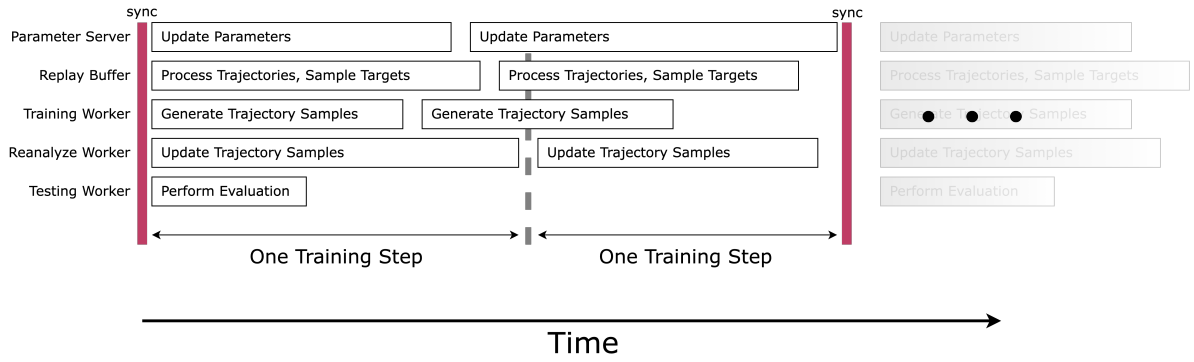

Figure 4.5: Timeline of training steps. The red bar indicates a synchronization barrier. The duration of each training step is decided by the last finished task.

```
start_training = False
trajectory_samples = []
parameter_server = make_parameter_server()
replay_buffer = make_replay_buffer()
training_worker = [make_train_worker() for i in range(num_train_workers)]
testing_worker = make_test_worker()
reanalyze_workers = [make_reanalyze_worker() for i in range(num_reanalyze_worker)]

for epoch in range(num_epochs):
    if not start_training:
        if start_training_condition_met():
            start_training = True
    for worker in training_worker + testing_worker + reanalyze_workers:
        if update_condition_met(worker):
            worker.set_parameters(parameter_server.get_parameters())

    replay_buffer.process_trajectory_samples(trajectory_samples)

    if start_training:
        for i in range(num_updates_per_epoch):
            batch = replay_buffer.sample_batch(batch_size)
            parameter_server.update(batch)

    trajectory_samples.clear()
    for worker in training_worker:
        sample = worker.run()
        trajectory_samples.append(sample)

    if testing_condition_met():
        test_result = testing_worker.run()

    if start_training:
        for worker in reanalyze_workers:
            trajectories_to_update = replay_buffer.get_trajectory_samples()
            worker.refill_trajectory(trajectories_to_update)
            updated_trajectories = worker.run()
            replay_buffer.add_trajs(updated_trajectories)
```

Algorithm 3: The driver.

4.5 Logging and Visualization

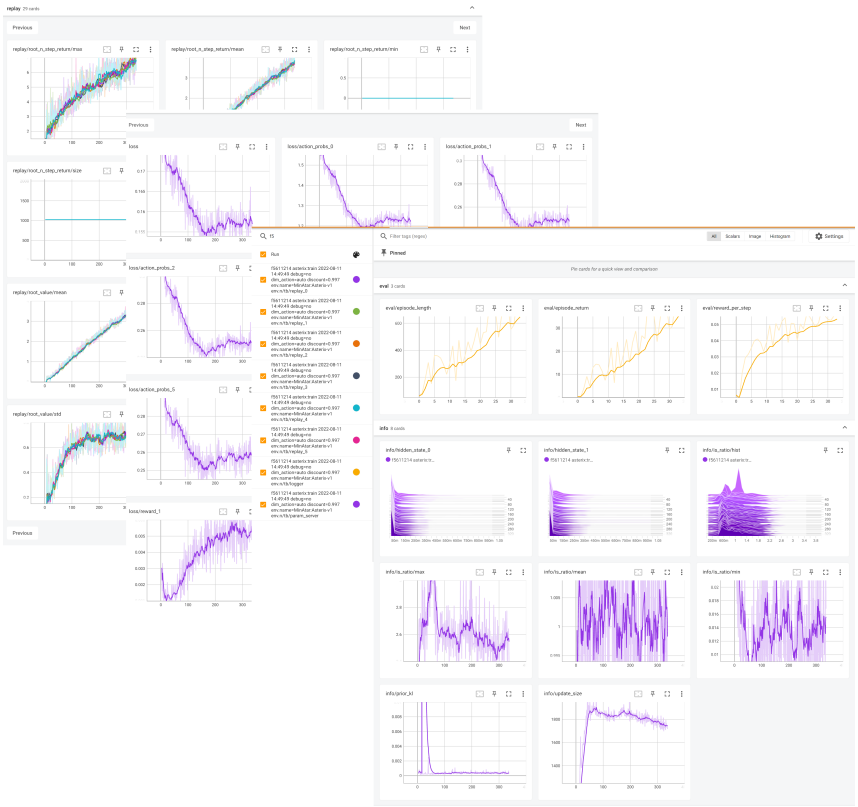

Figure 4.6: *MooZi Tensorboard* dashboard.

MooZi incorporates extensive logging and visualization utilities to help users understand its behavior better. All distributed processes maintain a dedicated log file that records all important events within the process. *MooZi* uses *TensorBoard* [1] to log informative scalar and vector quantities. Figure 4.6 shows the *MooZi* TensorBoard dashboard. *MooZi* logs over 50 different metrics into this dashboard. These include 29 metrics from the training workers, 3 metrics from the testing logger, 20 metrics from components of the loss function, 8 metrics from the parameter optimizer. Optionally, the dashboard shows distributions of parameters from all neural network layers. *MooZi* also provides utilities to visualize the behavior of the algorithm (shown in Section 5.7 and Section 5.9). The test worker saves trajectories as annotated GIFs that are easy to play and analyze on a frame-to-frame basis. Figure 4.7 shows an example of such a GIF tiled as images.

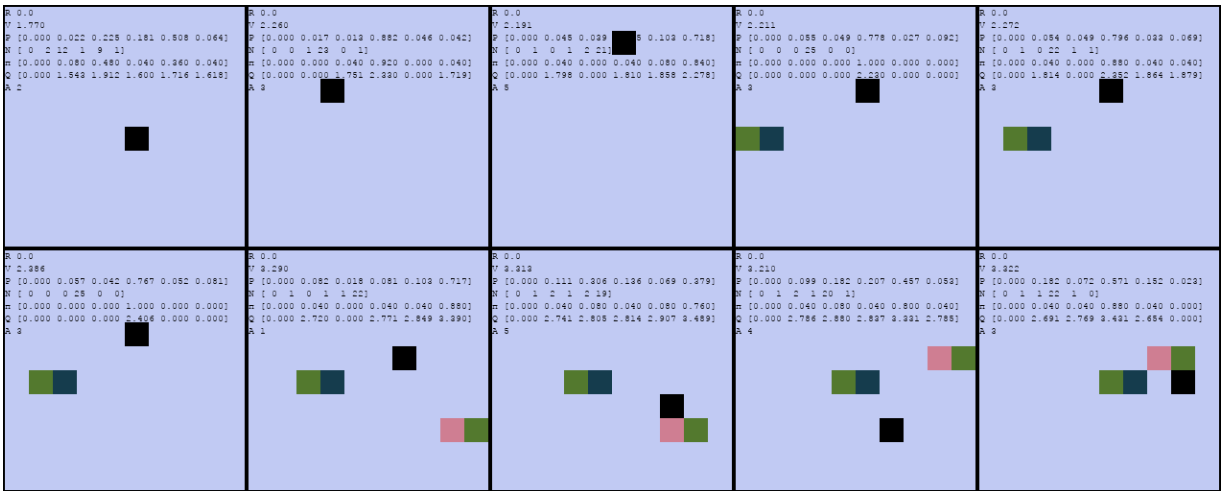

Figure 4.7: *MooZi* produces trajectories in *Asterix* as GIFs with annotations, presented here as tiled images of every four frames. In these images, R is the reward r_{t-1} given from the last timestep, V is the value prediction v_t^* after search, P is the prior policy $\mathbf{p}_t^0$ at the root before search, N are the visit counts at the root after search, π is the policy target $\mathbf{p}_t^*$ of the timestep, Q are the Q-values, and A is the index of the action to take.

5 Experiments

We run the *MooZi* system on both video games and board games to demonstrate *MooZi*'s ability to learn and perform in both domains. We use *MinAtar* environments (reviewed in Section 2.8.7) as video game environments, and Breakthrough as a board game environment. We compare *MooZi*'s performance in *MinAtar* environments with published results from Gymnax [51] and *MinAtar* [67]. We evaluate *MooZi*'s performance in Breakthrough by comparing *MooZi* models with different amount of training. We analyze *MooZi*'s behavior in these environments and discuss the behavior of the system during training.

5.1 Experiment Setup

5.1.1 Basic Configuration

For all of our experiments, we use *MooZi* with unrolled steps $K = 5$, history length $L = 4$, and bootstrap steps $N = 10$. We use a discount of 0.997 and a support Z from the interval $[-30, 30]$ ($|Z| = 61$). MuZero by Schrittwieser et al. only used the transformation Φ in Atari environments [56], but we discovered the same support works well for both *MinAtar* environments and *Breakthrough*, so we always use it. We run all experiments on a single computer with Intel Xeon CPUs (72 $\times$ 2.3 GHz), Nvidia Tesla V100 GPUs (8 $\times$ 32 GB), and 500 Gigabytes of system memory. Each run roughly consumes 50% of available computation resources of this computer, and we usually run two instances of the system at the same time.

5.1.2 Neural Network Configurations

We use the residual-blocks-based variant (see Section 4.3.5) of the network for all of our experiments.

Residual Block

We follow the residual block definition by He et al. [24]. The computation of one residual block is computed as follows:

- input x

- save a copy of x to x'
- apply a 2-D padded convolution on x, with kernel size 3 by 3, same channels
- apply batch normalization on x
- apply ReLU activation on x
- apply a 2-D padded convolution on x, with kernel size 3 by 3, same channels
- apply batch normalization on x
- add x' to x
- apply ReLU activation on x

The Representation Function Network

The representation function $\mathbf{x} = h$ is computed as follows:

- input a stacked history ψ of shape (H, W, C_h)
- apply a 2-D padded convolution on ψ, with kernel size 1 by 1 and 32 channels
- apply 6 residual blocks with 32 channels on ψ
- output the hidden state $\mathbf{x}$ of shape $(H, W, 32)$

The Prediction Function Network

The prediction function $\mathbf{p}, v = f(\mathbf{x})$ is computed as follows:

- input hidden state $\mathbf{x}$ of shape $(H, W, 32)$
- apply 1 residual block with 32 channels on $\mathbf{x}$
- flatten $\mathbf{x}$, now shape $(H * W * 32)$
- apply 1 dense layer with output size of 128 on flattened $\mathbf{x}$ to obtain the value head $\mathbf{x}_v$, now shape (128)
- apply batch normalization and ReLU activation on $\mathbf{x}_v$
- apply 1 dense layer with output size of $|Z|$ on $\mathbf{x}_v$, now shape (1)
- output the value head $\mathbf{x}_v$ as the value prediction v

- apply 1 dense layer with output size of 128 on flattened $\mathbf{x}$ to obtain the policy head $\mathbf{x}_p$, now shape (128)
- apply batch normalization and ReLU activation on $\mathbf{x}_p$
- apply 1 dense layer with output size equals to the action space size on $\mathbf{x}_p$, now shape $(|\mathcal{A}^a|)$
- output the policy head as the policy prediction $\mathbf{p}$

The Dynamics Function Network

The dynamics function $\mathbf{x}, \hat{r} = g(\mathbf{x})$ is computed as follows:

- input hidden state $\mathbf{x}$ of shape $(H, W, 32)$, action a as an integer
- encode a as action planes of shape $(H, W, |\mathcal{A}^a|)$ (same procedure as in Section 4.3.4)
- stack $\mathbf{x}$ on top of the encoded action, now shape $(H, W, 32 + |\mathcal{A}^a|)$
- apply a 2-D padded convolution on ψ, with kernel size 1 by 1, 32 channels, now shape $(H, W, 32)$
- apply 1 residual block with 32 channels on $\mathbf{x}$ to obtain the hidden state head $\mathbf{x}_s$
- apply 1 residual block with 32 channels on the hidden state head $\mathbf{x}_s$
- output the hidden state head $\mathbf{x}_s$ as the next hidden state $\mathbf{x}'$
- apply 1 dense layer with output size of 128 on $\mathbf{x}$ to obtain the reward head $\mathbf{x}_r$, now shape of (128)
- apply batch normalization and ReLU activation on $\mathbf{x}_r$
- apply 1 dense layer with output size of $|Z|$ on $\mathbf{x}_r$, now shape of $(|Z|)$
- output reward head $\mathbf{x}_r$ as the reward prediction $\hat{r}$

Ye et al. discovered that a small dynamics function network is sufficient for Atari environments, and our MinAtar experiments also use a small dynamics function network with only one residual block as the trunk. However, we noticed that such a small dynamics function network isn't sufficient to fully learn the environment dynamics in *Breakthrough*. As a result, we use a larger dynamics function network with 6 residual blocks for the *Breakthrough* experiments in Section 5.7.

Worker	c_1	c_2	Temperature	Dirichlet Noise	Simulations
Training	2.25	19652	1.0	0.2	25
Reanalyze	1.75	19652	-	0.2	50
Testing	1.75	19652	0.25	0.1	40

Table 5.1: Planner configurations.

The Projection Function Network

The projection function $\mathbf{x} = \varrho(\mathbf{x})$ is simply one residual block (see Section 4.3.5).

Network Training

In the loss function described in Section 4.3.8, we use parameters $c^v = 0.25$, $c^s = 2.0$, and $c^{L_2} = 1.0 \times 10^{-4}$. We use a batch size of 1024, a learning rate of 1.0×10^{-2}, and a global norm clipping of 5.0. We perform gradient updates with a number of samples that is four times the number of step samples generated in each training step. For example, if we have 30 training workers, each with 16 environments, and each performs 100 environment steps per training step, then the total number of step samples generated is $30 * 16 * 100 = 48000$. This means that we update the gradient using $4 * 48000 = 192000$ sampled training targets from the replay buffer. That is $\frac{192000}{1024} \approx 188$ gradient updates from mini-batches of size 1024 per training step.

5.1.3 Planner Configurations

Table 5.1 shows the configurations of planners from different type of workers. Reanalyze workers do not sample actions to act so the temperature parameter does not affect their behavior. c_1 is the exploration constant in the PUCT formula from Section 2.7. A greater c_1 favors the less visited actions more. c_2 is also an exploration constant in the PUCT formula. We use the same value as [56] because we find it sufficient to tune c_1 to balance exploration. The *temperature* controls how action is selected from the distribution of action visit counts from the root nodes. A temperature of 0 selects the most visited action at the root node. A temperature of ∞ means random action selection. The *dirichlet noise* controls the exploration noise added to the actions at the root node. A greater dirichlet noise adds more prior probability to less explored actions. The *simulations* parameter is the number of simulations that the planner performs at each timestep (see Section 2.3). The training workers favor exploration and generate data quickly with fewer simulations. The testing workers favor exploitation and spend more time on simulations to get better average return. The reanalyze workers do not need to interact with the environment so they spend more time performing simulations.

5.1.4 Driver Configuration

We use 30 training workers, each with 16 copies of the environment. For every training step, each training worker performs 100 environment steps. The *Freeway* environment has much longer episodes, so each training worker uses only 2 environments and performs 2500 environment steps per training step. We use 1 testing worker with one copy of the environment. For every 10 training steps, the testing worker collects 10 trajectories and logs the average return of the trajectories. We only use the reanalyze worker for the experiments in Section 5.5. All workers sync with the latest neural network parameters every 10 training steps.

5.2 *MooZi* vs PPO in *MinAtar* Environments

We run *MooZi* using all five environments in MinAtar: *Breakout*, *Space Invaders*, *Freeway*, *Asterix*, and *Seaquest* [67]. All environments produce frames of resolution 10×10, with four to seven environment channels C_e. In *Breakout*, the player moves a paddle left or right on the bottom of the screen to bounce a ball to hit the bricks. The reward is $+1$ for each brick broken and 0 in all other situations. The game ends when the paddle fails to catch the ball. In *Space Invaders*, the player controls a cannon that can move left, move right, or fire the cannon. A cluster of enemies moves across the screen and they fire at the player. The reward is $+1$ for each enemy hit by the player and 0 in all other situations. The game ends when the player is hit by an enemy's bullet. In *Freeway*, the player starts at the bottom of the screen and can move up or down once every three frames to travel across a road with traffic, or stay in place. Cars spawn randomly and travel horizontally across the screen. When they hit the player the player is moved back to its starting position. The reward is $+1$ for each time the player successfully travels across the whole road and 0 in all other situations. The game ends after 2500 timesteps. In *Asterix*, the player moves in four directions, and enemies and treasures spawn randomly on the edges. The reward is $+1$ if the player obtains a treasure and 0 in all other situations. The game ends when the player bumps into an enemy. Enemies have different speed indicated by the color of their tail.

Figure 5.1 shows screen shots of the MinAtar environments. MuZero is reported to have much better performance in more deterministic environments [48]. To compare *MooZi* in more deterministic environments, we compare with the results reported by Gymnax, in which algorithms are evaluated in environments with no sticky action. We discuss this difference in section 5.3.

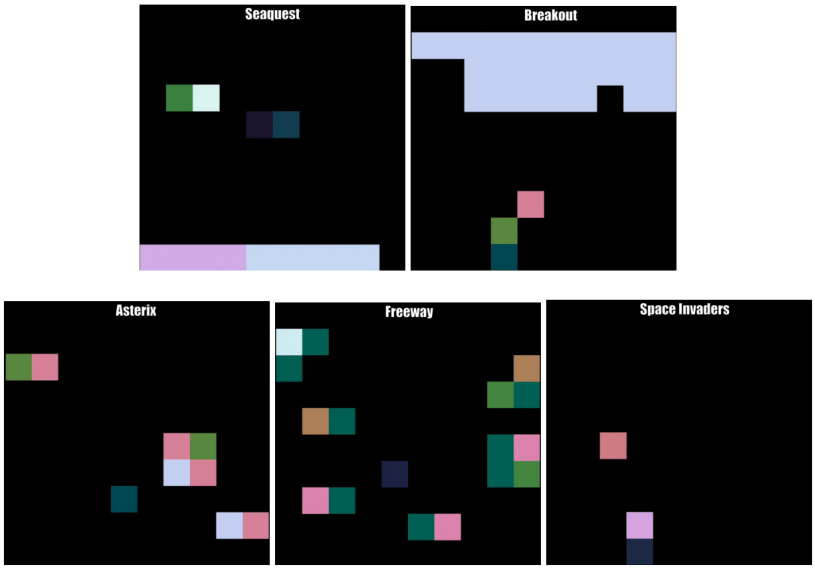

Figure 5.1: *MinAtar* Environments, from Young and Tian [67].

Figure 5.2 shows the evaluation of *MooZi* on *MinAtar* environments. We run *MooZi* with three seeds on each of these environment. Gymnax benchmarked the performance of PPO with three sets of hyper-parameters in each of these four games [51, 58]. We compare our results with the best performing PPO in each of these games. The average return uses the planner setting of a test worker in Section 5.1.3. In *Breakout*, *MooZi* obtained a near-optimal strategy and the paddle almost never fails to catch the ball. There is no default step limit in this environment so we cap the return at 100. Similarly in *Space Invaders*, *MooZi* obtained a near-optimal strategy and we cap the return at 300. In *Freeway*, two out of three runs obtained a near-optimal strategy, and one run failed to learn a stable policy. Since each episode in this environment is much longer than in other environments, we have to configure *MooZi* differently to run 2,500 worker steps per training step. This drastically reduced the cycles of updating the network and generating data using the newer network. As a result, *MooZi* learning is much less stable in this environment than in other environments. In *Asterix*, both *MooZi* and PPO do not obtain an optimal strategy, but *MooZi* achieves twice as much average return as PPO. In all environments, a single run takes about 10 hours. Gymnax does not provide their runtime statistics, but we presume their result is at least an order of magnitude faster than ours since they do not use planning in acting.

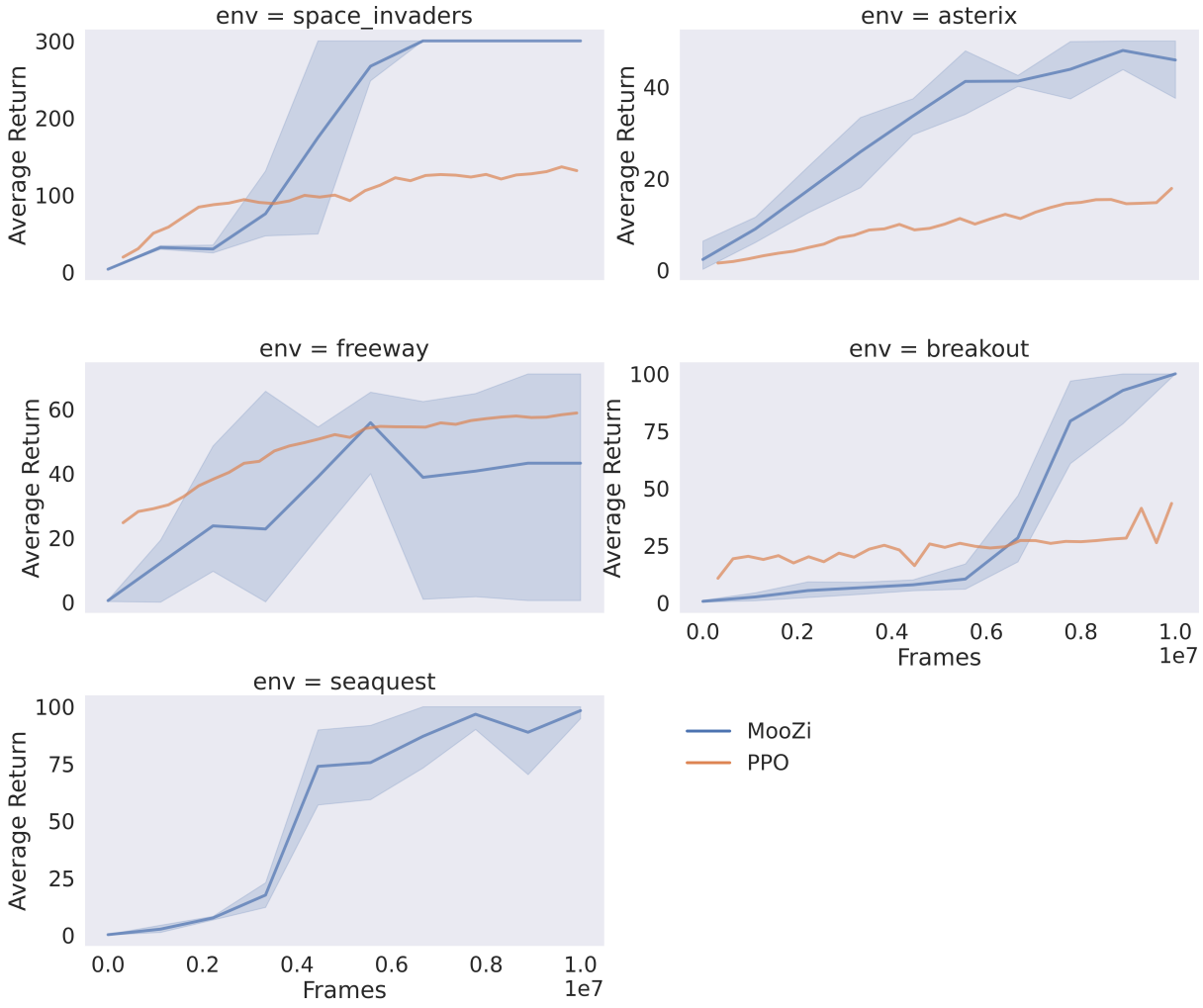

Figure 5.2: Evaluation of *MooZi* in *MinAtar* games. Each game has three *MooZi* runs and four games have one PPO run reported by Gymnax. The x axis shows the number of environment frames used in training. The y axis shows the average return of the algorithm in the environment.

5.3 Sticky Actions in MinAtar

MinAtar environments have a default sticky action probability of 10%. This means that in one out of ten timesteps, the environment uses the last taken action to step the environment instead of using the agent's output action. For example, assume that at time t, the agent outputs action $a_t = \textit{MoveLeft}$ and moves to the left. At time $t + 1$, the agent outputs action $a_{t+1} = \textit{MoveRight}$. However, if the environment applies the sticky action and overrides the action, then $a_{t+1} = \textit{MoveLeft}$, and the agent moves to the left even further. The presence of a non-zero sticky action probability adds stochasticity to environments and changes the set of optimal polices. For example, in *Space Invaders*, if the sticky action probability is 0, then the agent can move away from enemy bullets just in time, one frame before the agent is about to get hit. However, with a sticky action probability of 10%, moving away one frame in advance means there's a 10% chance that the agent will die, and moving away two frames in advance means there's a $(10\% * 10\%) = 1\%$ chance that the agent will die two frames later. We observe that a *MooZi* agent trained in *Space Invaders* with a 10% sticky action probability moves away from an enemy bullet right after the bullet becomes visible on the screen. Young and Tian shipped the MinAtar environments with four algorithms including two variants of DQN and two variants of an Actor-Critic (AC) method. The *MinAtar* paper [67] only reports *Breakout* results with a sticky action probability of 10%. Gymnax only reports *Breakout* results without sticky action probability. We compare with *MinAtar* and Gymnax in their respective testing environments using the same *MooZi* agent configuration. In Figure 5.3, we use the final average returns of PPO and AC reported in Gymnax and *MinAtar*'s paper respectively. In *Breakout* with sticky actions, *MooZi* learns more slowly and the final average return is much lower. In *Breakout* without sticky actions, *MooZi* quickly learns a near-optimal policy and never fails to return the ball. In both environments, *MooZi* out-performs the other algorithm.

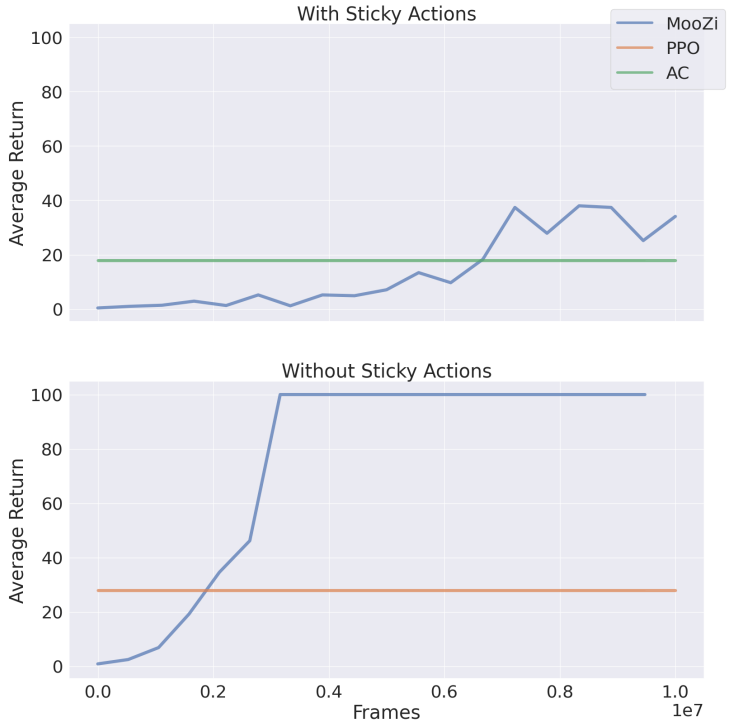

Figure 5.3: Evaluation of *MooZi* vs PPO vs AC in two variants of *Breakout*. The x axis shows the number of environment frames used in training, in units of 1.0×10^7 frames. The y axis shows the average return of the algorithm in the environment. *Top*: Non-deterministic variant of *Breakout* with sticky actions. *Bottom*: Deterministic variant of *Breakout* without sticky actions. The average return of PPO and AC are their final reported return.

5.4 Testing Strength when Scaling the Search Budget in *Asterix*

We use the trained *MooZi* model in the *Space Invaders* environment to evaluate its strength while varying the number of simulations. We only use model checkpoints from the first 3 million environment frames because after that the agent behaves optimally even when using the prior policy to act. For each of these checkpoints, we run a testing worker to collect 30 episodes and compute the average return of these episodes. The testing worker uses the same planner configuration as the testing worker in Section 5.1.3 except for the number of simulations. Figure 5.4 shows the result. We observe that with a greater number of simulations, the agent tends to perform better. With more training, the agent always performs better. The difference of performance due to simulation count seems to be smaller than the difference due to training. For example, with 2.5 million frames of training, acting using 64 simulations gives an average return around 70. With 3 million frames of training, acting according to the prior (1 simulation) does even better and gives an average return around 75. The prior policy, with a bit more training, quickly catches up to or even exceeds the deep search policy with less training. This finding aligns with the analysis by Hamrick et al. [20]: the planner contributes more to the algorithm by *generating better data for training the model* rather than *exploiting the model for better testing*.

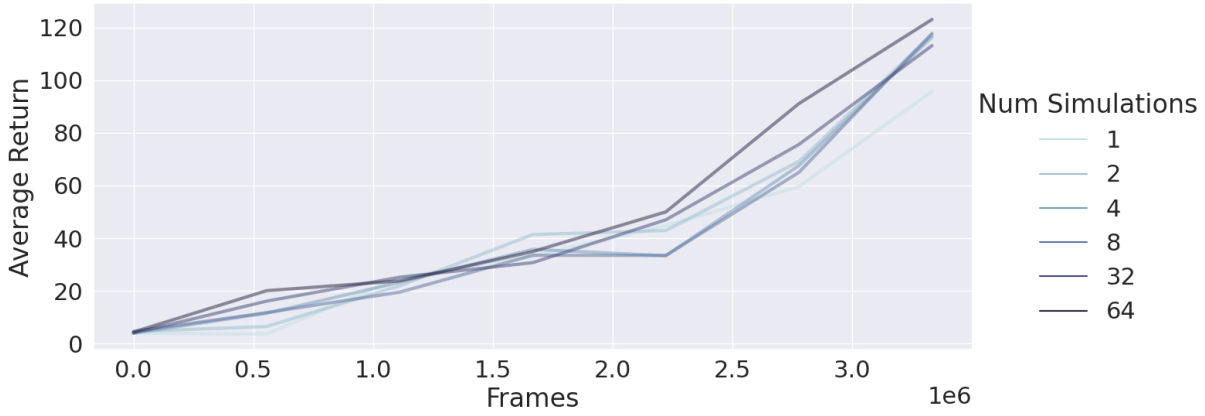

Figure 5.4: Agent strength with different number of simulations. The x axis shows the number of environment frames used in training, in unit of 1.0×10^6 frames. The y axis shows the average return of the algorithm.

5.5 Improving Sample Efficiency with Reanalyze in *Asterix*

We run three sets of parameters of *MooZi*, each with three experiments in the *Asterix* environment. We configure training workers and reanalyze workers to generate the exact same

amount of training targets for each training step. We control the total number of generated training targets and network updates per training step by fixing the sum of number of training workers and reanalyze workers to 20. The parameter set *20 : 0* indicates 20 training workers and 0 reanalyze workers. This set is also used in Section 5.2. The parameter set *16 : 4* uses 16 training workers and 4 reanalyze workers. Similarly, the parameter set *12 : 8* uses 12 training workers and 8 reanalyze workers. We fix the total environment frames to 10 million. Parameter sets with more reanalyze workers take more time to run, as they consume environment frames budget more slowly. Figure 5.5 shows the performance of *MooZi* with Reanalyze workers. The parameter set with 8 reanalyze workers used 3 million environment frames to reach an average return that takes the parameter set with 0 reanalyze workers 5 million environment frames. The number of training targets used for training in the former is $3 * (\frac{20}{12}) = 5$ million and this equals the 5 million mark of the latter . This means that in the early stage of the training, each training target generated by the reanalyze workers contributes to the training as much as one training target generated by the training workers. In the later stage of the training, all runs reached a similar final average return with the same amount of environment frames. This means that the training targets generated by the reanalyze workers do no longer contribute, or can even harm the training process in the later stage. We hypothesize that the reason for this observation is that in the early stage of the training, training targets shift more quickly, and updating past trajectories with the latest model is more valuable. However, in the later stage of the training, training targets stabilize and it becomes more important to have better performing trajectories than more accurate estimates from worse trajectories.

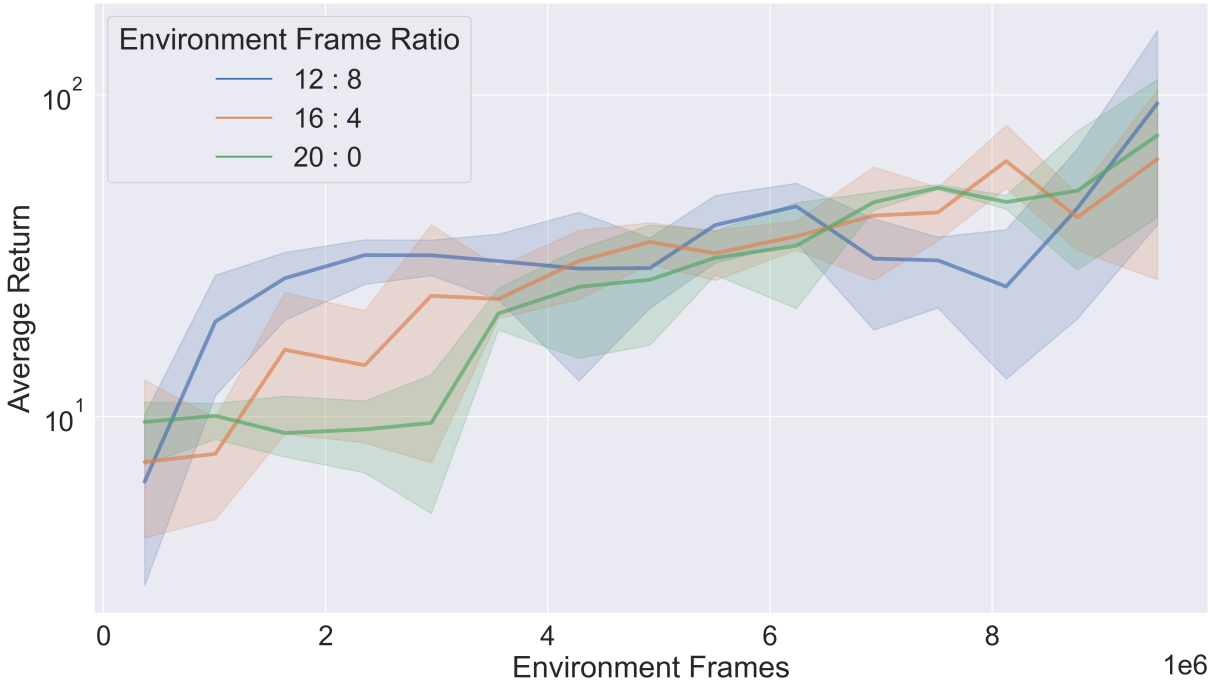

Figure 5.5: *MooZi* with reanalyze workers in *Asterix*. The x axis shows the number of environment frames used in training, in units of 1.0×10^6 frames. The y axis shows the average return of the algorithm on a log scale.

5.6 Analysis of Planning in *Space Invaders*

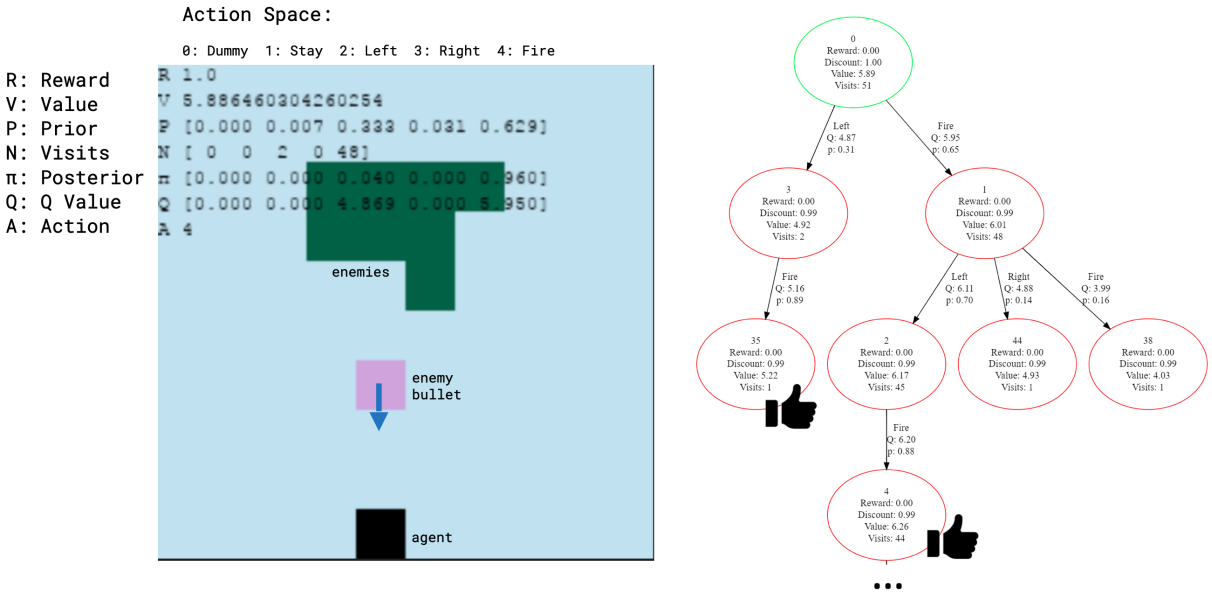

Figure 5.6: *MooZi* acting in *Space Invaders* environment by planning with a learned model. *Left*: Example of an observation o_t and the agent's statistics. *Right*: The search tree built by planning with the agent's learned model in this example.

We show an example of *MooZi* acting in the *Space Invaders* environment by planning with a learned model in Figure 5.6. The left hand side shows the latest environment observation o_t. We also annotate agent statistics on the image. The right hand side shows the search tree of the agent for deciding the next action. Row *R* shows the reward from the last timestep r_t. The agent hits an enemy in the last timestep so $r_t = 1$. Row *V* shows the value prediction v_t^* at the current timestep, which is also the value of the root node in the search tree on the right hand side. Row *P* shows the prior policy $\mathbf{p}_t^0$ before the tree search. The prior policy is obtained by performing one initial inference described in Section 4.3.5. Row *N* shows the visit counts of nodes at the root. π shows the posterior policy $\mathbf{p}_t^*$ after the tree search. Row *Q* shows the one-step returns of $\hat{r}_t^1 + \gamma v_t^1$. Row *A* shows the next action to take in the environment. The prior policy shows that the agent favors two actions, *Left* and *Fire*. The reason for going *Left* is because the agent observes an enemy bullet traveling towards the agent. The search reveals why *Fire* is also a good idea: if the agent decides to *Fire* at t, it could still dodge *Left* at $t+1$. The subtle difference is that by delaying the dodging by one timestep, the agent hits the enemy one timestep earlier, and receives a greater discounted return. The agent also learns to not dodge *Right* in this situation, and the prior of this action is so low that it is not even explored in the search. This is because the enemies are moving to the left. If the agent takes a *Right*, subsequent *Fire*s will miss. After the

search, the posterior probability of taking the action *Fire* is 96%, which means the agent is going to take this action 96% percent of the time. The training target created from this interaction will also shift the prior of this action towards 96% in future similar simulations.

5.7 Learning through Self-play in *Breakthrough*

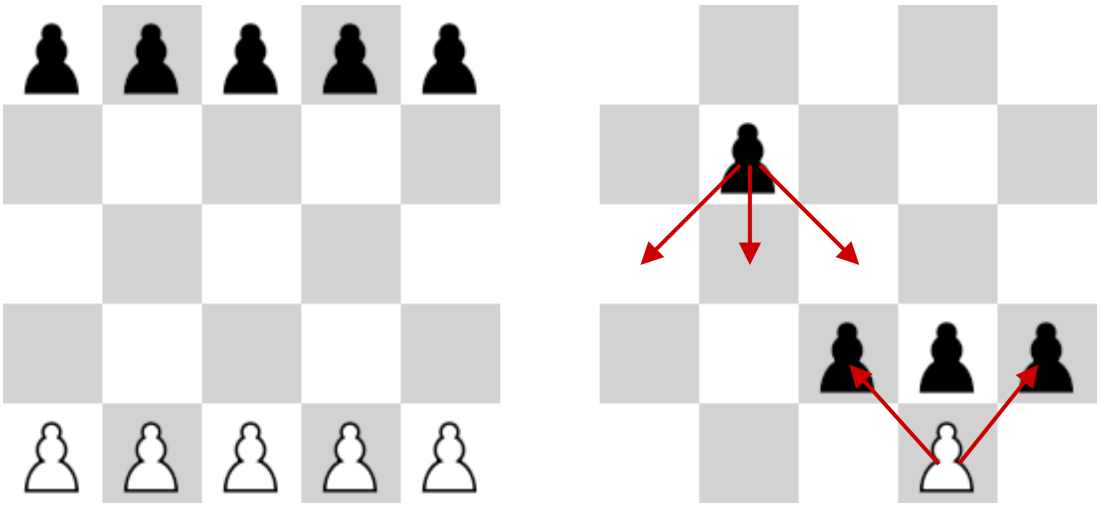

Figure 5.7: The board game *Breakthrough*. *Left*: The *Breakthrough* (5×5) starting board used in our experiment. *Right*: Illustration of the *Breakthrough* movement rule.

We evaluate *MooZi* in the board game *Breakthrough*. *Breakthrough* is played by two players, zero-sum, deterministic, and with perfect information for both players. The original *Breakthrough* is played on a 7×7 board. In our experiment we use smaller board size of 5×5, with fewer pieces for both players. Each piece can move forward, diagonally forward, or capture an enemy piece diagonally forward. The player who first reaches the home row of its opponent wins. Figure 5.7 illustrates the starting board used in our experiment and the movement rules. We use the *OpenSpiel* implementation of *Breakthrough*. The action space size of the environment is around 300, and a typical state has five to twelve legal actions.

We train *MooZi* through self-play with a shared network for both players. The planner is configured for a two-player game as described in Section 4.3.6. Figure 5.8 shows the evaluation of *MooZi* throughout training. We evaluate 23 model checkpoints by running a round-robin tournament in which all models pairs with all other models and play 16 matches alternating first and second player. We initialize the Elo of each checkpoint to zero and compute the new Elo rating based on the results of the tournament. The steady Elo rating increase shows that the model learns reliably through self-play.

Figure 5.8: Evaluation of *MooZi* training in *Breakthrough*. The x axis shows the number of training steps of 23 model checkpoints. The y axis shows the Elo rating of the checkpoints.

5.8 Analysis of Planning in *Breakthrough*

Figure 5.9 shows an example of *MooZi* planning in a two-player game using the learned model in *Breakthrough*. For creating the illustration, we use a game implementation with the perfect model, and rollout perfect game states using the actions sampled by the search to create a perfect game state tree. We overlap the *MooZi* search tree and the perfect game state tree to create the visualization in the figure. The learned model only knows the legal actions at the root level, so it is possible for the planner to create nodes using illegal actions beyond the root level. When this happens, such nodes will not have their corresponding legal game states. We call a search tree node without a corresponding legal game state a *delusion*, and we color these nodes with light pink or white. Node A is a node in the search tree with its corresponding legal game state rendered as a board position. Node B is a *delusion* created by the learned model by taking action of index 41 (annotated as "Action: 41" on the illustration) that's not legal in the game state corresponding to node A. Node C is not a *delusion* because action of index 101 is one of the legal actions in the game state of node A. The search cannot differentiate *delusional* nodes from valid nodes, and the value predictions of node B and node C are both used to update their parent node A through backpropagation. One special case of a *delusion* is when the search takes the dummy action after a terminal state. Even though the dummy action is not a legal action in the game, it represents a valid prediction that the game ends.

In the early stage of training, the model frequently samples illegal actions beyond the root

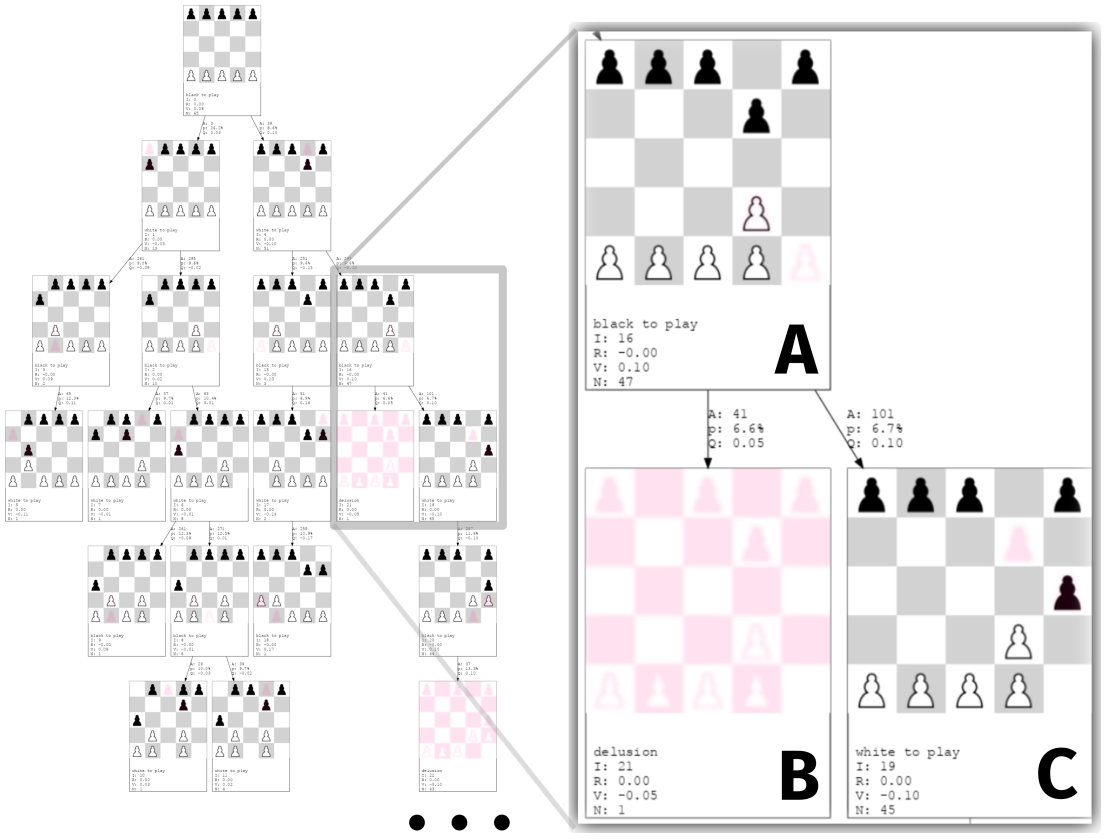

Figure 5.9: *MooZi* planning with a learned model in *Breakthrough*. On nodes: I is the node index, R is the reward prediction $\hat{r}$, V is the value prediction v, and N is the node visit count. On edges: A is the action index, p is the prior probability, Q is the Q-value.

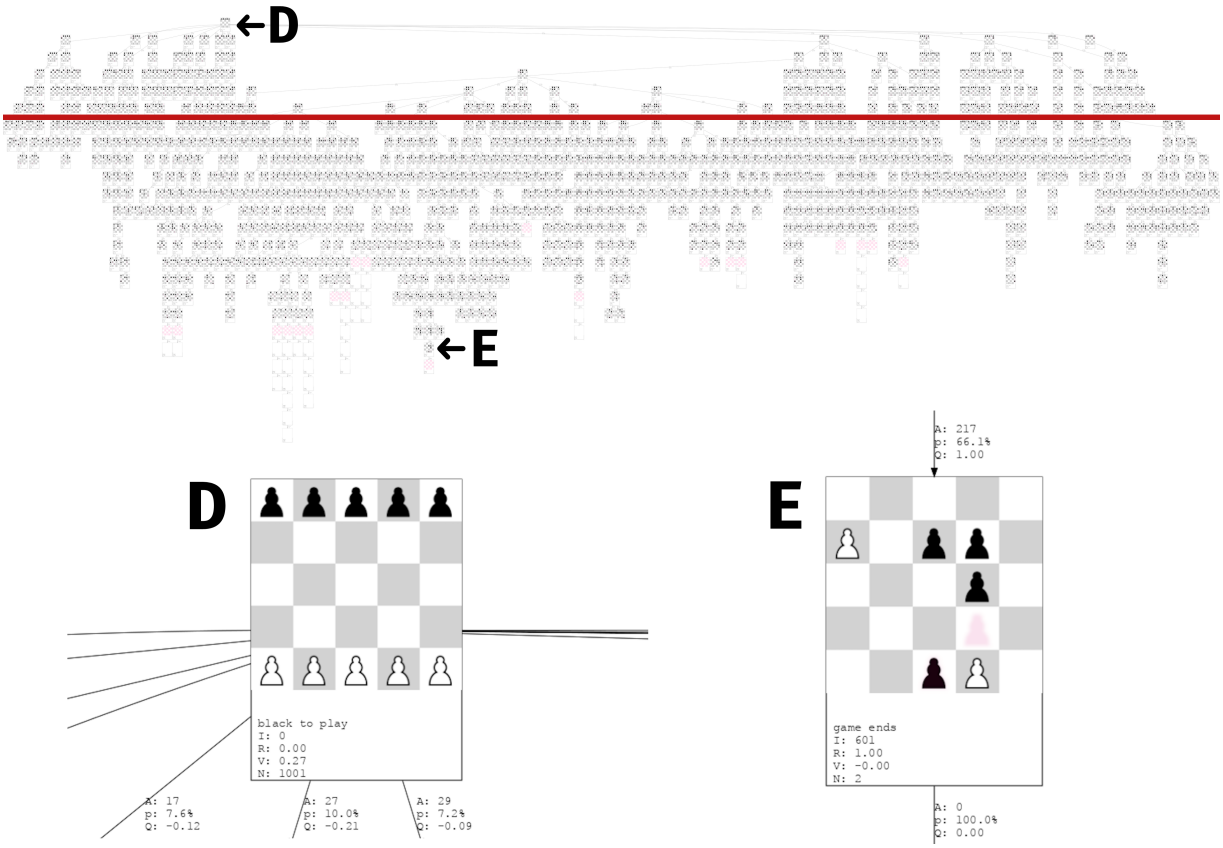

Figure 5.10: *MooZi* planning with a learned model that fully captures environment dynamics. The model accurately rollout game states from the starting state *D* all the way to the terminal state *E*. The red line is the cut-off of the training horizon, above which predictions are enforced by the training process.

level, and the search trees are full of *delusions* with arbitrary value and reward predictions. As the model learns, the search trees become more accurate, and can fully capture the environment dynamics. Figure 5.10 shows a search tree created using a well-learned model. This model is trained with $K = 5$ unroll steps to predict the value v, reward $\hat{r}$ and action distribution $\mathbf{p}$ within a tree depth of five. On top of the figure is the entire search tree with the root node D as the top node. The red line in the figure indicates the depth above which is within the training horizon and most nodes in the search tree are beyond the training horizon. The planner uses the learned model to unroll from the starting state D all the way to the terminal state E, correctly predicts a black win with a reward of 1, and correctly predicts game termination by selecting the dummy action afterwards. Moreover, this search tree satisfies all the following conditions: (1) every node is either associated with a legal game state, or correctly predicts game termination (2) every predicted reward is within 1% margin of the actual reward of the associated game state (3) every node after a terminal node takes the dummy action and predicts a zero value and a zero reward. This example shows that a well-learned model learns game dynamics far beyond five steps and can be used almost like a perfect model.

5.9 Visualizing the Hidden Space of the Learned Representation

We visualize the learned representation of *MooZi* by projecting hidden states into a lower dimension using the dimension reduction technique *t-distributed stochastic neighbor embedding (t-SNE)* [41]. We use a semi-trained (100K training steps) *Breakthrough* model and a planner with more exploration (*dirichlet fraction* = 0.5) to make the data more diverse. We collect data by running self-play for 5K steps and for each timestep t we record the value prediction after search v^*, hidden state at the root $\mathbf{x}^0$, and the current player b^{player}. We run *t-SNE* using only the hidden states to generate a two-dimensional embedding for each data point. Figure 5.11 illustrates a shared embedding colored based on their values, move numbers (timestep t), and player indices in three subplots. The bottom subplot shows hidden states are most distinct by their corresponding players. This is reasonable because the entire search is pivoted from the root player's point-of-view. The middle subplot shows a sense of "game progression" within clusters. Combining the middle and the top subplots, we observe that as the game progresses, the value predictions diverge to two extremes.

Figure 5.11: The hidden space of the learned representation visualized through *t-distributed stochastic neighbor embedding (t-SNE).* *Top*: colored by the value predictions after search v^*. *Middle*: colored by the move number (also timestep) t. *Bottom*: colored by the player index.

6 Conclusion

In this thesis, we present *MooZi*, a high-performance game-playing system that plans with a learned model. We developed a *MuZero*-based learning algorithm and parallelized the algorithm using a hierarchical control. We empirically showed that *MooZi* learns a model and plans with the model in both single-agent and two-player domains. In *MinAtar* environments, the agent fights enemies or collects resources and gains rewards along the way. In these environments, *MooZi* achieved a greater average return than the Proximal Policy Optimization algorithm in five deterministic *MinAtar* environments including *Breakout*, *Space Invaders*, *Asterix*, *Freeway*, and *Seaquest*. *MooZi* also achieved a greater average return than the Actor-Critic algorithm in the non-deterministic *MinAtar* environment *Breakout* with sticky-actions. In a two-player board game, the goal of the agent is to beat the other agent in competition. We trained *MooZi* in the two-player board game of *Breakthrough* and showed that *MooZi* learned to master the game through self-play. We visualized and analyzed the search trees in both domains to demonstrate how *MooZi* plans with a learned model. We showed an example where the learned model fully captures game dynamics beyond its training horizon. We projected the learned representation into a lower dimension and showed how the hidden states travel in the space as game progresses. Finally, we make *MooZi* publicly available to accelerate future research.

7 Future Work

MooZi is an on-going project and we aim to generalize *MooZi* even further and support more games. We reported results of *MooZi* in *Breakthrough*, but *MooZi* supports all other two-player perfect-information deterministic games in *OpenSpiel*. We will run *MooZi* in more such games and at the same time try to find a single set of hyperparameters that works well. Currently *MooZi* assumes the environment dynamics to be deterministic. A promising direction is to handle stochasticity and support non-deterministic video games and board games. Prior work extended the *MuZero* algorithm with encoded random nodes [48, 2], and it will be our top priority to extend *MooZi* in a similar fashion. *MooZi* currently does not support continuous action space and has poor support for large discrete action space. We can extend *MooZi* to support continuous action space using *Sampled MuZero* [29]. We can support large discrete action spaces better by reducing the memory footprint through packing data with a library that specializes in handling sparse N-dimensional arrays. *MooZi* already supports planning with Gumbel [11] through MCTX. We conducted preliminary experiments and we will follow up with a set of more rigorous experiments. Moreover, *MooZi* currently interweaves handling of episodes and timesteps. Games such as *Freeway* that have extremely long episodes require a large number of environment interactions per training step to generate at least one episode. We will unify the handling of long episodes by slicing episodes into fixed-size sequences. Moreover, experience represented as fixed-size sequences can be implemented using native JAX and speed up the system by making search targets ($\mathbf{p}^*$ and v^*) stay on GPUs or TPUs in their lifetime. In short, we will make *MooZi* more general as well as more efficient.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[2] Ioannis Antonoglou et al. "Planning in Stochastic Environments with a Learned Model". In: International Conference on Learning Representations. Mar. 15, 2022. URL: https://openreview.net/forum?id=X6D9bAHhBQ1 (visited on 09/03/2022).

[3] Karl Johan Åström. "Optimal Control of Markov Processes with Incomplete State Information I". In: *Journal of Mathematical Analysis and Applications* 10 (1965), pp. 174–205. ISSN: 0022-247X. URL: http://lup.lub.lu.se/record/8867084 (visited on 08/24/2022).

[4] *Atari 2600*. In: *Wikipedia*. July 24, 2022. URL: https://en.wikipedia.org/w/index.php?title=Atari_2600&oldid=1100194806 (visited on 07/29/2022).

[5] Marc G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 14, 2013), pp. 253–279. ISSN: 1076-9757. DOI: 10.1613/jair.3912. arXiv: 1207.4708.

[6] *Bridge Pattern*. In: *Wikipedia*. June 27, 2022. URL: https://en.wikipedia.org/w/index.php?title=Bridge_pattern&oldid=1095365747 (visited on 07/22/2022).

[7] Greg Brockman et al. "OpenAI Gym". June 5, 2016. DOI: 10.48550/arXiv.1606.01540. arXiv: 1606.01540 [cs].

[8] Albin Cassirer et al. *Reverb: A Framework For Experience Replay*. Feb. 9, 2021. arXiv: 2102.04736 [cs]. URL: http://arxiv.org/abs/2102.04736 (visited on 07/30/2022).

[9] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 4.1 (1 2008), pp. 216–217. ISSN: 2334-0924. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/18700 (visited on 09/03/2022).

[10] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Computers and Games*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers. Red. by David Hutchison et al. Vol. 4630. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83. ISBN: 978-3-540-75537-1 978-3-540-75538-8. DOI: 10.1007/978-3-540-75538-8_7.

[11] Ivo Danihelka et al. "Policy Improvement by Planning with Gumbel". In: International Conference on Learning Representations. Mar. 4, 2022. URL: https://openreview.net/forum?id=bERaNdoegnO (visited on 09/03/2022).

[12] Joery A. de Vries et al. "Visualizing MuZero Models". Mar. 3, 2021. arXiv: 2102.12924 [cs, stat]. URL: http://arxiv.org/abs/2102.12924 (visited on 10/28/2021).

[13] Werner Duvaud and Aurèle Hainaut. *MuZero General*. July 21, 2022. URL: https://github.com/werner-duvaud/muzero-general (visited on 07/22/2022).

[14] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *Proceedings of the 35th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, July 3, 2018, pp. 1407–1416. URL: https://proceedings.mlr.press/v80/espeholt18a.html (visited on 09/03/2022).

[15] Lasse Espeholt et al. "SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference". Feb. 11, 2020. arXiv: 1910.06591 [cs, stat]. URL: http://arxiv.org/abs/1910.06591 (visited on 09/18/2021).

[16] Roy Frostig, Matthew James Johnson, and Chris Leary. "Compiling Machine Learning Programs via High-Level Tracing". In: *Systems for Machine Learning* 4.9 (2018), p. 3.

[17] *Futures and Promises*. In: *Wikipedia*. Aug. 2, 2022. URL: https://en.wikipedia.org/w/index.php?title=Futures_and_promises&oldid=1101913451 (visited on 08/11/2022).

[18] Sylvain Gelly et al. *Modification of UCT with Patterns in Monte-Carlo Go*. [Research Report] RR-6062. INRIA, 2006. URL: https://hal.inria.fr/inria-00117266 (visited on 09/03/2022).

[19] *General Game Playing*. URL: http://www.ggp.org/docs/BeginnersGuide.html (visited on 09/27/2022).

[20] Jessica B. Hamrick et al. "On the Role of Planning in Model-Based Deep Reinforcement Learning". In: International Conference on Learning Representations. Feb. 10, 2022. URL: https://openreview.net/forum?id=IrM64DGB21 (visited on 09/03/2022).

[21] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107. ISSN: 2168-2887. DOI: 10.1109/TSSC.1968.300136.

[22] Hado Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems*. Vol. 23. Curran Associates, Inc., 2010. URL: https://proceedings.neurips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html (visited on 07/24/2022).

[23] Hado van Hasselt, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning". In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, Feb. 12, 2016, pp. 2094–2100.

[24] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Las Vegas, NV, USA: IEEE, June 2016, pp. 770–778. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90.

[25] Tom Hennigan et al. *Haiku: Sonnet for JAX*. Version 0.0.3. DeepMind, 2020. URL: http://github.com/deepmind/dm-haiku.

[26] Matteo Hessel et al. *Podracer Architectures for Scalable Reinforcement Learning*. Apr. 13, 2021. arXiv: 2104.06272 [cs]. URL: http://arxiv.org/abs/2104.06272 (visited on 07/19/2022).

[27] Matt Hoffman et al. "Acme: A Research Framework for Distributed Reinforcement Learning". June 1, 2020. arXiv: 2006.00979 [cs]. URL: http://arxiv.org/abs/2006.00979 (visited on 05/22/2021).

[28] Dan Horgan et al. "Distributed Prioritized Experience Replay". In: International Conference on Learning Representations. Feb. 10, 2022. URL: https://openreview.net/forum?id=H1Dy---0Z (visited on 09/03/2022).

[29] Thomas Hubert et al. "Learning and Planning in Complex Action Spaces". In: *Proceedings of the 38th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, July 1, 2021, pp. 4476–4486. URL: https://proceedings.mlr.press/v139/hubert21a.html (visited on 09/03/2022).

[30] Ivo Danihelka. *Mctx: MCTS-in-JAX*. DeepMind, July 30, 2022. URL: https://github.com/deepmind/mctx (visited on 08/06/2022).

[31] James Bradbury et al. *JAX: Composable Transformations of Python+NumPy Programs*. Version 0.3.16. Google, 2018. URL: https://github.com/google/jax (visited on 07/22/2022).

[32] Steven Kapturowski et al. "Recurrent Experience Replay in Distributed Reinforcement Learning". In: International Conference on Learning Representations. Feb. 10, 2022. URL: https://openreview.net/forum?id=r1lyTjAqYX (visited on 09/03/2022).

[33] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Version 9. Jan. 29, 2017. arXiv: 1412.6980 [cs]. URL: http://arxiv.org/abs/1412.6980 (visited on 08/05/2022).

[34] Donald E. Knuth and Ronald W. Moore. "An Analysis of Alpha-Beta Pruning". In: *Artificial Intelligence* 6.4 (Dec. 1, 1975), pp. 293–326. ISSN: 0004-3702. DOI: 10.1016/0004-3702(75)90019-3.

[35] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. "Siamese Neural Networks for One-shot Image Recognition". In: *ICML deep learning workshop* 2 (2015), p. 1.

[36] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *Machine Learning: ECML 2006*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Red. by David Hutchison et al. Vol. 4212. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. ISBN: 978-3-540-45375-8 978-3-540-46056-5. URL: http://link.springer.com/10.1007/11871842_29 (visited on 04/05/2021).

[37] Richard E. Korf. "Real-Time Heuristic Search". In: *Artificial Intelligence* 42.2 (Mar. 1, 1990), pp. 189–211. ISSN: 0004-3702. DOI: 10.1016/0004-3702(90)90054-4.

[38] Marc Lanctot et al. "OpenSpiel: A Framework for Reinforcement Learning in Games". Sept. 26, 2020. DOI: 10.48550/arXiv.1908.09453. arXiv: 1908.09453 [cs].

[39] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". In: *Proceedings of the 35th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, July 3, 2018, pp. 3053–3062. URL: https://proceedings.mlr.press/v80/liang18b.html (visited on 09/03/2022).

[40] Pantelis Linardatos, Vasilis Papastefanopoulos, and Sotiris Kotsiantis. "Explainable AI: A Review of Machine Learning Interpretability Methods". In: *Entropy* 23.1 (1 Jan. 2021), p. 18. ISSN: 1099-4300. DOI: 10.3390/e23010018.

[41] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data Using T-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. ISSN: 1533-7928. URL: http://jmlr.org/papers/v9/vandermaaten08a.html (visited on 09/03/2022).

[42] Matteo Hessel et al. *Optax: Composable Gradient Transformation and Optimisation, in JAX!* Version 0.1.3. DeepMind, 2020. URL: https://github.com/deepmind/optax (visited on 08/05/2022).

[43] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, June 11, 2016, pp. 1928–1937. URL: https://proceedings.mlr.press/v48/mniha16.html (visited on 09/03/2022).

[44] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *NIPS Deep Learning Workshop*. Dec. 19, 2013. arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602 (visited on 12/01/2020).

[45] *Mock Object*. In: *Wikipedia*. Oct. 12, 2021. URL: https://en.wikipedia.org/w/index.php?title=Mock_object&oldid=1049556133 (visited on 07/23/2022).

[46] *Monte Carlo Casino*. In: *Wikipedia*. May 28, 2022. URL: https://en.wikipedia.org/w/index.php?title=Monte_Carlo_Casino&oldid=1090263293 (visited on 06/03/2022).

[47] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging {AI} Applications". In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018, pp. 561–577. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/moritz (visited on 09/03/2022).

[48] Sherjil Ozair et al. "Vector Quantized Models for Planning". In: *Proceedings of the 38th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, July 1, 2021, pp. 8302–8313. URL: https://proceedings.mlr.press/v139/ozair21a.html (visited on 09/03/2022).

[49] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. "On the Difficulty of Training Recurrent Neural Networks". In: *Proceedings of the 30th International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, May 26, 2013, pp. 1310–1318. URL: https://proceedings.mlr.press/v28/pascanu13.html (visited on 09/03/2022).

[50] Tobias Pohlen et al. "Observe and Look Further: Achieving Consistent Performance on Atari". May 29, 2018. DOI: 10.48550/arXiv.1805.11593. arXiv: 1805.11593 [cs, stat].

[51] Robert Tjarko Lange. *Gymnax: A JAX-based Reinforcement Learning Environment Library*. Version 0.0.4. 2022. URL: https://github.com/RobertTLange/gymnax (visited on 07/29/2022).

[52] Christopher D. Rosin. "Multi-Armed Bandits with Episode Context". In: *Annals of Mathematics and Artificial Intelligence* 61.3 (Mar. 2011), pp. 203–230. ISSN: 1012-2443, 1573-7470. DOI: 10.1007/s10472-011-9258-6.

[53] Jonathan Roy. *Fresh Max_lcb_root Experiments · Issue #2282 · Leela-Zero/Leela-Zero*. GitHub. 2019. URL: https://github.com/leela-zero/leela-zero/issues/2282 (visited on 06/15/2022).

[54] J. Schaeffer. "The History Heuristic and Alpha-Beta Search Enhancements in Practice". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 11.11 (Nov./1989), pp. 1203–1212. ISSN: 01628828. DOI: 10.1109/34.42858.

[55] Tom Schaul et al. "Prioritized Experience Replay". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1511.05952.

[56] Julian Schrittwieser et al. "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model". In: *Nature* 588.7839 (Dec. 24, 2020), pp. 604–609. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-020-03051-4.

[57] Julian Schrittwieser et al. "Online and Offline Reinforcement Learning by Planning with a Learned Model". In: Advances in Neural Information Processing Systems. Oct. 25, 2021. URL: https://openreview.net/forum?id=HKtsGW-lNbw (visited on 08/02/2022).

[58] John Schulman et al. "Proximal Policy Optimization Algorithms". Aug. 28, 2017. arXiv: 1707.06347 [cs]. URL: http://arxiv.org/abs/1707.06347 (visited on 06/06/2021).

[59] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. Dec. 5, 2017. DOI: 10.48550/arXiv.1712.01815. arXiv: 1712.01815 [cs].

[60] David Silver et al. "Mastering the Game of Go without Human Knowledge". In: *Nature* 550.7676 (7676 Oct. 2017), pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270.

[61] *Stella: "A Multi-Platform Atari 2600 VCS Emulator"*. URL: https://stella-emu.github.io/ (visited on 07/29/2022).

[62] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018. 526 pp. ISBN: 978-0-262-03924-6.

[63] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: *Proceedings of The 33rd International Conference on Machine Learning*. International Conference on Machine Learning. PMLR, June 11, 2016, pp. 1995–2003. URL: https://proceedings.mlr.press/v48/wangf16.html (visited on 09/03/2022).

[64] David J. Wu. "Accelerating Self-Play Learning in Go". Nov. 9, 2020. arXiv: 1902.10565 [cs, stat]. URL: http://arxiv.org/abs/1902.10565 (visited on 06/15/2022).

[65] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-63518-7 978-3-319-63519-4. DOI: 10.1007/978-3-319-63519-4.

[66] Weirui Ye et al. "Mastering Atari Games with Limited Data". In: *Advances in Neural Information Processing Systems*. Vol. 34. Curran Associates, Inc., 2021, pp. 25476–25488. URL: https://proceedings.neurips.cc/paper/2021/hash/d5eca8dc3820cad9fe56a3bafda65ca1-Abstract.html (visited on 09/03/2022).

[67] Kenny Young and Tian Tian. "MinAtar: An Atari-Inspired Testbed for Thorough and Reproducible Reinforcement Learning Experiments". June 6, 2019. DOI: 10.48550/arXiv.1903.03176. arXiv: 1903.03176 [cs].