Todo list

Contents

1 Summary of Notation

s	state
a	action
r	reward
t	timestep
T	terminal timestep
o	partially observable environment frame
γ	discount
x	hidden state
G^N	N-step return
δ	TD-error
V	value function
Q	state-action value function
$\mathcal{A}^e$	environment action space
$\mathcal{A}^a$	agent action space, $\lvert\mathcal{A}^a\rvert = \lvert\mathcal{A}^e\rvert + 1$
$\mathcal{S}$	state space
$\mathcal{O}$	observation space
$\mathcal{T}_t$	step sample
$\mathcal{T}$	trajectory sample
$\mathcal{L}$	loss function
B	batch size
H	height
W	width
C_e	environment channels
C_h	history channels
C_x	hidden space channels
K	number of unrolled steps
L	history length
N	bootstrap length for N-step return
A	dimension of action
Z	dimension of scalar transformation support

2 Introduction

8 - 10 pages of introduction

Deep Learning (DL) is a branch of **Artificial Intelligence (AI)** that emphasizes the use of neural networks to fit the inputs and outputs of a dataset. The training of a neural network is done by computing the gradients of the loss function with respect to the weights and biases of the network. A better trained neural network can better approximate the function that maps the inputs to the outputs of the dataset.

Reinforcement Learning (RL) is a branch of AI that emphasizes on solving problems through trials and errors with delayed rewards. RL had most success in the domain of **Game Playing**: making agents that could play boardgames, Atari games, or other types of games. An extension to Game Playing is **General Game Playing (GGP)**, with the goal of designing agents that could play any type of game without having much prior knowledge of the games.

discuss policy

Deep Reinforcement Learning (DRL) is a rising branch that combines DL and RL techniques to solve problems. In a DRL system, RL usually defines the backbone structure of the algorithm especially the Agent-Environment interface. On the other hand, DL is responsible for approximating specific functions by using the generated data.

Planning refers to any computational process that analyzes a sequence of generated actions and their consequences in the environment. In the RL notation, planning specifically means the use of a model to improve a policy.

A **Distributed System** is a computer system that uses multiple processes with various purposes to complete tasks.

2.1 Contribution

In this thesis we present the project **MooZi**, a general game playing system that uses a learned model to play both boardgames and Atari games efficiently. More specifically, this project provides

- a collection of environment bridges that connect the system to various environments using a unified interface
- a neural network model that learns representation and could be used for planning
- a MCTS based planner that uses the learned model to perform planning.
- a distributed training system that efficiently trains the agent.
- a thesis with emperical studies and analysis

3 Literature Review

4 - 5 pages

3.1 Planning and Search

3.1.1 Introduction

Many AI problems can be reduced to a search problem [44, p.39]. Such search problems could 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 term **planning** and **search** are widely used across different domains, especially in AI, and are sometimes interchangable. Here we adopt the definition by Sutton and Barto [41].

s?

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 could view the planning process as an operator $\mathcal{I}$ that takes the policy as input and outputs an improved policy $\mathcal{I}\pi$.

Planning methods could be categorized into types based on the focus of the target state s 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 $S' \subset \mathcal{S}$:

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

Typical background planning methods include **dynamic programming** and **Dyna-Q**. 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 random subset of the state space is selected for update.

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

in Rich's book S_t is the specific state at time t, and s could be any state

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

We could also blend both types of planning. Algorithms such as AlphaGo use both types of planning when they 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. At the same time, the neural network is trained on past experience and the policy for all states is updated. The updates from this background planning are applied when the planner uses the latest weights of the neural network.

An early example of the use of search as a planning method is the **A*** algorithm. In 1968, Hart, Nilsson, and Raphael designed the A* algorithm for finding shortest path from a start vertex to a target vertex [15]. 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* does not yield an optimal solution under stochastic environments and it could be computationally infeasible on problems with high branching factors. More sophisticated search algorithms were developed to cater to the growing complexity of use cases.

In 1990, Korf noticed the problem of unbounded computation in the search algorithms at the time. Algorithms like A* could consume much more memory and spend much more time in certain states. This undesirable trait makes these algorithms difficult to apply to real-time 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. He also developed the **Real-Time-A*** algorithm as a modified version of A* with bounded computation [26].

Monte Carlo techniques were adopted to handle complex environments. Tree-based search algorithms such as **MiniMax** and **Alpha-Beta Pruning** were designed to play and solve two-player games. find the right citation

3.2 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 [32]. Therefore, **Monte Carlo (MC)** methods gained their name as a class of algorithms based on random samplings.

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 intermidate game states. One way of obtaining an evaluation on a state is by applying a heursitic 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 randomly. This evaluation process is called **random rollout** or **Monte Carlo rollout**.

3.3 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 bandits [25]. Rémi Coulom developed the general idea of **Monte-Carlo Tree Search** that combines both Monte-Carlo rollouts and tree search [6] for his Go program CrazyStone. Shortly afterwards, Gelly et al. implemented another Go program MoGo, that uses the UCT selection formula [13]. MCTS was generalized by Chaslot et al. as a framework for game AI [5]. This framework requires less domain knowledge than classic approaches to game AI while others giving better results. The core idea of this framework is to gradually build the search tree by iteratively applying four steps: **selction**, **expansion**, **evaluation**, and **backpropagation**. The search tree built in this way emphasizes 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, annd rollout results are aggregated to yield more accurate value. Here we detail the four steps in the MCTS framework by Chaslot et al. (see Figure 1).

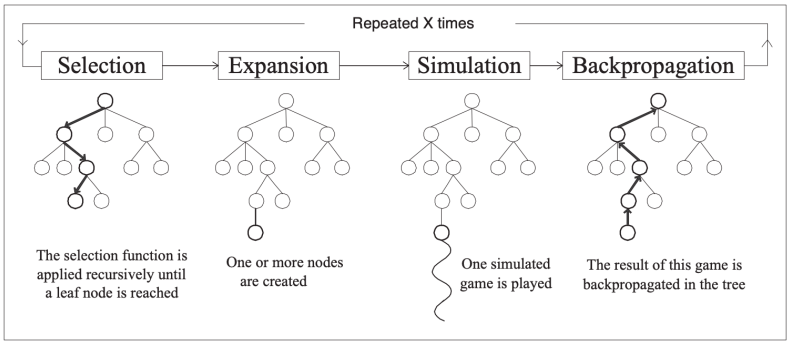

Figure 1: The Monte-Carlo Tree Search Framework

3.3.1 Selection

Thes 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 is based on the evaluation function E, and the exploration bonus B. For edges of a parent state $(s, a),\ a \in \mathcal{A}$, the selection $I(s)$ is based on

$$I(s) = \text{argmax}_{a \in \mathcal{A}} \left(E(s, a) + B(s, a) \right) \tag{1}$$

The prior score could 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 is usually based on the visit count of the child and the parent. The more visits a child gets, the less the exploration bonus will be. For example, the selection in the UCT algorithm is based on

$$\begin{aligned} I(s) &= \text{argmax}_{a \in \mathcal{A}} \left(E(s, a) + B(s, a) \right) \\ E(s, a) &= \frac{V(s)}{N(s, a)} \\ B(s, a) &= \sqrt{\frac{2 * \log(n_b)}{n_c}} \end{aligned}$$

where v_c is the value of the node, n_b and n_c are the visit counts of the parent and child, respectively. This Gelly et al. used this selection rule in their implementation of MoGo, the first computer Go program that uses UCT [13].

3.3.2 Expansion

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

3.3.3 Evaluation

The expanded node is evaluated by playing a game with a rollout policy, using an evaluation function, or using a blend of both approaches. Many MCTS algorithms use a random policy as the rollout policy and the game result as the evaluation. Early work on evaluation functions focused on hand-crafted heuristic functions based on experted knowledge. More recently, evaluation functions are mostly approximated by deep neural networks specifically trained for the problems.

add two examples here

I'm not sure if using bullets points is appropriate. I

3.3.4 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.

3.3.5 MCTS Iteration and Move Selection

The four steps are repeated until the budget runs out. 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 highest lower bound [35].

3.4 AlphaGo

In 2017, Silver et al. developed **AlphaGo**, the first Go program that beats a human Go champion on even terms [39]. 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. Given a state s as the input, this 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:

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

I'm not sure if I should use past tense. I think I am describing how AlphaGo works as an algorithm (which I think is timeless), not the acutal AlphaGo software that beats Lee.

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 so that $p_\rho = p_\sigma$. 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:

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

For the last stage, 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 similar to that described in 3.3. 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 select phase of each simulation, an action is selected for each traversed node using the same base formula (1). In AlphaGo, the exploitation score of the selection formula is the estimated value of the next state after taking the actions, namely $Q(s, a)$. The exploration bonus of edge (s, a) is based on the prior probability and decays as its visit count grows.

$$u(s, a) \propto \frac{P(s, a)}{1 + N(s, a)} \tag{2}$$

The action taken at time t maximizes the sum of the exploitation score and the exploration bonus

$$a_t = \underset{a}{\operatorname{argmax}} \left(Q\left(s_t, a\right) + u\left(s_t, a\right) \right) \tag{3}$$

AlphaGo evalutes 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 λ is used to balance these two types of evaluations into the final evaluation $V_(s_L)$

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

3.5 AlphaGo Zero

AlphaGo Zero is a successor of AlphaGo that beat AlphaGo 100-0 [39]. The first step in the AlphaGo machine learning pipeline is to learn from human expert moves. In constrast, AlphaGo Zero learns to play Go *tabula rasa*. This means it learns solely by reinforcement learning from self-play, starting from random play, without supervision from human data.

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

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

. To generate self-play games $s_1, ..., s_T$, an 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

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

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

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

where $(z - v)^2$ is the mean squared error regressing on value prediction, $-\boldsymbol{\pi}^{\mathrm{T}} \log \boldsymbol{p}$ is the cross-entropy loss over move probabilities, and $c\|\theta\|^2$ is the L2 weight regularization.

I want to say everything else I don't mention here is pretty much like AlphaGo, how do I say that?

3.6 AlphaZero

AlphaZero reduces game specific knowledge of AlphaGo Zero so that the same algorithm could be also applied to Shogi and chess [38]. One difference between AlphaZero and AlphaGo Zero is that AlphaZero the game result is no longer either winning or losing ($z \in \{-1, +1\}$), but also could be a draw ($z \in \{-1, 0, +1\}$). This adaptation takes account of games like chess have a draw condition.

The AlphaGo Zero I described above has little Go-specific information already. I don't know how to put it here.

3.7 MuZero

In 2020, Schrittwieser et al. developed **MuZero**, an algorithm that learns to play Atari, Go, chess and Shogi at superhuman level. Compared to the AlphaGo faimily algorithms, MuZero has less game specific knowledge and has no access to a perfect model. MuZero plans with a neural network that learns the game dynamics through experience. Since MuZero does not make the assumption of having access to a perfect model, MuZero could be applied to games where either the perfect model is not known or is infeasible to compute with.

MuZero's model has three main functions. The **representation function** h encodes a history of observations $o_1, o_2, ..., o_t$ into a hidden state s_t^0. The **dynamics function** g, given a hidden state s^k and action a^k, produces an immediate reward r^k and the next hidden state s^{k+1}. The **prediction function** f, given a hidden state s^k, produces a probability distribution p^k of actions and a value associated to that hidden state v^k. These functions are approximated jointly in a neural network with weights θ

$$x_t^0 = h_\theta(o_1, o_2, ..., o_t) \tag{4}$$
$$(x^{k+1}, r^{k+1}) = g_\theta(x^k, a^k) \tag{5}$$
$$(v^k, \boldsymbol{p}^k) = f_\theta(x^k) \tag{6}$$

MuZero plans with a search method based on the MCTS framework (discussed in 3.3). 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 s as a learned representation of the board state. The transition is no longer made by the perfect model but 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 3.3. At each timestep t, s_t^0 is created using (4). A variant of PUCT is used to select an action during the search

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

where c_1 and c_2 are two constants that adjust the exploration bonus. The selected edge (s^k, a^k) at depth k is expanded using (5) and evaluated using (6). At the end of the simulation, the statistics of the nodes along the search path are updated.

Notice since the transitions of the nodes 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 (s^0, a^0) of the root node is selected as the action to take in the environment.

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

$$\mathcal{L}_t(\theta) = \underbrace{\sum_{k=0}^{K} l^{\mathrm{p}}\left(\pi_{t+k}, p_t^k\right)}_{①} + \underbrace{\sum_{k=0}^{K} l^{\mathrm{v}}\left(z_{t+k}, v_t^k\right)}_{②} + \underbrace{\sum_{k=1}^{K} l^{\mathrm{r}}\left(u_{t+k}, r_t^k\right)}_{③} + \underbrace{c\|\theta\|^2}_{④} \tag{7}$$

where K is the number of rollout depth, ① is the loss of the predicted prior move probabilities and move probabilities imporved 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 L2 regularization.

MuZero Reanalyze is also used to generate training targets in addition to those generated through game play. MuZero Reanalyze re-executes MCTS on old games using the latest parameters and generates new training targets with potentially improved policy.

3.8 Atari Games Playing

3.8.1 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 were sold over its 15 years on market [1]. The most popular game, PacMan, was 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) [40]. Stella was ported to popular operating systems such as Linux, MacOS, and Windows, providing Atari 2600 experiences to users without owning physical copies of the equipment. In **ArcadeLearningEnvironment˙Bellemare.Naddaf.ea˙2013b**, **ArcadeLearningEnvironment˙Bellemare.Naddaf.ea˙2013b** introduced the **Arcade Learning Environment (ALE)** [**ArcadeLearningEnvironment˙Bellemare.Naddaf.ea˙2013b**]. ALE provided interfaces of over a hunderd of Atari game environments using Stella as the backend. Each ALE environment had specifications on visual representation, action space, and reward signals. This made ALE environments suitable for machine learning research, as data were well-represented and evaluation metrics were clearly defined. Moreover, ALE environments were diverse in their characteristics: while some environments required more mechanical mastery of the agent, others required more long-term planning. This made solving multiple ALE environments using the same algorithm a good general game playing problem (also see section 2).

3.8.2 Deep Q-Networks

Mnih et al. pioneered the study of using deep neural networks to learn in ALE envrionments [30]. 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 had a neural network that approximates the Q function, parametrized by weights θ, denoted Q_θ. Experiences were 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 partially observable frame provided by the environment at timestep t (also see section 5.2.6). Generated experience were stored in an experience replay represented by a FIFO queue. For each training step, a batch of uniformly sampled experience was drawn from the experience replay, and the loss was computed using

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

where θ' were network parameters updated less frequently than θ.

3.8.3 Double Q Learning

Hasselt analyzed the overestimation problem of Q values in Q-learning and developed the **double Q learning**. Central to double Q-learning was the double Q update that replaced the traditional Q update [16]. Double Q learning reduced the overestimation problem by introducing an additional Q estimator and updating two estimator 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 were two different Q estimators updated alternatively. Hasselt, Guez, and Silver followed up by applying the double Q learning in DQN [17]. Similar to the double Q update above, double Q update for neural networks was formulated as

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

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

3.8.4 Experience Replay

Schaul et al. studied the role of experience reply in DQN and developed the **prioritized experience replay** [36]. In the original work of DQN, all samples were drawn from the experience replay uniformly. In a prioritized experience

replay, however, samples were drawn according to a distribution based on their calculated priority

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

where $P(i)$ was the probability of of i-th sample being drawn, α was a constant, and p_i was the priority of the sample. Schaul et al. suggested two approaches to compute priorities of samples. The first one was **proportional sampling**, in which the priority p of sample i was calculated by

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

where δ_i was the temporal-difference error of the sample, and ϵ was a small constant to give all samples a probability to be drawn. The second one was **rank-based sampling**, in which the same temporal difference was calculated, but the final priority was based on the rank of the error,

$$\begin{aligned} \text{score}(i) &= |\delta_i| + \epsilon \\ \text{rank}(i) &= \text{index } i \text{ of} \operatorname{argsort}(-\text{score}(j) \text{ for } j \text{ in all samples}) \\ p_i &= \frac{1}{\text{rank}(i)} \end{aligned}$$

I really have no clue how to express the rank function here...

Horgan et al. followed up by implementing a distributed version of the prioritized experience replay [22]. **RECURRENTEXPERIENCEREPLAY˙Kapturowski.Ostrovski.ea˙2019a** investigated the challenges of using experience replaies for RNN-based agents and developed **Recurrent Replay Distributed DQN**.

3.8.5 Network Architectures

Wang et al. studied an alternative neural network architecture for ALE learning [42]. They developed the **Dueling Q-network**, that while retaining the inputs and outputs specifications of the Q-network used in DQN, structually represented the learning of the advantage function $A(s, a)$ defined as

$$A(s, a) = Q(s, a) - V(s)$$

The Q-network was parametrized by θ that could be futher divided into three parts: θ^{trunk}, the shared trunk of the network; θ^A, the advantage head; and θ^{V}, the value head. The network approximated the value function internally through the shared trunk and the value head, denoted $V_{\theta^{\text{trunk}},V}$, and the advantage function, denoted $A_{\theta^{\text{trunk}},A}$. The values computed by the two heads were combined to form the Q-value as follows

$$Q_{\theta^{\text{trunk, V, A}}}(s, a) = V_{\theta^{\text{trunk, V}}}(s) + \left(A_{\theta^{\text{trunk,A}}}(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A_{\theta^{\text{trunk},A}}(s, a') \right)$$

Similar to DQN, the dueling Q-network was trained through fitting to emperical data generated by interacting with the environment. Experiments showed that this architecture encouraged the network to learn to differentiate between the values of states and the values of state-action pairs, and led to better performance of the agent in general.

3.8.6 Scalar Transformation

Pohlen et al. introduced a few enhancements to achieve stable training in Atari games [34]. We will focus on the **transformed Bellman Operator** since MuZero and our project used a similar transformation operator. For different Atari games, reward signals could vary drastically both in density and scale. This led to high variance in training targets during training of the algorithms, causing algorithms to have difficulty converging. Mnih et al. cliped the reward signal to a range of $[-1, 1]$ to reduce such variance in DQN [30]. However, this clipping discarded information regarding the scale of rewards and consequently changed the set of optimal policies. The transformed Bellman Operator was developed to address this problem. The Q update of the new operator transformed scalar rewards as

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

where ϕ is an invertible transformation that satisfying a set of constraints. One example of such transformation is

$$\phi(x) = \text{sign}(x)\left(\sqrt{|x|+1} - 1\right) + \varepsilon x$$

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

This transformation was also used in MuZero to transform both value targets and reward targets.

3.8.7 Efficiency Problems with ALE

ALE environments use Atari 2600 emulator Stella as backend, producing environment frames by running the emulator is much more expensive than environments such as board games. For algorithms like DQN, most of the walltime spent on the system would be the environment stepping time since performing neural network inferences could be much faster. Additionally, neural network inferences could be batched and computed using specialized hardwares such as GPUs and TPUs. ALE environments, on the other hand, are CPU-only and linearly increasing number of environments always also linearly increase their memory and CPU cycle consumptions. Data hungry algorithms such as MuZero were trained with 20 billion environment frames, 1 million of training steps, and lasted more than 12 hours. It is difficult for researchers with limited computation resources to produce such work, let alone extending it.

3.8.8 MinAtar

MinAtar, developed by Young and Tian, is an open-source project that offers RL environments inspired by ALE. MinAtar offers five environments that pose similar challenges to ALE environments: one is the challenge of learning representation from raw pixels, the other one is the problem of learning behaviors that associate actions and delayed rewards. However, MinAtar environments are implemented in pure Python, have simpler environment dynamics, and are visually less rich than ALE environments. This makes MinAtar environments perfect test environments for researchers live on shoestring.

3.8.9 Consistency Loss

One interesting characteristic of Atari-like games is that environment frames are usually temporal consistent. For example, given the position of the player avatar for the last few frames, it would not be difficult to infer the rough position of the avatar in the next frame. Algorithms could take advantage of this property, and 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 [8]. As MuZero applies recurrent inferences to a hidden state, the transitions could be traced as a trajectory in the 3D space. They demonstrated two approaches to enforce consistency among inferences, and through the enforcement, transitions are more consistent (i.e., the trajectory in the 3D space is "smoother") and performance is better. Ye et al. drew inspiration from the Siamese neural networks and used a project-predict structure to enforce consistency [45, 24].

[20]

3.9 Deep Reinforcement Learning Systems

A key challenge in deep learning is to train deep learning systems efficiently. Deep reinforcement learning systems involve more irregular computation patterns, making designing such systems efficiently a greater challenge. Decisions the designer has to make including but not limited to (1) Where and how to generate experiences? (2) Where and how to store generated experiences? (3) Where to store the model and who have copies of it? (4) Where is gradient computation carried out? (4) How to orchestrate processes for stable training? Here we briefly review popular deep reinforcement learning system designs that utilize parallelization to achieve faster and more efficient training.

Mnih et al. selected four popular RL algorithms and developed asynchronous variants for them with the a parallelization structure using actor-learner processes. Each actor-learner process holds a delayed copy of the model, generates experiences locally using the model, accumulates gradients locally, and updates the global model once in a while by updating the global model using the locally accumulated gradients. Among the asynchronous variants, **Asynchronous Advantage Actor Critic (A3C)** had the best performance and achieved the state-of-the-art at the time using half the training time.

Espeholt et al. developed **IMPALA**, a scalable distributed deep reinforcement learning agent [11]. 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 a local storage and subsequently pushed into 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 perform batched forward pass and back-propagation using its model. Figure 2 shows two variants of this structure, one uses a single learner while the other one uses multiple learners.

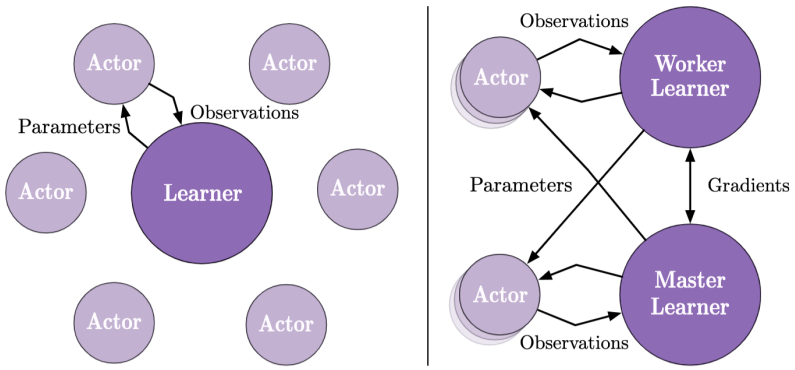

Figure 2: Left: a single learner computes all gradients; Right: multiple worker learners compute gradients and one master learner collects and aggregates gradients. Source: [11]

Espeholt et al. developed **Scalable, Efficient Deep-RL (SEED)** architecture to effectively utilize accelerators using a centralized inference server [12]. Similar to IMPALA, SEED also uses two main types of workers: actors and learners. However, the main difference between SEED and IMPALA is that the 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 store trajectories as in IMPALA, but also has a batching layer that batches actor queries and efficiently perform batched inference with the model. Since actors no longer need to pull neural network parameters from the learner, IO overhead from serializing and messaging parameters is eliminated. More over, since the learner batches queries from all actors, IO overhead from moving inputs and outputs to accelerators is also reduced and thus increasing the overal inferencing throughput. One downside of the the SEED architecture is that actors have to wait for reponse from the learner to take an action, and thus have a higher latency for taking a step.

Hoffman et al. developed the **Acme** research framework. Acme organizes its system similar to IMPALA: processes that interacting with the environment are actors, and processes that collect experiences 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 [4]. Figure 3 shows an example of a distributed asynchronous agent in Acme.

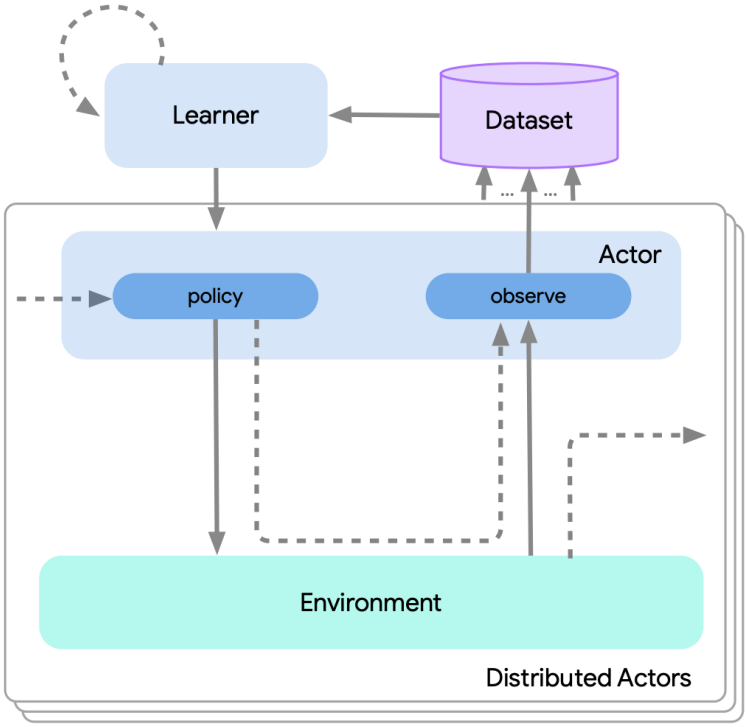

Figure 3: Example of a distributed asynchronous agent with Acme. Source: [21]

Moritz et al. designed and implemented **Ray**, a framework for scalable distributed computing. Ray enables both task-level parallelization and actor-level parallelization through a unified interface. **Ray Core** was designed with AI applications in mind and now it has powerful primitives that distributed AI systems find handy. 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 a separate process. Liang et al. developed the **RLlib**, an industrial-grade deep reinforcement learning library. RLlib builts on top of Ray Core and provids more abstractions that a broad range of DRL systems could make use of. See figure 4 for an illustration of RLlib's abstraction layers. As of the writing of this thesis, RLlib implemented 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 worker process and deploys a hierarchical control paradigm similar to RLlib (discussed in).

add reference here

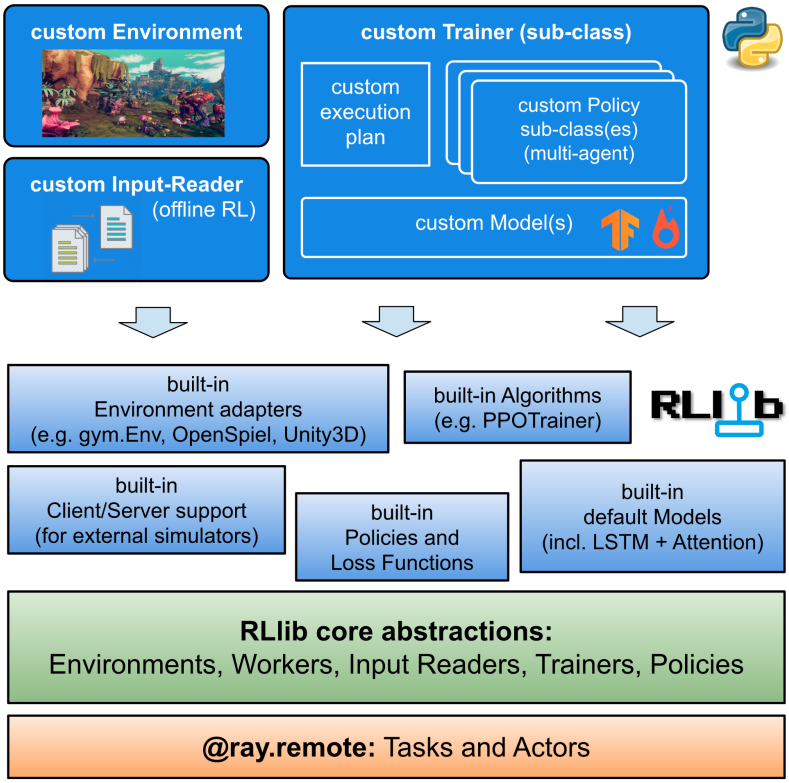

Figure 4: RLlib Abstraction Layers. *Source: [28]*

Hessel et al.

4 Problem Definition

(5 pages)

4.1 Markov Decision Process and Agent-Environment Interface

this section should be in the introduction, since it defines too many things that we need for later.

A RL problem is usually represented as a **Markov Decision Process (MDP)**. MDP is tuple of four elements where $\mathcal{S}$ is a set of states that forms the **state space** $\mathcal{A}$, is a set of actions that forms the **action space**; $P(s, a, s') = Pr[S_{t+1} = s' \mid S_t = s, A_t = a]$ is the **transition probability function**; $R(s, a, s')$ is the **reward function**. To solve a problem formulated as an MDP, we implement the **Agent-Environment Interface** (Figure 5). 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**.

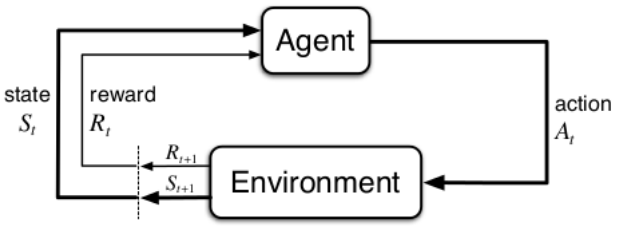

Figure 5: Agent-Environment Interface

my notation is messed up but I will fix them later for sure.

At each state s, then agent takes an action based on its **policy** $\pi(a \mid s)$. This policy represents the conditional probability of the agent taking an action given a state so that $\pi(a \mid s) = Pr[A_t = a \mid S_t = s]$. One way to specify the goal of the agent is to obtain a policy that maximizes the sum of expected reward from any state s

$$E_\pi \left[\sum_{k=0}^{T} R_{t+k+1} \mid S_t = s \right] \tag{8}$$

where E_π denotes the expectation of the agent following policy π. Another way is to also use a discount factor γ so to favor short-term rewards

$$E_\pi \left[\sum_{k=0}^{T} \gamma^t R_{t+k+1} \mid S_t = s \right] \tag{9}$$

Notice that (8) is a special case of (9) where $\gamma = 1$.

Maybe I should use one formula here to unify both.

4.2 Our Approach

(5 pages)

5 Method

(20 - 25 pages)

5.1 Design Philosophy

5.1.1 Use of Pure Functions

One of the most notable difference of MooZi implementation 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 compuations. For example, we use **JAX** and **Haiku** to implement neural network related modules. 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 variables that could be used with the specification to perform a forward pass.

add one or two more examples of using pure functions, also mention how tape in the MooZi is different from the tape in GII

5.1.2 Training Efficiency

One common problem with current open-sourced MuZero projects is their training efficiency. Even for simple environments, these projects could take hours to train.

There are a few major bottlenecks of training efficiency in this type of project. The first one is system parallelization.

The second one is the environment transition speed.

The third one is neural network inferences used in training.

Data needed; I've seen multiple issues on GitHub complaining about the training speed. I once assigned the task of "actually running the project and gather run time data" to Jiuqi but no follow up yet.

5.2 Project Structure

5.2.1 Overview

In MooZi, we use **Ray** library designed by Moritz et al. for orchestrating distributed processes. We also adopt the terminology used by Ray [33]. In a distributed system with **centralized control**, a single process is responsible for operating all other processes. This central process is called the **driver**. Other processes are either **tasks** or **actors** . **Tasks** are stateless functions that takes inputs and return outputs. **Actors** are statefull objects that group several methods that take inputs and return outputs. In 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 [12], [11]. Even though MooZi does not adopt the concept of a RL actor, we will use the term **Ray task** and **Ray actor** to avoid confusion. In contrast to distributed systems with **distributed control**, ray tasks and ray actors are reactive and do not have busy loops. The driver process when a ray task or ray actor is activated and what data should be used as inputs and where the outputs should go. In other words, the driver process orchestrates the data and control flow of the entire system, and ray tasks and ray actors merely response to instructions, processing inputs and returning outputs on-command.

include data from previous presentation to make the point

boardgames are fine, Atari games are slow, but in either cases we can't control

We illustrate MooZi's architecture design in Figure (6). There are six types of processes. The **driver** is the entrance of the program and is responsible for setting up configurations, spawning other processes as ray actors, and managing data flow among the ray actors. The **parameter optimizer** stores the latest copy of the network weights and performs batched updates to the weights. The **replay buffer** stores generated trajectories and process the trajectories into training targets. A

MCTS inference batching is slow due to IO overhead, include data here from previous presentation to make the point

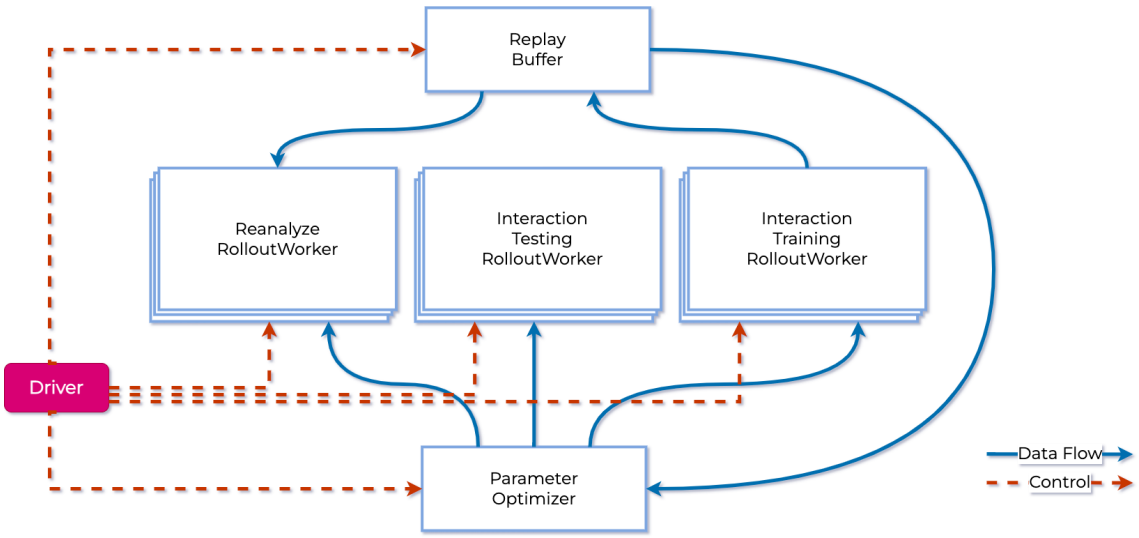

Figure 6: MooZi Architecture

interaction training rollout worker is a ray actor that responsible for generating experiences by interacting with the environment. A **interaction testing rollout worker** is a ray actor that responsible for evaluating the system (e.g., average episode return) by interacting with the environment. A **reanalyze rollout worker** is a ray actor that pulls history trajectories and updates their search statistics using the latest network weights.

5.2.2 Environment Bridges

Environment bridges unified environments defined in different libraries into a unified interface. In the software engineering nomenclature, environment bridges follow the **bridge design pattern** [2]. More specifically, in our project we implemented environment bridges for three types of environments that are commonly used in RL research including OpenAI Gym, OpenSpiel, and MinAtar [3, 27, 46]. The bridges first wrapped these environments into the **The DeepMind RL Environment API** [9]. In this format, each environment step outputs a four tuple. (1) **step_type**: an enumerated value indicates the type of the timestep, one of `first`, the first step in the environment, `mid`, intermidate steps, and `last`, the last step in the environment. (2) **reward**, a floating point value indicates the reward given by the environment by taking the last given action in the environment. (3) **discount**, a floating point value indicates the discount associated with the step. (4) **observation**, a data structure represents the new environment observation, could be an N-dimensional array in visual-only observation games, or a nested structure in boardgames where other information such as the next player are represented. We wrapped these environments again to produce a flat dictionary that the rest components in MooZi used.

The final wrapper environment have the same signature as follows:

- Inputs

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

  a_t: An integer indicates the last action taken by the agent.

- Outputs

o_t: An N-dimensional array that represents the observation of the current timestep. In the shape of (H, W, C).

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

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

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

$m_t^{A_a}$: A bit mask of legal action indices. Valid action indices of 1 and invalid actions have indices of 0. $|\mathcal{A}_\urcorner| + 1 = |\mathcal{A}|$ (see 5.2.4).

All environments were generalized as continuous tasks. This was done by passing an addition input b_t^{last} to the environment stepping argument. For an episodic task, the environment was reset internally when b_t^{last} is `True`. The policy still executed for the last environment step, but the resulting action is not used by the environment. For a continuous task, the environment always step with the lastest action and the b_t^{last} input is ignored. See 1 for pseudo-code of generalizing both types of environments to achieve a unified main loop interface.

Moreover, we also implemented a mock enviornment using the same interface [31]. A mock environment is initialized with a trajectory sample $\mathcal{T}$, and simulates the environment by outputing step samples one at a time. An agent could interact with this mock environment as if it is a real environment. However, the actions taken by the agent would not affect state transitions since they are predetermined by the given trajectory from initialization. This mock environment is used for reanalyze rollout workers described in 5.2.12.

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

# episodic task stepping
def step(env, action, is_last):
    if is_last:
        return env.reset()
    else:
        return env.step(action)

# continuous task stepping
def step(env, action, is_last):
    return env.step(action)
```

Algorithm 1: Environment Adapter Interface

5.2.3 Vectorized Environment

We also implement an environment supervisor that stacks multiple individual environments and forms a single vectorized environment. The resulting vectorized environment takes inputs and produces outputs similar to an 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). Previous scalar outputs such as reward are also stacked into vectors with size of B. Since environment adapters generalizes 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 1. Using vectorized environments increases the communication bandwidth between the environment and agent and facilitates designing an vectorized agent that processes a batched of observations and returns a batch of actions at a time.

The mocked environment described in 5.2.2 was less trivial to vectorize. Each mocked environment has to be initialized with a trajectory sample. To vectorize the mocked environments of size B, at least B trajectories have to be mocked and tracked at the same time. These B trajectories usually have different length and therefore terminate at different timesteps. Once one of the mocked trajectory reaches its termination, another trajectory has to fill the slot. We created a trajectory buffer to address this problem in vectorized mocked environment. When a new trajectory is needed by one of the mocked environment, the buffer supplies a new trajectory to that mocked environment. With the trajectory buffer, the vectorized mocked environment could process batched interactions like a regular vectorized environment until the trajectory buffer runs out of trajectory supply.

5.2.4 Action Space Augmentation

We augmented the action space by adding a dummy action indexed at 0. This dummy action has two major purposes. Firstly, this dummy action is to used construct history observations when the horizon is 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, the current timestep could be 0, which means the agent hasn't taken an action yet. We used zeroed frames with the same shape as history frames, and the augmented dummy action as history actions. Formally, we define

$$\begin{aligned} a_i &= 0 \quad \forall i < 0 \\ a_T &= 0 \end{aligned}$$

Secondly, since MooZi's planner does not have access to a perfect model, it does not know when a terminal state is reached. Node expansions do not stop at terminal states and the tree search could simulate multiple steps beyond legal game states. Search performed in these invalid subtrees not only wastes previous search budget, but also back-propagates value and reward estimates that are not learned from generated experience. We addressed this issue by letting the model to learn a policy that takes the dummy action beyond terminal states. The learned dummy action acts as a switch that, once taken, treats all nodes in its subtree as absorting states and edges that have zero values and rewards respectively.

5.2.5 MooZi Agent

The MCTS in MooZi Planner was similar to that described in 3.3 and 3.7. We used the open-source implementation of MCTS, **MCTX**, by Danihelka et al. [7]. This library was implemented in JAX and supported customized initial inference and

supports?

recurrent inference. We built a simple adapter that connected our neural network interface with MCTX.

5.2.6 History Stacking

In fully observable environments, the state s_t at timestep t observed by the agent entails sufficient information about future state distribution. However, for partially observable environments, this assumption 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. Atari games are usually partially observable environments. In **Deep Q Networks (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 $\pi(a \mid o_{t-3}, o_{t-2}, o_{t-1}, o_t)$. AlphaZero and MuZero not only stacked a history of environment frames, but also a history of past actions. We also adopted this practice in MooZi, and we use the last L environment frames and taken actions so that the signature of the learned model through the policy head is $\pi(a \mid o_{t-L-1}, \ldots, o_t, a_{t-L-2}, \ldots, a_{t-1})$. N is an adjustable configuration of the MooZi system and in fully observable environments we could set this to 1.

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

1. prepare saved L environment frames of shape (L, H, W, C_e)
2. stack the L dimension with the environment channels dimension C, now shape of $(H, W, N * C_e)$
3. prepare saved L past actions of shape (L), represented as action indices
4. one-hot encode the actions, now shape of (L, A)
5. stack the L dimension with the action dimension A, now shape of $(L * A)$
6. divide the action planes by the number of action, shape remains the same
7. tile action planes $(L * A)$ along the H and W dimensions, now shape of $(H, W, N * A)$
8. stack the environments planes and actions planes, now shape of $(H, W, L * (C_e + A))$
9. the history is now represented as an image with height of H, width of W, and $L * (C_e + A)$ channels

To process batched inputs from vectorized environments descirbed in 5.2.3, all operations aboved are performed with an additional batch dimension B, yielding the final output with the shape $(B, H, W, L * (C_e + A))$. We denote the channels of the final stacked history as C_h so that $C_h = N * (C_e + A)$, where the subscript h means the channel dimension for the representation function h.

5.2.7 MooZi Neural Network

neural network specifical should go into experiment section

We used JAX, and Haiku to build the neural network [**CompilingMachineLearning˙Frostig.Johnson.ea˙2018**, 14, 23]. We consulted other open-source projects that use neural networks to play games [10, 45, 43, 43]. We implemented the neural-network model with two different architectures in our project, one is multilayer-perceptron-based and the other one is residual-blocks-based [18]. We primarily used residual-blocks-based model for experiments so we will describe the architecture in full details here.

Similar to MuZero described in section 3.7, the model had the representation function, the dynamics function, and the dynamics function. Additionally, we also trained the MooZi model with a self-consistency loss similar to that described by Ye et al. and de Vries et al. [45, 8]. We used an additional function, named as the **projection function** for this purpose. The learned model was used for two purposes during tree searchs. The first one was to construct the root nodes using the representation function and the prediction function. We call this process the **initial inference**. The second one was to create edges and child nodes for a given node and action using the dynamics function and the prediction function. We call this process the **recurrent inference**.

This terminology is aligned with MuZero's pseudo-code and MCTX's real code

We implemented residual blocks using the same specification as He et al. [18]. One residual block was defined 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 h is parametrized as follows:

- input x of shape (H, W, C_h)
- apply a 2-D padded convolution on x, with kernel size 3 by 3, 32 channels
- apply batch normalization on x
- apply relu activation on x
- apply 6 residual blocks with 32 channels on x
- apply a 2-D padded convolution on x, with kernel size 3 by 3, 32 channels
- apply batch normalization on x

- apply relu activation on x
- output the hidden state head x_s

The prediction function f is parametrized as follows:

- input x
- apply a 2-D padded convolution on x, with kernel size 3 by 3, 32 channels
- apply batch normalization on x
- apply relu activation on x
- apply 1 residual block with 32 channels on x
- flatten x
- apply 1 dense layer with output size of 128 to obtain the value head x_v
- apply batch normalization on x_v
- apply relu activation on x_v
- apply 1 dense layer with output size of Z on x_v
- apply 1 dense layer with output size of 128 to obtain the policy head x_p
- apply batch normalization on x_p
- apply relu activation on x_p
- apply 1 dense layer with output size of A^a on x_p
- output the value head x_v and the policy head x_p

The dynamics function g is parametrized as follows:

- input x of shape $(H, W, 32)$, a as an integer
- encode a as action planes of shape (H, W, A) (described in 5.2.6)
- append a to x
- apply a 2-D padded convolution on x, with kernel size 3 by 3, 32 channels
- apply batch normalization on x
- apply relu activation on x
- apply 1 residual block with 32 channels on x
- apply 1 residual block with 32 channels on x to obtain the hidden state head x_s
- apply a 2-D padded convolution on x_s, with kernel size 3 by 3, 32 channels
- apply batch normalization on x_s

- apply relu activation on x_s
- apply 1 dense layer with output size of 128 on x to obtain the reward head x_r
- apply batch normalization on x_r
- apply relu activation on x_r
- apply 1 dense layer with output size of Z on x_r
- output the hidden state head x_s and the reward head x_r

For convinience, we made the output specification the same for both the initial inference and the recurrent inference. They both produced a tuple of (x, v, r, p), x was the hidden state, v was the value prediction, r was the reward prediction, and p was the policy prediction.

For the initial inference,

- input features $\psi_t = (o_{t-L+1}, \ldots, o_t, a_{t-L}, \ldots, a_{t-1})$
- obtain $\mathbf{x}_t^0 = h(\psi_t)$
- obtain $v_t^0, \mathbf{p}_t^0 = f(\mathbf{x}_t^0)$
- set $r_t^0 = 0$
- return $(\mathbf{x}_t^0, v_t^0, r_t^0, \mathbf{p}_t^0)$

For the recurrent inference,

- input features $\mathbf{x}_t^i, a_t^i$
- obtain $\mathbf{x}_t^{i+1}, r_t^{i+1} = g(s_t^i, a_t^i)$
- obtain $v_t^{i+1}, \mathbf{p}_t^{i+1} = f(\mathbf{x}_t^{i+1})$
- return $(\mathbf{x}_t^{i+1}, v_t^{i+1}, r_t^{i+1}, \mathbf{p}_t^{i+1})$

Moreover, we applied the invertible transformation ϕ described in section 3.8.6 to both the scalar reward targets and scalar value targets to create categorical representations with the same support size. The support we used for the transformation were integers from the interval $[-5, 5]$, with a total size of 11. Scalars were 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 the target of this scalar is 0.7 for the category 1, and 0.3 for the category 2. For training, the value head and the reward head first produced estimations as logits of size Z. These logits were aligned with the scalar targets to produce categorization loss as described in the 5.2.9. For acting, the neural network additionally applied the softmax function to the logits to generated a distribution over the support. The linear combination of the distribution and their corresponding integer values were computed and fed through the inverse of the transformation, namely ϕ^{-1}, to produce scalar values. This means from the perspective of the planner, the scalar estimations made by the model were in same shape and scale as those produced by the environment.

reference an example in the experiments section

5.2.8 Training Targets Generation

At each timestep t, the environment provides a tuple of data as described in seciton (5.2.2). The agent interacts with the environment by performing a tree search and taking action a_t. The search statistics of the tree search were also saved, including the updated value estimate of the root action $\hat{v}_t$, and the updated action probability distribution $\hat{p}_t$. These completes one step sample $\mathcal{T}_t$ for timestep t, which is a tuple of $(o_t, a_t, b_t^{\text{first}}, b_t^{\text{last}}, r_t, m_t^{A_a}, \hat{v}_t, \hat{p}_t)$. 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 with the size of T. For example, o_t is stacked from shape (H, W, C_e) to shape (T, H, W, C_e). The interaction training rollout workers described in 5.2.11 generate trajectories this way. The reanalyze rollout workers generate trajectories with the same signature, but through statistics update described in using a vectorized mocked environment (see 5.2.12 and 5.2.3).

Each trajectory sample with T step samples were processed into T training targets. For each training target at timestep i, we create a training target as follows:

- Observations $o_{i-H-1}, \ldots, o_{i+1}$ where H is the history stacking size. The first H observations were used to create policy inputs as described in 5.2.6, and the pair of observation o_i, o_{i+i} were used to compute self-consistency loss described in 5.2.9.
- Actions $a_{i-H-2}, \ldots, a_{i+K-1}$. Similarly, The first H actions were used for policy input and the pair of actions at (a_{i-1}, a_i) were used for self-consistency loss. The actions $a_i, \ldots, a_{i+K-1}$ were used to unroll the model during the training for K steps.
- Rewards $r_{i+1}, \ldots, r_{i+K}$ as targets of the reward head of the dynamics function.
- Action probabilities $\hat{p}_i, \ldots, \hat{p}_{i+K}$ from the statistics of $K+1$ search trees.
- Root values $\hat{v}_i, \ldots, \hat{v}_{i+K}$, similarly, from the statistics of $K+1$ search trees.
- N-step return $G_i^N, \ldots, G_{i+K}^N$. Each N-step return was computed based on the formula

$$G_t^N = \sum_{i=0}^{i=N-1} \gamma^i r_{t+i+1} + \gamma^N \hat{v}_{t+N}$$

- Importance sampling ratio $\rho = 1$. Placeholder value for future override based on replay buffer sampling weights (see 5.2.13).

5.2.9 The Loss Function

The loss function was defined similar to 3.7, but with additional self-consistency loss, terminal action loss, L_2 regularization loss, and

$$\mathcal{L}_t(\theta) = \underbrace{\sum_{k=0}^{K} l^{\text{p}}\left(\pi_{t+k}, p_t^k\right)}_{①} + \underbrace{\sum_{k=0}^{K} l^{\text{v}}\left(z_{t+k}, v_t^k\right)}_{②} + \underbrace{\sum_{k=1}^{K} l^{\text{r}}\left(u_{t+k}, r_t^k\right)}_{③} + \underbrace{c\|\theta\|^2}_{④} \tag{10}$$

5.2.10 Rollout Workers

Rollout workers are ray actors that store copies of environments or history trajectories and generated data by evaluating policies and interacting with the environments or history trajectories. A rollout worker does not inherently serve a specific purpose in the system and its behavior is mostly determined by the universe factory passed to it. There are three main types of rollout workers used in MooZi.

5.2.11 Interaction Training Rollout Worker

The main goal of **interaction training rollout workers** were to generated trajectories by interacting with environments. For each worker, a vectorized enviornment was created as described in 5.2.3, a history stacker was created as described in 5.2.6, and a planner was created using MCTS configurations as described in **??**: Step samples and trajectory samples were collected as the vectorized enviornment stepping, and finally collected trajectory samples were returned as the final output of one run of the worker. See 2 for the pseudo-code of this process.

```
vectorized_environment = make_vectorized_environment(name, number_of_environments)
observation_stacker = make_obs_stacker(history_length)
planner = make_planner(mcts_config)
step_samples: list = []

def run_once(steps):
    for i in range(steps):
        env_result = vectorized_environment.step(action, env_result.is_last)
        stacked_observation = observation_stacker.process(env_result.observation)
        action, search_statistics = planner.process(stacked_observation)
        step_sample = (env_result, action, search_statistics)
        step_samples.append(step_sample)

    trajectory_samples = take_trajectories(step_samples)
    return trajectory_samples
```

Algorithm 2: Interaction Training Rollout Worker

Interaction Testing rollout worker,

5.2.12 Reanalyze Rollout Worker

5.2.13 Replay Buffer

Since most training targets were expected to be sampled more than once, we precomputed the training targets for all received trajectory samples in the replay buffer. Training targets were computed with minimum information neccessary to be used in the loss function (see 5.2.8) so that the precomputation took least memory possible.

The replay buffer:

- stores trajectories generated by the rollout workers

- processes the trajectories into training targets
- stores processed training targets
- computes and updates priorities of training targets
- responsible for sampling and fetching batches of training targets

5.2.14 Parameter Optimizer

The parameter optimizer:

- stores a copy of the neural network specification
- stores the latest copy of neural network parameters
- stores the loss function
- stores the training state
- computes forward and backward passes and updates the parameters

5.2.15 Distributed Training

5.2.16 Logging and Visualization

6 Experiments

(20 pages)

7 Conclusion

(3 pages)

7.1 Future Work

(1 page)

References

[1] *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).

[2] *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).

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

[4] 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).

[5] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI". In: (2008), p. 2.

[6] 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.

[7] Ivo Danihelka et al. "POLICY IMPROVEMENT BY PLANNING WITH GUMBEL". In: (2022), p. 22.

[8] 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).

[9] *Dm_env: The DeepMind RL Environment API*. DeepMind, June 7, 2022. URL: https://github.com/deepmind/dm_env (visited on 06/09/2022).

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

[11] Lasse Espeholt et al. *IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures*. June 28, 2018. DOI: 10.48550/arXiv.1802.01561. arXiv: 1802.01561 [cs].

[12] 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).

[13] Sylvain Gelly et al. "Modification of UCT with Patterns in Monte-Carlo Go". In: (2006). URL: http://citeseerx.ist.psu.edu/viewdoc/citations?doi=10.1.1.96.7727 (visited on 06/03/2022).

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

[15] 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.

[16] 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).

[17] 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.

[18] 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.

[19] 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).

[20] Matteo Hessel et al. "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (Apr. 29, 2018). ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v32i1.11796.

[21] 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).

[22] Dan Horgan et al. "Distributed Prioritized Experience Replay". Mar. 2, 2018. arXiv: 1803.00933 [cs]. URL: http://arxiv.org/abs/1803.00933 (visited on 06/08/2021).

[23] James Bradbury et al. *JAX: Autograd and XLA*. Google, July 22, 2022. URL: https://github.com/google/jax (visited on 07/22/2022).

[24] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. "Siamese Neural Networks for One-shot Image Recognition". In: (), p. 8.

[25] 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).

[26] 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.

[27] 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].

[28] Eric Liang et al. "RLlib: Abstractions for Distributed Reinforcement Learning". June 28, 2018. arXiv: 1712.09381 [cs]. URL: http://arxiv.org/abs/1712.09381 (visited on 09/24/2021).

[29] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". June 16, 2016. arXiv: 1602.01783 [cs]. URL: http://arxiv.org/abs/1602.01783 (visited on 04/28/2021).

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

[31] *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).

[32] *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).

[33] Philipp Moritz et al. *Ray: A Distributed Framework for Emerging AI Applications*. Sept. 29, 2018. DOI: 10.48550/arXiv.1712.05889. arXiv: 1712.05889 [cs, stat].

[34] 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].

[35] 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).

[36] Tom Schaul et al. "Prioritized Experience Replay". Feb. 25, 2016. arXiv: 1511.05952 [cs]. URL: http://arxiv.org/abs/1511.05952 (visited on 06/09/2022).

[37] 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.

[38] 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].

[39] 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.

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

[41] 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.

[42] Ziyu Wang et al. "Dueling Network Architectures for Deep Reinforcement Learning". In: (), p. 9.

[43] 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).

[44] 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.

[45] Weirui Ye et al. "Mastering Atari Games with Limited Data". Oct. 30, 2021. arXiv: 2111.00210 [cs]. URL: http://arxiv.org/abs/2111.00210 (visited on 11/03/2021).

[46] 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].