Todo list

Contents

1 Summary of Notation

We adopt a similar notation to Sutton and Barto [13].

In a Markov Decision Process:

s, s'	states
a	an action
r	a reward
$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{A}(s)$	set of all actions available in state s
$\mathcal{R}$	set of all possible rewards, a finite subset of $\mathbb{R}$
$\subset$	subset of (e.g., $\mathcal{R} \subset \mathbb{R}$)
$\in$	is an element of; e.g. $(s \in \mathcal{S}, r \in \mathcal{R})$
$\lvert\mathcal{S}\rvert$	number of elements in set $\mathcal{S}$

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 will describle a framework for solving the problem of GGP. We first define a interaction interface that's similar to the Agent-Environment Interface established in the current literature. We will also detail **MooZi**, a system that implements the GGP framework and the **MuZero** algorithm for playing both boardgames and Atari games.

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 [14, 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 [13].

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 [5]. 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 [7].

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 [8]. 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 [6]. Rémi Coulom developed the general idea of **Monte-Carlo Tree Search** that combines both Monte-Carlo rollouts and tree search [3] for his Go program CrazyStone. Shortly afterwards, Gelly et al. implemented another Go program MoGo, that uses the UCT selection formula [4]. MCTS was generalized by Chaslot et al. as a framework for game AI [2]. 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).

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

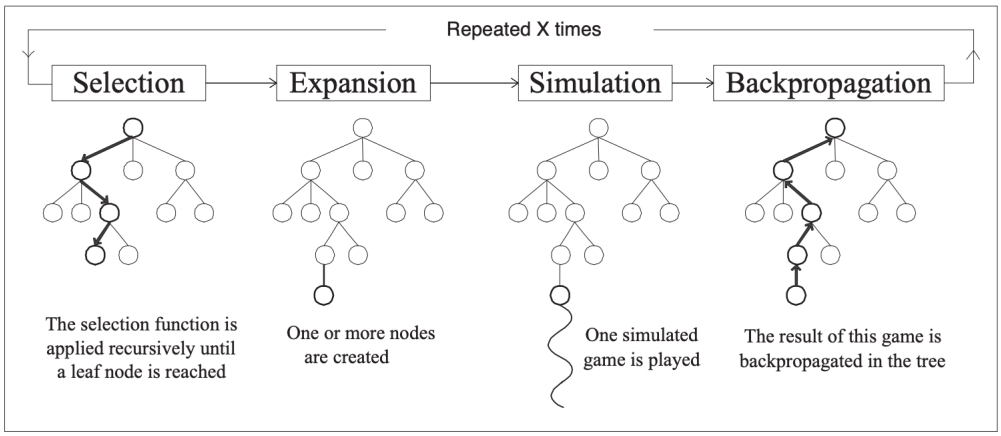

Figure 1: The Monte-Carlo Tree Search Framework

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

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

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 [4].

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

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.

I'm not sure if using bullets points is appropriate. I also tried using in-line number list (e.g., (1) ..., (2) ...,)

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 [9].

3.4 The AlphaGo Family

3.4.1 AlphaGo

In 2017, Silver et al. developed **AlphaGo**, the first Go program that beats a human Go champion on even terms [12]. 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 $V_(s_L)$

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

3.4.2 AlphaGo Zero

AlphaGo Zero is a successor of AlphaGo that beat AlphaGo 100-0 [12]. 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 ouputs 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?

3.4.3 AlphaZero

AlphaZero reduces game specific knowledge of AlphaGo Zero so that the same algorithm could be also applied to Shogi and chess [11]. 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

3.5 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

$$s_t^0 = h_\theta(o_1, o_2, ..., o_t) \tag{4}$$

$$(s^{k+1}, r^{k+1}) = g_\theta(s^k, a^k) \tag{5}$$

$$(v^k, \boldsymbol{p}^k) = f_\theta(s^k) \tag{6}$$

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.

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

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

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

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

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.

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 2). 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**.

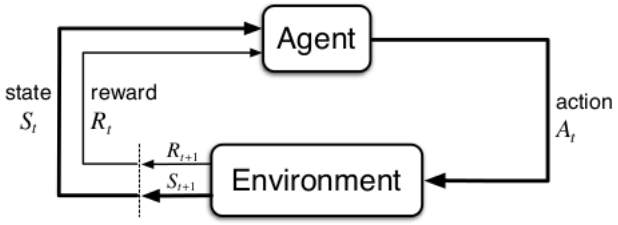

Figure 2: Agent-Environment Interface

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

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

where $\mathbb{E}_\pi$ denotes the expectation of the agent following policy π. Another way is to also use a discount factor λ so to favor short-term rewards

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

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 GII, **laws** are pure functions that read from and write to the **tape**. Agents implemented in Agent-Environment Interface usually do not separate the storage of data and the handling of data. 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 **state** 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 Structure Overview

5.2.1 Driver

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 to describe the process that stores a policy and interacts with an environment. Even though MooZi does not adopt the concept of a RL actor, we will use the term

include data from previous presentation to make the point

boardgames are

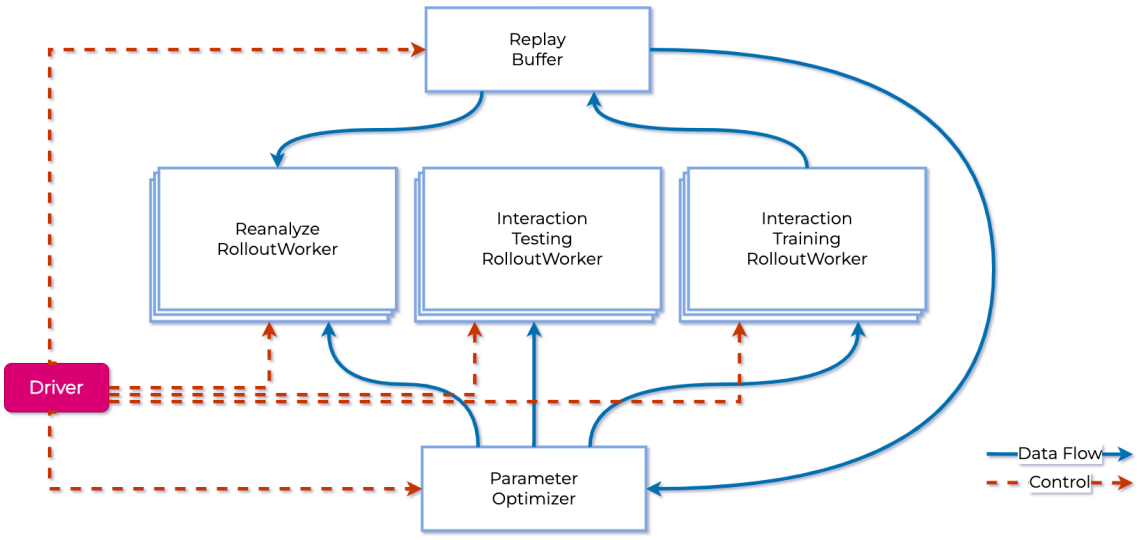

Figure 3: MooZi Architecture

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 decides 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.

5.2.2 Environment Adaptors

Environment adaptors unify environments defined in different libraries into a unified interface. In the software engineering nomenclature, environment adaptors follow the adaptor design pattern [1].

More specifically, in our project we implement environment adaptors for three types of environments that are commonly used in RL research: (1) OpenAI Gym (2) OpenSpiel (3) MinAtar. The adaptors convert the inputs and outputs of these environments into forms that GII accepts.

The adaptors have the same signature as follows:

- Inputs

  is_first: A boolean signals the episode start.

  is_last: A boolean signals the episode end.

  action: An integer indiates the last action taken by the agent. The valid range of the action is $[0, \text{number\_of\_available\_actions})$.

- Outputs

  obs: An N-dimensional array that represents the observation of the current timestep.

  is_first: A boolean signals the episode start.

  is_last: A boolean signals the episode end.

  to_play: An integer indicates the next player to take a move.

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

legal_actions_mask: A bit mask of legal action indicies.

5.2.3 Rollout Workers

A **rollout worker** is a ray actor that:

- stores a collection of universes, including tapes and laws in the universes
- stores a copy of the neural network specification
- stores a copy of the neural network parameters
- optionally stores batching layers that enable efficient computation

A rollout worker does not inherently serve a specific purpose in the system and its behavior is mostly determined by the list of laws created with the universes.

There are three main patterns of rollout workers used in MooZi: **interaction training rollout worker**, **interaction testing rollout worker**, and **reanalyze rollout worker**.

5.2.4 Replay Buffer

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

5.2.5 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

5.2.6 Distributed Training

```
for epoch in range(num_epochs):
for w in workers_env + workers_test + workers_reanalyze:
w.set_params_and_state(param_opt.get_params_and_state())

while traj_futures:
traj, traj_futures = ray.wait(traj_futures)
traj = traj[0]
replay_buffer.add_trajs(traj)

if epoch >= epoch_train_start:
train_batch = replay_buffer.get_train_targets_batch(
big_batch_size
)
param_opt.update(train_batch, batch_size)

env_trajs = [w.run(num_ticks_per_epoch) for w in workers_env]
reanalyze_trajs = [w.run() for w in workers_reanalyze]
traj_futures = env_trajs + reanalyze_trajs

if epoch % test_interval == 0:
test_result = workers_test[0].run(120)

for w in workers_reanalyze:
reanalyze_input = replay_buffer.get_train_targets_batch(
num_trajs_per_reanalyze_universe
* num_universes_per_reanalyze_worker
)
w.set_inputs(reanalyze_input)
```

explain driver pseudo-code

5.2.7 Monte-Carlo Tree Search

5.2.8 Logging and Visualization

6 Experiments

(20 pages)

7 Conclusion

(3 pages)

7.1 Future Work

(1 page)

References

[1] *Adapter Pattern*. In: *Wikipedia*. May 31, 2022. URL: https://en.wikipedia.org/w/index.php?title=Adapter_pattern&oldid=1090799086 (visited on 06/09/2022).

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

[3] 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.

[4] 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).

[5] 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.

[6] 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).

[7] 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.

[8] *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).

[9] 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).

[10] 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.

[11] 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].

[12] 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.

[13] 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.

[14] 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.