Todo list

Contents

1 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.

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 enviornment. 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.

1.1 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.

2 Literature Review

4 - 5 pages

2.1 Planning and Search

2.1.1 Introduction

Most, if not all, AI problems can be reduced to a search problem [**ArtificialIntelligenceGames˙Yannakakis.Togelius˙2018**]. Such search problems could be solved by determining the best plan, path, model, function, and so on, based some metrics of interest. Therefore, search has been playing a vital role in AI research since its dawn.

The term **planning** and **search** are widely used across different domains, especially in AI, and sometimes interchangable. Here we adopt the definition by **ReinforcementLearningIntroduction˙Sutton.Barto˙2018** [**ReinforcementLearningIntroduction˙Sutton.Barto˙2018**].

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 operater $\mathcal{I}$ 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 **background planning**. That is, for any timestep t:

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

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, an random subset of the state space is selected and all states in the subset are updated.

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:

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

We could also use blend both types of planning together. Technically, we could say that algorithms like AlphaGo use both types of planning when they self-play for training. The decision-time part is straight-forward: 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 all states are updated. The updates of this background planning is applied when the planner pulls the latest weights of the neural networks.

Early AI research has been focused on the use of search as a planning method. In 1968, **FormalBasisHeuristic˙Hart.Nilsson.ea˙1968** designed the **A*** algorithm for finding shortest path from a start vertex to a target vertex [**FormalBasisHeuristic˙Hart.Nilsson.ea˙1968**]. 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 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, **RealtimeHeuristicSearch˙Korf˙1990** framed the problem of **Real-Time Heursitic Search**, proposed the Real-Time-A* algorith, and pioneered the study of search algorithms with bounded computation [**RealtimeHeuristicSearch˙Korf˙1990**].

Monte-Carlo techniques were adopted to handle environment stochasticity. Tree-based search algorithms such as **MiniMax** and **Alpha-Beta Pruning** were also designed to better play two-player games.

2.2 Monte Carlo Methods

In 1873, Joseph Jagger observed the bias in roulette wheels at Monte Carlo Casino. He studied the bias by recording results of roulette wheels and captialized over 2 million francs over several days by betting on the most favorably biased wheel [**MonteCarloCasino¨2022**]. Therefore, **Monte Carlo (MC)** methods gained their name as a class of algorithms based on building expectation on random processes.

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, evaluations 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. Heursitic 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**.

2.3 Monte-Carlo Tree Search (MCTS)

BanditBasedMonteCarlo˙Kocsis.Szepesvari¨2006 developed the **Upper Confidence Bounds applied to Trees (UCT)** method as an extension of the Upper Confidence Bound (UCB) employed in bandits [**BanditBasedMonteCarlo˙Kocsis.Szepesvari¨2006**]. Rémi Coulom developed the general idea of **Monte-Carlo Tree Search** that combines both Monte-Carlo rollouts and tree search [**EfficientSelectivityBackup˙Coulom˙2007**] for Go program CrazyStone. Shortly afterwards, **ModificationUCTPatterns˙Gelly.Wang.ea˙2006** implemented MoGo that uses the UCT selection formula [**ModificationUCTPatterns˙Gelly.Wang.ea˙2006**]. MCTS is then generalized by **MonteCarloTreeSearch˙Chaslot.Bakkes.ea˙2008** as a framework for game AI [**MonteCarloTreeSearch˙Chaslot.Bakkes.ea˙2008**]. This framework requires less domain knowledge than other classic approaches to game AI while also being competent in strength. 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 this way will have an emphasize on more promising moves and game states. This means these more promising states are visited more often, have more children, and have deeper estimates that are aggregated to a more accurate value.

Here we detail the four steps in the MCTS framework by **MonteCarloTreeSearch˙Chaslot.Bakkes.ea˙2008** (see Figure **??**).

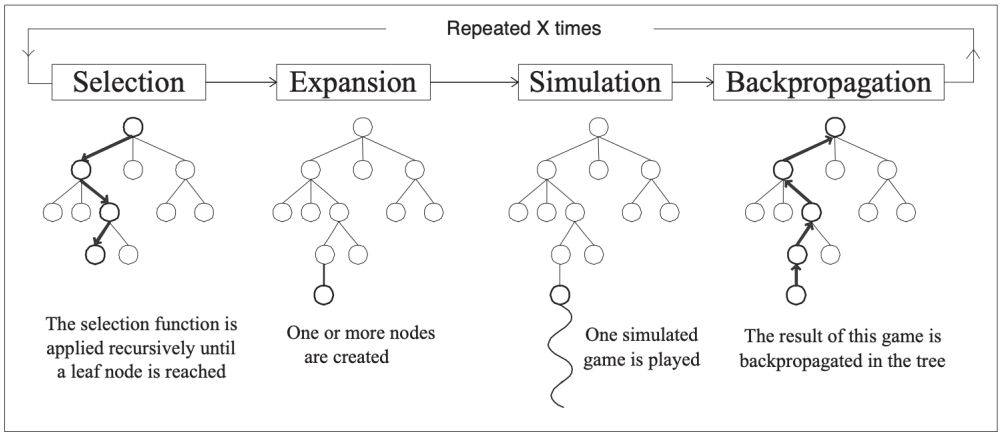

Figure 1: The Monte-Carlo Tree Search Framework

2.3.1 Selection

This selection process starts at the root node and repeats until a leaf node is reached. At each level of the tree, a child node is selected based on a selection formula. A selection formula usually has two parts, one part is based on the prior score x of the state, and the other part is the exploration bonus b. For a parent node p and children nodes C, the selection I is based on

$$I_p = \text{argmax}_{c \in C} \left(x_c + b_c \right)$$

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 $\pi(c \mid p)$. 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, **ModificationUCTPatterns˙Gelly.Wang.ea˙2006** used this following formula for selection in their MoGo implentation:

$$I_p = \text{argmax}_{c \in C} \left(\frac{v_c}{n_c} + \sqrt{\frac{2 * \log(n_b)}{n_c}} \right)$$

where v_c is the value of the node, n_b and n_c are the visit counts of the parent and child, respectively.

2.3.2 Expansion

The selected leaf node is then expanded by adding one or more children to the leaf node, each represents a succeeding game state of the selected leaf's state.

2.3.3 Evaluation

Each of the expanded nodes is now evaluated by using one of these methods:

- A game is played until the game ends using a random policy.
- A game is played until the game ends using a rollout policy.
- A evaluation function is applied to the node. This evaluation function could be either a hand-crafted heursitic or learned by a neural network

2.3.4 Backpropagation

After the expanded nodes are evaluated, the traversed nodes to reach the expanded nodes are updated. The statistics updated usually include visit count, estimated value and accumulated reward of the nodes.

These four steps are repeated until the budget runs out. After this search is performed, 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.

2.4 AlphaGo, AlphaGo Zero, and Alpha Zero

describe AlphaGo, AlphaGo Zero, and Alpha Zero in more details

AlphaGo is the first Go program that beats a human Go champion. AlphaGo learns a policy net that maps states to actions, and a value net that maps states to values.

AlphaGo Zero is a successor of AlphaGo with the main difference of not learning from human.

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

2.5 MuZero

MuZero is different from the Alpha Zero family where the model used for planning is learned instead of given. This difference further reduces game specific knowledge required for the algorithm. More specifically, MuZero no longer requires the access to a perfect model of the target game. Instead, a neural network that learns from the game experience is used in the search to replace the perfect model.

expand MuZero discussion here, include all the formulae, address differences in the methods section

(5 pages)

3 Problem Definition

3.1 Agent-Environment Interface and Markov Decision Process

Traditionally, RL problems and solutions are frame with the **Agent-Environment Interface** (Figure **??**).

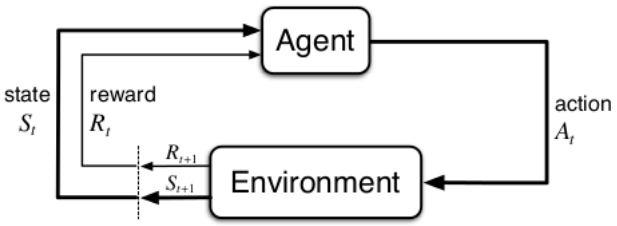

Figure 2: Agent-Environment Interface

The decision maker in this interface is called the **Agent**, and the agent interacts with the **Environment** continuously. A problem implements the interface could be represented as a **Markov Decision Process (MDP)**, which is tuple of four elements where:

- $\mathcal{S}$, a set of states that forms the *state space*
- $\mathcal{A}$, a set of actions that forms the *action space*
- $P(s, a, s') = Pr[S_{t+1} = s' \mid S_t = s, A_t = a]$, the transition probability function
- $R(s, a, s')$ the reward function

At each time step t, the agent starts at state $S_t \in \mathcal{S}$, picks an action $A_t \in \mathcal{A}$ transitions to state $S_{t+1} \in \mathcal{S}$ based on the probability function$P(S_{t+1} \mid S_t, A_t)$, and receives a reward $R(S_t, A_t, S_{t+1})$.

The agent interacts with enviornment and generates 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**. In the finite case, this interaction terminates until a terminal state is reached at time $t = T$, and this sequence is called an **episode**. In the infinite case, the interaction continues indefinitely. The goal of the agent in the problem is either to maximize the accumulative reward in the finite setting, or to maximize the average reward in the infinite case.

also describes policy π

address this is the most common formulation and how different

3.2 Shortcomings of the Agent-Environment Interface for General Game Playing

3.2.1 Multi-Agent Games

address OpenSpiel's design multiple agents

3.2.2 Partial Observability

address POMDP

3.2.3 Environment Stochasticity

address OpenSpiel's design of random node

3.2.4 Episodic vs Continuous

3.2.5 Self-Observability

barely seen in the literature, need more literature review

3.2.6 Environment Output Structure

3.3 Our Approach

(5 pages)

3.3.1 Generalized Interaction Interface

We propose the **Generalized Interaction Interface (GII)**. We define the **tape** E as the data storage of the interface, and a **law** L as a pure function that operates on the tape. An instance of such interface could consists of exactly one tape and multiple laws, and we define such an instance a **universe**. A universe **ticks** by applying the laws on the tape.

elaborate formally

We implement a simplified version of this interface in **MooZi**.

3.3.2 Advantages

pure functions are efficient

4 Method

(20 - 25 pages)

4.1 Design Philosophy

4.1.1 Use of Generalized Interaction Interface

One of the goals of the project is to demostrate the use of Generalized Interaction Interface (GII). All modules in the project will be implemented to align with the interface. Third-party libraries that include game environments are wrapped with special wrappers that converts the outputs into the GII format.

4.1.2 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

4.1.3 Being User Friendly

4.1.4 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.

4.2 Structure Overview

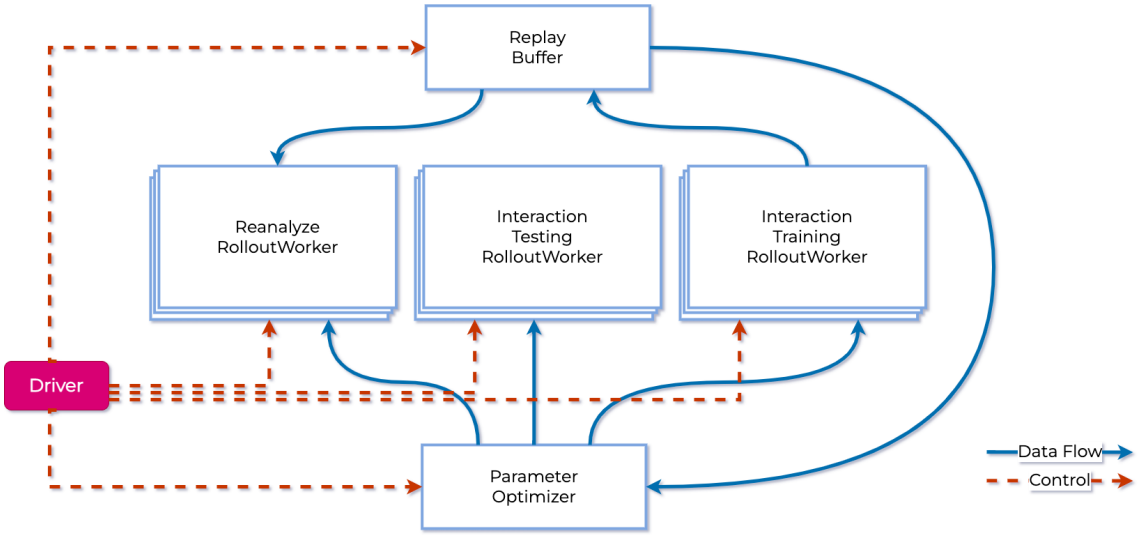

Figure 3: MooZi Architecture

include data from previous presentation to make the point

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

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

4.2.1 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 **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.

4.2.2 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 [**AdapterPattern¨2022**].

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:

- $\mathbf{is}_{f}irstabooleansignalstheepisodestart$

4.2.3 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**.

4.2.4 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

4.2.5 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

4.2.6 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

4.2.7 Monte-Carlo Tree Search

4.2.8 Logging and Visualization

5 Experiments

(20 pages)

6 Conclusion

(3 pages)

6.1 Future Work

(1 page)