Todo list

8 - 10 pages of introduction . . . 1
4 - 5 pages . . . 2
describe search in general and with a focus on MCTS . . . 2
describe MCTS, selection, expansion, e.t.c., also describes how MCTS and NN work together . . . 2
describe AlphaGo, AlphaGo Zero, and Alpha Zero in more details . . . 2
expand MuZero discussion here, include all the formulae, address differences in the methods section . . . 3
(5 pages) . . . 3
also describes policy π . . . 3
address this is the most common formulation and how different libraries implement the interface . . . 3
address OpenSpiel's design multiple agents . . . 4
address POMDP . . . 4
address OpenSpiel's design of random node . . . 4
barely seen in the literature, need more literature review . . . 4
(5 pages) . . . 4
elaborate formally . . . 4
pure functions are efficient . . . 4
(20 - 25 pages) . . . 4
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
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
include data from previous presentation to make the point . . . 5
boardgames are fine, Atari games are slow, but in either cases we can't control 5
MCTS inference batching is slow due to IO overhead, include data here from previous presentation to make the point . . . 5
explain driver pseudo-code . . . 7
(20 pages) . . . 7
(3 pages) . . . 7
(1 page) . . . 7

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 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

describe search in general and with a focus on MCTS

Early AI research has been focused on the use of search as a planning method. Algorithms like **A*** were designed to find the optimal path to goals. Although A* works quite well for many problems, 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.

Real-Time Heursitic Search pioneered the study of search algorithms with bounded computation. 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.

2.1 Monte-Carlo Tree Search (MCTS)

describe MCTS, selection, expansion, e.t.c., also describes how MCTS and NN work together

Monte-Carlo Tree Search (MCTS) is a search algorithm for game AI that combines both Monte-Carlo rollouts and tree search. MCTS requires less domain knowledge than other classic approaches to game AI while also being competent in strength.

2.2 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.

2.3 MuZero

MuZero is different from the Alpha Zero family where the model used for plannig 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 1).

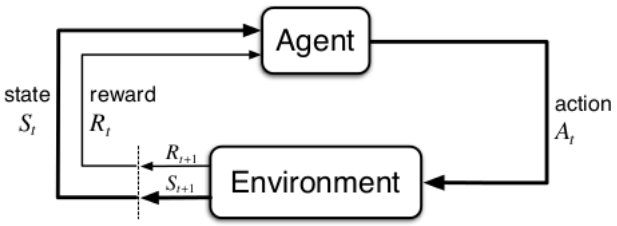

Figure 1: Agent-Environment Interface

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

3.2.6 Environment Output Structure

barely seen in the literature, need more literature review

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

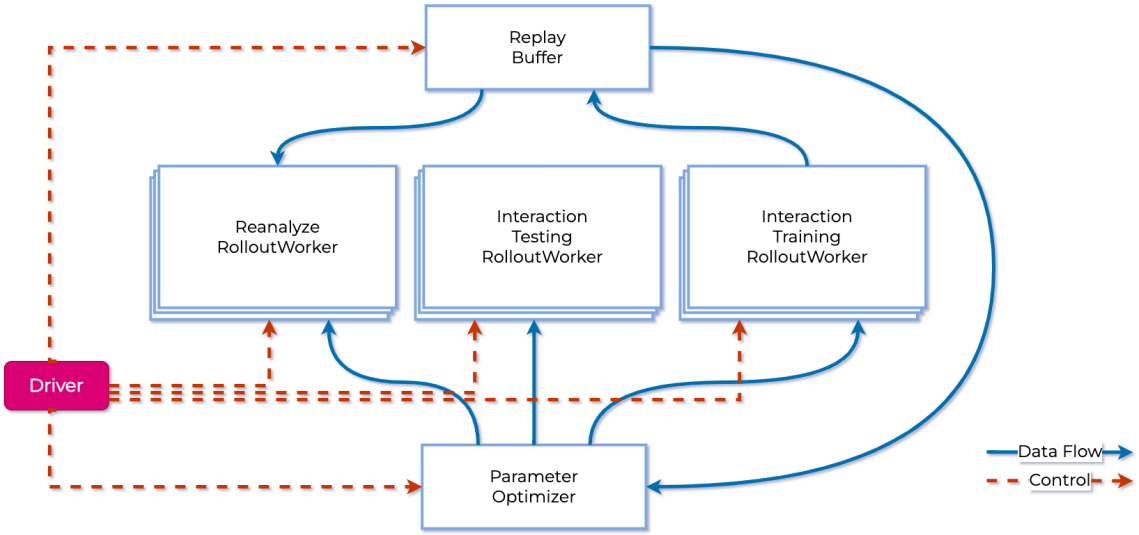

Figure 2: MooZi Architecture

include data from previous presentation to make the point

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

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

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

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 Wrapper

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)