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 [10, p.39]. 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 Sutton and Barto [9].

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, a 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, 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 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 framed the problem of **Real-Time Heursitic Search**, proposed the Real-Time-A* algorith, and pioneered the study of search algorithms with bounded computation [7].

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 [8]. 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)

Kocsis and Szepesvári developed the **Upper Confidence Bounds applied to Trees (UCT)** method as an extension of the Upper Confidence Bound (UCB) 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 Go program CrazyStone. Shortly afterwards, Gelly et al. implemented MoGo that uses the UCT selection formula [4]. MCTS is then generalized by Chaslot et al. as a

framework for game AI [2]. 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 Chaslot et al. (see Figure 1).

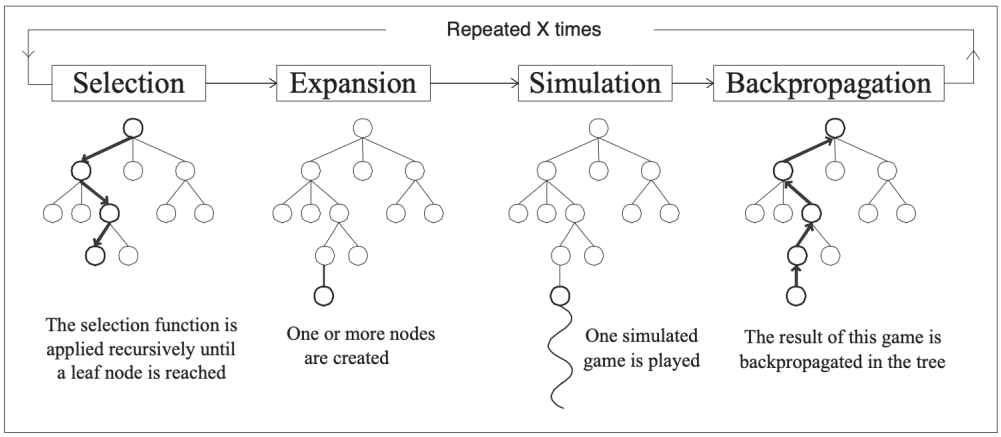

Figure 1: The Monte-Carlo Tree Search Framework

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_p is based on

$$I_p = \text{argmax}_{c \in C} \left(x_c + b_c \right) \tag{1}$$

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 $\pi(c \mid p)$. 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, Gelly et al. 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.
- An evaluation function is applied to the node. This evaluation function could be either a hand-crafted heuristic 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 highest lower bound.

2.4 AlphaGo, AlphaGo Zero, and Alpha Zero

AlphaGo is the first Go program that beats a human Go champion by 5 games to 0. The AlphaGo is 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. 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 ascend to maximize the probability of matching the human expert move:

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

For the second stage of training, the supervised policy p_σ is duplicated further trained with reinforcement learning. This reinforcement learning trained policy (or RL policy) is parametrized by weights ρ which is initialized to the same values as σ. In other words, the RL policy is initialized so that $p_\rho = p_\sigma$. The training data is then generated through self-play 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, $+1$ for winning, -1 for losing from the perspective of the current player. Weights ρ is then updated using gradient ascend to maximize the expected outcome using the update formula:

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

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 as if both players act according to the policy p_ρ. The 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 2.3. In the search tree, each edge (s, a) stores an action value $Q(s, a)$, a visit count $N(s, a)$, and a prior probability $P(s, a)$. At each time step, the search starts at the root node and simulates until the budget runs out. In the select phase of each simulation phase, an action is selected for each traversed node using the same base formula (1). In AlphaGo, the base score of the selection formula is the estimated value of the next state after taking the action, namely $Q(s, a)$. The exploration bonus of edge (s, a) is based on the prior probability and decays as visit count grows.

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

The action taken at time t is the action that maximizes the sum of the base 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 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.

Central to MuZero's design are three functions trained for acting and planing. The **representation function** h encodes a history of observations $o_1, o_2, ..., o_t$ into a hidden state s_t. 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.

MuZero plans with a search method based on the MCTS framework (discussed in 2.3). Due to the lack of access to a perfect model, MuZero's MCTS differs from a standard one in numerous ways. The nodes are no longer perfect representations of the board states. Instead, each node is associated with a hidden state s as a learned representation of the board state. The transition is no longer made by the perfect model but the dynamics function. There are technically no terminal states in the tree as the learned model does not predict game termination. For timestep t, the search starts by creating the root node of the search tree with a history of observations using the representation function

$$s^0 \leftarrow h_\theta(o_0, o_1, o_2, ..., o_t)$$