TÉCNICO LISBOA

A Modular Architecture for Model-Based Deep Reinforcement Learning

Tiago João Gaspar Ribeiro de Oliveira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Arlindo Manuel Limede de Oliveira

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. Arlindo Manuel Limede de Oliveira
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

October 2021

Acknowledgments

I will convey in a few lines a message that deserves much more.

I am forever grateful

to Instituto Superior Técnico, for the challenges and people that made me who I am today.

to Professor Arlindo Oliveira, for the guidance throughout this work that taught me so much.

to my friends, for the laughter, joy and silliness.

to my family, for the support, kindness and love.

Abstract

The model-based reinforcement learning (MBRL) paradigm, which uses planning algorithms, has recently achieved unprecedented results in the area of deep reinforcement learning (DRL). These agents are quite complex and involve multiple components, factors that can create challenges for research. In this work, we propose a modular software architecture [1] suited for these types of agents, which makes possible the implementation of different algorithms and for each component to be easily configured (such as different exploration policies, search algorithms...). We illustrate the use of this architecture by implementing several algorithms and experimenting with agents created using different combinations of these. We also suggest a new simple search algorithm called *averaged minimax* that achieved good results in this work. Our experiments also show that the best algorithm combination is problem-dependent.

Keywords

Deep Reinforcement Learning, Model-Based Reinforcement Learning, Neural Network, Architecture, Implementation.

[1] Our implementation can be found in https://github.com/GaspTO/Modular_MBRL

Resumo

O paradigma de aprendizagem por reforço com base em modelo, que utiliza algoritmos de planeamento, atingiu recentemente resultados sem precedentes na área de aprendizagem por reforço profunda. Estes agentes são bastante complexos e envolvem vários componentes, fatores que podem criar desafios para a investigação. Neste trabalho, propomos uma arquitetura de software modular [2] apropriada para este tipo de agentes, que torna possível que diferentes algoritmos sejam implementados e que cada componente possa ser facilmente configurado (como, por exemplo, diferentes políticas de exploração, estratégias de planeamento...). Demonstramos o uso desta através da implementação de vários algoritmos e experimentando com agentes resultantes de várias combinações destes. Sugerimos também um novo algoritmo de procura simples chamado *minimax médio* que conseguiu bons resultados neste trabalho. As nossas experiências também mostram que a melhor combinação de algoritmos é dependente de cada problema.

Palavras Chave

Aprendizagem Profunda por Reforço, Aprendizagem por Reforço baseada em Modelo, Rede Neuronal, Arquitetura, Implementação.

[2] A nossa implementação encontra-se em https://github.com/GaspTO/Modular_MBRL

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AI	artificial intelligence
BFMMS	best first minimax search
DRL	deep reinforcement learning
MBRL	model-based reinforcement learning
MCTS	Monte Carlo tree search
MDP	Markov decision process
MLP	multilayer perceptron
MUCB	masked upper confidence bound
PER	prioritized experience replay
POMDP	partially observable markov decision process
RL	reinforcement learning
SAVE	search with amortized value estimates
TD	temporal difference
UCB	upper confidence bound
UCT	upper confidence bound in trees
VPN	value prediction network

1

Introduction

Contents

In artificial intelligence (AI), machine learning concerns itself with making machines learn from data. One of its learning methods is called reinforcement learning (RL) and it does so by trial and error, a method human beings are very familiar with. We can see RL as a method where an agent interacts with a problem and, while doing so, learns how to predict the outcome of its own actions. It then uses these predictions to be as successful in a task as possible. Out of the many ways it is possible to learn these predictions, the most recent trend in research has been to employ deep learning methods. Deep learning is yet another area of machine learning and it involves all computational learning that is done using neural networks with high number of layers and parameters. Deep neural networks have made impressive breakthroughs in recent years. Their capacity to adapt easily to data has made them excellent models to solve complex tasks. Due to their success, these models have been applied in reinforcement learning, resulting in an area called deep reinforcement learning (DRL) [1].

Two recent accomplishments attracted the interest in this area. The first, in 2015, was the creation of Deep Q-networks [2], which were able to achieve human-level play in Atari games. The techniques used became standard in the area and opened the door for successive advancements after that. The second, in 2016, was when a program called AlphaGo [3] beat a Go world champion for the first time. Go was seen as the new milestone of AI since the defeat of the chess world champion, Gary Kasparov, in 1997 by DeepBlue [4] and it was finally beaten using reinforcement learning, neural networks and a search algorithm. Following that achievement, an even more powerful program called AlphaGo Zero [5] was created, learning without any human knowledge, indicating the strong potential of DRL.

It is not that any of those elements brought anything completely new to the field, but it was the first time they were assembled together and achieved outstanding results. The search component allowed deep reinforcement learning to reach a completely new level. Following this line of algorithms, two other very successful agents were created: AlphaZero, a generalized version of AlphaGo Zero, that became the best player in chess, shogi and Go; and MuZero, a program that achieved the same outstanding results, but it was not given the rules of the game, having to learn them as it played - a very promising line of research, since most of the world does not have a clear set of rules that can be explained to a computer beforehand.

1.1 Motivation

Despite all of this, there remains an overwhelming barrier concerning these algorithms. The hardware resources used to train these models were much higher than what is usually available to researchers (AlphaZero used more than 5000 first-generation Tensor Processing Units[1]). It is only normal that, after their success, the research on these algorithms has been increasing, trying to make them more practical

[1] https://cloud.google.com/tpu/docs/tpus

to use [6–10].

State of the art RL algorithms, especially these new model-based algorithms that use search, are considerably complex and involve multiple parts. Implementing these algorithms can be very time consuming, one of the reasons being deep learning's proclivity to *fail silently* due to its adaptable nature, which often leads to a large amount of time spent *debugging* and verifying code. It is not practical, every time one wants to try and study different ideas, to have to refactor and heavily recode the implementation. And, even though there has been a recent effort to publish open source implementations, these do not usually take their possible extensions into account.

1.2 Contributions

Our work attempts to help with these problems by introducing a modular software architecture for model-based reinforcement learning. Its objective is to separate the agent into distinctive components and allow the implementation of new strategies (the algorithms in each one of them), that can be added without having to modify the other components. The architecture should enable us to construct multiple agents easily by choosing different strategy combinations.

We provide a particular implementation of this architecture, using some common strategies in reinforcement learning and suggesting some new ones, namely a new simple search algorithm called *averaged minimax*. In the end, we make a comparative study of agents created by different strategies and show that the best agent is problem specific.

2

Background

Contents

This section explains concepts of reinforcement learning necessary to understand our work, mentioning some of the most important parts in a RL agent.

2.1 Reinforcement learning

In reinforcement learning, an agent interacts with an environment. We assume it to be described as a Markov decision process (MDP). At each time step t, the agent will decide the next action, a_t, to take. The environment will receive this action and transition its state, s_t, to the next one, s_{t+1}, giving back a reward, r_t, associated. The agent continues to interact with the environment until the episode reaches a terminal state, originating a sequence of transitions, called *trajectory*, $s_0, a_0, r_0, s_1, a_1, r_1, s_2...$

These environments hold the Markov property, meaning that the result of an action (next state and reward) depends solely on the current state and not on the prior history of the current episode. In a MDP, there is a state space $\mathcal{S}$, an action space $\mathcal{A}$, a probability function, $p(s'|s,a)$, that calculates the likelihood of the agent ending in state s' if action a is chosen in state s, and a reward function, $r(s,a,s')$, that gives the reward associated with this transition. These two functions describe the dynamics of the environment, for all $s', s \in \mathcal{S}$ and $a \in \mathcal{A}$.

There are also certain environments that hold the Markov property but the states are not completely visible to the agent (for example, a cards game where the agent does not know the cards of the opponents). These environments are formulated as a partially observable markov decision process (POMDP). In these, an agent has to act under the uncertainty of its current state, only having access to an observation. MDPs are just specializations of POMDPs.

The agent attempts to maximize the expected (discounted) reward observed in each episode,

$$G_t = \sum_{j=0} \gamma^j r_{t+j} \ , \tag{2.1}$$

and the purpose of RL is to teach the agent how to maximize this objective through successive interactions with the environment.

To simplify, we will assume MDPs for the rest of this section since their observations reflect perfectly the environment's inner state. The next subsections are described based on the notion of state.

2.2 Policy and Value function

A policy defines which action an agent should take in each state, $\pi(a|s)$, and it does not need to be deterministic, in which case it defines a probability distribution over the action space. A value function, $v_\pi(s)$, defines the expected return of an agent if it follows a policy π. The problem of estimating this

function is called the *prediction problem*. MDPs can be better understood using the Bellman's equations:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \cdot [r(s,a,s') + \gamma v_\pi(s')] \tag{2.2}$$

and

$$q_\pi(s,a) = \sum_{s'} p(s'|s,a) \cdot [r(s,a,s') + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')]. \tag{2.3}$$

The state-value function can be recovered from the action-value function,

$$v_\pi(s) = \max_a q_\pi(s,a). \tag{2.4}$$

The optimal policy, π^*, is the one that maximizes the state-value and action-value functions,

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.5}$$

and

$$q_*(s,a) = \max_\pi q_\pi(s,a). \tag{2.6}$$

The problem of finding this policy is called the control problem.

2.3 Dynamic Programming

Given a perfect model of the environment, it is theoretically possible to use dynamic programming techniques to estimate the optimal policy. We assume an array function, V, with one entry for every state. In dynamic programming, the value estimation for a state is updated based on the estimations of the possible next states - this process is called *bootstrapping*. The algorithms used in dynamic programming fall into either value iteration or policy iteration.

Policy iteration

Policy iteration algorithms iterate over two steps alternately. The first, called policy evaluation, attempts to find the state-value function for the current policy using an iterative version of equation 2.2,

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} p(s'|s,a) \cdot [r(s,a,s') + \gamma V(s')]. \tag{2.7}$$

The second step, called policy improvement, uses the result of the first one to come up with a better policy. A deterministic policy can then be extracted by taking the action that leads to the highest expected

value according to the estimated value function,

$$\pi(s) = \arg\max_a \sum_{s',r} p(s'|s,a)[r(s,a,s') + \gamma V(s')]. \tag{2.8}$$

The process repeats until some stopping condition is met, for instance, the convergence of V.

Value iteration

Policy iteration can be slow due to these two steps. Value iteration methods attempt to find the optimal value function first, using the following update rule until convergence:

$$V(s) \leftarrow \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma V(s')] \tag{2.9}$$

and only in the end deriving a policy from this function.

Final remarks

These techniques are unpractical, especially for large states or for environments without a known model, but they are of extreme importance to understand the reasoning behind reinforcement learning methods, since these attempt to get the same results as those of dynamic programming. Next, we review Monte Carlo and temporal difference methods which are more practical and useful for reinforcement learning.

2.4 Monte Carlo Methods

Monte Carlo methods do not require the environment model to be learned, but sampled episodes to be used. They can provide an alternative to the policy evaluation step (eq. 2.7) by estimating the value function for a policy using a large number of sampled episodes. They calculate, for each state, the discounted cumulative reward (eq. 2.1) seen during the episode and update the agent based on it. Even without knowing the model, if we run a very large number of episodes, following a certain policy, and update the value function based on the empirical mean of the cumulative rewards observed in them, it should tend to approximate the true value function of the policy (eq 2.2). After completing an episode, we can adjust the value of a state, s_t, using the following update rule:

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)]. \tag{2.10}$$

These methods have the obvious disadvantage of having to run full episodes before starting to conduct updates, which can take too long.

2.5 Temporal Difference

Temporal difference (TD), uses the idea of value bootstrapping to update the value of the states, just like in dynamic programming, but uses environment experience to do so, without making use of the model of the environment. A simple TD algorithm, called *TD(0)* [11], calculates an approximation of the value function for each state by conducting 1-step bootstrap updates in the trajectory,

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)], \tag{2.11}$$

where $\alpha \in \mathbb{R}$ and $\gamma \in [0, 1]$. The quantity $r_t + \gamma V(s_{t+1}) - V(s_t)$ is called TD error. This idea can be generalized to take into account, not just one time step difference between the states, but several, leading to the general idea of *n-step temporal difference*,

$$V(s_t) \leftarrow V(s_t) + \alpha[\left(\sum_{j=0}^{n-1} \gamma^j r_{t+j}\right) + \gamma^n V(s_{t+n}) - V(s_t)]. \tag{2.12}$$

The idea of using multiple transitions to bootstrap is called *multi-step bootstrapping*. We can see that, for infinite steps, the algorithm becomes a Monte Carlo method (we assume that $V(s_t) = 0$ when t is superior to the trajectory's size).

Two other very popular but simple algorithms that learn using 1-step bootstrapped values are *Q-learning* [12] and *Sarsa* [13], with the respective update rules:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{2.13}$$

and

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{2.14}$$

The difference between the two is that Q-learning updates its function *off-policy*, while SARSA updates *on-policy* (see subsection 2.6).

TD(λ)

We mentioned an algorithm called TD(0) and talked about a method to mediate between this algorithm and Monte Carlo algorithms by choosing to get boostrapped values using more than 1 step. There is another idea that allows us to mediate between these two. It relies on using multiple n-step returns to update the value of a state, instead of just one. For instance, we can use the average between a 1-step

and a 2-step return to conduct a temporal difference update,

$$V(s_t) \leftarrow V(s_t) + \alpha \frac{1}{2}\Big([r_t + \gamma V(s_{t+1}) - V(s_t)] + [r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) - V(s_t)]\Big). \tag{2.15}$$

If we denote a n-step return as

$$G_{t:t+n} = \sum_{j=0}^{n} \gamma^j r_{t+j} + \gamma^n V(s_{t+n}), \tag{2.16}$$

then we can generalize the previous idea for infinite steps and define the λ-*return* as

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \ , \tag{2.17}$$

which is a weighted average of all the n-step returns. The weights are given by λ^{n-1} for a n-step return, where $\lambda \in [0, 1]$. To make sure they sum to 1, we multiply them by $(1-\lambda)$. After $t+n$ has surpassed the episode's size, $G_{t:t+n}$ becomes the conventional return, G_t, defined in eq. 2.1. If we rewrite this expression for a finite episode, we get

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t \ , \tag{2.18}$$

where T is the episode length. The λ-return gives more importance to n-step returns with fewer steps. The lower the value of λ, the less important longer n-step returns become. If λ is 0, we get a 1-step return and, if it is 1, the algorithm is a normal Monte Carlo update. If we wait until the end of the episode to update the value function based on a λ-return, the algorithm is called *offline* λ*-return*.

However, we do not need to wait until the end of the episode. The λ-return is a sum of different n-step terms. Some can be calculated sooner than others. If our agent is currently in a state s_t, once we reach the state s_{t+1}, we can update it using the 1-step return, $G_{t:t+1}$ term. When we reach s_{t+2}, we can update it a second time, but now using the 2-step return, $G_{t:t+2}$. We repeat this for every new state reached during the episode. In this way, we do not need to wait until the end of the episode. This is the idea behind an algorithm called *TD(λ)*, which is described more formally in algorithm 2.1. When λ is 0, we get the TD(0) algorithm previously mentioned.

2.6 Learning on-policy vs off-policy

We can use the two algorithms introduced in section 2.5, SARSA and Q-learning, to introduce a very important notion in reinforcement learning. The value function learned by an agent does not need to correspond to the policy that generated the data used. The difference between SARSA and Q-learning

Algorithm 2.1 Tabular TD(λ)

Input Value function array V and policy π to evaluate
Parameters step size $\alpha > 0$ and trace decay rate $\lambda \in [0, 1]$

loop
  Set $e(s)$ = 0 for every state s
  Initialize s as the initial state
  while s not terminal **do**
    Choose $a \quad \pi(.|s)$
    Take action a, observe r,s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    for s in all states **do**
      $V(s) \leftarrow V(s) + \alpha\delta e(s)$
      $e(s) \leftarrow \gamma\lambda e(s)$
    end for
    $s \leftarrow s'$
  end while
end loop

has to do with this. They both generate data following some policy, but, when updating their value function, SARSA learns a value function for the policy that generated the data, while Q-learning tries to learn the optimal value function. Algorithms like SARSA are said to be *on-policy* and algorithms like Q-learning are called off-policy. The former tend to be simpler, faster to learn and more stable. The latter tends to have higher variance, but can lead to better results since it is not bounded by the quality of the policy that generated the data.

2.7 Model Based Algorithms

Monte Carlo and temporal difference algorithms do not necessarily need to understand anything about the dynamics of the environment they act on. But, as seen, that is not the case for dynamic programming techniques, which needs a model of the environment. Learning algorithms that require some information about the dynamics of the environment are called model-based reinforcement learning (MBRL) and those that do not are called model-free algorithms. For MBRL, the model can be given or can be learned by the agent through experience. Model based agents can apply state-space planning algorithms to their model and choose better actions, generating more insightful episodes. They can also learn using simulated experience derived from the model, which is especially useful when the environment dynamics are not known and the interactions with it are expensive or impossible (for example, when the agent is learning from previously gathered experience and can not acquire more, a problem called *offline reinforcement learning*).

2.8 Function Approximation

So far, we have not paid much attention to how these functions are are implemented in practice and simply assumed the values of these functions to be stored in an array with an entree for every state or state-action pair. For instance, if there are 1000 states and each state has 5 actions, we would need 5000 entrees in a table to store an action-value function. Obviously, this is not practical for several reasons. There are some environments that have a continuous space of states and actions. Others, although discrete, have a very large state-space (for example, the state-space complexity of chess is 10^{43}, go's is 10^{172} [14]). Besides, even if we could store a whole function in memory, we would still be neglecting the fact that some states share patterns between themselves. By generalizing and learning how to make decisions based on common patterns found, the agent can improve on multiple states at the same time, being able, in theory, to achieve good performances in states never seen. Out of all the function approximators, in this work, we use neural networks [1] to learn these functions. For example, the update rule for TD(0), using a neural network to approximate v with parameters θ is given by

$$\theta \leftarrow \theta + \alpha[r_t + \gamma v_\theta(s_{t+1}) - v_\theta(s_t, \theta)]\nabla_\theta v_\theta(s_t). \tag{2.19}$$

In other words, we update the weights against the gradient of the temporal error, $r_t + \gamma v_\theta(s_{t+1}) - v_\theta(s_t)$, in order to minimize it.

2.9 Policy Gradient

When using function approximation, it is possible to improve the policy directly without having to estimate a value function first. If we have a policy approximated by a neural network, we can update it to increase the probability of the actions that lead to better results according to the agent's experience. One of the biggest advantages of policy optimization methods is allowing stochastic policies very easily.

If the quality of the policy is defined by

$$J(\pi_\theta) = E_{\tau\sim\pi_\theta}[G(\tau)], \tag{2.20}$$

where τ is a trajectory sampled from the policy and $G(\tau)$ is the total discounted reward of the trajectory. It can be show [11] that

$$\nabla_\theta J(\pi_\theta) \propto E_{\tau\sim\pi_\theta}[\sum_{t=0} \nabla_\theta \log \pi_\theta(a_t|s_t)G(\tau)]. \tag{2.21}$$

To increase J, we follow its gradient,

$$\theta \leftarrow \theta + \alpha\nabla_\theta \log \pi_\theta(a_t|s_t)G(\tau). \tag{2.22}$$

The lower the probability of the action taken, $\pi_\theta(a|s)$, and the higher the total reward seen in the trajectory, the bigger the update step in equation 2.22. There are plenty of variations to this update rule. For instance, by using $G(\tau)$ in the gradient, we are updating the policy of future steps based on rewards received in the past. A more sensible idea would be to only use the total rewards seen after each state, originating a formula known as *reward-to-go policy gradient*,

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_k r_{t+k} \ . \tag{2.23}$$

It is also possible to replace the sum of *rewards to go* by an estimated value,

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) q_\omega(s_t, a_t). \tag{2.24}$$

There are many other variations, all using the idea of following the gradient of $J(\pi_\theta)$.

2.10 Exploration and Exploitation

Agents need to explore unseen states in order to find trajectories with higher rewards and improve the current policy. At the same time, they need to exploit in order to increase the results of their interaction with the environment. This leads to the known exploitation and exploration dilemma that studies how to balance between the two. Arguably, the most popular method is the ϵ-greedy algorithm that, in each state, chooses the best action according to the policy with 1-ϵ probability and a random action otherwise. Another simple exploration method for stochastic policies is to sample an action according to the probability distribution given by them.

A different but popular class of exploration-exploitation techniques is called count-based methods. They increase the exploration according to the number of times a certain action has been taken in some state. Obviously, this is not practical for very large domains, but it is especially useful when using search algorithms for planning. The search algorithm called upper confidence bound in trees (UCT) [15], is such example. In each node of the tree, the next action chosen is the one that maximizes the sum of two components: an exploitation part, that calculates the total reward expected if such action is taken, and an exploration component, that decreases with the number of times that action has been chosen in that state. We refer to the famous *multi-armed bandit* problem for more information about this dilemma [16].

2.11 Replay Buffers

One of the most important achievements in deep reinforcement learning was the creation of Deep Q-Networks [2], which were successful in achieving professional human performance in Atari games using

deep neural networks, only receiving image pixels and scores as input. To address the infamous RL instability problem, a replay buffer that sampled previously seen transitions uniformly was introduced. Experience replay has several advantages. First, for off-policy methods, they allow to sample transitions instead of full games, which breaks the correlation between the data used in each batch. Second, they help to prevent a phenomenon called *catastrophic forgetting* [17, 18], where an agent based on approximated functions forgets how to act in the simpler parts of the environment because it has not visited them for a long time. It also allows for better sample efficiency, decreasing the need for environment interactions.

A significant subsequent improvement was the idea of sampling transitions according to their importance - a method called prioritized experience replay (PER) [19]. Although PER is a general method, in the original work, the priority was given by the temporal difference error,

$$p = r_t + \gamma \max_a q_\theta(s_{t+1}, a) - q_\theta(s_t, a_t). \tag{2.25}$$

The sampling probability of the transition number i was given using the priorities by

$$Pr(i) = \frac{p_i^\alpha}{\sum_j p_j^\alpha}, \tag{2.26}$$

where α balances the prioritization of each transition. The use of replay buffers are now standard practice in DRL and multiple relevant works have been published recently about them [20, 21].

2.12 Search algorithms

As explained in section 2.7, model-based algorithms can plan using the a model of the environment for their benefit. The type of planning that interests us here is the application of search algorithms to look-ahead into the hypothetical future of the agent.

We consider *state-space algorithms*, where a node contains a state and can be linked to other nodes, referred to as successors. This *linkage* represents the concept of transitioning from a state to another, after the execution of an action. The node with the previous state is called the parent of the node with the current. The one without a parent is called root and stores the current environment state the agent is in. We refer to a structure with all the nodes linked as *search tree*. The kind of algorithms that will be discussed have the objective of estimating more accurate state-values , $V(s)$, using the successor *subtrees*. For the remainder of this subsection, we assume the environment to be deterministic and the model to be known. We define the action-value, $Q(s, a)$, as

$$Q(s, a) = r(s, a, s') + V(s'), \tag{2.27}$$

Some environments are multi-agent and adversarial, for instance a 2-player board game, in which case, if the player of the successor nodes is an opponent, the action-value then becomes

$$Q(s, a) = r(s, a, s') - V(s'). \tag{2.28}$$

We will now discuss some planning strategies that will be relevant for our work.

Pure Monte Carlo Search

A very simple method to choose between a set of actions is to conduct several random *rollouts* starting at each successor node of the root. A random rollout is the choice of random actions until a terminal state and its value is the total reward accumulated. We can then set the state-value of each root successor to the average value of their rollouts. The larger the number of them, the more precise the state estimation, but the longer the search will take. The agent then chooses the action associated with the highest action-value.

Minimax

The algorithm recursively assigns values to nodes given their best successor (algorithm 2.2). The idea is that each node updates its state-value to match their highest action-value. If the tree is adversarial, this algorithm is called minimax, but, for simplicity, this name will be used even if it is not.

Algorithm 2.2 recursive minimax evaluation

```
1: function MINIMAX_EVALUATE(n)
2:     if n is not Terminal then
3:         for n' in n.successors do
4:             Minimax_Evaluate(n')
5:         end for
6:         s ← n'.s                                   ▷ get state of node
7:         V(s) ← arg max_a Q(s, a)
8:     else
9:         V(s) ← 0
10:    end if
11: end function
```

In practice, the whole tree is not evaluated, at least for very large domains. Usually, a maximum depth is defined and the values of the leaf nodes are estimated, either by rollouts or by an heuristic function. Only then those values are propagated backwards until the root (figure 2.1). In minimax, the *principal variation* is the path that starts in the root and ends in the leaf that propagated its value all the way to the top.

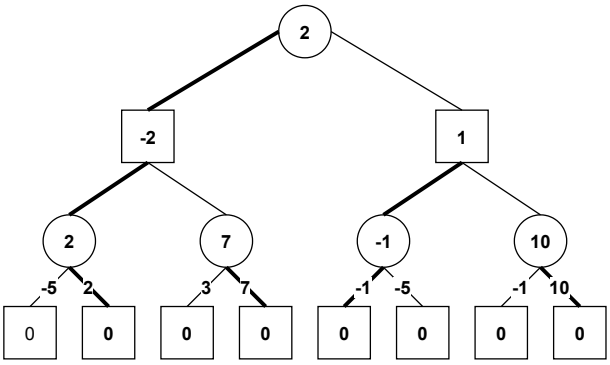

Figure 2.1: A minimax evaluation example. The circle and square nodes represent two adversaries. The leaf nodes, in this example, are all terminal nodes and have a value of 0 by definition. Its edges have a reward corresponding to the respective state transition. The other unlabeled edges have a reward of 0 in the example. The bolder ones show which nodes propagated their value to their parent. The bold path, that we can trace from the root to a leaf, is called the *principal variation*

Best First Minimax Search

Best first minimax search (BFMMS) [22] is a type of best first algorithm. These types of algorithms can be understood through the concept of iteration. The *best* leaf node in the tree (according to some criteria) is expanded in each iteration - its successors are added to the tree and usually some other operations, such as estimating their value, are done.

Each iteration in BFMMS descends the tree from the root until the leaf in the principal variation path, choosing always the best action-value. It then expands the leaf found, estimating the value of the new successors and propagating the maximum towards the root, just like normal minimax. The process is repeated for as many iterations as desired.

Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is also a best first algorithm and it is one of the most popular game tree search algorithms in artificial intelligence. The algorithm became so popular that many successful variations ended up being implemented [23]. We describe the most common one, but the overall idea is preserved among the others. In each iteration, the algorithm descends in a best first manner until encountering an unexpanded node. Then, it generates one successor and, to estimate its initial value, it plays one or more random rollouts starting from it. Next, this value is propagated backwards and sums to the total value seen in each node of that iteration's path. The value of each node is the average of all the propagated values that passed through them during the multiple iterations of the algorithm.

A straightforward variation, just as described in the previous algorithm, is to use a heuristic function instead of rollouts. Another is to generate all successors at once. All these variations preserve the basic idea of descending through a best first path and adding an estimated value to every node in the path.

Notes on Best First Algorithms

In best first algorithms, and as briefly mentioned before, it is possible to add exploration terms to their *best first equation*. The UCT, mentioned earlier, is the MCTS algorithm with an exploration component added when descending the tree. The main difference between BFMMS and MCTS is the philosophy behind the value of each node. In the first, the value is optimistic and it is the best possible value reached through that node. In the latter, the value is more prudent and it averages over all the best paths seen through that node. The latter escapes outliers more easily, but might require more iterations to be reliable, since it relies on the empirical mean of values. It is also worth noting that UCT has been shown to theoretically converge to a perfect minimax tree, for a infinite number of iterations [23].

3

Related Work

Contents

In this section, we review the work done regarding the AlphaGo family - AlphaGo, AlphaGo Zero, AlphaZero and MuZero. Then, some other relevant, but less impactful, work regarding model-based algorithms is discussed.

3.1 AlphaGo

AlphaGo [3] was the first computer program to beat the world champion in Go. Its algorithm is composed by three major steps. The first is to train two networks that parameterize a policy, p_σ and p_π, in a supervised manner on a dataset of expert moves. The difference between the two is the network complexity. The network parameterizing the policy p_π is *shorter* (and therefore faster, but less accurate) than p_σ. The second step is to instantiate p_ρ by copying p_σ and improve it by playing games against different (older) versions of itself. The third step is to train the evaluation network, v_θ, which is done by predicting the results in a game dataset generated by self-play using p_ρ. The true results are either $+1$ or -1, depending on whether the player won or loss, respectively. When playing the game, at test time, a Monte Carlo tree search algorithm is combined with the previous neural networks to provide considerably better moves. In their version of MCTS, each edge in the tree stores an action-value $Q(s,a)$, number of visits $N(s,a)$, initial probabilities $P(s,a)$. In each iteration, the algorithm will descend, choosing actions according to a variation of the PUCT formula [24],

$$\arg\max_a Q(s,a) + c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}, \tag{3.1}$$

where c_{puct} is a constant that mediates the balance between exploration and exploitation. When a leaf node with some state s^l is found, the probabilities $P(s^l,a)$ associated with each of its successor edges are initialized using the the policy network p_σ (the policy p_ρ did not perform better at this stage as compared to p_σ, being only used to train v_θ). Then, the value of this leaf node is estimated using both the value network, v_θ, and a fast rollout using the policy network p_π, according to

$$V(s^l) \leftarrow (1-\lambda) v_\theta(s^l) + \lambda z^l \qquad \lambda \in [0,1], \tag{3.2}$$

where z^l is the result of the rollout in the perspective of the player associated with s^l. Next, this leaf's evaluation is propagated backwards through the path until the tree root, updating Q. For every edge, (s,a), in the iteration path, we update both the action value,

$$Q(s,a) \leftarrow \frac{N(s,a)Q(s,a) + V(s^l)}{N(s,a)+1}, \tag{3.3}$$

and the visit count,

$$N(s,a) \leftarrow N(s,a) + 1\ . \tag{3.4}$$

We add the negative of $V(s^l)$ if the player of s is different from the player of the leaf. After all the iterations have been ran, the *real* action chosen by the agent is the one associated with the node with highest visit count.

3.2 AlphaGo Zero and AlphaZero

After its success, the team that developed AlphaGo released a more powerful algorithm, AlphaGo Zero [5], that could learn how to play Go without any prior knowledge, only by playing against itself. In 2017, an even more general version than AlphaGo Zero, called AlphaZero [25], was introduced. This version is pretty much the same as the previous, except certain details that were tuned specifically to the game of go were removed, in order for it to work in more board games. AlphaZero needs only the rules of the game to be given and it can learn how to play in any deterministic and completely observable environment. It became the strongest of all the programs, gaining the world champion title in chess, shogi and go.

We now describe the general idea behind the AlphaZero algorithm. Just like AlphaGo, it uses a policy and state-value function, p_ω and v_θ. The tree search algorithm is pretty much the same, descending the tree according to equation 3.1 until finding a leaf with some state s^l, only diverging with AlphaGo in the way that node is expanded. In AlphaGo Zero and AlphaZero, the initial value given to the node only uses the state-value function, becoming

$$V(s^l) \leftarrow v_\theta(s^l). \tag{3.5}$$

This state-value then gets propagated backwards as in equation 3.3 and the number of visits in each edge gets incremented, as in AlphaGo. In the end of all iterations, the root node describes an improved search policy vector, $\boldsymbol{\pi}$, where each component is given by

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}, \tag{3.6}$$

where τ is a temperature parameter that controls the amount of exploration. Invalid moves have a policy value of 0.

After a game ends, it is broken into tuples containing: a state in the trajectory, the result of the game in that player's perspective, z (-1,1 or 0, depending on whether it lost, won or tied, respectively), and the improved search policy, $\boldsymbol{\pi}$. These tuples are randomly sampled from the replay buffer and used to

update the neural networks through the following loss function:

$$l(\theta) = (z - v_\theta(s))^2 - \boldsymbol{\pi} \cdot \log p_\omega(s) + c||\theta||^2, \tag{3.7}$$

where the last term is the L2 regularization to prevent *overfitting* and c is the constant that controls it.

A final detail worth mentioning, in these two papers, the value and policy function are calculated with a *double-headed* network. This means their first layers are shared.

3.3 Muzero

The last algorithm released in the AlphaGo series was called Muzero [26] and, as it has been the trend in each new one, it it is even more versatile than the previous. For all the other algorithms, the model of the environment needs to be given. For Muzero, on the other hand, not even that much is required. The model is learned as the agent interacts with the environment. It is similar to AlphaGo Zero and AlphaZero in the overall idea. It will use a Monte Carlo tree search to play better moves and then update itself using the generated trajectories.

The algorithm does not learn to predict the environment states, but, instead, it learns *hidden states* that do not hold an explicit well-defined semantics. Unlike the previous algorithms, we do not assume the environment's observations to completely describe the inner state of the environment. We distinguish between observations, o_{t+k}, and the Muzero states, s_t^k, used during planning. In the notation[1], we use a superscript, k, to differentiate a time step, t, in the real trajectory, from a planning step during search. The state s_t^k corresponds to looking k steps ahead of observation o_t. Semantically, s_t^k corresponds to observation o_{t+k}. We omit the subscript t when the context so allows it. Unlike the previous algorithms that were made for board games, where there is only a final reward indicating who won, this algorithm assumes the existence of a reward in every transition. There are 3 functions used and learned throughout training. The first is called representation function h_θ and converts the current observation into an initial hidden state, $s_t = h_\theta(o_t)$. The second is called dynamics function, g_θ, and receives a hidden state and an action and returns both the next predicted reward and state, $r_\theta(s^k, a^k), s^{k+1} = g_\theta(s^k, a^k)$. The last one is the function that was common in the previous algorithm, called prediction function f_θ, and it receives a hidden state and calculates both the policy and value functions, $p_\theta(s^k), v_\theta(s^k) = f_\theta(s^k)$.

So, upon getting an observation from the environment, Muzero will start the planning by converting the current observation, o_t, to a hidden state, s_t^0. In every iteration after that, the search will descend the tree, choosing the next action using a slightly improved version of equation 3.1, until a leaf node has been reached. Then, just like in the last two algorithms, it runs the prediction function, f_θ, and it initializes the state-value and initial probabilities in the leaf node. The backpropagation idea is similar as

[1] we use a similar notation as the original paper [26]

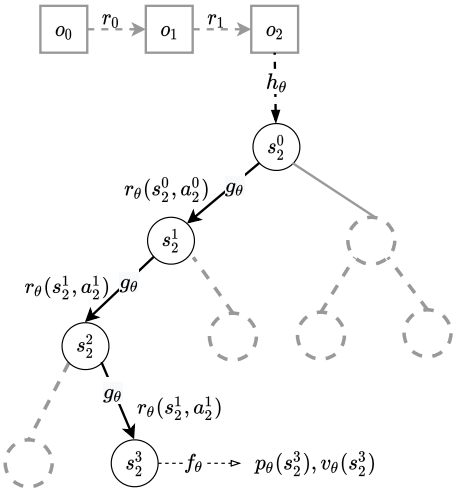

Figure 3.1: A Muzero MCTS iteration. The squares represent the game trajectory so far and the circles represent the tree nodes with their respective hidden states. Muzero is planning in the second observation. The arrows show the descending path taken in this iteration. The gray dashed ones were added in previous iterations.

before, but, now, as the leaf value backtracks and passes through the transitions, it accumulates their predicted rewards along the path. For example, to update the statistics of the k[th] node in the iteration path, the following update rule is used:

$$Q(s^k, a^k) \leftarrow \frac{N(s^k, a^k)Q(s^k, a^k) + G^k}{N(s^k, a^k) + 1}, \tag{3.8}$$

where G^k is the sum of the new estimated value of the leaf state, s^l, plus the total reward between the k[th] node we are updating and the leaf node. Formally, G^k is given by

$$G^k = \sum_{i=0}^{l-k-1} \gamma^i r_\theta(s^{k+i}, a^{k+i}) + \gamma^{l-k} v_\theta(s^l). \tag{3.9}$$

Lastly, we increment the visit count in each node. Figure 3.1 shows a search iteration.

Muzero's learned model assumes that every action is legal and terminal states do not exist. In other words, during search, the tree never reaches terminal states and maintains a constant branching factor. This means that, when all the iterations are complete, some iterations might have chosen illegal states, so the improved search policy, $\boldsymbol{\pi}$ (eq. 3.6), needs to be masked to only allow non-zero values for legal actions.

Regarding training, once a game has been played, it is stored. With each observation o_t, we also store the improved policy $\boldsymbol{\pi}_t$ and an n-step bootstrapped value, z_t, given by

$$z_t = \sum_{i=0}^{n-1} r_{t+i} + V(s_{t+n}^0), \tag{3.10}$$

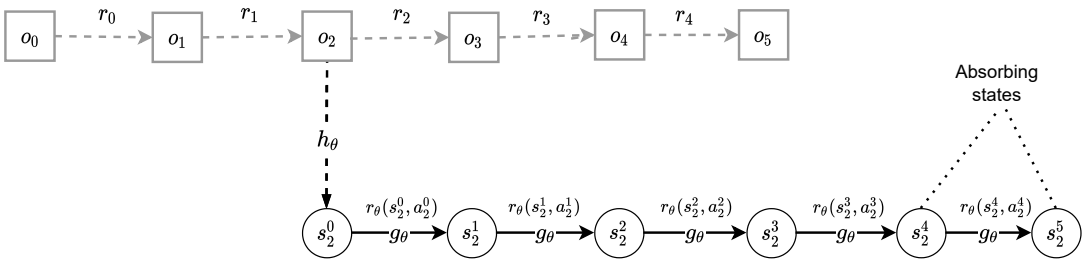

Figure 3.2: Muzero's unrolling process during training. An observation, o_2, from a game was sampled for training. The corresponding game is show in squares. The representation function h_θ is applied to this observation to get the initial hidden state. Then, the same action sequence that was used to play the game is applied when unrolling the hidden states, for e.g. the action used in the dynamics function g_θ to generate s_2^3 from s_2^2 is the action applied in the environment that led to observation o_2^5 from o_2^4. The unrolling happens for $K = 5$ steps. Random actions are used when the hidden state sequence surpasses the game as it is the case for the transition from s_2^3 to s_2^4 and from this to s_2^5. These two last states that do not align with the game are called *absorbing states*.

where $V(s_{t+n}^0)$ is the improved state-value stored in the root node of the tree at time step $t+n$, resulting from all the backups. This improved value is the average of all the propagated values until the root during search. The training part consists in sampling an observation of some stored game and unrolling it for K steps, as explained in more detail in Figure 3.2. The overall loss is

$$l_t(\theta) = c||\theta||^2 + \sum_{k=0}^{K} \Big[l^p(\boldsymbol{\pi}_{t+k}, p_\theta(s_t^k)) + l^v(z_{t+k}, v_\theta(s_t^k)) + l^r(r_{t+k}, r_\theta(s_t^k, a_t^k) \Big], \tag{3.11}$$

where l^p,l^v,l^r are the specific loss functions for policy, value and reward, respectively. The hidden states that go beyond the terminal observation, during the unrolling, are called absorbing states and the target rewards, r_{t+k}, and return, z_{t+k}, are 0. This makes the search avoid going beyond these states while planning. Regarding illegal moves, because the predicted policy will be updated to mimic the improved policy vector, where the illegal actions have a value of 0, the MCTS will tend to avoid exploring them.

3.4 Other Works

Value Prediction Networks

Value prediction networks (VPNs) [27] are, arguably, the predecessor of MuZero. This work introduced the idea of learning an abstract state model and using a planning algorithm with it. Both agents use a function that transforms observations into hidden states. However, VPNs do not use a policy function, but only a value function, and have one network to predict the reward and a completely different one to predict the next hidden state. The planning algorithm used is also different from MuZero's. VPNs uses a similar search algorithm to minimax, that descends through the best b nodes at each depth, pruning the

rest of the branches.

SAVE

A similar idea to AlphaZero is called search with amortized value estimates (SAVE) [9]. The algorithm, when expanding a node, initializes all its successors with an action-value function, $q_\theta(s, a)$, instead of using a policy function. The maximum action-value of the successors is then propagated backwards. This action-value function is updated based on two losses. The first is similar to AlphaZero and consists in using a trajectory to find a target value based on either a Monte Carlo or a bootstrapped value. The second fits the action-values predicted by the neural network with the improved action-values, Q, stored in the root successors after the planning step. In other words, the action-value function, q_θ, is updated using the *real* trajectory and the planning tree.

Alternative Planning Algorithms

In a recent paper [7], the algorithms Monte Carlo tree search, best first minimax search and minimax were compared for multiple games using a similar algorithm to AlphaZero (with a given model), showing that MCTS is not always the best.

Besides this, the paper also investigated the usage of the simulated data in learning. The previously seen algorithms discard most of the planning tree used. This work compared the usage of only trajectories with a strategy called *game tree bootstrapping* [28], that uses all the states of the planning tree and their backed up values to update its model. In their results, best first minimax was superior when combined with tree bootstrapping to MCTS using only trajectories.

4

Architecture

Contents

The architecture we propose is designed for MBRL agents. We conduct an overview of it and its separate elements. The architecture has the objective of allowing changes and extensions in each key component, without being too complex to use or understand.

A RL agent iterates repeatedly over two major steps. The first generates data by interacting with an environment and the second uses that data to update its functions, which provides for more accurate estimations. The way to do each of these steps can vary immensely according to each algorithm. The reasoning behind our work is to identify key steps in MBRL and implement each of these parts as independently as possible, delegating the responsibility to the user to assemble each block consistently, according to the intended agent. The interactions and dependencies between different blocks should be kept to a minimum and well defined.

A module conceptualizes an important component of the agent that we generalize. The architecture we propose has six: Environment, Loss, Model, Planning, Policy and Data. We consider every one of these except the Environment to constitute the concept of agent. The Loss, Model, Planning and Policy modules are part of the behavior of the agent, while the Data module concerns itself with the storage of data to be used between them. The purpose of each module is to have different algorithms that can be chosen to accomplish the module's *objective*. It should be easy to add a new algorithm to each module without changing the others. A simple scheme summarizing the structure of the architecture can be seen in figure 4.1.

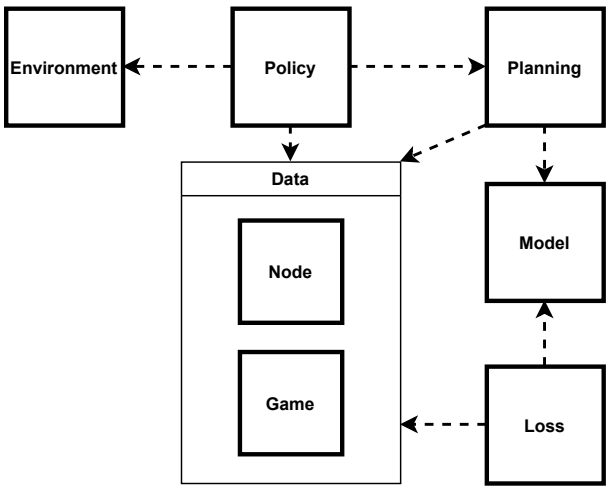

Figure 4.1: Basic architecture overview. The arrows show dependencies between modules. For instance, the Policy depends on the Environment and on the Planning.

4.1 Environment

A RL agent interacts with a problem, which we call environment. The different ways an agent can interact with an environment are called actions. After every action, there is some information given to the agent so that it can decide its next one, and so on until solving the problem. Its interface should, at least, provide for a mechanism to step through the episode given an action sequence and should return some information indicating its performance. The environment is completely independent from the agent - the latter should know how to interact with the first, but not vice-versa. These classes can implement themselves the problem we are trying to solve or simply serve as the intermediate between the agent and the concrete way to interact with the problem. For example, for a real world robot agent, this implemented class might receive the data from the sensors and structure it into observations (in case of a POMDP) and convert the agent's actions to the specific commands required to interact with the robot.

4.2 Model

This module's objective is to model, not only the environment, but also all the other functions required by the Planning module. How they are calculated concerns only each model. Different search algorithms might need new functions, so it is important for this component's implementation to be efficient and easily extendable. In this module, for the functions we want to learn, we assume them to be computed using neural networks, but do not make assumptions about the particular structure of these networks.

The architecture should support both MBRL agents that learn the environment's model and those that receive a perfect one. If this module is queried for a next state, if it knows the environment's dynamics, it can return the true next state. If it does not, it returns an estimation.

4.3 Data

The Data module is used to store and transport information between modules. We divide this one into two *submodules*. The first, the Node submodule, stores information about the planning done in each step of the episode. A node coincides with an environment state and it is created by the planning algorithm. The second is the Game submodule and it stores data returned by the environment and planning during the episode.

4.4 Planning

Just as in Sutton and Barto [11], in this work we consider *state-space* planning algorithms (as opposed to *plan-space planning* [29]). In these, actions mediate transitions between two states. In traditional deep reinforcement learning algorithms, in each step, the agent estimates the quality of the possible actions available using a neural network. The objective of the Planning module is to provide enough information about each action, in a given step of the episode, so the policy can choose the best possible action. However, it does this differently from traditional DRL algorithms. Instead of using a neural network alone to estimate the quality of each action, it uses a search algorithm in conjunction with the functions available in the model passed. The reasoning behind using a search algorithm is to look ahead into the space of (*hypothetical*) states to provide more accurate quality estimations about each action. It is important to notice that this module does not come up with a *plan* to use during the actual interaction with the environment. In fact, this module should not even be aware of the existence of an episode. It simply makes estimations just like a normal neural model would in DRL without planning. The root node, which represents the current state of the environment, will be returned to the policy and it will be used to make a decision about the next action to take.

Lastly, as mentioned, the planning class should receive a model. How the model works internally is irrelevant to the planning algorithm. As long as it can calculate all the required functions, it should work.

4.5 Policy

The Policy module has two responsibilities. It decides the action to take in every step of the episode and it needs to store the relevant data coming from the environment and planning in the Data module.

The policy iterates through the whole environment episode, balancing between exploration and exploitation. In every step, it calls a search algorithm, which returns a node with enough information about each action. Then chooses which one to take, according to its own algorithm (for instance, it might want to choose a random action with ϵ probability and the best one otherwise). It is also responsible for storing the whole episode information in a game instance, including the root node returned from the planning algorithm.

Each policy should receive the instance of the planning class to use. As long as the node returned by it has the interface needed by the policy, every planning algorithm should work. For instance, if the policy wants to use the number of times an action has been chosen, then any search algorithm that returns a node with this information (which is usually associated with best-first algorithms) should work.

In the end, after completing an episode, it will save in a game instance all the information returned by the environment and the nodes given by the planning.

4.6 Loss

After interacting with the environment, the model should be updated to be more accurate and insightful. This can be done using the experience stored with the Data module to calculate a loss value. This value is the result of a computational graph that is calculated using the neural networks in the model. It can then be used to update the parameters of the networks [30]. In general, the loss can be calculated using the trajectory stored or the nodes returned by the planning. Just like the policy, the loss class needs to know the node class passed and, just like the planning, it accepts any model as long as it can calculate the necessary functions needed to compute the loss value.

4.7 Summary and Final Notes

Our architecture provides these blocks, but assembling them together (in other words, creating the actual agent) is the responsibility of the user. In general, the user only needs to deal directly with the policy and loss interfaces. Regarding the other ones, the environment and planning should be instantiated and handled by the policy. The model is created and passed to the loss and planning classes. Lastly, the data ones are instantiated internally by the planning and policy, which are then returned by them.

In summary, the agent will ask the policy to generate data, which it does by interacting with the environment internally, using a planning algorithm to create a trajectory and saving the relevant information received from the environment and planning in a data class. This data is then returned by the policy to the agent. It can then be used by a loss algorithm to calculate a loss value in order to update the model.

5

Implementation

Contents

In this section, we explain in detail our implementation. We describe it from the perspective of someone interested in using our code, without over fitting to code details. The whole idea behind the implementation lies in module interactions, so we will explain these using the concept of a module's interface, the set of methods and properties that are visible to the other ones. Appendix A shows the class diagrams associated with each module.

Our implementation was done using Python 3[1], using Pytorch [30] to handle the tensors responsible for the machine learning part. Python is an interpreted language, which allows easily for code to be changed, without worrying too much about the program structure (as opposed to the usual compiled languages). As long as the one using the language is aware of the inner details of the code, python is extremely versatile. However, this advantage of the language can easily become its biggest problem. As implementations grow, it becomes unfeasible to simply trust the user to be aware of those many details in order to guarantee consistency. Deep learning code is notoriously known for being hard to debug, since, due to its adaptability nature, it can still produce satisfiable results while in the presence of errors. These problems can be managed if we apply good structural principles in our implementation.

5.1 Environment

The environment interface we use is similar to the openAI Gym environment [31]. These are modeled as POMDPs, which means they are episodic and start in an initial state, changing to the next one upon receiving an action by the agent. The observation does not necessarily reflect the complete internal state of the environment and the next state depends only on the action given and the current state, not on the previous environment history (Markov property). The first method that should be called is *reset*, that restarts the internal state of the environment. The *step* method receives the action number and returns three elements: an array corresponding to the next observation, the value of the reward associated with that transition and a boolean value about whether or not the episode has reached a terminal state. The action numbers start at zero and are sequentially numbered. Unlike the gym environments, our interface allows for the existence of multiple player games. At each step, the number of the current player can be found by calling *get_current_player*. The reward value is always given in the prespective of the player who just acted. Traditional problems with only one agent are conceptualized as a one-player problem, returning always the first player (number zero). Yet another extension from the gym interface is the existence of illegal actions. For instance, if we're playing tictactoe and choose to put a piece in a non empty bracket. There are two methods that can be called to know which are the available actions: *get_legal_actions*, that outputs a list with the actions' numbers, and *get_action_mask* that returns a vector with the action size where the zeros correspond to illegal actions and ones correspond to legal

[1] https://docs.python.org/3.0/

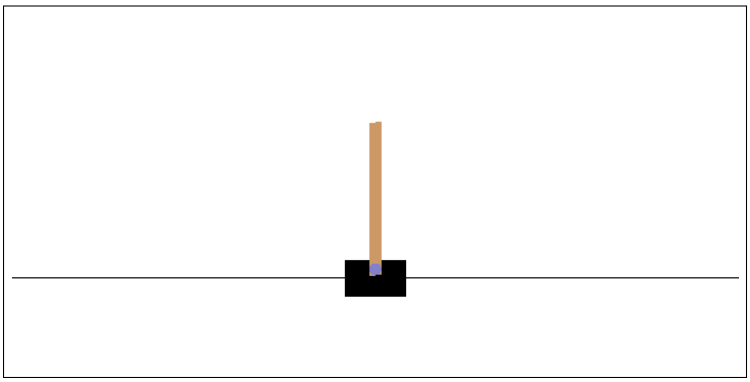

Figure 5.1: The cartpole image given by the gym environment when the method *render* is called after calling *reset*.

ones. The other useful methods that should be implemented in each environment are *get_action_size*, *get_num_of_players* and *get_input_shape*.

We implemented 3 environments: Cartpole, Minigrid and Tictactoe.

CartPole

Cartpole is one of the most well known environments from the openAI gym. It was described by Barto, Sutton and Anderson [32]. The idea is to balance a pole, which has one extremity fixated on a moving cart, by applying forces to the left and right of it (fig. 5.1). If the forces are applied correctly, the movement of the cart can balance the pole. The episode terminates when the pole is more than 12 degrees from its original position or the cart is further than 2.4 units from the center. The observation is an array with four values: cart position, velocity, pole angle and angular velocity. A reward of 1 is offered in each step, so the objective is to maximize the duration of each episode. We did not implement it from scratch. Our class simply adapts the original implementation, by calling it internally. We do have to complement it with the information about the action size (which is two), legal actions (both of them, except when the environment reached a terminal state) and players (only one player, the player number zero). In this way, because our interface is so close to the gym one, it is extremely easy to reuse gym environments without much effort.

MiniGrid

The minigrid environment[2] is a NxN grid. The agent starts in one of the squares and the objective is to reach a square goal (figure 5.2). Besides a position, at each moment, the agent has a direction it is pointing at. In our environment, the goal square is fixed, but we can optionally decide where the agent starts or choose for it to start in a random square each time it resets. This environment is partially observable, being possible to observe a grid of 7x7 in each step. The observations are given in 7x7x3

[2]https://github.com/maximecb/gym-minigrid

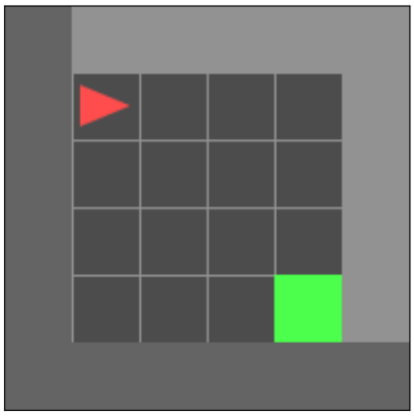

Figure 5.2: Rendered image of the initial state of a 4x4 grid world. The agent is currently on the top left corner and the goal is in the bottom right.

arrays. The rewards are 0 unless the goal is reached, in which case it is 1. There are three actions: turn left, turn right and move forward, in order. All the actions are always legal and when the agent tries to continue past a wall, it simply does not move. Upon initializing this environment we can limit the maximum number of steps and choose the value of N.

Tictactoe

Tictactoe is played in a 3x3 grid by two players. The two take turns filling an empty position in the grid. The first to fill three spaces in a row (horizontally, vertically or diagonally) wins. The observation is a numpy array with shape (2,3,3) - in other words, the observation is composed by two 3x3 arrays. The first has 1 in the positions where the current player has played and 0 otherwise, while the second has 1 in the positions the adversary has played and 0 otherwise. When initializing the environment, we can decide whether we want to play against an expert agent (in which case the environment has only one player) or we want to do *self-play*, meaning that the user of the environment chooses the actions for both players. The expert agent has three levels. The level 0 plays randomly. The next will play randomly except if it sees moves that will block the third opponent piece in a row. In level 2, it also recognizes immediate winning moves. In self-play, the reward is always 0, except for a play that leads to victory, which is 1. In single agent mode, the reward is 1 for an action that wins, -1 for an agent that allows the agent to win and 0 otherwise.

Final Remarks

Often, some algorithms and implementations need to change the structure of the environments in specific ways. For instance, Alphazero adds a stack of previous observations to the current one as input. It is also common to change the structure of rewards by scaling it by some factor or adding a stronger

penalty per step in the environments with sparse rewards. It is already customary in the openAI gym to use environment wrappers that perform these operations. In our implementation, we follow the same reasoning and assume these types of particular changes, to the outputs of the environment, to be the responsibility of this module, freeing the others from these details.

Something else worth asking is why is it that the openAI gym does not need action masks and ours does. The gym environments circumvent this need by, whenever an agent takes an illegal action, the internal state does not change and, sometimes, a negative reward can be given to disincentivize these actions. This solution can be applied to environments under our interface and, in fact, it is the one used with minigrid. However, the problem starts when we have more than one player. In tictactoe, players play in an alternating order. If the first player chooses a non-empty position on the board, under this solution, the player would still be the same in the next turn, which would break the consistency between player turns and force the planning algorithms to learn how to predict the next player.

5.2 Base Agent

Before advancing to the specific implemented details of the agent's modules, we explain here the reasoning behind the algorithms implemented. The purpose of the architecture is to be extendable. Obviously, we did not implement everything that it is possible. Strictly speaking, we can not say we implemented *an agent*, since each module has several alternatives that can compose different ones. However, there is a base idea for the algorithms in each module and that is what we explain here. To simplify, we denote by this idea by *base agent*. This agent is designed to operate in environments with unknown models.

In AlphaZero and MuZero, there are two relevant functions that impact the descending of the tree in the planning algortihms: the policy and the state-value function. For every node, its policy is calculated when that node's parent is expanded and the state-value is only calculated afterwards, when the tree descends to this node and expands it. In MuZero, the policy value, that mediates exploration, is set to zero for invalid actions, so the problem of expanding an invalid node does not exist. But what about algorithms that are not best first search, like minimax, and can not use a policy function to identify the invalid actions? Even if we consider best first search algorithms, recent proposed ones inspired in AlphaZero, that use planning [7, 9], have used best-first functions to descend the tree that, instead of estimating the state-value of the expanded node, they estimate its successors'. Meaning that, for invalid nodes, even if the policy is zero, if the estimated value is not, the algorithm might still choose to descend to these (see eq. 3.1). Since we are interested in a versatile base agent that works for diverse components, we propose a solution that, not only allows for the latter types of best-first search, but for non-best first algorithms like minimax. Our base agent uses a much simpler idea. It predicts and uses a mask vector or, in other words, it predicts directly which actions are legal and which are illegal.

Regarding other properties of the base agent, all the search algorithms we implemented estimate the state-value of the successor nodes, instead of the expanded ones, and do not use a policy function. Like MuZero, they predict the rewards received after taking each action and also use an encoded state without explicit semantics. We let the specific MuZero implementation as an extension for future work.

5.3 Model

The model has the most challenging interface to design due to its very demanding requirements. This module needs to be as efficient as possible, since it is used intensively in the Planning and Loss modules. Its internal structure (the neural network architectures) needs to be able to vary without constraint. The most demanding problem, however, is its need to accommodate any combination of functions, being easy to add new ones, in case a new search algorithm requires it. As an example of the problems that might arise, consider two arbitrary functions (for instance, the state-value and policy functions). If we want to use a double headed architecture [5], which means that both these functions share intermediate calculations, we might have a single method that returns both these results while, inside, reusing the common calculations, guaranteeing maximum efficiency. But what if we only require one of these operations later? With only a single method, we would be inefficiently calculating both functions when only one is necessary. To solve this, it might be tempting to create one method that returns both functions, and one method for each of the two alone, resulting in all possible combinations. But what if we want to extend and add another function to the model? Now, we would have to add four methods (for a total of seven). If we have four functions, 15 methods are needed. We can do better than this combinatorial demand.

The solution implemented separates conceptually each different function into different classes that we call *operations*. A specific model class should inherit all the operations corresponding to the functions it can calculate. In this way, any other module can verify if the model passed is capable of calculating all the functions necessary to its functioning (which also turns the code considerably more understandable). In order to allow each model to have complete control over the way these functions are calculated, these operations will share the same method. This method receives the information about which functions it should calculate as an argument, returning all the required results. We can see this as performing a *query* to the model: we tell it what to calculate and, internally, this method does so for the requested functions as it sees fit. In practice, however, we do not have only one method in every model, but one per type of input. So, for example, operations that require a hidden state to be passed share the same method between themselves but do not share it with those that require a state and an action as input. To maintain all of this logic structurally organized, there is a hierarchical class structure used with four inheritance levels. On top, there is the abstract class of all the operations. Inheriting from this one,

there are the classes that are associated with each particular type of input and define the interface of the query method associated with it. The next layer corresponds to operation classes that correspond semantically to a particular function. These have an attribute *key* that should be passed to the query method when the corresponding functions need to be calculated. Lastly, we have the specific class models that implement the operations it inherits from.

There are five specific functions used by our base agent. The first, representation function, h_θ, receives an observation and returns an encoded hidden state. The next two, state-value function, v_θ, and mask function, m_θ, receive an encoded state and return, respectively, the estimated value for that state and a predicted mask vector for its successors. Each index in the mask represents whether we estimate the action to be legal or illegal, respectively giving it values closer to 1 or to 0. The last two, reward, r_θ, and next state, g_θ, functions receive an encoded state and an action; and return, respectively, the predicted reward associated with that transition and the next predicted state. In practice, to be more efficient, the operations that calculate these functions receive a batch of arguments to allow the model to take advantage of the parallelism of tensor operations. Details, for instance, maximum batch size allowed, shape of the encoded state, vector normalization... should be left to the inner implementation of each model and their constructor.

Designing good neural networks is a typical deep learning problem and it is usually task specific, so it is not our focus here. We implement one model class (called *Disjoint multilayer perceptron*) that is general enough for our base agent and supports the previously described five functions. Each one of them is internally computed by an independent multi-layer perceptron. The structure of our implemented class is shown in figure 5.3 and an example describing (in pseudo code) its behavior is given in algorithm 5.1.

Algorithm 5.1 example of a function that receives an observation and an action, and returns the predicted reward and value of the state that results from the transition. In line nº4 we want both the reward and next state, so we pass both keys. In line nº5, if besides the value, we wanted to predict the mask, we would have to add the key corresponding to the mask.

```
1: function TRANSITION_REWARD_VALUE(o,a)
2:     model = DisjointMLP(args)
3:     s ← model.representation_query(o,RepresentationOp.KEY)
4:     r, s' ← model.dynamic_query(s,a,RewardOp.KEY,NextStateOp.KEY)
5:     v ← model.prediction_query(s',StateValueOp.KEY)
       return r, v
6: end function
```

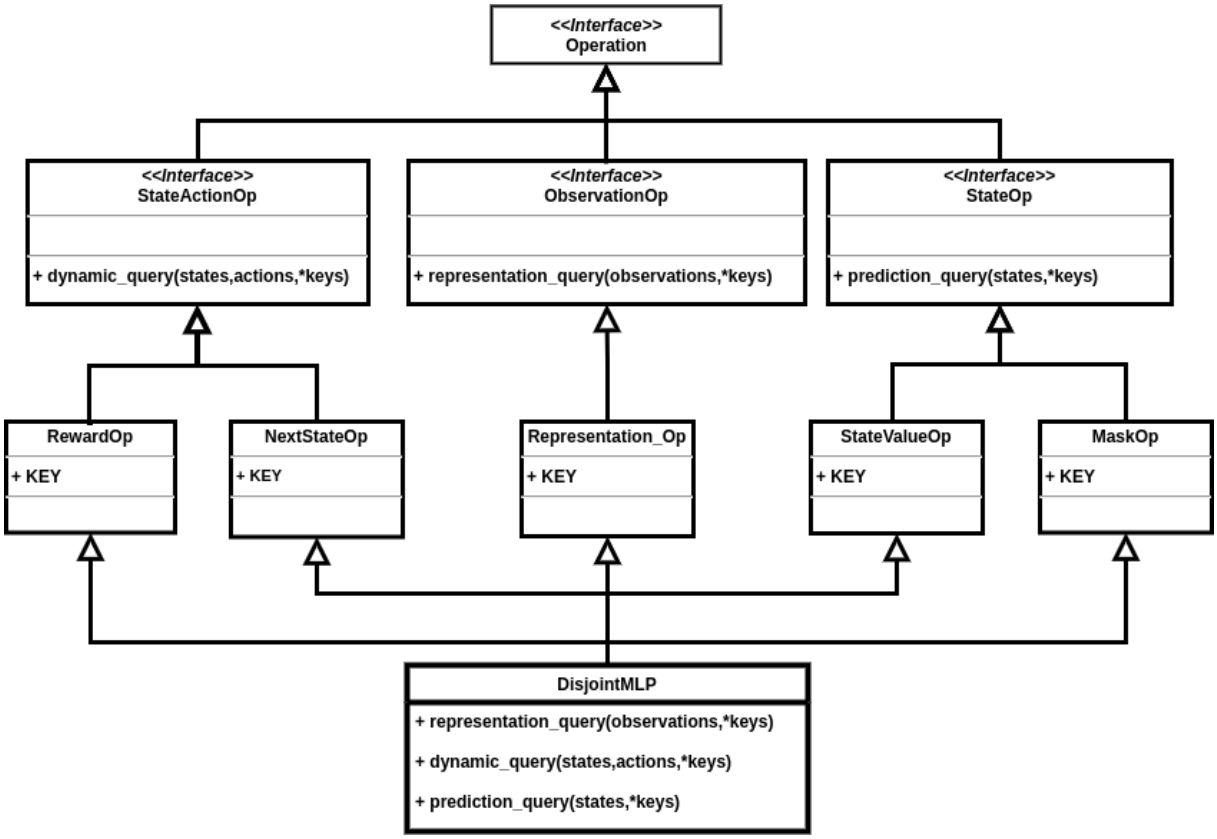

Figure 5.3: Class diagram of the implemented model, showing the operation classes it inherits from. The DisjointMLP model is therefore the combination of these 5 operations. When calling one of these methods, we have to pass the keys of the operations we want to calculate. How they are calculated is a responsibility of this model.

5.4 Data

The Game submodule is implemented as a single class. Since the goal of this submodule is to group the whole episode and planning information, and the data received by the environment is well defined, a simple class that saves the sequence of observations, rewards, nodes and relevant properties about the environment (array shape of the observation, number of actions, number of players) should suffice.

Regarding the Node submodule, its purpose is to be able to store properties and values necessary to the planning algorithm. There are three implemented. A basic node, the parent class of every other, that is associated to an hypothetical state of the problem, which is represented by a encoded tensor state s. The properties that can be retrieved by the basic node are a state-value, $V(s)$, that predicts the expected accumulated reward after that state, a mask vector that holds the validity value, $M(s, a)$, associated with an action a, the successor state of each transition, $S(s, a)$, the state's player, $Pl(s)$ and the reward stored per transition, $R(s, a)$. The second node class implemented is an extension of the first. It is supposed to be used by best first algorithms and, additionally, it stores the number of times each node has been visited by the algorithm, $N(s)$. The last node is yet an extension of the best first node and it is designed to be used only by a Monte Carlo tree search. It stores a sum of state-value estimations, $T(s)$, and it redefines the calculation of the value estimation to be the average of them,

$$V(s) = \frac{T(s)}{N(s)}. \tag{5.1}$$

The specific way these nodes are used can be better understood in the next section.

5.5 Planning

Our planning algorithms have a simple interface only composed by one method, *plan*, that receives the environment's current observation, player and mask. They support environments that are either single player or two players playing alternatively. After the search is done, it will return a (root) node associated with the current state of the environment. A tree structure resulting from the search algorithm is attached to this node, recursively through node successors. The general idea of these algorithms is to find more accurate estimations of a node's state-value by using the values of its successors. When the context allows it, we will use interchangeably the term node and state (since the properties stored in the node refer to its state).

We implemented four search algorithms. They all use the five functions discussed before. An encoded state and a mask vector is predicted for every state in the tree. We define the action-value of a node to be

$$Q(s^k, a) = \begin{cases} R(s^k, a) + V(s^{k+1}) & \text{if } Pl(s^k) = Pl(s^{k+1}) \\ R(s^k, a) - V(s^{k+1}) & \text{otherwise} \end{cases}, \tag{5.2}$$

where $s^{k+1} = S(s^k, a)$. When relevant, for clarity, we use a superscript k to denote that the state s^k was created looking ahead k steps[3], we can think of it as depth of the state in a planned trajectory. The action-value can be seen as a 1-step bootstrapped value.

We will also define the notion of masked action-value, used when considering actions,

$$Q_M(s,a) = M(s,a) \cdot Q(s,a) + \Big(1 - M(s,a)\Big) \cdot c_{\text{penalty}}. \tag{5.3}$$

For a perfect mask, we see that if the state is valid then the masked action-value is equal to the simple action-value, otherwise it is equal to the penalty constant, $c_{penalty}$, which penalizes state invalidity. Without the penalty, an invalid successor will have the value of zero, which might still be better than the alternative valid successors. An invalid state should never be chosen so, in theory, the constant c_{penalty} should be minus infinity. However, in practice, the predicted masks are not perfect so we have to choose a milder value, for instance a lower bound of the environment's reward function. The penalty term should only be used to choose an action, not to estimate the value of a state.

The four algorithms implemented start equally. They begin by creating the root node and the first encoded state using the representation function, h_θ, based on the observation passed as argument. They also save the player and mask passed in this node. After this, the algorithms start differing.

Breadth First

There are two breadth first algorithms: *minimax* and a variation of our own that we call *averaged minimax* (it resembles the expectiminimax algorithm [33]). Both incrementally build a tree until depth d. In each iteration, they gather all the current leaf states and expand them until reaching the maximum depth, one depth level at a time (in a *breadth first* manner). During a node's expansion, the expanded state's mask, the successor's encoded state and respective transition reward are predicted. If the environment has two players, the new nodes will be instantiated with the opposite player. The nodes at maximum depth have, in addition, their state-value also predicted. After the whole tree has been generated, it is time to propagate the leaves' state-values backwards until the root. The difference between these two algorithms is in the backup rule. For some state s, the backed up value in the averaged minimax is given by

$$V(s) \leftarrow \frac{\sum_{a \in \mathcal{A}} M(s,a) \cdot Q(s,a)}{\sum_{a \in \mathcal{A}} M(s,a)}. \tag{5.4}$$

For a perfect mask, we can see that the value of each state becomes the average of all the legal action-values. On the other hand, in minimax, the best successor is the one that maximizes the masked action-value, but, when updating the actual state-value, we assume the successor to be completely

[3] The same notation as in MuZero, described in the Related Work section

valid,

$$V(s) \leftarrow Q(s, \arg\max_a Q_M(s,a)). \tag{5.5}$$

Best First

The other two implemented algorithms are of type best first: best first minimax search and Monte Carlo tree search, both using a variation of the upper confidence bound (UCB) formula (used in the UCT algorithm) to take into account the mask, resulting in what we call masked upper confidence bound (MUCB),

$$\arg\max_a \left(Q_M(s,a) + c_{mucb} \cdot \sqrt{\frac{\log(N(s))}{N(S(s,a))+1}} \cdot M(s,a) \right), \tag{5.6}$$

where c_{mucb} is a constant that mediates the intensity of the exploration.

In an iteration, when expanding a leaf node, instead of estimating this state's value using the model, the value of the successor states are the ones estimated. Besides the value; the mask, the next encoded states and rewards are predicted. The difference between MCTS and BFMMS is also in the back propagation part. In the BFMMS, the update rule is the same as in minimax (equation 5.5). In the MCTS, similar to the traditional algorithm, the back propagation will add to each node at depth k, seen during this iteration, a path value G^k. For the leaf, this value is given by

$$G^k = \frac{\sum_{a \in \mathcal{A}} M(s^k, a) \cdot Q(s^k, a)}{\sum_{a \in \mathcal{A}} M(s^k, a)}. \tag{5.7}$$

In other words, the masked average of the leaf's newly created successors is added to it. For every node in shallower depths after that, the added quantity is recursively given by

$$G^k = \begin{cases} R(s^k, a^k) + G^{k+1} & \text{if } Pl(s^k) = Pl(s^{k+1}) \\ R(s^k, a^k) - G^{k+1} & \text{otherwise} \end{cases}, \tag{5.8}$$

where $0 \leq k < d$. In simpler words, G accumulates the rewards as it backtracks. This path value is added to each node and its improved state-value is now given by the average of all these path values,

$$V(s^k) \leftarrow \frac{\sum_{i=0}^{N(s^k)} G_i^k}{N(s^k)}. \tag{5.9}$$

Mask limitations

Finally, it is worth noticing that the mask is not useful when conducting updates when the algorithms are past terminal nodes and every successor is invalid. In these cases, the BFMMS and minimax will still choose an invalid successor and the MCTS and averaged minimax will still conduct an average over the values of invalid states. This is however not a problem, since, as we will see, for the specific case where

states are invalid because they are past a terminal node, the loss algorithms will teach the model to set their values to zero.

5.6 Policy

The policy classes receive an environment and have an interface with one method, *play_game*, that returns a played episode, by interacting with the environment. The common behavior of the policies implemented is to, in each step of the environment, make use of a planning algorithm to decide on the next action. As the interactions with the environment happen, the relevant information (observations, rewards, actions, nodes returned by the planning algorithm) is stored in an instance of the Game class and then returned. They differ in how they use the node given by the planning algorithm to decide the next action.

We implemented three specific policies: The first is an ϵ-value greedy. After calling the planning algorithm and receiving the root node, it will choose with 1-ϵ probability the action with the highest action-value and choose a random one otherwise. The second and third policies are only applicable to best first algorithms since they base their action choice on the number of times a node has been visited. The second policy is ϵ-visit greedy, which does the same as the first policy, but instead of judging based on the highest action-value, it uses the highest number of visits. The third and last policy is the one used by AlphaZero and MuZero. The policy, π, is based on a visit count distribution described by

$$\pi(a|s) = \frac{N(S(s,a))^{1/\tau}}{N(s)^{1/\tau}}, \tag{5.10}$$

τ is a *temperature parameter*, influencing the exploration/exploitation trade-off. The lower it is, the greedier the policy becomes.

5.7 Loss

In this module, a loss value should be calculated given a node. Because tensor operations can be easily parallelizable, the module's interface receives a list of nodes in the method *get_loss* and returns a single loss value, which might result, for instance, from the averaging of the losses of all the nodes in the list. Each node has access to its game instance, which means it also has access to all the actions, observations, masks, players, rewards and nodes of its episode. We chose the unit of learning to be the node and not the game since, when one node is a passed, only one game is also passed, while the opposite is not true.

We have implemented three loss classes. All of them calculate the loss values associated with

the state-value, mask and reward predictions, using a similiar unrolling process as MuZero and value prediction networks (figure 3.2). The state of the respective node is unrolled for k steps (the next k states are predicted using the actions taken during the real episode). The masks and transition rewards are predicted for the unrolled simulated trajectory and a loss measuring the difference between the predicted and real masks and rewards is calculated. Regarding the state-value prediction, the target value of the absorbing states, those past the terminal one, is 0. For non-absorbing, it depends on which loss class we choose. We implemented three different ones: *Monte Carlo loss*, *Offline Temporal Difference* and *Online Temporal Difference*.

In the Monte Carlo loss, the target value, for a specific state is the (discounted) sum of rewards from that state until the end of the trajectory. In both the offline and online temporal difference, the target is a n-step bootstrapped value. The difference between the two is where this value comes from. The offline TD is the one used in Muzero, where this value is the one stored in the root node returned by the planning algorithm. However, if we are using a replay buffer and reusing the same nodes to calculate our losses, this value stored in the node can easily become outdated - in fact, there is a version of MuZero described in the original paper [26] called *MuZero reanalyze* that periodically redoes the planning of some previous nodes, in order to update this stored value. We experiment with a simpler idea and bootstrap using a value re-estimated every time we calculate the loss. This is done by applying the representation funtion, h_θ, to the bootstrapped observation, followed by applying the value function, v_θ, to the resulting encoded state, obtaining a new value estimation for it. We call this idea *Online Temporal Difference* and a it is illustrated in figure 5.4.

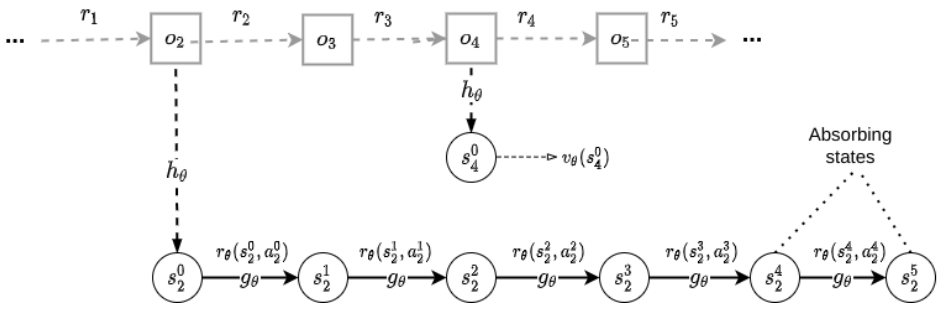

Figure 5.4: Online temporal difference loss with a 2-step bootstrapped target. The value of state s_2^0 is updated based on the target value $r_2 + r_3 + v_\theta(s_4^0)$.

Each loss class does not need to work for every node class. In fact, if we want to extend our implementation to use a policy function (as in AlphaZero and MuZero), a class responsible to calculate the loss for the policy would only work with best first nodes, since these are the ones that store the number of visits. This is simply a reflection on the fact that the Loss module might update the model using specific planning information, which is different according to the search algorithm used.

5.8 Summary and Final Notes

We create the components, but the user is the one who has to assemble these blocks according to the desired logic. An example of the logic of how to assemble these blocks in order to create a simple agent is shown in algorithm 5.2.

Algorithm 5.2 Pseudo code showing how to assemble the architecture blocks to create a simple agent that runs for N iterations.

```
1:  function SIMPLE AGENT(N)
2:      env = Environment()
3:      model = Model()
4:      planning = Planning(model)
5:      policy = Policy(env,planning)
6:      loss = Loss(model)
7:      optimizer = SGD(model)                          ▷ Pytorch optimizer
8:      for i in 0...N do
9:          game = policy.play_game()
10:         loss_value = loss.get_loss(game.nodes)
11:         optimizer.zero_grad()
12:         loss_value.backward()                       ▷ tensor operation
13:         optimizer.step()                            ▷ optimizer update
14:     end for
15: end function
```

In practice, the *get_loss* method returns, besides the loss value, a dictionary with relevant information that is class specific. It should pass intermediate steps that might be useful for *logging* or reusing.

There are other classes implemented that are not strictly necessary for the architecture to work, but are useful. We implemented two replay buffers: one that samples nodes uniformly and another that samples according to a priority associated to each node [19]. Also, to facilitate the optimization, we implemented an optimizer class that is capable of clipping the gradient norm, which is particularly useful to avoid exploding or vanishing gradients [1].

5.9 Example

We now show how to instantiate a simple agent using our code. In this example, our agent will act in cartpole using minimax and a ϵ-value greedy policy. First, we instantiate the environment and limit it to a maximum of 500 steps (lst. 5.1).

Listing 5.1: Instantiating the cartpole environment.

```
environment = CartPole(500)
```

In our model, we use two layers with 80 neurons, each, for the value function and one layer with 100 neurons for the rest (lst. 5.2).

Listing 5.2: Instantiating the implemented model, DisjointMLP.

```
model = Disjoint_MLP(
    observation_shape = environment.get_input_shape(),
    action_space_size = environment.get_action_size(),
    encoding_shape = (8,),
    fc_reward_layers = [100],
    fc_value_layers =  [80,80],
    fc_representation_layers = [100],
    fc_dynamics_layers = [100],
    fc_mask_layers = [100]
)
```

For planning, we use a minimax with a maximum depth of 3 and an invalid penalty of -1 (lst. 5.3).

Listing 5.3: Instantiating a 3-depth minimax planning algorithm.

```
action_size = environment.get_action_size()
num_of_players = environment.get_num_of_players()
max_depth = 3
invalid_penalty=-1
planning = Minimax( model,
                    action_size,
                    num_of_players,
                    max_depth,
                    invalid_penalty)
```

A ϵ-value greedy policy is used with 5% exploration. The reduction argument serves to indicate to the policy that is has to prune the node structure returned by planning (only keeping the root node), in order to save memory (lst. 5.4).

Listing 5.4: Instantiating an 5%-value greedy strategy policy.

```
epsilon = 0.05
reduction = 'root'
policy = EpsilonGreedyValue(environment,
                            planning,
                            epsilon,
                            reduction)
```

We use a Monte Carlo loss with 5 unroll steps and a γ discount of 0.99 (lst. 5.5).

Listing 5.5: Instantiating a Monte Carlo loss with 5 unroll steps.

```
unroll_steps=5
gamma_discount=0.99
loss_module = MonteCarloMVR(model,unroll_steps,gamma_discount)
```

Lastly, we instantiate two auxiliary classes. The first is a uniform replay buffer (lst. 5.6).

Listing 5.6: Instantiating a uniform replay buffer

```
max_buffer_size = 1000
storage = UniformBuffer(max_buffer_size)
```

and the second is an optimizer that will clip the norm after it surpassed the value of 20. We put a default optimizer and scheduler in the DisjointMLP model that we pass to the optimizer, but any other would work as well (lst. 5.7).

Listing 5.7: Instantiating an optimizer

```
max_grad_norm = 20
optimizer = SimpleOptimizer(model.parameters(),
                            model.get_optimizers(),
                            model.get_schedulers(),
                            max_grad_norm)
```

Now that we have instantiated all the necessary blocks, we have to assembled them to create the agent's logic. We will use the same logic as the algorithm 5.1 and update for 1000 episodes with 10 updates in each one and a batch size of 128 (lst. 5.8).

Listing 5.8: Assembling the instances of each module in order to create a simple training logic.

```
    episodes = 1000
    updates_per_episode = 10
```

```
    batch_size = 128
    scores = []
    for ep in range(episodes):
        game = policy.play_game()

        score = sum(game.rewards)
        print("episode:"+str(ep)+ " score:"+str(score))

        storage.add(game.nodes)

        for lep in range(updates_per_episode):
            nodes = storage.sample(batch_size)
            loss, info = loss_module.get_loss(nodes)
            optimizer.optimize(loss)
```

6

Experiments

Contents

In this section, we experiment our implementation using the three environments implemented (cartpole, minigrid, tictactoe). For every environment, we will try to find good agents - good combination of algorithms for each component. We will not try every single combination of algorithms. Instead, we run three different progressive experiments per environment. The first compares planning algorithms. The best one is then used in the second experiment to compare loss algorithms. The chosen planning algorithm with the best loss is then used to find the best policy, leading to the final agent.

For every environment, we play a full episode, store it, then conduct 10 updates, each with a batch size of 128 nodes. We show the results as a function of training steps, which are the number of updates done. In each experiment, the selection criterion is the algorithm with the highest average in the final step.

As default, the loss used is the Monte Carlo one and the policy used is ϵ-value greedy, whose parameters can be seen in each environment's second and third experiment, respectively.

6.1 Cartpole

For cartpole, we limit each episode to a maximum of 500 transitions and report the results for each episode as the average of the last 100. We use our Disjoint MLP with one layer of 80 neurons in every operation. The encoded state used has size 12. We use a uniform replay buffer with a maximum capacity of 10^5 transitions. Cartpole does not have illegal actions, our masked penalty used here is 0 as default. We ran every experiment for 1000 episodes (or 10000 training steps). Each is repeated 10 times and averaged. The summary of this experiment can be seen in table 6.1.

Planning

Cartpole has a branching factor of 2. In every planning algorithm we will expand 16 nodes. In the breadth first algorithms this corresponds to exploring a tree of depth 4. The results can be seen in figure 6.1. The best algorithm was the averaged minimax, which is used in the next experiments.

Loss

For the loss algorithms, 10 unroll steps and a gamma discount of 0.95 are used. For the temporal difference losses, the bootstrapping step is 10. The results can be seen in figure 6.2. The best loss algorithm was the offline temporal difference.

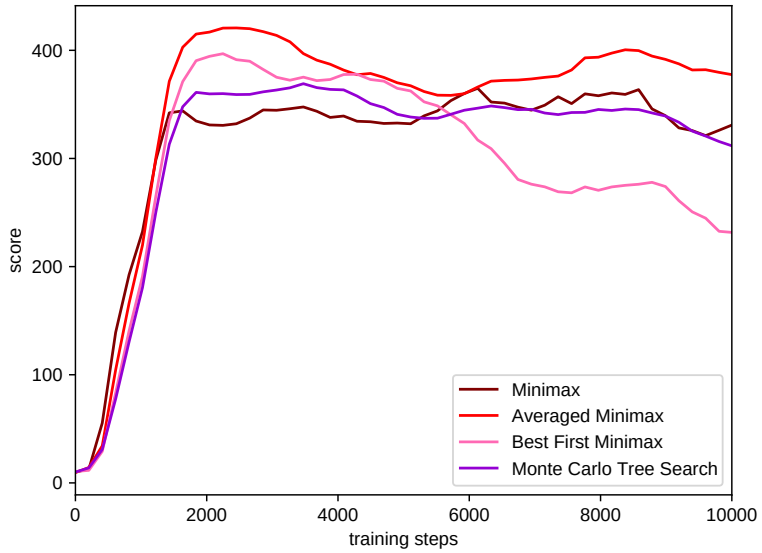

Figure 6.1: Comparison between planning algorithms for averaged minimax in cartpole

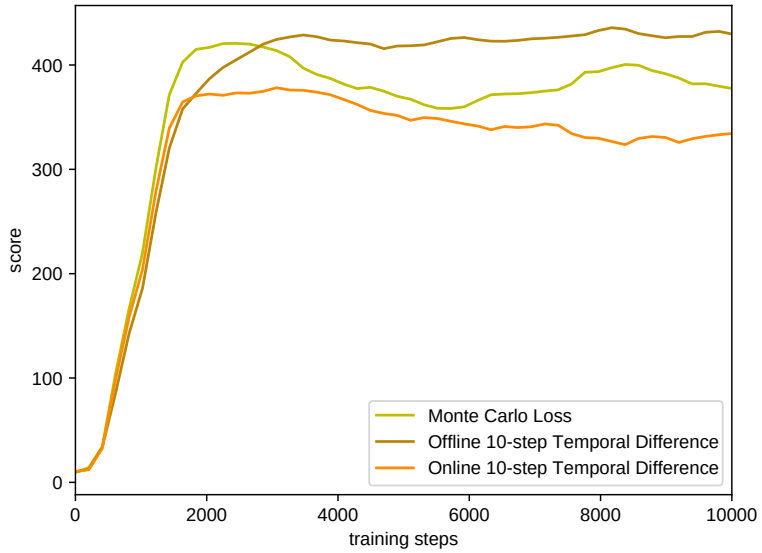

Figure 6.2: Comparison between loss algorithms for averaged minimax in cartpole

Policy

There is only one policy that is compatible with breadth first algorithms. In order to compare different policies, we picked MCTS, which was the best best-first search algorithm in the first experiment, ran the loss experiment to find the best loss for it and compared different policies. The loss experiment with

Results for Cartpole						
Search	Loss	Policy	Max	Min	Mean	Median
Avg. Minimax	MC loss	ϵ-value greedy	463.0	188.6	377.4	407.8
Avg. Minimax	ON TD	ϵ-value greedy	393.3	255.5	334.4	334.4
Avg. Minimax	OFF TD	ϵ-value greedy	476.5	**318.7**	**429.6**	459.6
BFMMS	MC loss	ϵ-value greedy	450.7	119.5	231.6	227.6
Minimax	MC loss	ϵ-value greedy	401.6	258.0	331.1	335.7
MCTS	MC loss	ϵ-value greedy	473.1	142.6	311.5	335.2
MCTS	ON TD	ϵ-value greedy	471.3	160.4	370.7	375.6
MCTS	OFF TD	ϵ-value greedy	**496.1**	202.8	392.8	**468.1**
MCTS	OFF TD	ϵ-visit greedy	476.2	138.6	360.7	410.9
MCTS	OFF TD	Visit distribution	494.0	139.6	303.0	303.3

Table 6.1: Summary of the final results of the agents tested for cartpole

MCTS is given in figure 6.3. The three loss algorithms have similar results, but the offline TD was slightly better in the final result.

For the policy experiment, we use 5% exploration for the ϵ-greedy strategies and a temperature of 1 for the visit distribution policy. The results can be seen in figure 6.4. The final best agent found, according to the maximum mean, was the averaged minimax with offline temporal difference and ϵ-value greedy.

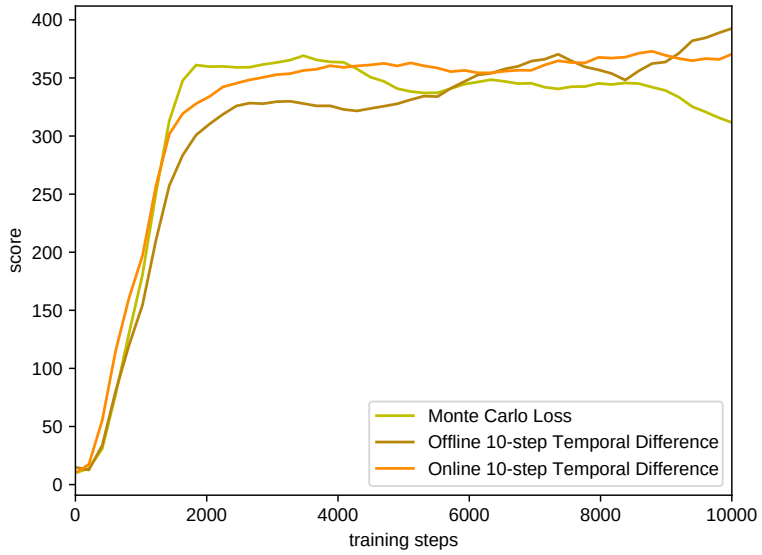

Figure 6.3: Comparison between loss algorithms for Monte Carlo tree search in cartpole

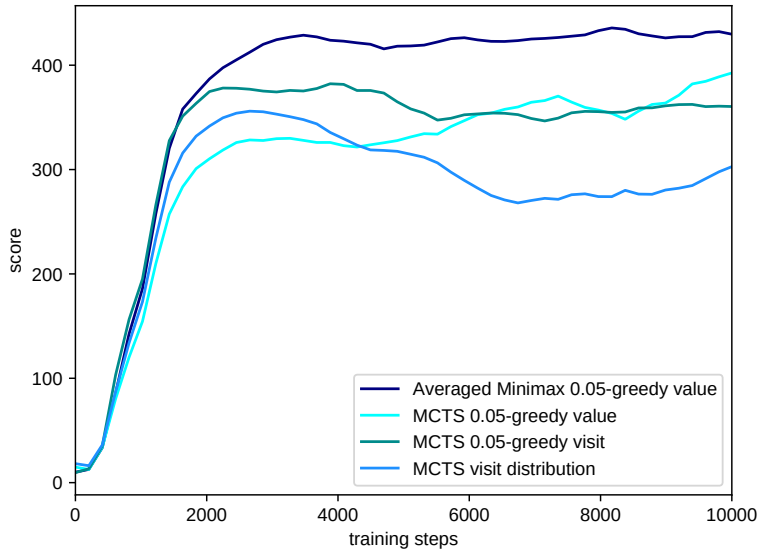

Figure 6.4: Comparison between MCTS policies and the best averaged minimax agent in cartpole

6.2 Minigrid

We instantiate the minigrid environment with a 6x6 grid. Each episode starts at a random position and is limited to a maximum of 15 transitions. Regarding our agent's parameters, our Disjoint MLP has one layer of 100 neurons in every operation. The hidden state has size 15. We use a prioritized replay buffer with a maximum capacity of 25000 transitions. Minigrid does not have illegal actions, so our masked penalty used here is the default value of 0. We ran every experiment for 2000 episodes (20000 training steps). Each is repeated 10 times and averaged. The results are given as the average of the last 100 episodes. The summary of this experiment can be seen in table 6.2.

Planning

Minigrid has a branching factor of 3. In every planning algorithm we will expand 27 nodes. In the breadth first algorithms, this corresponds to exploring a tree until a maximum depth of 3. The results can be seen in figure 6.5. The best algorithm here was Monte Carlo tree search.

Loss

We use 7 unroll steps and a gamma discount of 0.99 in every loss algorithm. For the temporal difference losses, we use a bootstrapping step of 1. The results can be seen in figure 6.6. The best loss was the Monte Carlo loss.

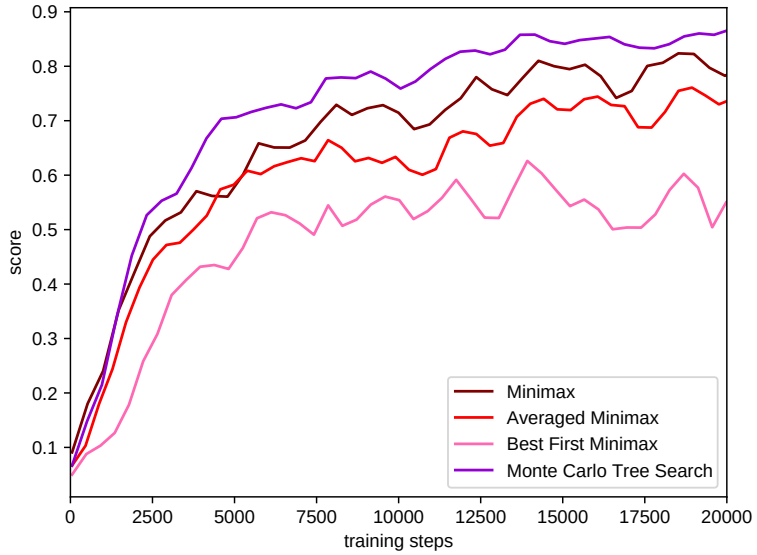

Figure 6.5: Comparison between different planning algorithms in minigrid

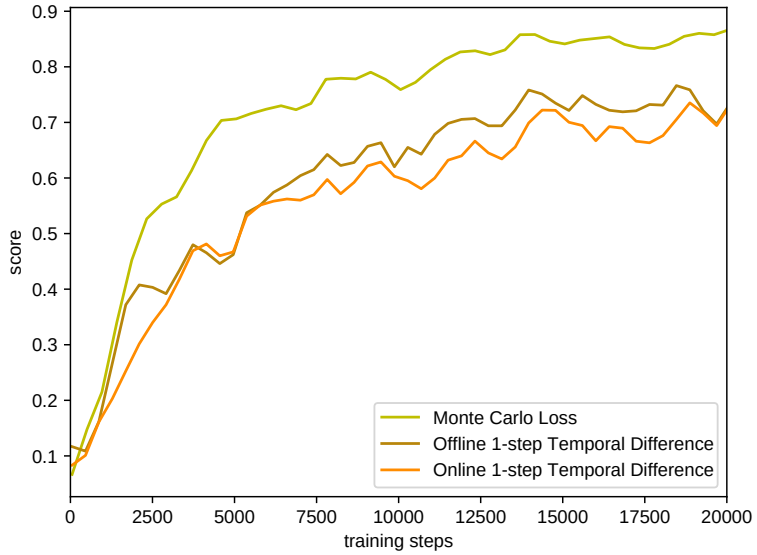

Figure 6.6: Comparison between different loss algorithms in minigrid

Policy

For the policies, we use 5% exploration for the ϵ-greedy strategies and a temperature of 1 for the visit distribution policy. The best agent found was Monte Carlo tree search with Monte Carlo loss and ϵ-value greedy for all measurements seen in table 6.2.

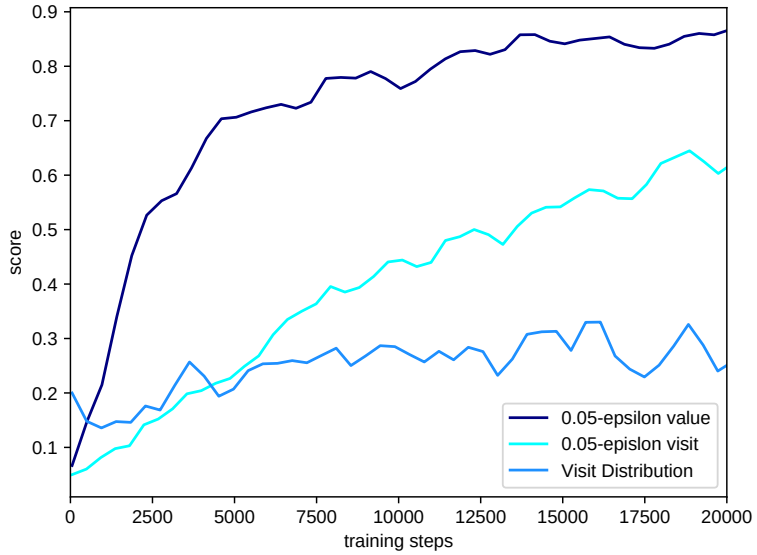

Figure 6.7: Comparison between different policy algorithms in minigrid

Results for Minigrid						
Search	Loss	Policy	Max	Min	Mean	Median
Avg. Minimax	MC loss	ϵ-value greedy	0.90	0.60	0.76	0.75
BFMMS	MC loss	ϵ-value greedy	0.63	0.41	0.54	0.56
MINIMAX	MC loss	ϵ-value greedy	0.91	0.71	0.79	0.78
MCTS	MC loss	ϵ-value greedy	**0.99**	**0.76**	**0.87**	**0.87**
MCTS	MC loss	ϵ-visit greedy	0.87	0.00	0.62	0.76
MCTS	MC loss	Visit distribution	0.33	0.22	0.26	0.24
MCTS	OFF TD	ϵ-value greedy	0.91	0.63	0.73	0.73
MCTS	ON TD	ϵ-value greedy	0.86	0.52	0.73	0.75

Table 6.2: Summary of the final results of the agents tested for minigrid

6.3 Tictactoe

For tictactoe, we learn using self-play. To test our agents, we play 100 games against an expert agent (level 1 in our tictactoe implementation) after every 500 self-play games (or 5000 training steps), and report the mean score. When playing against the expert we remove exploration from the policy. Our Disjoint MLP has one layer of 100 neurons in every operation, except in the operation that returns the next encoded state, which has 200. The hidden state has a size of 32. We use a prioritized replay buffer with a maximum capacity of $25 \cdot 10^3$ transitions. Tictactoe has illegal actions, we use a masked penalty of -1 which is a lower bound of the reward function. This environment is considerably more challenging than the previous, especially because we train it using self-play, so we run it for 10^4 episodes (10^5 training steps). Each experiment is repeated 5 times. The summary of this experiment can be seen in table 6.3.

Planning

Tictactoe has a branching factor of 9. In this experiment, we expand different amounts of nodes depending on whether it is a breadth first or best first algorithm. In the first, we explore the tree until a depth of 2, which corresponds to 81 nodes. However, tictactoe, unlike the previous environments, has illegal actions. Best first algorithms can easily avoid illegal actions. The breadth first ones could also avoid these if the model was given. To compensate, we use less expansions in the best first algorithms based on a simple heuristic: for trajectories that would end in a draw, there are 9 boards. Each one has one less legal move than the previous, so the minimax trees of depth 2 in each board would evaluate $9 * 8$, $8 * 7$ ... leaf nodes. The average is then given by $\sum_{i=1}^{9}(i(i-1)/9 = 26.6$. We will use a *rounder* number and expand 25 nodes in each best first algorithm. The results can be seen in figure 6.8. The best algorithm was the best first minimax search.

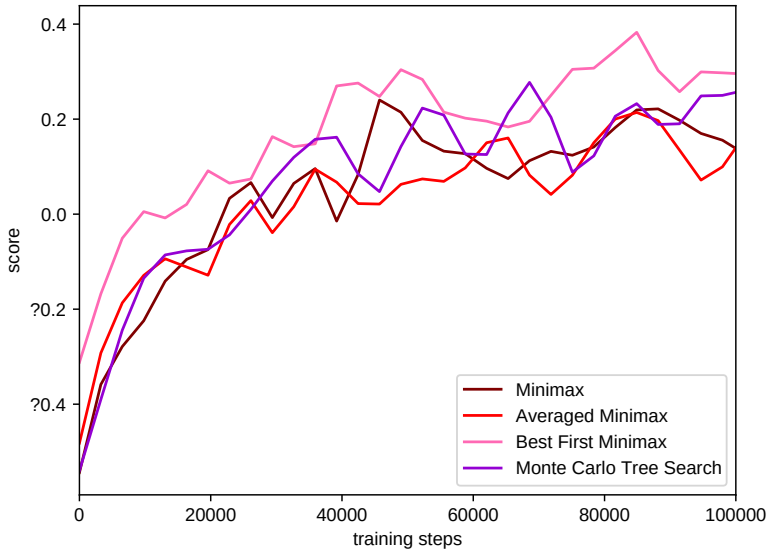

Figure 6.8: Comparison between different planning algorithms in tictactoe

Loss

We use 5 unroll steps and a gamma discount of 0.99. For the temporal difference losses, we use a bootstrapping step of 1. The results can be seen in figure 6.9. The offline temporal difference achieved a better mean score in the end.

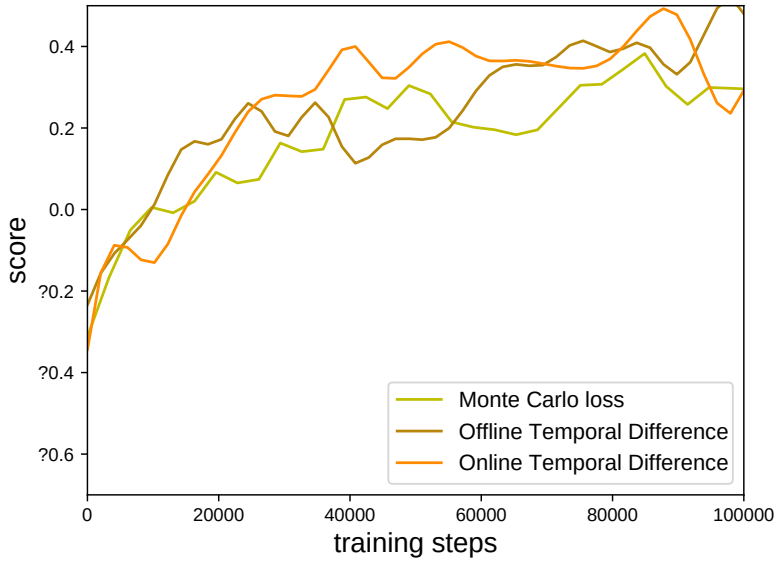

Figure 6.9: Comparison between different loss algorithms in tictactoe

Results for Tictactoe						
Search	Loss	Policy	Max	Min	Mean	Median
Avg. Minimax	MC loss	ϵ-value greedy	0.28	0.01	0.14	0.12
BFMMS	MC loss	ϵ-value greedy	0.48	-0.02	0.29	0.34
BFMMS	OFF TD	ϵ-value greedy	**0.54**	0.39	**0.48**	**0.49**
BFMMS	OFF TD	ϵ-visit greedy	0.41	0.14	0.30	0.31
BFMMS	OFF TD	Visit distribution	0.53	**0.45**	**0.48**	0.48
BFMMS	ON TD	ϵ-value greedy	0.38	0.23	0.29	0.30
Minimax	MC loss	ϵ-value greedy	0.21	0.12	0.14	0.13
MCTS	MC loss	ϵ-value greedy	0.35	0.02	0.25	0.28

Table 6.3: Summary of the final results of the agents tested for tictactoe

Policy

For the policies, we use 5% exploration for the ϵ-greedy strategies and a temperature of 1 for the visit distribution policy, when training and choose always the best action when testing against the expert. The results can be seen in figure 6.10. There is a tie between two agents, which use best first minimax search and offline temporal difference, but differ in policy. One uses the ϵ-value greedy and the other uses the visit distribution. Both have pretty much the same median and mean (table 6.3), but the agent that uses the visit distribution is consistently better throughout the experiment as seen in fig. 6.10, so we will consider this one to be the best agent.

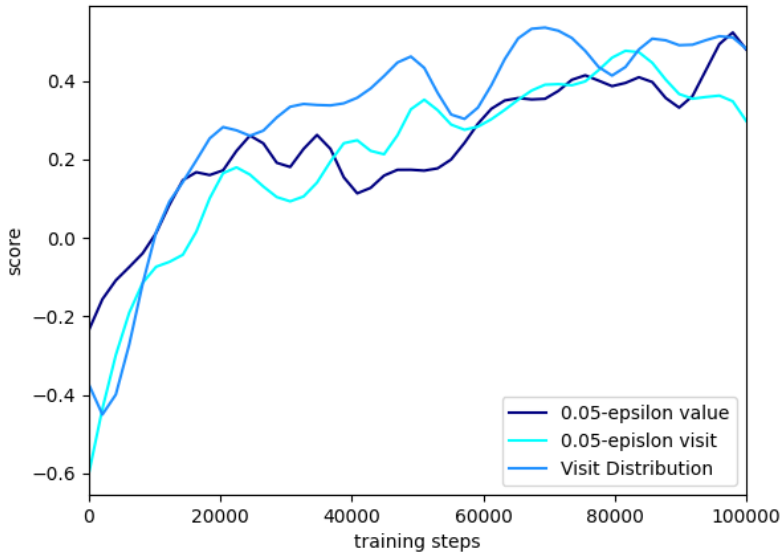

Figure 6.10: Comparison between different policy algorithms in tictactoe

6.4 Discussion

Every problem is different and the results captured for these do not automatically guarantee the same for any other. However, these have distinctive characteristics of their own, making them good *testbeds* to see if our implementation works. Cartpole has a strong reward signal, but is very long; as opposed to minigrid, which has a sparse reward signal but short episodes. These last two do not have illegal actions, which make them suitable to see if our implementation works in *common* environments. Tictactoe, on the other hand, can be seen as the final challenge, since it requires a good use of the mask learned and due to the self-play used during training, a technique known to be challenging in RL. In all of these, the best agents found achieved very good results, reaching almost the maximum score in cartpole and minigrid. For tictactoe, the expert is good at defending against obvious plays and it is not good at attacking, so it is expected that a very good agent will win frequently when it starts and tie when the opponent does, leading to a score near 0.5. Our best agent gets very close to this value. The idea of predicting a mask directly and the algorithm adaptations described in this thesis seem to work well.

The most surprising result is the best planning algorithm in cartpole being the averaged minimax. This algorithm was created by us in this work based on the very *naive* idea of averaging over all successors in each node. Out of all four algorithms, it is the only one that does not theoretically converge to the minimax function. Still, it was better than all the others in cartpole; achieved a better result than BFMMS in minigrid and had the same result as minimax in tictactoe.

Regarding our other idea, the online temporal difference, it was not particularly better than the other

two, but it was still fairly consistent, achieving better results than Monte Carlo loss in some experiments.

The most important observation, however, is that, for each component, different algorithms were better according to the environment. For example, the BFMMS, that was the worst in cartpole and minigrid, was the best in tictactoe. Similarly, the visit distribution policy that was bad in the initial two, was very good in the third environment. This motivates the existence of works like ours that facilitate the variation of different agent combinations. It also suggests the possibility of improving state of the art MBRL agents, like Muzero, in certain environments, by modifying their components and finding more suitable algorithm combinations.

7

Conclusion

Contents

7.1 Conclusions

In this thesis we presented a modular architecture that identifies separate relevant components for MBRL agents that use planning algorithms. Then we provided our own specific implementation of this architecture, capable of searching in domains with a variable set of actions in each state, by learning an action mask directly. We implemented several search algorithms, adapting them to the mask. One of them, called averaged minimax, was proposed in this work and yielded surprisingly good results. Different loss methods, policies and environments were also implemented. Lastly, we experimented this implementation, demonstrating that it works and achieves good results for the environments implemented. The experimentation done seemed to indicate that the best algorithm combination in an agent is problem dependent, motivating the need for tools that facilitate the implementation and variation of the different parts of an agent, like the ones introduced in this thesis.

7.2 Future Work

This work was motivated by the need for tools that allow extension, so it is only natural that we have some suggestions about how to extend our work.

The most obvious suggestion is to implement AlphaZero and MuZero and study if, by varying some of their parts, it is possible to bring the hardware requirements down. If the hardware resources so allow it, a comparison, using more challenging environments, between these agents and the one proposed in this thesis might be interesting.

Some other ideas are, for instance, teaching the model to predict the observations exactly as they are given by the environment; using every node in the search tree as data [28]; add loss methods based on the TD(λ) algorithm; implement our base agent for the case of non-deterministic environments (which is also a suggestion described in the paper of Muzero [26]); add search algorithms like A* [34], K-BFS [35], IDA* [36], B* [37], among others.

We do not expect our work to allow every single idea regarding MBRL algorithms, as that would be a very ambitious objective, but to simply be able to accommodate a diverse range of them that are common in research.

Bibliography

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[4] M. Campbell, A. Hoane, and F. hsiung Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002.

[5] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[6] H. Wang, M. Emmerich, M. Preuss, and A. Plaat, "Alternative loss functions in alphazero-like self-play," in *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 155–162, 2019.

[7] Q. Cohen-Solal, "Learning to play two-player perfect-information games without knowledge," *arXiv preprint arXiv:2008.01188*, 2020.

[8] A. Borges and A. Oliveira, "Combining off and on-policy training in model-based reinforcement learning," *arXiv preprint arXiv:2102.12194*, 2021.

[9] J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, T. Pfaff, T. Weber, L. Buesing, and P. W. Battaglia, "Combining q-learning and search with amortized value estimates," *arXiv preprint arXiv:1912.02807*, 2019.

[10] J. A. de Vries, K. S. Voskuil, T. M. Moerland, and A. Plaat, "Visualizing muzero models," *arXiv preprint arXiv:2102.12924*, 2021.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT press, 2018.

[12] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[13] G. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," Tech. Rep. CUED/F-INFENG/TR 166, Cambridge University, 1994.

[14] L. V. Allis *et al.*, *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

[15] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European Conference on Machine Learning*, pp. 282–293, Springer, 2006.

[16] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[17] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of Learning and Motivation*, vol. 24, pp. 109–165, Elsevier, 1989.

[18] R. Ratcliff, "Connectionist models of recognition memory: constraints imposed by learning and forgetting functions.," *Psychological Review*, vol. 97, no. 2, p. 285, 1990.

[19] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[20] S. Zhang and R. S. Sutton, "A deeper look at experience replay," *arXiv preprint arXiv:1712.01275*, 2017.

[21] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," *arXiv preprint arXiv:1707.01495*, 2017.

[22] R. E. Korf and D. M. Chickering, "Best-first minimax search," *Artificial Intelligence*, vol. 84, no. 1-2, pp. 299–337, 1996.

[23] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[24] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, 2011.

[25] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[26] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[27] J. Oh, S. Singh, and H. Lee, "Value prediction network," *arXiv preprint arXiv:1707.03497*, 2017.

[28] J. Veness, D. Silver, A. Blair, and W. Uther, "Bootstrapping from game tree search," in *Advances in Neural Information Processing Systems* (Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, eds.), vol. 22, Curran Associates, Inc., 2009.

[29] S. Russell and P. Norvig, "Artificial intelligence: A modern approach," 2002.

[30] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[31] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[32] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *IEEE Transactions on Systems, Man, and Cybernetics*, no. 5, pp. 834–846, 1983.

[33] D. Michie, "Game-playing and game-learning automata," in *Advances in programming and non-numerical computation*, pp. 183–200, Elsevier, 1966.

[34] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[35] A. Felner, S. Kraus, and R. E. Korf, "Kbfs: K-best-first search," *Annals of Mathematics and Artificial Intelligence*, vol. 39, no. 1, pp. 19–39, 2003.

[36] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[37] H. Berliner, "The B* tree search algorithm: A best-first proof procedure," in *Readings in Artificial Intelligence*, pp. 79–87, Elsevier, 1981.

A

Diagrams

This appendix section shows the class diagrams associated with the particular implementation proposed.

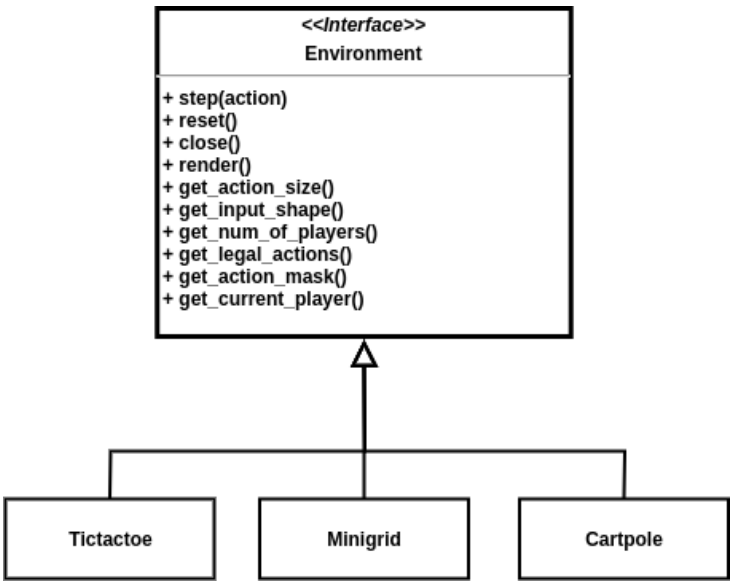

Figure A.1: Class diagram of the implemented environment module

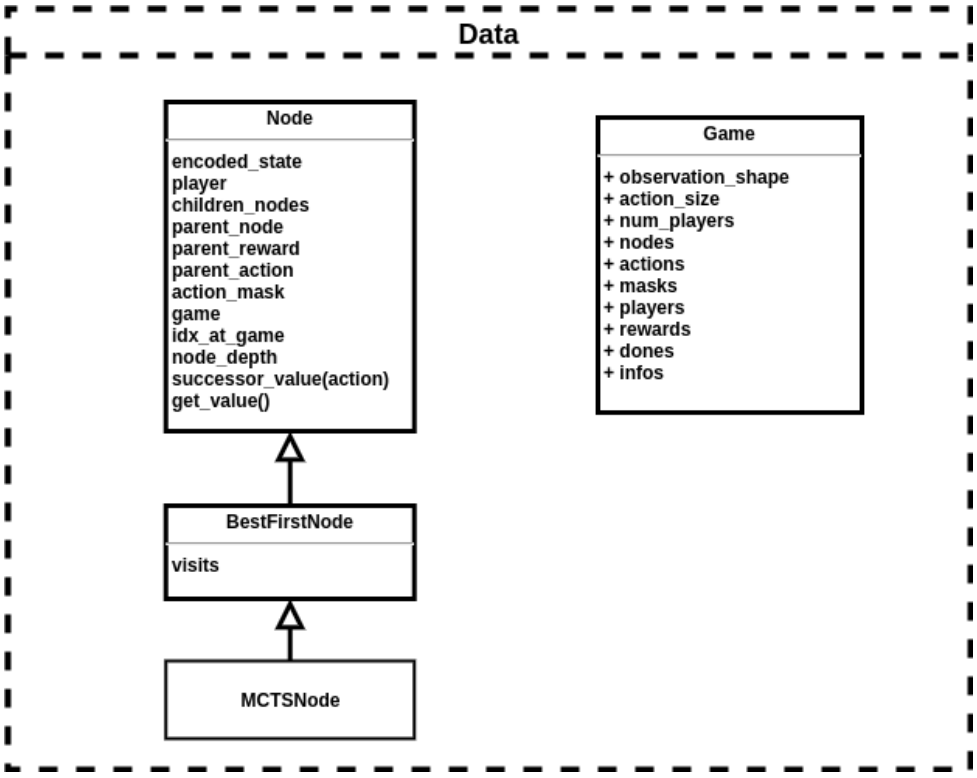

Figure A.2: A simplified class diagram of the implemented Data module. In practice, the Node's interface is slightly longer, contained other auxiliary methods, and the attributes are only accessible through methods.

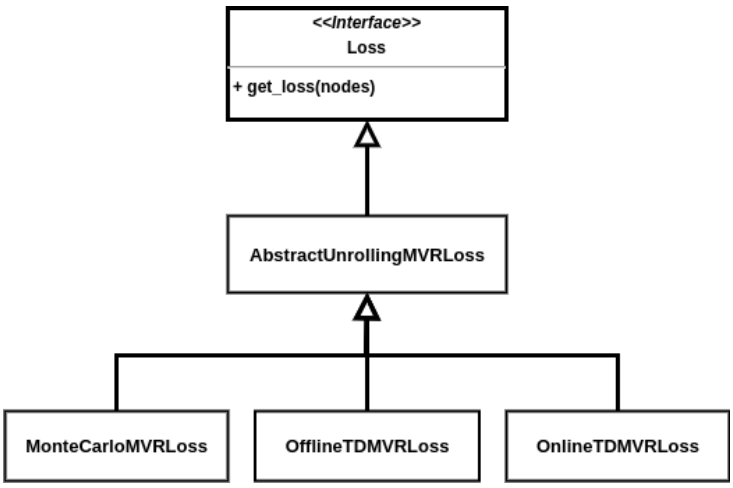

Figure A.3: Class diagram of the implemented loss module

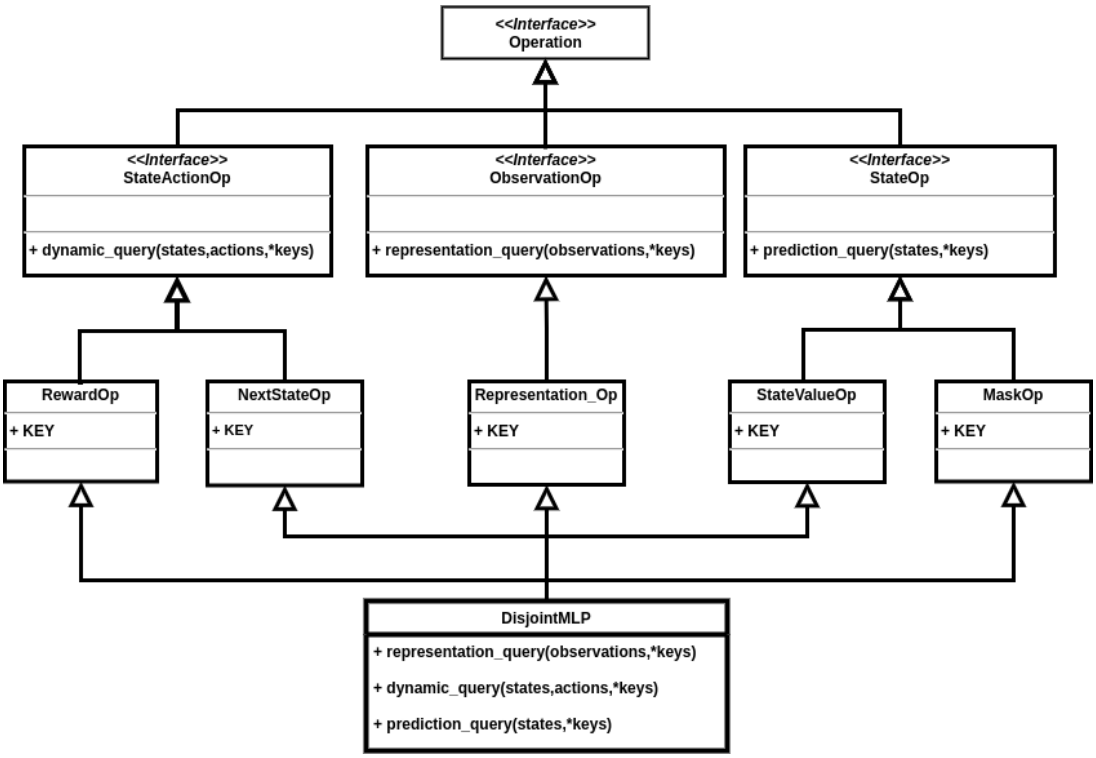

Figure A.4: Class diagram of the implemented model module

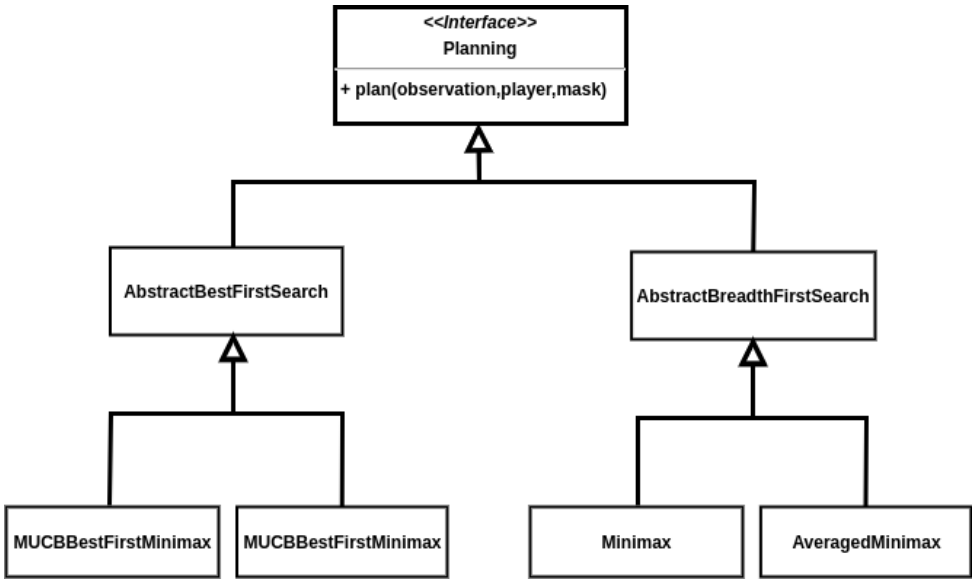

Figure A.5: Class diagram of the implemented planning module

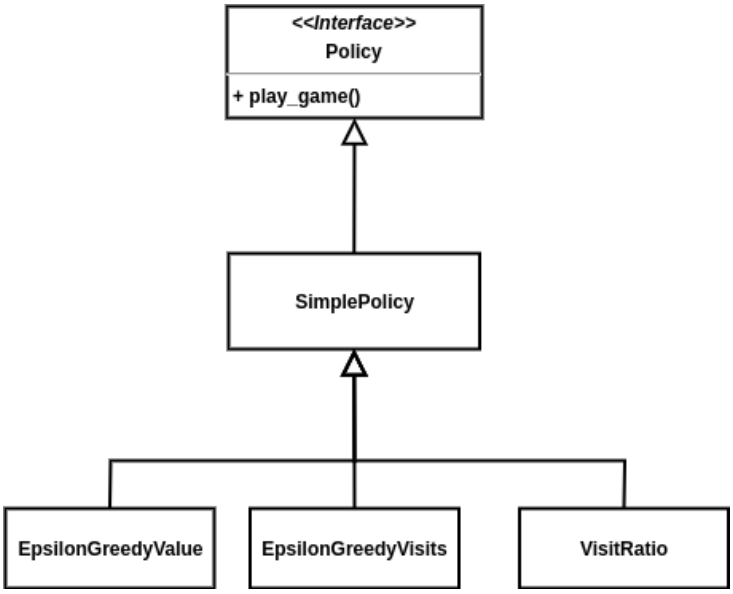

Figure A.6: Class diagram of the implemented policy module