A Modular Architecture for Model-Based Deep Reinforcement Learning

Tiago Oliveira
tiagojgroliveira@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2021

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. Introduction

In 2016, a program called AlphaGo [15] beat a Go world champion for the first time. Go was seen as the new milestone of artificial intelligence since the former world chess champion, Gary Kasparov, lost to DeepBlue in 1997 [5]; and it was finally mastered using reinforcement learning (RL), neural networks and a search algorithm. Following this achievement, an even more powerful program called AlphaGo Zero [16] was created, improving exclusively from playing against itself, indicating the strong potential of deep reinforcement learning.

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.

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[2]). It is only normal that, after their success, the research on these algorithms has been increasing, trying to make them more practical to use [3, 6, 8, 10, 19].

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

[1] Our implementation can be found in `https://github.com/GaspTO/Modular\_MBRL`
[2] https://cloud.google.com/tpu/docs/tpus

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. 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 facilitate 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

2.1. Reinforcement Learning

In reinforcement learning, an agent interacts with an environment, that is usually 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}, returning an associated reward r_t. 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 (next state and reward) after an action has been taken depends solely on the current state and not on the prior history of the current episode. In an 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 transitions. 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 where states are not completely visible to the agent. These environments are formulated as a partially observable Markov decision processes (POMDPs). 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.

2.2. Muzero

Muzero is a model based reinforcement learning (MBRL) agent. It uses a Monte Carlo tree search (MCTS) [7] to augment the capacity of its state evaluation by looking ahead in the state-space. In each iteration, the search will descend the tree looking for a leaf node to expand, using a best-first formula that mediates between exploitation and exploration. When expanding, it estimates a value for the leaf node's state, which influences and estimates a policy distribution for the successors. The value estimated is added to the total value of each node in the path descended and the average of these values makes up the improved state value of that node, $V(s)$. The search repeats for a certain number of iterations and, in the end, when it is time to choose an action to use in the *real* environment, the successors of the root node define a policy function, π, using their visit distribution:

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

where $N(s,a)$ is the number of visits associated with the transition (s,a) and τ is a *temperature parameter*, influencing the exploration/exploitation trade-off. The lower it is, the greedier the policy becomes.

Muzero learns the environment by interacting with it, where the states learned do not have specific semantics. The environment is learned through a representation function, h_θ, that converts a real observation to a hidden state and a dynamics function, g_θ, that receives a hidden state and action and returns the next state and reward associated. The policy and value are estimated by a prediction function, f_θ, which receives a hidden state as argument. An observation from a played episode is sampled and is unrolled by applying the representation function, converting it to an hidden state, followed by applying the next k actions used in the episode to the successive hidden states, using the dynamics function. Then, the prediction function is used in each of these hidden states, returning the policy and value estimations. To learn these functions, the rewards obtained are updated to match the real observed ones. The policy is updated based on the visit distribution given by the root node of the respective observation and the value function is improved based on a Monte Carlo value or by bootstrapping from an improved state value.

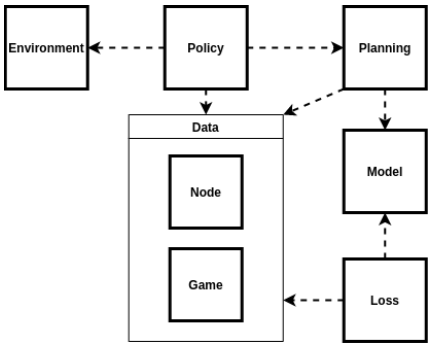

Figure 1: Basic architecture overview. The arrows show dependencies between modules. For instance, the policy module needs to know how to interact with the environment module.

3. Architecture

The architecture we propose has the objective of allowing changes and extensions in each key component, without being too complex to use or understand. It is designed to support MBRL agents. The reasoning behind it is to identify key parts in these agents 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. We propose an architecture with six different parts, which are called modules: 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 its behavior, while the Data module concerns itself with the storage of data to be used between them. The purpose of each 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 one of them without having to refactor anything else. A simple scheme summarizing the structure of the architecture is given in figure 1.

3.1. Environment

A RL agent interacts with an environment through actions. After every action, there is some information given to the agent so that it can decide its next ones, in order to solve the problem. The agent depends on the environment, but not vice-versa.

3.2. Module

This module models not only the environment, but also all the other functions required by the Planning module. How they are calculated concerns only the model. Different search algorithms might need new functions, so it is important that this component's implementation is easily extendable. The architecture should support both agents that learn the environment's model and those that receive a perfect one. If this module is queried for a next state, it returns the true next state or a simple estimation, depending on whether it knows the environment's model.

3.3. Data

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

3.4. Planning

This module equips the agents with stronger estimations by allowing it to look ahead into the possible future states. By having a specific module for this part, the exact search algorithm used should be able to vary. We consider state-space planning algorithms [17]. In these, actions mediate transitions between two states and a notion of value is attached to each state. A graph or tree structure, composed by nodes, results from these algorithms. They explore the hypothetical state space in order to return a node structure with enough information for the agent to make better decisions during the *real* episode.

3.5. Policy

The policy is responsible to iterate through the whole environment episode, balancing between exploration and exploitation. It makes use of a planning algorithm to come up with an action sequence to interact

with the environment. It is also responsible for storing the whole episode information using the Game submodule, including the planning data.

3.6. Loss

After interacting with the environment, the model should be updated to be more accurate and insightful. This is done by using the experience stored using the Data module to calculate a loss value. In general, the loss can be calculated solely using the trajectory stored and using the data acquired during planning.

3.7. Summary

The agent interacts with an environment through a policy, which uses a search algorithm and saves the relevant information received from the environment and planning in a game instance. The stored data is passed to a loss algorithm that calculates a loss value in order to update the model. Our architecture provides these blocks, that should be assembled by the user to create the agent.

4. Implementation

In this section, we describe a possible RL implementation using the proposed architecture.

4.1. Environment

The environment is modeled as a POMDP and the interface is similar to the openAI Gym environment [4]. In each step of an episode, the agent executes an action, observing the next state and reward. Unlike the Gym environments, ours allows multiple agents. Traditional problems where only one agent acts upon are conceptualized as a single agent problem. Another difference is the existence of illegal actions. The Gym environments have a fixed set of them in every state. The way they handle the actions chosen that do not make sense is by remaining in the same state and, often, returning a negative reward to disincentivize these actions. The problem starts when we have more than one player. In tictatoe, for example, players play in 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. The specific environments implemented are tictactoe, cartpole [1] and minigrid [**?**].

Cartpole. In cartpole, the agent has to balance a pole, which has one extremity fixated on a moving cart, by applying forces to the left and right of it. The environment has these two actions and only one agent. The episode ends when the pole is more than 12 degrees from its original position or the cart is further than 2.4 units from the center. Optionally, we can limit the maximum number of steps. The observation is an array with four values: cart position, velocity, pole angle, and angular velocity.

Minigrid. This environment is a NxN grid. The agent starts in one of the squares and the objective is to reach a square goal. It has a direction and a position. The goal square is fixed, but we can optionally decide where the agent starts or choose for it to start in a random square every time the episode restarts. This environment is partially observable, being able to observe a grid of 7x7 in each step. The observations are given in 7x7x3 arrays. The rewards are 1 if the goal is reached and 0 otherwise. There are three actions: turn left, right and move forward. 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 number of steps.

Tictactoe. It is a game played by two agents in a 3x3 grid. 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 an array 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 or we want to do *self-play*, in which case the agents choose the actions for both players. The expert agent has three levels. The level 0 plays randomly. The next will play like the previous 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 mode, the reward is always 0, except for a play that leads to victory, which is 1. In single-agent mode it receives 1 if the action led to a victory, -1 to a defeat and 0 otherwise.

4.2. Base Agent

Just like in the environment module, we implement alternative algorithms in the agent's modules, which will allow us to compose different agents. However, there is a base idea behind all of these. To simplify, we call this base idea by *base agent*.

Our base agent learns the model of the environment as it interacts with it and, similarly to MuZero, the hidden states learned do not have a specific semantics. It will learn how to estimate the value of each state and the reward associated with each transition. The idea introduced, however, is to learn to predict

the legal actions directly, so that it avoids exploring and conducting updates based on illegal parts of the tree during planning. MuZero avoids doing this using a policy function that sets the exploration of those nodes to 0, but this is unsuitable for algorithms that are non-best first or for those that are but estimate the value of the successors of the leaf expanded, instead of the leaf itself. By predicting the legal actions directly, our agent is much more compatible with different planning strategies.

4.3. Model

This module should be easy to extend in order to accommodate any new functions needed by new search algorithms. The model should be used in the most efficient way possible since deep learning computations can be very time consuming. The efficiency, however, is dependent on the internal structure (the neural architectures used) and this needs to be easy to change. For instance, AlphaZero shares the same initial layers to calculate both the policy and value functions, which should not be calculated twice, but reused, in case we need both of them. To handle this, the models have a *query* method per type of input. For example, functions that receive a state share the same query method between them, but not with the functions that receive a state and an action as arguments. Each function has an attribute key associated with it that is passed to the respective query method when it needs to be calculated. In this way, if we want to calculate multiple functions (with the same input type), we make only one call to a query method, pass all the keys accordingly and the model decides internally how to calculate each one of the requested functions, reusing the necessary calculations for the highest efficiency. In practice, this is implemented using a hierarchical class structure that allows the Planning and Loss module to easily verify if the model passed to them supports every needed functions.

There are five relevant 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 value represents whether the agent estimates the action to be legal or illegal, respectively giving 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.

We implement one specific model (*disjoint multilayer perceptron*) that supports the previously described five functions. Each one of them, internally, is computed by an independent multi-layer perceptron.

4.4. Data

Game submodule. A game stores the information given by the environment during the episode: observations, actions, rewards, legal actions and players, and the data (nodes) returned by the planning algorithm.

Node submodule. The specific instance of this submodule is called *node* and its purpose is to be able to store properties and values necessary to the policy and planning algorithms. Different types of search strategies might need different types of nodes. We implement three. The first one stores: a hidden state s; an improved state value $V(s)$; each successor's hidden state, $S(s, a)$, and their validity value $M(s, a)$, according to an action mask; the state's player, $Pl(s)$; and the reward per transition, $R(s, a)$. The second node implemented is an extension of the first. It is supposed to be used by best first algorithms and it stores, additionally, the number of visits in the tree, $N(s)$. Finally, the third node is an extension of the second one 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 improved state value to be the average of them:

$$V(s) = \frac{T(s)}{N(s)}. \tag{2}$$

4.5. Planning

A search algorithm will receive the current observation, player and action mask. 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 objective of these algorithms is to find more accurate estimations of a node's state value by using the values of its successors. 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 are predicted for every state in the tree. We define the *action-value* as

$$Q(s^k, a) = \begin{cases} R(s^k, a) + V(s^{k+1}) & \text{if } Pl(s^k) = Pl(s^{k+1}) \\ R(s^k, a) - V(s^{k+1}) & \text{otherwise} \end{cases}, \tag{3}$$

where $s^{k+1} = S(s^k, a)$. When relevant, we use a superscript k to denote that the state s^k was created looking ahead k steps. We can think of it as depth of the state in a hypothetical trajectory. The action-value can be seen as a 1-step bootstrapped value. We will also define the notion of masked action-value,

$$Q_M(s, a) = M(s, a) \cdot Q(s, a) + \Big(1 - M(s, a)\Big) \cdot c_{\text{penalty}} \tag{4}$$

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, such as 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 in the same way. 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. If the environment has two players, then its children nodes will be instantiated with the opposite player. After this first part, the algorithms start differing.

Breadth First. There are two breadth first algorithms: *minimax* and a variation of our own that we call *averaged minimax*. 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 the respective transition reward are predicted. If the successor is at maximum depth, their state value is also predicted. After the whole tree has been generated, the values of the leaf states are propagated backward until the root. These two algorithms differ in the backup rule. For some state s, the update rule in the averaged minimax is given by

$$V(s) \leftarrow \frac{\sum_{a \in \mathcal{A}} M(s, a) \cdot Q(s, a)}{\sum_{a \in \mathcal{A}} M(s, a)}. \tag{5}$$

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

$$V(s) \leftarrow Q(s, \arg\max_a Q_M(s, a)). \tag{6}$$

Best First. The other two implemented algorithms are of type best first: *Monte Carlo tree search* (MCTS) and *best first minimax* (BFMMS), both using a variation of the UCB formula (used in the UCT algorithm [12]) to take into account the mask, resulting in what we call masked upper confidence bound (MUCB). In each step of a best first iteration, the next action to be selected in a state, s, is given by

$$\arg\max_a \left(Q_M(s, a) + c_{mucb} \cdot \sqrt{\frac{\log(N(s))}{N(S(s, a)) + 1}} \cdot M(s, a) \right), \tag{7}$$

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

In an iteration, when expanding a leaf node, the states of its successors and their state values are estimated. The rewards and mask associated with these transitions are also predicted. The difference between MCTS and BFMMS is also in the backpropagation part. In BFMMS, the update rule is the same as in minimax (equation 6). In MCTS, the backpropagation will add, to each node, a path value, G, estimated after expanding the leaf node. The value added to the leaf after its expansion at depth d is the masked average of its newly created successors

$$G^d = \frac{\sum_{a \in \mathcal{A}} M(s^d, a) \cdot Q(s^d, a)}{\sum_{a \in \mathcal{A}} M(s^d, a)}. \tag{8}$$

For every node in shallower depths after that, the added quantity is recursively given by

$$G^k = \begin{cases} R(s^k, a^k) + G^{k+1} & \text{if } Pl(s^k) = Pl(s^{k+1}) \\ R(s^k, a^k) - G^{k+1} & \text{otherwise} \end{cases}, \tag{9}$$

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

$$V(s^k) \leftarrow \frac{\sum_{i=0}^{N(s^k)} G_i^k}{N(s^k)}. \tag{10}$$

4.6. Policy

In each step, the policy will use the planning algorithm to choose the next action. It will iterate with the environment until the episode is over, saving the planning and environment's data using the Game submodule. We implemented three 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 corresponding to 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 choice on the number of visits. The second policy is ϵ-visit greedy, which does the same as the first policy, but, instead of the action-value, it chooses based on the visits of each successor node. The third policy, π, is given by a visit count distribution just like in MuZero (equation 1).

4.7. Loss

A loss class will receive a set of nodes and calculate a loss value as tensor that can be used to update the model. Each node has access to its game, which means it also has access to all the actions, observations, nodes... of its episode. We implemented three loss algorithms. All of them use the same *unrolling* process as MuZero and Value Prediction Networks to update the reward and mask functions: 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 three losses differ in what target value to use to update the model.

- **Monte Carlo loss**. In this loss algorithm the target value, for a specific state, is the (discounted) sum of rewards from that state until the end of the trajectory.
- **Offline Temporal Difference**. This is the one used in Muzero, where the target value is bootstrapped from the state n steps away from the current ones, which is stored in the respective root node.
- **Online Temporal Difference**. If the agent uses a replay buffer and reuses the same nodes for a long time, the value stored in them can easily become outdated. This loss is similar to the previous, but, instead of using the improved state value, $V(s_{t+n})$, it applies the representation function to the observation n steps after the current one, $s_{t+n} = h_\theta(o_{t+n})$, and uses the value function to get a new value to bootstrap from, $v_\theta(s_{t+n})$.

If the unrolled states surpass the trajectory's size, they are modeled as absorbing states and the target value used with them is 0. Each loss class is not under the obligation of working for every type of node. In fact, if we want to extend our implementation to use a policy function, 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 loss functions might use specific information created during planning, so they can not work using nodes that result from algorithms that do not use and create the necessary information to update the model.

5. Experimental Evaluation

In this section, we assess our implementation by creating different agents for the three environments implemented. For each environment, we start by comparing planning algorithms, then use the best one to compare different losses and, finally, compare policies. In the end, we should have a good agent for each environment. For each experiment, the agent plays a full episode, stores it and then conducts 10 updates, each with a batch size of 128 nodes. The results are shown as a function of training steps (the number of updates). In each experiment, the criterion to select the best algorithm is to choose the one with the highest average score in the last 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.

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 dist.	494.0	139.6	303.0	303.3

Table 1: Summary of the final results of the agents tested for cartpole.

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 dist.	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 2: Summary of the final results of the agents tested for minigrid.

5.1. Cartpole

For cartpole, we limit each episode to a maximum of 500 transitions and report, for each training step, the results as the mean of the last 100 episodes. The Disjoint MLP has one layer, each with 80 neurons, for every function. The encoded state used has size 12. We use a uniform replay buffer with a maximum capacity of 10^5 transitions. The masked penalty is 0 as default. We repeated every experiment 10 times, each for 10^4 training steps. The final results for every experiment for cartpole can be seen in table 1 and figure 4.

- **Planning**. The best first algorithms use 16 iterations and the breadth first ones explore a maximum depth of 4. The best algorithm was the averaged minimax, which is used in the next experiments.
- **Loss**. The loss algorithms use 10 unroll steps and a gamma discount of 0.95. For the temporal difference losses, the bootstrapping step is 10. The best loss was the offline temporal difference.
- **Policy**. There is only one policy that is compatible with breadth first algorithms. To compare different policies, we picked MCTS, which was the best best-first search algorithm in the first experiment, and ran the loss experiment to figure the best one, which was the offline temporal difference. For the policy experiment, using the MCTS, we use 5% exploration for the ϵ-greedy strategies and a temperature of 1 for the visit distribution policy. The final best agent was the averaged minimax with offline temporal difference and ϵ-value greedy.

5.2. Minigrid

We use a 6x6 minigrid environment. Each episode starts at a random position and is limited to a maximum of 15 transitions. The Disjoint MLP has one layer of 100 neurons per function and the hidden state has size 15. We use a prioritized replay buffer with a maximum capacity of 25000 transitions. The masked penalty used is 0. We repeated every experiment 10 times, each for 20000 training steps. The results are given, per training step, as the mean score of the last 100 episodes. The final results for every experiment for minigrid can be seen in table 2 and figure 3.

- **Planning**. The number of iterations used by the best first algorithms is 27 and the breadth first ones use a depth of 3. MCTS was the best planning algorithm.
- **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 best loss was the Monte Carlo loss.
- **Policy**. We use 5% exploration for the ϵ-greedy strategies and a temperature of 1 for the visit distribution policy. The best agent found was MCTS with Monte Carlo loss and ϵ-value greedy.

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 dist.	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 3: Summary of the final results of the agents tested for tictactoe

5.3. Tictactoe

The agents are trained in this environment by doing self-play. Every 5000 training steps, we play 100 games against an expert agent (level 1 in our implementation) and report the mean score. When playing against the expert we remove exploration from the policy. The Disjoint MLP has one layer of 100 neurons in every function, except in the next state function, which has 200. The hidden state has a size of 32. We use a prioritized replay buffer with a maximum capacity of 25000 tsitions. The masked penalty is -1. Each experiment is run for 10^5 training steps and repeated 5 times. The final results for every experiment for tictactoe can be seen in table 3 and figure 2.

- **Planning**. The number of iterations used by the best first algorithms is 25 and the breadth first ones use a depth of 2. The best algorithm was the BFMMS.
- **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 offline temporal difference was the best loss.
- **Policy**. The policies 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. There is a tie between two agents, which use BFMMS and offline temporal difference but differ in policy. One uses the ϵ-value greedy and the other the visit distribution. Both have pretty much the same final median and mean results.

5.4. 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 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 work seem to function 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 the new loss, the online temporal difference, was not particularly better than the other two, but it was still fairly consistent, achieving better results than Monte Carlo loss in cartpole and the same in minigrid.

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 got poor results in the initial two, achieved the maximum mean score 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.

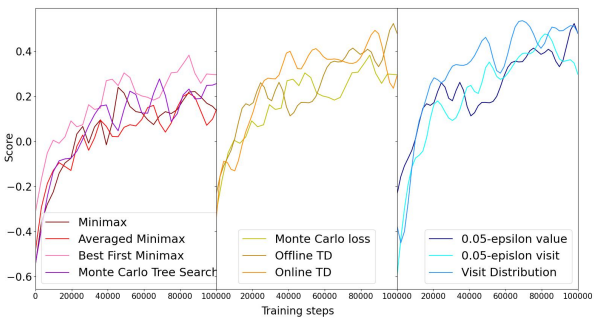

Figure 2: Results for tictactoe. From left to right: the experiences with Planning, Loss and Policy

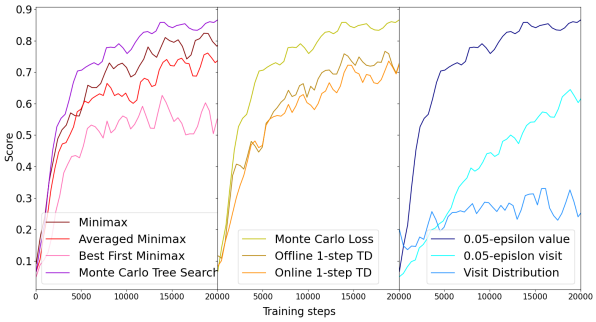

Figure 3: Results for minigrid. From left to right: the experiences with Planning, Loss and Policy

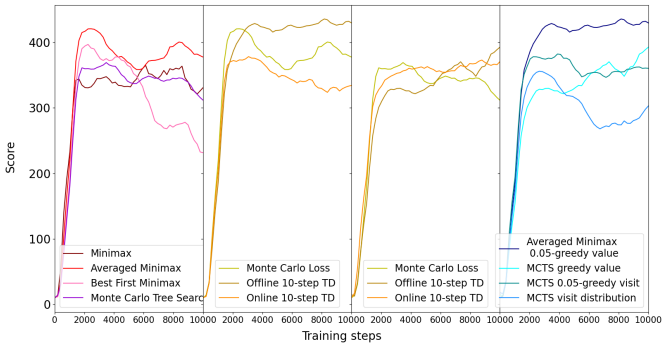

Figure 4: Results for cartpole. From left to right: the experiences with Planning, Loss for averaged Minimax, Loss for MCTS and Policy.

6. Conclusions and Future Work

In this work, 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 yield surprisingly good results. Different loss methods, policies and environments were also implemented. Lastly, we experimented this implementation by creating different agents and demonstrated 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 work.

6.1. 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 it. 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 allow it, a comparison, using more challenging environments, between these agents and our implementation might be interesting.

Some other ideas are, for instance, teaching the model to predict the observations perfectly; using every node in the search tree as data [18]; adding loss methods based on the TD(λ) algorithm [17]; implementing our base agent for the case of non-deterministic environments (which is also a suggestion described in the paper of Muzero [14]); adding search algorithms like A*[11], K-BFS [9], IDA* [13], B* [2], among other.

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 ideas that are common in research.

References

[1] 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*, (5):834–846, 1983.

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

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

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

[5] M. Campbell, A. Hoane, and F. hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

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

[7] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, pages 72–83. Springer, 2006.

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

[9] A. Felner, S. Kraus, and R. E. Korf. Kbfs: K-best-first search. *Annals of Mathematics and Artificial Intelligence*, 39(1):19–39, 2003.

[10] 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.

[11] 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*, 4(2):100–107, 1968.

[12] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European Conference on Machine Learning*, pages 282–293. Springer, 2006.

[13] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[14] 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*, 588(7839):604–609, 2020.

[15] 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*, 529(7587):484–489, 2016.

[16] 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*, 550(7676):354–359, 2017.

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

[18] J. Veness, D. Silver, A. Blair, and W. Uther. Bootstrapping from game tree search. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[19] 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)*, pages 155–162, 2019.