UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

OTÁVIO FLORES JACOBI

O-MuZero: Abstract Planning Models Induced by Options on the MuZero Algorithm

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Anderson Rocha Tavares
Coadvisor: Prof. Dr. Bruno Castro Da Silva

Porto Alegre
December 2021

“A learning experience is one of those things that says,

’You know that thing you just did? Don’t do that.”’

— Douglas Adams

ACKNOWLEDGMENTS

First, I would like to thank the professors Anderson Rocha Tavares and Bruno Castro da Silva for the support, feedback and discussions that resulted in this work.

I would like to thank my family, my mom, professor Luciane Flores Jacobi for the inspiration and help, my dad Elton Rogerio Teixeira Jacobi and sister Natália Flores Jacobi for the unconditional support and understanding of my absence for the last years since I left home to study at UFRGS. I love you all.

I would like to thank my partner Nathália Carpenedo Ferrari for always staying positive and understanding the challenges during the weekends I was developing this work. All my time on the university I had the help of many friends and I would not have gotten this far without you all, but specially, I want to thanks Arthur Medeiros and Felipe Leivas for the laughs together along the way. Thank you all for always staying with me.

ABSTRACT

Training Reinforcement Learning agents that learn both the value function and the environment model can be a very time consuming method, one of the main reasons for that is that these agents learn by actions one step at the time (primitive actions), while humans learn in a more abstract way. In this work we introduce O-MuZero: a method for guiding a Monte-Carlo Tree Search through the use of *options* (temporally-extended actions). Most related work use options to guide the planning but only acts with primitive actions. Our method, on the other hand, proposes to plan and act with the options used for planning. In order to achieve such result, we modify the Monte-Carlo Tree Search structure, where each node of the tree still represents a state but each edge is an option transition. We expect that our method allows the agent to see further into the state space and therefore, have a better quality planning. We show that our method can be combined with state-of-the-art on-line planning algorithms that uses a learned model. We evaluate different variations of our technique on previously established grid-world benchmarks and compare to the MuZero algorithm baseline, which is an algorithm that plans under a learned model and traditionally does not use options. Our method not only helps the agent to learn faster but also yields better results during on-line execution with limited time budgets. We empirically show that our method also improves model robustness, which means the ability of the model to play on environments slightly different from the one it trained.

Keywords: Model-Based Reinforcement Learning. Options. Monte-Carlo Tree Search. On-line planning.

RESUMO

Agentes de aprendizado por reforço que aprendem tanto a função de valor quanto o modelo do ambiente são métodos que podem consumir muito tempo, uma das principais razões para isso é que esses agentes aprendem através de ações com passo de cada vez (ações primitivas), enquanto os humanos aprendem de uma forma mais abstrata. Neste trabalho introduzimos O-MuZero: um método para guiar a busca de árvore Monte-Carlo através do uso de *options*. A maioria dos trabalhos relacionados utiliza *options* para guiar o planejamento, mas só joga com ações primitivas, nosso método, por outro lado, se propõe a planejar e jogar com as *options* usadas no planejamento. Para alcançar esse resultado, modificamos a estrutura da Árvore de Busca de Monte-Carlo para que cada nodo da árvore ainda represente um estado, mas cada aresta é uma transação de uma *option*. Esperamos que nosso método permita que o agente veja mais além no espaço do estado e, portanto, faça um planejamento de melhor qualidade. Mostramos que nosso método pode ser combinado com algoritmos de planejamento on-line que jogam com um modelo aprendido. Avaliamos diferentes variações de nossa técnica em *benchmarks* previamente estabelecidos do ambiente e comparamos com a técnica de base. Nosso método não só ajuda o agente a aprender mais rapidamente, mas também produz melhores resultados durante o jogo. Empiricamente mostramos que o uso de nosso método também melhora a resiliência do modelo, o que significa a capacidade do modelo de jogar em ambientes ligeiramente diferentes daquele em que foi treinado.

Palavras-chave: Aprendizado por Reforço *model-based*, *Options*, Busca de Monte-Carlo em Árvores, Planejamento *on-line* .

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Using Reinforcement Learning (RL) techniques that combine planning and learning have achieved remarkable success in Artificial Intelligence. These techniques often learn a model of the Markov Decision Process (MDP) representing the environment. This learning process uses a planning technique to improve the agent policy allowing the learning process to happen without previous knowledge of the environment. However, such techniques often require millions of interactions with the environment to achieve significant results, falling far behind from the level of a human player, that can obtain similar results with much less interactions.

One reason why RL techniques may require so many steps is that they have to interact on the environment on an action level, while humans do it in a more abstract and hierarchical fashion. For example, when playing a video game, a human player is able to quickly break down the game into abstract sub-goals (e.g. get a key to open a door). The behavior of achieving a sub-goal can be simulated in RL by using a temporal abstraction framework called *options* (SUTTON; PRECUP; SINGH, 1999). Options can be seen as a sequence of actions or other options (defined by the option policy) to achieve a certain sub-goal. Options define a certain hierarchy into the RL algorithm (PATERIA et al., 2021) and can help these algorithms to plan their actions taking into account possible sub-goals.

Problems that require some sort of planning with sub-goals arise in many occasions. Video game playing has seen some success with these techniques, but they can be generalized for many others. For example, for a robot, the torque to apply on each joint in order to move its legs in a way to do a step, then, abstracting a sequence of steps into how to walk straight, and further, learn to use these steps and plan to reach a given goal. Each of these problems can make use of the lower-level options and then build upon them for abstracting further into different hierarchies.

Another reason why RL planning techniques sometimes require many interactions with the environment is due the fact that estimating the value of a given state can be hard if the total return of an episode is unbounded and has high variance. For example, a Monte-Carlo Tree Search (MCTS) algorithm using a random *rollout* at the simulation step, thus, will require many interactions to have a realistic value estimate of a given state. In addition, a model of the environment might not always be available.

When learning an environment model or a policy, options can be used to better explore which parts of the environment are interesting: instead of searching the entire

space (which can be infinitely large) when guided by abstract options, the agent will gather more information about the search space around the option trajectory, instead of the whole search space. In (GREGOR; REZENDE; WIERSTRA, 2016) the authors do exactly this: they provide the agent with intrinsic exploration bonuses based on modeling options. However, the opposite can also occur: an option can also deviate the agent curiosity from its main goal to a sub-goal that can be less relevant or non-optimal.

Most of the state-of-the-art techniques for playing games focus either on planning with options (WAARD; ROIJERS; BAKKES, 2016) or planning with a learned model of the environment (SCHRITTWIESER et al., 2020). Techniques that play with a learned model often have a very long training time and the first iterations of on-line planning usually are very random and usually result in an environment time-out (this is not required, but the agent can possibly play randomly indefinitely) which greatly increases training time and costs.

In this work, we want to explore options as a way for reducing training time and reducing the amount of timed out simulations in algorithms that play with a learned model. We propose O-MuZero, a planning algorithm that can use options and learns a model of the environment. Our hypothesis is that options will guide the agent through learning about more distant (and still relevant) parts of the environment more quickly, facilitating learning and planning on environments with high variance returns. The main idea of this algorithm is to extend the MuZero algorithm with options, allowing the agent to directly see more into the future at each time step.

In this work, O-MuZero was tested in different versions of one of the 2D pathfinding benchmarks proposed by (STURTEVANT, 2012). We propose a simplification of the MuZero algorithm presented by (SCHRITTWIESER et al., 2020) to use a tabular method instead of Neural Networks (NN), given the challenges of finding appropriate hardware for training these approximators. We propose metrics to evaluate the difference between the original MuZero and our O-MuZero and finally present comparisons between them. Preliminary results showed that using options can help decrease training time, decrease the amount of timed-out simulations and even increase the algorithm robustness. The scope in which our algorithm was tested is limited and more testing is still needed to assert that our results transfer properly for harder domains.

This work presents the basic ideas of how we could extend on-line planning algorithms that play with learned model to use options. However, some points remain open, regarding the use of function approximators such as Neural Networks and the automated

learning of options. We discuss these open questions, with possible approaches and issues.

The chapters in this work are organized as follows: Chapter 2 discusses the relevant theoretical background. Chapter 3 presents the related work in the bibliography. Chapter 4 defines our techniques, similarities and differences from related studies. Chapter 5 presents our experiments setup and results and, finally, in Chapter 6 we present a conclusion and possible future work.

2 BACKGROUND

This chapter explains basic concepts prior to understanding this work. Section 2.1 discusses core concepts of Reinforcement Learning (RL). Next, Section 2.2 introduces the framework for temporal abstraction (options). Finally, Section 2.3 presents how we can use RL for planning, specially, using MCTS algorithm.

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a field of Artificial Intelligence dedicated to study on how to learn to map situations to actions, guided by a reward signal (SUTTON; BARTO, 2018). Differently from supervised or unsupervised learning, where an algorithm tries to learn something using either labeled or unlabeled data, RL algorithms learn by interacting with an environment while only receiving a numerical reward signal. Recently, RL has achieved outstanding results in different fields such as: video game playing (SCHRITTWIESER et al., 2020), code generation (ELLIS et al., 2020) and natural language processing (KENESHLOO et al., 2019).

Usually, RL problems are formulated as Markov Decision Processes (MDP). More formally, a MDP $\mathcal{M}$ is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is a function defining the probability of transitioning to state $s' \in \mathcal{S}$ after choosing action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$, also denoted as $p(s'|s, a)$ such that $\sum_{s' \in \mathcal{S}} p(s'|s, a) = 1, \forall s \in \mathcal{S}, a \in \mathcal{A}$. In addition, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the function that returns the expected reward for executing action a in state s. Finally, $\gamma \in [0, 1]$ is the discount factor that indicates how important future rewards are.

In order to behave in an environment defined by the MDP, we define a policy function $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ specifying the probability of selecting action a on state s, given that $\sum_{a \in \mathcal{A}} \pi(a|s) = 1, \forall s \in \mathcal{S}$. We can define the return at discrete time t as the total cumulative and discounted reward the agent can receive from that point onward. More formally, let R_t be the random variable representing the reward at time step t, then the return can be represented by the random variable G_t according to Equation 2.1. The goal of an RL algorithm is to learn an optimal policy π^* that maximizes G_t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1} \tag{2.1}$$

When following a policy π we denote the *state-value* of a state under this policy as $v_\pi(s)$. The value represents how much return the agent expects to obtain in a given state s following π, thus we define $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$. Similarly, we can extend the the idea of a state-value function to the *action-value* function, which represents the expected return starting from state s, first taking action a, and following π afterwards. We denote the action-value function as $q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$.

We can roughly break down RL algorithms into two classes: model-free and model-based algorithms. Model-free methods try to learn a policy directly from interacting with the environment without any prior knowledge of the underlying MDP and do not learn nor use an model of the environment at any point. Such class of techniques has achieved many impressive results, specially when combined with Convolutional Neural Networks (CNNs) (MNIH et al., 2013). Model-based algorithms on the other hand, have a *model* of the MDP (either learned or given) and given a state and an action, they can predict the next state and reward. In RL, using an environment model to produce or improve a policy is called *planning*. Improving the policy based only on the reward signal received from interaction with the actual environment, that is called *learning*.

2.1.1 Tabular Methods

Tabular Methods consists of using RL algorithms in their simplest form: when state and action spaces are small enough for storing their values combinations in a table. These table values can be updated based on temporal-difference (TD) learning (SUTTON, 1988). TD learning methods predict the state-value/action-value given samples of the total return using a table V which is used to estimate v and Q to estimate q, as in Equation 2.2 and Equation 2.3 where α is the *learning rate*.

$$V(S_t) = (1 - \alpha)V(S_t) + \alpha G_t \tag{2.2}$$

$$Q(S_t, A_t) = (1 - \alpha)Q(S_t, A_t) + \alpha G_t \tag{2.3}$$

If instead of using the full return G_t we use *n* intermediate steps and then the current estimate of the final state, we say that this algorithm *bootstraps*. *Bootstraping* means that the estimate of an state-value/action-value depends on another state-value/action-value which might have not yet been updated as shown in 2.4. Equation 2.5 shows the

special case where we have exactly one intermediate step, also called TD(0). Equation 2.6 derivation follows the same process as Equation 2.4, notice that for the particular case of Equation 2.6 we can have different ways of estimating $v(S_{t+1})$ and different methods emerge from it.

$$V(S_t) = (1-\alpha)V(S_t) + \alpha[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \gamma^{n-1} R_n + \gamma^n V(S_n)] \quad (2.4)$$

$$V(S_t) = (1-\alpha)V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1})] \quad (2.5)$$

$$Q(S_t, A_t) = (1-\alpha)Q(S_t, A_t) + \alpha[R_{t+1} + \gamma v(S_{t+1})] \quad (2.6)$$

Two famous tabular methods are Q-learning (WATKINS, 1989) and Sarsa (RUMMERY; NIRANJAN, 1994). Both of theses methods are based on TD learning, where an agent updates the policy every time step after interacting with the environment. RL algorithms can also be classified as on-policy or off-policy methods. On-policy methods learn a policy while following it (Sarsa is an on-policy method). Off-policy methods learns about one policy while following a different policy (Q-learning is an example of an off-policy method).

Q-learning, for example, searches for the policy that maximizes the action-value such that for a given policy π we have $q^* = \max q_\pi(s, a)$. Having a table Q of size $|\mathcal{S}| \times |\mathcal{A}|$ we can use it to estimate the true value of q^*. Using the Q-table and Equation 2.6 we have Q-learning update as in Equation 2.7. It is proven that with the possibility to visit all states and actions with enough interactions with the environment, the Q-table estimates will converge to q^*.

$$Q(S_t, A_t) = (1-\alpha)Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a)] \quad (2.7)$$

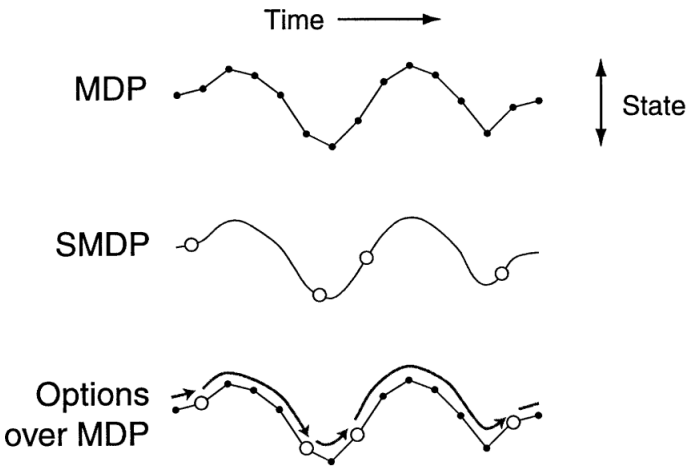
2.2 Options - Temporal Abstraction in RL

Learning to behave in an environment, might be difficult for an agent with only primitive actions. Humans, for example, may learn in a much more hierarchical way. We learn one behavior that can be re-used in different situations. This is the basic idea

of temporal abstraction: instead of having one action at one time step we can have one *option*, which is a sequence of actions (or other options) in the environment during many time steps. Adding options to an RL agent can have several advantages but also several drawbacks. (SUTTON; PRECUP; SINGH, 1999) introduces the theoretical foundation for the options framework and show what needs to be done for them to be interchangeable with low-level actions.

The first formalism needed to understand how options are added to an RL algorithm is the one of a Semi-Markov Decision Process (SMDP). When using options, the MDP concept is extended to deal with temporal abstraction. In a MDP, the time step between each action is constant, however, in SMDPs this is not necessary. In an SMDP, the option execution is transparent for the agent but with options the agent can observe the intermediary states and actions. Options can be defined over any of these two formalisms as shown in Figure 2.1

Figure 2.1: Difference between MDPs, SMDPs and options



Source: (SUTTON; PRECUP; SINGH, 1999)

Formally, an option o can be defined as a tuple $\langle \pi_o, \beta_o, \mathcal{I}_o \rangle$ where the option policy $\pi_o : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ is the probability of taking an action a in the state s given that we are executing the option o. The termination condition $\beta_o : \mathcal{S} \rightarrow [0, 1]$ is the probability that the option o will finish at a given state s and, the initiation set $\mathcal{I}_o \subseteq \mathcal{S}$ represents all the states that an option can start executing its policy.

The intuition behind options lies in the fact that we can reuse a behavior. For example, if an agent task is to find an object in a large map with multiple rooms, an option could be the sequence of actions it has to take to go to the next room. This option has a mapped higher level behavior that can be expressed through a sequence of low-level

actions. If an option o takes the agent from room A to room B, the initiation set would be all the positions in room A and the termination function would be 1 for all states in room B and 0 for all other states.

Options can be defined over MDPs having the guarantees of an SMDP. The SMDP allow us to use the same planning and learning algorithms that use only primitive actions as long as the *history* of the executed option is available. For example, in an SMDP, if an option o starts executing at time n and finishes its execution at time $n+k$, a decision taken by any algorithm at time step τ, $n \leq \tau < n+k$, may depend on the entire option history $\mathcal{H}^o_\tau = s_n, a_n, r_{n+1}, s_{n+1}, a_{n+1}, r_{n+2} \ldots r_\tau, s_\tau$ whereas in an MDP the algorithm only has access to $s_{\tau-1}, a_{\tau-1}, r_\tau, s_\tau$. It is important to notice that the option does not have access to the history before time n neither after time $n+k$.

Having defined that an option has access to its execution history $\mathcal{H}^o_\tau$ we can modify other algorithms to work directly with options instead of actions. We can define the option reward r_o as the discounted return yielded by the option o as $r_o = r_{n+1} + \gamma r_{n+2} + \gamma^2 r_{n+3} + \cdots + \gamma^{k-1} r_{n+k}$. Finally, we can change, for example, the state-value function to operate directly on rewards from options as shown in Equation 2.8 below. In this equation, we consider that a generic option o started at time n and finished at time $n+k$.

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} \ldots \gamma^{n-1} r_o + \gamma^{n+k-1} v_\pi(S_{n+k}) | S_t = s] \tag{2.8}$$

2.3 Model-Based RL

Model-based RL methods have a model of the environment that can be used to predict how the environment will respond for a given action (SUTTON; BARTO, 2018). This model can be anything that achieves this goal. It can be a exact replication of the environment dynamics or a learned model. This is called the *forward model* and its goal is to mimic the MDP transition function $\mathcal{P}$ and reward function $\mathcal{R}$. The goal of using a *forward model* is that we can reduce the number of real interactions with the environment when this is restricted, but also, we can do $planning$ with this forward model in order to have an improved version of the current policy.

2.3.1 Planning

In RL, planning methods refers to any algorithm that given a model of the environment can create or improve a policy to act in that environment. In this work, we are interested in state-space planning, which means searching the state space for an optimal policy. The main idea in state-space planning is that we need to compute the value of each state using simulated experience as an intermediate step for planning. After calculating the value of a state, we need to update (or *backup*) this value and finally improve the policy.

According to (SUTTON; BARTO, 2018):

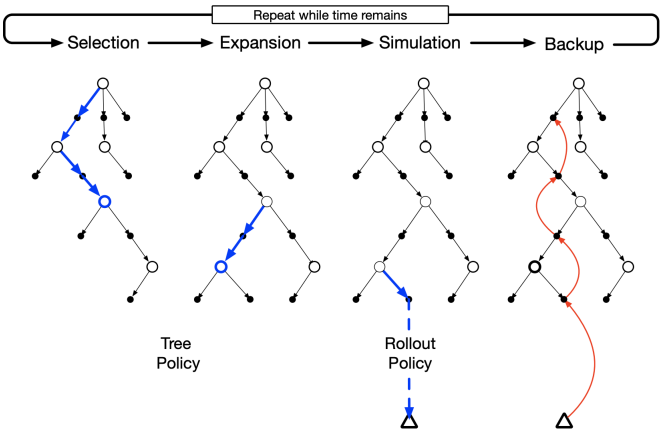
> *"The heart of both learning and planning methods is the estimation of value functions by backing-up update operations. The difference is that whereas planning uses simulated experience generated by a model, learning methods use real experience generated by the environment".*

2.3.2 Monte-Carlo Tree Search

Among various planning algorithms in RL, Monte-Carlo Tree Searches (MCTS) has had a lot of success in recent years (BROWNE et al., 2012). From general video-game playing (WAARD; ROIJERS; BAKKES, 2016) to general board-game playing (SILVER et al., 2017), algorithms using MCTS have surpassed human capabilities.

MCTS is an example of an algorithm usually used for decision-time planning and is a *rollout* algorithm that in its original form does not bootstrap. The original algorithm is explained next and can be followed with Figure 2.2.

Figure 2.2: How MCTS is executed for planning



Source: (SUTTON; BARTO, 2018, pg. 186)

The algorithm has 4 different steps that are repeated for a given time always in this order: selection, expansion, simulation and backup.

Selection: At the start of the algorithm, one of the current tree leafs is selected to be expanded. If the tree is not yet initialized, it selects the root node. After that, the algorithm descend into the tree usually selecting the next node using some kind of upper-confidence bound (UCB) algorithm (AUER, 2002) applied to trees (UCT) (KOCSIS; SZEPESVÁRI, 2006). The basic idea is expressed in Equation 2.9 where the count and value for each edge is updated in the backup phase. Each edge represents the transition of the parent node state s with a given action a for a next state s'. $Q(s,a)$ is the value of an edge and $N(s,a)$ the number of times that this edge has been visited during backup. The goal of UCT is to balance exploiting more interesting nodes while still exploring. The exploration rate is controlled by the hyper-parameter c.

$$UCT(s,a) = Q(s,a) + c\sqrt{\frac{\ln\left(\sum_{a' \in \mathcal{A}} N(s,a')\right)}{N(s,a)}} \tag{2.9}$$

Expansion: After a new leaf node is selected, we expand this node by creating all its children from the possible actions in that state. These are the new leaf nodes that will be possibly selected in the selection phase in the next MCTS iteration.

Simulation: In this stage, we run a *rollout* for the node. There are many strategies for doing so, we can run a random rollout (just play random actions until the end of the episode) or have a heuristic for choosing the actions, which may introduce some bias into the simulations but can help the agent in cases where return may be too sparse. The total discounted return G_t is the result of the simulation. It is important to notice that this stage can be very time consuming (random rollouts in some environments might never finish).

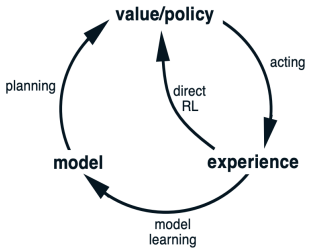
Backup: Finally, the G_t value calculated at the simulation stage is backed-up through the tree. Each parent edge will have its $Q(s,a)$ value updated based on its state estimated return value G_t (as defined in Equation 2.1), and will have its $N(s,a)$ count also incremented by 1 until the root node is reached.

After executing as many iterations of the algorithm as possible, we can select the "best" action based on the count of the edges (or also based in $Q(s,a)$) of the root node or have a policy π of acting based on these counts.

2.3.3 Integrating Planning and Learning

One of the first ideas for integrating planning and learning comes from Dyna (SUTTON, 1991). The main idea of Dyna architecture is that we can do planning online at every interaction with the environment while also learning a model of the environment. Figure 2.3 shows a diagram with this concept.

Figure 2.3: Dyna Architecture



Source: (SUTTON; BARTO, 2018, pg. 162)

At every interaction with the environment, we first learn a part of the MDP model (model learning) and use this learned model to do planning. By running a planning algorithm in the learned model, we can use this simulated experience (no real simulations with the environment) to improve the value/policy. We also improve the value and policy with data from real experience.

3 RELATED WORK

In this chapter, we discuss methods related to O-MuZero. First, we present the main ideas of MuZero algorithm and specially how it performs its MCTS search (Section 3.1). Then, we present previous works that have already combined MCTS and options and were inspiration for our approach (Section 3.2).

3.1 MuZero

The MuZero algorithm (SCHRITTWIESER et al., 2020) is the current state-of-the-art technique for general video-game playing and game board playing. It has surpassed humans and many other algorithms when playing Atari games and board games such as Go, Shogi and Chess.

MuZero algorithm is based on many techniques previously presented. First, MuZero learns to play the game without prior knowledge of a model of the environment. Like the Dyna algorithm, MuZero learns a model of the environment and then uses this model for planning. The output from the planning algorithm (MCTS) is then used to improve the policy and value function in a online manner.

One key aspect of the MuZero MCTS is that it does not rely of running the (possible very long) simulation stage of the MCTS algorithm. Instead, the algorithm uses a function approximator (more specifically, a Neural Network) f that predicts the policy $p : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ and the value $v : \mathcal{S} \rightarrow \mathbb{R}$ of a given state. More formally, the f function is conditioned on the neural network weights θ such that $p, v = f_\theta(s_t)$ where p is the current predicted policy and v is the current predicted value for state s_t. The MuZero algorithm also uses a NN g_θ to simulate the model of the environment such that $s_{t+1}, r_{t+1} = g_\theta(s_t, a_t)$. More details of how MuZero algorithm creates its own internal representation of the state using yet another NN h_θ which the interested reader can get more details at (SCHRITTWIESER et al., 2020). The MuZero MCTS adaptation uses the network f_θ output p to guide its exploration and instead of having to use a simulation at the end of each MCTS iteration it uses the estimated v value from the neural network. This greatly increases performance and the v value converges to the correct value of the state by using the real experience outputs called z. Finally, at each discrete environment step, when the MCTS algorithm is executed, we extract the policy π from the tree, based on the number of times that each action was executed from the root node.

At each time step the values s_t, a_t, s_{t+1}, r_{t+1}, π and z are stored in a Prioritized Replay Buffer (SCHAUL et al., 2015) and the NN weights θ of f_θ and g_θ are trained with data from this buffer. The network will predict at each time step more realistic outputs and improve the quality of simulations at each run. This is the main idea of MuZero and more important details about its MCTS adaptation are explained in Section 3.1.1.

3.1.1 MuZero MCTS search details

MuZero MCTS search is slightly different from the standard MCTS algorithm presented in this section in a few key aspects. First and most importantly, there is no real simulation step: instead of running a (possible) long random rollout, we get the value of the node by using the neural network f_θ. However, more adaptations are needed: *(i)* we need a way to balance the default priors (the default p vector from f_θ) with the state value; *(ii)* we need to update each edge of MCTS taking into account intermediate rewards; and *(iii)* we also can not use the same exploration parameter c for all games, given that minimum and maximum total return can be completely different between those and we do not want to give the agent any previous information about the game.

To address the issue *(i)* mentioned above, MuZero algorithm uses a UCT algorithm controlled by two parameters: c_1, which controls the influence of the default prior to the exploration, and $c2$, which is used to control the exploration based on less visited nodes. The update rule using c_1 and c_2 is presented on Equation 3.1. All the notation used here is the same as presented in (SCHRITTWIESER et al., 2020, Appendix B).

$$UCT(s,a) = Q(s,a) + P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \left[c_1 + \log\left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right] \tag{3.1}$$

To fix the issue *(ii)*, the value Q of each state is updated using a n-step return of the cumulative discounted rewards, bootstrapping from the v value where forward pass on f_θ network happened.

Finally, to address *(iii)* UCT normalizes the Q value to $\overline{Q}$ in range $[0,1]$ by using the current maximum and minimum Q values on the tree, as presented in Equation 3.2.

$$\overline{Q}(s^{k-1}, a^k) = \frac{Q(s^{k-1}, a^k) - \min_{s,a \in Tree} Q(s,a)}{\max_{s,a \in Tree} Q(s,a) - \min_{s,a \in Tree} Q(s,a)} \tag{3.2}$$

All the final equations and details briefly explained here can be found with details on Appendix B of (SCHRITTWIESER et al., 2020).

3.2 MCTS with options

Using options with MCTS has been attempted in the literature in different projects.

In (WAARD; ROIJERS; BAKKES, 2016) the authors propose an algorithm called O-MCTS (Option Monte-Carlo Tree Search) that uses options to expand the search tree and select an action to be played at each time step. They further extend this technique to the called OL-MCTS (Option Learning Monte Carlo Tree Search) which is able to learn the best options from a given set of options. Their technique presents promising results for General Video Game Playing. However, their technique can only select the immediate action to be played in a given state instead of the full option that should be followed. They also show that the success of their technique is due to the fact that, by using options, the MCTS branching factor is reduced (and thus, the average depth of the MCTS increases), allowing for a more focused search while expanding the tree.

The technique presented in (PINTO; COUTINHO, 2018) has a step for previously selecting an option following a ϵ-greedy rule and then running the MCTS only with the actions allowed by this option to succeed in a Computer Fighting Game. Using such a technique allows them to have very good results. However, such technique can not be directly applied to the MuZero algorithm given that the option selection and MCTS simulation are already based on a ϵ-greedy policy for the Q-learning algorithm.

In both (PEREZ et al., 2013a) and (PEREZ-LIEBANA et al., 2017) the authors increment the MCTS algorithm with simple options called macro-actions. A macro-action is an option that just repeats a given action n times. More formally, a macro-action is an option where the initiation set $\mathcal{I}_o$ contains all possible states of the environment, the termination condition β_o is independent of the state and just yields $n-1$ consecutive zeros followed by a final 1 and the policy π_o always return 1 for the option action and 0 for all the other actions. In both publications, they show that adding macro-actions to the planning algorithm can greatly increase how far the agent can see (and therefore, *plan*) in the search tree, allowing for a better planning. One could argue that macro-actions are the simplest form of a multi-step option, but, as shown by (PEREZ et al., 2013b) they can be very effective, specially in tasks that do not require a very fine-grained search.

Finally, (VIEN; TOUSSAINT, 2015) discusses how to further extend an Option

guided MCTS in Partially Observed Markov Decision Processes (POMDPs). By using the algorithm POMCP (Partially Observable Monte-Carlo Planning) proposed in (SILVER; VENESS, 2010), they modify each node in the MCTS to also store a belief b about the possible next states, thus, allowing the MCTS to operate in stochastic environments. Combining this belief information they create options based on sub-goals (which they call a *sub-task*) and then they derive two new algorithms: the H-UCT and H-POMCP: These algorithms generalize the search tree to operate with many step sub-tasks (options) and also incorporate the possibility of a stochastic environment . This work is then further incremented in (BAI; SRIVASTAVA; RUSSELL, 2016) where not only actions are abstracted through the use of options, but states are also represented in a abstracted (compacted) way, increasing the capability of the agent of doing online planning. Similar to (WAARD; ROIJERS; BAKKES, 2016) this technique only select the immediate action to be played and not the full option after using it to plan.

3.3 Summary

The methods discussed in this section provide different perspectives to MCTS guided by options. The work developed with macro-actions is particularly interesting given the fact that even the simplest forms of options can already greatly increase the planning capability. Table 3.1 presents a comparison between all previous works and ours. The three comparison points were: *(i)* the need for a forward (environment) model, *(ii)* if the agent can plan and act with selected options and *(iii)* if the option can be directly selected by an approximator (no ϵ-greedy policy on top, for example).

Table 3.1: Previous approaches and ours

	Requires prior forward model	**Can plan and act with options**	**Can use approximator for option selection**
Approach			
(WAARD; ROIJERS; BAKKES, 2016)	✓		✓
(PINTO; COUTINHO, 2018)	✓	✓	
(PEREZ et al., 2013a)	✓		✓
(PEREZ-LIEBANA et al., 2017)	✓		✓
(VIEN; TOUSSAINT, 2015)	✓		✓
Ours	✓	✓	✓

Source: The Author

Besides that, we can see that many techniques have been applied, however, they all share one similarity: even though the planning is done using options, the structure of the search tree remains the same: each node represents one state and each edge of the tree one primitive action, thus, the agent can only know which would be the best action to play in a given state, but not if this action was selected independently or because of an option.

In this work, we slightly change this behavior, given that we want to both plan and play with the selected option, which can allow us to save queries in the forward model and possibly increase the amount of simulations an agent can run in the total planning time budget.

4 O-MUZERO

In this chapter, we introduce how we extend the MCTS algorithm to search and act with options (Section 4.1). Then, we extend the state-of-the-art online planning technique MuZero to use option-guided MCTS and solve arising issues to result in our O-MuZero algorithm (Section 4.2). Finally, we do a full theoretical analysis of many aspects O-MuZero (Section 4.3).

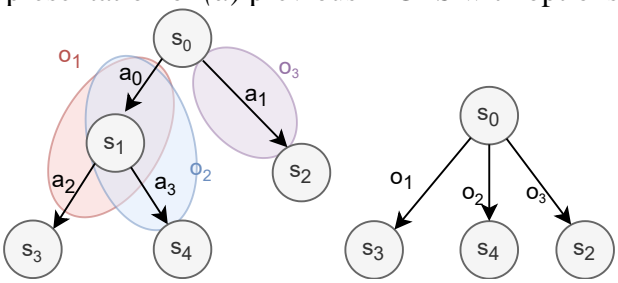
4.1 Option-Guided MCTS

All the works presented in Section 3.2 use options to guide the MCTS execution but only predict the next primitive action to be taken. Suppose that an agent can choose between two options o_1 and o_2 which both share the same common first primitive action a, because the structure of the tree (one node for a state and one edge for a primitive action) in the previous works stays the same. In this case, we can not know if the agent has taken primitive action a because it was following o_1 or o_2.

Because of this limitation, we say that these techniques plan with options but act with actions. In this work, we want to extend the MCTS to be able to plan and act with the chosen options. This change can have several positive and negative aspects that are discussed more in depth in Section 4.3.

In order for the MCTS algorithm to output an option o instead of an action a, we propose a change in the tree representation of MCTS. As usual, each node of the tree represents a state in the environment, however, each edge represents an option o drawn from the possible options in the parents state. Figure 4.1 illustrates this idea.

Figure 4.1: Tree representation of *(a)* previous MCTS with options *(b)* our approach.



Source: The Author

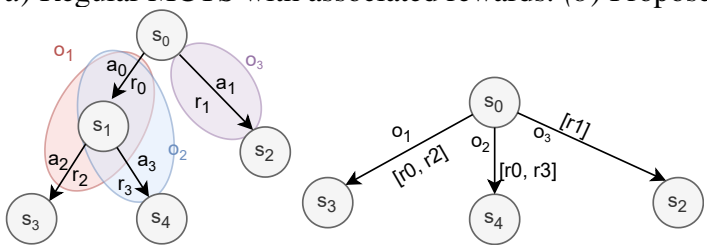
Consider the example in Figure 4.1. In both cases the root node with state s_0 was expanded with three different options, $o_1 = [a_0; a_2], o_2 = [a_0; a_3], o_3 = [a_1]$. If we

follow the algorithm as in Figure 4.1*(a)*, we can only choose the best action from the root state. If, for example, the action chosen as best was a_0, we can not trace back if we chose a_0 because we found that option o_1 or option o_2 was best. However, using our method, each edge in the search tree represents an option and thus, we have the information about which option was the best according to that search. It is interesting to notice that with our technique we allow the MCTS to run without need of storing intermediate nodes (for example, s_1 state visit was suppressed in Figure 4.1*(b)*).

Changing the tree representation has several impacts for the MCTS algorithm, given that each edge is a potentially multi-step option. Specially, how do we generalize for the case of multiple-step return? In the traditional representation, each action (and therefore each edge) was associated with one reward r obtained when executing action a in state s. However, when following a k-step option o, multiple rewards $r_1, r_2, \ldots r_k$ will be associated with this option.

Looking back at the seminal options paper (SUTTON; PRECUP; SINGH, 1999), it is important to see that each option, by its definition, is associated with its history $\mathcal{H}^o_\tau = s_n, a_n, r_{n+1}, s_{n+1}, a_{n+1}, r_{n+2} \ldots r_\tau, s_\tau$. Then, instead of associating each edge with an action a, we can associate it to the full option history $\mathcal{H}^o_\tau$, or in more pratical terms (in order to save memory), we can associate it only with the sequence of rewards $r_1, r_2, \ldots r_k$ from the history. This way, we can calculate the correct return for the full execution. Figure 4.2 shows the same comparison from Figure 4.1 but with the matched reward list.

Figure 4.2: *(a)* Regular MCTS with associated rewards. *(b)* Proposed approach.



Source: The Author

4.2 O-MuZero algorithm

In this section, we propose an extension of the MuZero algorithm that uses our Option-Guided MCTS technique. A direct replacement of MuZero standard MCTS by our Option-Guided MCTS raises important issues and we address them in our O-MuZero algorithm.

4.2.1 Prediction and Dynamics functions

Traditionally, MuZero operates with: *(i)* the prediction function $p, v = f_\theta(s_t)$, where p is the policy distribution over actions and v is a scalar value representing the value of a given state and *(ii)* the dynamics function $s_{t+1}, r_{t+1} = g_\theta(s_t, a_t)$ where s_{t+1} and r_{t+1} are respectively the predicted next state and next reward. Both functions must be adopted to operate with options. The representation function h_θ does not require any adaptation.

The prediction function f_θ does not require any special change in its formulation besides the fact that the predicted policy over actions p needs to be distributed over all possible options instead of actions. For clarity, we denote the policy over all options as p_o. The adapted prediction function is $p_o, v = f_\theta(s_t)$. This also means that the prediction of value v is unchanged.

The dynamics function $s_{t+1}, r_{t+1} = g_\theta(s_t, a_t)$ requires a bit more work to be adapted. First, we should use an option instead of an action as input because we know that this is the option that will be executed and we do not require multiple calls for this function when predicting the next state and reward. However,the function must predict the full list of rewards given by the input option. Therefore, we change the behavior of the function g_θ to predict a list of rewards $\mathbf{R}_{t+1:k} = (r_{t+1}, r_{t+2}, ...r_{t+k})$ and the last state s_{t+k} where k represents how many steps the option takes to execute. The adapted dynamics function is $s_{t+k}, \mathbf{R}_{t+1:t+k} = g_\theta(s_t, o_t)$.

4.2.2 Return value adaptation

Adapting the MuZero functions is a first step towards using options in the MuZero algorithm. The next step is to properly use the values calculated by these functions when backing up the return in MCTS. The traditional return function used in MuZero algorithm works as follows: starting from the root state s_0, when MCTS finds (through the UCT algorithm) a leaf node in state s_l, this node is expanded and its value v^l is calculated using f_θ and next state and reward are calculated using g_θ. All the states from s_0 to s_l and its respective rewards are stored in tables S and R for constant time lookup during backup. Finally, during backup, for each hypothetical planning step $k = l \dots 0$ (from leaf to root)

the return G^k is calculated based on Equation 4.1 bootstrapping from estimated value v^l.

$$G^k = \left(\sum_{\tau=0}^{l-1-k} \gamma^\tau r_{k+1+\tau} \right) + \gamma^{l-k} v^l \tag{4.1}$$

However, this function does not take into account the fact that now we have a list of rewards for each edge (and this list is stored in the R table). To adapt the Equation 4.1, the root node would have a sequence of the reward lists on each edge until the recently added node with value v^l. The first child of the root node in the path would have a full list except the reward list on the edge from the root to it. Let us denote this list of lists of rewards up to leaf node l as $\mathcal{J} = (\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_l)$ where $\mathbf{R}_i$ is a list of rewards for the option executed on that edge. The return value G^k for each edge along $k = l \ldots 0$ is presented in Equation 4.2. We denote as $\mathcal{U}^{k:l}$ as the unrolled list of lists $\mathcal{J}$ starting from $\mathcal{J}_k$ until $\mathcal{J}_l$ (last reward list). $\mathcal{U}_i^{k:l}$ represents the reward element at index i from the unrolled list $\mathcal{U}^{k:l}$. Appendix A has an example of a O-MuZero execution in case the reader wants to better understand the presented equation.

$$G^k = \left(\sum_{\tau=0}^{|\mathcal{U}^{k:l}|} \gamma^\tau \mathcal{U}_\tau^{k:l} \right) + \gamma^{l-k} v^l \tag{4.2}$$

With this adapted equation, we can now plug in this return value into the default MuZero Q-value update rule (SCHRITTWIESER et al., 2020), Equation (4). With this, our technique is almost in place. However, one final issue arises when boostraping with options that can possibly have different duration.

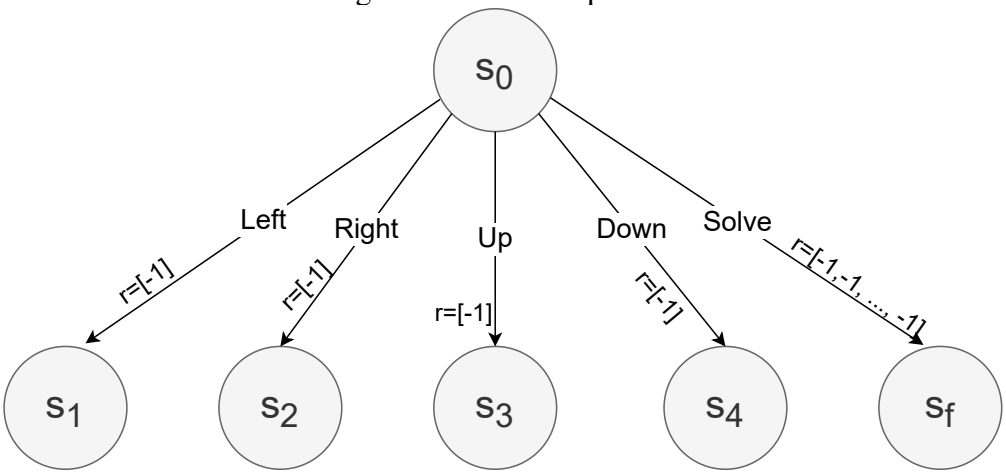
4.2.3 Bootstrap issue

If we update our MCTS Q-values using Equation 4.2 we may give an unfair penalty or advantage towards using or not options rather then primitive actions, creating a unfair bias. In more details: when we are starting to learn the prediction and dynamics functions, we have do not have a realistic estimate of v^l. If we are using a neural network, for example, this value is based on randomly initialized weights of the neural network and is therefore, random. If we use another method (such as tabular), this can be a default value such as 0. In either case, as options can have multiple steps, it is not fair that they bootstrap from the same value v^l.

Lets take for example an environment of a grid world where the agent takes a

reward of -1 at each step and finishes when the agent reaches a certain pre-defined position in this world. In this environment we have 4 primitive options $\{Left, Right, Up, Down\}$ and an option $Solve$ that takes the agent to the final position. When running MCTS with these 5 options, the tree would look like Figure 4.3.

Figure 4.3: Bootstrap issue



Source: The Author

Because the value v^l for each leaf node $\{s_1, s_2, s_3, s_4, s_f\}$ is the same (or at least drawn from the same distribution), the propagated G_k value for each of the states $s_1, \ldots, s_4$ would be $-1 + \gamma v^l$ and for s_f this would be $\left(\sum_{\tau=0}^{k} -1\gamma^\tau\right) + \gamma^k v^l$. Because of this problem, the value of state S_f would be far lower than the others states, and MCTS would end up visiting it less (because of UCT rule). Finally, the resulting π distribution based on the number of visits on each state would be unfairly low for option $Solve$, which would cause the next episodes MCTS to visit even less this node (because this π distribution is used to update the f_θ function) preventing the algorithm from considering this option in the future.

To address this issue we use one idea from (SUTTON; BARTO, 2018, Section 10.3, Average Reward: A New Problem Setting for Continuing Tasks). We keep a set of all states that were already used to update in f_θ and g_θ (and therefore we have at least a more realistic value for v^l) and if the current state was never used to update these functions, instead of using the cumulative discounted reward for the backup of that state, we use the average reward in all execution, as in Equation 4.3.

$$G^k = \begin{cases} \overline{\mathcal{U}_k} + \gamma^{l-k} v^l & \text{otherwise,} \\ \left(\sum_{\tau=0}^{|\mathcal{U}^{k:l}|} \gamma^\mathcal{T} \mathcal{U}_\tau^{k:l} \right) + \gamma^{l-k} v^l & \text{if } v^l \text{ was already updated} \end{cases} \tag{4.3}$$

Finally, it is important to notice that many of these proposed changes can have several impact in how the algorithm performs. These changes and possible issues are discussed in more detail in the following section.

4.3 Algorithm Analysis

In this section with present a few of possible advantages and disadvantages of our technique. The experiments on Chapter 5 were based on this section, either to support or refute our initial analysis.

4.3.1 Increased planning horizon

We argue that, when using options, the planning algorithm is able to look further into the planning tree, and therefore do a better planning. The ability to look further can possibly help the agent to reduce variance in its estimates. However, the options also introduce bias, that could be beneficial at the beginning but could possible prevent the agent from finding the optimal solution. For example, if in the same grid world example in Section 4.2.3, the option *Solve* navigated to the goal state by a non-optimal path, this could cause MCTS to visit this option much more and the primitive actions that could eventually find the optimal path much less.

Differently from other MCTS with options techniques (Section 3.2) we actually store the next state of following the option entirely (we assume a deterministic environment), therefore when we are planning with the learned model of the environment, when selecting an option, we look the model up only once to get the result of following a k-step option, rather than looking up k times, once for each step, thus, allowing not for only looking further into the planning space but doing so without increasing the number of lookups in the forward model.

It is also interesting to notice that our approach is an increment on the MCTS algorithm: we can obtain the same original MCTS by just providing primitive options.

4.3.2 Decreased amount of simulations

One possible disadvantage of our technique is that, during play-time, the agent usually has limited time budget for each step. Given that our technique will try at least once one simulation for each possible option, by providing more options we end up increasing the branching factor for MCTS and, therefore, reducing the number of MCTS iterations. This issue does not arise during training: with no strict time constraints, we can use a budget of iterations rather than time.

This can possibly cause several problems in play-time, given that the branching factor of the tree will increase directly based on the number of possible options for each state and as demonstrated in (SILVER et al., 2017, Figure 6), only using the learned model play probabilities, without the planning during play-time (or with reduced amount of MCTS executions), greatly degrades the agent performance.

4.3.3 Pre-fetch in play-time

In order to mitigate the issue commented in Section 4.3.2, we can exploit the fact that we plan and act with options, with a technique we call pre-fetching. The main idea of pre-fetching is that, when following an option, the agent is basically idle during almost all its time budget and can use this idle time to plan. We can use our learned model to pre-fetch which will be the agent final state s_f after finishing the active option and use the time budget of the current step to run MCTS simulations with s_f as root, therefore, increasing the number of simulations we can obtain in total and improving our value estimates.

4.4 Summary

In this chapter we presented our version of the Option-Guided MCTS (Section 4.1) which changes the MCTS representation to use an option as a MCTS edge. Then, we introduced O-MuZero which presented several changes to the original MuZero algorithm to use options and the issues that arise with it, specially, the bootstrap issue which was solved using a mean reward approach (Section 4.2). Finally, we analyzed a few ideas of the algorithm (Section 4.3). Next chapter will present all the experiments we executed to corroborate or refute our technique advantages and disadvantages.

5 EXPERIMENTS

In this chapter we evaluate our technique on different Grid World environments based on benchmarks proposed by Sturtevant (2012). In Section 5.1 we present the environments and options we used for testing and the adaptations needed to use our technique in combination with the MuZero algorithm. Then, in Section 5.3 we present multiple experiments to empirically evaluate our technique in both learning and playing time.

5.1 Setting

This section briefly describes all the environments we tested and the options we developed to compare. We also describe how we simplified MuZero algorithm to work with tabular approximators instead of neural networks.

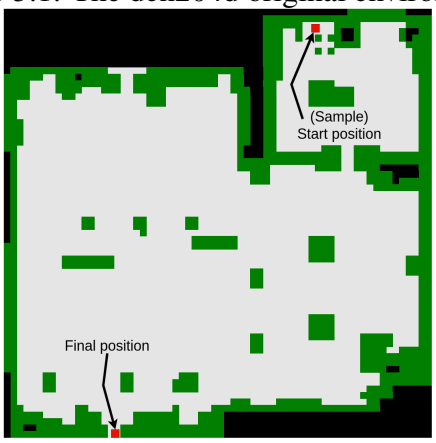
5.1.1 Environments

We evaluate our method on 4 variations of the map den204d from Sturtevant (2012). This is originally a 66×66 deterministic grid environment. The grid world works as an escape room: the agent starts at one position (which can be fixed or chosen randomly from a set of positions) and finishes when reaching another (fixed) position. For each step on the environment the agent receives a reward of -1, therefore, the sooner the agent finishes the environment, the higher the cumulative return. The agent always has 4 primitive actions available: $\{Left, Right, Up, Down\}$ which move it accordingly in the environment. The environment may contain walls (path blocking). When the agent would take an action that would move it in the wall position, it just stays in the same position, getting the regular -1 reward.

In order to avoid possible very long random simulations or even infinite ones, we make the environment non-Markov, by adding a time-out when the agent already took τ timesteps. In all our experiments we set $\tau = 10000$. We set the discount factor $\gamma = 0.95$.

The original den240d has two rooms: a bigger wider room and a smaller one. As a escape room simulation, the agent starts in the smaller room and has to first escape this room in order to be able to escape the bigger room. Figure 5.1 is an illustration of the environment.

Figure 5.1: The den204d original environment



Source: The Author

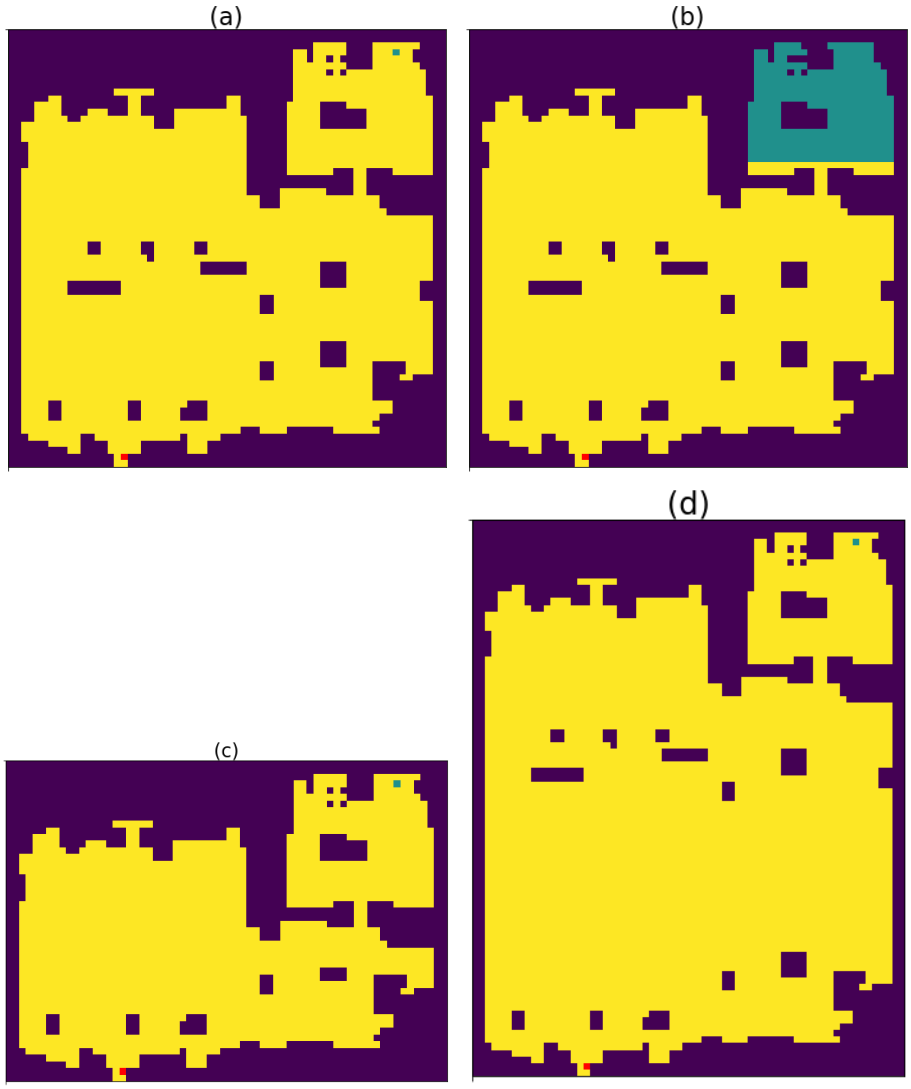
In order to evaluate our method in different settings, we propose 4 variations on the original den204d environment (Figure 5.2 shows them):

- **Single**: Agent starts at one fixed position and has to escape to another fixed position.
- **Multiple**: Agent starts at randomly chosen positions from a start set and has to escape to another fixed position (never in the start set).
- **Reduced**: Same as single, but slightly smaller environment(48×66).
- **Enlarged**: Same as single, but slightly bigger environment(85×66).

5.1.2 Options

In all our experiments we consider a **primitive** baseline agent, which uses only the 4 primitive actions. We also developed an agent with a **door** option with the initiation set contains all the positions in the first room. The door option just takes the agent from the smaller room to a fixed position in the outer room. Finally, we created an agent with a **macro** option, which just repeats a given action n times. In all our experiments we set $n = 3$. The agents always have access to the primitive actions. When we mention door option, we refer to the agent with primitive actions and the door option. The same is valid for macro options.

Figure 5.2: All den204d variations. Yellow are the legal positions and purple are the blocked path(ilegal) positions. *(a)* The **Single** environment. The agent starts in the green position on the top of the map and finishes on the red at the bottom. *(b)* The **Multiple** environment. The agent may start in position in the green region at the top and finishes on the green at the bottom. *(c)* The **Reduced** environment. *(d)* the **Enlarged** environment.



Source: The Author

5.1.3 Tabular MuZero

In order to apply our technique associated with on-line planning technique of Schrittwieser et al. (2020), we made a few simplifications in the standard MuZero algorithm.

First, the dynamic $g_\theta(s_t, o_t)$ and prediction $f_\theta(s_t)$ functions are replaced by tabular methods. Instead of using a Neural Network parametrized on θ, we use the tables G and F for predicting the same values as the neural network would. For default values, if G

does not contain the key of the state-option pair (s_t, o_t), it will predict the next state to be the same state s_t and a reward of 0. If F does not contain the state s_t, it will predict a uniform distribution over all possible options and a value of 0. These values are mostly used when learning is starting and the agent has not collected any data to train.

Secondly, since the states in the grid world are independent and having a history of the visited states does not change anything we set the number of stacked states passed for the encode h_θ function to 1. Furthermore, because we can encode the full state precisely enough using directly the coordinates (x, y) we can totally remove the need for a encode h_θ function.

Finally, the last simplification we did was on the update of the tables' values. MuZero has a separated thread using a prioritized buffer, because they have to collect data to keep training the network. In our simplified version, for the prediction model f we just use value iteration to update the table with the data collected at the end of each episode. For the dynamics model g, given a deterministic environment, we can just directly update the table content with the observations of the last episode. We do not keep the data in any sort of buffer. Equation 5.1 and Equation 5.2 are used to update the probabilities and values estimates in the F table.

$$F_v(s_t) = (1 - \alpha)F_v(s_t) + \alpha z \tag{5.1}$$

$$F_p(s_t) = (1 - \alpha)F_p(s_t) + \alpha \pi \tag{5.2}$$

5.2 Implementation Details

In this section we will briefly mention the technical details of our implementation. As previously mentioned all the code is publicly available on github [1] with all the results previously stored with the pickle library [2].

The whole project presented here was implemented from scratch using mainly Python 3.8 and numpy [3] (HARRIS et al., 2020). For parts of the code where performance could be a big bottleneck we used Cython [4] (BEHNEL et al., 2011), compiling

[1] https://github.com/otaviojacobi/tcc/tree/main/muzero-options
[2] https://docs.python.org/3/library/pickle.html
[3] https://numpy.org/
[4] https://cython.org/

our Python-like code to C++ code with much better performance. The experiments (independent trainings and independent plays) ran in parallel processes using the ray [5] library (MORITZ et al., 2018).

5.3 Results

In this section we present all the results and metrics which we evaluated our technique. Metrics can be seen in two separated moments: the metrics during learning time and play-time.

5.3.1 Evaluation metrics

We evaluated learning on two main metrics: first, the amount of steps taken to complete each episode and secondly on how many episodes the agent has timed out (> 10000 steps). While training, we used a fixed number of simulations (40) for each on-line planning step (given that during training time we do not have to limit ourselves to a time budget). We also present a few illustrations of the process of learning the value function.

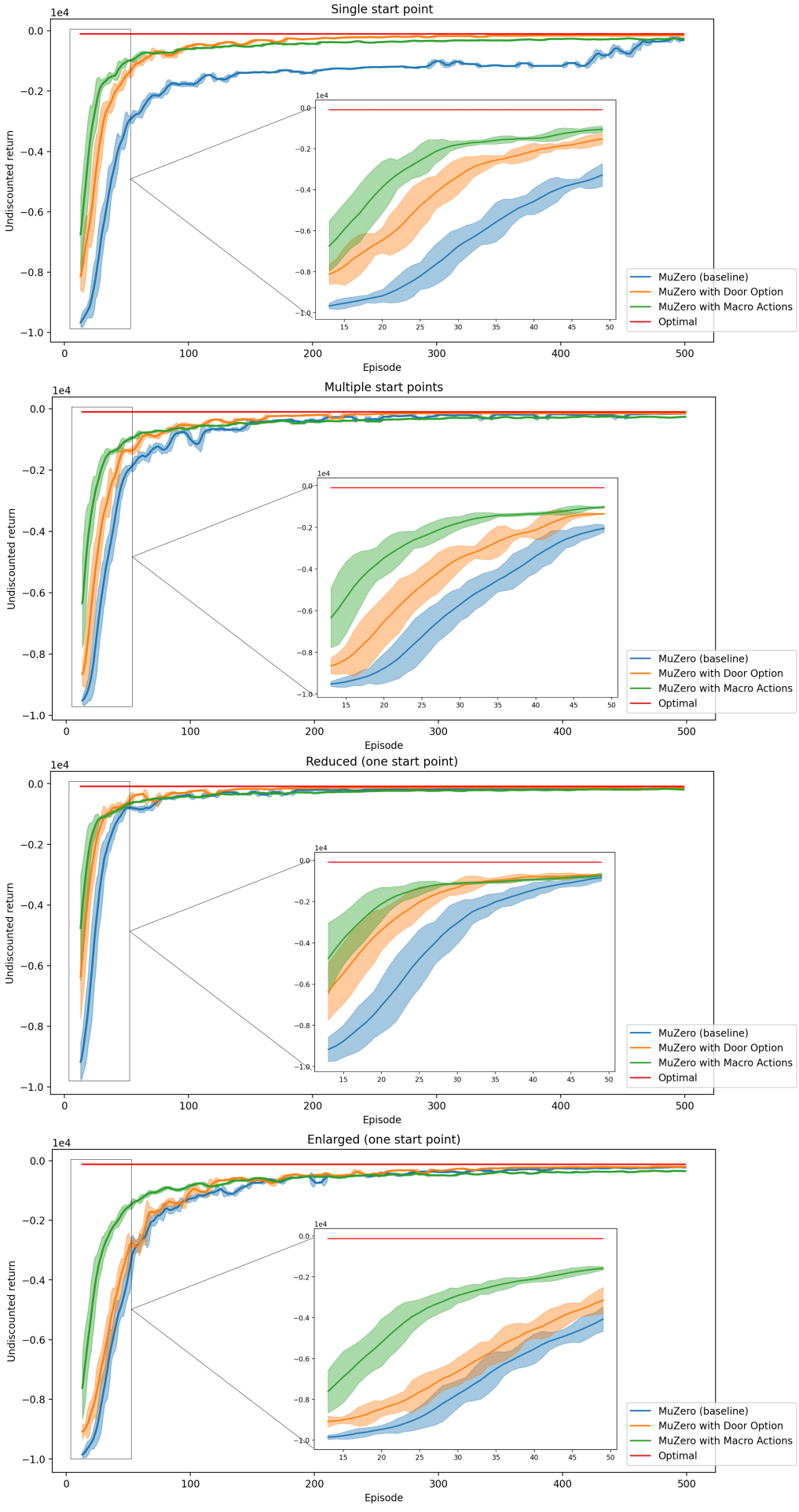
The play-time evaluation was done using a fixed time budget (40ms) for each action. We compare results of how many steps the agent took to exit the environment with or without pre-fetch (Section 4.3.3) and also compare the average number of simulations we were able to run for each state using or not pre-fetch.

5.3.2 Learning results

In this experiment we trained the full MuZero algorithm in the 4 proposed environments using the 3 different options (primitive, door, macro) for 500 episodes. The experiment that runs only the primitive actions is the exact MuZero algorithm and serves as a baseline reference. We repeated this experiment across 10 independent runs. Figure 5.3 highlights the moving average of the return for the first 50 episodes of the learning process, averaged over the 10 runs for each episode, showing that our technique learns faster in the initial stages of the learning process and therefore, converges faster to near-optimal results.

[5] https://www.ray.io/

Figure 5.3: Moving average of the return by episode across 10 independent trainings (50 episodes)



The results presented on Figure 5.3 are interesting because they show that, in the testbed environment, the use of options (macro actions and door option) greatly increases the learning speed, specially in the first episodes. It is also interesting to notice that the macro-actions are consistently better over all the environments, while the door option is better on the reduced environment than in the enlarged one. The reason for that is that in the reduced environment, the door option leaves the agent closer to solution, reducing the search space.

We now do a quantitative non-parametric analysis of how many times the environment timed-out. Table 5.1 shows the average times the environment timed out while learning. The table is read as: when using only primitive actions in the Single environment, out of the 500 training episodes, on an average of 10 repetitions, the agent timed out on 64.5 episodes.

Table 5.1: Average timeouts across 10 independent trainings (500 steps).

	Single	**Multiple**	**Reduced**	**Enlarged**	**Average**
option					
primitive	64.5	23.4	15.1	29.3	33.075
door	13.3	14.8	7.5	24.6	15.050
macro	6.0	5.1	2.8	8.3	5.550

Source: The Author

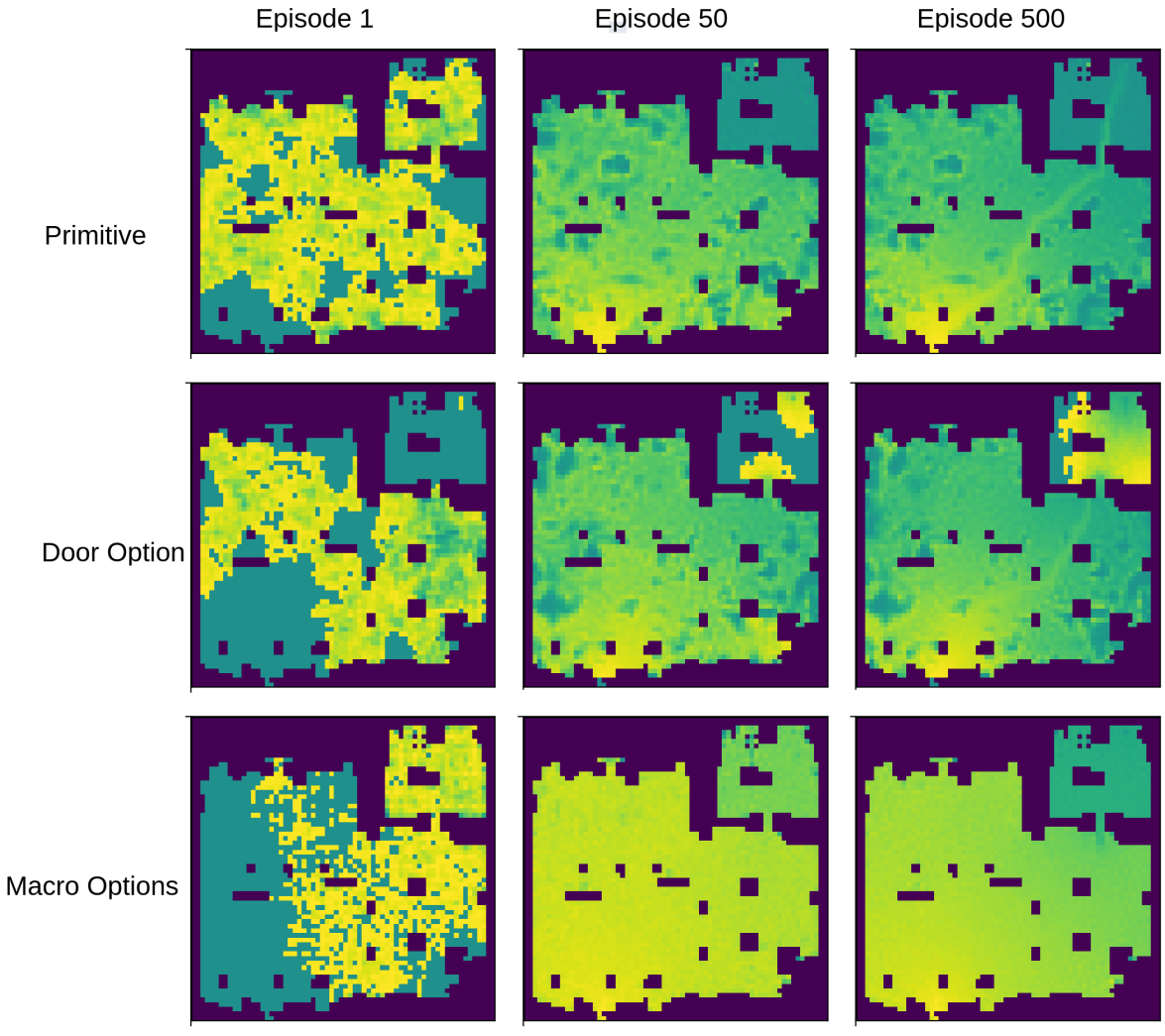
In order to prove that the results obtained by using options are significantly better than with only the primitive actions, we made a Kruskal-Wallis analysis followed by Dunn's test for each environment using $p = 0.05$. The Kruskal-Wallis test informed that for all environments there was a statistically difference in the results.

After executing the Dunn's test for each environment, we had lower p-values than the previously setted value 0.05 for all environments and across all options, except between the primitive and door option on the enlarged environment, where we had p=0.3. This is to be expected given that in the enlarged environment, the door option only barely helps the agent escape the first room, but the real challenge on this environment is the enlarged main room, which is bigger and therefore has higher variance. For this particular environment, we do not identify the door option as statistically better (the macro option, however, is considered better).

5.3.3 Learning the value function

In this subsection we present a few illustrations of how the agent learns the value function of the environment. We present these results as a way to illustrate how the agent sees the world building from zero (on the first iteration, the agent does not know anything neither about how the environment works nor about the value function). Figure 5.4 shows the estimate of the value function at different training iterations on the Single environment (closer to yellow means higher value).

Figure 5.4: Estimated value on different episodes while learning.



Source: The Author

The first interesting aspect of Figure 5.4 is that even though the final training result of the three models is similar, the learned value function has important differences. While the primitive model properly learns to give a higher value estimate to values near the final position, it also "overfits" for the optimal path of this environment from the starting

position to the final position (notice the smoother path coming from the first room until the final position).

The door model suffers with the same problem with but with one difference: because it learns to always take the door option (as it drives the agent out of the first room with the optimal path) it does not properly explore the positions inside the room.

Finally, the model using macro actions is very consistent: not only it does not overfit for the optimal path from the specific starting point, it has a more realistic value function with far less iterations (only 50 episodes and it already has a good estimate), it learns a smooth value function across all the environment. It is also interesting to notice that when using macro actions, the agent learns that positions inside the smaller room are worse than the ones outside of it. This happens because getting out of the room requires fine-grained control of the actions, which is harder for the agent with macro actions (even though it can theoretically achieve this fine-grained control given that it has the primitive actions, it is harder because it also expands the tree using the macro options).

5.3.4 Playing results

After analyzing the effects of our technique during learning time, we now do an analysis in order to discover if not only the MCTS with options helps the agent learning to behave in the environment but also if this results in better execution in play time.

All the experiments in this section used one of the trained models from Section 5.3.2. In all our experiments we set the agent thinking (planning) time to 40ms. For models where non-primitive options are available (door and macro) we tested using the standard version of the proposed algorithm and a version with the pre-fetch technique from Section (4.3.3). We compare both the number of steps needed to find the exit (the fewer, the better) and the number of iterations that the MCTS was able to run in the thinking time.

In Table 5.2 we show the averaged play results (across 20 independent games) of the standard (Std) and pre-fetch (PFetch) version of the algorithm. Each environment was tested with the model trained on that environment. We also added the optimal (minimum) result for each environment and the average amount of timesteps a random agent takes for solving it (in this case, we removed the 10000 steps limitation, because it was too frequent in random agents). For the Multiple environment, the optimal value presented is the average optimal value of all possible start positions.

Table 5.2: Number of steps to solve the environment across 20 independent runs. We highlight the best results (besides optimal) for each environment in bold.

	Single		**Multiple**		**Reduced**		**Enlarged**	
option	Std	PFetch	Std	PFetch	Std	PFetch	Std	PFetch
primitive	187.0	-	120.9	-	130.1	-	180.0	-
door	235.9	226.8	231.3	254.1	127.4	129.8	167.2	174.5
macro	219.6	**110.7**	123.0	**120.4**	85.8	**85.7**	138.4	**131.7**
random	31052.7		37044.5		23388.4		53193.5	
optimal	102		104		84		120	

Source: The Author

For all environments, using the macro-actions with pre-fecth performed better than all the other options. Pre-fetching with the door option had almost no effect during play time (and in many cases even made it worse than not using this option), as this option is only used once in playtime (once the agent is out of the smaller room, it never returned to it) the effect of pre-fetching can not be noticed.

It is also interesting to notice that the effect of pre-fetching is less impactfull in smaller environments. This happens because in smaller environments the 40ms budget allows for simulations that constantly reach the end of the episode (and backup sooner) than the ones in the bigger environment.

Table 5.3 shows the mean number of MCTS iterations on each state at playtime. Notice that the amount of simulations per state sharply increases when using pre-fetch and that this directly correlates with the increased performance shown in Table 5.2.

Table 5.3: Average amount of MCTS simulations executed on each state.

	Single		**Multiple**		**Reduced**		**Enlarged**	
option	Std	PFetch	Std	PFetch	Std	PFetch	Std	PFetch
primitive	198.4	-	224.0	-	180.8	-	202.5	-
door	213.2	221.7	215.4	227.5	192.2	211.8	211.7	207.2
macro	204.9	394.0	201.7	372.1	198.7	407.1	208.8	362.9

Source: The Author

These results suggests that not only our proposed MuZero with options help the agent learn faster, but also improve play time performance (when bound by a time budget) by properly exploiting the use of options in play time with pre-fetching.

Our last experiment, we test the robustness of the trained models. In special, we wanted to compare how would the model trained on the Single environment perform in the Multiple environment. We show in Table 5.4 that options can greatly improve the performance of the agent in this environment with different start positions. The agent trained in the Single environment with only primitive options performs poorly, whereas

the agent trained in the Single environment with options (both door and macro) have much better results. The main reason behind these results are, in the case of the door option, the agent can reuse the knowledge of its option across different environments and even though we changed the starting point for the agent trained on the Single environment, the fact that they are similar and share the door option allow it to perform better besides this change. For the macro action case, the agent performed better because it learned a better value function overall.

Table 5.4: Steps to solve the Multiple after training on Single.

	Std	**PFetch**
option		
primitive	2156.3	-
door	114.7	113.6
macro	113.0	123.2

Source: The Author

5.4 Summary

This chapter presented several experiments and results suggests that we can obtain great improvements on on-line planning algorithms through the use of options. Not only these experiments showed that we can improve learning speed but also the performance of the final played games. We also showed that pre-fetching further improves the performance of our approach. Finally, we showed that options can also improve the model robustness to play on different start conditions than the one the agent trained on. Finally, maybe one of the most interesting aspects of this research is the fact that the options used were very simple (specially the macro action) and already yielded great results. In the next chapter we further discuss the conclusions we can take from these results and many of the future challenges arising on this field.

6 CONCLUSION

In this chapter we discuss more in depth the conclusions we can take from this research (Section 6.1) and possible future work to be developed (Section 6.2).

6.1 Overview

In this work we have introduced a novel algorithm that uses the option framework (SUTTON; PRECUP; SINGH, 1999) to guide a Monte Carlo Tree Search. We then combined our algorithm with a simplified version of state-of-the-art on-line MCTS planning algorithm MuZero (SCHRITTWIESER et al., 2020) to empirically show that our algorithm can enhance both the training and playing stages of this technique.

In this work we also identified possible arising issues when combining our technique with MuZero, such as the bootstrap issue (Section 4.2.3). We proposed one possible solution for it, however, this approach is still very initial and does require more investigation about possible impacts of using the mean reward of an option when bootstrapping. We also discussed how using too many options can worsen our algorithm performance by increasing too much the MCTS branching factor. In order to mitigate this issue we introduced the pre-fetching technique, that can greatly enhance the amount of simulations the planning agent can execute during play time.

We evaluated our method in different variations of a escape room grid world based on standard grid world benchmarks proposed by (STURTEVANT, 2012). We showed that our algorithm using macro-actions combined with pre-fetching technique yielded better results in all environments across multiple independent runs. We also showed that pre-fetching can sharply increase the amount of simulations executed on a given time budget specially on options that have a large initiation set. Moreover, we conducted one last experiment to show that, besides all previously cited benefits of our technique, a model trained with options may be more robust to small changes on the play environment, given that options may help the agent to generalize its behavior independent of these changes.

Finally, we can conclude that our technique seems promising on small and controlled environments such as the Grid World, however, many challenges are still open, specially regarding on how can this technique be scaled for larger and continuous space/action domains.

6.2 Future work

The results presented are exciting and corroborate with the main ideas that we initially wanted to prove initially, however, many issues are still left open. In this final section we detail more about these issues and future research direction. The real challenge and question for future work is 'How do we scale up MCTS with options?'. How can our technique be associated with linear function approximators (e.g. neural networks) and operate on continuous state/action environments? How can the options used by our technique be automatically discovered and what benefits could that yield? This section goes over a few of this ideas.

6.2.1 Function Approximators

In our work, we replaced MuZero neural networks by a table and used a discrete-state environment. More challenging domains such as Video Game Playing do not allow for such simplifications. In order to apply our technique in these environments, the function approximators would have to change their outputs, specially for the dynamics model g_θ. Firstly, we would have to encode an option to be fed as input to the approximator. Similar to what they do on (SCHRITTWIESER et al., 2020) we could use an id for the option and represent it as a input vector for the approximator.

The other challenge for the dynamics model is the fact that the output of this environment is now a list of rewards $\mathbf{R}$. Given that we need to predict the option duration before executing it, we do not have access to the list size previously. Outputting a list of arbitrary size is a complex task for a approximator such as a Neural Network. Fixing a size of the output list is also not a good option given that it would limit the execution size of the options. For now, the best idea we have analyzed is make the approximator predict two values for a given state and option: the fully discounted return of the option and for how many steps it will execute. This way we can replace the full list of rewards on each edge by the discounted cumulative reward and option size. The neural network could be trained having these values coming from real experience and stored on a replay buffer. Another approach could be calling the model multiple times and building the list from many calls to the environment model, however, we are not much in favor of this second idea as one of the main goals of our technique is to keep the single call to the dynamics model.

6.2.2 Option Learning

Other aspect of our work is that all the options used (door option and macro-actions) were manually created by us. Even though macro actions are very simple and present very good results, it is hard to say to which extent our technique can be even more beneficial to the learning process if we had engineered more options. However, manually defining each option can be a very challenging task and different works try to automatically learn options based on sub-goals (MCGOVERN; BARTO, 2001) or by parametrizing an option policy and then directly optimizing it (BACON; HARB; PRECUP, 2017).

Our technique can be combined with both ideas and we would expect very interesting results coming from this, given that these learned options could possibly be used across many different environments, leveraging learning and playing in a more generic way.

REFERENCES

AUER, P. Using confidence bounds for exploitation-exploration trade-offs. **Journal of Machine Learning Research**, v. 3, n. Nov, p. 397–422, 2002.

BACON, P.-L.; HARB, J.; PRECUP, D. The option-critic architecture. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2017. v. 31, n. 1.

BAI, A.; SRIVASTAVA, S.; RUSSELL, S. J. Markovian state and action abstractions for mdps via hierarchical mcts. In: **IJCAI**. [S.l.: s.n.], 2016. p. 3029–3039.

BEHNEL, S. et al. Cython: The best of both worlds. **Computing in Science & Engineering**, IEEE, v. 13, n. 2, p. 31–39, 2011.

BROWNE, C. B. et al. A survey of monte carlo tree search methods. **IEEE Transactions on Computational Intelligence and AI in games**, IEEE, v. 4, n. 1, p. 1–43, 2012.

ELLIS, K. et al. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. **arXiv preprint arXiv:2006.08381**, 2020.

GREGOR, K.; REZENDE, D. J.; WIERSTRA, D. Variational intrinsic control. **arXiv preprint arXiv:1611.07507**, 2016.

HARRIS, C. R. et al. Array programming with NumPy. **Nature**, Springer Science and Business Media LLC, v. 585, n. 7825, p. 357–362, sep. 2020. Available from Internet: <https://doi.org/10.1038/s41586-020-2649-2>.

KENESHLOO, Y. et al. Deep reinforcement learning for sequence-to-sequence models. **IEEE transactions on neural networks and learning systems**, IEEE, v. 31, n. 7, p. 2469–2489, 2019.

KOCSIS, L.; SZEPESVÁRI, C. Bandit based monte-carlo planning. In: SPRINGER. **European conference on machine learning**. [S.l.], 2006. p. 282–293.

MCGOVERN, A.; BARTO, A. G. Automatic discovery of subgoals in reinforcement learning using diverse density. 2001.

MNIH, V. et al. Playing atari with deep reinforcement learning. **arXiv preprint arXiv:1312.5602**, 2013.

MORITZ, P. et al. Ray: A distributed framework for emerging {AI} applications. In: **13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)**. [S.l.: s.n.], 2018. p. 561–577.

PATERIA, S. et al. Hierarchical reinforcement learning: A comprehensive survey. **ACM Computing Surveys (CSUR)**, ACM New York, v. 54, p. 1–35, 2021.

PEREZ, D. et al. Solving the physical traveling salesman problem: Tree search and macro actions. **IEEE Transactions on Computational Intelligence and AI in Games**, IEEE, v. 6, n. 1, p. 31–45, 2013.

PEREZ, D. et al. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In: **Proceedings of the 15th annual conference on Genetic and evolutionary computation**. [S.l.: s.n.], 2013. p. 351–358.

PEREZ-LIEBANA, D. et al. Introducing real world physics and macro-actions to general video game ai. In: IEEE. **2017 IEEE Conference on Computational Intelligence and Games (CIG)**. [S.l.], 2017. p. 248–255.

PINTO, I. P.; COUTINHO, L. R. Hierarchical reinforcement learning with monte carlo tree search in computer fighting game. **IEEE transactions on games**, IEEE, v. 11, n. 3, p. 290–295, 2018.

RUMMERY, G. A.; NIRANJAN, M. **On-line Q-learning using connectionist systems**. [S.l.]: Citeseer, 1994.

SCHAUL, T. et al. Prioritized experience replay. **arXiv preprint arXiv:1511.05952**, 2015.

SCHRITTWIESER, J. et al. Mastering atari, go, chess and shogi by planning with a learned model. **Nature**, Nature Publishing Group, v. 588, n. 7839, p. 604–609, 2020.

SILVER, D. et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. **arXiv preprint arXiv:1712.01815**, 2017.

SILVER, D.; VENESS, J. Monte-carlo planning in large pomdps. In: NEURAL INFORMATION PROCESSING SYSTEMS. [S.l.], 2010.

STURTEVANT, N. Benchmarks for grid-based pathfinding. **Transactions on Computational Intelligence and AI in Games**, v. 4, n. 2, p. 144 – 148, 2012. Available from Internet: <http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf>.

SUTTON, R. S. Learning to predict by the methods of temporal differences. **Machine learning**, Springer, v. 3, n. 1, p. 9–44, 1988.

SUTTON, R. S. Dyna, an integrated architecture for learning, planning, and reacting. **ACM Sigart Bulletin**, ACM New York, NY, USA, v. 2, n. 4, p. 160–163, 1991.

SUTTON, R. S.; BARTO, A. G. **Reinforcement learning: An introduction**. [S.l.]: MIT press, 2018.

SUTTON, R. S.; PRECUP, D.; SINGH, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. **Artificial intelligence**, v. 112, p. 181–211, 1999.

VIEN, N. A.; TOUSSAINT, M. Hierarchical monte-carlo planning. In: **Twenty-Ninth AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2015.

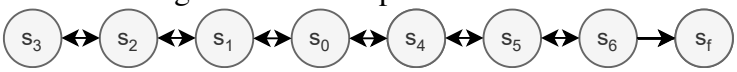
WAARD, M. D.; ROIJERS, D. M.; BAKKES, S. C. Monte carlo tree search with options for general video game playing. In: IEEE. **2016 IEEE Conference on Computational Intelligence and Games (CIG)**. [S.l.], 2016. p. 1–8.

WATKINS, C. J. C. H. **Learning from delayed rewards**. Thesis (PhD) - King's College, Cambridge United Kingdom, 1989.

APPENDIX A — O-MUZERO EXECUTION EXAMPLE

Here we present an example of a O-MuZero execution in order to better illustrate how the return value for each node is calculated (as presented in Equation 4.2). In this example we use an environment as in Figure A.1. The agent starts on s_0 and its goal is to reach s_f. The agent has two actions $\{Left, Right\}$. For each transaction on the environment the agent receives a reward of -1 except when it goes from s_6 to s_f when it receives a reward of 1000. The episode finishes when it reaches s_f. If the agent takes $Left$ on s_3 it stays in the same place and gets the -1 reward.

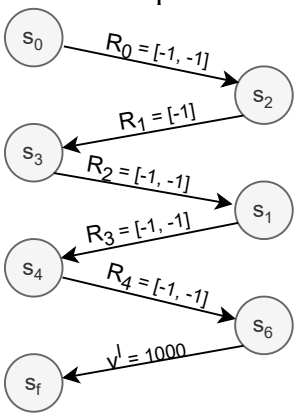
Figure A.1: Example environment



Source: The Author

In this example, we enhance the set of actions with two macro-actions that execute the action for two time-steps O^2_{Left} and O^2_{Right}. The full set options of which the O-MuZero can choose from is: $\{Left, Right, O^2_{Left}, O^2_{Right}\}$ .We now present one hypothetical execution of the MCTS algorithm which explores the following sequence of states and options: s_0, O^2_{Left}, s_2, $Left$, s_3, O^2_{Right}, s_1, O^2_{Right}, s_4, O^2_{Right}, s_6, $Right$, s_f. Where s_f is the node recently expanded. Figure A.2 illustrates this execution relevant nodes (other MCTS nodes are hidden for clarity).

Figure A.2: Example MCTS nodes



Source: The Authors

After executing this, we run the O-MuZero algorithm as explained in Chapter 4, for each node s_k we will backup the G_k value. In this example we will calculate the return value to be updated on each node. As in the algorithm, backup starts from the node before the just expanded s_f node, which in our case is s_6.

The full list $\mathcal{J} = (\mathbf{R}_0, \mathbf{R}_1, \ldots, \mathbf{R}_4) = ([-1,-1],[-1],[-1,-1],[-1,-1],[-1,-1])$. Now, for each node in the tree, starting from one before the last expanded, we compute its G_k. Let us start with s_6 and its return value G_5.

First we get the unrolled list $\mathcal{U}^{5:5}$, which is an empty list, and therefore according to Equation 4.2, $G_5 = \gamma^0 v^l = 1000$.

Now, we move on to node s_4 and its return G_4. Following the same process we first compute the unrolled list $\mathcal{U}^{4:5} = [-1,-1]$ and now we plug it into Equation 4.2 to obtain $G_4 = -1\gamma^0 - 1\gamma^1 + 1000\gamma^2$, which assuming $\gamma = 0.95$ is, $G_4 = 900.55$.

Following the same for s_1 and G_3 we have $\mathcal{U}^{3:5} = [-1,-1,-1,-1]$ and $G_3 = 810.8$. For s_3 and G_2 we have $\mathcal{U}^{2:5} = [-1,-1,-1,-1,-1,-1]$ and $G_2 = 729.8$. For s_2 and G_1 we have $\mathcal{U}^{1:5} = [-1,-1,-1,-1,-1,-1,-1]$ and $G_1 = 692.3$. Finally, for S_0 and G_0 we have the full unrolled list $\mathcal{U}^{0:5} = [-1,-1,-1,-1,-1,-1,-1,-1,-1]$ and $G_0 = 622.9$.