Reinforcement Learning with Hard-Coded policies

Federico Taschin
KTH Royal Institute of Technology
Stockholm, Sweden
taschin@kth.se

***Abstract*—Reinforcement Learning deals with learning complex behaviors purely from experience, maximizing a reward function. Such complex behaviors have been traditionally developed by careful algorithm design, which has been their main limitation. As RL promises to achieve what programmers cannot encode, it still requires huge amount of experience and computational power. In this work, we research techniques to learn optimal behaviors by exploiting sub-optimal hard-coded policies. First experiment RL pre-training techniques from a given sub-optimal policy. Then, we develop an RL architecture that uses the given policy as a backbone and learns to improve it. Finally, we develop an off-policy hierarchical architecture, composed of a manager policy and two sub-policies, one hard-coded and one learning. We then try undertaking an high-complexity task in the Minecraft environment, although we did not achieve this last goal.**

***Index Terms*—reinforcement learning, behavior tree, off-policy, hierarchical**

I. Introduction

Developing algorithms capable of complex behavior has been a longstanding goal of several research areas, such as robotics, finance, or video games. Such behaviors are often carefully designed by programmers or specialists, and typically require days or months of work. Moreover, it is often impossible to design every part of such behaviors. On the other hand, Reinforcement Learning is being increasingly used to learn such behaviors purely from experience. Its most successful applications showed super-human performance in high-complexity games such as Go[17] and Dota[16]. It is however computationally expensive to learn high-complexity tasks -the game of Go was solved using more than 100k CPUs and 256 GPUs. In this work, we try to bridge the gap between hard-coded policies and Reinforcement Learning, by exploring ways in which an hard-coded policy may be improved with RL, or vice versa. We focus on off-policy RL algorithms, that generally require less interactions with the environment, and Behavior Trees, control structures that allow to easily code complex behaviors. We first evaluate several ideas to improve an hard. -coded policy with RL, and then develop a BT to solve an high-complexity task. Although this last goal is not achieved, we highlight many issues and ideas for further research.

II. Background and previous work

In this section we give an overview of the relevant theory on which our work relies. We describe the Behavior Tree architecture and the theoretical grounds of Reinforcement Learning, along with the algorithms we use.

A. Behavior Trees

Behavior Trees [4] are hierarchical, tree-like control structures that allow efficient and modular designing of complex behaviors. Behavior Trees have been extensively employed in the video game industry [11] and are increasingly used in robotics applications [10]. In a Behavior Tree, atomic tasks (actions) are represented as leaves, while the rest of the tree is composed of control nodes that determine their execution flow. A tick signal is propagated from the root of the tree at a certain frequency, allowing re-evaluation of control nodes and thus a reactive behavior. Every time a node is executed, it returns a control signal that can be *Success*, *Failure*, *Running*. The most important control nodes are *Sequence*, *Fallback*, *Parallel*, and *Decorator*. The *Sequence* node ticks its children sequentially until one returns *Failure* or *Running*, and returns that signal. The *Selector* node (also called *Fallback*) propagates the tick until a child node returns *Success* or *Running*, and returns that signal. The *Parallel* node ticks all its children and returns *Success* or *Failure* based on some predefined thresholds, below which it returns *Running*. The *Decorator* node allows to change the return signal of a child node.

B. Reinforcement Learning

In the Reinforcement Learning framework an agent interacts with an environment by reading its *state* and performing *actions*, with the goal of maximizing a *reward* function that is generally pre-defined and environment-dependent. A common assumption is that the aforementioned interaction can be described as a Markov Decision Process (MDP) on the state space $S \subseteq \mathbb{R}^d$ and action space $A \subseteq \mathbb{R}^l$ in an environment that behaves according to the conditional distribution $P(s_{t+1}|s_t, a_t)$. The reward is a random variable $r \sim p(r_t|s_t, a_t)$, although it is often reduced to the deterministic case where it is given by a function $S \times A \to \mathbb{R}$ which is generally non-differentiable. The agent chooses actions according to a *policy* $\pi(a|s)$. We denote with uppercase S_t, A_t, and R_t the nodes at time-step t in the Markov chain induced by the MDP, and with lowercase s_t, a_t, and r_t the possible values they can assume in that time-step. A *trajectory* τ is a sequence $(s_0, a_0, r_0, ..., s_t, a_t, r_t)$ experienced by the agent and its distribution depends on both the environment *transition* distribution $P(s_{t+1}|s_t, a_t)$ and the agent's policy. The goal is to maximize the total *discounted return* $G_t^\gamma = \sum_{t=t'}^{\infty} \gamma^{t'-t} r_{t'}$. The *value function* $V(s)$ of a state s at time-step t is defined as the expected value of the discounted return given the current state: $V(s_t) = \mathbb{E}[G_t^\gamma | S_t = s_t]$. The *action-value function*

$Q(s,a)$ is similarly defined as the expected discounted return given the current state s and the chosen action a at time-step t: $Q(s_t,a_t) = \mathbb{E}[G_t^\gamma | S_t = s, A_t = a_t]$. For a given policy π, the action-value function can be written as a *Bellman Expectation Equation*:

$$Q^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}\left[\mathbb{E}_{a'}[Q^\pi(s',a')]\right] \tag{1}$$

And for an optimal policy π^*, the optimal action-value function satisfies Bellman's Principle of Optimality[21]:

$$Q^*(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}\left[\max_{a'} Q^\pi(s',a')\right] \tag{2}$$

Off-policy RL algorithms make use of 2 to learn an optimal policy from experience obtained by past versions of the learned policy. In this work, we implement and use for our experiments the DQN[14] and TD3[6] algorithms.

1) DQN: The DQN algorithm[14] is a model-free RL algorithm that works in discrete-actions settings and exploits Neural Networks to learn the Q function. In DQN, experience at each time-step is stored in a *replay buffer* in the form of transitions (s,a,r,s'). A Neural Network Q_θ is trained to learn the value of each action for a given state. To improve stability, a target network Q_ϕ is employed and training is performed with gradient steps given by:

$$\nabla_\theta L(\theta) = \mathbb{E}_{s,a,s'}\left[(y - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a)\right] \tag{3}$$

where

$$y = r + \gamma \max_{a'} Q_\phi(s',a') \tag{4}$$

is called the target value. The expected value of 3 is computed by sampling batches of transitions from the replay buffer and averaging the gradients. The target network $Q_\phi$is kept fixed and updated with Q_θ every k steps. In our work we implement the Double Q-Learning[9] technique to reduce overestimation of the Q values. During training, actions are selected with an ϵ-greedy policy, that selects the $\arg\max$ action with probability $1-\epsilon$ and a random action with probability ϵ.

2) Twin Delayed DDPG (TD3): The TD3 algorithm[6] can be viewed as the continuous-actions extension of DQN. TD3 employs an actor network π_ϕ to map states into actions and two Q networks Q_θ^1 and Q_θ^2 that compute the value of the given state-action pairs. To reduce overestimation of the Q values, the $\min$ between Q^1 and Q^2 is used. The Q networks are trained as in 3. The targets y are computed by sampling an action $a' \sim \pi_\phi(a'|s')$ and computing:

$$y = r + \gamma Q_\theta(s',a') \tag{5}$$

The $\max$ operation is performed by training the actor to maximize the value predicted by Q_θ using the chain rule $\frac{dQ_\theta}{d\phi} = \frac{dQ_\theta}{da}\frac{da}{d\phi}$. During training, noise $\epsilon \sim \mathcal{N}(0,\sigma^2)$ is applied to the action for exploration.

C. Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning[3] deals with learning hierarchies of policies. In this work we make use of the Options framework[20]. In the Options framework, an option $o \in \Omega$ is a tuple $(\mathcal{I}, \pi, \beta)$, where $\mathcal{I} \subseteq S$ is a set of initial states, π is a policy over actions, and β is a termination condition. In the original Options formulation, the policy π is a given policy that can be selected in state $s \in \mathcal{I}$ at time t, and is executed for any number k of steps, terminating according to $\beta(s)$. At any time-step t, the probability of executing an option o is given by the policy over options $\mu : S \times O \to [0,1]$. The option-value function is then defined as:

$$Q_\Omega(s,o) = \mathbb{E}_{a\sim\pi_o(a|s)}\left[Q_U(s,o,a)\right] \tag{6}$$

where where π_o is the internal policy of option o and $Q_U(s,o,a)$ is the value of executing the action a in state s under the option o:

$$Q_U(s,o,a) = r(s,a) + \gamma \mathbb{E}_{s'}\left[U(o,s')\right] \tag{7}$$

and $U(o,s')$ is the value function *upon arrival*, i.e. the expected discounted return from s' in option o onward. The Option-Critic architecture[2] provides a Policy Gradient[18] based algorithm to learn not only the policy over options, but also the option internal policies and their termination conditions.

III. Methods

A. Pre-training the actor policy

To warm-start the RL training we use expert demonstrations or hardcoded policies to pre-train the networks. This is often not a simple task as expert demonstrations generally lack of exploration. More generally, performing Reinforcement Learning completely off-policy suffers from extrapolation issues[7], overestimating the value of unseen state-action pairs. We experiment how to pre-train the TD3 algorithm from a given hard-coded policy. When pre-training the TD3 algorithm we need to learn both the actor policy and the critic values. While pre-training the actor is easy -one can just learn to mimic the hardcoded policy- learning the critic values is more difficult. We pre-train TD3 by performing actions from the hardcoded policy and storing the transitions obtained. We add Gaussian noise $\epsilon \sim \mathcal{N}(0,\sigma^2)$ to the hard-coded actions to ensure exploration. The actor is trained by minimizing the MSE loss between the hard-coded policy μ and the actor π_ϕ:

$$\mathcal{L}(\phi) = \sum_{n=1}^{N} (\mu(s) - \pi_\theta)^2 \tag{8}$$

The critic is instead trained by minimizing the squared *temporal-difference* error[19] similarly to 4:

$$\mathcal{L}(\phi) = \left(r + \gamma \hat{Q}_\theta(s',a') - Q_\theta(s,a)\right)^2 \tag{9}$$

where $\hat{Q}_\theta$ is a target network that is kept fixed and updated as in TD3. The next action a' is taken from the training policy π_θ. Additionally, we try to fully train the policy in a TD3 fashion from the collected experience only, i.e. without any new interaction with the environment. Results can be seen in Section IV.

B. Hardcoded Policy as a Backbone

Pre-training the Neural Networks from an hard-coded policy can give a good warm-start and speed up training. However, it discards the hard-coded policy after the pre-training is done, and it is equivalent to learning the full policy. In many situations we do not want to dispose of the hard-coded policy, but we may want to improve it, for example, when the complexity of the task is high or the given policy already has good performances.

In the *Backbone* training we use the hard-coded policy μ together with an RL policy π_ϕ. Assuming that μ is sub-optimal but performs better than random, the RL policy π_ϕ learns to correct the action taken by μ. For a given state s, the resulting action that is taken in the environment is therefore

$$a = \mu(s) + \pi_\phi(s) \tag{10}$$

We implement the Backbone training on the TD3 algorithm with minimal changes. The replay buffer stores separately the $\mu(s)$ and $\pi_\phi(s)$ components of actions, and it is pre-filled by taking random actions $a = \mu(s) + \epsilon$ in the environment, with $\epsilon \sim \mathcal{N}(0, \sigma^2)$. The Q networks of the TD3 algorithm now predict the value of $(s, \pi_\phi(s)))$ pairs instead of (s, a). The RL policy π therefore learns the "correction" to be applied to the action chosen by the hard-coded policy μ, which acts as a "backbone". During training, noise is added to the correction $\pi_\phi(s)$ in the same way of TD3. In this setting, the goal is to make the policy μ drive the behavior and the exploration, biasing the algorithm along "good" trajectories. The correction predicted by π_ϕ can be clipped in the range $[a_{min}, a_{max}]$ to tune this behavior. As results in Section IV show, a range with smaller $|a_{max} - a_{min}|$ leads to a stable training that closely follows the hard-coded policy μ, at the cost of having less margin for improvement. A larger range allows achieving higher rewards as the RL policy can perform stronger corrections, at the cost of a longer training.

The Backbone training can be extended to the case when the hard-coded policy is another instance of a TD3 algorithm or, more generally, a pair of actor policy $\hat{\mu}$ and action-value estimator $\hat{Q}$. In this case, the target values of 5 are computed as

$$y = r + \gamma \left(\hat{Q}(s', \hat{\mu(s')}) + Q_\theta(s', \pi_\phi(s')) \right) \tag{11}$$

In this case the critic network Q_θ learns to correct the value estimated by $\hat{Q}$.

C. Hierarchical Training

Another way of incorporating an hard-coded policy into the RL framework is to view it as an option in the Options Framework described in Section II-C. We take inspiration from the Option Critic Architecture[2] described in Section II-C, but our work differs in the fact that we use an off-policy algorithm (TD3) to learn the internal RL policy. Our options correspond to the hard-coded policy π^H and the RL policy π^{RL}_ϕ. Also, we do not learn the termination condition $\beta(s)$ and we instead switch between the two options depending on which one has the higher value as in [13].

We define a simple hierarchical architecture in which the option-value network $Q_\theta(s, o) : S \times \Omega \rightarrow \mathbb{R}$ approximates the value of choosing option o in state s. Once the option (π^H or π^{RL}) is chosen, the correspondent sub-policy performs a number k of steps, collecting a trajectory $(s_0, a_0, r_0, ..., s_{k-1}, a_{k-1}, r_{k-1}, s_k)$. The policy-over-options μ therefore operates on an higher temporal scale.

It is important to note that in such a configurations sub-policies are not independent. The value of an action a in state s in option o with sub-policy π is given by

$$Q^\pi(s, a) = r(s, a) + \gamma \mathbb{E}_{s'} \left[U(s', .) \right] \tag{12}$$

with $U(s')$ being the expected total return from the next state s' onward. Since an option is performed for k steps, at step $i < k$ this is $U(s', o)$, i.e. the value of state s' by continuing with option o. At step k instead, $U(s')$ is given by $\mathbb{E}_\mu[U(s', \mu(s'))]$ and depends on the policy-over-options in state s'. From the point of view of the policy-over-options μ and its option-value function Q_θ, the value of a state-option pair (s, o) is given by

$$Q(s, o) = \left(\sum_{i=0}^{k-1} \gamma^i r_i \right) + \gamma^k \mathbb{E}_\mu \left[Q(s, o') \right] \tag{13}$$

The policy-over-options μ is parametrized by a neural network that we train in an off-policy fashion using Double Q Learning [9] and sampling from a replay buffer of the collected multi-step transitions. The RL policy π^{RL} is parametrized by two networks: an actor network π^{RL}_ϕ and a critic network Q^{RL}_β that are trained with the TD3 algorithm [6].

The configuration we just described poses the issue of learning from multi-step off-policy trajectories. Common value-based off-policy methods such as DQN or [14], TD3 [6] can effectively learn off-policy from a single step transition exploiting equations 3, 4, and 5 because the reward of a pair (s, a) does not depend on the current policy. However, the same does not apply to the multi-step reward $r_k = \sum_{i=0}^{k-1} \gamma^i r_i$ since rewards for $i = 1..k-1$ depend on the actions taken in those steps. We therefore perform off-policy learning by modifying the TD3 target with the Retrace [15] target. The Retrace target computes the target value used in the critic update by applying a clipped importance weight factor. The recursive definition of the Retrace target Q^{ret} is

$$\begin{aligned} Q^{ret}(s, a) = r + \gamma \rho \left[Q^{ret}(s', a') - Q(s', a') \right] \\ + \gamma V(s') \end{aligned} \tag{14}$$

where $\rho = \frac{\pi(a'|s')}{\pi^{old}(a'|s')}$, i.e. the ratio of the likelihood of a under the target sub-policy π and under the version of the exploration sub-policy π^{old} at the time the trajectory was collected. $Q(s', a')$ is estimated with the TD3 target network while $V(s')$ is computed from the option-value network. In particular, $V(s) = \max_{o'} Q_\theta(s', o')$ if s' corresponds to the last step of the trajectory, and $V(s') = Q_\theta(s', o)$ otherwise, where o is the current option being executed.

D. The Minecraft Behavior Tree

We develop a Behavior Tree for a Minecraft agent with the goal of obtaining a diamond. We use the Project Malmo platform[12] as environment. The task is particularly hard, as it requires to collect resources in the Minecraft environment, craft several types of items, survive to hunger and enemies, and dig until a diamond is found. Such a problem would be extremely hard to learn by pure Reinforcement Learning, but writing a good hardcoded policy is not an easy task either. We exploit the Behavior Tree framework [4] and the PyTrees[1] Python3 library to develop a Behavior Tree to achieve such a task. The whole Behavior Tree is too large to be explained in this paper, and we will focus on the sub-tree that moves the agent to a desired destination block.

The Minecraft environment is composed of blocks that can be placed or destroyed. We allow our agent to destroy blocks to create a path if necessary. In order to plan the path, we use the A star algorithm[8] on an augmented space. The state space is given by the position of the agent and a 3-dimensional cube $S \in \{0,1\}^{15\times15\times15}$ where each block has value 1 if empty or 0 if full. The transition between a state and its successor is defined by a target block and an action that can be *destroy* or *move_to*. *destroy* can be applied only to a neighboring block and has a cost $C(destroy) = 5$. The *move_to* action can be applied from a block b_{source} only to an adjacent block b_{dest}. The cost of $move_to(b_{source}, b_{dest})$ is 1 if the path is free, 2 if it is blocked by an obstacle that can be jumped, and $+inf$ otherwise. To avoid computational and memory issues, subsequent states are indexed in memory by only the element that changed between them -that can only be a change in position, or a block that has been destroyed. The sub-tree for travelling can be summed up (for compactness) as in Figure 1. The sub-tree is re-executed continuously, re-planning for every action executed, for robustness. Although the algorithm is able to find paths to the desired destination, the control is not robust enough, and the agent often gets stuck. For this reason, we explore ways of exploiting Reinforcement Learning to improve on the hardcoded Behavior Tree policy.

The observations from the Minecraft platform are encoded to be read by the Neural Networks of the RL policies as a vector of numerical values. The first components of this vector are positional information (such as position w.r.t. the world origin, orientation), and the 3-dimensional cube is vectorized as a sequence $\{0,1\}^{15\text{i}5\text{i}5}$ where 0 means empty block and 1 means full block. No distinction is made on the type of blocks. The k closest items are represent by their coordinate only.

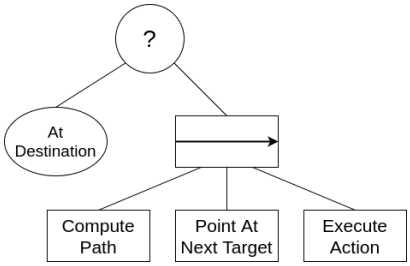

Fig. 1. The sub-tree for travelling from a source block to destination.

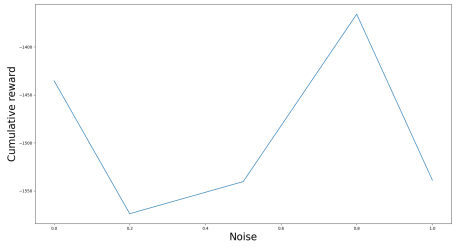

Fig. 2. Evaluation scores over 100 episodes of TD3 training from transitions collected by the hardcoded policy, for different values of the additional noise σ. Each value corresponds to the average of 5 training runs.

IV. Results and Discussion

A. Pre-training the Actor Policy

We trained the TD3 algorithm in the Pendulum environment from experience collected by the hardcoded policy only. First, we collect a buffer of $N = 10000$ transitions obtained by taking steps with the hardcoded policy in the environment adding Gaussian noise with variance σ^2 to the actions. Then we perform the TD3 training for 20×10^3 steps and we evaluate the algorithm averaging over 100 episodes. Figure 2 shows the results for different values of the additional noise. The training results are poor, scoring less rewards than the hardcoded policy, despite being trained on a number of transitions that in standard TD3 training leads to an optimal policy. Diversifying the dataset of trajectories by adding noise to the hardcoded actions does not help. Despite learning from a large and diverse dataset, TD3 is not able to learn a policy by being completely off-policy. This results contributes to the points made in [7] adding experiments with an algorithm that authors did not test.

We then pre-train the actor by minimizing the actor and critic loss with 8 and 9 and start a standard TD3 training from the pre-trained model. The TD3 training shows a huge drop in performances as soon as the model starts to train. Since the actor initially performs as good as the hardcoded policy, the drop must be due to the critic not learning the real value of actions not present or present according to a wrong distribution in the dataset. For this reason, we freeze the actor network when performing the TD3 training for 5000 steps, allowing the critic to improve without destroying the policy. Figure 3 shows the results. While freezing the actor prevents it from dropping, it also prevents it to improve, giving no benefit with respect to the standard training. We argue that the difficulties in learning good values for the critic from the backbone policy are due to the extrapolation and distributional shift issues shown by [7]. These issues are due to the hard-coded policy inducing a biased distribution of state-action pair in the dataset, that negatively affects learning with function approximation.

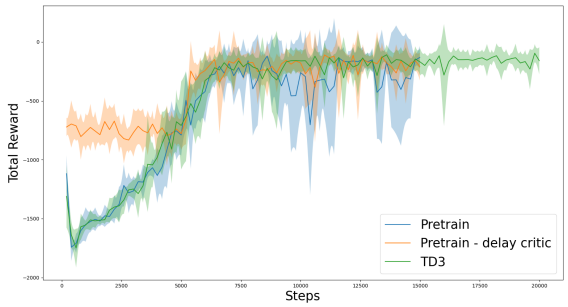

Fig. 3. Standard TD3 training (green line), TD3 from pretrained actor-critic (blue line) and TD3 from pretrained actor-critic with actor frozen for the first 5000 steps. Each line is the average of 10 runs.

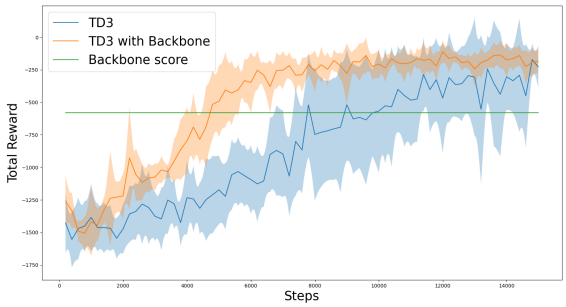

Fig. 4. Mean and standard deviation of cumulative reward for TD3 with and without the hard-coded policy as backbone over 10 runs. Green line shows rewards of the hard-coded policy alone.

B. Hardcoded Policy as a Backbone

We experiment the backbone training described in section III-B in the continuous action and state space setting of the Pendulum Gym environment. Here, the goal is to balance a pendulum upwards. A negative reward is given proportional to the angle (vertical is zero) plus a quadratic negative term depending to the force applied. We develop a simple hardcoded policy that pushes the pendulum in the direction of the angular velocity $\dot{\theta}$ when below pivot, and opposite to $\dot{\theta}$ when above the pivot. Such a policy obtains on average a total reward of -580. For comparison, a random policy obtains on average -1200. We train a TD3 policy in the backbone configuration using this policy. Figure 4 shows a comparison (averaged over 20 runs) of standard TD3 and TD3 using the hardcoded policy as backbone.

The backbone policy performs better than the standard TD3, and the average of 10 runs shows lower variance. It does not, however, reach higher rewards, that are clearly possible since they are sporadically reached by the standard TD3 policy. This makes us wonder whether the Backbone configuration results in a correction policy that is too tied to the backbone and is therefore unable to improve enough. Additional exploration techniques may be employed, such as an Ornstein-Uhlenbeck process [5], to make the RL policy explore better the action space around the backbone policy. While doing so, it is important that the exploration process keeps a zero expected value, otherwise the training may not be biased by the backbone policy anymore.

C. Hierarchical Training

We experiment the Hierarchical Training discussed in section II-C on the Pendulum environment using the hardcoded policy described below. The policy-over-options is a neural network with 3 inputs (the state) and 2 outputs, corresponding to the hardcoded and RL policy. We performed experiments for different time scales of the policy-over-options, number of training steps and alternating the training between the policy-over-options and subpolicies. However, in all the experiments the RL policy failed to learn, with the hardcoded policy

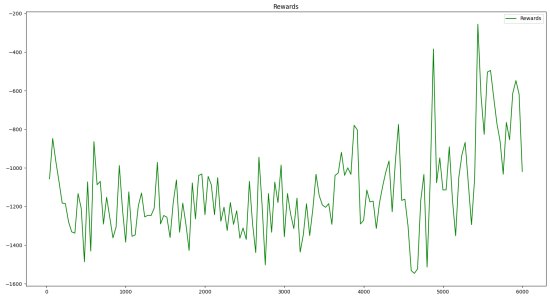

Fig. 5. Rewards for an example run of the Hierarchical Training.

taking over more as the ϵ parameter of the ϵ-greedy policy-over-option decreased. Figures 5 and 6 show and example of this behavior. Note that although the rewards in Figure 5 seem to be increasing in the end, they are actually just reaching the average of the hardcoded policy, since it becomes the only one used. Several trials for longer training and different configurations and values of hyperparameters showed no improvement over the hardcoded policy.

We argue that one of the reasons this happen may be the highly non-stationarity of the problem. In fact, the policy-over-options has to learn off-policy from a sub-policy that constantly changes. On the other hand, the RL policy needs to learn from a target value that constantly changes. This is due to the fact that the expectation $V(s)$ is taken w.r.t. the distribution of the following actions, which depend on the ever changing policy-over-options. It is also not entirely clear -and may need further research- how to correctly perform off policy multi-step learning in the TD3 algorithm, as the target policy does not represent a distribution since it is deterministic.

D. Minecraft Agent

We experienced several issues with trying Reinforcement Learning on the Malmo platform. The environment is in fact asynchronous and while the execution of the Behavior Tree runs without problems, the slower and heavier execution of the RL training makes it skipping too many useful frames.

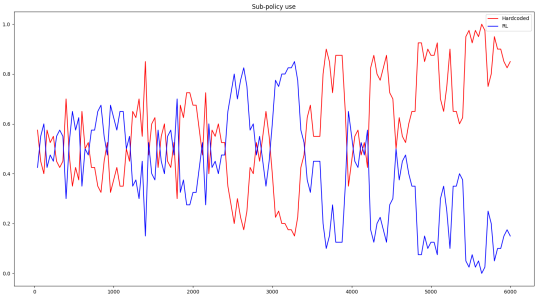

Fig. 6. Use of hardcoded (red) and RL (blue) subpolicies over time.

Despite our best effort, we have not been able to make the platform running synchronously. Moreover, the bad results of the Hierarchical Training -our first and preferred choice- did not allowed us to have a reliable way of performing the training. Indeed, further research could be done by trying the Backbone Training on the Minecraft environment. On-policy methods -such as the Option Critic architecture[2]- may also be tried, as they may provide a more stable training.

V. Conclusion

In this work, we explore several approaches to bridge the gap between hard-coded policies, that are widely used to design complex behaviors, and off-policy Reinforcement Learning. We try to pre-train a TD3 agent and we propose explanations of why it did not work. We then propose a RL architecture that uses the hard-coded policy as a a backbone, biasing the RL policy towards better trajectories and making it improve faster. Finally, we experiment whether the Hierarchical RL framework of Options could be beneficial to such a task. We propose an hierarchical architecture that uses the hard-coded and TD3 policies as sub-policies. To the best of our knowledge, no other work tried to put together multi-step off policy TD3 as a RL Option. The results are below expectations, confirming the difficulty of learning options off-policy.

References

[1] URL: https://py-trees.readthedocs.io/en/devel/faq.html.

[2] Pierre-Luc Bacon, Jean Harb, and Doina Precup. *The Option-Critic Architecture*. 2016. arXiv: 1609.05140 [cs.AI].

[3] Andrew G. Barto and Sridhar Mahadevan. "Recent Advances in Hierarchical Reinforcement Learning". In: *Discrete Event Dynamic Systems* 13.1–2 (Jan. 2003), pp. 41–77. ISSN: 0924-6703. DOI: 10.1023/A:1022140919877. URL: https://doi.org/10.1023/A:1022140919877.

[4] Michele Colledanchise and Petter Ögren. "Behavior Trees in Robotics and AI". In: (July 2018). DOI: 10.1201/9780429489105. URL: http://dx.doi.org/10.1201/9780429489105.

[5] Steven Finch. *Ornstein-Uhlenbeck Process*. 2004.

[6] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. arXiv: 1802.09477 [cs.AI].

[7] Scott Fujimoto, David Meger, and Doina Precup. *Off-Policy Deep Reinforcement Learning without Exploration*. 2019. arXiv: 1812.02900 [cs.LG].

[8] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/tssc.1968.300136. URL: https://doi.org/10.1109/tssc.1968.300136.

[9] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[10] Matteo Iovino et al. *A Survey of Behavior Trees in Robotics and AI*. 2020. arXiv: 2005.05842 [cs.RO].

[11] Damian Isla. "Halo 3-building a Better Battle". In: *In Game Developers Conference*. 2008.

[12] Matthew Johnson et al. "The Malmo Platform for Artificial Intelligence Experimentation". In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI'16. New York, New York, USA: AAAI Press, 2016, pp. 4246–4247. ISBN: 9781577357704.

[13] Timothy Mann, Daniel Mankowitz, and Shie Mannor. "Time-Regularized Interrupting Options (TRIO)". In: ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, 22–24 Jun 2014, pp. 1350–1358. URL: http://proceedings.mlr.press/v32/mannb14.html.

[14] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[15] Rémi Munos et al. *Safe and Efficient Off-Policy Reinforcement Learning*. 2016. arXiv: 1606.02647 [cs.LG].

[16] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG].

[17] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[18] Richard Sutton et al. "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Adv. Neural Inf. Process. Syst* 12 (Feb. 2000).

[19] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[20] Richard S. Sutton, Doina Precup, and Satinder Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial Intelligence* 112.1 (1999), pp. 181–211. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(99)00052-1. URL: http://www.sciencedirect.com/science/article/pii/S0004370299000521.

[21] Kazuyoshi Wakuta. “The Bellman’s principle of optimality in the discounted dynamic programming”. In: *Journal of Mathematical Analysis and Applications* 125.1 (1987), pp. 213–217. ISSN: 0022-247X. DOI: https://doi.org/10.1016/0022-247X(87)90176-4. URL: http://www.sciencedirect.com/science/article/pii/0022247X87901764.

APPENDIX A
ANSWER TO PEER REVIEWS

Peer Review text is shown in *Italic*, comments addressing it are shown in **bold**.

A. Peer Review 1

This paper generally presents behavior trees, reinforcement learning, and their applications. It also describes different RL algorithms and how they are applied in this research. As the abstract and introduction are only outlined, this section will not be evaluated. That said, the presented plan for the introduction looks like it will cover all the important topics introducing the research. The background section is very detailed, including lots of references to related work and relevant concepts. The reinforcement learning background information is very well done, including many relevant formulas and concepts. That said, it could maybe be shortened slightly, describing only the important details and explaining how it was applied in this specific research in the methods section. If this information is to stay in the background section, it would be nice to keep it in context of the research that was conducted.

I reduced the Background section to be shorter and more compact, avoiding definitions that go too into the details of RL. However, all the theory explained is relevant to the research. The length of the section is mainly due to the fact that in my work I employed two different RL algorithms, and an additional Hierarchical RL framework. I also believe that the definitions that I give in this revised paper are all relevant, as they provide grounds for some algorithmic choices that I made. However, the section should be now looking better without loss of readability.

The author explains the difficulty and importance of the task in the methods section, giving useful context to the reader. I assume more of this analysis will be included in the introduction and results sections. The author has also included discussion around why certain methods were chosen or deemed inadequate. There is always justification and background for each method used. While the results section is yet to come, there should also be descriptions of any experiments performed to show how the methods were applied to a specific problem.

I have now added an Introduction section, explaining why the task is important and what is the scope of the research

Without the introduction, it is difficult to determine the application of these methods—from what I remember, it is related to Minecraft. This may no longer be the case, but I think additional context details throughout the paper related to the applications would be useful. There was clearly a lot of research conducted into behavior trees and reinforcement learning which makes the paper very informative and descriptive. Clearly much care was put into the notation and method details. Overall, the parts of this paper that were presented were detailed and informative, but it is difficult to give a wholistic evaluation without the results and analysis. The best aspects of the paper are the background information details, formulations of the methods, and overall organization and language. That said, depending on the results and analysis, maybe the background section could be more concise. The area for the most improvement is adding more visuals and context about the application or experiments conducted to provide additional context to the results that are presented.

I now added, among all those performed, the results that I believe are most meaningful.

The project seems very interesting and I am excited to see the final results. Great job! To summarize, these are my suggestions for the final paper:

- Make background sections more concise if possible (**addressed**)
- Keep in mind the application of the methods to add context (**addressed**)
- Add details about the experiments conducted and the application (there is no mention of Minecraft currently which I believe the project to be about). This will be important for reproducibility if experiments were conducted. (**addressed**)
- Add visuals to illustrate the ideas presented (**addressed**)

B. Peer Review 2

The report introduces certain concepts related to reinforcement learning and behaviour trees. The author further uses a few algorithms for training the reinforcement learning policies and describes their use-case scenarios. Although, I believe, there is a lack of proper problem formulation and there could have been a better motivation for the work done.

I have now added both an abstract and an Introduction section in which I better state the problem, and what is the main purpose of this research. I also tried to highlight this more in the Methods section at the beginning of each sub-section

I appreciate the theoretical introduction to the subject given by the author. The background section is clear and I believe it is relevantly backed by the equations. Since there are relevant equations to the problem, addition of a problem formulation section might be beneficial and more explanatory. The background section is followed by the methods section, wherein the author introduces different training algorithms used. I believe the author introduces the relevant algorithm nicely and addition of a comparison table might be of help in terms of comparing amongst them. The author introduces relevant equations as well which were quite understandable and followed the flow of the text. The structure of the text was easy to follow, with defined sections and structured flow of

text. I could think of a few grammatical errors but that didn't make following the text uneasy in any way.

I have fixed many grammar errors, and tried to reword some sentences. I have also generally shortened the paper, trying to state concepts more clearly.

Conclusively, congratulations for the project and good work!