Hierarchical Reinforcement Learning With Monte Carlo Tree Search in Computer Fighting Game

Ivan Pereira Pinto and Luciano Reis Coutinho

***Abstract*—Fighting games are complex environments where challenging action-selection problems arise, mainly due to a diversity of opponents and possible actions. In this paper, we present the design and evaluation of a fighting player on top of the FightingICE platform that is used in the Fighting Game Artificial Intelligence (FTGAI) competition. Our proposal is based on hierarchical reinforcement learning (HRL) in combination with Monte Carlo tree search (MCTS) designed as options. By using the FightingICE framework, we evaluate our player against state-of-the-art FTGAIs. We train our player against the current FTGAI champion (GigaThunder). The resulting learned policy is comparable with the champion in direct confront in regard to the number of victories, with the advantage of having less need for expert knowledge. We also evaluate the proposed player against the runners-up and show that adaptation to the strategies of each opponent is necessary for building stronger fighting players.**

***Index Terms*—Fighting game, hierarchical reinforcement learning (HRL), Monte Carlo tree search (MCTS), Option framework.**

I. Introduction

The fighting genre has a significant presence in the gaming community giving rise to amateur and professional tournaments and being characterized as an e-sport. In contrast to other genres (e-sports) such as the multiplayer online battle arena (MOBA), exemplified by games like *Defense of the Ancients (DOTA)* and *League of Legends (LoL)*, the fighting genre spans diverse titles, some that last many years and others that vanish quickly. The most successful and played fighting games are present on the Evolution Championship (EVO) that is held annually, attracting pro-gamers and amateurs.

A typical fighting game is a two-player adversarial game, with the exception of a few tag games where four players or more can play on teams. The game has a time limit, and each player controls a character having an associated life bar. If this life bar depletes completely, the corresponding player loses the game. If the time limit runs out, the winner is the player whose character has more life left. Actions such as punch and kick can inflict damage, and most fighting games allow these actions to be performed in combinations or combos, inflicting extra damage.

Scientific interest in fighting games comes primarily from the artificial intelligence (AI) community, mostly regarding the development of strong fighting players. For the evaluation of fighting players, there is an annual Fighting Game AI (FTGAI) competition [1], organized by ICE Lab[1] and present at the Computational Intelligence and Games (CIG) conference since 2014. In the 2017 edition, there has been a shift from rule-based players to Monte Carlo tree search (MCTS) ones. The champion and runners-up all have used a hybrid of MCTS and rules based on detailed expert knowledge.

We list the main contributions of this paper as the following.

1) We present the design and evaluation of a fighting player for the FTGAI competition developed by means of hierarchical reinforcement learning (HRL).
2) Our proposed method uses the Option framework [2] with function approximation for high-level control policies, and MCTS [3] for inner working of these policies. While works that use MCTS planning over set of Options exists [4], a few or none have explored the use of MCTS searches as options.
3) We evaluate our approach against the rule-based MCTS methods that have dominated the 2017 FTGAI competition.

Our original fighting player, called Mogakumono, which participated in the same competition, has been modified and overall improved in this work, and we show its competitiveness against state-of-the-art fighting players.

This paper is organized as follows. Section II briefly reviews related works. Section III details the core elements on which our proposal is built: the fighting game platform, the theoretical background of RL, the notions of HRL, and the MCTS method. The design of the fighting player is presented in Section IV, and experiments regarding training and evaluation are shown in Section V. Section VI presents conclusion and future work.

II. Related Works

RL has been applied with relative success to fighting games. In [5], tabular Q-learning is used to developed a game called *Knock'em*, which can change its behavior by choosing among three levels (easy, medium, and hard) in accord to life difference from the opponent. The result, according to the authors, is comparable to the traditional Q-learning and the rule-based and naive random approaches.

Closer to our work, we highlight [6] and [7]. In [6], the authors present a real-time dynamic scripting AI that won the 2015 FTGAI competition. It uses RL as the learning procedure for selecting the order in which the rules will appear in the script. The second work [7] is based on the UCB [8] algorithm for selecting and switching rule-based controllers at given time intervals. Our work differs from both in that it does not use rules for selecting behaviors, and has a more fine-grained control for switching between behaviors at each step.

More recently, in [9], there has been an initial effort to apply deep RL to the ICE Lab fighting game. While the trained agent learned to play against a stationary opponent, it was not competitive enough for adversarial gameplay. The lack of massive data and available time may be a factor in this result.

Many other techniques have been explored for fighting games, notably the use of evolutionary algorithms for modeling sequence of actions as genomes and finding the best combo through genetic optimization [10], and the use of k-nearest neighbors (KNN) for prediction of the opponent's actions [11].

Manuscript received January 8, 2018; revised March 19, 2018; accepted May 30, 2018. Date of publication June 11, 2018; date of current version September 13, 2019. *(Corresponding author: Ivan Pereira Pinto.)*

The authors are with the Federal University of Maranhão, São Luís, Maranhão 65080-805, Brazil (e-mail: navi1921@gmail.com; luciano.rc@ufma.br).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TG.2018.2846028

[1] http://www.ice.ci.ritsumei.ac.jp/

III. Background

A. FightingICE

FightingICE [1] is a platform for research in AI based on fighting games. While many studies in the fighting game genre involve the joint development of the game engine along with the technique researched, FightingICE offers an engine with read access to internal state variables. Furthermore, it is possible to use a simulation module, which is fast enough for real-time gameplay, allowing techniques that make use of forward search to be employed. Source code from previous FTGAI competitions is also available, so that different techniques can be compared.

The game has three playable characters: Zen, Garnet, and Lud. Inputs can be four directional keys (left, right, up, and down) and two attack keys (A and B). Composition of multiple keys can form new actions, such as down and B being Crouch B. FightingICE has all possible combinations of these keys available in the form of 56 actions, which can be seen as atomic (indivisible). Some of these actions can only be executed when the player is above the ground, and others only when touching the ground, which we call *Air* and *Ground* actions, respectively. In this work, we consider these 56 actions as primary actions.

Last, an important aspect that sets apart the FightingICE platform from other fighting game emulators is that it enforces 250.05 ms of delay on state variables. This is set on purpose, to force fighting players in not reacting on superhuman reflexes once the opponent begins its attack motions, but to make predictions, on a more human fashion.

B. Reinforcement Learning

RL is one of the approaches of machine learning for problem solving. An agent learns by interacting with the environment, where it has to decide which actions to take in the current state. More formally, the agent tries to solve a Markov decision process (MDP), formalized as M = (S, A, P, r, γ), where:

1) S = $\{s_1, ..., s_n\}$ is a discrete set of states;
2) A = $\{a_1, ..., a_m\}$ is a set of actions;
3) P(s, a, s') is the probability of reaching the next state s' from the current state s and selected action a;
4) r(s) is a reward function for state s;
5) γ is the discount factor for adjusting the importance given for immediate rewards.

Other important notions to define here are as follows. The agent has a policy π, which is a mapping from states to actions, defining the action to be taken. The total accumulated reward in an episode is called *return* R. The main objective is then to learn an optimal policy that maximizes the expected return.

One of the most popular approaches for solving MDPs in RL are called temporal-difference (TD) methods. Some of their principal characteristics are being model free (i.e., do not need to learn the transition model), needing only the environment model for sampling experience, and being able to adjust itself during the episode. Among TD methods, Q-learning stands out as an off-policy method, meaning that it learns an optimal policy without using the current learned policy. The update rule is

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a' \in A} Q(s',a')] \quad (1)$$

where $Q(s,a)$ is the action-value function that gives the expected return for taking action a from state s, $\max_{a' \in A} Q(s',a')$ is the best next estimate between all possible actions from state s', and $0 \leq \alpha \leq 1$ is a learning factor. This update formula is performed at each step in the environment by the learning agent, and basically sets the current estimate of the state-action function a little bit toward the difference between the old current estimate and the next.

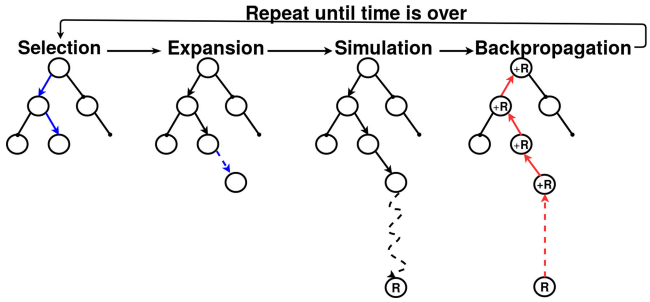

Fig. 1. Monte Carlo tree search.

C. Options

In many environments, it might be useful to make decisions based on more than one primitive action. Deciding between extended course of actions means that once the decision has been taken, all that the agent needs to do is to follow the chosen course of actions. The Option [2] framework allows one to represent those extended course of actions as policies, and learning over these high-level actions is often named HRL. Options are formalized as $o = (I, \mu, \beta)$, where I is the initiation set, a set of states from which the options can be executed; μ is a policy that defines what action is chosen under this option; and β is the stopping condition for the option.

The use of options turns the MPD formulation into a semi-Markov decision process (SMDP), which allows the modeling of extended actions, updating the Q-learning rule to

$$Q(s,o) = (1-\alpha)Q(s,o) + \alpha[r + \gamma \max_{o' \in O} Q(s',o')] \quad (2)$$

where O is the set of all options. The HRL agent then tries to learn a policy over options just as if they were single actions.

D. Monte Carlo Tree Search

There are other approaches that have been used in games such as planning algorithms that search the space state directly for solutions instead of predicting from stored knowledge. A somewhat hybrid method related to both RL and planning is MCTS [3], a forward planner that constructs a search tree by means of Monte Carlo simulations, learning estimates of actions nodes in the process. MCTS has been implemented on FightingICE [12]. The algorithm builds a search tree iterating through four steps until the time budget is over, as shown in Fig. 1. The general inner-working is roughly as follows.

1) Selection: The algorithm descends the search tree that is already in memory, in accord to the upper confidence bounds (UCB1) formula [8]

$$\text{UCB1} = Q(s,a) + C\sqrt{\frac{2\log N(s)}{N(s,a)}} \quad (3)$$

where $Q(s,a)$ is the expected return of the action-value that corresponds to the current node, C is a constant for tuning, and $N(s,a)$ and $N(s)$ are counters for the number of visits on this node and its parent, respectively. The algorithm selects greedily our player node-actions with highest $Q(s,a)$.
The opponent nodes are randomly selected, as to make up for not knowing how it acts. The stop condition is when it finds a node without all children explored, or it reaches a leaf node.

2) Expansion: The algorithm adds a node to the tree where the selection has stopped.
3) Simulation: The algorithm plays randomly until either the end of the episode has been reached, or the simulation time budget is over.
4) Backpropagation: The algorithm returns the reward to all visited nodes and increments the visited nodes counters. Reward is computed as the difference between damage inflicted and damage received.

Once the search is over, there is a problem to elect one of the children from the root node. The basic choices are the most visited child node, or the one with greater expected return. We choose the second one as the elect method used in this paper.

We remark that there is a constraint limiting an accurate MCTS: Fighting games need that decisions be taken in less than 1 s, so the simulation depth and the overall time budget have to be reasonably administered.

IV. Methodology

In this section, we describe the development process of the HRL agent.

A. Function Approximation

The implementation of $Q(s, a)$ or $Q(s, o)$ in practice requires a table having entries for all possible states, multiplied by the number of actions/options. While this number can be manageable in small grid worlds, it is infeasible in real and more complex applications, be it for the necessary space or for the time required to access such a table. Function approximation (FA) methods [13] try to deal with this problem by generalizing new states to previously encountered ones, learning an approximate *Q* function.

We choose to use linear FA with binary features, as it has the advantage of being simple, understandable, and having success cases in many games [14], [15]. The option value function is estimated by a linear combination of the binary features ϕ and weights θ as follows:

$$Q_\pi(s, o) = \phi(s, o)\theta^T \tag{4}$$

$$\theta = \theta + \delta\phi(s, o) \tag{5}$$

with δ being the update rule described in (2). The features have to be manually designed, so domain knowledge is needed, but we keep as simple as possible. Our features for the players (ours and the opponent) are as follows.

1) Five features for the life difference between our player to the opponent: difference ≤ 10, $10 <$ difference ≤ 50, $50 <$ difference ≤ 100, $100 <$ difference ≤ 200, difference > 200.
2) Ten features for both players energy: energy ≤ 3, $3 <$ energy ≤ 10, $10 <$ energy ≤ 50, $50 <$ energy ≤ 100, energy > 100.
3) Sixteen features for 2-D positions: The *x* space is divided in five equal intervals, and the *y* space in three equal intervals. It is checked for both players if they are in these intervals.
4) Eight features for the 2-D movement, if the players are going right, left, up, or down;
5) Six features for players distance to projectiles: near (distance ≤ 100), intermediary ($100 <$ distance ≤ 300), and far (distance > 300).

B. Developing Options

In this section, we propose the creation of options policies for the HRL agent. Instead of hard-coding the policy through rules or learning them from scratch, we infer policies from different subsets of actions that MCTS searches from.

We redefine *A* as a set composed of both player's and opponent's actions, totaling 112 actions. We construct $A_1, \ldots, A_k$ subsets of A that are nonidentical and may have intersecting actions, and each option o has a corresponding subset A_o, having a total of k subsets and options. The MCTS uses the selected option chosen at runtime by the agent to build the tree from both its player and opponent actions. Actions from the opponent are required for the MCTS to simulate their possible behavior.

There are two advantages in this approach with respect to the standard MCTS: with less actions, there is more time for building the search tree, giving more precise results, and they can be easily tested to see if they match the desired policy behavior.

The process of creating options needs to be carefully designed to be effective. Our reasoning is that options representing distinct policy behaviors in the target game are more useful, and many of these behaviors do not have to consider all possible actions. In the case of fighting games, it is basic knowledge for players that they need to have strategies that involve some specific actions based on the situation. As an example of that, one would only choose between long-ranged actions and movement actions when the opponent is far way, and never consider short-ranged actions that need the opponent to be close by to deal with the damage. Options for others games can be defined by the same pattern, where one has to group only the actions that are useful for some distinct behavior.

So, for the FightingIce we choose to create options composed of actions we consider essential for some behavior, and those behaviors are not manifested enough in the full MCTS. All options have the predefined structure $\langle I, \beta, \mu \rangle$ that will be detailed later. They are summarized (without any priority) as follows.

1) Ground risky attacks: Ground actions from the player, but none from the opponent. They are risky by not accounting for opponent counter attacks.
2) Long ranged attacks: Ground actions that have the most range from both player and opponent. They are kicks, normally.
3) Aerial risky attacks: All Air actions from the player and none from the opponent.
4) Falling attacks: Air actions that make the player fall quickly while attacking.
5) Antiaerial attacks: Ground actions from the player only, with a high-range overhead, generally effective when opponent is on the air.
6) Guard: Defending actions, only from the player.
7) MCTS: Original search that considers all Air and Ground actions from both players.

C. Proposed Method Overview

Here, we give an overall explanation of the method used for our HRL agents in the experiments later on.

For the Option framework, there is the need to specify the initiation set *I*, and the termination condition β for each option, a problem hard to be tackled without having expert knowledge about the game and the options themselves, so that they are active only when useful. Our approach is to minimize this knowledge, by setting *I* as the set of all states *S*, and the termination condition to one step after choosing the option.

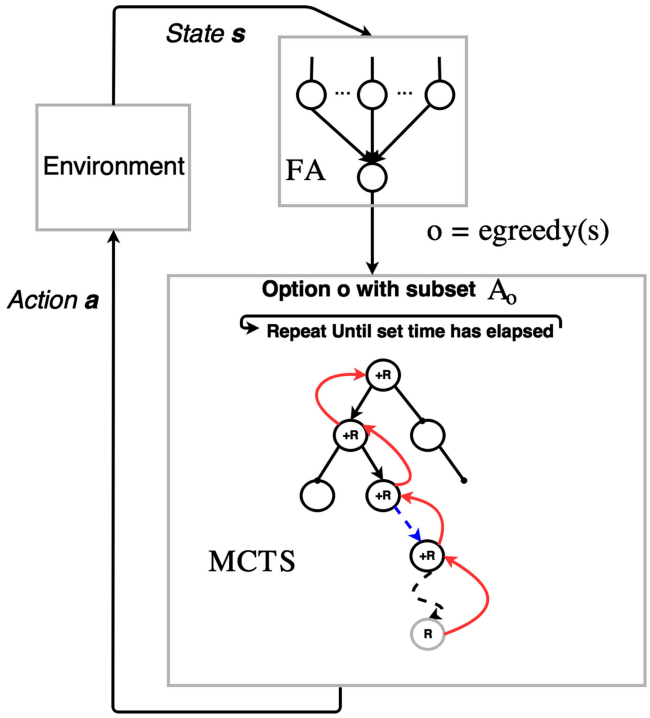

Fig. 2. Proposed method.

We argue that these choices have the following advantage. By making the agent decide its options at each step, from any state, we are essentially inducing it to learn when to use and interrupt the options. If, for example, one option is proven to be better than another at some states, the agent will just continue to follow it, and only stop when it ceases to be. The other advantage is the possibility to mix options, so the agent can learn to switch the option that is not suitable at some particular state to some other better suited. In this way, a policy where options cover each other flaws is possible.

As mentioned earlier, a linear regression model has been chosen for approximating the option-value function, with the weights being trained by Q-learning. The chosen selection policy is *e-greedy*, where with probability e, a random option will be selected, and with $1 - e$, the option of the largest Q value is selected.

Fig. 2 illustrates the proposed method. The FA model receives the state from the environment, transforms it into features, computes $Q(s, o)$ values, and chooses an option according to the *e-greedy* policy. Each option corresponds to a different set of actions A_o upon which MCTS will be executed. The MCTS runs as it would normally, but only having the actions from the selected option to construct its search tree from, and then returns an action a that is executed on the environment. We note that all this happens at each step, since the options are immediately interrupted.

V. Experimental Setup

In this section, we detail the training and experiments performed to evaluate the HRL agent.

A. Training Setup

Regarding Q-learning parameters, after some experimentation, we have chosen the following: the discount factor γ is equal to 0.97, and the learning rate set to the formula $\frac{\alpha}{\text{number of active features}}$ where α is a constant of 0.01, so that the changes when there are less features are more significant. We set *e-greedy* to a decaying rate from 0.99 (totally random) to 0.01. Regarding MCTS, the C parameter was set to 2.

TABLE I
Chosen Options Frequency Rate

Options	*1*	*2*	*3*	*4*	*5*	*6*	*7*
ZEN	4.71%	41.26%	20.26%	3.76%	5.55%	20.38%	4.04%
GARNET	1.98%	44.75%	8.56%	7.03%	2.54%	7.59%	27.51%
LUD	7.67%	62.83%	12.02%	4.59%	2.00%	4.69%	5.96%

B. Training Evaluation

We have trained the HRL player against the winner of the 2016 and 2017 edition of the FTGAI competition, named GigaThunder. This winner is a hybrid of rule- and MCTS-based approaches, using rules for specific situations where the search is not efficient, and narrowing the search with subsets of actions in a way similar to our approach, but having rules to decide when to do so.

We set 2000 training episodes, where each episode is a round of the fighting game. The players are trained only against the same character type (i.e., Zen, Garnet, and Lud). Besides the HRL agent, we train a pure RL agent for comparison, with the same features and Q-learning parameters presented before, and the 56 default actions. Fig. 3 shows the training statistics, presenting the accumulated reward over the episodes smoothed on a ten-episode window. Positive reward means that the player inflicted more damage than it has received, and a negative means the opposite.

In the beginning, the accumulated reward is negative. This shows that randomly choosing options is not sufficient to win. We note that the reward mean rate increases for the three characters, showing that the learning process is indeed happening. HRL Zen character converges to 0 accumulated reward, which means that it is mostly tying in terms of damage given and received. HRL Garnet surpasses GigaThunder Garnet, and has a faster convergence, keeping a lead after episode 50. HRL Lud begins from a reward mean lower than the other two characters, and gets closer to the 0 baseline. All traditional RL characters obtain very low accumulated reward, showing that a pure Q-learner agent fails to be competitive against MCTS-based ones. The HRL characters, however, learned policies on par with current FTGAI champion.

Another important statistic is the option selection frequency. We play again 1000 rounds against GigaThunder, and record the options selected for the three characters. The resulting frequency is normalized to the 0–1 range, as shown in Table I. Over all options, we see a preference for option 2 (falling attacks). It is interesting that option 7 (MCTS) is not the most used, since it considers all actions. We can infer that the resulting trained characters believe that the designed options will give them a better expected return in many situations.

C. Tournament Evaluation

While the training rate against the champion player is a good metric to evaluate our player, it is important to compare it to other players because of the possibility of over-fitting. In this way, we have also compared the three HRL characters against the runners-up of the 2017 FTGAI tournament, namely FooAI, JayBot, and Mutagen.

All three runners-up use MCTS with some sort of modifications. JayBot [16] uses action tables that collect most frequently used actions from many AI methods for each metric rule, such as distance from the opponent. Then, it uses the top frequent actions to construct the set of actions MCTS will be searching on. Mutagen similarly uses rules to

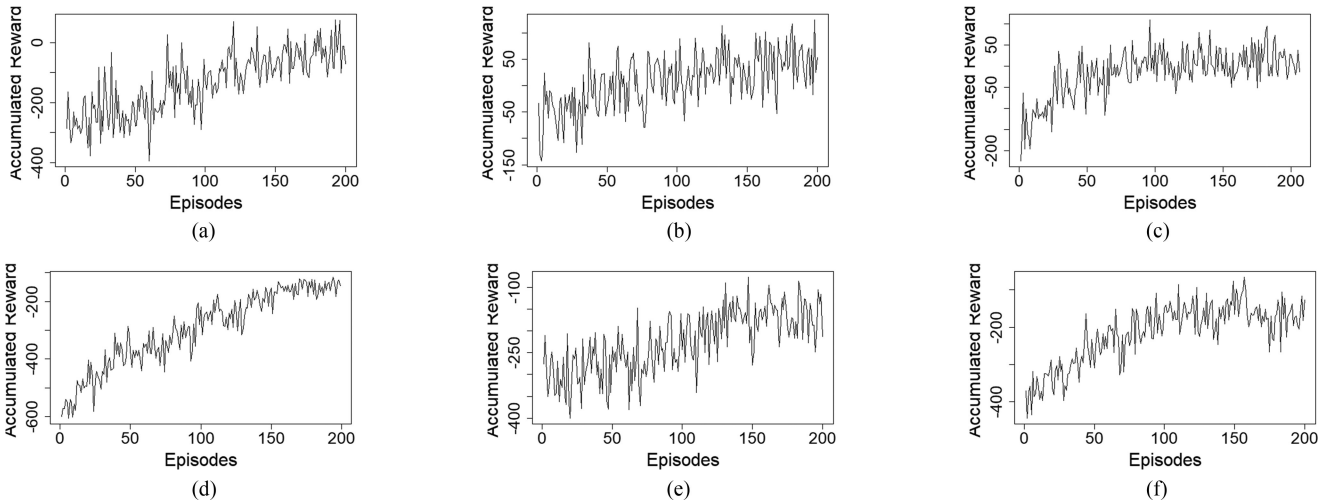

Fig. 3. Training rates of HRL and RL characters against GigaThunder characters. (a) HRL Zen × GigaThunder. (b) HRL Garnet × GigaThunder. (c) HRL Lud × GigaThunder. (d) RL Zen × GigaThunder. (e) RL Garnet × GigaThunder. (f) RL Lud × GigaThunder.

TABLE II
PERCENTAGE OF WINS AGAINST TOP 4

HRL Player	*GigaThunder*	*FooAI*	*JayBot*	*Mutagen*
ZEN	48.42%	66.45%	62.37%	34.15%
GARNET	66.76%	28.84%	34.49%	29.28%
LUD	51.53%	41.04%	57.81%	37.22%

TABLE III
PERCENTAGE OF WINS AGAINST RUNNERS-UP

HRL Player	*FooAI*	*JayBot*	*Mutagen*
ZEN	65.15%	71.75%	52.20%
GARNET	57.81%	50.40%	44.13%
LUD	61.96%	63.29%	48.04%

decide which sets of actions to search with MCTS for a given distance. Furthermore, it uses rules for specific situations such as when to guard. The resulting player had the best score when confronting the other top players. FooAI too uses rules to improve on weak points of the default MCTS. We note that while all of them are MCTS-based players, they have different resulting behaviors.

To evaluate our agent, we choose to perform direct confronts, which is the FTGAI tournament approach. There they evaluate 4–6 rounds in the competition called Normal League. We however evaluate 2000 rounds, with a result of a better estimate of the players strength. Table II shows the comparison results, in terms of winning percentage. For this experiment, we have disabled the learning process and configured the HRL agent to select the best options previously learned. Similarly to the training rates of Fig. 3, the winning rates against GigaThunder are almost tied, with HRL Garnet having advantage. Regarding the runners-up, as suspected, there have been considerable defeats against them. HRL characters, except Zen, have not been successful against FooAI characters. Against Mutagen characters, all HRL characters obtained the worst rates. Finally, against JayBot, Zen, and Lud characters have won consistently, but Garnet have obtained lower scores.

These results indicate that the HRL agent trained only with GigaThunder does not generalize well against the runners-up. The MDP (and by extension the SMDP) model assumes a stationary environment, so the agent effectiveness is not guaranteed when there is a possible changing policy from the opponent composing this environment. For good performance, the agent either has to somehow learn a strong policy against any opponent, or learn to adapt against them, as noted in [16].

After these, we have rerun the experiment for 1000 rounds, but we now enabled the HRL characters to update their pretrained linear function weights. The results are shown in Table III. In virtually all confronts, there has been gain in the percentage of victories. From this experiment, we can infer that, by allowing retraining of weights pretrained from GigaThunder, the HRL characters can adapt and generalize better against the opponents in up to 1000 rounds.

Finally, these results indicate that, given enough time to adapt, the proposed technique can be applied successfully in fighting games, with its performance at least on par with state-of-the-art players.

VI. CONCLUSION AND FUTURE WORK

In this work, we have presented a fighting player developed by using HRL in combination with MCTS. Our approach does not use expert knowledge beyond the design of basic features and options, and is able to learn a policy that matches the current FTGAI champion. We have shown that the learned policy switches between options, instead of following only the best one. We have evaluated the approach against the runners-up of the FTGAI tournament, concluding that our agent did overfit the champion. A subsequent evaluation was performed allowing our player to change its learned policy showing that this leads to a better performance against the opponents.

Current limitations of our method include the cost of training thousands of matches in real time, and a number of retraining matches beyond what is available in the FTGAI tournament, showing a need of fast adaptation for competitive fighting AIs.

The proposed hierarchical reinforcement model allows for many possible improvements. In a future work, we plan to extend the model to other types of function approximators. Methods for automatic learning of options such as [17] can be explored. Better performance of options may be possible by using other variations of MCTS, or by combining them with heuristics functions.

REFERENCES

[1] F. Lu, K. Yamamoto, L. H. Nomura, S. Mizuno, Y. Lee, and R. Thawonmas, "Fighting game artificial intelligence competition platform," in *Proc. IEEE 2nd Global Conf. Consum. Electron.*, 2013, pp. 320–323.

[2] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artif. Intell.*, vol. 112, no. 1–2, pp. 181–211, 1999.

[3] C. B. Browne *et al.*, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[4] M. de Waard, D. M. Roijers, and S. C. Bakkes, "Monte Carlo tree search with options for general video game playing," in *Proc. IEEE Conf. Comput. Intell. Games*, 2016, pp. 1–8.

[5] G. Andrade, G. Ramalho, H. Santana, and V. Corruble, "Extending reinforcement learning to provide dynamic game balancing," in *Proc. Workshop Reason. Represent. Learn. Comput. Games, 19th Int. Joint Conf. Artif. Intell.*, 2005, pp. 7–12.

[6] K. Majchrzak, J. Quadflieg, and G. Rudolph, "Advanced dynamic scripting for fighting game AI," in *Proc. Int. Conf. Entertainment Comput.*, 2015, pp. 86–99.

[7] N. Sato, S. Temsiririrkkul, S. Sone, and K. Ikeda, "Adaptive fighting game computer player by switching multiple rule-based controllers," in *Proc. IEEE 3rd Int. Conf. Appl. Comput. Inf. Technol./2nd Int. Conf. Comput. Sci. Intell.)*, 2015, pp. 52–59.

[8] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *Proc. 17th Eur. Conf. Mach. Learn.*, vol. 6. pp. 282–293.

[9] S. Yoon and K.-J. Kim, "Deep Q networks for visual fighting game AI," in *Proc. IEEE Comput. Intell. Games*, 2017, pp. 306–308.

[10] G. L. Zuin, Y. Macedo, L. Chaimowicz, and G. L. Pappa, "Discovering combos in fighting games with evolutionary algorithms," in *Proc. Genetic Evol. Comput. Conf.*, 2016, pp. 277–284.

[11] Y. Nakagawa, K. Yamamoto, C. C. Yin, T. Harada, and R. Thawonmas, "Predicting the opponent's action using the k-nearest neighbor algorithm and a substring tree structure," in *Proc. IEEE 4th Global Conf. Consum. Electron.*, 2015, pp. 533–534.

[12] S. Yoshida, M. Ishihara, T. Miyazaki, Y. Nakagawa, T. Harada, and R. Thawonmas, "Application of Monte-Carlo tree search in a fighting game AI," in *Proc. IEEE 5th Global Conf. Consum. Electron.*, 2016, pp. 1–2.

[13] M. Geist and O. Pietquin, "A brief survey of parametric value function approximation," Tech. Rep., Supélec, Sep. 2010.

[14] M. Buro, J. R. Long, T. Furtak, and N. R. Sturtevant, "Improving state evaluation, inference, and search in trick-based card games," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, 2009, pp. 1407–1413.

[15] M. Buro, "Logistello: A strong learning Othello program," in *Proc. 19th Annu. Conf. Gesellschaft für Klassifikation eV*, 1995, vol. 2, pp. 1–3.

[16] M.-J. Kim and K.-J. Kim, "Opponent modeling based on action table for MCTS-based fighting game AI," in *Proc. IEEE Conf. Comput. Intell. Games*, 2017, pp. 178–180.

[17] M. Tamassia, F. Zambetta, W. Raffe, F. Mueller, and X. Li, "Learning options from demonstrations: A Pac-Man case study," *IEEE Trans. Comput. Intell. AI Games*, vol. 10, no. 1, pp. 91–96, Mar. 2018.