Game Adaptation by Using Deep Q-Learning Over Meta Games

Diogo Luís Cerqueira Carneiro da Silva
Master in Informatics and Computing Engineering
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
up201405742@fe.up.pt

João Pedro Teixeira Pereira de Sá
Master in Informatics and Computing Engineering
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
up201506252@fe.up.pt

José Manuel Faria Azevedo
Master in Informatics and Computing Engineering
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
up201506448@fe.up.pt

Luís Fernando Frutuoso Fernandes Mouta
Master in Informatics and Computing Engineering
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
up201808916@fe.up.pt

***Abstract*—Abstract**

***Index Terms*—Computer games, Dynamic difficulty adjustment, Deep Q-Learning, Multi agent systems**

I. Introduction

Game Balance is one of the most important factor for a good game, affecting greatly gameplay enjoyment by the user, which can be achieved by game adaptation. Traditionally, a set of static gameplay difficulties are usually provided, or a list of sliders that the user can manually tune, or enable/disable some elements of the game, or none of those, causing the user to be stuck at one difficulty gameplay. However, these solutions can be faulty, causing the game to be either too difficult or too easy, which can make the user lose interest in the game, or causing the user to lose his free time trying to adjust the game to his liking using game sliders, which can be a complicated task.

Achieving game balance has been a great challenge in recent years in machine learning, having been published a great number of articles trying different approaches in order to solve this problem.

In Reis, S., Reis, L.P. and Lau, N., 2020 [1], it is proposed Adjusted Bounded Weighted Policy Learner (ABWPL), a reinforcement learning (RI) algorithm, which tries to improve game adaptation task development efficiency, by modeling the game adaptation design as a game itself.

In this paper, we validate their work, comparing it to a method of deep learning, specifically Deep Q-Learning (DQL), which, as mentioned in their article, is critical for large state and continuous environments.

A. Abbreviations and Acronyms

- ABWPL - Adjusted Bounded Weighted Policy Learner
- DL - Deep Learning
- DQL - Deep Q-Learning
- DRL - Deep Reinforcement Learning
- MAS - Multi Agent System
- MDP - Markov Decision Process
- RL - Reinforcement Learning
- WPL - Weighted Policy Learner

II. Background

A. Game Adaptation by Using Reinforcement Learning Over Meta Games [1]

In the paper cited, the contributions for game balance using machine learning were:

1) A proposal of a game adaptation problem, restricted by a set of adaptation constraints;
2) a framework for automatic balance of a game, where the balance task design is reduced to the design of a reward function, an action space and meta space state;
3) validation of the model to achieve game balance via the implementation of adaptive grid-world scenario.

The formulation of a game adaptation problem is the incorporation over the base game of an agent which assumes the role of game master, restricted by a set of adaptation constraints. During the adaptation process, the game master interacts with the players in the game following a multi agent system (MAS), bounded by a set of balance constraints, learning the optimal strategy to achieve game balance.

The reinforcement learning algorithm, for both player agents and game master agent, used, as previously mentioned, is Adjusted Bounded Weighted Policy Learner [2], which is an extension of Weighted Policy Learner (WPL) based on state counting. WPL is an adapted mixed policy algorithms to a deep architecture, which does not require any knowledge of other players' actions or rewards to reach equilibrium strategies, employing a variable learning rate and dynamic differential equations.

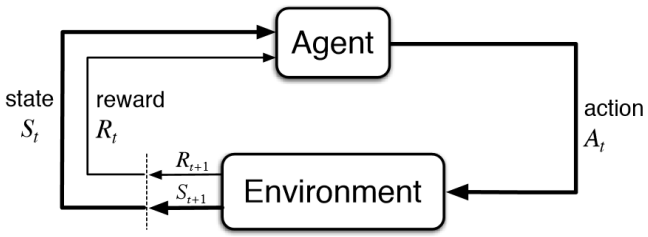

Fig. 1. Agent-Environment interaction in RL.

B. Deep Q-Learning

Being one of the first deep learning models applied to games [4], DQL is a deep reinforcement learning (DRL) algorithm. Deep learning (DL), which derives from artificial neural networks, is a neural network composed of multiple hidden layers of neurons used to discover distributed characteristics and features in data. RL (fig. 1) is a process of continuous decision-making. It does not give any data annotations, just returning a function which determines the current state, and so the objective is to find the optimal return function of the process. In mathematical terms, it is a Markov decision process (MDP). DRL is the combination of the 2 methods, in which RL provides the goal of optimization and DL is the method used to solve the problem [3].

In fig. 1, the agent receives a representation of the environment state $s_t \in S$ (S is the set of all possible states) in each time step t and executes action $a_t \in A$ (A is the set of all possible actions for state s_t). In the next time step $(t+1)$, the agent receives a numerical reward r_{t+1} as a consequence of the action taken, along with a new state s_{t+1}. At each time step, the state maps the state to probabilities of each possible action (agent policy π_t). The goal of the agent is to maximize the total reward value.

The state value function $V^\pi(s)$, which specifies how good is state s, is defined as:

$$V^\pi(s) = E_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s] \tag{1}$$

,in which r_t is the reward at time-step t, $E_\pi[.]$ is the expectation with respect to the policy π and the state and γ^t is the discount factor for future rewards (value between 0 and 1).

The state-action value function $Q^\pi(s,a)$, which specifies how good it is for the agent to perform action a in a state s, according to policy π, can be defined as:

$$Q^\pi(s,a) = E_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a] \tag{2}$$

The optimal state and state-action value function, according to the Bellman optimality, satisfies:

$$V^*(s) = E[R_{t+1} + \gamma \max_{\pi} V(s_{t+1}) | S_t = s] \tag{3}$$

$$Q^*(s,a) = E[R_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') | S_t = s, A_t = a] \tag{4}$$

,in which $R(s,a)$ is the expected reward received after taking action a in state s. Therefore the optimal state-action value function is:

$$Q^*(s,a) = \max_{\pi} Q^\pi(s,a) \tag{5}$$

and the optimal policy is:

$$\pi^*(s) = \arg\max_{a} Q^*(s,a) \tag{6}$$

Q-learning is an off-policy control method. Q-learning learns the state-action value function from experience with temporal difference learning, with bootstrapping, in a model-free, online and fully incremental way. Therefore, the update rule for the state and state-action value function is:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \tag{7}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{8}$$

DRL methods are obtained when deep neural networks are used to approximate any of following components of reinforcement learning: value function, $V(s;\theta)$ or $Q(s,a;\theta)$, policy $\pi(a|s;\theta)$ and model (state, transition and reward). θ are the weights in the neural network.

So, the Q-learning algorithm can be represented as a neural network, taking the state and action as input and obtaining the q-value as an output.

In Algorithm 1, the DQL algorithm uses the following loss function for each iteration i:

$$L_i(\theta_i) = E_{(s,a,r,s')}[(y_i - Q(s,a;\theta_i))^2] \tag{9}$$

,with,

$$y_i = r + \gamma \max_{a'} Q(s', a'; \theta_i^-) \tag{10}$$

,in which θ_i^- are the network parameters used to compute target at iteration i.

The gradient, which calculates the variation of the loss function changing the parameters, is calculated as:

$$\begin{aligned} \nabla_{\theta_i} L(\theta_i) = E_{(s,a,r,s')}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) \\ - Q(s,a;\theta_i)) \nabla_{\theta_i} Q(s,a;\theta_i)] \end{aligned} \tag{11}$$

Require: Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize the target action-value function $\hat{Q}$ with weights $\theta^- = \theta$

for $episode = 1$ **to** M **do**
  initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$;
  for $t = 1$ **to** T **do**
    With probability ϵ select a random action a_t;
    Otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$;
    Execute action a_t in emulator and observe reward r_t and image x_{t+1};
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$;
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D;
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D;
    Set $y_j = r_j$ if episode terminates at step $j+1$, otherwise $y_j = r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-)$;
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameter θ;
    Every C steps reset $\hat{Q} = Q$, i.e., $\theta^- = \theta$;
  end
end

Algorithm 1: Deep Q-learning with experience replay

III. Experimentation Environment

The experimentation environment used used is the same as Reis, S. et. al. 2020 [1]: a maze environment with 2 different settings: single player setting 2 and multi player setting 3. The objetive is for the player (blue) reach the goal (green), navigating through the empty spaces (white) and avoiding walls (black), in a limited number of steps. The game master controls a moving wall (red), which is used to balance the game. The players and the game master have 4 possible actions per turn: up, down, right or left. The game master, beyond the 4 actions available to the players, has 1 more action which is to remain still. The game ends once 1 player reaches the goal, or the limit of steps is reached.

In single player setting, there are 2 balance objetives:

- if $w \leq X$, (w - player's win rate, X - arbitrary threshold) the game master should remain still;
- if $w > X$, the game master should make it more difficult for the player to reach the goal.

The balance state machine for this setting can be observed on Fig. 2(right), with 2 balance states: L for when $w \leq X$, and W for when $w > X$.

In multi player setting, there are 3 balance objectives:

- if $w_1 > w_2$ (w_1 - win rate from player 1, w_2 - win rate from player 2) the game master should try to make it more difficult for player 1;

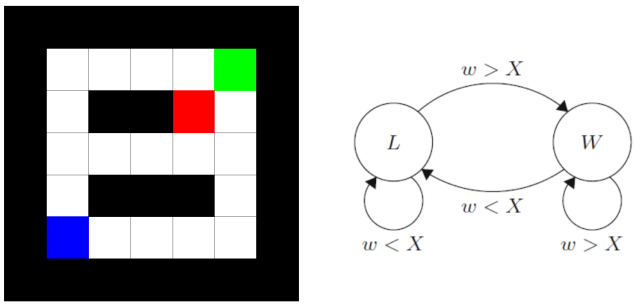

Fig. 2. Single player setting: environment (left) and balance state machine (right).

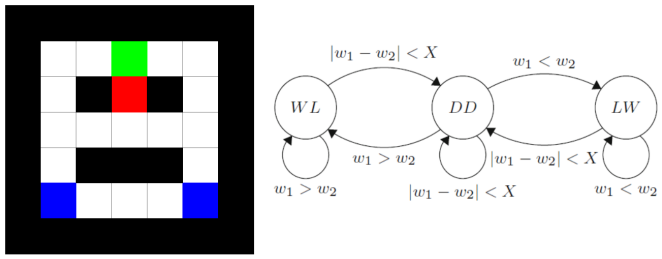

Fig. 3. Multi player setting: environment (left) and balance state machine (right).

- if $w_1 < w_2$ the game master should try to make it more difficult for player 2;
- $|w_1 - w_2| < X$ (X - arbitrary threshold) the game master should remain still;

The balance state machine for this setting can be observed on Fig. 3(right), with 3 balance states: WL (Win Loss) for when $w_1 > w_2$, LW (Loss Win) for when $w_1 < w_2$ and DD for when $|w_1 - w_2| < X$.

On Fig. 4 and Fig. 5, it's presented the balance strategies and rewards of the game master in this environment (t - terminal event, c-collision event with a wall, s - game master state, s_0 - game master initial state, g_0 - goal event player 0, g_1 - goal event player 1).

On single player setting (Fig. 2), we reward the game master for staying still in L balance state, and punish him if the player reaches the goal in W state or if collides with a wall on L state.

On multi player setting (Fig. 3), we reward the game master if he stays still in DD state. On WL state, we reward the game master for trying to block player 1, and punish him if player 1

State W	State L	
t	t	$t \wedge c$
	$s = s_0$	
-1.0	1.0	-0.5

Fig. 4. Single player reward function (table from Reis, S. et. al. 2020 [1])

State WL			State LW			State DD		
		c			c			c
	g_1		g_0	$\neg g_0 \wedge g_1$		$s = s_0 \wedge (g_0 \vee g_1)$		
-1.0	1.0	-0.5	-1.0	1.0	-0.5	-1.0	1.0	-0.5

Fig. 5. Multi player reward function (table from Reis, S. et. al. 2020 [1])

reaches the goal. The reverse happens on LW state, the game master is rewarded if he tries to block player 2 from reaching the goal and is punished player 2 reaches the goal. It is minor punished if he collides with a wall.

IV. Deep Q Learning Implementation

A. Tools used

The original work from Reis, S. et. al. 2020 [1] was developed using *Python*. *PyTorch* was used to implement the algorithm proposed in order to reuse and to take advantage of all the functionality from the previous work.

PyTorch [5] is an optimized tensor library for deep learning using GPUs and CPUs for *Python*, which provides packages not only for neural networks, but also for their optimization.

B. Deep Q-Agent

In the Deep Q-Learning Agent, there are generated 2 similar networks, with hypermeters:

- $batch\_size = 64$: number of samples per training round;
- $\gamma = 0.99$: discount factor;
- $\tau = 1 \times 10^{-3}$: soft update of target network parameters;
- $\lambda = 5 \times 10^{-4}$: learning rate;
- $EPS\_START = 0.9$, $EPS\_END = 0.05$ and $EPS\_DECAY = 200$: hyperparameters that determine the probability of choosing a random action: start at EPS_START decays exponentially to EPS_END, at EPS_DECAY rate;
- $UPDATE\_EVERY = 4$: how often is the network updated;

The 2 networks generated are:

- Policy Network, which is updated every step, and which goal is to train a policy that maximizes state-action value function;
- Target Network, which is used to backpropagate through and train the Policy Network. Using the target network Q-values to train the Policy Network improves the stability of the training.

The replay memory, which stores the transitions that the agent observes for later reuse, has size 10000.

The loss function used measures the mean squared error (Equation 9).

C. Demos

For demonstration purposes, we setup up two different environments:

- Single Player Demo: a single player setting in which a Deep Q-Agent plays the game, with a deep Q Agent as the game master;

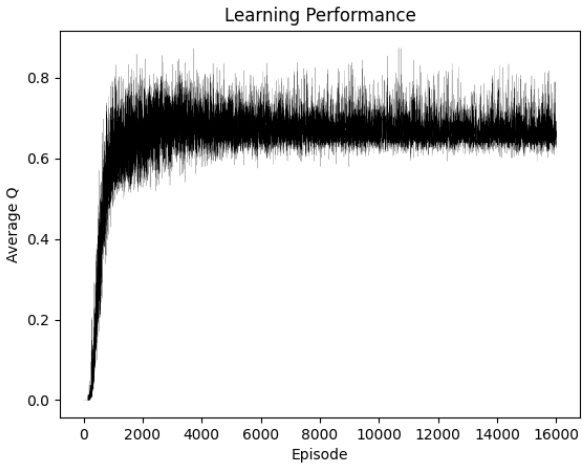

Fig. 6. ABWPL single player setting.

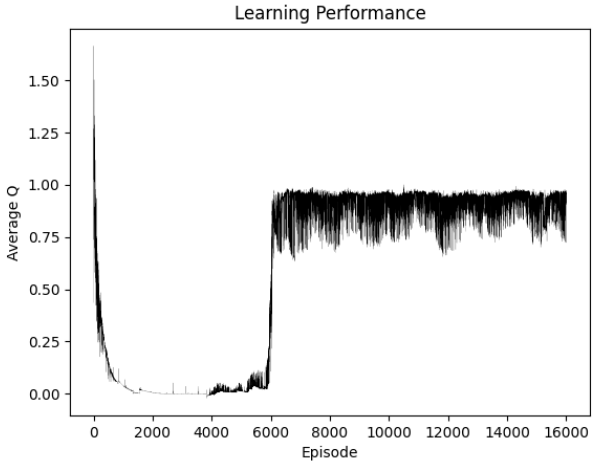

Fig. 7. DQL single player setting.

- Multi Player Demo: a multi player setting in which 2 Deep Q-Agents play against each other, with a deep Q agent as the game master.

In each demo, the model is trained 16000 times.

V. Evaluation of Results

Remembering the goal of this study, the objective is to validate the work from the original paper [1] comparing it to a state-of-art deep reinforcement algorithm. Therefore, we evaluate the results comparing the average q-values and the convergence of q-values from the DQL algorithm and the ABWPL algorithm.

Comparing both single player demos (ABWPL - Fig. 6, DQL - Fig. 7), both converge to similar average q-values, but the ABWPL algorithm converges much quicker, converging around the 1000^{th} episode, while DQL converges around the 6000^{th} episode.

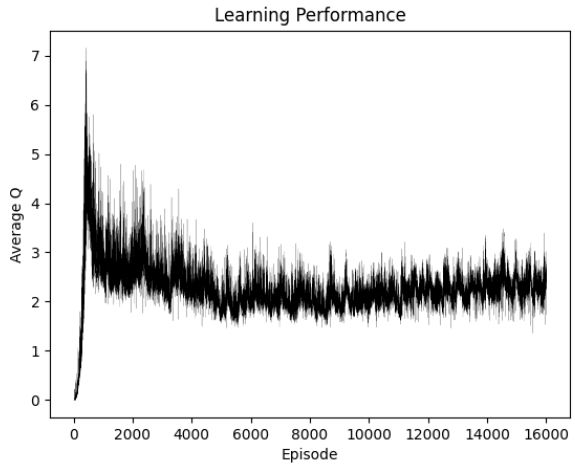

Fig. 8. ABWPL multi player setting.

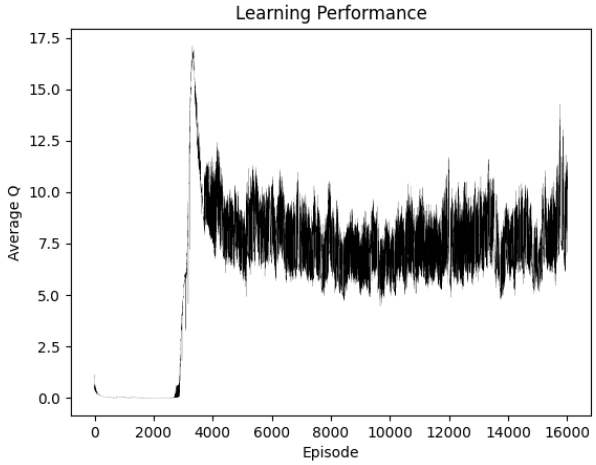

Fig. 9. DQL multi player setting.

On multi player setting (ABWPL - Fig. 8, DQL - Fig. 9), although the ABWPL algorithm converges a bit quicker than the DQL algorithm (1000^{th} episode compared to 4000^{th}), the game master is better rewarded on DQL, converging to an average q-value of 8, while on ABWPL, the value converges to about 2.5.

Also, we observed that the DQL simulations more time to execute than the ABWPL simulations.

VI. Conclusion

The main focus of the work was successful. ABWPL algorithm proved to be better than Deep Q-learning. It had quicker execution, quicker convergion and was more consistent (the average q values returned on different experiments were more consistent on ABWPL).

For future work other types of q-learning could be used such as:

- Delayed Q-learning
- Greedy GQ with linear function approximation
- Double Q-learning
- Deep Q-learning

References

[1] Reis, S., Reis, L.P. and Lau, N., 2020. Game Adaptation by Using Reinforcement Learning Over Meta Games. Group Decision and Negotiation, pp.1-20.

[2] Simões, D., Lau, N. and Reis, L.P., 2018, June. Adjusted bounded weighted policy learner. In Robot World Cup (pp. 324-336). Springer, Cham.

[3] Tan, F., Yan, P. and Guan, X., 2017, November. Deep reinforcement learning: from Q-learning to deep Q-learning. In International Conference on Neural Information Processing (pp. 475-483). Springer, Cham.

[4] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[5] PyTorch documentation. PyTorch documentation - PyTorch 1.7.0 documentation. Available at: https://pytorch.org/docs/stable/index.html [Accessed January 6, 2021].