Deploying Deep Q-Learning in games ”WaterWorld” and ”FlappyBird”

Alexandre Matton, Cyril Malbranke, Paul Bonduelle

***Abstract*—DeepMind started deep reinforcement learning in 2013 with a paper called ”Playing Atari with Deep Reinforcement Learning” [1]. The authors showed how deep neural networks could be trained to map screen-shots of Atari games (set of pixels) to the right action to take (to estimate a q-value function). Taking the pixels as input for the networks allowed them to generalize their algorithm to a bunch of environments: one agent could learn to take actions in several games, not in only one specific game. Here, we want to use deep q-learning to solve specific games (which will require less computational power than doing one for all games, e.g. by considering the pixels). Unlike what was done by the DeepMind team, our agent will be customized to perform well in a given environment. We will deploy it in the ”WaterWorld” PLE environment, in ”Flappy Bird” and in the ”Alife” environment. We want to discuss technical questions as they rise (architecture of the network etc..) and compare the result with other state of the art techniques.**

I. Introduction

We started by deploying deep Q-learning (DQL) in the WaterWorld environment. In Waterworld, you are playing a blue circle surrounded by movind red and green circle : you must eat the green ones and avoid the red ones. It is an environment which has a high dimensional and continuous state space, making DQL a potentially good agent, so that is why.

One of the big challenges of deploying a DQL agent is the amount of time it takes to train the network. Having little computational power available and making constant updates to our agent to improve it, we weren't able to train our agent for long enough to obtain a good strategy on the WaterWorld environment.

We then decided to deploy it in a simpler environment (with less variables, and less possible outputs), namely the FlappyBird environment, and we obtained an intelligent agent.

We also deployed it in the Alife environment, but as for WaterWorld, it takes more resources than we have to train agents that would give good strategies.

We made available our code[1].

II. Background and Related Work on DQL

We are interested here in situations where the state space S is continuous and where the action space is finite (or can be discretized in to be finite) $A = \{1, 2, ..., K\}$.

Like in Sarsa or Q-learning, the principle of deep Q-learning is to learn in a model-free way an action-value function Q^* satisfying

$$Q^*(s,a) = \max_{\pi}\{\mathbb{E}[R_t|s_t = s, a_t = a, \pi]\}$$

[1]Our code is available here: https://github.com/Nutemm/INF581_AI

Knowing the action-value function for each state and action (s, a) gives us a natural policy: $\pi(s) = argmax_{a\in A}\{Q^*(s, a)\}$

Since the state space is continuous, we must use an approximation of the Q function $Q(s, a; \theta) \approx Q^*(s, a)$. Instead of choosing a linear approximation like we saw in [2], deep Q-learning uses a non-linear approximation of the Q function with a neural network (where parameters θ are the network parameters), referred to as Q-network.

There are two ways of parameterizing Q using a neural network (figure 1). The first way would be to map state-action pairs to scalar estimates of their Q-value, but a separate forward pass is then required to compute the Q-value of each action, resulting in a cost that scales linearly with the number of actions. We can instead use an architecture in which there is a separate output unit for each possible action corresponding to the predicted Q-values of the individual actions for the input state. We can then compute Q-values for all possible actions in a given state with only a single forward pass through the network.

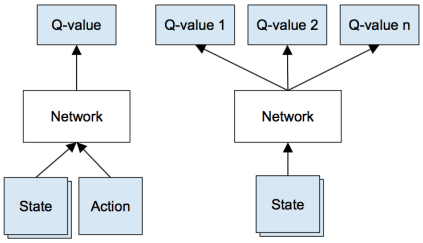

Fig. 1. Two possible configurations

Starting from a random initialization of the network parameters, interacting with its environment (e.g. using the policy given by the Q-network in a $\epsilon-$greedy way) and observing a sequence (s_t, a_t, R_t, s_{t+1}), the Q-network has to be trained by minimizing a loss function that should intuitively look like

$$L(\theta) = (R_t + \gamma \max_{a\in A} Q(s_{t+1}, a; \theta) - Q(s_t, a_t; \theta))^2$$

However, [3] warns instability can be caused by the correlations present in the sequence of observations and the correlations between the action-values (Q) and the target values $R_t + \gamma max_a Q(s_{t+1}, a; \theta)$.

In order to avoid those instabilities, [3] suggests to use experience replay: we can store the agents experiences $e_t = (s_t, a_t, R_t, s_t)$ at each time-step t in a data set $D_t = \{e_1, ..., e_t\}$. Only at certain times (e.g. every N steps), we perform training by applying Q-learning updates on samples

(or mini-batches) of experience $(s_t, a_t, R_t, s_{t+1}) \sim U(D)$ drawn uniformly at random from the pool of stored samples. During training, the loss function becomes

$$L(\theta_i) = \mathop{\mathbb{E}}_{(s_t,a_t,R_t,s_{t+1})\sim\ U(D)}[(R_t+\gamma \max_{a\in A} Q(s_{t+1}, a; \theta_i^-) - Q(s_t, a_t; \theta_i))^2]$$

where θ_i are the parameters of the Q-network at iteration i and θ_i^- are only updated with the Q-network parameters every C steps and are held fixed between individual updates in order to avoid correlations in the action sequences (this technique has been seen in lecture [4]).

III. The Environments

A. Description of the environments

We chose the games WaterWorld and FlappyBird from the PLE (PyGame Learning Environment) library[2] as our environments.

PyGame Learning Environment (PLE) is a learning environment, mimicking the Arcade Learning Environment interface, allowing to do Reinforcement Learning tasks in Python. PLE allows us to focus on the design of models and experiments instead of environment design. However, we altered this environment in many ways, to be able to train our algorithms faster or to change its objectives.

In WaterWorld (figure 2), the agent, a blue circle, must navigate around the world capturing green circles while avoiding red ones. After capture a circle it will re-spawn in a random location as either red or green. The game is over if all the green circles have been captured. Valid actions are up, down, left, right and still. It adds velocity to the agent which decays over time. For each green circle captured the agent receives a positive reward of +1, while hitting a red circle causes a negative reward of -1.

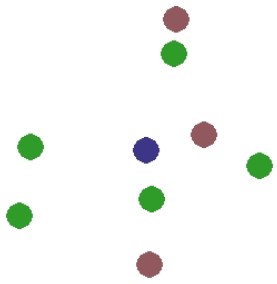

Fig. 2. Screen-shot of the WaterWorld game

This WaterWorld environment is interesting because the state-space is complicated and high-dimensional. Indeed the agent may be faced with very different configurations of its environment (its position, the number of red and green circles, their speed and positions). The actions, on the other hand, are discrete. It thus makes sense to us to try to deploy DQL on that environment.

FlappyBird (figure 8) is a side-scrolling game where the agent must successfully navigate through gaps between pipes. There are only two actions: jumping up or not jumping.

[2]Code available here: https://github.com/ntasfi/PyGame-Learning-Environment

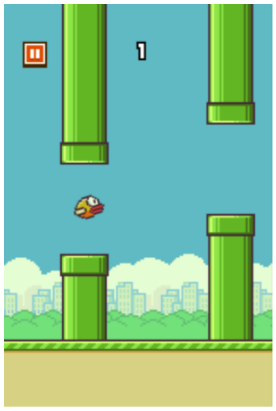

Fig. 3. Screen-shot of the FlappyBird game

Jumping causes the bird to accelerate upwards. If the bird makes contact with the ground, pipes or goes above the top of the screen the game is over. For each pipe it passes through it gains a positive reward of +1. Each time a terminal state is reached it receives a negative reward of -1. This environment is simpler than WaterWorld because its state-space is only 8-dimensional (position and velocity of the bird, distance to the two next pipes, and coordinates of the holes in the two next pipes), thus less interesting, but DQN agents in this environment will be faster to train.

TABLE I
Environments summary

Environment	State space	Action Space
WaterWorld	Continuous, 84-dimensional	Discrete, 5 actions
FlappyBird	Continuous, 8-dimensional	Discrete, 2 actions
Alife	Continuous, 10-dimensional	Continuous, 2-dimensional

B. Adaptations

We made a few modifications in the source code of the WaterWorld environment (as released in the PLE library) before training our agent.

We changed the way circles appear. When a green circle disappears, it is now replaced necessarily by a green circle and same for red circles, keeping a constant ratio of green vs. red. We want to avoid situations in which only one green circle remains, because the agent would hardly find it.

We changed the state-space (which boils down to doing feature engineering before feeding the Q-network) to summarize better the knowledge our agent needs to have to learn how to play. Inspired by a work released by Stanford[3], we gave the agent 16 sensors pointing in 16 directions around him. Each sensor observes in its specific direction the presence of a circle, the distance to the object (if present) and the velocity of that object. Adding to those features the 2D position and velocity of the agent, we get an 84-dimensional state space.

We also slowed down the speed at which the agent moves, thinking it would be easier for it to train. If the agent goes too fast, it can land rapidly on another side of the map and may

[3]https://cs.stanford.edu/people/karpathy/reinforcejs/waterworld.html

accidentally touch red or green circles, which could result in unjustified reward values.

IV. The Agent

As said previously, DQN agents are well suited to solve learning tasks in environments where the state space S is continuous (and potentially multi-dimensional).

What we observed however is that a DQN agent is extremely long to train if there are too many inputs and outputs, i.e. if the environment is too complex. Thus, we couldn't observe any valuable improvement in WaterWorld, so we adapted the same algorithm to FlappyBird and eventually we observed in a few hours very good results.

One thing we realized is that the values of the hyper-parameters are extremely important w.r.t. the overall quality of our agent. On simple games such as FlappyBird, since the agent is faster to train, it makes it easier to see what works and what doesn't by doing lots of trials, contrary to the more complex ones such as WaterWorld.

For FlappyBird for instance, we determined that a small epsilon at the very beginning (such as 0.01) is crucial for the algorithm to manage to go beyond the first pipe (as the agent can decide to go upward approximately 30 times per second). Starting with too high a value won't make it learn at all.

A. Training of the agent

Our agent uses a second target network, which we use to compute target Q-values during our updates (as discussed in section II). If we had used only one network, the network could have became destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target networks weights are fixed, and only periodically updated to the primary Q-networks values. In this way training can proceed in a more stable manner. The first network is updated every 10 steps whereas the second one is updated every 300 steps. We determined these two values empirically : the latter has to be bigger than the former, but not too much, because if so, the formula $L(\theta)$ loses its senses as $Q(s, a; \theta^-)$ values become disconnected from $Q(s, a; \theta)$ ones.

Instead of running Q-learning on state-action pairs as they occur during simulation, our agent stores the data (experience replay) in a buffer and replays it at certain times by pulling random samples from the buffer. That way we make a more efficient use of previous experiences by learning with it multiple times and we get a better convergence behavior, because the data is drawn closer to i.i.d. variables.

Thus, the training phase follows this principle : we use s_t as an input to our network to determine an array containing all the $Q(s_t, a; \theta)$, and for the backward propagation we compare this array to itself, except that we replace the case containing $Q(s_t, a_t; \theta)$ by $r_t + max_a Q(s_{t+1}, a; \theta^-)$

B. Modifications from the initial algorithm

We saw just above that the value of a $Q(s_t, a_t)$ only depends on the ones of the next step s_{t+1}. However, in the case of FlappyBird, rewards are unusual, and for instance an action at a time t may have important consequences over the rewards at time s_{t+5} . To make the algorithm learn faster, and to take into account rewards in a more distant future, we fixed a variable MAX_FEEDBACK, and at each step t, we decided to update the values of $r_{t-1}, r_{t-2}, ..., r_{t-MAX\_FEEDBACK}$ according to the formula :

$$r_{t-i} = r_{t-i} + r_t * \gamma^i$$

C. Prioritized Experience Learning

The replay part of the training agent can be improved. Until now, we selected training examples from those registered with a uniform probability. Many of the training examples are not interesting. The idea is to take in priority the examples that the agent can learn from, that is to say the ones in which the TD difference is big. To do that, we took inspiration from this paper [6] and this github [7]. We sorted our experiences according to their TD difference, and then used their ranking to create a probability distribution:

$$P(experience\ rank\ i\ is\ chosen) = \frac{\frac{1}{i+10}}{\sum_{j \in [|1, max\ rank|]} \frac{1}{j+10}}$$

The implementation is quite complex (more complex than in the papers we cited), because we wanted to keep the modification that we talked about just above. To mix those two tricks while keeping a good complexity, we used heaps (via the python module heapq) and coded some functions for them that weren't available in the module. As it is not the subject, we won't talk in details about our implementation, but you can look at the commented code if you want to know more.

D. The Network

It is hard to find the optimal neural network, as there is no way to find the optimal number of hidden layers and neurons per layer mathematically. However we thought our network should be a fully connected neural network (rather than convolutional or any other form) because the inputs of the network (which represents the state of the agent) are already features of interest for our agent (as opposed to the raw pixel values in the paper published by the DeepMind team, who used constitutional neural networks). We tried a few different networks with 1 and 2 hidden layers, the results are displayed and analyzed in the next parts. The only important point we can highlight here is that the best activation function we found for our NN is the ReLu one. We also quickly tried tanh and sigmoid activations, but they were much less promising, probably because of their maximum threshold. Finally, as we already had to focus on many other parameters, we only tried the classic Adam optimizer.

V. Results and Discussion

A. Performance of our agent in FlappyBird

You can run our code yourself, simply by running `run_task_flappy_bird.py` (you can change the hyper-parameters in `evolver_flappy_bird.py`).

To quantify how well our agent is doing in FlappyBird, we chose to measure the mean number of pipes that he successfully jumped over during the last 500 games (1 game is finished when the bird crashes). This gives as a score. We plotted this score while training the agent to visualize the learning curve.

Our first try was with a simple 1-hidden-layer Q-network of 100 nodes. We got the following curve.

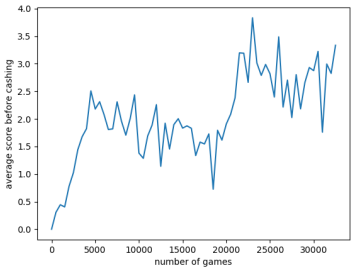

Fig. 4. 1-hidden-layer Q-network of 100 nodes

We then implemented our feedback system, and we managed to get great improvements. The number of steps between two pipes is approximately 30. We tested different values and concluded that setting MAX_FEEDBACK = 10 was the best choice.

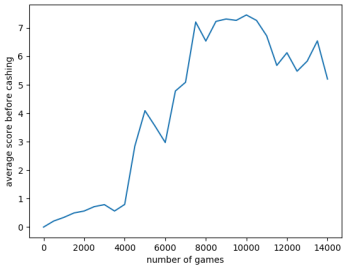

Fig. 5. 1-hidden-layer Q-network of 100 nodes with MAX_FEEDBACK = 10

We also trained the algorithm with two hidden layers in two configurations (100,100 and 24,24). We didn't see any real learning. This may be due to the fact that the number of parameters is much greater than with one layer, so it may need more time to train, and our ϵ decreased to fast.

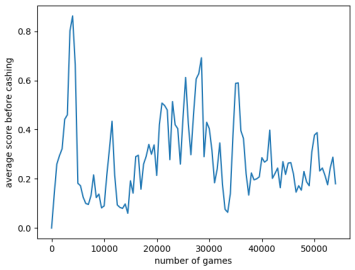

Fig. 6. 2-hidden-layer Q-network of 24 nodes with MAX_FEEDBACK

Finally, we trained a 1-hidden-layer Q-network of 1000 nodes, it trains more slowly than the one with 100 nodes but is more promising.

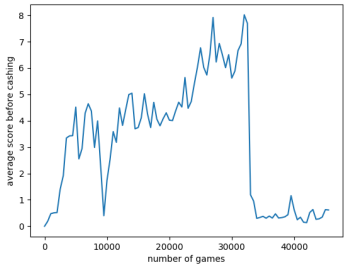

Fig. 7. 1-hidden-layer Q-network of 1000 nodes with MAX_FEEDBACK

What we also observed is the fact that once our ϵ reached a very low value, the algorithms got a tendency to unlearn what they had learnt. With no randomness, algorithms tend to do what they already know, and they can't learn new ways to improve their score. Moreover, they get bad reactions when they crash, and those bad reactions only become worse and more important, until they reach a point in which the agents don't manage to score anymore. For instance, when randomness disappeared, our bird got too many incentives to go up, and that is why, after a small time, he could only do that. Concerning the ϵ decrease speed, we tried a few different things: exponential decreasing, linear decreasing, and no decreasing at all. The exponential decreasing which is the most classic form of the three is implemented in the agent, and we designed a linear decreasing in the main file (run_task) for convenience reasons. We didn't really observe major differences. Moreover, it's hard to determine which one is better than the other because of the randomness of the training. Launching twice the same training may not raise the same results, because of the random initialization of the NN and the ϵ-greedy approach. The only interesting fact we observed is the fact that ϵ should never reach too small (nor too high) a value. 0.005 is an excellent value for this training environment.

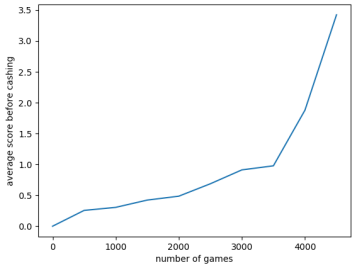

Fig. 8. 1-hidden-layer Q-network of 100 nodes with MAX_FEEDBACK and Prioritized Experience Replay

We didn't have much time left to train Prioritized Experience Replay, and it trained more slowly than without this trick (even though we optimized its efficiency), so we only have the beginning of the curve. Yet, it turns out to be

very promising.

Eventually, we also coded a few other agents to compare them with DQN. Wur ith ocode you can run a REINFORCE agent (`run_task_flappy_bird_reinforce.py`) and a random search or hill-climbing agent (`run_task_flappy_bird_random_search.py`). They all gave very bad results and did not manage to improve, probably because of the fact that positive rewards appear very rarely, contrary to other games such as cartpole, which may be more adapted to these kinds of agents. DQN is different from these algorithms, because the replay phase allows it to overcome the lack of positive rewards, and to improve. The results we got showed that DQN is an algorithm which is perfectly adapted to this game.

B. Performance of our agent in WaterWorld

A natural way to quantify how well the agent is doing in WaterWorld is simply to calculate the ratio of green over red circles hit during the last N periods. We did not manage to see a real improving phase as the ratio value stayed at approximately 1.

C. Performance of our Agent in the ALife Environment

We deployed our agent in the ALife[4] environment.

The first issue to address in deploying DQL in Alife is the computational power needed to train several agents. Indeed in Alife, there are many agents interacting with the environment simultaneously. If we were to train a different Q-network for each agent, it would require much computational power. Instead, to speed up, we decided that all agents of the same type have to share the parameters of one unique Q-network. Each type of insect will be an instance of a class where the network parameters are a global variable. Thus, at a given time-step, all agents deriving from the same class will have the same strategy. When we relaunch the game, we load the weights previously saved.

We also changed the source code of Alife so that when all bugs have died, a new game is launched automatically.

The second issue to address is that the action space A is continuous here (two continuous parameters: angle and velocity). We thought of two ways to fix this. Either we could change the network shape so that state-action pairs are given as input (as discussed in section II), or we could discretize the action space. We took the second option.

Unfortunately, the agent was much too slow to train, so we have no good results to show. However, you can look at the code.

VI. Conclusion

We managed to implement Deep Q learning on various games. We got conclusive results with FlappyBird but didn't manage to get the same ones on WaterWorld and ALife, probably due to the fact that the spaces in which the neural networks are involved are too large, so the training is complex and requires too much time for a single computer. Furthermore, we succeeded in getting better results than with a simple Deep Q learning algorithm by adding new concepts, such as the feedback, for which we took our inspiration from the INF581 course and the SARSA(λ) algorithm. We also implemented prioritized experience replay, which showed promising results. Finally, we had the opportunity to explore many different values for our hyper-parameters, and managed to get a sense of their purpose. For instance, we now know that a too low value for ϵ induces no randomness, which is very hazardous for the learning phase. Fixing hyper-parameters to the right values appeared to be vital to make our agents learn the best they could.

[4]https://github.com/jmread/alife

References

[1] V. Mnih et al. Playing Atari with Deep Reinforcement Learning, *DeepMind Technologies*, 2013.

[2] N. Tziortziotis. Lecture V - Approximate and Bayesian Reinforcement Learning. *INF581 Advanced Topics in Artificial Intelligence*, 2018.

[3] V. Mnih et al. Human-level control through deep reinforcement learning, *Nature*, vol. 518, 2015.

[4] O. Pietquin. Lecture V - Deep Reinforcement Learning. *INF581 Advanced Topics in Artificial Intelligence*, Feb. 2018.

[5] O. Vinyals et al. StarCraft II: A New Challenge for Reinforcement Learning. https://arxiv.org/abs/1708.04782, 2017.

[6] One of the first papers dealing with Prioritized experience replay : https://arxiv.org/pdf/1511.05952.pdf.

[7] Prioritized experience replay is explained in a concrete way in this example : https://github.com/msohcw/deep-q-learning-flappy-bird.