LIÈGE université

Optimal decision making for complex problems

FINAL PROJECT - CATCHER

LOWETTE Quentin - MOITROUX Olivier - ROEKENS Joachim
s150853 - s152122 - s150860

MAY 2019

---

1. Domain

The game is a deterministic, fully observable, time-invariant, single agent environment that can be either continuous or discrete. It is a catcher game where fruits fall in a given 2D space and the agent must catch them all with a moving bar at the bottom of this space. If a fruit is missed, the agent loses a life. After 3 miss, the game is over.

A state in this game is described by

[*x coordinate of the center of the bar, speed of the bar, x coordinate of the fruit, y coordinate of the fruit*]

At each time step, the fruit fall according to the formula

$$y_{t+1} = y_t + speed * dt$$

where, by default, $speed = 0.00095 * 64 = 0.0608$ and $dt = 30$.

When a fruit is caught or is missed, a new one appears at random in the top half of the space.

The agent can take an action to move the bar of the catcher. For the discrete case, these actions are either `LEFT`, `STAY` and `RIGHT`, respectively -1, 0 and 1. For the continuous case, actions are real numbers in the interval $[-a, a]$ where $a = 0.021 \times \text{width\_game\_space}$.

Whatever the case, the bar, initially centered, will slide at each time step according to

$$x_{t+1} = x_t + speed_{t+1}$$

where $speed_{t+1}$ is updated by

$$speed_{t+1} = (speed_t + dx) * 0.9$$

where $dx = max(min(action, 0.021 * width), -0.021 * width)$

For each time step, the agent receives from the game a reward that is equal to

$$reward = \begin{cases} +3 & \text{if the fruit is caught} \\ -1 & \text{if the fruit is missed} \\ +1, & \text{otherwise} \end{cases}$$

2. Performance experimental protocol

To assess the performance of the different algorithms explored, during the training of a model, at multiple times, the model is saved. Then those saved architectures which correspond to the model freezed at different training

time, will be run on 3 or 5 episodes with 1000 time steps time limit. If time allowed, this has been repeated for the same technique and thus the results are averaged over multiple models at multiple time of training for multiple episodes played. This allows to average a bit the environment as well as the average performance of the policies trained.

3. Policy Search Techniques

3.1 Fitted Q Iteration

The Fitted Q iteration algorithm seen in the course was implemented to test the discrete action space. It has been tested with neural networks and ensemble of regression trees for the Q estimator. The testing set (TS) was limited to 80000 time steps.

For the ensemble of trees, the models are trained with :

- Number of iteration max : 20
- Number of trees : 50
- Discounted factor γ : 0.95

The testing was done by averaging the results of 10 models. For each, its state has been saved at 10 times during training and each state played 5 episodes (cf. Figure 1). Has it can be seen, after 10 iterations, the model is already capable of catching every fruits and it will never let a single fruit fall on the ground in the next iterations (as the standard deviation is null). Note that even if each game has a the same time limit, as the fruit reappear randomly in the top half of the window, the number of fruits caught differ even though the number missed stayed at 0.

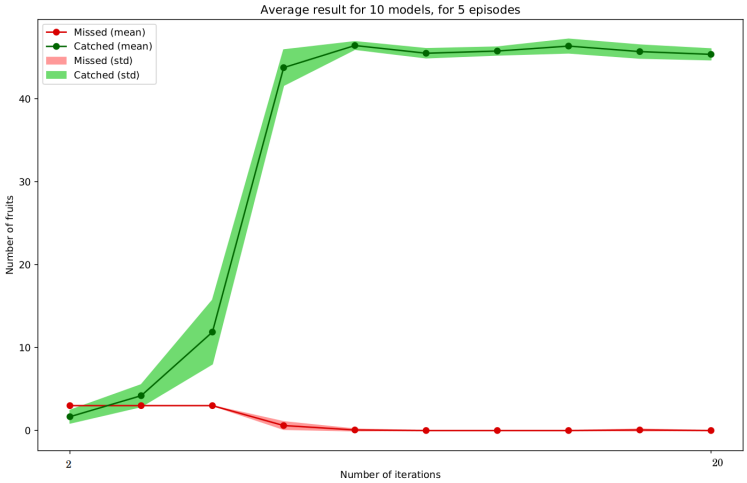
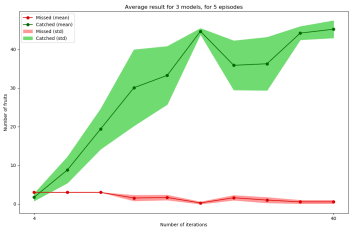


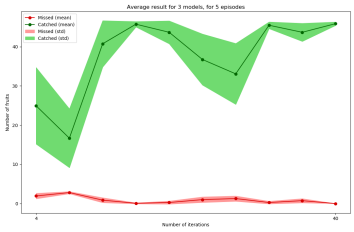
FIGURE 1 – Result for the FQI using the ensemble of trees at different time of the iteration. For each iteration (each point) it gives the mean and standard deviation of the number of caught and missed fruits for 3 episodes of maximum 1000 time steps for the 10 models.

For the neural networks, multiples configurations were tested (cf. Figure 2). By experiment, it seemed necessary to increase the number of iteration to 40 compared to the ensemble of trees. It can be observed that using neural networks is less stable than ensemble of trees. Indeed in all configuration, expect the third one, there exist a time during the training where the model was working perfectly but then it losses in performance in the next iterations. As each training was done on a different TS, it seems that the best number of iteration depends solely on the configuration.

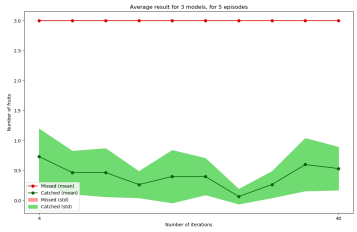
The best result were obtained with the 4th configuration. This configuration differs from the other by its smaller architecture with 2 hidden layer (20x20) against the two hidden layers (64x32) for the others (cf. Appendix A for details) . As it can be seen with Figures 2d and 2e, the approach is not perfect as some one of the model worked perfectly but not especially the ones trained longer. However, the fact remains that it is possible to get a good model with the FQI and neurals networks.



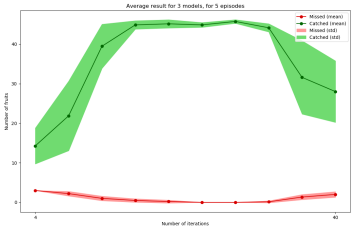
(a) Configuration 1



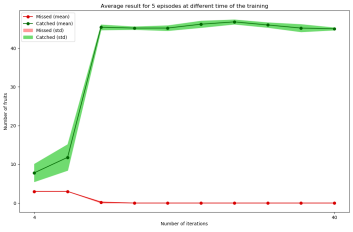
(b) Configuration 2



(c) Configuration 3



(d) Configuration 4



(e) Configuration 4 : first of the three models

FIGURE 2 – Fitted-Q algorithm with neural network tested on different configurations (details in Appendix A).

3.2 Deep Q Learning

Insights

The paper [Mni+13] introduced an algorithm called deep Q learning that aims at learning a control policy from state action pairs. It resolves the problem by directly using samples from the environment without learning explicitly an estimate of it. In this manner, it is a *model-free* algorithm. Moreover, it is *off-policy* because it learns a strategy to maximize the expected cumulative reward (with ϵ-greedy to tackle the state-space exploration dilemma). A Q-network is used to estimate the expected cumulative reward and is trained by minimizing a sequence of Loss functions $L_i(\theta_i)$:

$$L_i(\theta_i) = \mathbf{E}_{s,a}[y_i - Q_\theta(s,a)] \tag{1}$$

where $y_i = r(s_i, a_i) + \gamma \max_{a'} Q_\theta(s'_i, a'_i)$ for each iteration i.

Starting from this baseline, we used a replay buffer to alleviate the problem of correlated data and of non-stationary distributions. Randomly sampling previous transitions, has the following advantages :

— Each step of experience can be used for several weights updates which result in greater data efficiency..
— Because of the correlation between samples, learning directly from consecutive samples is inefficient. The randomness can break this correlation.
— Replay buffer helps avoid getting stuck in local poor minimum or even to diminish risk of divergence.

The original pseudo-code of the paper is shown in figure 3.

Based on this work, we implemented, using the *Keras* framework, a neural network of 3 hidden layers (64-64-32) as a Q-estimator. As we did not get good results, we downgrade the architecture to a simpler (100-50). We trained the network with the parameters given in table 1.

Results

Figure 4 gives an insight of the expected score the DQN can make. We can easily observe the unstability of the training. While some intermediate networks performs ok, the next ones may be much less good. The best

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

FIGURE 3 – Deep Q learning [Mni+13]

Traing parameter	Value
γ	0.95
`N episodes`	500
`epTmax`	250
`xplrSteps`	`nEpisodes*epTMax`/2
ϵ	1 :xplrSteps :0.1
`Batch size`	20
`Buffer size`	800

TABLE 1 – Training parameters for deep Q learning

network is able to catch 18 fruits (average of 10 runs) while still having one life before being interrupted by our threshold on the number of timesteps per episode set to 1000.

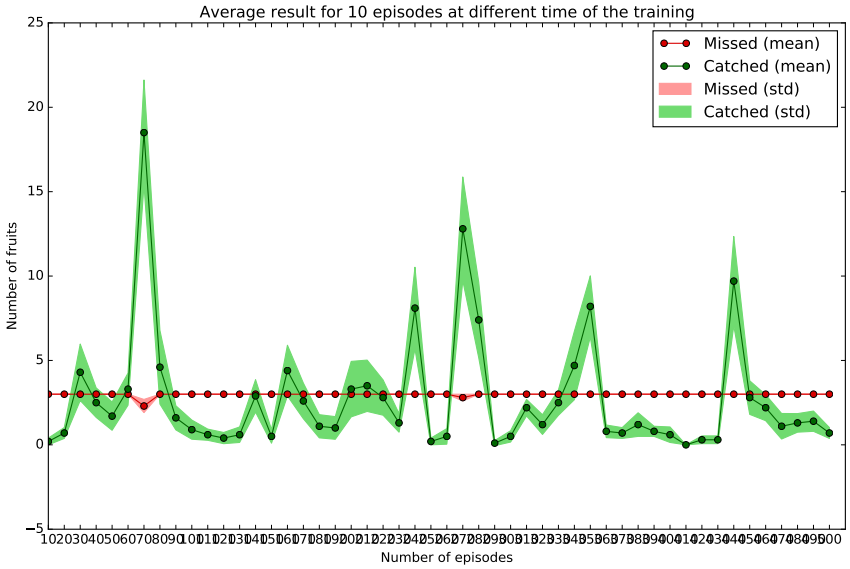


FIGURE 4 – Performance of deep Q learning

While an error in the implementation is perhaps possible, we notice from the plot that the DQN algorithm, despite the use of a replay buffer, is still very unstable. Thus, in order to improve this technique, we investigated double deep Q learning that is the subject of the next section.

3.3 Double deep Q learning

In equation 1, we notice that we use the same weights for estimating the target and the Q value. This leads to a big correlation between the target and the parameters θ that we are learning. In other words, at every step of training, the Q values shift but so as the target value. The direct consequence is big oscillations in training (cf. Figure 4).

In order to improve the robustness of the learning process and the speed of convergence, we use double Q-learning. The pseudo code of the original [HGS15] paper is the following :

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

FIGURE 5 – Double deep Q learning

To tackle this problem, we use a separate network with fixed parameters for estimating the target. Then, at every τ steps, we update its parameters with the weights of the Q network. This way, the bellman equation is modified to incorporate the new network created that shares the same architecture as the Q network. The change in parameters (weights w or ϕ in general) become :

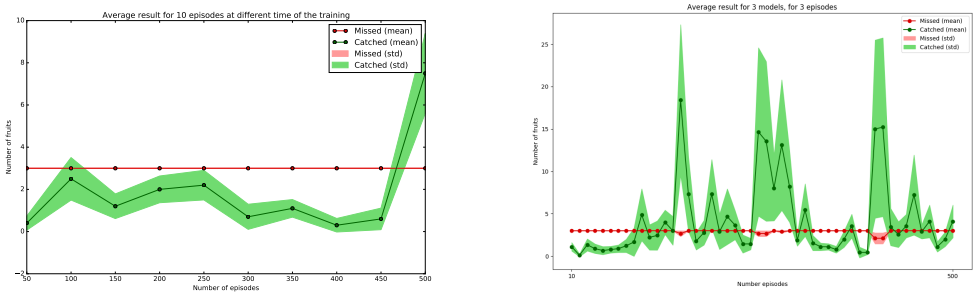
$$\Delta\phi = \alpha[(R + \gamma \max{}_a \hat{Q}_{\phi'}(s',a)) - \hat{Q}_\phi(s,a)]\nabla_\phi \hat{Q}_\phi(s,a) \tag{2}$$

The first term of the difference designate the maximum expected cumulative reward for the next state and is estimated by the Q-target network. The second term, the current predicted Q value, is estimated by the q-network. The gradient is computed on the current predicted expected cumulative reward Q.

The update can be rewritten as :

$$\phi \leftarrow \phi - \alpha \sum_i \frac{d\hat{Q}_\phi}{d\phi}(s_i,a_i)((\hat{Q}_\phi(s_i,a_i) - [(r(s_i,a_i) + \gamma \max{}_{a'} \hat{Q}_{\phi'}(s'_i,a'_i)]) \tag{3}$$

3.4 Results



(a) configuration 1

(b) Configuration 2

FIGURE 6 – Double deep Q learning in a discrete action space.

Figure 6 shows the results obtained with double deep Q learning. In both configuration, the value network is of the same architecture as the target network. Table 2 give some insights about the architecture difference. The rest of the parameters are the same as the one shown for deep Q learning in the previous section.

	Optimizer	Learning rate	architecture
Configuration 1	Adam	0.001	100-50
Configuration 2	Adam	0.0001	40-40

TABLE 2 – Configuration for double deep Q learning

We can see that decreasing the learning rate and simplifying a bit the architecture proved to give a little bit better results but the results are still very poor. It should be noted that figure 6b is the result of the average over 3 simulations of the policy while figure 6a is not. Implementing a prioritized replay buffer could also improve the speed and stability of the training process but was not investigated.

4. Deep Deterministic Policy Gradient and continuous action space

4.1 DDPG

Going from a discrete action space to a continuous action space can easily be done by discretizing the space. However it has two drawbacks. On one hand, it reduces the choice of the possible values of the actions. On the other hand, if the dimensionality of the space increases, the number of discrete actions quickly explodes.

Thus it is often intractable or not efficient to discretize the action space and another technique had to be used here. The approach chosen is the Deep Deterministic Policy Gradient (DDPG) from a paper from Lillicrap et al. [Lil+15]. This uses two principal neural networks : the actor and the critic. The critic is the NN responsible for predicting the expected cumulative reward for a given state-action pair while the actor is responsible for giving the best action to do for a given state. It takes inspiration from the Deterministic policy gradient.

The pseudo code is in the Figure 7. It starts by initalizing the networks at random. Then it constructs two targets networks which will learns the weights more slowly in order to have a better stability and convergence. This would have been very useful in light of the result of Deep Q learning (cf Figure 4). Then it initializes the replay buffer as done previously. It defines a noise process $\mathcal{N}$ which will serve as an exploration algorithm to explore new actions. Instead of the random process of the paper, we investigated both the usual epsilon greedy approach and a random normal disturbance of mean 0 and standard deviation equal to 1. Then it has a loop over a certain amount of iterations as done previously in our algorithm implemented. Here, there is the training on the critic based on a mean squared error loss and the training of the actor based on a sample policy gradient. This ends with the update of the two target networks.

There is a lot of similitude between what was implemented in the discrete case and the DDPG. It was made on purpose, the idea was to go from simple and grows the complexity of the algorithm. The implementation has been carried out in Keras. However, making the backpropagation of the sampled policy gradient turned out not to be straightforward with this framework (PyTorch would have been simpler in this regards). As our understanding of Tensorflow was too restricted to tackle the special gradient of an actor-critic setting, we decided to use part of the code of this article to perform the gradient in the lower-level backend tensorflow.

4.2 Results

While the implementation of the algorithm has been done, it is impossible to predict good actions in its current state. Indeed, the network still predicts the maximum allowed action value, whatever the state that it is given in input. For this reason, no plots have been generated. It is however expected to work at least as good as the discrete solutions explored before.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

FIGURE 7 – Pseudo code for the DDPG

5. Improvements

Here is a list of the possible improvements to the project

— DQN and DDQN performs poorly. We can further tune the algorithm, perhaps with grid search.
— DDPG still outputs wrong actions. We should review the code with external sources or better tune it.
— We can use replay buffer with prioritized experience replay to increase the robustness and speed of the training process to find an optimal policy.

6. Conclusion

In a discrete action setting, we achieve good results for the FQI using extra trees as a Q estimator and interesting but less stable one with neural networks. However, extra trees are considerably slower. We discovered that a simpler neural network may give better results in our problem. Then, we built incrementally better policy estimators by first using deep Q learning and then double Q learning to try to address its unstability issues. It should have help but the results are still disappointing. Finally, we implemented the last piece of the puzzle by considering a continuous action space and training a policy estimator computed via the DDPG algorithm without succeeding to extract good predictions out of it.

In conclusion, the best results were obtained with FQI with extra trees. in a discrete action settings where the agent is capable of playing what seems to be an optimal policy as all the fruits are caught.

Annexe A

FQI with neural networks - architectures

Below are the detailed configurations for the test in Figure 2 :

	Config. 1	Config. 2	Config. 3	Config. 4
Num. time steps TS	80000	80000	80000	80000
Num. iterations	40	40	40	40
γ	0.95	0.95	0.95	0.95
Num. epochs	1	3	3	3
Num. hidden layers	2	2	2	2
Num. unit L. 1	64	64	64	20
Act. function L. 1	relu	relu	linear	relu
Num. unit L. 2	32	32	32	20
Act. function L. 2	relu	relu	linear	relu
Act. function final L.	linear	linear	linear	linear
Optimizer	adam	adam	adam	adam

TABLE A.1 – Different configurations tested for the FQI with NN

Bibliographie

[Mni+13] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In : *CoRR* abs/1312.5602 (2013). arXiv : 1312.5602. URL : http://arxiv.org/abs/1312.5602.

[HGS15] Hado van Hasselt, Arthur Guez et David Silver. "Deep Reinforcement Learning with Double Q-learning". In : *CoRR* abs/1509.06461 (2015). arXiv : 1509.06461. URL : http://arxiv.org/abs/1509.06461.

[Lil+15] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In : *arXiv e-prints*, arXiv :1509.02971 (sept. 2015), arXiv :1509.02971. arXiv : 1509.02971 [cs.LG].