

# Notes for the navigation project

Raphael Gross

2018  
October

## 1 Introduction

This is the first project of the Udacity DRL course. My goal for this project is to get a better understanding of Deep Q Learning by implementing some of the latest extensions of the DQN [3] algorithm in pytorch. In order to achieve this, I first describe the current state of the art of Deep Q Learning, then i will present some of the results I obtained on the Banana environment.

Extension of the DQN algorithm are treated in the following order:

- DQN [3]
- DDQN [2]
- Dueling network Architecture [1]
- Prioritized Experience Replay
- A Distributional Perspective on Reinforcement Learning
- Rainbow: Combining Improvements in Deep Reinforcement Learning
- Distributional Reinforcement Learning with Quantile
- Neural Episodic Control

Note: for now, I only covered the 3 first: DQN, DDQN, and dueling. I am currently woring on the PER (Prioritized Experience Replay). My plan is to understand, implement and test all of them by the end of the year.

## 2 Testing environment

I trained the agent using different variation of the DQN algorithm in order to navigate and collect bananas in a large, square world see Figure 1 A reward of +1 is provided for collecting a yellow banana, and a reward of -1 is provided for collecting a blue banana. Thus, the goal of my agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

1. move forward.
2. move backward.
3. turn left.
4. turn right.

The task is episodic, and in order to solve the environment, the agent must get an average score of +13 over 100 consecutive episodes.

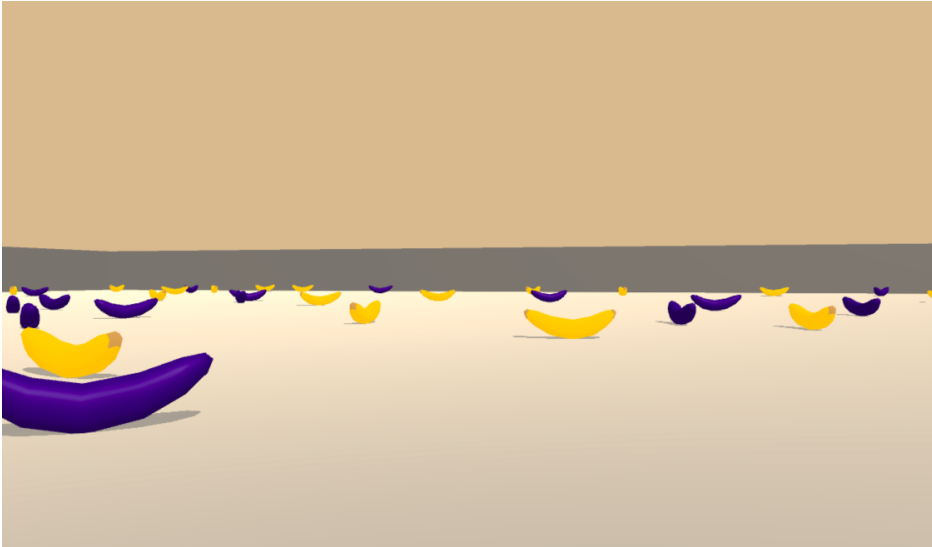


Figure 1: Banana environment

## 3 State of the Art

### 3.1 Definition

I consider in this project an agent that interacts with the environment in a sequence of observations  $s$ , actions  $a$ , rewards  $r$ , new state  $s'$  such as  $(s, a, r, s')$ . At each time step  $t$ , we have a tuple  $(s, a, r, s')$  and each episode is finite. All episode are terminated in a finite number of time-steps. So, we have a well defined Markov decision process.

The agent will interact with the simulator by selecting actions in a way that maximizes future rewards. I made the assumption that futur rewards are discounted by a factor of  $\gamma = 0.995$  par time-step. So for a time step  $t$  the futur reward is:

$$R_t = \sum_{t'=t}^T \gamma^{t'-t} r_{t'} \quad (1)$$

where  $T$  is the time step at which the game terminates.

The optimal action-value function  $Q^*(s, a)$  correspond to the maximum expected return achievable by following any policy after seeing a sequence  $s$  and then taking action  $a$ :

$$Q^*(s, a) = \max_{\pi} E[R_t | s_t = s, a_t = a, \pi] \quad (2)$$

in which  $\pi$  is a policy mapping sequences to actions.

So if the optimal action value  $Q^*$  was know for the next observation  $s'$  for all possible actions  $a'$ , the optimal strategy is to select the action  $a'$  maximizing the expected value of  $r + \gamma Q^*(s', a')$  such as:

$$Q^*(s, a) = E_{s'}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \quad (3)$$

the idea behind this reinforcement problem is to estimate the action-value function by using the Bellman equation as an iterative update:

$$Q_{i+1}(s, a) = E_{s'}[r + \gamma \max_{a'} Q^*(s', a') | s, a]. \quad (4)$$

with  $Q_i$  converging towards  $Q^*$  as  $i$  goes to infini. In practice this is difficult because the action-value function is estimated separately for each sequence, without any generalization. This is why a function approximator

$Q(s, a, \theta)$  is used to estimate the action value function  $Q^*(s, a)$ . I used here a  $Q$  network for approximator where  $\theta$  are the weights.

## Deep Q network or DQN

DQN [3] combine reinforcement learning with deep neural networks. The idea is to use a deep neural network such as a convolutional neural network to approximate the optimal action-value function.

However, reinforcement learning is instable and may even diverge when a nonlinear function approximator such as a neural network is used to represent the action-value function  $Q$ . This instability has several causes:

- correlations present in the sequence of observations.
- small updates to  $Q$  may change the policy and therefore change the data distribution.
- the correlations between the action-values  $Q$  and the target values  $r + \gamma \max_{a'} Q(s', a')$ .

One solution possible is to use experience replay or memory replay. It will randomize the data and remove correlations in the observation sequence. It will also smooth the changes in the data distribution. During learning, we use minibatches of experience  $(s, a, r, s')$  drawn uniformly at random from a pool of stored samples.

Another possibility that helps in reducing the correlations with the target is to use an iterative update that adjusts the action values towards targets values that are only periodically updated. The  $Q$  function approximator is parametrize such as  $Q = Q(s, a, \theta_i)$  with  $\theta_i$  being the weights from the  $Q$  neural network at iteration  $i$ . At each iteration  $i$  we try to reduce the mean-squared error in the Bellman equation, where the optimal target values  $r + \gamma \max_{a'} Q(s', a')$  are substituted with the approximated target values  $y = r + \gamma \max_{a'} Q(s', a', \theta_i^-)$  with  $\theta_i^-$  being the weight of the  $Q$  network from some previous iteration. The parameters  $\theta^-$  are therefore only updated every  $C$  steps and fixed the rest of the time.

The  $Q$  learning loss function is defined as:

$$L_i(\theta_i) = E_{(s,a,r)}[(E_{s'}[y|s, a] - Q(s, a; \theta_i))^2] \quad (5)$$

$$L_i(\theta_i) = E_{(s,a,r,s')}[(y - Q(s, a; \theta_i))^2] + E_{(s,a,r)}[V_{s'}[y]] \quad (6)$$

Since we want to optimize  $\theta$  we can ignore the variance of the targets as it is independant of  $\theta$ .

$$L_i(\theta_i) = E_{(s,a,r,s')}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \quad (7)$$

The loss function is optimize using a stochastic gradient descent. The method is also off policy (memory replay). We ensure the exploration of the state and space by using an  $\epsilon$ -greedy policy.

## Double Deep Q network or DDQN

DQN tend to overestimate the action values especially when the function approximator is inaccurate or the environnement is noisy. The problem with the overestimation is that it is not evenly distribued. Therefore, it may results in a modification in action choice and lead to a sub-optimal policy.

In DQN we use the same values both to select and to evaluate an action. For DDQN [2], we will use the online weights  $\theta_i$  to select an action and the target weights  $\theta_i^-$  to evaluate the action. This idea come from double Q learning were two value functions are learned. One set of weights is used to determine the greedy policy action while the other its value. This change will enable more accurate values estimate and therefore lead towards better policies.

The DQN equations:

$$y_{DQN} = r + \gamma \max_{a'} Q(s', a', \theta_i^-) \quad (8)$$

$$y_{DQN} = r + \gamma Q(s', \operatorname{argmax}_{a'} Q(s', a', \theta_i^-); \theta_i^-) \quad (9)$$

But for DDQN, we used:

$$y_{DDQN} = r + \gamma Q(s', \operatorname{argmax}_{a'} Q(s', a', \theta_i); \theta_i^-) \quad (10)$$

where  $\theta_{1_i}$  and  $\theta_{2_i}$  can exchange position.

## Dueling Network architecture

Compared to a regular DQN or DDQN network, the dueling network architecture [1] separetes the evaluation of the state values function  $V(s)$  and the action advantages function  $A(s, a)$  before evaluting via aggregatin the state action value function  $Q(s, a)$ .

So the dueling network can learn which state is valuable or not without having to learn the effect of each action for each state.

- $V(s)$ : the value of being at that state
- $A(s, a)$ : the advantage of taking that action at that state (how much better is to take this action versus all other possible actions at that state).

from the expression of advantage function we have :

$$Q^\pi(s, a) = V^\pi(s) + A^\pi(s, a) \quad (11)$$

$$V^\pi(s) = E_{a \sim \pi(s)}[Q^\pi(s, a)] \quad (12)$$

$$E_{a \sim \pi(s)}[A^\pi(s, a)] = 0 \quad (13)$$

Therefore, for a deterministic policy we have:  $Q^*(s, a^*) = V(s)$  and  $A(s, a^*) = 0$  for  $a^* = \operatorname{argmax}_a Q(s, a)$ .

Compared to DQN and DDQN, the action value function, of the dueling network use a parametrize estimator for  $Q(s, a)$ :

$$Q(s, a; \theta, \alpha, \beta) = V(s, \theta, \beta) + A(s, a; \theta, \alpha) \quad (14)$$

Even though, the action value is written under this form, it is not possible to use  $V(s, \theta, \beta)$  as a good estimator of  $V(s)$  and  $A(s, \theta, \alpha)$  as a good estimator of  $A(s, a)$ . The network is build to evaluate  $Q(s, a)$  and so it lack identifiability.

By changing the previous expression a little bit, we can write it such that  $V(s, \theta, \beta) = V(s)$  when  $a^* = \operatorname{argmax}_{a'} Q(s, a')$  when:

$$Q(s, a; \theta, \alpha, \beta) = V(s, \theta, \beta) + (A(s, a; \theta, \alpha) - \max_{a'} A(s, a'; \theta, \alpha)) \quad (15)$$

This formulation does not give any advantage to the chosen action. Another possibility which give more stable results is to substract by the mean instead of the maximum:

$$Q(s, a; \theta, \alpha, \beta) = V(s, \theta, \beta) + (A(s, a; \theta, \alpha) - 1/|A| * \sum_{a'} A(s, a'; \theta, \alpha)) \quad (16)$$

Here, the advantage only need to change as fast as the mean instead of the optimal action. Also, as we substract a constant from  $A(s, a'; \theta, \alpha)$  we do not change the ranking for the action value function estimator  $Q(s, a; \theta, \alpha, \beta)$ .

# Results

## 3.2 Models

The architecture of the network used by the regular DQN and DDQN is composed of 3 linear layers:

- $fc1 : nn.Linear(state\_size, fc1\_units)$
- $fc2 : nn.Linear(fc1\_units, fc2\_units)$
- $fc3 : nn.Linear(fc2\_units, action\_size)$

where

- $state\_size = 37$
- $fc1\_units = 64$
- $fc2\_units = 64$
- $action\_size = 4$

In the forward pass,  $fc1$  and  $fc2$  are combined with a *ReLU* rectifier. The dueling network version is a bit more complex as we want to express the Q value function as a sum of the advantage function and the state value function  $V(s) + (A(s, a) - 1/|A| * \sum A(s, a'))$ .

- $fc1 = nn.Linear(state\_size, fc1\_units)$
- $fc2 = nn.Linear(fc1\_units, fc2\_units)$
- $V = nn.Linear(fc2\_units, 1)$
- $A = nn.Linear(fc2\_units, action\_size)$

### 3.3 Parameters study

The default value of parameters to run the different simulation are:

- MEMORY SIZE =  $\text{int}(1e5)$  (replay buffer size)
- BATCH SIZE = 64 (minibatch size)
- GAMMA = 0.99 (discount factor)
- TAU =  $1e-3$  (parameter value used for the soft update of the target weights)
- LR =  $5e-4$  (learning rate)
- UPDATE EVERY = 4 (how often the target network is updated)

I obtained these values after testing several changes for the DQN algorithm. As the dqn converged under 500 iteration, I limited the number of episode to 700. If the agent manage to get a score over 13 for a window of 100 episode, i considere it as converged (C) otherwise I label it as not converged (N).

#### 3.3.1 Repay memory size study

Fixed parameters: batch = 32, tau= $1e-3$ , UPDATE=4, LR= $5e-04$

MEM SIZE	Max iter	score	C/N
100	700	-	N
1e3	700	-	N
1e4	423	13.01	C
1e5	389	13.06	C
1e6	451	13.02	C

Table 1: Memory size study

#### 3.3.2 Batch size study

Fixed parameters: MEM = 100000, tau= $1e-3$ , UPDATE=4, LR= $5e-04$



Batch size	Max iter	score	C/N
16	469	-	N
32	389	13.06	C
64	411	-	N

Table 2: Bath size study

Tau	Max iter	score	C/N
1e-2	700	-	N
1e-3	389	13.06	C
1e-4	700	-	N

Table 3: Tau parameter study

### 3.3.3 Tau parameter study

Fixed parameters: MEM = 100000, batch=32, UPDATE=4, LR=5e-04

### 3.3.4 UPDATE parameter study

fixed parameters: MEM = 100000, batch=32, Tau=1e-4, LR=5e-04

UPDATE	Max iter	score	C/N
2	428	13.08	N
4	389	13.06	C
6	447	13.00	N

Table 4: UPDATE parameter study

### 3.3.5 Learning rate (LR) parameter study

Study of the LR parameter: fixed parameters: MEM = 100000, batch=32, Tau=1e-4, UPDATE=4, LR=5e-04

LR	Max iter	score	C/N
1e-03	470	13.02	C
5e-04	389	13.06	C
1e-05	700	-	N

Table 5: Learning rate study

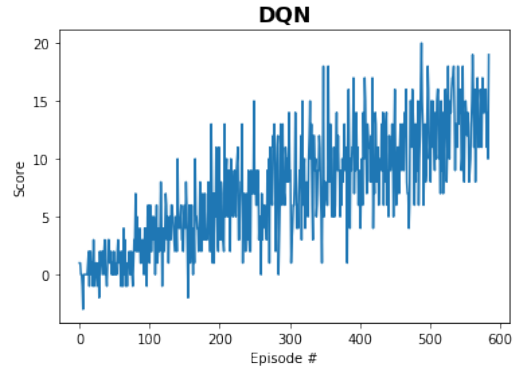


Figure 2: DQN, evolution of the score

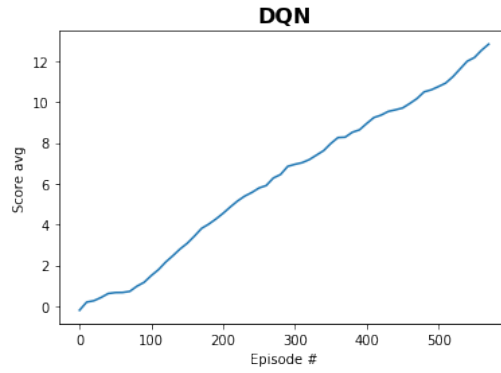


Figure 3: DQN, evolution of the score averaged over 100 window

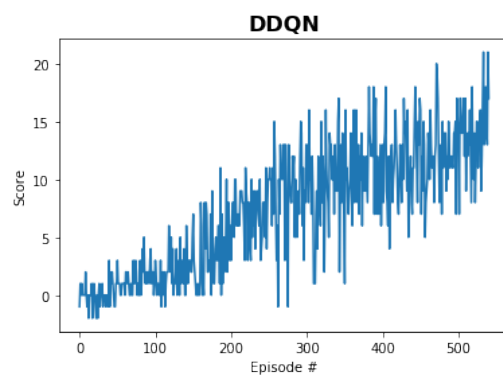


Figure 4: DDQN, evolution of the score

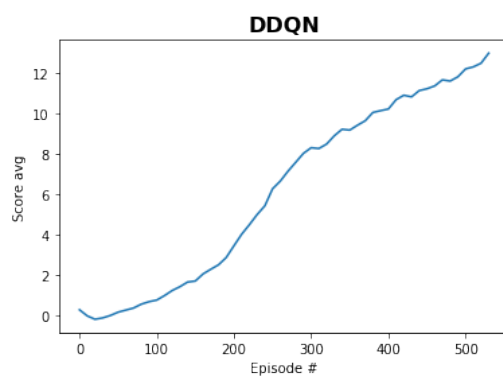


Figure 5: DDQN, evolution of the score averaged over 100 window

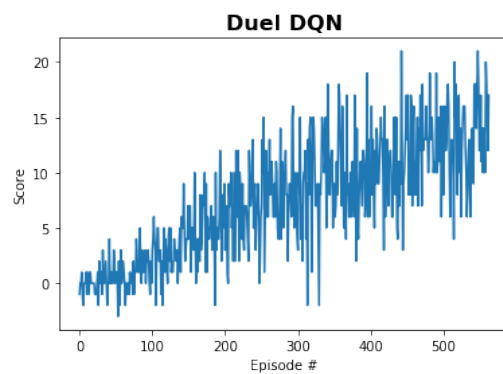


Figure 6: Dueling DQN, evolution of the score

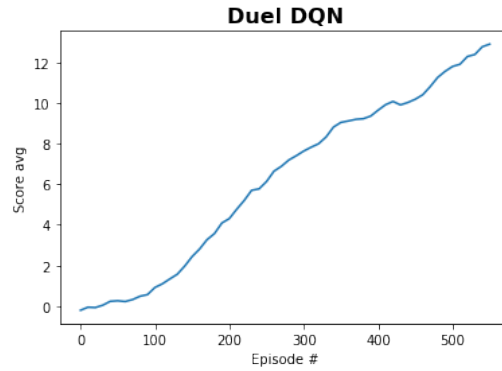


Figure 7: Dueling DQN, evolution of the average score

### 3.3.6 DQN

### 3.3.7 DDQN

### 3.3.8 Dueling DQN

### 3.3.9 Dueling DDQN

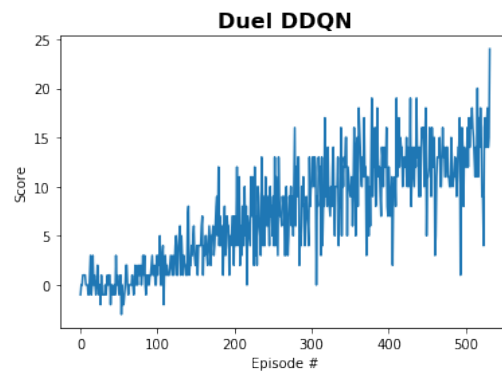


Figure 8: Dueling DDQN, evolution of the score

## References

- [1] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017.

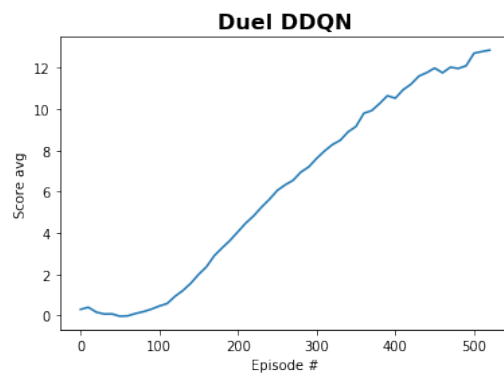


Figure 9: Dueling DDQN, evolution of the average score

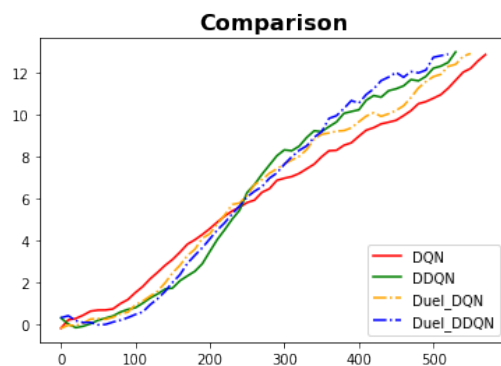


Figure 10: Comparison of the average score for DQN, DDQN, dueling DQN and dueling DDQN

- [2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2094–2100. AAAI Press, 2016.
- [3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.