Notes for the navigation project

Raphael Gross

2018
November

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 network (DQN) by implementing some of the latest extensions of DQN [3] algorithm in pytorch. In order to achieve my goal, I first describe the current state of the art of Deep Q netwok, then I will describe 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]

For now, I only covered these 3 items: DQN, DDQN, and dueling. I am currently working on the PER (Prioritized Experience Replay). My plan is to understand, implement and test the next methods too

- Prioritized experience replay
- A distributional perspective on reinforcement learning
- Rainbow: combining improvements in deep reinforcement learning
- Distributional reinforcement learning with quantile

2 Testing environment

I trained my agent using variations of the DQN algorithm in order to 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 the 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:

- move forward.
- move backward.
- turn left.
- turn right.

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

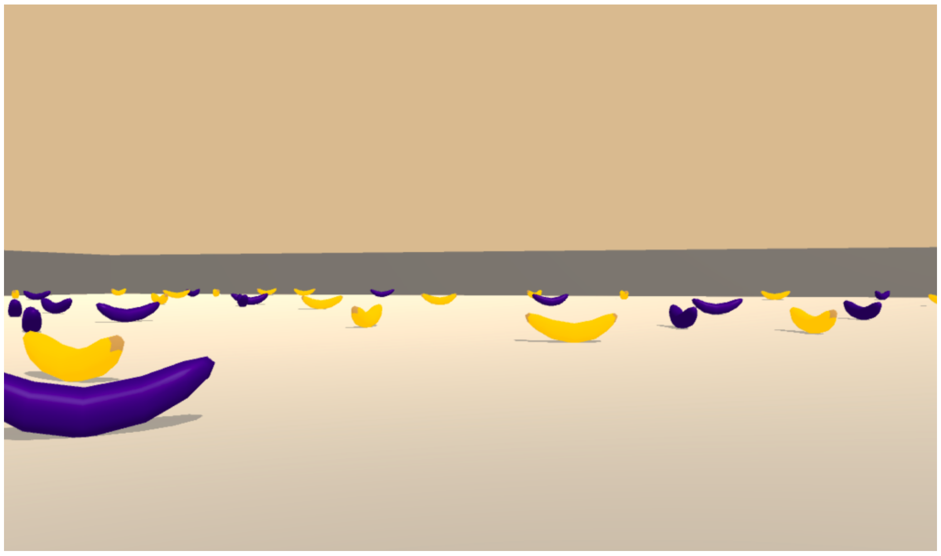

Figure 1: Banana environment

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 states 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 episodes 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 future rewards are discounted by a factor of $\gamma = 0.99$ par time-step. So for a time step t the future reward is:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{1}$$

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

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

in which π 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:

$$Q^*(s, a) = \mathrm{E}_{s'}[r + \gamma \max_{a'} Q^*(s', a') | s, a] \tag{3}$$

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

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

with Q_i converging towards Q^* as i goes to infinity. In practice, this is difficult because the action-value function is estimated separately for each sequence, without any generalization. This is why a function approximitor $Q(s, a, \theta)$ is used to estimate the action value function $Q^*(s, a)$. I used here a Q network for approximator where θ are the weights of the network.

Deep Q network or DQN

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

However, reinforcement learning is unstable 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 $r + \gamma \max_{a'} Q(s', a')$.

A solution 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 mini-batches 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 θ_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 substitued with the approximated target values $y = r + \gamma \max_{a'} Q(s', a', \theta_i^-)$ with θ_i^- being the weight of the Q network from some previous iteration. The parameters θ^- are therefore only updated every C steps and fixed the rest of the time.

The Q learning loss function is defined as:

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

$$L_i(\theta_i) = \mathrm{E}_{(s,a,r,s')}[(y - Q(s,a;\theta_i))^2] + \mathrm{E}_{(s,a,r)}[\mathrm{V}_{s'}[y]] \tag{6}$$

Since we want to optimize θ we can ignore the variance of the targets as it is independent of θ.

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

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

Double Deep Q network or DDQN

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

DQN uses the same values both to select and to evaluate an action where DDQN [2] uses the online weights θ_i to select an action while the target weights θ_i^- evaluate the action. This idea comes 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 to estimate and therefore lead to better policies.

The DQN equation:

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

or

$$y_{DQN} = r + \gamma Q(s', argmax_{a'} Q(s', a', \theta_i^-); \theta_i^-) \tag{9}$$

The DDQN equation:

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

Dueling Network architecture

Compared to regular DQN or DDQN networks, the dueling network architecture [1] separetes the evaluation of the state value function $V(s)$ and the action advantage function $A(s, a)$ before evaluting via aggregation 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:

$$Q^{\pi}(s, a) = V^{\pi}(s) + A^{\pi}(s, a) \tag{11}$$

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

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

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

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

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

Even thought, 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 lacks identifiability.

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

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

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

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

Here, the advantage only needs to change as fast as the mean instead of the optimal action. Also, as we subtract 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 architecute of the network use 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 *Qvalue* function as a sum of the advantage function and the state value function $V(s) + (A(s,a) - 1/|A| * sumA(s,a'))$.

- $fc1 = nn.Linear(state\_size, fc1_units)$
- $fc2 = nn.Linear(fc1\_units, fc2_units)$
- $V = nn.Linear(fc2\_units, 1)$
- $A = nn.Linear(fc2\_units, action_size)$

3.3 Parametric studies for DQN

In this section, I display some results of the parametric studies I made in order to found the best DQN configuration possible.

The best result I obtained for DQN was with the next parameters value:

- MEMORY SIZE = int(1e5) (replay buffer size)
- BATCH SIZE = 32 (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 will use these values for comparing the results between DQN, DDQN, and the dueling version of DQN and DDQN.

Note: as the first DQN I build manage to solve the system under 500 iteration, I limited the number of episodes to 700 for the parametric studies. If the agent manages to get a score over 13 for a window of 100 episodes, I considered it as converged (C) otherwise I label it as not converged (N).

3.3.1 Replay 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 parametric study

3.3.2 Batch size study

Fixed parameters: MEM = 10000, 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 parametric study

3.3.3 Tau parameter study

Fixed parameters:MEM = 10000, batch=32, UPDATE=4, LR=5e-04

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

Table 3: Tau parametric study

3.3.4 UPDATE parameter study

fixed parameters:MEM = 10000, 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 parametric study

3.3.5 Learning rate (LR) parameter study

Study of the LR parameter: fixed parameters:MEM = 10000, 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 parametric study

3.3.6 DQN

In Figure 2, I display the score evolution for the DQN method. The DQN method needs at least 436 episodes before converging to a mean score of 13.03. In Figure 3, I plotted a smoothed version of the score evolution.

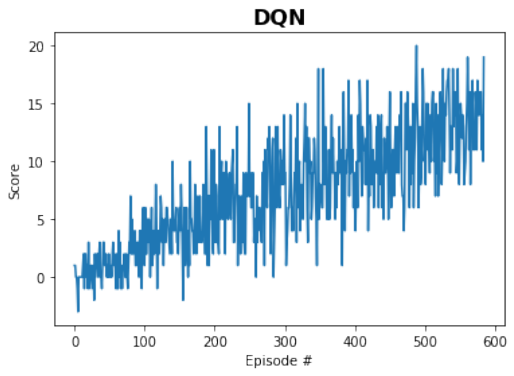

Figure 2: DQN, evolution of the score

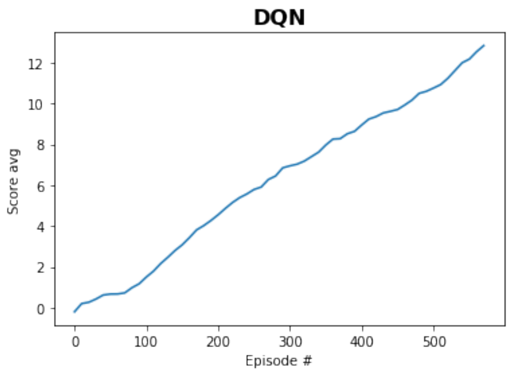

Figure 3: DQN, evolution of the score averaged over 100 window

3.3.7 DDQN

In Figure 4 and Figure 5, I show the score evolution for the DDQN method and the smoothed score curve. The DDQN method took 419 episode to reach a score of 13.

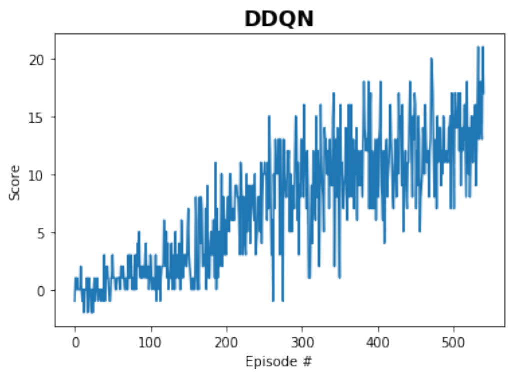

Figure 4: DDQN, evolution of the score

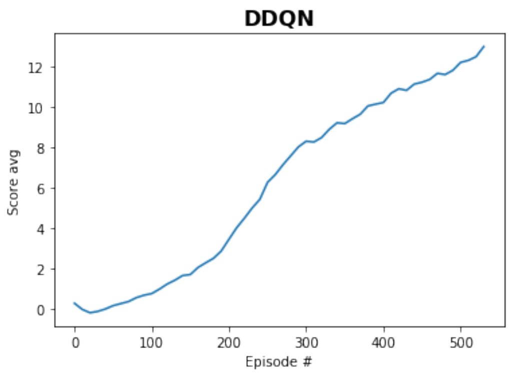

Figure 5: DDQN, evolution of the score averaged over 100 window

3.3.8 Dueling DQN

In Figure 6, I plot the score evolution for the DDQN method and in Figure 7 I display the smoothed graph. The dueling version of DQN took 444 episodes before reaching a score of 13 which is a few iterations less than the regular DQN version.

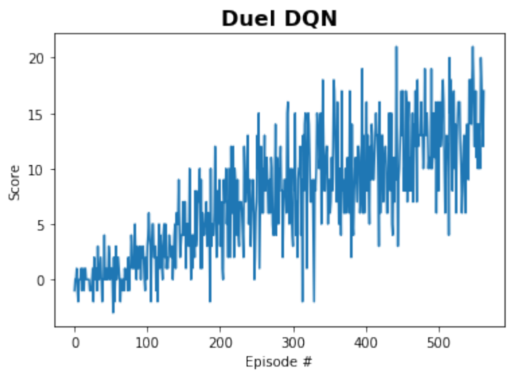

Figure 6: Dueling DQN, evolution of the score

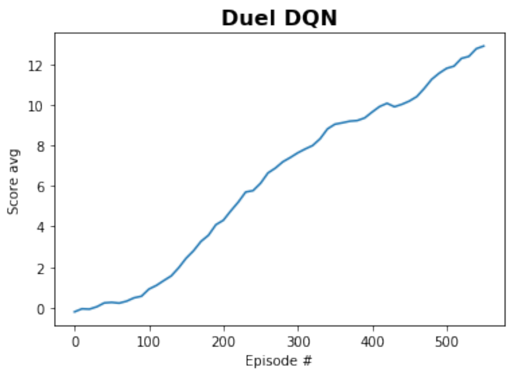

Figure 7: Dueling DQN, evolution of the average score

3.3.9 Dueling DDQN

In Figure 8, I show the score evolution for the DDQN method and in Figure 9 I plot the smoothed score. The dueling version of DDQN only took 375 episodes which is the best result.

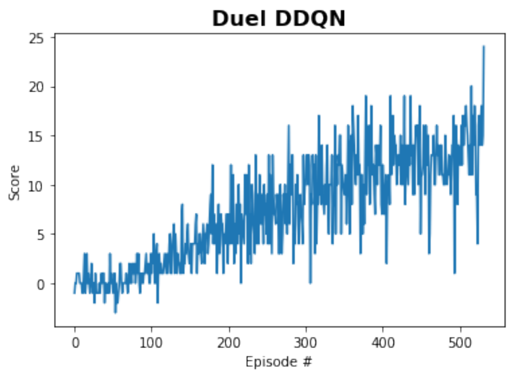

Figure 8: Dueling DDQN, evolution of the score

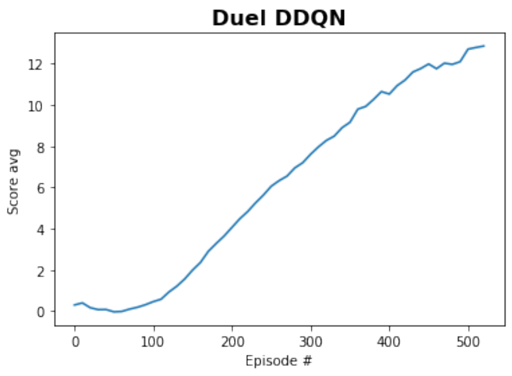

Figure 9: Dueling DDQN, evolution of the average score

3.3.10 Comparison

In Figure 10, I plot the smoothed score evolution for each method. We can see that the dueling version of DQN and DDQN are slower to learn at the beginning but tend to learn at a faster speed later on. Finally, we got the more improvement by switching from DQN to DDQN.

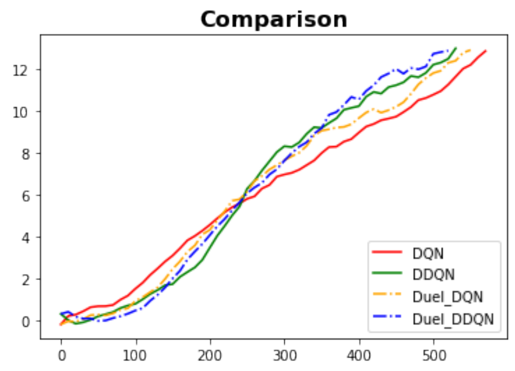

Figure 10: Comparison of the average score for DQN, DDQN, dueling DQN and dueling DDQN

4 Future developments and improvements

As I mentioned in the introduction I covered DQN, DDQN, and dueling network. I am currently working on the PER (Prioritized Experience Replay) and my plan is to implement and test the next methods too:

- Prioritized experience replay
- A distributional perspective on reinforcement learning
- Rainbow: combining improvements in deep reinforcement learning
- Distributional reinforcement learning with quantile

There are other several things that I did not check for now and that could also improve my results. One of them would be to change the size of the network such as the number of hidden layers.

References

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

[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, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.