Reinforcement Learning Project
Learn Atari game *Gopher* through Deep Reinforcement Learning

Ivan Bergonzani

June 30, 2018

Abstract

In this project were tested different deep Q network architectures against the Atari 2600 game 'Gopher'. Base Deep Q network from [3] [4] together with Double Q network [6] and Dueling DQN [7] were trained on the environment provided by OpenAI Gym each for a total of 2 million frames. Despite the smaller training time with respect to the original articles, the three network were able to learn the game. They were tested over a 1000 epsiodes scoring respectively a mean reward of 150, 152, 1521.

1 Introduction

In the last few years the reinforcement learning field has made great advancement and showed potential in the resolution of difficult problems. Examples of these achievement are given by the development of neural networks capable of beating Atari 2600 games as shown in [3] and following works, or by AlphaGO which is an artificial intelligence built from DeepMind that was able to beat the world champion in a complex game like GO. Moreover, reinforcement learning is used in other fields such as robotics, finance and in the medical field.

Differently from the supervised approach used for regression and classification or from the unsupervised approach exploited for clustering, in this branch of machine learning the classical situation is of an agent acting on an environment trying to maximize its performance. Formally, at each step the agent observes the current state $s \in \mathcal{S}$, performs an action from a set $\mathcal{A}$ and collect a reward $r \in \mathbb{R}$: the more the cumulative reward is over time the better is the performance. In fact, during training the agent tries to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the value function $v : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, hence a policy that suggests the best action to take in each possible state. This policy function can be obtained through algorithms like *value iteration*, which estimates first the value function v and then build π on top of it, or like *policy iteration* that instead tries to directly compute π.

Another common algorithm is *Q-learning*, in which the training estimates a function $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ that represents how good is to take action a in a state s. The correspondent policy function will be constructed as $\pi(s) = argmax_a \; Q(s, a)$. The Q function is learned by following an iterative process where at each time t the agent perform an action a_t in the current state s_t, collects a reward r, observes the new state s_{t+1} and finally update the estimate of Q as follow:

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r + \gamma \cdot \max_a Q(s_{t+1}, a)) \tag{1}$$

in which α is the learning rate and γ is the discount factor in the cumulative rewards (*i.e.* how much the future rewards matter).

However, all these basic algorithms work if all the possible state-action pairs are visited infinitely often. For this reason they're used together with an ϵ-greedy strategy in order to have a good trade-off between the exploration and the exploitation phases. In the exploration phase the actions are chosen randomly while during the exploitation phase the actions are chosen using the current policy. In a ϵ-greedy strategy the action is taken randomly with a probability ϵ or it is taken following the policy with $1 - \epsilon$ probability. Usually the value of ϵ varies over time, starting with an high value to encourage exploration in the initial phases and decreasing over time to enforce exploitation.

Recently, Q-learning methods were succesfully used in combination with deep neural networks as firstly shown in [3]. On top of that different learning algorithm and architectures for deep Q-learning were then developed like double DQN [6] and dueling DQN [7].

In this work, these three networks have been tested against *Gopher*, an Atari 2600 game, showing themselves capable of learning the game; all of them are described more in detail in the following section. Section 3 illustrates the game *Gopher* and the environment used to play it. Finally, in section 4, there are reported the performed experiments and the obtained results.

2 Deep Reinforcement Learning

As already said, the agent tries to select action in order to maximize its performance, hence trying to maximize at time t the cumulative discounted reward $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T represents the end game timestep. Then, the training objective is to estimate the optimal state-action value function $Q^*(s, a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$ which is subject to the Bellman equation property and can be rewritten as:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \varepsilon}\left[r + \gamma \max_a Q^*(s', a') \middle| \, s, a \right] \tag{2}$$

The optimal state-action valur function Q^* reflect the optimal policy π^* and in theory can be estimated using the algorithm mentioned in the previous section. In practice this coul be really impractical for difficult problems. As presented in [3], it is possible to approximate Q^* using a neural network: paramatrized by the weights θ, this type of network is referred as Q-network.

Like classical neural network, a Q-network is trained by minimizing a loss function $L_i(\theta_i)$ but this change at each iteration i. Considering for example a L2 loss, the equation is

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{3}$$

in which $y_i = \mathbb{E}_{s'\sim\varepsilon}\left[r + \gamma \text{max}_{a'} Q(s',a';\theta_{i-1})|s,a\right]$ is the update target. At each step the loss L_i can be minimized with the standard stochastic gradient descent algorithm. Notice that, unlike the supervised learnig situation, also the target y_i depends on the network weights θ_i. By differentiating with respect the network weights, the correct loss gradient is

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}\left[\left(r + \gamma\text{max}_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right] \tag{4}$$

Updating the weights at every step and using a single sample as expectation, the obtained algorithm corresponds to the classical Q-learning algorithm.

A useful technique used in the training phase is given by the used of an experience replay buffer. The observed state transitions are continuely saved inside a buffer as (state, action, reward, next state, endgame) tuples. At each training step, a minibatch of transitions is sampled from the buffer and used to update the network.Since the minibatch are randomly sampled, learning will not suffer the correlation related to consecutive transitions.

With the aim to increase the stability of the learning algorithm, it has been used for the training a second network, referenced as *target* network [4]. At each update the target network is used to evaluate the target term y_j and every C steps it is updated by cloning the main network. By using only one network, un update that increase the state-action value $Q(s_t, a_t)$ will increase also $Q(s_{t+1,a})$ for all the actions a and consequently the target term y_j will be increased as well. This behaviour could bring to undesidered oscillation in the policy estimate. With the support of a target network, an increase of $Q(s_t, a_t)$ doesn't influence immediately the target value y_j, reducing the undesired oscillations.

As suggested in Mnih et al. (2015, Nature), to further improve the stability of the algorithm, the error term in the update should be clipped between -1 and 1. This is achieved by using the Huber Loss instead of equation 3:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'\sim U(\mathcal{D})}\left[L_\delta\left(r + \gamma \cdot max_{a'} Q(s',a';\theta_i') - Q(s,a;\theta_i)\right)\right] \tag{5}$$

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{if } |a| \le 1 \\ \delta|x-y| - \frac{1}{2}\delta & \text{otherwise} \end{cases} \tag{6}$$

Obtained by combining the concepts described above, the final learning method for DQN is listed in algorithm 1.

Algorithm 1 Deep Q-learning using experience replay and target network

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $Q'$ with weights $\theta' = \theta$
for $episode = 1, M$ do
    Initialize sequence $s1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ 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 $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + max_{a'} Q'(\phi_{j+1}, a'; \theta') & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $L(y_j - Q(\phi_j, a_j; \theta))$ with respect to weights $\theta$
        Every $C$ steps update target network $Q' = Q$
    end for
end for
```

2.1 Double Q network

A known problem of q-learning is its leaning to overestimate the action values, especially on stochastic environment. This happens because in the q-learning algorithm, as we can see in equation 1, the maximum action value is directly used in the estimation of the value function. For this reason it is easier for the algorithm to choose already overestimated action values and then continue to wrongly increase their estimate.

This problem have been discussed by van Hasselt in [1], where it was proposed the double q-learning algorithm as solution. In double q-learning two different estimators Q, Q' are kept and alternately used for their updates as follow:

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r + \gamma \cdot Q'(s_{t+1}, \text{argmax}_a Q(s_{t+1}, a))) \quad (7)$$

This idea was then used in combination with deep q-learning; presented in [6] from van Hassely et al., the resultant algorithm is called double DQN.

Since the original DQN already used a target network, there's no necessity to introduce another network as second estimators. In fact the target network can be used for this purpose even though this isn't exactly equivalent to the double q-learning update. The resultant training algorithm is the same as algorithm 1 but with a different updating rule: considering the target network paramaters θ', at

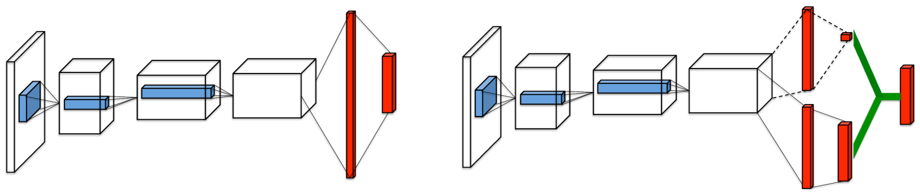

Figure 1: On the left a standard DQN architecture, on the right the dueling deep Q network architecture with the value and advantage streams.

time t the training target y_t is computed as:

$$y_t = r_t - \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a; \theta_t);\ \theta'_t) \tag{8}$$

Differently from double q-learning, this update is used only fot the main network, while the target network is uodated every n steps as in the original deep q learning method.

2.2 Dueling Q network

In some environments it is possible to encounter states where not all the actions are necessary to estimate the value function for the state itself. An example of that is given by the Atari game *Enduro*, in which moving left or right is only useful when a collision is eminent.

In Wang et al. [7] it has been proposed a network architecture that handles this situation by creating two different estimators: the value estimator V and the advantage estimators A. The advantage estimator $A(s, a; \theta, \alpha)$ gives an estimate of the goodness of a particular action a in a state s. On the other hand the value estimator $V(s, \theta, \beta)$ measures how good is to be in a given state s.

More in detail, for an agent that operates in accordance to a policy π the values of the state-action pair (s, a) and the states s are described by:

$$Q^\pi(s, a) = \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right] \tag{9}$$

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}\left[Q^\pi(s, a)\right] \tag{10}$$

Using the value function V and the state-action function Q, then the advantage function is defined as:

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

As shown in figure 1, the advantage function and the value function are implemented in the network architecture as two different branches parametrized respectively from the weights α, β. These branches originate from the same subnetwork parametrized by θ. Finally, considering the set of apossible actions $\mathcal{A}$, the network

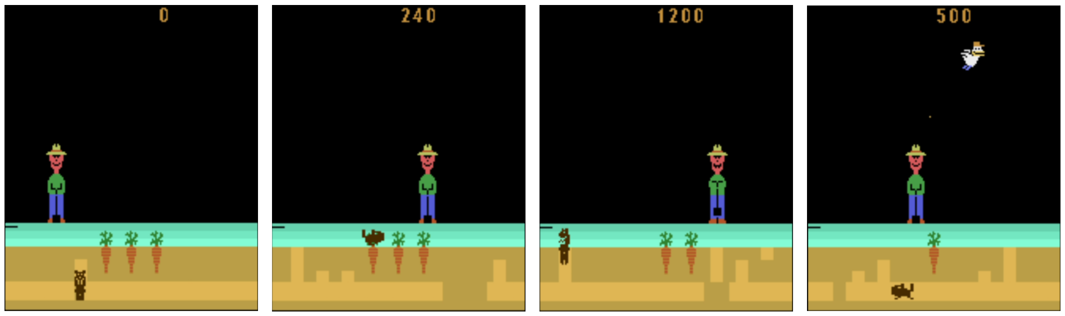

Figure 2: Different in game situations. From left to right: gopher digs underground, gopher eats a carrots, farmer covers holes and gopher reaches the surface, birds with new seed.

output is given by the combination of the value stream V and the advantage stream A as described by the next equation:

$$Q(s,a;\theta,\alpha,\beta) = V(s,;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right) \tag{12}$$

Using the dueling architecture, the network improve its ability to learn the state value function V. The reason behind this behaviour is that the value stream is always updated during each update of the network. Instead in the original architecture only the value related to one action is updated. The dueling architecture has proven to improve more the performances with a larger number of actions, outclassing the original single stream architecture.

Since the dueling Q network consists in just a change of the network architecture, it can be directly used with the existing learning algorithm, like for example double Q learning and SARSA, or with the ones that will be developed in the future.

3 Game and Environment

As already said, the game used for the experiments is *Gopher*. In this game the player controls a farmer and has to keep safe three carrots from a gopher. The gopher digs tunnels underground and try to to reach the surface. When this happen, he will eat a carrot and the player will lose a life. In the moment when all of the three carrots are eaten, the player loses the game. In order to avoid this, the player has to cover the holes that are present on the ground; for each covered hole the player's score will increase. Typical situations of the game are presented in image 2.

The environment used for the game virtualization is provided by OpenAI Gym and it is called *GopherNoFrameskip-v0*. This environment provides methods for resetting, rendering and making steps in the game. After each step, it returns the current observed state (which is represented by the raw pixels of the game), the obtained reward and a boolean that indicates if the episodes is ended or not[1].

In order to decreased the number of parameters, the observations are resized from 210×160 to 84×84 pixel, the resulting images are converted in greyscale and at the end scaled between 0 and 1. Since for the majority of the Atari 2600 games it is not always possible to tell the status of the game from a single frame, the state is augmented to contain the last 4 frames. One example is given by Pong, where it is not possible to establish the ball movement from a single frame. This problem does not really affect Gopher but it has been decided to mantain the frame stack so to be consistent with the articles referred in the previous sections. Considering the small changes in the game from one frame to another, each action is repeated 3 times. In this way the network updates itself one time out of three observations, speeding up the training process[2]. Finally, as explained in [3], the rewards are clipped between 1 and -1 using their sign. This is useful to limit the scale of the error derivatives in case of high rewards. On the other hand the learning could be negatively affected because there are no more differences between rewards of different magnitude.

4 Networks Architectures and Experiments

The experiments on *Gopher* consist in a comparison between a deep q-network, a double DQN and a dueling DQN which also uses the double deep q-learning algorithm. The structures of the trained DQN and double DQN consist in an input layer $(84 \times 84 \times 4)$ followed by 3 convolutional layers. The first layer has 32 filters of size (8×8) with strides 4 on each axis. The second layer has 64 filters of size (4×4) with strides 2. The last convolutional layer has 64 filters of size (3×3) with strides 1. After the convolutional layers there are a fully connected layer with 512 output units and another fully connected layer with an output unit for each possible action in the game. Except for the input and output layer, after each layer there is a ReLU activation function.

The dueling DQN presents the same architecture for the input layer and the convolutional layers. After the last convolutional layer the flow is divided in two stream: the advantage stream and the value stream. Both of them start with a fully connected layer with 512 output units. The advantage stream continues with a FC layer with an output unit for each action, while the value stream ends with a FC with 1 output unit. These two layers are then combined together as described by equation 12. Similarly to the previous case, the layers are separated by a rectifier non-linearity.

For all the networks it was used the RMSProp optimizer with a learning rate of 0.00025 and a momentum of 0.95.

[1] In this environment an episodes end when a carrot is eaten, hence when the player loses one of the three lifes.

[2] One step in the training algorithm would perform three steps in the game.

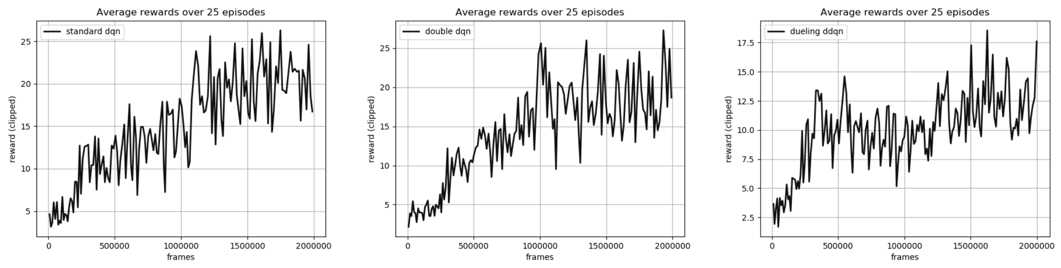

Figure 3: Average rewards collected over 25 episodes during the training of DQN, double DQN and dueling DDQN.

Agent	Mean score	Max score
Random	289.38 ± 237.7	1500
DQN	1283.02 ± 718	6400
Double DQN	1288.4 ± 722.4	6380
Dueling DDQN	906.04 ± 504.6	3100

Table 1: Comparison among the different q-networks, tested on 1000 games.

The training consisted in a sequence of 2 million steps for each network, using random action in the first 50000 steps in order to populate the experience replay buffer, which had a total capacity of 100000 transitions. In order to have a good space exloration, the training used an ϵ-greedy strategy with a decreasing ϵ from 1.0 to 0.1 in the first million steps. For the remaining part of the training the ϵ was constant at 0.1 so to mantain a little exploration. After the first 50000 steps, the network is updated after each action using a minibatch of 32 transitions uniformely sampled from the experience replay buffer. The target network is updated every 10000 steps.

All the experiments were executed on a laptop equipped with an Intel i7-6700HQ processor, 16GB of RAM and Nvidia GTX 960m graphics card. The networks were developed using Python and Tensorflow.

As shown After the training phase, the networks have been tested on a series of 1000 games, wher a game consists in the set of episodes that leads from the starting situation, hence 3 carrots, to the final game situation with zero carrots. The number of episodes can be variable because of the possibility for the farmer to plant new seeds. During the test it has been used a fixed ϵ-greedy value of 0.05. The obtained results are reported in the table 1.

From the results it is clear that all the networks learned the basics of the game, performing well above a random agent. Still the obtained scores are not close to the ones reported in [2], even though the models are not really comparable due to the limited training for the networks described in this work. Moreover the dueling architecture performed badly, even worse than the standard DQN architecture; it is possible that the poor performance could be related to a bad seed for the initial network weights.

5 Conclusion

In this project it has been possible to study different learning algorithms and network architectures related to deep reinforcement learning. The developed models have been tested against the Atari 2600 game *Gopher*, showing their capabilities at learning the game dynamics, despite not reaching the scores obtained in the referenced papers. The reason is attributable to the smaller training time. Anyway, all of the trained models has shown to play way better than a random agent. On top of that, it's possible to learn better policies by using prioritized experience replay [5] in the learning algorithm. This mechanism gives a different weight to the transitions stored in the experience replay buffer based on their temporal error. By prioritizing the examples in this way, the more significative ones has a greater probability of being chosen, leading to a better learning.

References

[1] Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[2] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[4] 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*, 518:529 EP –, Feb 2015.

[5] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[6] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[7] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.