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

Ivan Bergonzani

June 27, 2018

Abstract

In this project were tested different deep Q network architectures against the Atari 2600 game 'Gopher'. Base Deep Q network from [2] [3] together with Double Q network [4] and Dueling DQN [5] 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 [2] 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 $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ 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 [2]. On top of that different architectures for deep Q-learning were then developed like double DQN [4] and dueling DQN [5].

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

$$dsf \tag{2}$$

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 θ
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 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 $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + max_{a'} Q'(\phi_{j+1}, a'; \theta') & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 2 with respect to weights θ
    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.

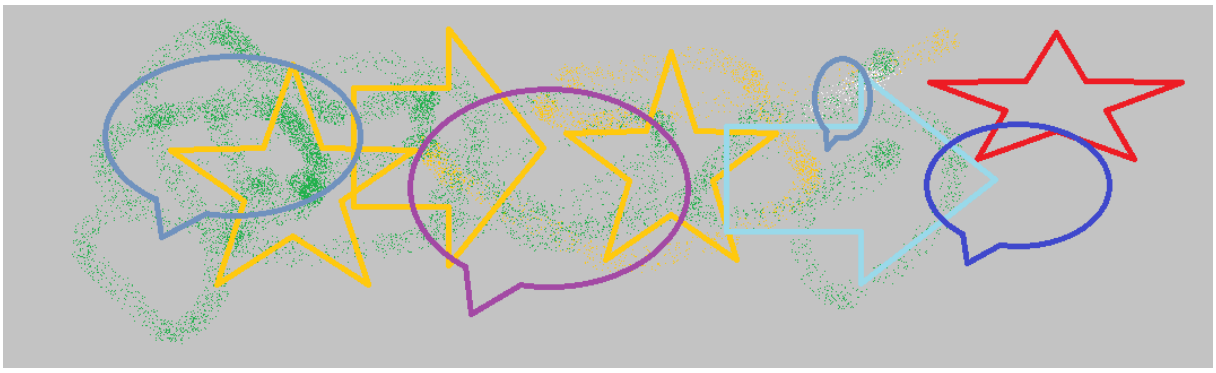

Figure 1: Different in game situations. From left to right: game starting sitution, mole digging underground, mole reaching the soil, mole eating a carrots, farmer covering holes.

For this reason it is easier for the algorithm to choose already overestimated action values and then 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:

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r + \gamma \cdot Q'(s_{t+1}, argmax_a Q(s_{t+1}, a))) \tag{3}$$

This idea was then used in combination with deep q-learning; presented in [4] 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 time t the training target Y_t is computed as:

$$Y_t = r_t - \gamma Q(s_{t+1}, argmax_a Q(s_{t+1}, a; \theta_t);\ \theta'_t) \tag{4}$$

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

(5)

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 1.

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 and the resulting images are converted in greyscale. 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 [2], 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. 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 5. 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.

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.

After the training phase, the networks have been tested on a series of 1000 games, weher 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. The obtained results are reported in the table 1.

[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.

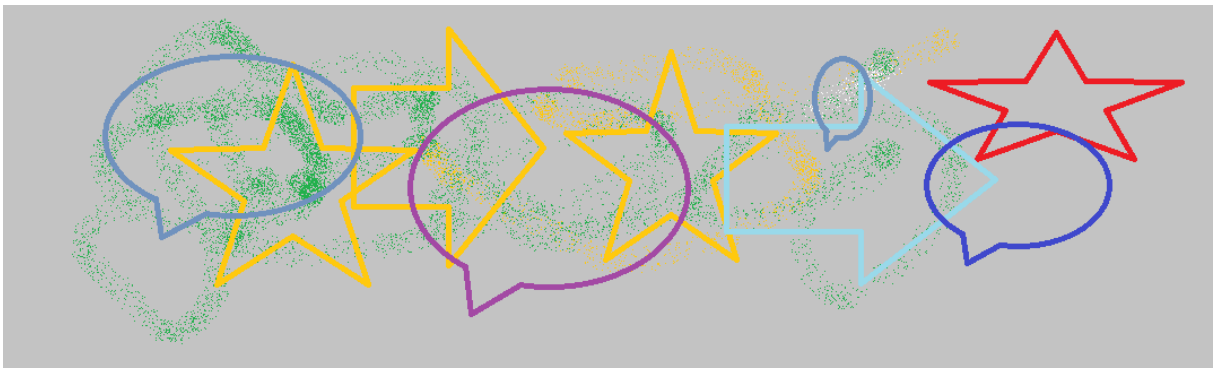

Figure 2: Average rewards collected over 25 episodes during the training of DQN, double DQN, dueling DQN.

Agent	Mean score	Max score
Random		
DQN		
Double DQN		
Dueling DQN		

Table 1: Comparison among the different q-networks

5 Conclusion

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] 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.

[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*, 518:529 EP –, Feb 2015.

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

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