Application of Deep Q-Learning techniques on a GridWorld game

Federico Vaccaro

October 2020 - MultiAgent Systems course project - Università degli Studi di Firenze

Abstract

The purpose of this project is to apply a reinforcement learning technique - the Deep Q-Learning - for allow an agent to learn how to reach a given arrival point in a grid-shaped space, while taking minimal the count of steps and avoiding obstacles.

1 Introduction

The three main paradigms of the Machine Learning are: **Unsupervised** learning, where the model try to understand the underlying distribution of the dataset without any other input aside from the data themselves; **Supervised** learning, where paired with each sample there is a label, and the learning algorithm tries to fit the distribution that bound each sample to its label; lastly, there is the **Reinforcement** learning, where given an *agent* operating within an environment $\mathcal{E}$, has to 'figure out' the *best policy*, i.e. the agent should determine a sequence of steps for maximizing a *cumulative sum of rewards* (and/or minimizing a punishment!) obtained at each time-step (Fig. 1). The first difficulty, respect to the other paradigms, is the ambiguous definition of the problem: there is no explicit dataset and no labels, which makes at first glance unclear how formulating a **loss function** for driving the training. In fact, there are various strategy for tackling the problem: the most known are **value iteration**, **policy iteration** and **Q-Learning**. In this project, we focused on the application of the Q-Learning, however the three listed strategy have in common the property of respecting the **Bellman equation of Dynamic Programmming**. In this report, we will first formulate the *Deep Q-Learning* algorithm, then we will see its application to a simple game we designed.

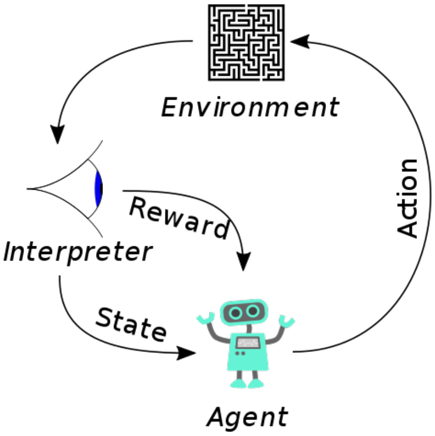

Figure 1: Graphic depiction of Reinforcement Learning training loop.

2 Problem formulation

We consider tasks in which an agent interacts with an environment $\mathcal{E}$, in this case out *GridWorld* game, in a sequence of actions, observations and rewards. At each time-step the agent selects an action a_t from the set of legal game actions, $A = \{1, ..., K\}$. The action is passed to the emulator and modifies its internal state and the game score. In general $\mathcal{E}$ may be stochastic. The emulator's internal state is not observed by the agent; instead it observes an image $s_t \in \mathbf{R}^d$ from the emulator. In addition it receives a reward r_t representing the change in game score. Note that in general the game score may depend on the whole prior sequence of actions and observations; feedback about an action may only be received after many time-steps have elapsed: *e.g.* we will see we designed our game to return a negative 'reward' if a cell has been already visited.

As previously stated, goal of the agent is to interact with the emulator by selecting actions in a way that maximises future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted return at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r^{t'}$, where T is the time-step at which the game terminates. With the discount factor, we discourage the agent to complete the task in too many steps (at $t' = +\infty$, there is no reward). We define the optimal action-value function $Q^\star(s, a)$ as the maximum expected return achievable by following any strategy, after seeing some state/sequence s and then taking some action a, $Q^\star(s, a) = max_\pi \mathbf{E}[R_t | s_t = s, a_t = a, \pi]$, where π is a policy mapping sequences to actions (or distributions over actions).

The optimal action-value function obeys an important identity known as the (previsionale mentioned) **Bellman equation** (Equation 1). This is based on the following intuition: if the optimal value $Q^\star(s_0, a_0)$ of the sequence s_0 at the next time-step was known for all possible actions a_0 , then the optimal strategy is to select the action a_0, s.t. maximize the expected value of $r + \gamma Q^\star(s_0, a_0)$

$$Q^\star(s,a) = \mathbf{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q^\star(s',a')|s,a] \tag{1}$$

3 Deep Q-Learning

We will use a function approximator (as neural networks, which we refer as *Q-networks*), with parameters θ, to approximate the action-value function.

$$Q(s,a;\theta) \approx Q^\star(s,a). \tag{2}$$

A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i.

$$\begin{aligned} L_i(\theta_i) &= (y_i - Q(s,a|\theta_i))^2 \\ y_i &= \mathbf{E}_{s' \sim \mathcal{E}}[r + \gamma \max_{a'} Q^\star(s',a'|\theta_{i-1})|s,a] \end{aligned} \tag{3}$$

The formulation of the loss $L_i(\theta_i)$ in (3) is differentiable, thus we will apply a *stochastic gradient descent* for minimizing the loss. Where the parameters θ_{i-1} are held fixed during the weights update; othewise, with a gradient descent step the target would also try to match the Q-network values, instead of the opposite. Note that this method is **model-free**, i.e. it has no prior knowledge/estimation of the emulator $\mathcal{E}$, but it directly observe its ”external” state. Also, the method is **off-policy**, because it does not learn by directly sampling from a policy (a sequence of steps), but rather training on different actions, even randomly sampled.

Before introducing the final Deep Q-Learning algorithm, we first need to describe the **replay memory** and the **exploration-exploitation** technique for action sampling.

3.1 Replay memory

At each time-step, we collect a tuple from the experience of the agent: $e_t = (s_t, a_t, s_{t+1}, r_t)$. We mantain these experiences in a pool, which we refer as *replay memory*; as the training continues, we build a dataset of experiences. Then we are abilitated to sample mini-batches of tuples from this dataset, making the reinforcement learning algorithm more similar to a ”classic” supervised learning task. This is efficient for two reasons: first, it allows to build minibatches of **indipendent samples**, which are mandatory for a correct estimation of the gradient via SGD; also it is **computationally-efficient**, since we can leverage parallel-computing hardware and optimized software.

3.2 Exploration-exploitation

During the training, we let the agent playing multiple matches, from where we collect experiences. However, especially in the first stages of the training, the

agent does not know how the environment will change to a chosen action, nor the reward it will receive. Thus, it is reasonable to chose a random action instead of the action with maximum Q-score, in order to **explore** the mechanisms of the environment. Furthermore, in the very first stages the parameters are close to being randomly initialized, then the output Q-scores have really few sense. In order to achieving this, we select a probability ϵ of selecting a random action ($1-\epsilon$ selects the *greedy* action) during the training. As the agent keep accumulating experience and training, e will tend to decrease, letting the agent improving the learned policies. We refer to this mechanism as **exploration-exploitation**.
This strategy can also be interpreted from a **Global Optimization** perspective, as an "hill-climbing" method for exiting local minima, or similar to what is known in the Deep Learning community as parameters "warm-up".

Now, we are able to define the **Deep Q-Learning training algorithm**.

3.3 Deep Q-Learning algorithm [1]

In the Alg. 1 we show the pseudocode for the Deep-Q-Learning algorithm.

Algorithm 1 Deep Q-Learning algorithm

```
Initialize ReplayMemory $\mathcal{D}$ with capacity N
Initialize DQN $Q(x, a|\theta)$ with random weights $\theta_0$
for episode $1, M$ do
    Initialize sequence $s_1 = \{x_1\}$
    for $t = 1, T$ do
        With probability $\epsilon$ select action $a_t$
        otherwise select $a_t = \max_a Q(x_t, a|\theta)$
        Execute $a_t$ and observe reward $r_t$ and state change $x_{t+1}$
        set sequence $s_{t+1} = (x_t, a_t, x_{t+1}, r_t)$
        store sequence $s_{t+1}$ in $\mathcal{D}$
        sample a minibatch of $B$ transitions $s_j = (x_j, a_j, x_{j+1}, r_j)$
        set $y_j = \begin{cases} r_t \text{ if } x_{t+1} \text{ is terminal} \\ r_t + \gamma \max_{a'} Q(x_{t+1}, a'|\theta) \text{ otherwise} \end{cases}$
        Compute $L(\theta) = \sum_j^B \frac{1}{B}(y_j - Q(x_j, a_j|\theta))^2$
        Perform a gradient descent update based on $\nabla_\theta L\theta$
    end for
end for
```

3.4 Double Deep-Q Learning [2]

But it is known to sometimes learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values. One way to alleviate this issue is to decoupling the training network from the computing of the target values y_i in (3). If we expand the equation, then we obtain (in a deterministic environment):

$$\begin{aligned} y_i = r + \gamma \max_{a'} Q(s', a'|\theta) = \\ r + \gamma Q(s', \arg\max_{a} Q(s, a|\theta)|\theta) = \end{aligned} \tag{4}$$

In the equation (4) appears that we use the same set of parameters θ for predicting the $\arg\max$ and both computing the targets, which induces the overestimation. By decoupling this task into two set of parameters (*i.e.* into two network with parameters θ and θ^t) we alleviate this issue. During the training, the roles of these two Q-network will be switched after a number of steps.

4 Experimental results

4.1 GridWorld-style game

For testing our implementation of the DQL algorithm, we implemented a game in a *GridWorld* fashion (Fig. 2). The game consists of a grid with shape $D \times D$, where we set $D = 16$. For enabling the rendering to an RGB image, each cell in the grid has 3 channels (thus the state is a tensor $D \times D \times C$). By default, a cell is white (has value 1.0 for each channel). Then, the agent should move its position, represented when a cell is *green* colored to the red square which is the ending (winning) cell. Both start and ending positions are randomly generated at each match to be on the perimeter of the grid, but to be always distant) Also, we generate a number of *obstacles* uniformly spread across the grid, which are represented with the black color. For simplicity, the position of the holes is always the same. The agent can move at each step the green square in one of the four direction (up, right, down, left). The reward system works as follows:

- $r_t = 1$ If the agent get the green closer to the ending square but in a white cell;
- $r_t = -1$ If the agent move the green square upon an obstacle;
- $r_t = -1$ If the agent enter again in an already visited cell (for discouraging loops) **or** try a move towards a wall ;
- $r_t = 2$ If the agent can reach the red square.
- $r_t = 0$ If the agent gets farther from the objective, but in a white cell. There is no penalty here for avoiding punishing the agent for going around an obstacle. We let the discount factor limiting the rewards for being obtained later.

The game terminates when the agent reach the green square; otherwise we adopted an early exit strategy, meaning when the game cumulative reward reaches a negative score of -500 the game is also terminated. This problem, can be interpreted as a different modeling of finding the **shortest-path** between two cells.

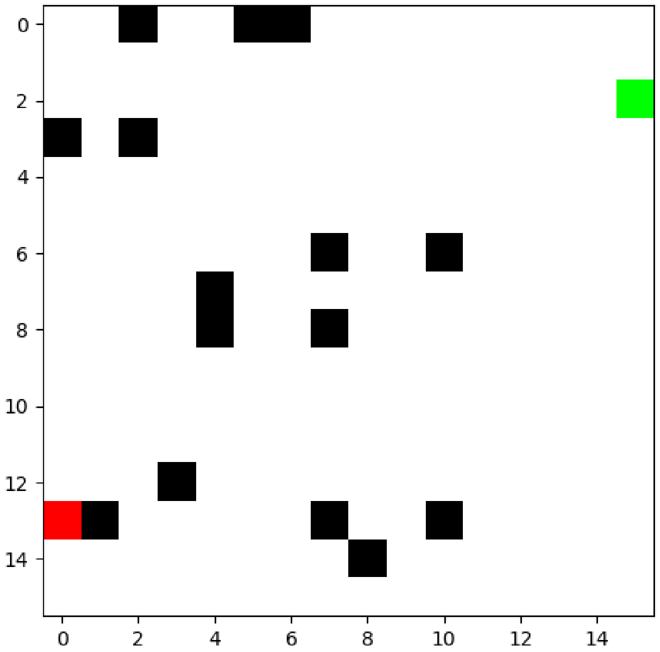

Figure 2: Example of game initial state.

4.2 Implementation details

The network architecture we utilized is composed from one fully-connected (FC) hidden layer with 1024 output units, followed by a batch normalization layer which we found fundamental for regularization purposes, and a final FC layer for producing the Q-scores. After the hidden layer, is applied a ReLU function for obtaining non-linearity. The state tensor, which is the input of the network, is reshaped to a 1-d vector of dimension 768. Summarizing, the computing pipeline is as in (5). We did not find useful to include the last few frames (as is common practice) in the computing of the Q values, since the optimal solution does not depend on the previous steps.

$$Q(\mathbf{s}|\theta) = \mathbf{s} \rightarrow FC_{1024} \rightarrow BN \rightarrow ReLU \rightarrow FC_4 \tag{5}$$

The network is trained for 500 episodes. Each episode, the network is updated 1000 times with the samples drawn from the Replay Memory in mini-batches of 32. The discount factor is fixed at $\gamma = 0.85$. The exploration rate ϵ starts at 0.9, linearly decaying until the quarter-way episode until it reaches value $\epsilon = 0.1$. The employed SGD optimizer is Adam, with *learning rate* $\eta = 10^{-4}$ and *weight decay* (L2 regularization) $= 10^{-6}$. Before computing the action-value function, the state is processed as follows: it is normalized to fit in the range $[-1, 1]$. When using the Double-Q Learning, the network are swapped every 5 episodes.

4.3 Evaluating the models

For the evaluation, we let the agent playing 1,000 matches of the game, we evaluate our agent in terms of counting the number of matches terminated with a positive score, the number of matches terminated on the arrival square and the average score including all the 1,000 matches. We summarize these scores in Table 4.3.

Method	Positives score (%)	Arrivals score (%)	Avg. final reward
DQL	95.8	99.8	18.06
Double DQL	94.6	99.5	14.65

Table 4.3. Evaluation of the Deep Q-Learning methods.

In Figures 3-6 also are shown the reward and loss curves for each episode of training. This shows how the agent actually learns during the training process, increasing the game total reward and correctly approximating the Q-function as the training proceeds.

5 Conclusion

In this project, we experienced a different way of solving problem: the Reinforcement Learning, applying the Deep Q-Learning algorithm. In particular, we found interesting how a small neural network is able to successfully approximate

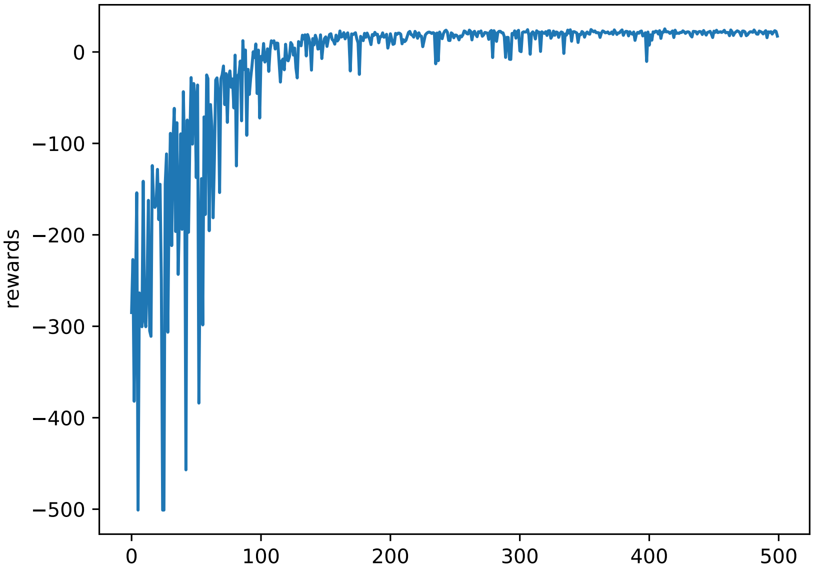

Figure 3: Reward curve during training of DQL algorithm.

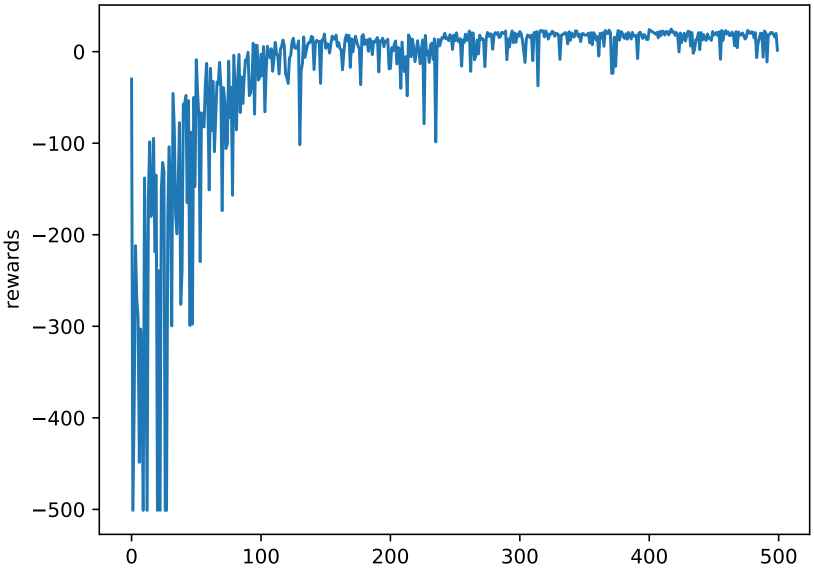

Figure 4: Reward curve during training of DoubleDQL algorithm.

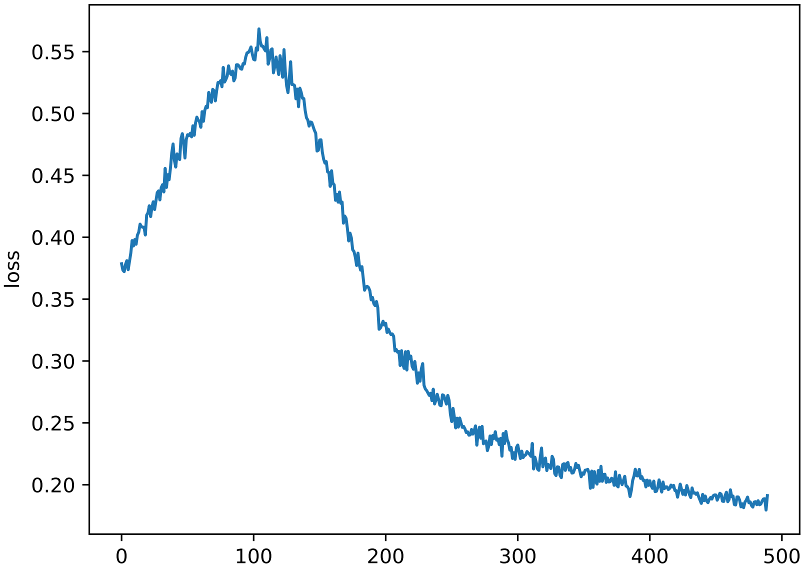

Figure 5: Loss function curve during training of DQL algorithm.

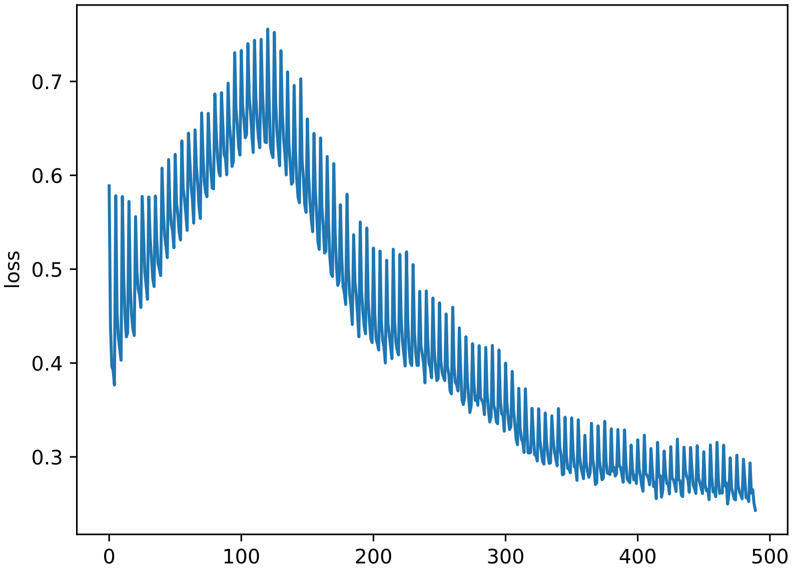

Figure 6: Loss function curve during training of DoubleDQL algorithm.

a classic pathfinding algorithm *e.g. Dijkstra, or the $A^\star$ algorithms* without explicitly coding its rules, but rather by designing a simple reward system, and by approximating the solution to the Bellman equation. However, the real advantage of with this machine learning paradigm, is that we are potentially able approximate a wide range of tasks, not only the class of algorithms we saw in this experiment.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[2] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.