Application of Deep Q-Learning techniques on a GridWorld environment

Federico Vaccaro

Università degli studi di Firenze

federico.vaccaro@stud.unifi.it

October 6, 2020

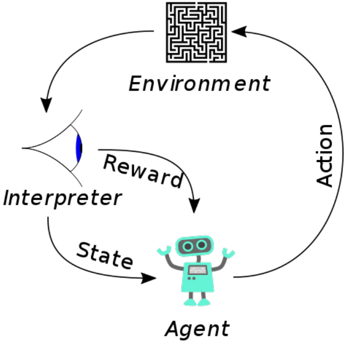
Overview

Paradigms of Machine Learning

- **Unsupervised learning**: learning without annotations;
  e.g. K-means, Mean Shift, GMM...
- **Supervised learning**: learning with annotations:
  e.g. classification, regression with SVM, Decision Trees, Neural Networks...
- **Reinforcement learning**: learning *policies with rewards*:
  - Q-Learning
  - Value iteration
  - Policy iteration

An environment $\mathcal{E}$ and the interaction with an agent modeled as a dynamical system: $s_{t+1} = f(s_t, a_t, \xi_t)$.
The purpose of the agent is **maximizing** the expected reward.

$$\mathcal{V} = \mathbb{E}_\xi\{\sum_{t=0}^{\infty} r(t, s_t, a_t)\} = \mathbb{E}_\xi\{\sum_{t=0}^{\infty} R_t\} \tag{1}$$



Reinforcement Learning

Each of the three strategies is designed to obey a fundamental equation, in reinforcement learning, which is the **Bellman equation of the dynamic programming**.

Theorem (Bellman equation of the dynamic programming at infinite horizon)

$R_t = \alpha^t r(s_t, a_t)$
$\mathcal{V} = \sum_{t=0}^{\infty} R_t$ *is limited if* r *is limited*
$\alpha \in (0, 1)$

$$\begin{aligned} \mathcal{V}^o(s_t, t) = \max_{a_t} \left\{ r(s_t, a_t) + \alpha \mathcal{V}^o(s_{t+1}, t+1) \right\} \\ \gamma^o(s_t, t) = \arg\max_{a_t} \left\{ \mathcal{V}^o(s_t, t) \right\} \end{aligned} \tag{2}$$

Issue: **curse of dimensionality**! Complex to solve when state space $\mathbb{X}$ or action space $\mathbb{U}$ are large!

Q-Learning

We can define the **action-value function** $Q(s, a)$. The Q function is the result of **applying** the action a while in the state s, getting the reward, and adding to it the **total reward** from the next state s_{t+1} obtained by applying the optimal policy $\gamma^o(s_{t+1}, t)$.

$$Q(s_t, a_t) = r(s_t, a_t) + \alpha \mathcal{V}^o(s_{t+1}, t+1) \tag{3}$$

Theorem ($Q^\star$ satisfies the Bellman equation)

$Q^\star(s_t, a_t) = \max_a Q(s_t, a)$
$\mathcal{V}^o(s_t, t) = Q^\star(s_t, a_t)$ *by definition*
$\implies Q^\star(s_t, a_t) = r(s_t, a_t) + \alpha \max_{a'} Q^\star(s_{t+1}, a_{t+1})$

Therefore, the optimal policy $\gamma^o(s_0, 0)$ is simply defined by **selecting at each time-step** the action with **higher Q-score**.

Approximating the Q-function

To counter the course of dimensionality, we can think to **approximate** the Q-function with a Neural Network, which is a **universal function approximator**, parametrized with a set of weights and biases θ.

$$Q(s,a) \approx Q(s,a|\theta) \tag{4}$$

It is trained with a **regression** over the "real" Q-function.

$$\begin{aligned} L_i(\theta_i) &= (y_i - Q(s,a|\theta_i))^2 \\ y_i &= r + \alpha \max_{a'} Q^\star(s',a'|\theta_{i-1}) \end{aligned} \tag{5}$$

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.

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**. Then, we are abilitated to sample mini-batches of tuples from this data-structure.

This is efficient for two reasons:

1. it allows to build minibatches of **indipendent samples**, which are mandatory for a correct estimation of the gradient via SGD;
2. it is **computationally-efficient**, since we can leverage parallel-computing hardware and optimized software.

Exploration-exploitation trade-off

During the first stages of training, the parameters are close to being randomly initialized, then the output Q-scores have really few sense (Recall: $y_i = r + \alpha \max_{a'} Q^{\star}(s', a'|\theta_{i-1})$). To achieving a meaningful approximation, with a probability of ϵ we select a random action ($1 - \epsilon$ selects the *greedy* action) during the training. As the agent keep accumulating experience and training, ϵ will tend to decrease, letting the agent improving the learned policies. We refer to this mechanism as **exploration-exploitation** trade-off: in my experiments, the value is initialized at 0.9, and linearly decays to 0.1 at the 25% of completion of the training.

This technique makes the Deep Q-Learning **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.

DQL algorithm

Algorithm 1 Deep Q-Learning algorithm

Initialize ReplayMemory $\mathcal{D}$ with capacity N
Initialize DQN $Q(x, a|\theta)$ with random weights θ_0
for episode $1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$
  for $t = 1, T$ **do**
    With probability ϵ_{ep} 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

Double Deep Q-Learning

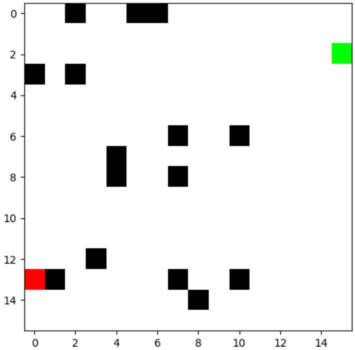
- Sometimes the model learn unrealistically high action values because it includes a maximization step over estimated action values, which tends to prefer overestimated to underestimated values.
- We can decopule the **training network** from the **target values network** y_i (*i.e.* into two network with parameters θ and θ^t). If we expand the equation, then we obtain (in a deterministic environment):

$$\begin{gathered} y_i = r + \alpha \max_{a'} Q(s', a'|\theta) = \\ r + \alpha Q(s', \arg\max_{a} Q(s, a|\theta)|\theta) \end{gathered} \tag{6}$$

The Eq. 6 becomes

$$r + \alpha Q(s', \arg\max_{a} Q(s, a|\theta)|\theta^t) \tag{7}$$

During the training, the roles of the two Q-network will be switched after a number of steps.



- The state is a tensor of dimension $16 \times 16 \times 3$;
- The agent can move the green square at each step the green square in one of the four direction (up, right, down, left);
- The goal is to avoid the obstacles (black cells) and reach the arrival point (red square).
- There are 16×16 reachable position for the agent, multiplied by all the possible start and finish combinations which are more than ~ 60. The number of possible states then is over 15360.

Environment reward system

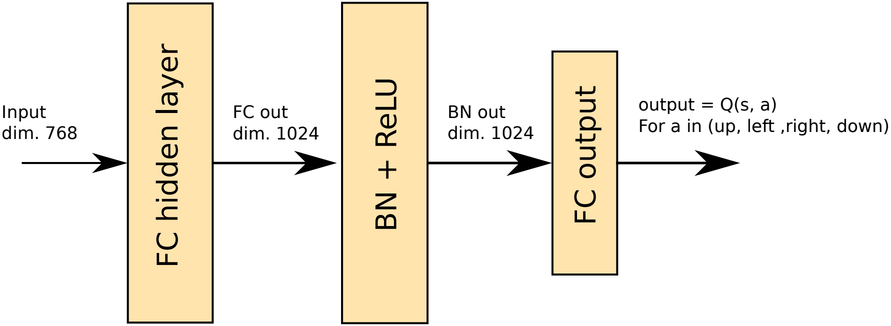
The reward system works as follows:

- $r_t = 1$ 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 not visited yet.

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.

Network architecture

We used a MLP architecture with 1 hidden layer and *Batch Normalization*;



Training hyper-parameters

- The network is trained for 500 episodes;
- Each episode, the network is updated 1000 times;
- The samples drawn from the Replay Memory in mini-batches of 32
- The discount factor is fixed at $\alpha = 0.85$.
- 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 flattened and normalized to fit in the range $[-1, 1]$ with the processing $s_{normalized} = (s - 0.5) * 2$.
- When using the Double-Q Learning, the network are swapped every 5 episodes.

Evaluating the model

After the agent plays 1,000 matches of the game, we evaluate our agent in terms of:

- The number of matches terminated with a **positive score**;
- The number of matches terminated on the arrival square (**Arrivals score**);
- The **average total reward** and **average positive reward** of all the matches. Additionally, this results are sided by the result reached by an "oracle agent", computed as $d(start, finish) + 1$, where $d(*, *)$ is the *Manhattan* distance between two cells.

Method	Pos. score (%)	Arr. score (%)	Avg. final reward	Avg. pos. reward	Avg. oracle reward
DQL	96.0	99.8	18.06	23.95	
Double DQL	94.5	99.5	14.65	23.95	

Reward per episode curves

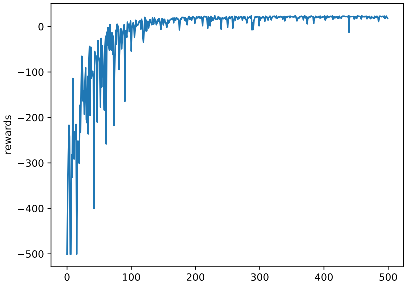


Figure: Reward curve of DQL algorithm.

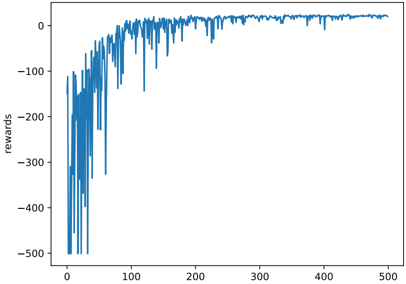


Figure: Reward curve of DoubleDQL algorithm.

Loss curves

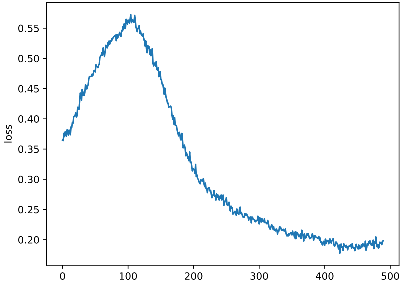


Figure: Loss of DQL algorithm.

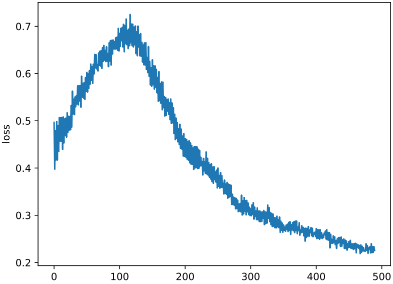


Figure: Loss curve of DoubleDQL algorithm.

References

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves and Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller (2013)
Playing Atari with Deep Reinforcement Learning

Deep Reinforcement Learning with Double Q-learning (2012)
Hado van Hasselt and Arthur Guez and David Silver (2015)

Thanks for listening!