Chapter 09 – DQN

Author: Gianmarco Scarano

gianmarcoscarano@gmail.com

1. Introduction

In order to introduce DQN, we must talk about Stochastic Gradient Descent with Experience Replay

Given an experience $D = \{(s_1,a_1,Q^{\pi}{}_1),(s_2,a_2,Q^{\pi}{}_2),\ldots,(s_n,a_n,Q^{\pi}{}_n)\}$

Repeat:

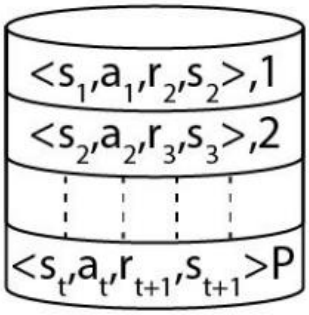
- Sample $(s,a,Q^{\pi}) \sim D$
- Apply the stochastic gradient descent update

$$\Delta w_t = \alpha(Q^{\pi} - Q(s,a,w))\ \nabla_w Q(s,a,w)$$

What we do here is that we sample a batch from the dataset, apply the SGD and loop this process.

TARGET = y

1.1 Replay Buffer

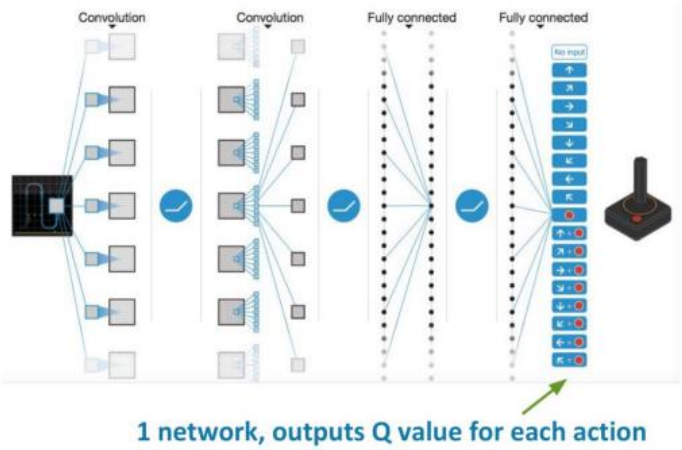


In most recent RL algorithms, a replay buffer is exploited which stores previous transitions experienced by an agent. These transitions are used later in time for training. The replay buffer is made up of two components:

- **Capacity**: Total number of transitions stored in the buffer
- **Age of transitions**: Number of gradient steps taken since the transition was generated

We use the Replay Buffer since we can collect real data costs and it's like having and efficient use of previous experience. Also, it has better convergence with function approximators.

2. Deep-Q Networks



s_1, a_1, r_2, s_2
s_2, a_2, r_3, s_3
s_3, a_3, r_4, s_4
...
$s_t, a_t, r_{t+1}, s_{t+1}$

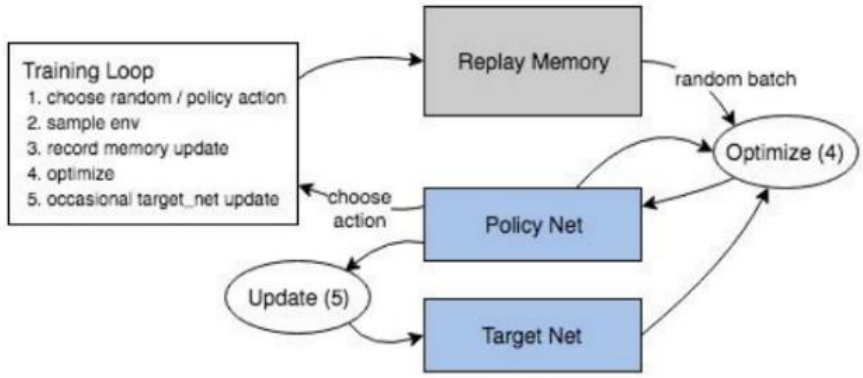
$\rightarrow \quad s, a, r, s'$

Deep-Q Networks were first used for Atari games. They use Convolutional Layers along with ReLU activation functions.

The input is a stack of last four frames (In order to get a bit of history of what has happened during the game), while the output instead are Q functions $Q(s,a)$ for 18 joystick buttons.

The DQN uses experience replay, meaning that the sampled tuple is used to compute the update targets.

In DQN, the target y is computed by using an estimation of itself and since the weights get updated on the next round, the target value also changes. To improve stability of the system, we have a specific target network use in the target calculation for multiple updates and fix its weights. After that we update such weights every once a while copying the Q-Network in it.



In this image below, we can see how this is happening. We use another network along with our network. Basically, we'll give the weights of the optimal network to the old network, which we'll use to do target calculation. We update the old network once in a while.

3. Double Q-Learning / Double Deep-Q Networks (DDQN)

Before introducing DDQN, we have to visualize the problem of maximization bias.

Greedy and epsilon greedy require a maximization step that can lead to a positive bias.

We can visualize the problem in this way:

- Consider a state where many actions have $Q(s,a) = 0$
- Estimated values are uncertain, some above and some below zero.
- The maximum of the true values is zero, but the max over estimates is positive (as stated before).

The problem is related to using same samples both to determine the max action and the estimate of its value. For this reason, we could have two independent estimates where one could be used to determine the max and the other could be used to do the estimation.

3.1 Double Q-Learning

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q_1(s,a)$ and $Q_2(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, such that $Q(terminal,\cdot) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S',a)\big) - Q_1(S,A)\Big)$
    else:
      $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S',a)\big) - Q_2(S,A)\Big)$
    $S \leftarrow S'$
  until S is terminal

Same exact pseudocode of Q-Learning, but with probability 0.5 the prediction is done using a different table (Q_2).

3.2 Double Deep-Q Networks

We simply extend what we have just seen in Double Q-Learning to DQN where we use our current Q-Network for action selection and use another network (older one, the target -> w^-) for action evaluation.

The update step then becomes:

$$\Delta \boldsymbol{w} = \alpha\Big(r + \gamma \overbrace{\hat{Q}\big(\underbrace{\arg\max_{a'} \hat{Q}(s', a'; \boldsymbol{w})}_{\text{Action selection: } \boldsymbol{w}}; \boldsymbol{w}^-\big)}^{\text{Action evaluation: } \boldsymbol{w}^-} - \hat{Q}(s, a; \boldsymbol{w})\Big)$$

4. Prioritized Experience Replay (PER)

The idea in the PER is that we want to replay more frequently transitions with high expected learning progress, as measured by the magnitude of their temporal-difference (*TD*) error.

Substantially, if the TD error in various transitions is very high, I want to give it more weight/priority.

A possible view on this is to directly store the magnitude of TD error in transitions, but this would require to keep all of them up-to-date (consider that Experience Replay has a LOT of data).

A possible solution, then, is to update items that are sampled during the updates (stochastic).

For doing so, we compute priorities p in 2 possible ways:

- Rank-based method: Sort items based on TD error magnitude $p_i = \frac{1}{rank(i)}$
- Proportional method: $p_i = |\delta_i| + \epsilon$
- For new samples, p is initialized as the max value

After this, we can compute the probability of sampling the i-th data-point as:

$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$ where α controls the level of prioritization (0 means none).

This system though, is a lot biased due to the fact that we are changing the same distribution as the one that we are using for our updates.

A possible fix for Bias correction then, is to update more certain transitions instead of others (called Importance sampling).

- Use importance sampling for updates
- If P(i) becomes 0, w becomes large
- If P(i) = 1/N, we have uniform sampling and w = 1
- Beta defines amount of correction

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

5. Few remarks on On-Policy / Off-Policy

Few chapters ago, we talked about on/off policy, defining them as:

- **On-policy** = Learn by what you do
- **Off-policy** = Learn by looking at someone else (by reusing experience, exploring, observing other agents etc.).

We can now apply Importance Sampling in Monte-Carlo where we want to use the returns from policy μ to evaluate our policy π.

Compute $G^{\pi/\mu}$ by multiplying **importance sampling corrections**

$$G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \cdots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t$$

The same thing applies for Temporal difference:

Consider the TD setting: we want to weight the TD target

$$\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}(R_{t+1} + \gamma V(S_{t+1}))$$

In both cases, we can see that $\pi(A_t|S_t)$ is equal to the policy we are trying to learn. So, basically, we want to try learning the desired policy by executing our actual policy that we have $= \mu(A_t|S_t)$.

This doesn't happen in Q-Learning since we don't want to sample another distribution due to the fact that we directly use the action from the target policy.