

Deep-Q-Learning

Sajad Safarveisi

December 2023

1 Introduction

Model-free reinforcement learning (RL) algorithms are employed in scenarios where the environmental model, including transition probabilities, is unavailable. Deep Q learning stands out as an exemplary algorithm in this category, utilizing a deep neural network known as a deep-Q network to estimate the action-value function, denoted as $Q(., .)$. Specifically, for each action a within the set of feasible actions for a given state s (referred to as \mathcal{A}), the objective is to approximate $Q(s, a)$. This tutorial delves into advanced DQN-based RL algorithms, elucidating their implementation's technical intricacies and considerations. It should be noted that these algorithms are called off-policy as the selection of actions at each state s is done randomly or via the action-value function Q that we want to approximate (no place for policy function).

We will also talk about policy gradient algorithms that are suitable for RL problems where the action space is continuous (usually is the case with real-world problems). These algorithms are on-policy since the action selection is based on the policy function.

2 DQN algorithms

The algorithms discussed herein are grounded in the Bellman optimality equation, expressed as:

$$Q^*(s, a) = \mathcal{R}(s, a) + \gamma \mathbb{E}[\max_{a'} Q^*(S_{t+1}, a') \mid S_t = s, A_t = a], \quad s \in \mathcal{N}, a \in \mathcal{A}, t = 0, 1, \dots, T-1, \quad (1)$$

where Q^* signifies the action-value function corresponding to the optimal policy π^* , $\mathcal{R}(s, a)$ is the expected reward upon selecting action a in state s , γ is the discount factor, \mathcal{N} denotes the set of non-terminal states, and T represents the budget. The optimal policy is distinctive in that it achieves the maximum value of Q for every state s and feasible action a . In practical scenarios, and owing to the model-free assumption inherent in learning algorithms discussed in this tutorial, Equation (1) can be expressed in the following manner:

$$Q(s, a; \omega) = r + \gamma \max_{a'} Q(s', a'; \omega)$$

In the context provided, $Q(., .; \omega)$, s , a , r , s' , and a' correspond to a function that approximates the Q^* (e.g., a deep neural network), the current state of the environment, the

selected action, the received reward, the next state from the environment, and the action chosen in the subsequent state, respectively. Subtracting the left-hand side of the recent equation from the right-hand side yields the temporal difference (TD), denoted as δ . This quantity, serving as a measure of approximation error, will be used in a cost function (e.g., Mean Squared Error) that is sought to be minimized during the learning process (training the agent) for each observed experience (s, a, r, s') , thereby converging towards Q^* .

$$\delta = r + \gamma \max_{a'} Q(s', a'; \omega) - Q(s, a; \omega) \quad (2)$$

To learn from each experience, we assess the output of $Q(s, a; \omega)$ against $r + \gamma \max_{a'} Q(s', a'; \omega)$, treating it as the target for evaluating the approximation error. Let L_ω represent the cost function; subsequently, we initiate a single step of the gradient descent algorithm by computing the gradient of L_ω concerning ω —the specific parameters of the Q network. In the literature, the Q network is interchangeably referred to as the policy network. Thus, we can rewrite (2) in the following manner:

$$\delta = r + \underbrace{\gamma \max_{a'} Q_{policy}(s', a'; \omega)}_{\text{target}} - \underbrace{Q_{policy}(s, a; \omega)}_{\text{approximation}} \quad (3)$$

In practical applications, it is important to highlight that every interaction with the environment, or experience, is typically stored in a buffer known as the experience replay memory. This enables the reuse of experiences in subsequent iterations. Without this mechanism, experiences would be discarded after each update to the parameters of the policy network, a practice that is undesirable as it significantly prolongs and diminishes the effectiveness of the learning process.

Before delving into the subsequent section, it is beneficial to outline the stages involved in constructing an experience within an online learning mode.

1. While in state s , select an action a based on the ϵ -greedy policy (selects the action with the highest Q_{policy} with probability $1 - \epsilon$ and uniformly at random one of the actions possible at state s with probability ϵ).
2. Receive the next state s' and the corresponding reward r from the environment.
3. Formulate the experience $e = (s, a, r, s')$

With memory in place, we can select a batch of experiences (how this is done depends on the algorithm) and perform batch learning which dramatically increases the likelihood of convergence due to much higher stability.

2.1 DQN

For this algorithm, a batch of experiences is chosen **uniformly at random** from the memory, and for each the TD error (see (3)) is evaluated before computing the cost function. We then perform one step of stochastic gradient descent. This will update the parameters (ω) of $Q_{policy}(\cdot, \cdot; \omega)$.

In practice, the Huber cost function is used due to its less sensitivity to outliers (high values of δ).



Drawback

The drawback of this approach lies in the non-fixed nature of the targets. With each update to the parameters of the policy network, the target is also updated, given that the two networks employed are identical (refer to the approximation and the target in (3)). This introduces a challenge, as in traditional supervised learning algorithms, a fixed set of pre-selected targets is expected. The continual adjustment of the target, synchronized with updates to the policy network parameters, poses a deviation from the typical supervised learning paradigm, potentially complicating the convergence and stability of the learning process.

To mitigate the aforementioned limitation, in (3), we replace Q_{policy} in the target with an alternative network having an identical structure to the policy network in the approximation. This newly introduced network is referred to as the *target* network.

$$\delta = r + \gamma \max_{a'} Q_{target}(s', a'; \omega_{target}) - Q_{policy}(s, a; \omega_{policy}) \quad (4)$$

The trick here is to update the parameters of the target network very slowly (soft update). To this end, we can use the following update formula

$$\omega_{target} = \omega_{policy} \tau + \omega_{target} (1 - \tau), \quad (5)$$

where τ is a very small positive number ($\tau \ll 1$). After each update on the parameters of the policy network, we use (5) to update the parameters of the target network. Next time we take a sample from the memory and the same experience shows up, its target is extremely close to its target for the previous replay (having fixed targets as expected for the supervised learning problem we are solving).



Drawback

The challenge with this approach arises from the likelihood of overestimating the expected returns Q towards the conclusion of the learning process. This overestimation is attributed to the maximum operation (max) in the target, which can potentially introduce an upward bias in the estimation of the action values.

2.2 Double DQN

To address the recent drawback, we can use the Double DQN (DDQN) algorithm.

$$\delta = r + \gamma Q_{target}(s', \arg\max_{a'} Q_{policy}(s', a'; \omega_{policy}); \omega_{target}) - Q_{policy}(s, a; \omega_{policy}) \quad (6)$$

From (6) we can see that the action to be selected for the next state s' is not based on the target network anymore, but on the policy network. For the next state, we will select an action with the maximum Q_{policy} . This decouples action selection and action evaluation (with no max operand) leading to a much lower likelihood of the overestimation of expected returns at the end of the learning process.

Note that similar to the DQN algorithm we take experiences uniformly at random from the experience memory.

2.3 Prioritized-experience-replay DDQN


In the antecedent algorithm, experiences in the memory were chosen in a uniformly random manner, whereby experiences were selected commensurate with their entry into memory. Additionally, certain experiences within the memory exhibit heightened significance due to their rarity. Under the prevailing uniform selection scheme, these crucial experiences are afforded an equivalent probability of being replayed, a circumstance deemed undesirable. Consequently, there arises a necessity for a mechanism to prioritize the selection of experiences for the forthcoming batch.

A method to prioritize selection (in a non-greedy manner) involves utilizing the temporal difference (TD) error of the experiences in the memory after the preceding iteration of batch learning. In the subsequent batch selection process, experiences are chosen in proportion to their updated absolute TD error ($|\delta_e|$). In mathematical terms, this can be expressed as follows:

$$P(e_i) = \frac{p_{e_i}}{\sum_{j=1}^M p_{e_j}}, \quad i = 1, 2, \dots, M, \quad (7)$$

$$p_{e_i} = |\delta_{e_i}| + \epsilon, \quad 0 < \epsilon \ll 1,$$

where $P(e_i)$ represents the probability of selecting the experience e_i for the upcoming batch, and M denotes the total number of experiences in the memory. The parameter ϵ ensures a nonzero probability of selection even for experiences with zero TD error.



p_e for newly added experiences

When an experience is added to the memory ($M \leftarrow e_{new}$), it lacks an associated TD-error. Consequently, these experiences are endowed with the maximum TD error observed among experiences already in the memory.

$$p_{e_{new}} = \max_{e \in M} |\delta_e| + \epsilon,$$



A note on batch size

Choose the batch size in consideration of the memory capacity to ensure that experiences in the memory are replayed sufficiently. Prioritized sampling specifically focuses on selecting experiences from memory. If the batch size is excessively large, it becomes ineffective as experiences might be discarded too soon due to reaching the memory's maximum capacity, leading to the overwriting of old experiences.

2.4 Dueling DDQN

In contrast to the previously mentioned approaches, this method delves into the architectural aspects of the problem. The core concept involves implementing a deep neural network, specifically a policy network, with two branches or sub-networks. These branches serve the purpose of estimating action values: one is dedicated to assessing the state value function, denoted as $V(s)$, while the other focuses on evaluating action advantages for a given input state. Action advantage, defined as the surplus value gained by selecting a particular action in comparison to the average action values across all possible actions ($Q(s, a)$ for different actions a in state s), is expressed mathematically as follows (assuming that \mathcal{E} is the state space),

$$A(s, a) = Q(s, a) - V(s) \quad s \in \mathcal{E}, a \in \mathcal{A}$$

From the equation above, we have

$$Q(s, a) = V(s) + A(s, a)$$

Since we are estimating the functions above (using a deep neural network), we can rewrite the recent equation as

$$Q(s, a; \theta; \alpha; \beta) = V(s; \theta; \alpha) + A(s, a; \theta; \beta), \quad (8)$$

where θ , α , and β correspond to the parameters for the deep neural network in the shared layers (between the two branches), in the layers for the estimation of the state value function, and in the layers for the estimation of the advantages. To enhance optimization stability, an adjusted version of the equation (8) is proposed:

$$Q(s, a; \theta; \alpha; \beta) = V(s; \theta; \alpha) + \left(A(s, a; \theta; \beta) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta; \beta) \right) \quad (9)$$

It's noteworthy that the introduced alterations in architecture do not preclude the application of previously discussed algorithms. Despite the modifications, the network output still comprises Q values for actions, rendering it compatible with existing model-free reinforcement learning algorithms.

2.5 Policy gradient algorithms

When dealing with a vast action space, especially one that is continuous or characterized by high cardinality, or when confronted with high-dimensional challenges, value function-based algorithms prove ineffective. In such scenarios, the process of selecting an action by refining a policy through updating the Q -value function becomes impractical. Policy gradient algorithms (referred to as PG), on the other hand, emerge as effective solutions. In this context, we directly approximate a stochastic policy ($\pi(s, a, \theta)$), offering a notable advantage in natural exploration. Furthermore, PG excels in discovering the optimal stochastic policy. This distinction becomes particularly relevant in the realm of Markov Decision Processes (MDPs). While MDPs inherently possess an optimal deterministic policy, real-world applications often involve partially-observable MDPs, where the set of optimal policies may exclusively consist of stochastic alternative.

The main disadvantage of PG algorithms is that because they are based on gradient ascent, they typically converge to a local optimum whereas value function-based algorithms converge to a global optimum. Furthermore, the policy evaluation (computing $V(s)$ following the policy π) of PG is typically inefficient and can have high variance (G_t in the REINFORCE algorithm). To alleviate this drawback to some extent, Actor-Critic algorithms can be employed, or the introduction of the Baseline function in their update schema can be considered.