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 $\mathcal{A}(s)$), the objective is to approximate $Q(s,a)$. This tutorial delves into advanced DQN-based RL algorithms, elucidating the technical intricacies and considerations involved in their implementation.

2 DQN algorithms

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

$$Q^*(S_t, A_t) = R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') \quad t = 0, 1, \ldots, T-1, \tag{1}$$

where T represents the budget, and Q^* signifies the action-value function corresponding to the optimal policy π^*. The optimal policy is a unique policy where the maximum of Q is attained for each state S and selectable action A. It is important to note that the use of S, A, and R emphasizes the stochastic nature of the RL problem, although this is not universally applicable to all problems. Subtracting the left-hand side of the equation from the right-hand side yields the temporal difference (TD), denoted as δ_t. 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 $(S_t, A_t, R_{t+1}, S_{t+1})$, thereby converging towards Q^*.

$$\delta_t = R_t + \gamma \max_{a'} Q(S_{t+1}, a'; \omega) - Q(S_t, A_t; \omega) \quad t = 0, 1, \ldots, T-1, \tag{2}$$

To learn from each experience, we assess the output of $Q(S_t, A_t; \omega)$ against $R_t + \gamma \max_{a'} Q(S_{t+1}, a'; \omega)$, 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 with respect to 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:

$$\delta_t = \underbrace{R_t + \gamma \max_{a'} Q_{policy}(S_{t+1}, a'; \omega)}_{target} - \underbrace{Q_{policy}(S_t, A_t; \omega)}_{approximation} \tag{3}$$

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_t, select an action A_t 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_t with probability ϵ).

2. Receive the next state S_{t+1} and the corresponding reward R_{t+1} from the environment.

3. Formulate the experience $e_t = (S_t, A_t, R_{t+1}, S_{t+1})$

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

2.1 DQN

For this algorithm, a batch of experiences are 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 $Q_{policy}(., .; \omega)$.
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.

$$\delta_t = R_t + \gamma \max_{a'} Q_{target}(S_{t+1}, a'; \omega_{target}) - Q_{policy}(S_t, A_t; \omega_{policy}) \tag{4}$$

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

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

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.

$$\begin{aligned}\delta_t = R_t + \gamma\, Q_{target}(S_{t+1}, \operatorname*{argmax}_{a'} Q_{policy}(S_{t+1}, a'; \omega_{policy}); \omega_{target}) \\ - Q_{policy}(S_t, A_t; \omega_{policy})\end{aligned} \tag{6}$$

From (6) we can see that the action to be selected for the next state S_{t+1} is not based on the target network anymore, but based 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 much lower likelihood for 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 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:

$$\begin{aligned} P(e_i) &= \frac{p_{e_i}}{\sum_{j=1}^{M} p_{e_j}}, \quad i = 1, 2, \ldots, M, \\ p_{e_i} &= |\delta_{e_i}| + \epsilon, \quad 0 < \epsilon \ll 1, \end{aligned} \tag{7}$$

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 that are already in the memory.

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