Deep Q Networks

March 4, 2025

1 Value-Based Methods Motivation

Motivation for usind Value-Based Methods is the following: If we happen to know the values of all of the states, then we can easily derive the optimal policy. This is because the optimal policy is to select the action that maximizes the expected value of the next state. Therefore there is no need in separate policy network, as it all comes down to the argmax over actions.

2 Supporting Theory

2.1 Definitions

2.1.1 Markov Decision Process (MDP)

A Markov Decision Process (MDP) is defined as a tuple:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \gamma)$$

where:

- $\mathcal{S}$ is the state space.
- $\mathcal{A}$ is the action space.
- $P(s' \mid s, a)$ is the transition probability function, which defines the probability of transitioning to state $s' \in \mathcal{S}$ given the current state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$.
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, which gives the expected reward received after taking action a in state s.
- $\gamma \in [0, 1]$ is the discount factor, which determines how much future rewards are valued.

2.1.2 Policy

A policy π defines the agent's behavior and is a mapping:

$$\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$$

where $\pi(a \mid s)$ is the probability of selecting action a in state s. A policy can be:

- **Deterministic:** $\pi(s) = a$, meaning a fixed action is chosen in each state.
- **Stochastic:** $\pi(a \mid s)$ defines a probability distribution over actions.

2.1.3 State Value Function

The value function of a policy π is defined as:

$$V^\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(S_t, A_t) \mid S_0 = s \right].$$

This represents the expected cumulative discounted reward starting from state s and following policy π.

2.1.4 Action Value Function (Q-function)

The action-value function (Q-function) is defined as:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t r(S_t, A_t) \mid S_0 = s, A_0 = a \right].$$

This represents the expected cumulative reward when taking action a in state s and following policy π thereafter.

2.2 Dynamic Programming

We can estimate the $V^\pi(s,a)$ and $Q^\pi(s,a)$ using the following iterative process from a bootstrapped update, according to the Bellman equation:

$$V_{k+1}^\pi(s) \leftarrow \mathbb{E}_{a \sim \pi(a|s)} \left[r(s,a) + \gamma \mathbb{E}_{s' \sim P(s'|s,a)} \left[V_k^\pi(s') \right] \right]$$

or just:

$$V_{k+1}^\pi(s) \leftarrow r(s,a) + \gamma \mathbb{E}_{s' \sim P(s'|s,a)} \left[V_k^\pi(s') \right]$$

And due to the fact that Q function also follows the Bellman equation, we have:

$$Q_{k+1}^\pi(s,a) \leftarrow \mathbb{E}_{s' \sim P(s'|s,a)} \left[r(s,a) + \gamma \mathbb{E}_{a' \sim \pi(a'|s')} \left[Q_k^\pi(s',a') \right] \right]$$

And the simplified version, with the argmax deterministic policy:

$$Q_{k+1}^\pi(s,a) \leftarrow r(s,a) + \gamma \mathbb{E}_{s' \sim P(s'|s,a)} \left[\max_{a'} Q_k^\pi(s',a') \right]$$

2.3 Policy Iteration

Policy iteration is an algorithm that alternates between policy evaluation and policy improvement. In the policy evaluation step, the value function is computed for the current policy. In the policy improvement step, the policy is updated to be greedy with respect to the value function. This process is repeated until convergence.

1. **Policy Evaluation:** Compute the Advantages (or Q-functions) $A^\pi(s,a)$ for the current policy π.

2. **Policy Improvement:** Compute the new policy π' by acting greedily with respect to the Advantages (or Q-functions) $A^\pi(s,a)$.

Convergence: Repeat until the policy converges.

This serves as a Motivation for the following algorithm:

2.4 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm to learn the action-value function $Q(s, a)$ directly. The algorithm is based on the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a) \right]$$

where:

- $Q(s, a)$ is the action-value function.
- α is the learning rate.
- r is the reward received after taking action a in state s.
- γ is the discount factor.
- s' is the next state.

The difference between Q-Learning and Policy Iteration is that Q-Learning does not require a model of the environment, and it learns the optimal policy directly from the interaction with the environment. The full algorithm is as follows:

1. Initialize the Q-function $Q(s, a)$.
2. Repeat for each episode:
   (a) Initialize the state s.
   (b) Repeat for each step of the episode:
      i. Select an action a using an exploration strategy.
      ii. Take action a and observe the reward r and the next state s'.
      iii. Update the Q-function using the Q-Learning update rule.
      iv. Set the state s to the next state s'.

It only needs to get diverse samples from the environment, and it will converge to the optimal policy. Therefore we introduce another concept:

2.5 Exploration vs Exploitation

Exploration is the process of trying out different actions to discover the environment. Exploitation is the process of selecting the best-known action to maximize the reward. The trade-off between exploration and exploitation is a fundamental problem in reinforcement learning.

But why do we need it? Well, we can‘t just use the greedy method(exploitation), because we will get stuck in a local minimum, by not exploring new strategies. On the other hand, if we only explore, it will be exponentially hard to find the optimal policy. Therefore, we need to balance between the two.

2.5.1 Epsilon-Greedy Strategy

The ϵ-greedy strategy is a simple method to balance exploration and exploitation. With probability ϵ, a random action is selected, and with probability $1 - \epsilon$, the action with the highest Q-value is selected.

$$\pi(a \mid s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}$$

2.5.2 Other ways of exploration

- **Softmax Exploration:** The probability of selecting an action is proportional to the exponentiated Q-value.
- **Boltzmann Exploration:** Similar to softmax exploration, but with a temperature parameter that controls the randomness of the selection.
- **Noisy Nets:** Add noise to the weights of the neural network to encourage exploration.
- **Bootstrapped DQN:** Train multiple Q-networks in parallel with different random initializations to encourage exploration.
-

We will discuss some of these more advanced exploration strategies in more detail in the future sections.

3 Deep Q Networks

3.1 Introduction

But there are obviously many issues with the tabular representation of state-action pairs. If either gets really large, the Q-table will be too large to store in memory. Therefore, we need to approximate the Q-function with a function approximator. This is where Deep Q Networks come in.
We can approximate the Q-function with a neural network, using gradient descend. We define the loss function J, as the mean squared error between the predicted Q-value and the target Q-value:

$$J(\theta) = \mathbb{E}_{s,a,r,s'} \left[\left(Q(s,a;\theta) - (r + \gamma \max_{a'} Q(s',a';\theta)) \right)^2 \right]$$

In order to perform the gradient descend we need to find the derivative of the loss function with respect to the weights of the neural network:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s,a,r,s'} \left[\nabla_\theta Q(s,a;\theta) \left(Q(s,a;\theta) - (r + \gamma \max_{a'} Q(s',a';\theta)) \right) \right]$$

The gradient descend:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta) = \theta - \alpha \mathbb{E}_{s,a,r,s'} \left[\nabla_\theta Q(s,a;\theta) \left(Q(s,a;\theta) - (r + \gamma \max_{a'} Q(s',a';\theta)) \right) \right]$$

Unfortunately, unlike in a tabular case, there is no guarantee of convergence, but in practice, it works well. We will discuss some of the improvements and solutions in the following sections.

We introduce some additional concepts:

3.2 Experience Replay

If we keep training the neural network on a consecutive samples, we will be overfitting to them, as they are highly correlated. Hence we may never generalize well to basically anything. Therefore we introduce a concept of experience replay.

Experience replay is a technique that stores the agent's experiences in a replay buffer and samples a batch of experiences to update the Q-network. This has several benefits:

- **Breaks Correlation:** The data is sampled uniformly from the replay buffer, which breaks the correlation between consecutive samples.
- **Reuses Data:** The data is reused multiple times, which helps improve the data efficiency.

The second point, where we reuse data is one of the key features in DQN-like algorithms. This follows from the fact, that at no point do we use that we need to use actions from the current policy. This means we can use arbitrary samples collected by any given policy, or old versions of ourselves. Lastly, we can reuse the data multiple times, making the algorithm very sample efficient.

3.3 Target Networks

Another issue with DQN-s lies in the way we implement the gradient descend.

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta) = \theta - \alpha \mathbb{E}_{s,a,r,s'} \left[\nabla_\theta Q(s,a;\theta) \left(Q(s,a;\theta) - (r + \gamma \max_{a'} Q(s',a';\theta)) \right) \right]$$

Well we can check, that this algorithm is not quite a gradient descend algorithm, as we dont backpropagate trough our targets:

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

This is a problem, because our targets are moving. Therefore we attempt to decorrelate the targets from the Q-network by introducing a target network. The target network is a copy of the Q-network that is updated less frequently. The target network is used to compute the target Q-values for the loss function, while the Q-network is used to compute the predicted Q-values.

$$Targets : y_i = r + \gamma \max_{a'} Q(s',a';\theta_{target})$$

Gradient descend step:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta) = \theta - \alpha \mathbb{E}_{s,a,r,s'} \left[\nabla_\theta Q(s,a;\theta) \left(Q(s,a;\theta) - y_i \right) \right]$$

This way we can stabilize the training process, and hopefully make it converge.

3.4 Standard DQN algorithm

Here we provide the pseudocode for the standard DQN algorithm:

Algorithm 1 Basic Deep Q-Learning Algorithm

```
Initialize Q-network $Q_\theta$ with random weights $\theta$
Initialize target Q-network $Q_{\theta'}$ with weights $\theta' = \theta$
Initialize experience replay buffer $\mathcal{D}$
for each episode do
    Initialize state $s$
    for each time step $t$ do
        Select action $a_t$ using an $\epsilon$-greedy policy:
            $$a_t = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q_\theta(s_t, a), & \text{otherwise} \end{cases}$$
        Execute $a_t$, observe reward $r_t$ and next state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $\mathcal{D}$
        Sample a minibatch of transitions $(s_j, a_j, r_j, s'_j)$ from $\mathcal{D}$
        Compute target Q-value using target network:
            $$y_j = \begin{cases} r_j, & \text{if } s'_j \text{ is terminal} \\ r_j + \gamma \max_{a'} Q_{\theta'}(s'_j, a'), & \text{otherwise} \end{cases}$$
        Update Q-network by minimizing loss:
            $$L(\theta) = \frac{1}{N} \sum_j (y_j - Q_\theta(s_j, a_j))^2$$
        Perform gradient descent step on $L(\theta)$ with respect to $\theta$
        Periodically update target network: $\theta' \leftarrow \theta$
    end for
end for
```

4 Extensions to the DQN Algorithm

4.1 Double DQN

If we were to look at the Q-value estimates of our model, we would see, that there is a tendency to greatly overestimate the q values. The source of this lies in the way we estimate the Q-targets recursivelly:

$$y_j = r_j + \gamma \max_{a'} Q_{\theta'}(s'_j, a')$$

The max operator tends to overestimate the Q-values, as it also accounts for a positive noise in our training. What do we do know, is we select the greatest Q value from the ones generated by our target network.
The way we adress this issue is by introducing a second Q-network, that is used to select the action, and the first one is used to evaluate the Q-value of that action. This way we can reduce the overestimation of the Q-values, as we decorrelate the action selection and the q-value evaluation.
Now our targets look like this:

$$y_{Aj} = r_j + \gamma Q_{\theta^B}(s'_j, \arg\max_{a'} Q_{\theta^A}(s'_j, a'))$$

$$y_{Bj} = r_j + \gamma Q_{\theta^A}(s'_j, \arg\max_{a'} Q_{\theta^B}(s'_j, a'))$$

In reality, we dont need to introduce any new networks, as we already have the behavior and target networks. Although in such case there appears some correlation, it works well enough when updating target network less frequently.
Our final targets look as follows:

$$y_j = r_j + \gamma Q_{\theta_{target}}(s'_j, \arg\max_{a'} Q_{\theta}(s'_j, a'))$$

4.2 Prioritized Experience Replay

Another improvement can be done in how we utilize our samples. There is always some redundant data present in our dataset, and we would like to study some unusual moments better. One idea is to introduce the Prioritized Experience Replay.
The idea is to sample the data with a probability proportional to the TD-error, which is defined as follows:

$$\delta_j = y_j - Q_\theta(s_j, a_j)$$

We assign each sample a probability:

$$P(j) = \frac{p_j^\alpha}{\sum_k p_k^\alpha}$$

where the priority p_j is defined as:

$$p_j = |\delta_j| + \epsilon$$

The hyperparameter α controls the level of prioritization, with $\alpha = 0$ corresponding to uniform sampling. The hyperparameter ϵ is a small positive constant that ensures that no transition has zero priority.
In execution we now sample the data according to the new probability distribution.
Now as we are not sampling uniformly, we get some bias in our updates, therefore we need to introduce importance sampling weights:

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

where the hyperparameter β controls the level of importance sampling, with $\beta = 1$ corresponding to full importance sampling. The weights are normalized by:

$$w_j = \frac{w_j}{\max_k w_k}$$

So our final update rule for each sample looks like this:

$$\theta \leftarrow \theta - \alpha w_j \nabla_\theta J(\theta)$$

It is useful to anneal the β parameter over time, to reduce the bias in the early stages of training. In later stages, we can increase the β parameter to focus more on the high-priority samples.