Introduction to Q-learning

Niranjani Prasad, Gregory Gundersen

19 October 2017

1 Big Picture

1. MDP notation
2. Policy gradient methods $\rightarrow$ Q-learning
3. Q-learning
4. Neural fitted Q iteration (NFQ)
5. Deep Q-network (DQN)

2 MDP Notation

- $s \in S$, a set of states.
- $a \in A$, a set of actions.
- π, a policy for deciding on an action given a state.
  - $\pi(s) = a$, a deterministic policy. Q-learning is deterministic. Might need to use some form of ϵ-greedy methods to avoid local minima.
  - $\pi(a|s) = (0, 1]$, a stochastic policy. Policy gradient methods are stochastic.
- $r(s, a)$, a reward for action a on state s. Often probabilistic.
- $\gamma \in [0, 1]$, a discount factor. Captures the intuition that future rewards are worth less. Exponential decay.
- $R^\pi = \mathbb{E}_\pi[\sum_t^T \gamma^t r(s_t, a_t)]$, the return or expected discounted future reward given π. Expectation captures the transition function $T(s, a) = \mathbb{P}(s'|s, a)$ that you've seen with MDPs.

3 Policy gradient methods → Q-learning

- Model-based learning: explicitly learn some transition function $T(s,a) = \mathbb{P}(s'|s,a)$ and some reward function $r(s,a)$.
- Model-free learning:
  - Policy gradient methods: just learn mapping $F : s \to a$. Don't care about estimating transitions or rewards.
  - Q-learning: $F : \langle s,a \rangle \to Q(s,a)$. Learn some Q-function that computes a Q-value for every state-action pair.

4 Q-learning: Definitions and Algorithm

4.1 Definitions

- **Action value** - "Value", i.e. expected discounted return, of a given state action pair (s,a), following behaviour policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{t} \gamma^{t} r_t | s_t = s, a_t = a\right]$$

- **State value** - Value of a given state s, following behaviour policy π:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{t} \gamma^{t} r_t | s_t = s\right] = \max_{a'} Q^{\pi}(s,a')$$

4.2 Algorithm

The objective of Q-learning is to learn an optimal policy π^* such that $Q^{\pi^*}(s,a) = max_{\pi} Q^{\pi}(s,a)\ \forall s,a$, that maximises this expected return. We can evaluate Q for the optimal policy, by using the following recursive relationship (the Bellman equation):

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[r_t + \gamma \max_{a'} Q(s',a')\right]$$

i.e. the Q-value of the current state-action pair is given by the immediate reward plus the expected value of the next state. Given sample transitions $\langle s,a,r,s' \rangle$, Q-learning leverages the Bellman equation to iteratively learn as estimate of Q, as shown in Algorithm 1. The first paper presents proof that this converges given all state-actions pairs are continually seen.

Q-learning in its simplest form is dealing with discrete state and action spaces. In order to generalize to continuous state spaces, we need for function approximator that takes as input some vector representation of the state, and maps to an action-value...

Algorithm 1 Q-learning

Initialize $\hat{Q}(s,a) = 0 \ \forall s, a$
Observe initial state $s = s_0$
repeat
(1) Choose action a (following some exploratory policy)
(2) Observe reward r, new state s'
(3) Update $\hat{Q}$ as a convex sum of old, new estimates:
$\hat{Q}(s,a) = (1-\alpha)\hat{Q} + \alpha[r + \gamma \max_{a'} Q(s',a')]$ where α is learning rate
until convergence
Optimal policy: $\pi^*(s) = \max_a \hat{Q}(s,a)$

5 Neural fitted Q iteration (NFQ)

- We have a continuous state-space and want to regress on a loss function.
- Regress a loss function on-line

$$\mathcal{L}(s,a) = (\overbrace{\hat{Q}(s,a)}^{A} - \overbrace{(r(s,a) + \gamma \max_b \hat{Q}(s',b))}^{B})^2$$

  - A, our current estimate of the Q-function
  - B, our current reward plus the best we can do with our current Q-function

- Problem: disproportionate effect estimate or overgeneralization.
  - Intuition: A mouse is in a maze with cheese in two corners; one corner with cheese also has a mouse trap. If the mouse does a "parameter update" after seeing just the cheese, it might incorrectly undo some of what it learned about cheese in the context of mouse traps. It generalizes too quickly.
  - *Quote from NFQ paper* A weight change induced by an update in a certain part of the state space might influence the values in arbitrary other regions - and therefore destroy the effort done so far in other regions. This leads to typically very long learning times or even to the final failure of learning at all.
- NFQ solution
  - Collect experiences D
  - Train off-line in batches

5.1 NFQ algorithm

Algorithm 2 NFQ

Input a collection of experiences, $D = \{(s, a, r, s')\}_{i=1,\ldots,m}$
Initialize $\hat{Q}_0$
Initialize $k = 1$
repeat
(1) Create training set (X, Y) where:

- $X = \{(s, a)\}_{i=1,\ldots,m}$
- $Y = \{r + \gamma \max_b \hat{Q}_k(s', b)\}_{i=1,\ldots,m}$

(2) Regress a brand new neural network $\hat{Q}_{k+1}$ on (X, Y)
until convergence
Output Q_K

6 Deep Q-Networks

Neural fitted Q provides one way of dealing with sources of instability in learning a big non-linear function approximator for Q, but as it stands, it requires us to retrain neural networks from scratch at each iteration, which is infeasible for more complex architectures, like the deep convnets used for learning to play Atari games, for example. The third paper instead proposes a 'minibatch' approach, with the following key deviations from usual approximate Q-learning:

- **Experience Replay** which randomizes over the data, removing correlations in the observation sequence and smoothing changes in the data distribution: collect experience tuples $e_t = \langle s_t, a_t, r_t, s_{t+1} \rangle$ in some data buffer D; draw a minibatch of samples $s, a, r, s' \sim U(D)$ from this buffer; apply Q-learning update to this minibatch by running gradient descent with the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{e \sim U(D)} \left[(r + \gamma \max_{a'} Q(s', a'; \theta_i^t) - Q(s, a; \theta_i))^2 \right]$$

- **Periodic parameter updates**: The above loss function highlights another key difference - DQN maintains two separate networks: a target network with parameters θ^t, the actual Q-network with parameters θ, which are copied over to the target network only every t steps, in order to reduce temporal correlations between the Q-value used in action evaluation and in the target.

- **Network architecture**: Finally, while traditional Q-function approximators take as input the state and action, and output Q-value; the DQN takes just state as input, and outputs a vector of Q-values for each possible action, as shown in Figure 1. This speeds up training by allowing us to get Q-values for all possible actions with a single forward pass through the network.

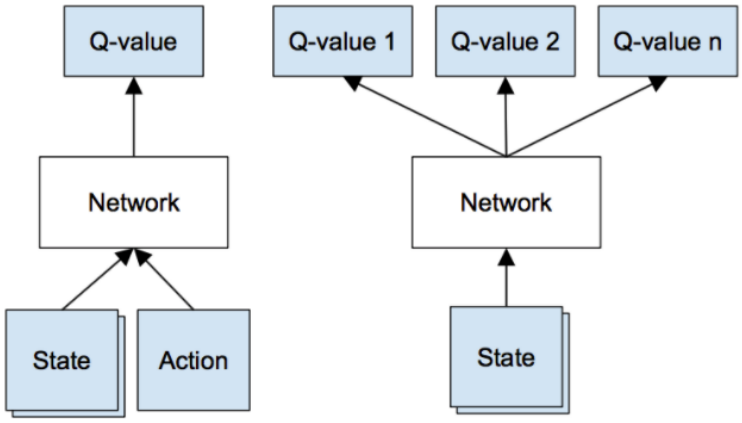

Figure 3: *Left:* Naive formulation of deep Q-network. *Right:* More optimized architecture of deep Q-network, used in DeepMind paper.

Figure 1: DQN Architecture (Source)

Some interesting extensions to the DQN include:

- Double Q-learning: tries to reduce optimism in Q-estimates by decoupling action selection and evaluation.
- Dueling Architectures: learns state value and advantages for actions separately, for better policy evaluation in the presence of many similar-valued actions
- Deterministic policy gradients (DPG): actor-critic extension for handling continuous action spaces.