

6.036 Spring 2018 : Week 9

March 11, 2018

10 Reinforcement learning

So far we have tried to solve MDPs when everything is known, states, actions, transition probabilities, as well as rewards. Here we consider learning to act when only the states and actions are known, along with the discount factor which is for us to choose in any case. We continue to assume that there is an underlying MDP that the robot is interacting with, it's just not known to the robot.

What can the robot sense? We assume that it can always tell which state it is in, and what action it is taking. Once the transition takes place, it can also sense the new state as well as the associated immediate reward but only corresponding to the action actually taken. Thus, as the robot interacts in the world, it will generate experience in the form of $\{(s, a, s'), R(s, a, s')\}$ that it can use in one of two different ways. It can try to construct the underlying MDP and subsequently solve it as discussed earlier. To this end, it would have to use the experience to construct estimates of $T(s, a, s')$ and $R(s, a, s')$. An alternative, more direct strategy, is to try to use the experience to update current estimate of $Q(s, a)$, bypassing the problem of estimating full transition probabilities.

10.1 Model-based learning

As the robot collects information about transitions involving state s and action a and the next state s', it can estimate T and R as follows

$$\hat{T}(s, a, s') = \frac{count(s, a, s')}{\sum_{s'} count(s, a, s')} \quad (1)$$

$$\hat{R}(s, a, s') = \frac{\sum_t R_t(s, a, s')}{count(s, a, s')} \quad (2)$$

where $R_t(s, a, s')$ is the t^{th} realization of the reward we observed when starting in state s, taking action a, and transitioning to s'. If the reward is noisy, observed reward values $R_t()$ may vary from one instance to another. The above estimate is simply the average of all observations of $R(s, a, s')$. In practice, this MDP construction strategy is quite ineffective for any non-trivial state space. We would have to explore randomly, taking actions and moving from one state to another. Most likely, this strategy would preclude us from reaching different portions of complex state space. Of course, one could imagine (and people have proposed) more refined exploration strategies, guided by the estimates produced so far, and

diverting resources to states and actions that we do not yet know consequences of. The other problem with this strategy is that while we typically have a small number of possible actions, the state space tends to be very large. Thus the problem of estimating the full set of transition probabilities is computationally quite expensive, requiring $\mathcal{O}(|S|^2|A|)$ for storage alone.

10.2 Model-free reinforcement learning

Recall the Q-value iteration algorithm:

$$Q^*_{k+1}(s,a) = \sum_{s' \in S} T(s,a,s')[R(s,a,s') + \gamma \max_{a' \in A} Q^*_k(s',a')] \tag{3}$$

where, after each iteration, we obtain better estimates of the values in the sense that they are closer to $Q^*(s,a)$. If $Q^*_k(s,a)$ is already close to $Q^*(s,a)$ so will the associated policies. In other words, $\pi^*_k(s) = \text{argmax}_{a \in A} Q^*_k(s,a)$ will be close to the optimal policy $\pi^*(s) = \text{argmax}_{a \in A} Q^*(s,a)$, not necessarily in terms of the selected actions, but in terms of the discounted rewards that those actions will imply.

Our goal is to approximate the Q-value iteration without constructing an estimate of the full transition probability matrix. To this end, suppose we visit the same state s a few times and some of those times take an action a. Let the associated next state observations be $s'_1, \ldots, s'_N$. So, we can construct

$$\begin{gathered} sample_1 : R(s,a,s'_1) + \gamma \max_{a' \in A} Q_k(s'_1, a') \\ sample_2 : R(s,a,s'_2) + \gamma \max_{a' \in A} Q_k(s'_2, a') \\ \ldots \\ sample_k : R(s,a,s'_N) + \gamma \max_{a' \in A} Q_k(s'_N, a') \end{gathered}$$

By averaging these samples, we would obtain an estimate of the Q-value update

$$Q_{k+1}(s,a) = \frac{1}{N} \sum_{l=1}^{N} \left[R(s,a,s'_l) + \gamma \max_{a' \in A} Q_k(s'_l, a') \right] \tag{4}$$

For large N, this update would be very close to the Q-value iteration. The only difference between this and storing an estimate of $T(s,a,s')$ is that we simply use the samples to update Q-values without storing the samples themselves. At one extreme, we can just use a single sampled experience (s,a,s') but, instead of fully updating the Q-values, we will nudge them a bit in the direction of this stochastic update (cf. stochastic gradient):

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left[R(s,a,s') + \gamma \max_{a' \in A} Q(s',a') \right] \tag{5}$$

$$= Q(s,a) + \alpha \left[R(s,a,s') + \gamma \max_{a' \in A} Q(s',a') - Q(s,a) \right] \tag{6}$$

where α takes the role of the learning rate. For small values of α, we end up averaging many sampled updates before changing $Q(s,a)$ significantly, thus moving in the direction

of the full Q-value iteration update. Note that we dropped the subindex k as we no longer perform full Q-value iterations. Also, only $Q(s, a)$ corresponding to the experience (s, a, s') is updated, not Q-values for all states and actions. This asynchronous algorithm that follows the experience generated as the robot interacts in the world is known as the *Q-learning algorithm.*

Q-learning Algorithm

- Collect experiences $\{(s, a, s'), R(s, a, s')\}$ by interacting in the world
- Update the single Q-value corresponding to each such experience:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left[R(s, a, s') + \gamma \max_{a' \in A} Q(s', a') \right] \tag{7}$$

Note that, during the iterations of the algorithm, likely states, including next states s', contribute more often to the Q-values estimates. As the algorithm progresses, old estimates fade, making the Q-value more consistent with the recent experience.

The Q-learning algorithm closely resembles stochastic gradient decent updates. In fact, it has similar convergence conditions as the gradient ascent algorithm. Each sample corresponds to (s, a), i.e., being in state s and taking action a. We can assign a separate learning rate for each such case, i.e., α is now $\alpha_k(s, a)$, where k is the number of times that we saw (s, a) as part of the experience. Then, in order to ensure convergence of the Q-values, we should have

$$\sum_k \alpha_k(s, a) \rightarrow \infty, \quad \sum_k \alpha_k^2(s, a) < \infty$$

Of course, these conditions merely state that we should continue repeatedly visiting (s, a) so as to collect enough experience to update the associated Q-value. If the learning rate is too large, we will not converge to the average as required by Q-value iteration and instead focus too much on the most recent experience. If the learning rate is too small, we will run out of steam to reach the Q-value iteration updates.

Exploration/Exploitation Trade-Off Note that the Q-learning algorithm does not specify how we should interact in the world so as to learn quickly. It merely updates the values based on the experience collected. If we explore randomly, i.e., always select actions at random out of A, we would most likely not get anywhere. Imagine how long it would take for a robot that moves in a random direction in each step to reach a goal state on a large complex environment. A better option is to exploit what we have already learned, as summarized by current Q-values. We can always act *greedily* with respect to the current estimates, i.e., take an action $\pi(s) = \text{argmax}_{a \in A} Q(s, a)$. Of course, early on, these are not necessarily very good actions. For this reason, a typical exploration strategy is to follow a so-called ϵ-greedy policy:

$$\begin{aligned} &\text{with probability } \epsilon \text{ take a random action out of } A \\ &\text{with probability } 1 - \epsilon \text{ follow } \pi(s) = \text{argmax}_{a \in A} Q(s, a) \end{aligned}$$

The value of ϵ here balances exploration vs exploitation. A large value of ϵ means exploring more (randomly), not using much of what we have learned. A small ϵ, on the other hand, will generate experience consistent with the current estimates of Q-values.

10.3 Deep Q-learning

Recall our formulation for reinforcement learning tasks. We have an agent such as a robot that wishes to act in the world in a manner that maximizes the sum of discounted rewards it receives in response to actions. The agent can reliably sense the state of the world s (e.g., its location), knows the set of possible states S it can be in (has a map of possible locations), and is aware of the set of possible actions A available to it at each step (the set may vary depending on where the agent is). After taking an action $a \in A$ in state s, the state will change to a new state s' (e.g., robot moves to a different location sensed as s'). Following the action, and the new state, the agent receives a one-step reward $R(s, a, s')$. By exploring different actions in states, the agent will create *experiences* or snippets involving s, a, s' and the associated (potentially random) reward $R(s, a, s')$. The agent may choose to use these experiences in many ways. For example, it can build a model of the underlying MDP (such as transition probabilities) and solve the problem afterwards with Q-value iteration. In a direct, model free approach, in contrast, the agent never builds a model of the world but aims to directly learn a look-up table for Q-values. These Q-values are updated based on experience snippets according to

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[\overbrace{R(s,a,s') + \gamma \max_{a' \in A} Q(s',a')}^{\text{stochastic look-ahead estimate}} - \overbrace{Q(s,a)}^{\text{old value}} \right] \tag{8}$$

where α is the learning rate that should be decreased as the learning progresses. Note that the Q-learning algorithm only updates the state-action value corresponding to the state it was in (here s) and the action a that it took. The remaining Q-values stay as they were, affected only by experiences beginning with the corresponding state and action.

The Q-learning algorithm described in this way works fine for learning discrete state problems as in Figure 1 (left). The state is nicely enumerable and there aren't too many of them (the set of actions is typically much smaller). Representing Q-values in a lookup table is perfectly sensible in this case. We expect to be able to visit the exact same state multiple times which allows us to try alternative actions during different visits and learn reliable estimates of the corresponding Q-values. This is, however, not at all the case for "space invaders" problem in Figure 1 (right). The state of the game presented to the agent is now merely an image, and there are a huge number of possible images! Given the vast state space we can no longer hope to store Q-values in a lookup table nor would it be useful even if we could. Agent can explore only for a limited time and thus need to learn to generalize from states actually visited to those that are somehow similar. Indeed, we will have to resort to *function approximation*, i.e., use a parametric model to represent the Q-values. The model maps the state observation (e.g., s can be an image) to Q-values $Q(s, a; \theta)$, $a \in A$, corresponding to the possible actions (assumed a small, fixed number). We must modify our Q-learning algorithm so that we can learn the parameters θ instead of lookup table entries based on experience snippets. In a *deep Q-learning algorithm* the mapping from state observation to values is provided by a deep neural network such as the one in Figure 2 below.

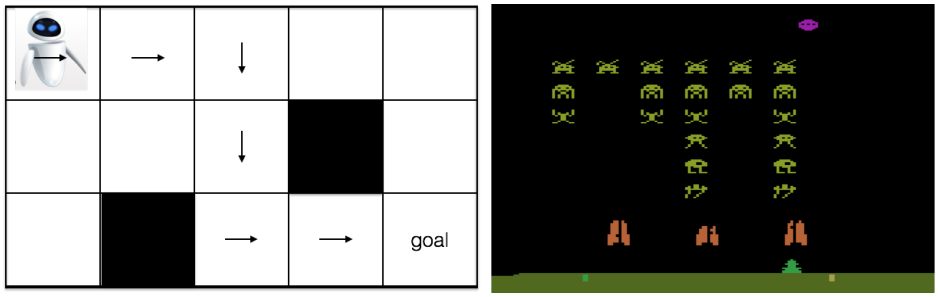

Figure 1: Left: a simple grid-world robot navigation task where the state corresponds to the grid location (square). Right: state of the game as an image, i.e., visual layout of the space invaders game [Mnih et al., 2013].

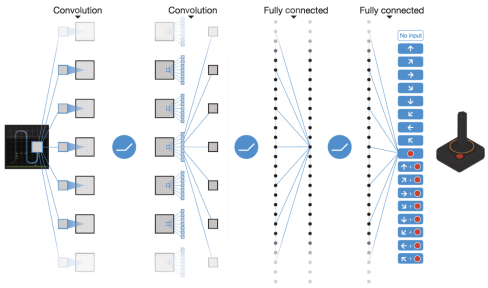

Figure 2: A deep convolutional Q-network for representing Q-values based on image states [Mnih et al., 2013]. The output layer units correspond to possible actions and represent $Q(s, a)$, $a \in A$, for the image s provided as an input.

10.3.1 Challenges and solutions

The Q-learning algorithm enjoys a nice theoretical convergence guarantee if the state and action spaces are finite and the state-action values are represented in a lookup table. This no longer holds with function approximation. Indeed, switching to function approximation and more complex environments brings up a number of challenges. We will discuss a few them here.

- **Bellman equation** Unlike with lookup tables, there may no longer be any setting of the parameters θ such that the Q-values would satisfy the Bellman equation for all states (e.g., game layouts) $s \in S$

$$Q(s, a; \theta) \stackrel{?}{=} E\left\{ R(s, a, s') + \gamma \max_{a' \in A} Q(s', a'; \theta) \mid s, a \right\} \tag{9}$$

  where the expectation is over the next state s' (if stochastic) and any possible randomness in the reward function. For example, if we use a simple linear classifier operating on an image as state, we clearly cannot perfectly reproduce all possible state action values that the game may require. If we cannot expect to satisfy the Bellman equation, what do we do?

- **Unstable function approximation** Using function approximation is always a bit problematic since updating parameters θ (however we choose to do it) imply changes for other states and other actions as well. Are these other changes beneficial? Not necessarily. Consider a simple example where s is one dimensional real number and we use a linear function approximation. If $Q(s, a = 1)$ should be -1 for $s = -1$, zero when $s = 0$, and -1 when $s = 1$, we cannot possibly find a perfect linear approximation $Q(s, a = 1) = ws + w_0$. Thus updating based on states $s = -1$ and $s = 0$ will have an adverse effect on the value $Q(s = 1, a = 1)$, and vice versa. We can mitigate this issue by using flexible deep learning architectures for function approximation such as the one in Figure 2.

- **Correlated experience** As the agent explores, e.g., playing a game, successive experiences that it generates in the form of $(s_t, a_t, s_{t+1}, R(s_t, a_t, s_{t+1}))$, $t = 1, 2, \ldots$ are highly correlated since game states change just a little based on each action. If we start updating parameters θ along these experience traces, we run into problems. The problem is similar to estimating a classifier where the training examples are arranged such that we will first get all the positive examples, then all the negative examples. Perceptron or any gradient based on-line classifier, if trained by consuming the training examples in this funny order, would oscillate badly in large waves.

Let's see how we can address the remaining two (first and last) challenges. Since we cannot expect to necessarily satisfy the Bellman equation, we could try to minimize the *Bellman error* instead. In other words, we could minimize

$$\left(R(s_t, a_t, s_{t+1}) + \gamma \max_{a' \in A} Q(s_{t+1}, a'; \theta) - Q(s_t, a_t; \theta) \right)^2 \tag{10}$$

over the experiences generated. To solve the correlated experience issue, we adopt a randomized strategy known as *experience replay*. In other words, we will sample uniformly at random experiences $e = (s, a, s', R(s, a, s'))$ out of those we have generated along game playing sequences $e_t = (s_t, a_t, s_{t+1}, R(s_t, a_t, s_{t+1}))$, $t = 1, 2, \ldots,$. So, in expectation, we could aim to minimize

$$E \left\{ \left(R(s, a, s') + \gamma \max_{a' \in A} Q(s', a'; \theta) - Q(s, a; \theta) \right)^2 \right\} \tag{11}$$

with respect to parameters θ, where the expected value is over experience replay. Our algorithm can then just involve stochastic gradient descent steps to minimize this criterion.

We are almost there to define a workable deep Q-learning algorithm. One last issue we have to deal with is that the "target values"

$$R(s, a, s') + \gamma \max_{a' \in A} Q(s', a'; \theta) \tag{12}$$

also change every time we change θ. This makes it hard to learn parameters in a stable fashion since our deep learning model for $Q(s, a; \theta)$ is trying to approximate a moving target (which is correlated with changes in θ). So, to alleviate this problem, we will separate the parameters of the network used for evaluating target values from those used for $Q(s, a; \theta)$. The target network parameters are updated, simply copied from $Q(s, a; \theta)$, at a slower pace.

10.3.2 Deep Q-learning algorithm

The algorithm we highlight below mirrors the algorithm provided in Mnih et al., "Human-Level Control through Deep Reinforcement Learning", Nature, 2013. We will simply illustrate the key steps of the algorithm; many variations exist in terms of how to intermingle or modify these steps.

- Initialize Q-network parameters θ and set $\theta^- = \theta$ for evaluating target values
- Iterate
  1. Generate and store new experiences $e_t = (s_t, a_t, s_{t+1}, R(s_t, a_t, s_{t+1}))$ by following ϵ-greedy policy implied by $Q(s, a; \theta)$
  2. Sample a small set of experiences D' uniformly at random from those already generated. For each e_t in this set, use SGD to update θ

     $$\theta \leftarrow \theta - \eta \nabla_\theta \ (y_t - Q(s_t, a_t; \theta))^2 \tag{13}$$

     where $y_t = R(s_t, a_t, s_{t+1}) + \gamma \max_{a' \in A} Q(s_{t+1}, a'; \theta^-)$. Note that y_t does not depend on θ because we have separated their parameters.
  3. Every C steps, set $\theta^- = \theta$ so that the target values start tracking the estimated Q-values (but at a slower pace).