Chapter 9

Q-Learning

Reading

1. Sutton & Barto, Chapter 6, 11
2. Human-level control through deep reinforcement learning https://www.nature.com/articles/nature14236
3. Deterministic Policy Gradient Algorithms, http://proceedings.mlr.press/v32/silver14.html
4. Addressing Function Approximation Error in Actor-Critic Methods https://arxiv.org/abs/1802.09477
5. An Application of Reinforcement Learning to Aerobatic Helicopter Flight, https://papers.nips.cc/paper/3151-an-application-of-reinforcement-learning-to-aerobatic-helicopter-flight

In the previous chapter, we looked at what are called "on-policy" methods, these are methods where the current controller u_{θ^k} is used to draw fresh data from the dynamical system and used to update to parameters θ^k. *The key inefficiency in on-policy methods is that this data is thrown away in the next iteration.* We need to draw a fresh set of trajectories from the system for $u_{\theta^{k+1}}$. This lecture will discuss off-policy methods which are a way to reuse past data. These methods require much fewer data than on-policy methods (in practice, about 10–100$\times$ less).

9.1 Tabular Q-Learning

Recall the value iteration algorithm for discrete (and finite) state and control spaces; this is also called "tabular" Q-Learning in the RL literature because we can store the Q-function $q(x, u)$ as a large table with number of rows being the number of states and number of columns being the number of controls, with

each entry in this table being the value $q(x, u)$. Value iteration when written using the Q-function at the k^{th} iteration for the tabular setting looks like

$$\begin{aligned} q^{(k+1)}(x,u) &= \sum_{x' \in X} \mathrm{P}(x' \mid x, u) \left(r(x,u) + \gamma \max_{u'} q^{(k)}(x', u') \right) \\ &= \mathop{\mathrm{E}}_{x' \sim \mathrm{P}(\cdot \mid x, u)} \left[r(x,u) + \gamma \max_{u'} q^{(k)}(x', u') \right]. \end{aligned}$$

> In the simplest possible instantiation of Q-learning, the expectation in the value iteration above (which we can only compute if we know a model of the dynamics) is replaced by samples drawn from the environment.

We will imagine the robot as using an *arbitrary* controller

$$u_e(\cdot \mid x)$$

that has a fairly large degree of randomness in how it picks actions. We call such a controller an "exploratory controller". Conceptually, its goal is to lead the robot to diverse states in the state-space so that we get a faithful estimate of the expectation in value iteration. We maintain the value $q^{(k)}(x, u)$ for all states $x \in X$ and controls $u \in U$ and update these values to get $q^{(k+1)}$ after *each step* of the robot.

From the results on Bellman iteration, we know that any Q-function that satisfies the above equation is the optimal Q-function; we would therefore like our Q-function to satisfy

$$q^*(x_k, u_k) \approx r(x_k, u_k) + \gamma \max_{u'} q^*(x_{k+1}, u').$$

over samples (x_k, u_k, x_{k+1}) collected as the robot explores the environment.

> **Tabular Q-Learning** Let us imagine the robot travels for n trajectories each of T time-steps each. We can now solve for q^* by minimizing the objective
>
> $$\min_q \frac{1}{n(T+1)} \sum_{i=1}^{n} \sum_{k=0}^{T} \| q(x_k^i, u_k^i) - r(x_k^i, u_k^i) - \gamma \max_{u' \in U} \; q(x_{k+1}^i, u') \|_2^2. \tag{9.1}$$
>
> on the data collected by the robot. The variable of optimization here are all values $q^*(x, u)$ for $x \in X$ and $u \in U$.

Notice a few important things about the above optimization problem. First, the last term is a maximization over $u' \in U$, it is $\max_{u' \in U} q(x_{k+1}^i, u')$ and not $q(x_{k+1}^i, u_{k+1}^i)$. In practice, you should imagine a robot performing Q-Learning in a grid-world setting where it seeks to find the optimal trajectory to go from a source location to a target location. If at each step, the robot has 4 controls to choose from, computing this last term involves taking the maximum of 4 different values (4 columns in the tabular Q-function).

Notice that for finite-horizon dynamic programming we initialized the Q-function at the terminal time to a known value (the terminal cost). Similarly, for infinite-horizon value iteration, we discussed how we can converge to the optimal Q-function with any initialization. In the above case, we do not impose any such constraint upon the Q-function, but there is an implicit constraint. All values $q(x,u)$ have to be consistent with each other and ideally, the residual

$$\|q(x_k^i, u_k^i) - r(x_k^i, u_k^i) - \gamma \max_{u' \in U} \; q(x_{k+1}^i, u')\|_2^2 = 0$$

for all trajectories i and all timesteps T.

Solving tabular Q-Learning How should we solve the optimization problem in (9.1)? This is easy, every entry $q(x,u)$ for $x \in U$ and $u \in U$ is a variable of this objective and each $\|\cdot\|_2^2$ term in the objective simply represents a constraint that ties these different values of the Q-function together. We can solve for all $q(x,u)$ iteratively as

$$\begin{aligned} q(x,u) \; &\leftarrow \; q(x,u) - \eta \, \nabla_{q(x,u)} \, \ell(q) \\ &= (1-\eta) \, q(x,u) - \eta \left(r(x,u) + \gamma \max_{u'} q(x', u') \right) \end{aligned} \tag{9.2}$$

where $\ell(q)$ is the entire objective $\frac{1}{n(T+1)} \sum_i \sum_k \cdots$ above and $(x,u,x') \equiv (x_k^i, u_k^i, x_{k+1}^i)$ in the second equation. An important point to note here is that although the robot collects a finite number of data

$$D = \left\{ (x_k^i, u_k^i)_{k=0,1,\ldots,T} \right\}_{i=1}^n$$

we have an estimate for the value $q(x,u)$ at all states $x \in X$. Intuitively, tabular Q-learning looks at the returns obtained by the robot after starting from a state x (the reward-to-come $J(x)$) and patches the returns from nearby states x, x' using the constraints in the objective (9.1).

Terminal state One must be very careful about the terminal state in such implementations of Q-learning. Typically, most research papers imagine that they are solving an infinite horizon problem but use simulators that have an explicit terminal state, i.e., the simulator does not proceed to the next timestep after the robot reaches the goal. A workaround for using such simulators (this applies for essentially all simulators) is to modify (9.2) as

$$q(x,u) = (1-\eta) \, q(x,u) - \eta \left(r(x,u) + \gamma \left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) \max_{u'} q(x', u') \right).$$

Effectively, we are setting $q(x', u) = 0$ for all $u \in U$ if x' is a terminal state of problem. This is a very important point to remember and Q-Learning will never work if you forget to include the term $\mathbf{1}_{\{x' \text{ is terminal}\}}$ in your expression.

What is the controller in tabular Q-Learning? The controller in tabular Q-Learning is easy to get after we solve (9.1). At test time, we use a deterministic

controller given by

$$u^*(x) = \operatorname*{argmax}_{u'} q^*(x, u').$$

9.1.1 How to perform exploration in Q-Learning

The exploratory controller used by the robot $u_e(\cdot \mid x)$ is critical to perform Q-Learning well. If the exploratory controller does not explore much, we do not get states from all parts of the state-space. This is quite bad, because in this case the estimates of Q-function at *all states* will be bad, not just at the states that the robot did not visit. To make this intuitive, imagine if we cordoned off some nodes in the graph for the backward version of Dijkstra's algorithm and never used them to update the dist variable. We would never get to the optimal cost-to-go for *all* states in this case because there could be trajectories that go through these cordoned off states that lead to a smaller cost-to-go. So it is quite important to pick the right exploratory controller.

It turns out that a random exploratory controller, e.g., a controller $u_e(\cdot \mid x)$ that picks controls uniformly randomly is pretty good. We can show that our tabular Q-Learning will converge to the optimal Q-function $q^*(x, u)$ as the amount of data drawn from the random controller goes to infinity, even if we initialize the table to arbitrary values. In other words, if we are guaranteed that the robot visits each state in the finite MDP infinitely often, it is a classical result that updates of the form (9.2) for minimizing the objective in (9.1) converge to the optimal Q-function.

ℹ This is again the power of dynamic programming at work. The Bellman equation guarantees the convergence of value iteration provided we compute the expectation exactly. But if the robot does give us lots of data from the environment, then Q-Learning also inherits this property of convergence to the optimal Q-function from any initialization.

Epsilon-greedy exploration Instead of the robot using a arbitrary controller $u_e(\cdot \mid x)$ to gather data, we can use the current estimate of the Q-function with some added randomness to ensure that the robot visits all states in the state-space. This is a key idea in Q-Learning and is known as "epsilon-greedy" exploration. We set

$$u_e(u \mid x) = \begin{cases} \operatorname{argmax}_u q(x, u) & \text{with probability } 1 - \epsilon \\ \text{uniform}(U) & \text{with probability } \epsilon. \end{cases} \tag{9.3}$$

for some user-chosen value of ϵ. Effectively, the robot repeats the controls it took in the past with probability $1 - \epsilon$ and uniformly samples from the entire control space with probability ϵ. The former ensures that the robot moves towards the parts of the state-space where states have a high return-to-come (after all, that is the what the Q-function $q(x, u)$ indicates). The latter ensures that even if the robot's estimate of the Q-function is bad, it is still visiting every state in the state-space infinitely often.

A different perspective on Q-Learning Conceptually, we can think of tabular Q-learning as happening in two stages. In the first stage, the robot gathers a large amount of data

$$D = \left\{(x_k^i, u_k^i)_{k=0,1,\ldots,T}\right\}_{i=1}^n$$

using the exploratory controller $u_e(\cdot \mid x)$; let us consider the case when we are using an arbitrary exploratory controller, not epsilon-greedy exploration. Using

this data, the robot fits a model for the system, i.e., it learns the underlying MDP

$$\mathrm{P}(x' \mid x, u);$$

this is very similar to the step in the Baum-Welch algorithm that we saw for learning the Markov state transition matrix of the HMM in Chapter 2. We simply take frequency counts to estimate this probability

$$\mathrm{P}(x' \mid x, u) \approx \frac{1}{N} \sum_i \mathbf{1}_{\{x' \text{ was reached from } x \text{ using control } u\}}$$

where N is the number of the times the robot took control u at state x. Given this transition matrix, we can now perform value iteration on the MDP to learn the Q-function

$$q^{(k+1)}(x, u) = \mathop{\mathrm{E}}_{x' \sim \mathrm{P}(\cdot|x,u)} \left[r(x, u) + \gamma \max_{u'} q^{(k)}(x', u) \right].$$

The success of this two-stage approach depends upon how accurate our estimate of $\mathrm{P}(x' \mid x, u)$ is. This in turn depends on how much the robot explored the domain and the size of the dataset it collected, both of these need to be large. We can therefore think of Q-learning as interleaving these two stages in a single algorithm, it learns the dynamics of the system and the Q-function for that dynamics simultaneously. But the Q-Learning algorithm does not really maintain a representation of the dynamics, i.e., at the end of running Q-Learning, we do not know what $\mathrm{P}(x' \mid x, u)$ is.

9.2 Function approximation (Deep Q Networks)

Tabular methods are really nice but they do not scale to large problems. The grid-world in the homework problem on policy iteration had 100 states, a typical game of Tetris has about 10^{60} states. For comparison, the number of atoms in the known universe is about 10^{80}. The number of different states in a typical Atari game is more than 10^{300}. These are all problems with a discrete number of states and controls, for continuous state/control-space, the number of distinct states/controls is infinite. So it is essentially impossible to run the tabular Q-Learning method from the previous section for most real-world problems. In this section, we will look at a powerful set of algorithms that parameterize the Q-function using a neural network to work around this problem.

We use the same idea from the previous chapter, that of parameterizing the Q-function using a deep network. We will denote

$$q_\varphi(x, u) : X \times U \mapsto \mathbb{R}$$

as the Q-function and our goal is to fit the deep network to obtain the weights $\hat{\varphi}$, instead of maintaining a very large table of size $|X| \times |U|$ for the Q-function. Fitting the Q-function is quite similar to the tabular case: given a dataset

$D = \left\{(x_t^i, u_t^i)_{t=0,1,\ldots,T}\right\}_{i=1}^n$ from the system, we want to enforce

$$q_\varphi(x_t^i, u_t^i) = r(x_t^i, u_t^i) + \gamma \max_{u'} q_\varphi(x_{t+1}^i, u')$$

for all tuples $(x_t^i, u_t^i, x_{t+1}^i)$ in the dataset. Just like the previous section, we will solve

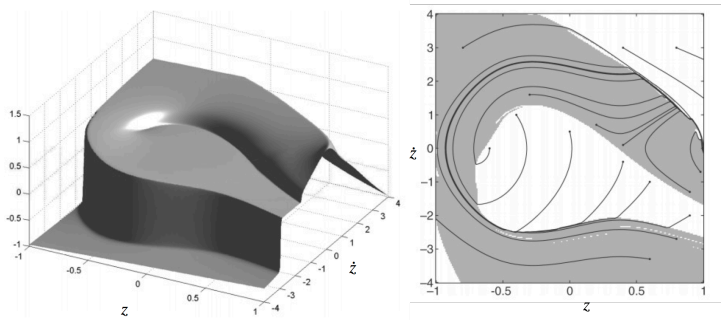
$$\begin{aligned}\hat{\varphi} &= \operatorname*{argmin}_\varphi \frac{1}{n(T+1)} \sum_{i=1}^n \sum_{t=1}^T \left(q_\varphi(x_t^i, u_t^i) - \underbrace{r(x_t^i, u_t^i) - \gamma \left(1 - \mathbf{1}_{\left\{x_{t+1}^i \text{ is terminal}\right\}}\right) \max_{u'} q_\varphi(x_{t+1}^i, u')}_{\text{target}(x';\varphi)} \right)^2 \\ &\equiv \operatorname*{argmin}_\varphi \frac{1}{n(T+1)} \sum_{i=1}^n \sum_{t=1}^T \left(q_\varphi(x_t^i, u_t^i) - \text{target}(x_{t+1}^i; \varphi)\right)^2 \end{aligned} \tag{9.4}$$

The last two terms in this expression above are together called the "target" because the problem is very similar to least squares regression, except that the targets also depend on the weights φ. This is what makes it challenging to solve.

As discussed above, Q-Learning with function approximation is known as "Fitted Q Iteration". Remember that very important point that the robot collects data using the exploratory controller $u_e(\cdot \mid x)$ but the Q-function that we fit is the *optimal* Q-function.

Fitted Q-Iteration with function approximation may not converge to the optimal Q-function It turns out that (9.4) has certain mathematical intricacies that prevent it from converging to the optimal Q-function. We will first give the intuitive reason. In the tabular Q-Learning setting, if we modify some entry $q(x, u)$ for an $x \in X$ and $u \in U$, the other entires (which are tied together using the Bellman equation) are all modified. This is akin to you changing the dist value of one node in Dijkstra's algorithm; the dist values of *all* other nodes will have to change to satisfy the Bellman equation. This is what (9.2) achieves if implemented with a decaying step-size η; see http://users.isr.ist.utl.pt/∼mtjspaan/readingGroup/ProofQlearning.pdf for the proof. This does not hold for (9.4). Even if the objective in (9.4) is zero on our collected dataset, i.e., the Q-function fits data collected by the robot perfectly, the Q-function may not be the *optimal* Q-function. An intuitive way of understanding this problem is that even if the Bellman error is zero on samples in the dataset, the optimization objective says nothing about states that are not present in the dataset; the Bellman error on them is completely dependent upon the smoothness properties of the function expressed by the neural architecture. Contrast this comment with the solution of the HJB equation in Chapter 6 where the value function was quite non-smooth at some places. If our sampled dataset does not contain those places, there is no way the neural network can know the optimal form of the value function.

ℹ The mathematical reason behind this is that the Bellman operator, i.e., the update to the Q/value-function is a contraction for the tabular setting, this is not the case for Fitted Q-Iteration unless the function approximation has some technical conditions imposed upon it.



9.2.1 Embellishments to Q-Learning

We next discuss a few practical aspects of implementing Q-Learning. Each of the following points is extremely important to understand how to get RL to work on real-world problems, so you should internalize these.

Pick mini-batches from different trajectories in SGD . In practice, we fit the Q-function using stochastic gradient descent. At each iteration we sample a mini-batch of inputs $(x_t^i, u_t^i, x_{t+1}^i)$ from different trajectories $i \in \{1, \ldots, n\}$ and update the weights φ in the direction of the negative gradient.

$$\varphi^{k+1} = \varphi^k - \eta \nabla_\varphi \left(q_{p^k}(x, u) - \text{target}(x'; \varphi^k) \right)^2 .$$

The mini-batch is picked to have samples from different trajectories because samples from the same trajectory are correlated to each other (after all, the robot obtains the next tuple (x', u', x'') from the previous tuple (x, u, x')).

Replay buffer The dataset D is known as the replay buffer.

Off-policy learning The replay buffer is typically not fixed during training. Instead of drawing data from the exploratory controller u_e, we can think of the following algorithm. Initialize the Q-function weights to φ^0 and the dataset to $D = \emptyset$. At the k^{th} iteration,

- Draw a dataset D^k of n trajectories from the ϵ-greedy policy

$$u_e(u \mid x) = \begin{cases} \text{argmax}_u \, q^k(x, u) & \text{with probability } 1 - \epsilon \\ \text{uniform}(U) & \text{with probability } \epsilon. \end{cases}$$

- Add new trajectories to the dataset

$$D \leftarrow D \cup D^k.$$

- Update weights to q^{k+1} using all past data D using (9.4).

Compare this algorithm to policy-gradient-based methods which throw away the data from the previous iteration. Indeed, when we want to compute the gradient $\nabla_\theta \, \mathrm{E}_{\tau \sim p_{\theta^k}} [R(\tau)]$, we should sample trajectories from current weights θ^k, we cannot use trajectories from some old weights. In contrast,

in Q-Learning, we maintain a cumulative dataset D that contains trajectories from all the past ϵ-greedy controllers and use it to find new weights of the Q-function. We can do so because of the powerful Bellman equation, Q-Iteration is learning the *optimal* value function and no matter what dataset (9.4) is evaluated upon, if the error is zero, we are guaranteed that Q-function learned is the optimal one. Policy gradients do not use the Bellman equation and that is why they are so inefficient. This is also the reason Q-Learning with a replay buffer is called "off-policy" learning because it learns the optimal controller even if the data that it uses comes from some other non-optimal controller (the exploratory controller or the ϵ-greedy controller).

Using off-policy learning is an old idea, the DQN paper which demonstrated very impressive results on Atari games using RL brought it back into prominence.

Setting a good value of ϵ for exploration is critical Towards the beginning of training, we want a large value for ϵ to gather diverse data from the environment. As training progresses, we want to reduce ϵ because presumably we have a few good control trajectories that result in good returns and can focus on searching the neighborhood of these trajectories.

Prioritized experience replay is an idea where instead of sampling from the replay buffer D uniformly randomly when we fit the Q-function in (9.4), we only sample data points (x_t^i, u_t^i) which have a high Bellman error

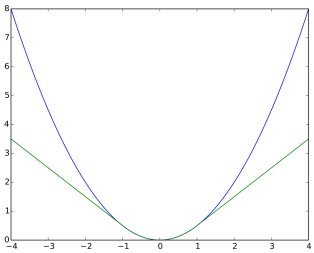
$$|q_\varphi(x_t^i, u_t^i) - r(x_t^i, u_t^i) - \gamma \left(1 - \mathbf{1}_{\{x_{t+1}^i \text{ is terminal}\}}\right) \max_{u'} q_\varphi(x_{t+1}^i, u')|$$

This is a reasonable idea but is not very useful in practice for two reasons. First, if we use deep networks for parameterizing the Q-function, the network *can* fit even very complex datasets so there is no reason to not use the data points with low Bellman error in (9.4); the gradient using them will be small anyway. Second, there are a lot of hyper-parameters that determine prioritized sampling, e.g., the threshold beyond which we consider the Bellman error to be high. These hyper-parameters are quite difficult to use in practice and therefore it is a good idea to not use prioritized experience replay at the beginning of development of your method on a new problem.

Using robust regression to fit the Q-function There may be states in the replay buffer with very high Bellman error, e.g., the kinks in the value function for the mountain car obtained from HJB above, if we happen to sample those. For instance, these are states where the controller "switches" and is discontinuous function of state x. In these cases, instead of these few states dominating the gradient for the entire dataset, we can use ideas in robust regression to reduce their effect on the gradient. A popular way to do so is to use a Huber-loss in place of the quadratic loss in (9.4)

$$\text{huber}_\delta(a) = \begin{cases} \frac{a^2}{2} & \text{for } |a| \leq \delta \\ \delta\left(|a| - \frac{\delta}{2}\right) & \text{otherwise.} \end{cases} \tag{9.5}$$

ℹ Huber loss for $\delta = 1$ (green) compared to the squared error loss (blue).



Delayed target Notice that the target also depends upon the weights φ:

$$\text{target}(x';\varphi) := r(x,u) + \gamma\left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right)\max_{u'} q_\varphi(x',u').$$

This creates a very big problem when we fit the Q-function. Effectively, both the covariate and the target in (9.4) depend upon the weights of the Q-function. Minimizing the objective in (9.4) is akin to performing least squares regression where the targets keep changing every time you solve for the solution. This is the root cause of why Q-Learning is difficult to use in practice. A popular hack to get around this problem is to use some old weights to compute the target, i.e., use the loss

$$\frac{1}{n(T+1)}\sum_{i,t}\left(q_{\varphi^k}(x_t^i,u_t^i) - \text{target}(x_{t+1}^i;\ \varphi^{k'})\right)^2. \tag{9.6}$$

in place of (9.4). Here k' is an iterate much older than k, say $k' = k - 100$. This trick is called "delayed target".

Exponential averaging to update the target Notice that in order to implement delayed targets as discussed above we will have to save all weights $\varphi^k, \varphi^{k-1}, \ldots, \varphi^{k-100}$, which can be cumbersome. We can however do yet another clever hack and initialize two copies of the weights, one for the actual Q-function φ^k and another for the target, let us call it φ'^k. We set the target equal to the Q-function at initialization. The target copy is updated at each iteration to be

$$\varphi'^{k+1} = (1-\alpha)\varphi'^k + \alpha\varphi^{k+1} \tag{9.7}$$

with some small value, say $\alpha = 0.05$. The target's weights are therefore an exponentially averaged version of the weights of the Q-function.

Why are delayed targets essential for Q-Learning to work? There are many explanations given why delayed targets are essential in practice but the correct one is not really known yet.

1. For example, one reason could be that since $q_{\varphi^k}(x,u)$ for a given state typically increases as we train for more iterations in Q-Learning, the old weights inside a delayed target are an underestimate of the true target. This might lead to some stability in situations when the Q-function's weights φ^k change too quickly when we fit (9.4) or we do not have enough data in the replay buffer yet.
2. Another reason one could hypothesize is related to concepts like self-distillation. For example, we may write a new objective for Q-Learning that looks like

$$\left(q_{\varphi^k}(x_t^i,u_t^i) - \text{target}(x_{t+1}^i;\ \varphi^k)\right)^2 + \frac{1}{2\lambda}\|\varphi^k - \varphi^{k'}\|_2^2$$

where the second term is known as proximal term that prevents the weights φ^k from change too much from their old values $\varphi^{k'}$. Proximal objectives are more stable versions of the standard quadratic objective

in (9.4) and help in cases when one is solving Q-Learning using SGD updates.

Double Q-Learning Even a delayed target may not be sufficient to get Q-Learning to lead to good returns in practice. Focus on one state x. One problem arise from the max operator in (9.4). Suppose that the Q-function q_{φ^k} corresponds to a particularly bad controller, say a controller that picks a control

$$\operatorname*{argmax}_u q_{\varphi^k}(x,u)$$

that is very different from the optimal control

$$\operatorname*{argmax}_u q^*(x,u)$$

then, even the delayed target $q_{\varphi^{k'}}$ may be a similarly poor controller. The ideal target is of course the return-to-come, or the value of the optimal Q-function $\max_{u'} q^*(x',u')$, but we do not know it while fitting the Q-function. The same problem also occurs if our Q-function (or its delayed version, the target) is too optimistic about the values of certain control inputs, it will consistently pick those controls in the max operator. One hack to get around this problem is to pick the maximizing control input using the non-delayed Q-function but use the value of the delayed target

$$\text{target}_{\text{DDQN}}(x^i_{t+1};{\varphi'}^k) = r(x,u) + \gamma\left(1 - \mathbf{1}_{\left\{x^i_{t+1} \text{ is terminal}\right\}}\right) q_{\varphi'^k}(x^i_{t+1},u'). \tag{9.8}$$

where

$$u' = \underbrace{\operatorname*{argmax}_u q_{\varphi^k}(x^i_{t+1},u)}_{\text{control chosen by the Q-function}}.$$

Training two Q-functions We can also train two copies of the Q-function simultaneously, each with its own delayed target and mix-and-match their targets. Let ${\varphi^{(1)}}^k$ and ${\varphi'^{(1)}}^k$ be one Q-function and target pair and ${\varphi^{(2)}}^k$ and ${\varphi'^{(2)}}^k$ be another pair. We update both of them using the following objective.

$$\begin{aligned}
&\text{For } \varphi^{(1)}: \left({q^{(1)}}^k(x,u) - r(x,u) - \gamma\left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right)\text{target}_{\text{DDQN}}(x',{\varphi'^{(2)}}^k)\right)^2 \\
&\text{For } \varphi^{(2)}: \left({q^{(2)}}^k(x,u) - r(x,u) - \gamma\left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right)\text{target}_{\text{DDQN}}(x',{\varphi'^{(1)}}^k)\right)^2
\end{aligned} \tag{9.9}$$

Sometimes we also use only one target that is the minimum of the two targets (this helps because it is more pessimistic estimate of the true target)

$$\text{target}(x') := \min\left\{\text{target}_{\text{DDQN}}(x',{\varphi'^{(1)}}^k), \text{target}_{\text{DDQN}}(x',{\varphi'^{(2)}}^k)\right\}.$$

You will also see many papers train multiple Q-functions, many more than 2. In such cases, it is a good idea to pick the control for evaluation using all the

Q-functions:

$$u^*(x) := \operatorname*{argmax}_u \sum_k q_{\varphi^{(k)}}(x,u).$$

rather than only one of them, as is often done in research papers.

A remark on the various tricks used to compute the target It may seem that a lot of these tricks are about being pessimistic while computing the target. This is our current understanding in RL and it is born out of the following observation: typically in practice, you will observe that the Q-function estimates can become very large. Even if the TD error is small, the values $q_\varphi(x,u)$ can be arbitrarily large; see Figure 1 in Continuous Doubly Constrained Batch Reinforcement Learning for an example in a slightly different setting. This occurs because we pick the control that maximizes the Q-value of a particular state x in (9.8). Effectively, if the Q-value $q_\varphi(x',u)$ of a particular control $u \in U$ is an over-estimate, the target will keep selecting this control as the maximizing control, which drives up the value of the Q-function at $q_\varphi(x,u)$ as well. This problem is a bit more drastic in the next section on continuous-valued controls. It is however unclear how to best address this issue and design mathematically sound methods that do not use arbitrary heuristics such as "pessimism".

ℹ Mathematically, the fundamental problem in function-approximation-based RL is actually clear: even if the Bellman operation is a contraction for tabular RL, it need not be a contraction when we are approximating the Q-function using a neural network. Therefore minimizing TD-error which works quite well for the tabular case need not work well in the function-approximation case. There may exist other, more robust, ways of computing the Bellman fixed point $q_\varphi(x,u) = r(x,u) + \max_{u'} \gamma\, q_\varphi(x',u')$ other than minimizing the the squared TD error but we do not have good candidates yet.

9.3 Q-Learning for continuous control spaces

All the methods we have looked at in this chapter are for discrete control spaces, i.e., the set of controls that the robot can take is a finite set. In this case we can easily compute the maximizing control of the Q-function.

$$u^*(x) = \operatorname*{argmax}_u q_\varphi(x,u).$$

Certainly a lot of real-world problems have continuous-valued controls and we therefore need Q-Learning-based methods to handle this.

Deterministic policy gradient A natural way, although non-rigorous, to think about this is to assume that we are given a Q-function $q_\varphi(x,u)$ (we will leave the controller for which this is the Q-function vague for now) and a dataset $D = \left\{(x_t^i, u_t^i)_{t=0}^T\right\}_{i=1}^n$. We can find a deterministic feedback controller that takes controls that lead to good values as

$$\theta^* = \max_\theta \frac{1}{n(T+1)} \sum_{i=1}^n \sum_{t=0}^T q_\varphi(x_t^i, u_\theta(x_t^i)). \tag{9.10}$$

Effectively we are fitting a feedback controller that takes controls $u_{\theta^*}(x)$ that are the maximizers of the Q-function. This is a natural analogue of the argmax over controls for discrete/finite control spaces. Again we should think of having a deep network that parametrizes the deterministic controller and fitting

its parameters θ using stochastic gradient descent on (9.10)

$$\begin{aligned}\theta^{k+1} &= \theta^k + \eta\, \nabla_\theta\, q_\varphi(x^\omega, u_{\theta^k}(x^\omega)) \\ &= \theta^k + \eta\, (\nabla_u\, q_\varphi(x^\omega, u))\, (\nabla_\theta\, u_{\theta^k}(x^\omega))\end{aligned} \tag{9.11}$$

where ω is the index of the datum in the dataset D. The equality was obtained by applying the chain rule. This result is called the "deterministic policy gradient" and we should think of it as the limit of the policy gradient for a stochastic controller as the stochasticity goes to zero. Also notice that the term

$$\nabla_u\, q_\varphi(x^\omega, u)$$

is the gradient of the output of the Q-function $q_\varphi : X \times U \mapsto \mathbb{R}$ with respect to its second input u. Such gradients can also be easily computed using backpropagation in PyTorch. It is different than the gradient of the output with respect to its weights

$$\nabla_\varphi\, q_\varphi(x^\omega, u).$$

On-policy deterministic actor-critic Let us now construct an analogue of the policy gradient method for the case of a deterministic controller. The algorithm would proceed as follows. We initialize weights of a Q-function φ^0 and weights of the deterministic controller θ^0.

1. At the k^{th} iteration, we collect a dataset from the robot using the latest controller u_{θ^k}. Let this dataset be D^k that consists of tuples (x, u, x', u').
2. Fit a Q-function q^{θ^k} to this dataset by minimizing the temporal difference error

$$\varphi^{k+1} = \operatorname*{argmin}_{\varphi} \sum_{(x,u,x',u') \in D^k} \left(q_\varphi(x,u) - r(x,u) - \gamma \left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) q_{\varphi'}(x',u')\right)^2. \tag{9.12}$$

   Notice an important difference in the expression above, instead of using $\max_u$ in the target, we are using the control that the current controller, namely u_{θ^k} has taken. This is because we want to evaluate the controller u_{θ^k} and simply parameterize the Q-function using weights φ^{k+1}. More precisely, we hope that we have

$$q_{\varphi^{k+1}}(x, u_{\theta^k}(x)) \approx \max_u q^{\theta^k}(x,u).$$

3. We can now update the controller using this Q-function:

$$\theta^{k+1} = \theta^k + \eta\, \nabla_\theta\, q_{\varphi^{k+1}}(x^\omega, u_{\theta^k}(x^\omega)) \tag{9.13}$$

ℹ SARSA is an old algorithm in RL that is the tabular version of what we did here. It stands for state-action-reward-state-action ...

This algorithm is called "on-policy SARSA" because at each iteration we draw fresh data D^k from the environment; this is the direct analogue of actor-critic methods that we studied in the previous chapter for deterministic controllers.

Off-policy deterministic actor-critic methods We can also run the above algorithm using data from an exploratory controller. The only difference is

that the we now do not throw away the data D^k from older iterations

$$D = D^1 \cup \cdots \cup D^k$$

and therefore have to change (9.12) to be

$$\varphi^{k+1} = \operatorname*{argmin}_{\varphi} \sum_{(x,u,x',u') \in D} \left(q_\varphi(x,u) - r(x,u) - \gamma \left(1 - \mathbf{1}_{\{x' \text{ is terminal}\}}\right) q_{\varphi'}(x', \underbrace{u_{\theta^k}(x')}_{\text{notice the difference}}) \right)^2 . \tag{9.14}$$

Effectively, we are fitting the optimal Q-function using the data D but since we can no longer take the maximum over controls directly, we plug in the controller in the computation of the target. This is natural; we think of the controller as the one that maximizes the Q-function when we update (9.13). When used with deep networks, this is called the "deep deterministic policy gradient" algorithm, it is popular by the name DDPG.