TD-learning and Q-learning

http://bicmr.pku.edu.cn/~wenzw/bigdata2020.html

Acknowledgement: this slides is based on Prof. Shipra Agrawal's lecture notes

Outline

TD-learning

- TD-learning is essentially approximate version of policy evaluation using samples. Adding policy improvement gives an approximate version of policy iteration.
- Since $V^{\pi}(s)$ is defined as the expectation of the random return when the process is started from the given state s, an obvious way of estimating this value is to compute an average over multiple independent realizations started from the given state. This is an instance of the so-called Monte-Carlo method.
- Unfortunately, the variance of the observed returns can be high. The Monte-Carlo technique is further difficult to apply if the system is not accessible through a simulator but rather estimation happens while actually interacting with the system.

TD(0)-learning

- Policy evaluation is about estimating $V^{\pi}(\cdot)$, which by Bellman equations is equivalent to finding a stationary point of

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi(s)}\left[R(s,a) + \gamma \sum_{s'} P\left(s,a,s'\right) V^{\pi}\left(s'\right)\right], \forall s$$

- However, we need to estimate this using only observations r_t, s_{t+1} on playing some action at a_t current state s_t.
- Let the current estimate of $V(s)$ is $\hat{V}(s)$. Let on taking action $a_t = \pi\left(s_t\right)$ in the current state s_t, s_{t+1} is the observed (sample) next state. The predicted value function for the next state s_{t+1} is $\hat{V}\left(s_{t+1}\right)$, giving another prediction of value function at state s_t $r_t + \gamma\hat{V}\left(s_{t+1}\right)$. Note that

$$\mathbb{E}\left[r_t + \hat{V}\left(s_{t+1}\right) | s_t, \hat{V}\right] = \mathbb{E}_{a\sim\pi(s_t)}\left[R\left(s_t,a\right) + \sum_{s'} P\left(s_t,a,s'\right)\hat{V}\left(s'\right) | s_t, \hat{V}\right]$$

TD(0)-learning

- From Bellman equations, we are looking for $\hat{V}$ such that

$$\hat{V}(s_t) \approx r_t + \hat{V}(s_{t+1})$$

- The TD method performs the following update to the value function estimate at s_t, moving it towards the new estimate:

$$\hat{V}(s_t) \leftarrow (1-\alpha_t)\hat{V}(s_t) + \alpha_t\left(r_t + \gamma\hat{V}(s_{t+1})\right)$$

- Let δ_t be the following gap:

$$\delta_t := r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$$

  referred to as temporal difference, i.e., the difference between current estimate, and one-lookahead estimate. Then, the above also be written as:

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha_t\delta_t \tag{1}$$

SGD interpretation: a general principal

- Let z_s be a random variable independent to θ.
- Consider:

$$\min_{\theta} \quad (x_\theta - \mathbf{E}_s[z_s])^2$$

- It is equivalent to

$$\begin{aligned}
& \min_{\theta} \quad (x_\theta - \mathbf{E}_s[z_s])^2 - (\mathbf{E}_s[z_s])^2 + \mathbf{E}_s[z_s^2] \\
\Longleftrightarrow \quad & \min_{\theta} \quad x_\theta^2 + (\mathbf{E}_s[z_s])^2 - 2x_\theta \mathbf{E}_s[z_s] - (\mathbf{E}_s[z_s])^2 + \mathbf{E}_s[z_s^2] \\
\Longleftrightarrow \quad & \min_{\theta} \quad x_\theta^2 - 2x_\theta \mathbf{E}_s[z_s] + \mathbf{E}_s[z_s^2] \\
\Longleftrightarrow \quad & \min_{\theta} \quad \mathbf{E}_s[x_\theta - z_s]^2
\end{aligned}$$

Expectation is taken out. It enables us to perform sample on z_s.

SGD interpretation

- Recall the Bellman equation:

$$V^\pi(s) = \mathop{\mathbf{E}}_{\substack{a\sim\pi \\ s'\sim P}} \left[R(s,a,s') + \gamma V^\pi(s')\right],$$

- we introduce a target V^π_{targ} and approximate:

$$V^\pi(s) \approx \mathop{\mathbf{E}}_{\substack{a\sim\pi \\ s'\sim P}} \left[R(s,a,s') + \gamma V^\pi_{targ}(s')\right]$$

- construct a least-squares problem:

$$\begin{aligned} & \min_{V^\pi(s)} \quad \left(V^\pi(s) - \mathop{\mathbf{E}}_{\substack{a\sim\pi \\ s'\sim P}} \left[R(s,a,s') + \gamma V^\pi_{targ}(s')\right]\right)^2 \\ \iff & \min_{V^\pi(s)} \quad \mathop{\mathbf{E}}_{\substack{a\sim\pi \\ s'\sim P}} \left[V^\pi(s) - [R(s,a,s') + \gamma V^\pi_{targ}(s')]\right]^2 \end{aligned}$$

SGD interpretation

- use SGD to solve:

$$\min_{V^\pi(s)} \quad \frac{1}{2} \mathop{\mathbf{E}}_{\substack{a\sim\pi \\ s'\sim P}} \left[V^\pi(s) - [R(s,a,s') + \gamma V^\pi_{targ}(s')]\right]^2$$

- The gradient with respect to $V^\pi(s)$ is

$$\mathop{\mathbf{E}}_{\substack{a\sim\pi \\ s'\sim P}} \left[V^\pi(s) - [R(s,a,s') + \gamma V^\pi_{targ}(s')]\right].$$

  Take one sample:

$$V^\pi(s_t) - [R(s_t,a,s_{t+1}) + \gamma V^\pi_{targ}(s_{t+1})] = -\delta_t$$

- Hence, one step of SGD is

$$V^\pi(s_t) \leftarrow V^\pi(s_t) + \alpha\delta_t$$

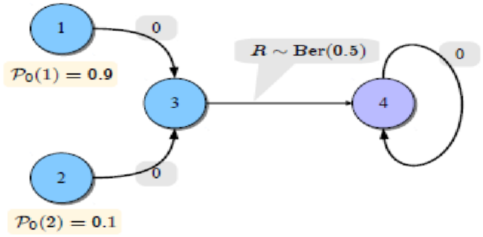
Tabular TD(0) method for policy evaluation

Algorithm 1 Tabular TD(0) method for policy evaluation

```
1:  Initialization: Given a starting state distribution $D_0$, policy $\pi$, the
    method evaluates $V^\pi(s)$ for all states $s$.
    Initialize $\hat{V}$ as an empty list/array for storing the value estimates.
2:  **repeat**
3:      Set $t = 1$, $s_1 \sim D_0$. Choose step sizes $\alpha_1, \alpha_2, \ldots$.
4:      Perform TD($0$) updates over an episode:
5:      **repeat**
6:          Take action at $a_t \sim \pi(s_t)$. Observe reward $r_t$, and new state
            $s_{t+1}$.
7:          $\delta_t := r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t)$
8:          Update $\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha_t \delta_t$
9:          $t = t + 1$
10:     **until** episode terminates
11: **until** change in $\hat{V}$ over consecutive episodes is small
```

Monte Carlo method

Why use only 1-step lookahead to construct target z? Why not lookahead entire trajectory (in problems where there is a terminal state, also referred to as episodic MDPs)?



[Szepesvari, 1999] In this example, all transitions are deterministic. The reward is zero, except when transitioning from state 3 to state 4, when it is given by a Bernoulli random variable with parameter 0.5. State 4 is a terminal state. When the process reaches the terminal state, it is reset to start at state 1 or 2. The probability of starting at state 1 is 0.9, while the probability of starting at state 2 is 0.1.

Monte Carlo method

- The resulting method is referred to as Monte Carlo method, here for z a sample trajectory starting at s_t is used

$$z = \sum_{n=0}^{\infty} \gamma^n r_{t+n} =: \mathcal{R}_t$$

so that

$$\delta_t = z - \hat{V}(s_t) = \mathcal{R}_t - \hat{V}(s_t)$$
$$\hat{V}(s_t) = (1-\alpha)\hat{V}(s_t) + \alpha\mathcal{R}_t$$

TD(0) or Monte-Carlo?

This example is taken from page $22-23$, *Szepesvari* [1999]

- First, let us consider an example when $TD(0)$ converges faster. Consider the above undiscounted episodic MRP shown on the above figure.
- The initial states are either 1 or 2. With high probability the process starts at state 1, while the process starts at state 2 less frequently.
- Consider now how $TD(0)$ will behave at state 2. By the time state 2 is visited the k^{th} time, on the average state 3 has already been visited 10 k times.
- Assume that $\alpha_t = 1/(t+1)$ (the TD updates with this step size reduce to averaging of target observations). At state 1 and 2, the target is $\hat{V}(3)$ (since immediate reward is 0 and transition probability to state 3 is 1).

TD(0) or Monte-Carlo?

- Therefore, whenever state 2 is visited the $TD(0)$ sets its value as the average of estimates $\hat{V}^t(3)$ over the time steps t when state 1 was visited (similarly for state 2). At state 3 the $TD(0)$ update reduces to averaging the Bernoulli rewards incurred upon leaving state 3. At the k^{th} visit of state 2, $\mathrm{Var}(\hat{V}(3)) \simeq 1/(10k)$ Clearly, $\mathbb{E}[\hat{V}(3)] = 0.5$. Thus, the target of the update of state 2 will be an estimate of the true value of state 2 with accuracy increasing with k.
- Now, consider the Monte-Carlo method. The Monte-Carlo method ignores the estimate of the value of state 3 and uses the Bernoulli rewards directly. In particular, $\mathrm{Var}\left(\mathcal{R}_t | s_t = 2\right) = 0.25$, i.e., the variance of the target does not change with time.
- On this example, this makes the Monte-Carlo method slower to converge, showing that sometimes bootstrapping might indeed help.

TD(0) or Monte-Carlo?

- To see an example when bootstrapping is not helpful, imagine that the problem is modified so that the reward associated with the transition from state 3 to state 4 is made deterministically equal to one.
- In this case, the Monte-Carlo method becomes faster since $\mathcal{R}_t = 1$ is the true target value, while for the value of state 2 to get close to its true value, $TD(0)$ has to wait until the estimate of the value at state 3 becomes close to its true value. This slows down the convergence of $TD(0)$.
- In fact, one can imagine a longer chain of states, where state $i + 1$ follows state i, for $i \in 1, \ldots, N$ and the only time a nonzero reward is incurred is when transitioning from state $N - 1$ to state N.
- In this example, the rate of convergence of the Monte-Carlo method is not impacted by the value of N, while $TD(0)$ would get slower with N increasing.

TD(λ)

- TD(λ) is a "middle-ground" between $TD(0)$ and Monte-Carlo evaluation.
- Here, the algorithm considers ℓ-step predictions:

$$z_t^\ell = \sum_{n=0}^{\ell} \gamma^n r_{t+n} + \gamma^{\ell+1} \hat{V}\left(s_{t+\ell+1}\right)$$

with temporal difference:

$$\begin{aligned}
\delta_t^\ell &= z - \hat{V}\left(s_t\right) \\
&= \sum_{n=0}^{\ell} \gamma^n r_{t+n} + \gamma^{\ell+1} \hat{V}\left(s_{t+\ell+1}\right) - \hat{V}\left(s_t\right) \\
&= \sum_{n=0}^{\ell} \gamma^n \left(r_{t+n} + \gamma \hat{V}\left(s_{t+n+1}\right) - \hat{V}\left(s_{t+n}\right)\right) \\
&= \sum_{n=0}^{\ell} \gamma^n \delta_{t+n}
\end{aligned}$$

TD(λ)

- In TD(λ) method, a mixture of ℓ-step predictions is used, with weight $(1-\lambda)\lambda^\ell$ for $\ell \geq 0$. Therefore, $\lambda = 0$ gives $TD(0)$, and $\lambda \to 1$ gives Monte-Carlo method. $\lambda > 1$ gives a multi-step method. To summarize, the $TD(\lambda)$ update is given as:

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \alpha_t \sum_{\ell=0}^{\infty} (1-\lambda)\lambda^\ell \delta_t^\ell = \hat{V}(s_t) + \alpha_t \sum_{n=0}^{\infty} \lambda^n \gamma^n \delta_{t+n}$$

Policy improvement with TD-learning

TD-learning allows evaluating a policy. For using TD-learning for finding optimal policy, we need to be able to improve the policy. Recall policy iteration effectively requires evaluating Q-value of a policy, where $Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s'} P^{\pi}(s,a,s') V^{\pi}(s')$. With simple modification, TD-learning can be used to estimate Q-value of a policy. There, the updates would be replaced by:
$\delta_t := r_t + \gamma \hat{Q}(s_{t+1}, \pi(s_{t+1})) - \hat{Q}(s_t, a_t)$
Update $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha_t \delta_t$
Then, the scheme for policy improvement is similar to policy iteration. Repeat the following until convergence to some policy:

- Use TD-learning to evaluate the policy π^k. The method outputs $\hat{Q}^{\pi^k}(s,a), \forall s, a$
- Compute new 'improved policy' π^{k+1} as $\pi^{k+1}(s) \leftarrow \arg\max_a \hat{Q}^{\pi^k}(s,a)$.

Outline

Q-learning (tabular)

- Q-learning is a sample based version of Q-value iteration. This method attempts to directly find optimal Q-values, instead of computing Q-values of a given policy.
- Recall Q-value iteration: for all s, a update,

$$Q_{k+1}(s,a) \leftarrow R(s,a) + \gamma \sum_{s'} P\left(s,a,s'\right) \left(\max_{a'} Q_k\left(s',a'\right) \right)$$

  Q-learning approximates these updates using sample observations, similar to TD-learning.
- In steps $t = 1, 2, \ldots$ of an episode, the algorithm observes reward r_t and next state $s_{t+1} \sim P\left(\cdot, s_t, a_t\right)$ for some action a_t. It updates the Q-estimates for pair $\left(s_t, a_t\right)$ as follows:

$$Q_{k+1}\left(s_t,a_t\right) = (1-\alpha) Q_k\left(s_t,a_t\right) + \alpha \underbrace{\left(r_t + \gamma \max_{a'} Q_k\left(s_{t+1},a'\right) \right)}_{\text{target}}$$

SGD Interpretation

- The Bellman optimal equation is

$$Q^*(s,a) = \mathop{\mathbf{E}}_{s'\sim P}\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right]$$

- We introduce a target $Q_{targ}(s,a)$ and approximate:

$$Q(s,a) \approx \mathop{\mathbf{E}}_{s'\sim P}\left[R(s,a,s') + \gamma \max_{a'} Q_{targ}(s',a')\right]$$

- Construct a least square problem:

$$\begin{aligned} & \min \quad \left(Q(s,a) - \mathop{\mathbf{E}}_{s'\sim P}\left[R(s,a,s') + \gamma \max_{a'} Q_{targ}(s',a')\right]\right)^2 \\ \iff & \min \quad \mathop{\mathbf{E}}_{s'\sim P}\left[Q(s,a) - [R(s,a,s') + \gamma \max_{a'} Q_{targ}(s',a')]\right]^2 \end{aligned}$$

SGD Interpretation

- use SGD to solve the least square problem:

$$\min \quad \frac{1}{2} \mathop{\mathbf{E}}_{s' \sim P} \left[Q(s,a) - [R(s,a,s') + \gamma \max_{a'} Q_{targ}(s',a')] \right]^2$$

- One sample of the gradient is

$$Q(s,a) - [R(s,a,s_{t+1}) + \gamma \max_{a'} Q_{targ}(s_{t+1},a')] = -\delta_t.$$

- One SGD step is

$$Q(s,a) \leftarrow Q(s,a) + \alpha_t \delta_t.$$

Q-learning (tabular)

Algorithm 2 Tabular Q-learning method

```
 1: Initialization: Given a starting state distribution $D_0$.
    Initialize $\hat{Q}$ as an empty list/array for storing the $Q$-value estimates.
 2: **repeat**
 3:     Set $t = 1$, $s_1 \sim D_0$. Choose step sizes $\alpha_1, \alpha_2, \ldots$.
 4:     Perform $Q$-learning updates over an episode:
 5:     **repeat**
 6:         Take action at $a_t$. Observe reward $r_t$, and new state $s_{t+1}$.
 7:         $\delta_t := \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a')\right) - \hat{Q}(s_t, a_t)$
 8:         Update $\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha_t \delta_t$
 9:         $t = t + 1$
10:     **until** episode terminates
11: **until** change in $\hat{Q}$ over consecutive episodes is small
```

How to select actions, the issue of exploration

- The convergence results (discussed below) for Q-learning will say that if all actions and states are infinitely sampled, learning rate is small, but does not decrease too quickly, then Q-learning converges. (Does not matter how you select actions, as long as they are infinitely sampled).
- One option is to select actions greedily according to the current estimate $\max_a Q_k(s, a)$. But, this will reinforce past errors, and may fail to sample and estimate Q-values for actions which have higher error levels. We may get stuck at a subset of (suboptimal) actions.
- Therefore, exploration is required. The ϵ-greedy approach (i.e., with ϵ probability pick an action uniformly at random instead of greedy choice) can ensure infinite sampling of every action, but can be very inefficient.

How to select actions, the issue of exploration

- The same issue occurs in TD-learning based policy improvement methods. The choice of action is specified as the greedy policy according to the previous episode estimates. Without exploration this may not ensure that all actions and states are infinitely sampled.
- One option is to replace policy improvement step by greedy choice. That is, the policy improvement step will now compute the new 'improved policy' π^{k+1} as the randomized policy:

$$\pi^{k+1}(s) = \begin{cases} a_k^* := \arg\max_a \hat{Q}^{\pi^k}(s,a), & \text{with probability } 1-\epsilon+\frac{\epsilon}{|A|} \\ a, & \text{with probability } \frac{\epsilon}{|A|}, a \neq a_k^* \end{cases}$$

- Then, in policy evaluation, this 'randomized policy' must be used.

7 : $\delta_t := r_t + \gamma \mathbb{E}_{a \sim \pi^{k+1}(s_{t+1})}\left[\hat{Q}\left(s_{t+1}, a\right)\right] - \hat{Q}\left(s_t, a_t\right)$

8 : Update $\hat{Q}\left(s_t, a_t\right) \leftarrow \hat{Q}\left(s_t, a_t\right) + \alpha_t \delta_t$

Convergence theorem

Theorem 1 (Watkins and Dayan [1992])

Given bounded rewards $|r_t| \leq R$, learning rates $0 \leq \alpha_t < 1$, and

$$\sum_{i=1}^{\infty} \alpha_{n^i(s,a)} = \infty, \sum_{i=1}^{\infty} \left(\alpha_{n^i(s,a)}\right)^2 < \infty$$

then $\hat{Q}^t(s,a) \to Q(s,a)$ as $t \to \infty$ for all s, a with probability 1. Here, $n^i(s,a)$ is the index of the i^{th} time the action a is tried in state s, and $\hat{Q}^t(s,a)$ is the estimate $\hat{Q}$ in round t.

- If all actions and states are infinitely sampled, learning rate is small, but does not decrease too quickly, then Q-learning converges. (Does not matter how you select actions, as long as they are infinitely sampled).
- The proof of this and many similar results in RL algorithms follow the analysis of a more general online learning/optimization method - the stochastic approximation method.

Stochastic Approximation method

- The stochastic approximation (SA) algorithm essentially solves a system of (nonlinear) equations of the form

$$h(\theta) = 0$$

  for unknown $h(\cdot)$, based on noisy measurements of $h(\theta)$.
- More specifically, consider a (continuous) function $\mathbb{R}^d \to \mathbb{R}^d$, with $d \geq 1$, which depends on a set of parameters $\theta \in \mathbb{R}^d$. Suppose that $h(\theta)$ is unknown. However, for any θ we can measure $Z = h(\theta) + \omega$, where ω is some 0-mean noise. The classical SA algorithm (Robbins and Monro [1951]) is of the form

$$\begin{aligned} \theta_{n+1} &= \theta_n + \alpha_n Z_n \\ &= \theta_n + \alpha_n \left(h\left(\theta_n\right) + \omega_n \right), \quad n \geq 0 \end{aligned}$$

- Since ω_n is 0-mean noise, the stationary points of the above algorithm coincide with the solutions of $h(\theta) = 0$.

Asynchronous version

- More relevant to the RL methods discussed here is the asynchronous version of the SA method. In the asynchronous version of SA method, we may observe only one coordinate (say i^{th}) of $Z_n = h(\theta_n) + \omega_n$ at a time step, and we use that to update ith component of our parameter estimate:

$$\theta_{n+1}[i] = \theta_n[i] + \alpha_n Z_n[i]$$

- The convergence for this method will be proven similarly to the synchronous version, under the assumption that every coordinate is sampled infinitely often.

Outline

Q-learning with function approximation

- The tabular Q-learning does not scale with increase in the size of state space. In most real applications, there are too many states to keep visit, and keep track of.
- For scalability, we want to generalize, i.e., use what we have learned about already visited (relatively small number of) states, and generalize it to new, similar states.
- A fundamental idea is to use 'function approximation', i.e., use a lower dimensional feature representation of the state- action pair s, a and learn a parametric approximation $Q_\theta(s, a)$.

Q-learning with function approximation

- For example, the function $Q_\theta(s,a)$ can simply be a linear function in θ and features $Q_\theta(s,a) = \theta_0 f_0(s,a) + \theta_1 f_1(s,a) + \ldots + \theta_n f_n(s,a)$, or a deep neural net. Given parameter θ, the Q-function can be computed for unseen s, a. Instead of learning the $|S| \times |A|$ dimensional Q-table, the Q-learning algorithm will learn the parameter θ. Here, on observing sample transition to s' from s on playing action a, instead of updating the estimate of $Q(s,a)$ in the Q-table, the algorithm updates the estimate of θ.
- Intuitively, we are trying to find a θ such that for every s, a the Bellman equation,

$$Q_\theta(s,a) = \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[R\left(s,a,s'\right) + \gamma \max_{a'} Q_\theta\left(s',a'\right) \right]$$

can be approximated well for all s, a.

SGD Interpretation

- Similarly, we obtain a least square problem:

$$\begin{aligned}
&\min_{\theta} \quad \frac{1}{2} \mathop{\mathbf{E}}_{s' \sim P} \left[Q_\theta(s,a) - [R(s,a,s') + \gamma \max_{a'} Q_{\theta_{targ}}(s',a')] \right]^2 \\
&\min_{\theta} \quad \ell_\theta(s,a) = \mathop{\mathbf{E}}_{s' \sim P} \left[\ell_\theta \left(s,a,s' \right) \right]
\end{aligned}$$

- One sample of the gradient is

$$\begin{aligned}
& \nabla_\theta \ell_\theta(s,a,s') \\
= \quad & \left(Q_\theta(s,a) - [R(s,a,s') + \gamma \max_{a'} Q_{targ}(s_{t+1},a')] \right) \nabla_\theta Q_\theta(s,a) \\
= \quad & -\delta_t \nabla_\theta Q_\theta(s,a).
\end{aligned}$$

- One SGD step is

$$Q(s,a) \leftarrow Q(s,a) + \alpha_t \delta_t \nabla_\theta Q_\theta(s,a).$$

Q-learning Algorithm overview

Start with initial state $s = s_0$. In iteration $k = 1, 2, \ldots,$

- Take an action a.
- Observe reward r, transition to state $s' \sim P(\cdot|s, a)$.
- $\theta_{k+1} \leftarrow \theta_k - \alpha_k \nabla_{\theta_k} \ell_{\theta_k}(s, a, s')$, where

$$\nabla_\theta \ell_{\theta_k}\left(s, a, s'\right) = -\delta_t \nabla_\theta Q_{\theta_k}(s, a)$$

$$\delta_t = r + \gamma \max_{a'} Q_{\theta_k}\left(s', a'\right) - Q_{\theta_k}(s, a)$$

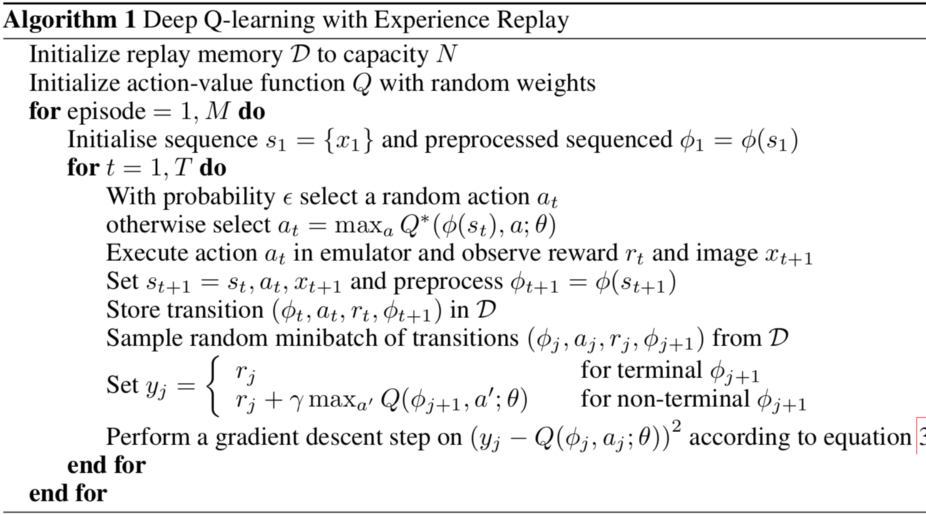
- $s \leftarrow s'$,

If s' reached at some point is a terminal state, s is reset to starting state.

Outline

Deep Q-Networks — Algorithm

DQN: θ is a deep neural network



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
**for** episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    **for** $t = 1, T$ **do**
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation
    **end for**
**end for**
```

Deep Deterministic Policy Gradient

- Deep Deterministic Policy Gradient (DDPG) is an algorithm which concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy.
- This approach is closely connected to Q-learning, and is motivated the same way: if you know the optimal action-value function $Q^*(s,a)$, then in any given state, the optimal action $a^*(s)$ can be found by solving

$$a^*(s) = \arg\max_a Q^*(s,a).$$

- DDPG interleaves learning an approximator to $Q^*(s,a)$ with learning an approximator to $a^*(s)$, and it does so in a way which is specifically adapted for environments with continuous action spaces. But what does it mean that DDPG is adapted *specifically* for environments with continuous action spaces? It relates to how we compute the max over actions in $\max_a Q^*(s,a)$.

- When there are a finite number of discrete actions, the max poses no problem, because we can just compute the Q-values for each action separately and directly compare them. (This also immediately gives us the action which maximizes the Q-value.) But when the action space is continuous, we can't exhaustively evaluate the space, and solving the optimization problem is highly non-trivial. Using a normal optimization algorithm would make calculating $\max_a Q^*(s,a)$ a painfully expensive subroutine. And since it would need to be run every time the agent wants to take an action in the environment, this is unacceptable.

- Because the action space is continuous, the function $Q^*(s,a)$ is presumed to be differentiable with respect to the action argument. This allows us to set up an efficient, gradient-based learning rule for a policy $\mu(s)$ which exploits that fact. Then, instead of running an expensive optimization subroutine each time we wish to compute $\max_a Q(s,a)$, we can approximate it with $\max_a Q(s,a) \approx Q(s,\mu(s))$.

The Q-Learning Side of DDPG

- First, let's recap the Bellman equation describing the optimal action-value function, $Q^*(s,a)$. It's given by

$$Q^*(s,a) = \mathop{\mathrm{E}}_{s' \sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right]$$

  where $s' \sim P$ is shorthand for saying that the next state, s', is sampled by the environment from a distribution $P(\cdot|s,a)$.

- This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d) (where d indicates whether state s' is terminal). We can set up a **mean-squared Bellman error (MSBE)** function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi,\mathcal{D}) = \mathop{\mathrm{E}}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right)\right)^2\right]$$

- Here, in evaluating $(1-d)$, we've used: "True" to 1 and "False" to zero. Thus, when "d==True"—which is to say, when s' is a terminal state—the Q-function should show that the agent gets no additional rewards after the current state.
- Q-learning algorithms for function approximators, such as DQN (and all its variants) and DDPG, are largely based on minimizing this MSBE loss function. There are two main tricks employed by all of them which are worth describing, and then a specific detail for DDPG.
- **Trick One: Replay Buffers.** All standard algorithms for training a deep neural network to approximate $Q^*(s, a)$ make use of an experience replay buffer. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning. This may take some tuning to get right.

- **Trick Two: Target Networks.** Q-learning algorithms make use of **target networks.** The term

$$r + \gamma(1 - d) \max_{a'} Q_{\phi}(s', a')$$

is called the **target**, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: ϕ. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay—that is to say, a second network, called the target network, which lags the first. The parameters of the target network are denoted ϕ_{targ}.

- In DQN-based algorithms, the target network is just copied over from the main network every some-fixed-number of steps. In DDPG-style algorithms, the target network is updated once per main network update by polyak averaging:

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$

  where ρ is between 0 and 1 (usually close to 1).

- **DDPG Detail: Calculating the Max Over Actions in the Target.** As mentioned earlier: computing the maximum over actions in the target is a challenge in continuous action spaces. DDPG deals with this by using a **target policy network** to compute an action which approximately maximizes $Q_{\phi_{\text{targ}}}$. The target policy network is found the same way as the target Q-function: by polyak averaging the policy parameters over the course of training.

- Putting it all together, Q-learning in DDPG is performed by minimizing the following MSBE loss with stochastic gradient descent:

$$L(\phi, \mathcal{D}) = \underset{(s,a,r,s',d)\sim\mathcal{D}}{\mathrm{E}}\left[\left(Q_\phi(s,a) - \left(r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))\right)\right)^2\right],$$

where $\mu_{\theta_{\text{targ}}}$ is the target policy.

The Policy Learning Side of DDPG

- Policy learning in DDPG is fairly simple. We want to learn a deterministic policy $\mu_\theta(s)$ which gives the action that maximizes $Q_\phi(s, a)$. Because the action space is continuous, and we assume the Q-function is differentiable with respect to action, we can just perform gradient ascent (with respect to policy parameters only) to solve

$$\max_{\theta} \underset{s \sim \mathcal{D}}{\mathrm{E}} \left[Q_\phi(s, \mu_\theta(s)) \right].$$

- Note that the Q-function parameters are treated as constants here.

Pseudocode: DDPG

Algorithm 3 Deep Deterministic Policy Gradient

```
1:  Input: initial policy parameters $\theta$, Q-function parameters $\phi$, empty replay buffer $\mathcal{D}$
2:  Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
3:  repeat
4:      Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
5:      Execute $a$ in the environment
6:      Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
7:      Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
8:      If $s'$ is terminal, reset environment state.
9:      if it's time to update then
10:         for however many updates do
11:             Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
12:             Compute targets $y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$
13:             Update Q-function by one step of gradient descent using $\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} (Q_\phi(s, a) - y(r, s', d))^2$
14:             Update policy by one step of gradient ascent using $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$
15:             Update target networks with

                    $\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$
                    $\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$

16:         end for
17:     end if
18: until convergence
```

Twin Delayed DDPG (TD3)

- A common failure mode for DDPG is that the learned Q-function begins to dramatically overestimate Q-values, which then leads to the policy breaking, because it exploits the errors in the Q-function.
- **Trick One: Clipped Double-Q Learning.** TD3 learns *two* Q-functions instead of one (hence "twin"), and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.
- **Trick Two: "Delayed" Policy Updates.** TD3 updates the policy (and target networks) less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.
- **Trick Three: Target Policy Smoothing.** TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

Key Equations: target policy smoothing

TD3 concurrently learns two Q-functions, Q_{ϕ_1} and Q_{ϕ_2}, by mean square Bellman error minimization, in almost the same way that DDPG learns its single Q-function.

- **target policy smoothing.** Actions used to form the Q-learning target are based on the target policy, $\mu_{\theta_{\text{targ}}}$, but with clipped noise added on each dimension of the action. After adding the clipped noise, the target action is then clipped to lie in the valid action range (all valid actions, a, satisfy $a_{Low} \leq a \leq a_{High}$). The target actions are thus:

$$a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \qquad \epsilon \sim \mathcal{N}(0, \sigma)$$

- Target policy smoothing essentially serves as a regularizer for the algorithm. It addresses a particular failure mode that can happen in DDPG: if the Q-function approximator develops an incorrect sharp peak for some actions, the policy will quickly exploit that peak and then have brittle or incorrect behavior. This can be averted by smoothing out the Q-function over similar actions, which target policy smoothing is designed to do.

clipped double-Q learning

- Both Q-functions use a single target, calculated using whichever of the two Q-functions gives a smaller target value:

$$y(r, s', d) = r + \gamma(1 - d) \min_{i=1,2} Q_{\phi_{i,\text{targ}}}(s', a'(s')),$$

  and then both are learned by regressing to this target:

$$L(\phi_1, \mathcal{D}) = \mathop{\mathbf{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_1}(s, a) - y(r, s', d) \right)^2 \right],$$

$$L(\phi_2, \mathcal{D}) = \mathop{\mathbf{E}}_{(s,a,r,s',d)\sim\mathcal{D}} \left[\left(Q_{\phi_2}(s, a) - y(r, s', d) \right)^2 \right].$$

- Using the smaller Q-value for the target, and regressing towards that, helps fend off overestimation in the Q-function.

- the policy is learned just by maximizing Q_{ϕ_1}:

$$\max_{\theta} \underset{s\sim\mathcal{D}}{\mathrm{E}}\left[Q_{\phi_1}(s, \mu_\theta(s))\right],$$

  which is pretty much unchanged from DDPG. However, in TD3, the policy is updated less frequently than the Q-functions are. This helps damp the volatility that normally arises in DDPG because of how a policy update changes the target.

- **Exploration vs. Exploitation**: TD3 trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make TD3 policies explore better, we add noise to their actions at training time, typically uncorrelated mean-zero Gaussian noise. To facilitate getting higher-quality training data, you may reduce the scale of the noise over the course of training.

Pseudocode: TD3

Algorithm 4 Twin Delayed DDPG

```
1: Input: initial policy $\theta$, Q-function $\phi_1$, $\phi_2$, empty replay buffer $\mathcal{D}$. Set $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
2: repeat
3:     Observe state $s$ and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
4:     Execute $a$ in the environment
5:     Observe next state $s'$, reward $r$, and done signal $d$ to indicate whether $s'$ is terminal
6:     Store $(s, a, r, s', d)$ in replay buffer $\mathcal{D}$
7:     If $s'$ is terminal, reset environment state.
8:     if it's time to update then
9:         for $j$ in range(however many updates) do
10:            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
11:            Compute target actions $a'(s') = \text{clip}\left(\mu_{\theta_{\text{targ}}}(s') + \text{clip}(\epsilon, -c, c), a_{Low}, a_{High}\right), \quad \epsilon \sim \mathcal{N}(0, \sigma)$
12:            Compute targets $y(r, s', d) = r + \gamma(1-d) \min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', a'(s'))$
13:            Update Q-functions by one gradient step: $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi,i}(s, a) - y(r, s', d)\right)^2$ for $i = 1, 2$
14:            if $j \mod \texttt{policy\_delay} = 0$ then
15:                Update policy by one step of gradient ascent using $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_{\phi,1}(s, \mu_\theta(s))$
16:                Update target networks with
                       $$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1-\rho)\phi_i \text{ for } i = 1, 2, \;\; \theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1-\rho)\theta$$
17:            end if
18:        end for
19:    end if
20: until convergence
```