CS3243: Introduction to Artificial Intelligence
Lecture Notes 9: Q-Learning

1. Value Iteration

In the previous lecture, we learnt the **Bellman Equation**, which is

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U(s')$$

$R(s)$ refers to the reward received right now, γ refers to the discounting factor, and $\sum_{s'} P(s' \mid s, a)\, U(s')$ refers to the utility in the future.

Suppose that you have n states, and you want to find the utility of all the states. You will hence have a set of n non-linear equations, since max is non-linear. To solve them, use the following algorithm:

Initialization:
Initialize utilities to some value (e.g. 0).

Repeat:
$U_i(s) \leftarrow$ utility value at iteration i
$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U_i(s')$
Termination:
If $\forall s, |U_{i+1}(s) - U_i(s)| < threshold$, then we can stop repeating the above step.

By carrying out this algorithm, we are exploiting the property that

$$\lim_{i \to \infty} U_i(s) = U(s)$$

2. Policy Iteration

Given an optimal policy π^*, the max term in the Bellman Equation would be removed and we would be left with

$$U^{\pi^*}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi^*(s))\, U(s')$$

However, in the case where we do not know what π^* is, to find such an optimal policy, we carry out the following algorithm:

Initialization:
Initialize policy as an arbitrary mapping $\pi_0(s)$: $States \rightarrow Actions$.

Repeat:
Now that we know policy π_i, for each state s, form the equation

$$U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s))\, U^{\pi_i}(s')$$

We have n states, hence we generate n linear equations. Solve them using **Gaussian Elimination**, which runs in $O(n^3)$ time. Following that, update the policy as

$$\pi_{i+1}(s) = \underset{a \in A(s)}{argmax} \sum_{s'} P(s' \mid s, a)\, U^{\pi_i}(s)$$

Convergence:
If $\forall s \lim_{i \to \infty} \pi_i(s) = \pi^*(s)$, we can terminate the loop.

3. Q-Learning

So far, everything we have learnt required us to specify the transition model $T(s, a)$. For every state s, we want to learn action a.

Let us assume that there exists an expression $Q(s, a)$ such that the utility of state s can be expressed as

$$U(s) = \max_{a \in A(s)} Q(s, a)$$

We find that $Q(s, a)$ can be defined as the following expressions.

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' \mid s, a)\, U(s')$$

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a' \in A(s')} Q(s', a')$$

$Q(s, a)$ can be interpreted as the sum of the reward right now and the expected utility in the future after taking action a. To find $Q(s, a)$, we make use of this algorithm:

Initialization:
Initialize $\hat{Q}(s, a)$.

Repeat:
`ChooseAction(s)`, go to new state s, `update` $\hat{Q}(s, a)$.

The way in which we choose an action a from a state s is the following:

`ChooseAction(s):`
$\hat{a} \leftarrow \underset{a \in A(s)}{argmax}\, \hat{Q}(s, a)$

With probability β we choose $\hat{a}$, and with probability $(1-\beta)$ we choose an action randomly or based on $\hat{Q}(s,a)$, depending on the problem.

The way in which we update $\hat{Q}(s,a)$ is the following:

`Update()`:
Update $\hat{Q}(s,a) \leftarrow \alpha\left(R(s) + \gamma \max_{a'} Q(s',a')\right) + (1-\alpha)\,\hat{Q}(s,a)$

Initially, we want α to be high, but with time we want $\alpha \to 0$.

- We can have a parameter t that keeps track of the time since the start of the algorithm and define $\alpha(t) = \frac{1}{t}$ or some other function that decreases with t.
- Alternatively, we can have a counter $N(s,a)$ that counts how many times we have taken action a at state s, and define $\alpha(N(s,a))$ as a decreasing function in $N(s,a)$.

4. Approximate Q-Learning

The space of (s,a) can be very large, so we want to reduce it via **Dimensionality Reduction**.

For example, in a game of chess, the number of possible states on a chessboard can be very large due to the features like number of pawns, whether each queen is in the upper half or lower half, etc. However, we can choose to come up with a small number of features such that multiple states can map to a small number of features.

We can express $Q(s,a)$ as a linear combination of features, with each feature being assigned a weight as seen below,

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a)\, w_i$$

The updating of $\hat{Q}(s,a)$, as covered in the previous session, can also be rewritten as

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha[R(s) + \gamma \max_{a'} Q(s',a') - \hat{Q}(s,a)]$$

To perform the updating of weights,

Initialization:
Initialize the weights as $\widehat{w}_i$ such that

$$\hat{Q}(s,a) = \sum_{i=1}^{n} f_i(s,a)\, \widehat{w}_i$$

Update weights:

$$\widehat{w}_i \leftarrow \widehat{w}_i + \alpha[R(s) + \gamma \max_{a'} Q(s', a') - \hat{Q}(s, a)](\frac{\partial \hat{Q}(s,a)}{\partial \widehat{w}_i})$$

Note that $\frac{\partial \hat{Q}(s,a)}{\partial \widehat{w}_i}$ is the gradient of $\hat{Q}(s, a)$ with respect to $\widehat{w}_i$. It can be viewed as the same as $f_i(s, a)$.