7 Reinforcement Learning

Our MDP-based approach works well when the entire transition model can be specified. However, in real-life situations such as autonomous vehicle driving, the probability of transitioning to another state for a given action at the current state could be unknown. Therefore, the idea that we will find out the optimal policy in advance is not going to work.

There comes the *Reinforcement Learning*, the idea of explore and learn. For the strategy of reinforcement learning, we program the agent to observe rewards as to learn to identify an optimal (or near optimal) policy in the environment. The agent need not have a complete model of the environment or have knowledge of the reward function.

7.1 Q-learning Agent

An agent generally must make a trade-off between exploitation to maximize its current rewards (reflected in expected utility estimations), versus exploration that contributes to learning the true model and helps promote long-term well-being. A good strategy carefully balances this trade-off to maximize long-term expected utility.

In Q-learning, for every state s, we want to learn about the utility of an action a without knowing the transition model. We can denote a state-action pair as the tuple (s, a). Next, we modify the Bellman equation to define the Q-function $Q(s, a)$.

$$\begin{aligned} U(s) &= R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \\ &= \max_{a \in A(s)} \left(R(s) + \gamma \sum_{s'} P(s'|s,a)U(s') \right) \\ &= \max_{a \in A(s)} Q(s,a) \end{aligned}$$

Where we define $Q(s, a)$ to be,

$$\begin{aligned} Q(s,a) &= R(s) + \gamma \sum_{s'} P(s'|s,a)U(s') \\ &= R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a' \in A(s')} Q(s',a') \end{aligned}$$

The Q function can be interpreted as the reward right now plus the expected utility in the future after taking action a, assuming optimal actions are taken in the future.

However, because we do not know the transition model, $P(s'|s, a)$, we assume $P(s'|s, a) = 1$ for now to get,

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

However, this assumption tends to overestimate the actual expected utility in the future. Thus, to compensate for that, each time we update the agent's estimate of Q, that is $\hat{Q}$, we weight the new estimate by α and the previous estimate by $1 - \alpha$.

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

Equivalently, rearranging the terms,

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

The equation above is also known as the **Bellman update**. Initially, we want α to be high (close to 1) but then tends to 0 with time. An example of α as a function of time is the following.

$$\alpha(t) = \frac{1}{t}$$

α can take any function that decreases with input t. To make α dependent on (s,a), we could have a dictionary $N(s,a)$ that tracks how many times the agent has taken action a at state s. We can then define alpha as $\alpha(N(s,a))$, where alpha takes in $N(s,a)$ as an input instead of time, but is still decreasing in $N(s,a)$.

7.2 Approximate Q-Learning

One fundamental weakness of Q Learning is that it may require the agent to store numerous tuples of state-action combinations, and iterate over them many times to learn how to 'play the game'. This can be infeasible for more complex games such as chess which as 10^4 possible states (not even considering actions), or even real world problems.

To reduce the state-action space, we attempt to project the problem onto some other space that is based on features instead. Let function $f_i(s,a)$ denote a feature i of state s and action a that returns some numerical value. For example, features for chess could be the number pawns, rooks or bishops. We now define a Q function that is an approximate function of the original Q function, as a weighted linear function of the features.

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

Thus, in the Approximate Q-Learning algorithm, we no longer need to store a huge dictionary of $\hat{Q}$ values for each state and action. Rather, with some predefined functions f_i corresponding to each estimated weight $\hat{w_i}$ for all $i \in [1,n]$, we can compute $\hat{Q}$ with the following function.

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

Moreover, instead of directly updating $\hat{Q}$ values at each iteration, the weights are updated instead.

$$\hat{w_i} \leftarrow \hat{w_i} + \alpha \left(R(s) + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a) \right) \frac{\partial \hat{Q}(s,a)}{\partial \hat{w_i}}$$

For the case where $Q(s,a)$ is defined as a weighted linear function of features, $\frac{\partial \hat{Q}(s,a)}{\partial \hat{w_i}} = f_i(s,a)$

For the Q-Learning algorithm, we make the assumption that the agent has access to the functions $f_1(s,a), f_2(s,a), \cdots f_n(s,a)$, as well as weights $\hat{w_1}, \hat{w_2}, \cdots, \hat{w_n}$ in addition to the data structures discussed before. The dictionary $\hat{Q}[s]$ is now only used to store the terminal state and the associated rewards.

Naturally, the more useful parameters (or features) we include, the better the approximation of the true utility. But more parameters also increase the computational complexity as a cost.