CS3243: Introduction to Artificial Intelligence **Fall 2020**

Lecture 9: October 21

Lecturer: Prof. Kuldeep S. Meel *Scribe: Ang Zheng Yong*

9.1 Markov Decision Processes

Recall that in our last lecture, we have discussed the Bellman equation:

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

where

1. $R(s)$ is the reward that we would receive in the current iteration.
2. γ is the discounting factor (i.e. reduces the size of future rewards).
3. $\max_{a \in A(s)}$ indicates that we are adopting the optimal policy.
4. $\sum_{s'} P(s'|s,a)U(s')$ is the expected utility that we would receive in future.

However, max is a nonlinear operator, and thus we cannot use linear algebra techniques to solve the Bellman equation.

Remark: Why is max nonlinear?

Algorithm 1 max

```
1: procedure MAX(x_1, x_2)
2:     if x_1 > x_2 then
3:         return x_1
4:     else
5:         return x_2
6:     end if
7: end procedure
```

The above algorithm clearly can't be represented using a sequence of linear equations.

9.1.1 Value iteration

How then could we efficiently calculate $U(s)$? We could apply the following algorithm:

1. Initialize all utilities $U(s)$ to some value, e.g. 0.
2. Update $U_i(s)$ according to the following equation:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

   - Here, $U_i(s)$ represents the utility value at iteration i, and we will continuously update the utility via a while loop.

3. We will stop the algorithm when $|U_{i+1}(s) - U_i(s)| < \epsilon$, where ϵ is some arbitrarily small threshold.
   - In other words, our algorithm will converge to some optimal value $U(s)$ after a number of iterations, i.e. $\forall s, \lim_{i \to \infty} U_i(s) = U(s)$.

9.1.2 Policy iteration

There is another way of solving the Bellman equation. For instance, wouldn't it be great if we did not have the max operator? Should we know the optimal policy, then we could easily remove the max operator from the Bellman equation. The updated Bellman equation would be

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

However, we often do not know what the optimal policy is. The idea to start with some policy π_0 and attempt to find π^*; this is known as **policy iteration**. The algorithm is:

1. Initialize policy $\pi_0 : \textit{States} \rightarrow \textit{Actions}$ (some arbitrary mapping).
2. Compute $U^{\pi_i}(s)$ according to the following equation:

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

   - Since the equation is linear, $U^{\pi_i}(s)$ can be computed for all s using Gaussian Elimination in $O(n^3)$ time.

3. Update $\pi_{i+1}(s)$ according to the following equation:

$$\pi_{i+1}(s) \leftarrow \arg \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U^{\pi_i}(s')$$

4. Eventually, the algorithm will converge to the optimal policy, i.e. $\forall s, \lim_{i \to \infty} \pi_i(s) = \pi^*(s)$.
   - How long would it take for the algorithm to converge? At the moment, this is still an open research question. However, the algorithm has worked well in practice so far.

9.2 Reinforcement Learning

So far, everything we have discussed requires us to specify the transition model $T(s, a)$, which could be potentially very big. For instance, a mere game of chess could easily comprise of approximately 10^{40} states. In other real-world applications, such as autonomous vehicles, there are many more parameters to specify, and thus the number of states would be much bigger.

The key ideas of this section include **exploration** and **exploitation**.

9.2.1 Q-learning

For every state s, we would want to learn the action a. More specifically, we want our algorithm to learn the tuples (s, a). We could achieve this by modifying our Bellman equation accordingly:

$$\begin{aligned} U(s) &= R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s, a)U(s') \\ &= \max_{a \in A(s)} \left\{ R(s) + \gamma \sum_{s'} P(s'|s, a)U(s') \right\} \\ &= \max_{a \in A(s)} Q(s, a) \end{aligned} \tag{9.1}$$

where $Q(s, a)$ can be represented as the following recurrence:

$$\begin{aligned} Q(s, a) &= R(s) + \gamma \sum_{s'} P(s'|s, a)U(s') \\ &= R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a' \in A(s)} Q(s', a') \qquad \text{from equation 9.1} \end{aligned} \tag{9.2}$$

In equation 9.2, $Q(s, a)$ can be interpreted as the reward in the current iteration + the expected future utility from taking action a. Now, we will be able to obtain $Q(s, a)$ with the following algorithm:

1. Initialize $\hat{Q}(s, a)$.
2. Choose an action a, get into a new state s', and update $\hat{Q}(s, a)$.
   - We will update $\hat{Q}(s, a)$ according to the following equation:

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

     In other words, we will choose the action $\hat{a} = \arg\max_{a \in A(s)} \hat{Q}(s, a)$ with probability α, and choose randomly or based on $\hat{Q}(s, a)$ with probability $1 - \alpha$.
   - α is also known as the **learning rate**. Initially, we would want α to be high (i.e. exploration), but with time, α should tend towards 0 (i.e. exploitation). Thus, α can be defined as:
     - $\alpha(t) = \frac{1}{t}$, or some function that decreases with t, where t is the time (e.g. number of iterations) since the start of the algorithm, or
     - $\alpha(N(s, a))$, some function that decreases with $N(s, a)$, where $N(s, a)$ represents the number of times that we have taken action a at state s.
   - Notice that this idea is similar to that of simulated annealing, where we will allow the algorithm to "make mistakes" with a certain probability $1 - \alpha$.

9.2.2 Approximate Q-learning

The space of $\{(s, a)\}$ can be very large, and we might want to reduce it. This can be achieved by adopting an alternate formula for $Q(s, a)$,

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

where

- $f_i(s, a)$ represents the features of the model, and
- w_i represents the weight attributed to each feature.

For example, in chess, features may consist of:

- the number of pawns on the board,
- whether the Queen is in the upper half of the board,
- etc.

By expressing $Q(s, a)$ as a linear combination of weights w_i, we will be able to reduce the dimension[1]of the problem. Hence, we would only need to worry about learning the weights w_i.

From the updating of $\hat{Q}(s, a)$, we had

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

Since $\hat{Q}(s, a) = \sum_{i=1}^{n} f_i(s, a)\hat{w}_i$, we can thus update $\hat{w}_i$ with the following equation:

$$\hat{w}_i \leftarrow \hat{w}_i + \alpha(R(s) + \gamma \max_{a' \in A(s)} \hat{Q}(s', a') - \hat{Q}(s, a)) \frac{\delta \hat{Q}(s, a)}{\delta w_i}$$

In fact, for linear combinations, $\frac{\delta \hat{Q}(s,a)}{\delta w_i}$ is indeed $f_i(s, a)$.

[1]see dimensionality reduction