

07-280 Notes

Pre-reading: Approximate Q-Learning

Carnegie Mellon University
School of Computer Science

Contents

1	Why Tabular Q-Learning Falls Short	1
2	Feature Functions	2
3	Approximate Q-Values	2
4	Updating the Weights	3
4.1	Where the Update Comes From	3
4.2	Example	3
5	Tabular vs. Approximate Q-Learning	4

1 Why Tabular Q-Learning Falls Short

So far, we've been doing "tabular" Q-learning, where the agent maintains one number $Q(s, a)$ value for every state-action pair and updates that entry directly after each experience:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

This works well when the number of states is small, but it breaks down immediately in realistic problems. Consider Pac-Man on a grid with just 100 food locations: because each location can independently contain food or not, the number of distinct states is 2^{100} , far too many to store in a table, let alone visit enough times to learn reliable values.

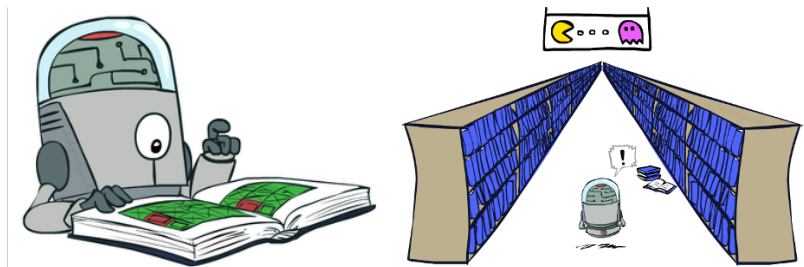


Image credit: Ketrina Yim, UC Berkeley

Worse, the table provides *no generalization*. The three boards below are nearly identical, but a lookup table treats each as a completely unrelated state; experience on any one board teaches the agent nothing about the others.

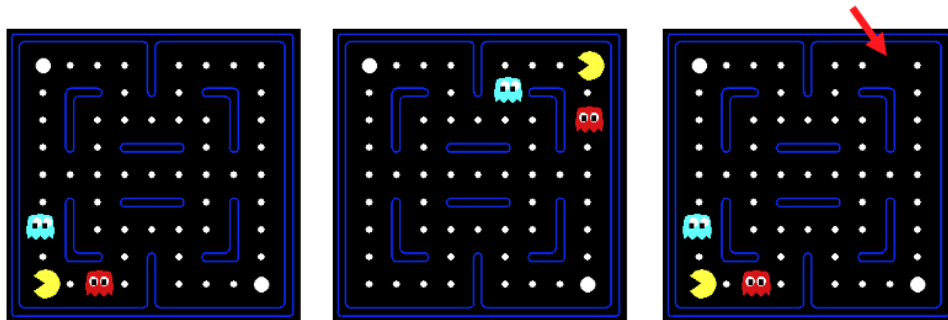


Image credit: UC Berkeley AI

We need *generalization*: similar states should yield similar Q-values.

2 Feature Functions

Generalization requires describing states in terms of meaningful **features** rather than raw identities. A **feature function** $f_i(s, a)$ maps a state-action pair to a real number capturing one aspect of the situation. Example features for Pac-Man are shown below:

Example features:

- Distance to closest ghost
- Distance to closest dot
- Number of ghosts
- $1 / (\text{dist to dot})^2$
- Is Pac-Man in a tunnel? (0/1)



Figure credit: UC Berkeley AI

Feature functions are human-designed and fixed; they do not change during learning. What the agent *learns* is how much weight to assign each feature.

3 Approximate Q-Values

Instead of a lookup table, represent the Q-value as a *linear combination* of features:

$$Q_{\mathbf{w}}(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_M f_M(s, a) = \sum_{i=1}^M w_i f_i(s, a)$$

The weight vector $\mathbf{w} = (w_1, \dots, w_M)$ is all the agent needs to store, just M numbers instead of one per state-action pair.

Why this generalizes Two states that look different in raw form may have very similar feature values. The Pac-Man boards above would produce nearly identical feature vectors, so $Q_{\mathbf{w}}$ assigns them nearly the same Q-value. Learning transfers automatically across similar states.

4 Updating the Weights

The temporal-difference structure from tabular Q-learning carries over unchanged. After the agent takes action a in state s , receives reward r , and observes next state s' , compute the **sample** (TD target):

$$sample = r + \gamma \max_{a'} Q_{\mathbf{w}}(s', a')$$

Then update *each weight* by:

$$w_i \leftarrow w_i + \alpha (sample - Q_{\mathbf{w}}(s, a)) f_i(s, a)$$

In words: if $sample > Q_{\mathbf{w}}(s, a)$ the prediction was too low, so increase the weights of features that were active (large f_i). If the sample is lower than the prediction, decrease them.

4.1 Where the Update Comes From

The update rule is gradient descent on the **loss** $\ell(\mathbf{w}) = \frac{1}{2} (sample - Q_{\mathbf{w}}(s, a))^2$ (the $\frac{1}{2}$ is included for convenience to cancel the 2 in the gradient). With a single feature for clarity,

$$Q_w(s, a) = w f(s, a), \quad \ell(w) = \frac{1}{2} (sample - w f(s, a))^2$$

Taking the derivative with respect to w (chain rule):

$$\frac{\partial \ell}{\partial w} = -(sample - w f(s, a)) \cdot f(s, a)$$

Gradient descent steps *opposite* the gradient:

$$w \leftarrow w - \alpha \frac{\partial \ell}{\partial w} = w + \alpha (sample - w f(s, a)) f(s, a)$$

Extending to multiple features gives the update rule above, because $\partial Q_{\mathbf{w}} / \partial w_i = f_i(s, a)$.

4.2 Example

Suppose an agent has two features and is in state s taking action a :

	Feature value	Initial weight
$f_1(s, a)$	3	$w_1 = 1.0$
$f_2(s, a)$	1	$w_2 = 2.0$

With $\alpha = 0.5$ and $\gamma = 0.5$:

Step 1. Compute the current Q-value:

$$Q_{\mathbf{w}}(s, a) = 1.0 \cdot 3 + 2.0 \cdot 1 = 5$$

Step 2. Take action a . Receive $r = 6$ and reach a terminal state s' (so $\max_{a'} Q_{\mathbf{w}}(s', a') = 0$):

$$sample = 6 + 0.5 \times 0 = 6$$

Step 3. Update each weight ($sample - Q_{\mathbf{w}}(s, a) = 6 - 5 = 1$):

$$w_1 \leftarrow 1.0 + 0.5 \times 1 \times 3 = 2.5 \quad w_2 \leftarrow 2.0 + 0.5 \times 1 \times 1 = 2.5$$

Both weights increased because the sample exceeded the prediction: the agent underestimated the Q-value and should predict higher from states with these feature values.

5 Tabular vs. Approximate Q-Learning

	Tabular	Approximate
What is stored	$Q(s, a)$ for every (s, a)	weight vector $\mathbf{w}$ (M numbers)
What is updated	$Q(s, a)$ directly	each w_i
Update rule (linear)	$Q \leftarrow Q + \alpha (sample - Q)$	$w_i \leftarrow w_i + \alpha (sample - Q_{\mathbf{w}}) f_i$
Generalizes?	No	Yes (via shared weights)
Memory	$ S \times A $	M (number of features)
Requires visiting state	Every (s, a) many times	Features must cover new states

The core idea is the same in both cases: use experience to push the current prediction toward a bootstrapped sample of the true Q-value. The only difference is *what* gets updated, a single table entry versus a shared weight that influences every state with a similar feature profile.