In this lecture...

- Introduction to temporal-difference learning
- SARSA: On-policy TD control
- Q-learning: Off-policy TD control
- Planning and learning with tabular methods
Temporal-difference (TD) learning

Temporal-difference methods are similar to

**Dynamic programming** update estimates based in part on other learned estimates, without waiting for the final outcome (they bootstrap)

**Monte Carlo methods** learn directly from raw experience without a model of the environment’s dynamics
TD prediction

- TD methods only wait until the next time step to update the value estimates.
- At time $t + 1$ they immediately form a target and make an update using the observed reward $r_{t+1}$ and the current estimate $V(S_{t+1})$.

$$V(s_t) \leftarrow V(s_t) + \alpha (r_{t+1} + \gamma V(s_{t+1}) - V(s_t)),$$

where $\alpha > 0$ is a step-size parameter.
- Note that this is similar to the MC update except that it takes place at every step.
- Similar to DP methods, the TD method bases its update in part on an existing estimate – a bootstrapping method.
**TD error** arises in various forms throughout reinforcement learning

\[ \delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \]

The TD error at each time is the error in the estimate made at that time. Because the TD error at step \( t \) depends on the next state and next reward, it is not actually available until step \( t + 1 \). Updating the value function with the TD-error is called a **backup**. The TD error is related to the Bellman equation.
SARSA: On-policy TD control

- TD prediction for control ie action-selection
- A generalised policy iteration method
- Balances between exploration and exploitation
- Learns tabular Q-function

\[
Q(s_t, a_t) \leftarrow (s_t, a_t) + \alpha (r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))
\]

This update is done after every transition from a non-terminal state \(s_t\). If \(s_{t+1}\) is terminal, then \(Q(s_{t+1}, a_{t+1})\) is defined as zero. This rule uses every element of the quintuple of events, \((s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})\), hence the name.
Algorithm 1 SARSA

1: Initialise $Q$ arbitrarily, $Q(terminal, \cdot) = 0$
2: repeat
3: Initialize $s$
4: Choose $a \epsilon$-greedily
5: repeat
6: Take action $a$, observe $r, s'$
7: Choose $a' \epsilon$-greedily
8: $Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma Q(s', a') - Q(s, a))$
9: $s \leftarrow s', a \leftarrow a'$
10: until $s$ is terminal
11: until convergence
Properties of SARSA

- SARSA is an on-policy algorithm which means that while learning the optimal policy it uses the current estimate of the optimal policy to generate the behaviour.
- SARSA converges to an optimal policy as long as all state-action pairs are visited an infinite number of times and the policy converges in the limit to the greedy policy ($\epsilon = \frac{1}{t}$).
In Q-learning the learned action-value function, $Q$, directly approximates the optimal action-value function, independent of the policy being followed.

$$Q(s_t, a_t) \leftarrow (s_t, a_t) + \alpha \left( r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$

This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs: all that is required for correct convergence is that all pairs continue to be updated.
Algorithm 2 Q-learning

1: Initialise $Q$ arbitrarily, $Q(terminal, \cdot) = 0$
2: repeat
3: Initialize $s$
4: repeat
5: Choose $a$ $\epsilon$-greedily
6: Take action $a$, observe $r$, $s'$
7: $Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
8: $s \leftarrow s'$
9: until $s$ is terminal
10: until convergence
**SARSA vs Q-learning**

Comparison of the SARSA and the Q-learning algorithm on the cliff-walking task (a variant of grid-world). The results show the advantage of on-policy methods during the learning process.

![Graph comparing SARSA and Q-learning](image)

**Figure 6.5:** The cliff-walking task. The results are from a single run, but smoothed by averaging the reward sums from 10 successive episodes.

The lower part of Figure 6.5 shows the performance of the Sarsa and Q-learning methods with $\epsilon$-greedy action selection, $\epsilon = 0.1$. After an initial transient, Q-learning learns values for the optimal policy, that which travels right along the edge of the cliff. Unfortunately, this results in its occasionally falling off the cliff because of the $\epsilon$-greedy action selection. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its on-line performance is worse than that of Sarsa, which learns the roundabout policy. Of course, if $\epsilon$ were gradually reduced, then both methods would asymptotically converge to the optimal policy.

**Exercise 6.9**

Why is Q-learning considered an off-policy control method?
Expected Sarsa

- An alternative to taking a random action and using the estimate of the Q-function for that action in TD-error (as in SARSA) is to use the expected value of the Q-function.

\[ Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha (E[Q(s_{t+1}, a_{t+1}) \mid s_{t+1}] - Q(s_t, a_t)) \]

\[ = Q(s_t, a_t) + \alpha \left( r_{t+1} + \gamma \sum_{a'} \pi(a' \mid s_{t+1}) Q(s_{t+1}, a') - Q(s_t, a_t) \right) \]

- Although computationally more complex, this method has a lower variance.
- Generally performs better and it can be either on-policy or off-policy.
Summary

- Prediction: the value function must accurately reflect the policy
- Improvement: the policy must improve locally (e.g., $\epsilon$-greedy) with respect to the current value function
- SARSA is an on-policy TD method
- Q-learning is an off-policy TD method
- Expected SARSA can be either an on-policy or an off-policy method
- They can be applied on-line, with a minimal amount of computation, to learn from interaction with an environment
Planning and learning with tabular methods

A unified view of

Planning Methods which require the model of the environment

Learning Methods which do not require the model of the environment
Models and planning

**Model** of the environment – anything that an agent can use to predict how the environment will respond to its actions. Models can be used to *simulate* experience: given a starting state and action, the model produces a possible transition.

**Planning** – any computational process that takes a model as input and produces or improves a policy for interacting with the modelled environment.
Planning

Planning is based on two basic ideas:

1. all state-space planning methods involve computing value functions as a key intermediate step toward improving the policy

2. they compute their value functions by backup operations (TD updates) applied to simulated experience.
Dyna: integrating planning, acting, and learning

A planning agent can be used to:

- **model-learning** improve the model (to match the real environment)
- **reinforcement learning** directly improve the value function and policy

---

**Figure 1**: Planning agent
Dyna-Q includes all of the processes shown in Figure 1: planning, acting, model-learning, and direct RL – all occurring continually.

**Planning** the Q-learning applied to samples from the model

**Model-learning** table-based and assumes the world is deterministic

**RL** after each transition $s_t, a_t; \rightarrow r_{t+1}, s_{t+1}$, the model records in its table entry for $s_t, a_t$ the prediction that $r_{t+1}, s_{t+1}$ will deterministically follow.

The planning algorithm randomly samples only from state-action pairs that have previously been experienced, so the model is never queried with a pair about which it has no information.
by recognizing the similarities between these two sides than by opposing them. For example, in this book we have emphasized the deep similarities between dynamic programming and temporal-difference methods, even though one was designed for planning and the other for model-free learning.

Dyna-Q includes all of the processes shown in Figure 8.1—planning, acting, model-learning, and direct RL—all occurring continually. The planning method is the random-sample one-step tabular Q-planning method given in Figure 8.1. The direct RL method is one-step tabular Q-learning. The model-learning method is also table-based and assumes the world is deterministic. After each transition \( S_t, A_t \rightarrow R_{t+1}, S_{t+1} \), the model records in its table entry for \( S_t, A_t \) the prediction that \( R_{t+1}, S_{t+1} \) will deterministically follow. Thus, if the model is queried with a state–action pair that has been experienced before, it simply returns the last-observed next state and next reward as its prediction. During planning, the Q-planning algorithm randomly samples only from state–action pairs that have previously been experienced (in Step 1), so the model is never queried with a pair about which it has no information.

The overall architecture of Dyna agents, of which the Dyna-Q algorithm is one example, is shown in Figure 8.2. The central column represents the basic interaction between agent and environment, giving rise to a trajectory of real experience. The arrow on the left of the figure represents direct reinforcement learning operating on real experience to improve the value function and the policy. On the right are model-based processes. The model is learned from real experience and gives rise to simulated experience. We use the term search control to refer to the process that selects the starting states and actions for the simulated experiences generated by the model. Finally, planning is achieved by applying reinforcement learning methods to the simulated experiences just as if they had really happened. Typically, as in real direct RL update.

---

**Dyna architecture**

- Policy/value functions
  - planning update
  - simulated experience
- Real experience
  - model learning
  - search control
- Environment
- Model

- Direct RL update
- Search control
Algorithm 3 Tabular Dyna-Q

1: Initialise $Q(s, a)$ and $Model(s, a)$ arbitrarily
2: repeat
3: Initialize $s$
4: Choose $a$ $\epsilon$-greedily
5: Take action $a$, observe $r, s'$ \{real experience\}
6: $Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$ \{RL\}
7: $Model(s, a) \leftarrow r, s'$ \{model learning deterministically\}
8: repeat
9: $s, a$ random previously observed state-action pair \{search control\}
10: $r, s' \leftarrow Model(s, a)$ \{simulated experience\}
11: $Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$ \{planning\}
12: $s \leftarrow s'$
13: until \(n\) times
14: until convergence
Dyna: properties

- Learning and planning are accomplished by exactly the same algorithm, operating on real experience for learning and on simulated experience for planning.
- Planning proceeds incrementally, it is trivial to intermix planning and acting.
- The agent responds instantly to the latest sensory information and yet always plans in the background.
- As new information is gained, the model is updated to better match reality.
Prioritised sweeping

- Upto now simulated transitions in state-action pairs are selected uniformly at random from all previously experienced pairs.
- The number of updates grows rapidly but not all updates are equally useful.
- The value of some state-action pairs have changed a lot while the value of other state-action pairs has changed little.
- In a stochastic environment, variations in estimated transition probabilities also contribute to the magnitude of the change.
Prioritised sweeping

Prioritize the backups according to a measure of their urgency

- Base urgency on the TD-error
- If a state-action pair was updated all state-actions preceding this pair must be updated too.
- Perform the backups in order of priority
Prioritised sweeping for a deterministic env.

Algorithm 4 Prioritised sweeping

1: Initialise $Q(s, a)$, $Model(s, a)$ arbitrarily and $PQueue$ to empty
2: repeat
3: Initialize $s$; choose $a \in$-greedily
4: Take action $a$, observe $r$, $s'$; $Model(s, a) \leftarrow r$, $s'$
5: $P \leftarrow |r + \gamma \max_{a'} Q(s', a') - Q(s, a)|$ \{TD-error\}
6: if $P > \theta$ then insert $s, a$ into $PQueue$ with priority $P$
7: repeat
8: $s, a \leftarrow \text{first}(PQueue)$, $r, s' \leftarrow Model(s, a)$
9: $Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$
10: for all $\hat{s}, \hat{a}$ predicted to lead to $s$ do
11: $\hat{r}$ reward for $\hat{s}, \hat{a}, s$; $P \leftarrow |\hat{r} + \gamma \max_{a} Q(s, a) - Q(\hat{s}, \hat{a})|$
12: if $P > \theta$ then insert $\hat{s}, \hat{a}$ into $PQueue$ with priority $P$
13: end for
14: until $PQueue$ empty
15: until convergence
Benefit of prioritised sweeping

Example 8.5: Rod Maneuvering

The objective in this task is to maneuver a rod around some awkwardly placed obstacles within a limited rectangular work space to a goal position in the fewest number of steps (see Figure 8.8). The rod can be translated along its long axis or perpendicular to that axis, or it can be rotated in either direction around its center. The distance of each movement is approximately $1/20$ of the work space, and the rotation increment is 10 degrees. Translations are deterministic and quantized to one of $20 \times 20$ positions. The figure shows the obstacles and the shortest solution from start to goal, found by prioritized sweeping. This problem is still deterministic, but has four actions and 14,400 potential states (some of these are unreachable because of the obstacles). This problem is probably too large to be solved with unprioritized methods.

Extensions of prioritized sweeping to stochastic environments are straightforward. The model is maintained by keeping counts of the number of times each state–action pair has been experienced and of what the next states were. It is natural then to backup each pair not with a sample backup, as we have been using so far, but with a full backup, taking into account all possible next states and their probabilities of occurring.

Prioritized sweeping is just one way of distributing computations to improve planning efficiency, and probably not the best way. One of prioritized sweeping's limitations is that it uses full backups, which in stochastic environments may waste lots of computation on low-probability transitions. In many cases, sample backups can get closer to the true value function with less computation despite the variance introduced by sampling (see Sutton & Barto, 1998, Section 9.5). Sample backups can win...
Summary

- Planning optimal behaviour and learning optimal behaviour involve estimating the same value functions.
- Any of the learning methods can be converted into planning methods simply by applying them to simulated (model-generated) experience rather than to real experience.
- Prioritized sweeping focuses backward on the predecessors of states whose values have recently changed significantly.
Next lecture

- Approximate solutions and function approximation