# Reinforcement Learning (RL)

Why Reinforcement Learning?

- The task involves a series of actions.

- For a given state, there is no teacher to tell the agent what the "correct" or "optimal" action is from that state.

- External feedbacks (reward/penalty) are available, but not after every action.

- Example scenario: Playing a game without a given evaluation function. Can the agent learn an evaluation function from its experience?

Markov Decision Processes

■ We talk about **Markov Decision Processes** (MDPs) because these are how the tasks of RL are represented.

■ The content of a MDP consists of:

● States.

● Allowed actions of the states.

● Immediate rewards of the states.

● Probabilistic transition function $P(s'|s,a)$.
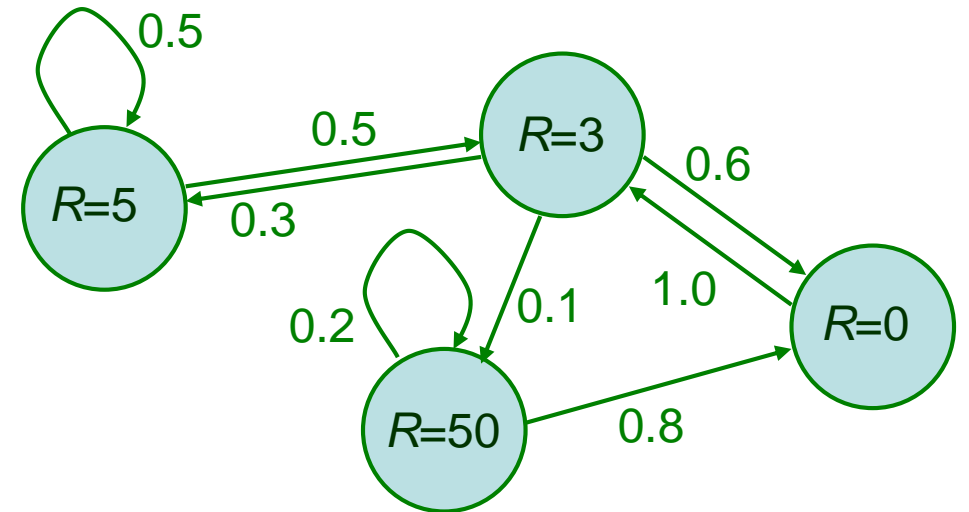
Markov Decision Processes

■ In a MDP, we can not find an optimal "path" because it is stochastic.

■ What we try to optimize is a **policy**, $\pi(s)$, which is a function from states to actions.

■ A **state-value function** (often called just "value function") of a state, $V^\pi(s)$, represents the "expected total rewards" when we start from state s with policy π.

- This is like the averaged reward of all the possible paths from the given state to the terminal states, weighted by probabilities of the paths.

- We represent the optimal policy as π^* and the corresponding optimal state-value function as $V^*(s)$.

Rewards

■ The total reward of a path with state sequence s_0, s_1, s_2, … is given by

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$$

■ **Discounted Reward**:.

- Far-away future rewards are less important than immediate rewards (when $\gamma < 1$).

- Here γ is called the discount factor.

- When $\gamma = 1$, we say we have additive rewards.

Policy, Reward and State-Values

Bellman's equation (fixed policy):

Reward

Probabilistic transition function

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

State-values

Discount Factor

Policy (action at s)

Bellman's equation (optimal policy):

$$V^{*}(s) = R(s) + \gamma \max_{a} \sum_{s'} P(s' | s, a) V^{*}(s')$$

Value Iteration

■ **Bellman update**: We take the Bellman's equation for optimal policy, and convert it to right-to-left assignment. Applying it iteratively allows us to estimate the state-value function (when the policy is optimal):

$$V(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V(s')$$

■ We start with all-zero state values.

■ The optimal policy is linked to the estimated state-value function as

$$\pi^*(s) = \arg\max_a \sum_{s'} P(s'|s,a) V(s')$$

Policy Iteration

Alternating iteration of these two steps:

■ **Policy Evaluation**: Estimate the state-value function using the current policy:

$$V^{\pi}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^{\pi}(s')$$

● For small MDPs, this can be solved exactly.

● For larger MDPs, we can just apply several iterations of Bellman update.

● The policy is randomly initialized.

■ (Greedy) **Policy Improvement**: Update the policy so that it is optimal with the current state-value function:

$$\pi(s) \leftarrow \arg\max_{a} \sum_{s'} P(s'|s, a) V(s')$$

Policy Parameterization

- While we have a very large state space, the policy can no longer be expressed as a look-up table (like we have assumed so far).

- What we can do is to represent the policy with a set of parameters (e.g., using a neural network).

- Policy optimization becomes the optimization of the parameters.

- With policy parameterization, we commonly have a soft (continuous) representation of the policy, $\pi^\theta(a|s)$, where θ represents the set of parameters.

- With a soft policy, a small change on the parameters only leads to a small change of the agent behavior. This allows for easier optimization.

- The parameters can be optimized using gradient based methods (**policy gradient**) or methods that do not use gradients, such as evolutionary methods.

# Model-Based vs. Model-Free Learning

- So far, we have assumed that the MDP (in particular, the transition function) is known. Methods based on this assumption are considered **model-based**. However, this is not the case for many real-world problems.

- We need methods to learn a policy without knowing or estimating the MDP; such methods are considered **model-free**.

- Such learning is achieved by **sampling** paths through the state space in order to collect information.

- Monte-Carlo Tree Search is a kind of such sampling.

REINFORCE

- **REINFORCE**, also called **Monte-Carlo Policy Gradient**, is a gradient-ascend method that updates the policy parameters after a path is sampled.

- Updates are favored for the parameters that enhance the probabilities of actions along paths with larger accumulated rewards.

- Objective function:

$$U(\theta) = \sum_{\tau} P(\tau;\theta)\, R(\tau)$$

  - τ : A trajectory (sequence of states in sampled path)

  - $R(\tau)$: Reward of the trajectory

  - $P(\tau;\theta)$: Probability of the trajectory given the policy

- Gradient ascend:

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} U(\theta)$$

Temporal Difference (TD) Learning

- It is unnecessary to do updates only after complete sample paths (episodes).

- Idea: Use "estimated evaluations at future states" (which tend to be more accurate) to update the evaluation of the current state.
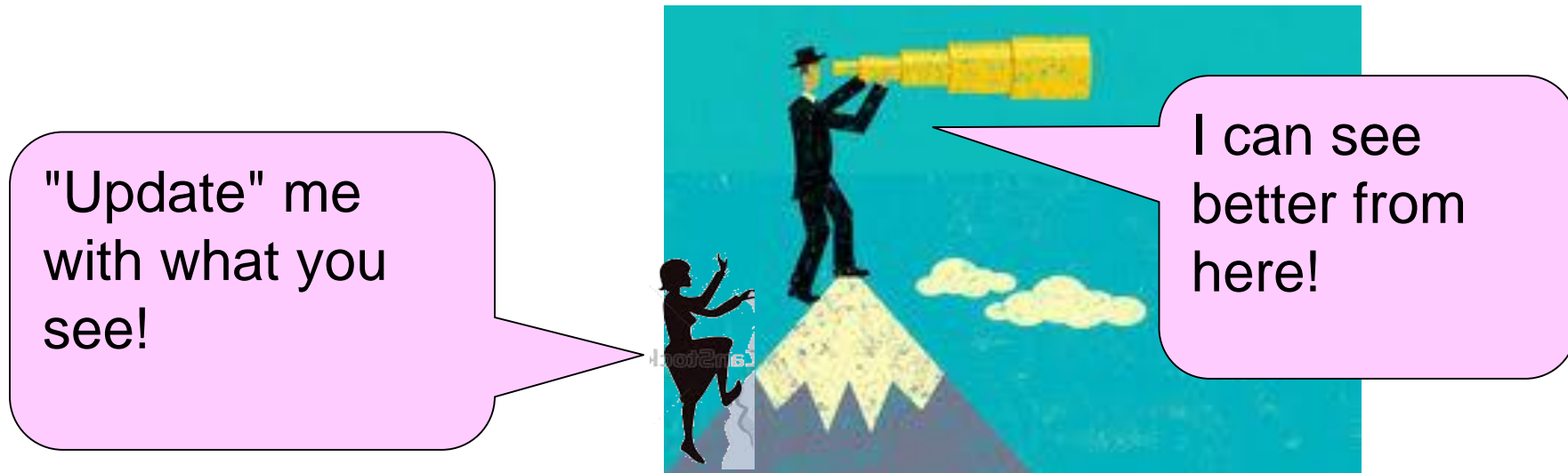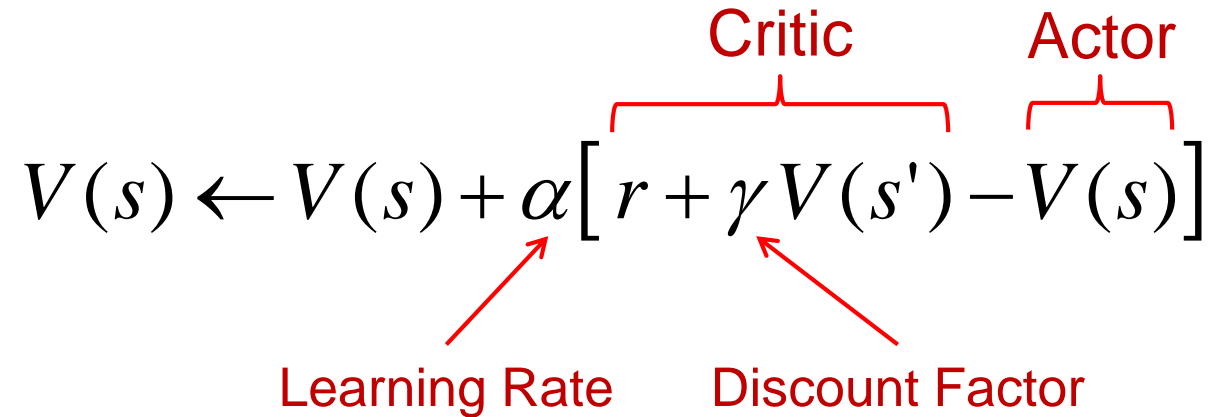
Temporal Difference (TD) Learning

■ A basic TD learning step:

Critic   Actor

$$V(s) \leftarrow V(s) + \alpha \left[r + \gamma V(s') - V(s) \right]$$

Learning Rate   Discount Factor

■ This is an **actor-critic method** in RL. (The "critic" is the supposedly better estimation from a subsequent state.)

Q-Learning

- A really popular approach of reinforcement learning.

- The algorithm follows the idea of TD learning.

- **Q-functions** are **action-value functions**: Each entry, $Q(s,a)$, represents the expected total reward of taking action a at state s.

- The goal: To learn the Q-functions

  - When the learning converges, the best policy is to follow the action with the best Q value.

  - The "expected total reward" assumes that the best-Q-value action is taken for all subsequent steps until a terminal state is reached.

- Q functions, like state-value functions, can be parameterized.

The Q-Learning Algorithm

- Initialize all the Q values (for example, to zero).

- A typical Q-learning procedure is to repeat the following many times (learning episodes):

  - Start at any valid initial state.

  - Repeat until a terminal state is reached:

    - Choose a valid action a from the current state s. Let s' be the resulting new state, and let r be the reward incurred for this action.

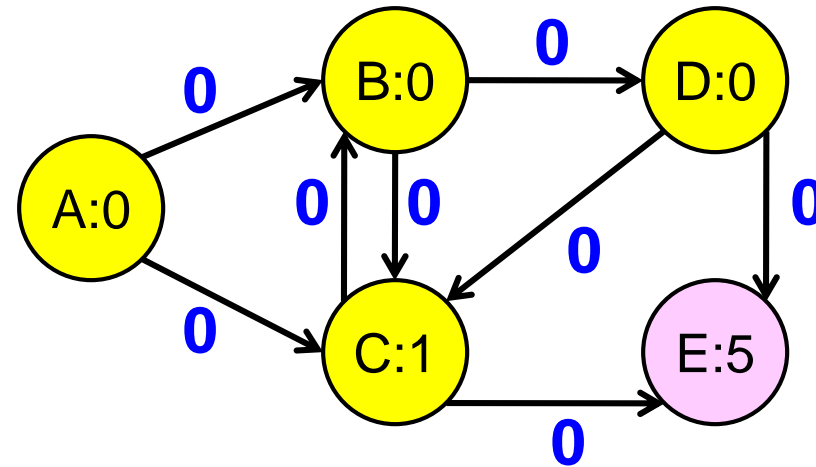   - Update $Q(s,a)$: (This occurs only for non-terminal s.)

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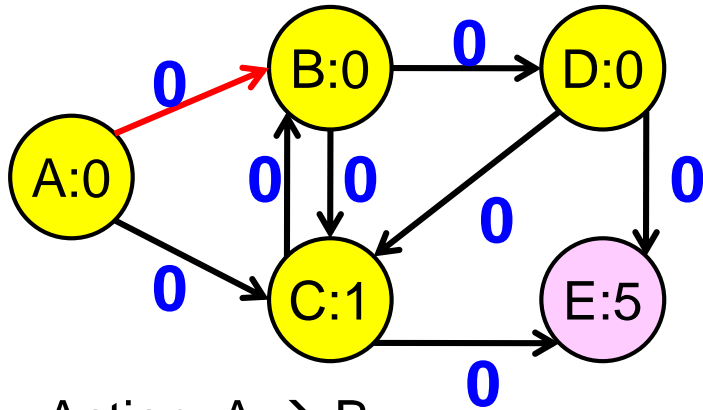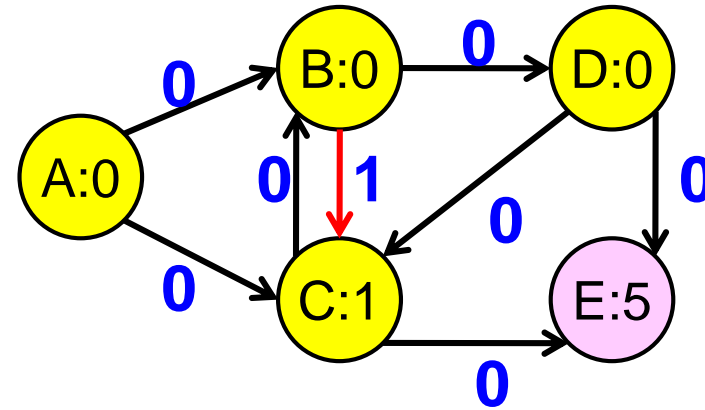
Learning Rate   Discount Factor

Q-Learning Example

■ For simplicity, we will consider only a deterministic environment here, and all the Q values are initialized to zero.

■ Settings (state space given below): Start states = {A}. Learning rate = 1. Discount factor = 1. Terminal states = {E}.
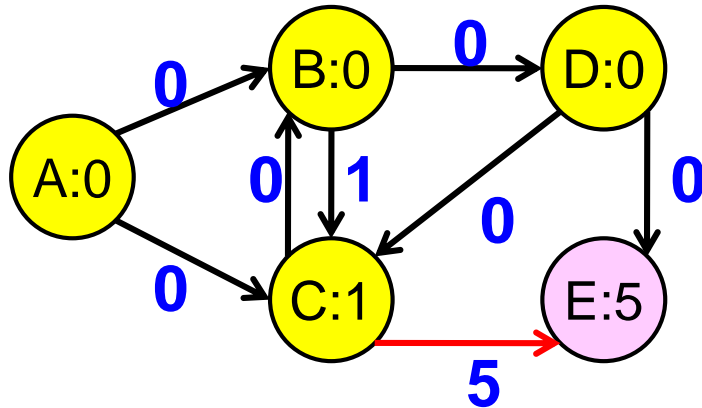
Q-Learning Example

Action: A → B
Q(A, A→B) ← 0

Action: B → C
Q(B, B→C) ← 1 + max(0,0) = 1

Action: C → E
Q(C, C→E) ← 5

new episode
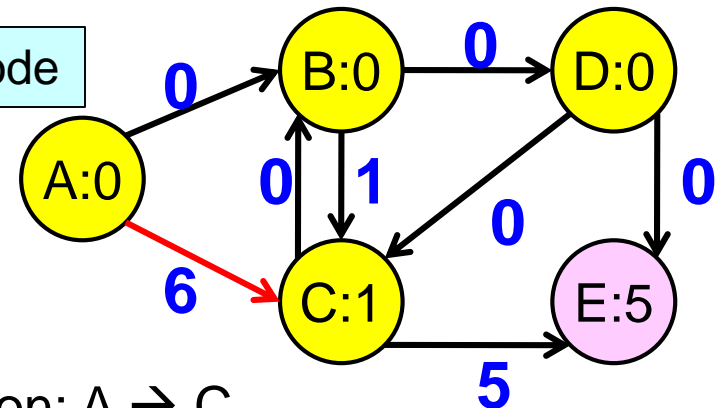
Action: A → C
Q(A, A→C) ← 1+ max(0,5) = 6

Q Tables

- Q-table (the most common representation): A table of values of each combination of a state and a valid action from that state.

- Convenient for problems with finite states and finite valid actions per state.

- Quantization can be used for environments with continuous states and/or actions.
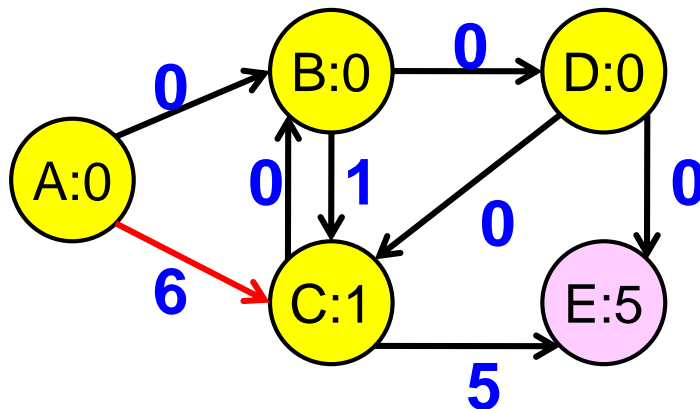
|	→A	→B	→C	→D	→E
A	-	0	6	-	-
B	-	-	1	0	-
C	-	0	-	-	5
D	-	-	0	-	0
E	-	-	-	-	-

Q-Learning: Exploration vs. Exploitation

■ Problem: During a training episode, how to choose the action from a state?

- Best Q-value (greedy approach): Exploitation

- Random action: Exploration

■ ε-greedy: Use a probability (ε) to choose between the two. (More exploration initially, and more exploitation later to facilitate convergence.)

■ (Optional) Adjustment of the discount factor: smaller initially (to avoid propagation of "noise") and larger later.

SARSA vs. Q-Learning

- **SARSA** (State-Action-Reward-State-Action) is a learning method very similar to Q-learning: It also attempts to learn a Q-function.

- The main difference is in how the Q values are updated:

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

  - SARSA: $Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',\pi(s')) - Q(s,a) \right]$

- The policy π in the update equation of SARSA is the policy used to run the episode, so the update is based on what will happen later in this episode.

- In comparison, Q-learning assumes a greedy policy after the current step, which is likely different from the current policy.

- Therefore, Q-learning is considered an off-policy method and SARSA is considered an on-policy method.

SARSA vs. Q-Learning

- During the learning process, SARSA tends to be more conservative as it will try to avoid risks in training episodes.

- SARSA might give a more robust (less risky) policy with limited learning.

- A well-known comparison (Cliff-Walk):
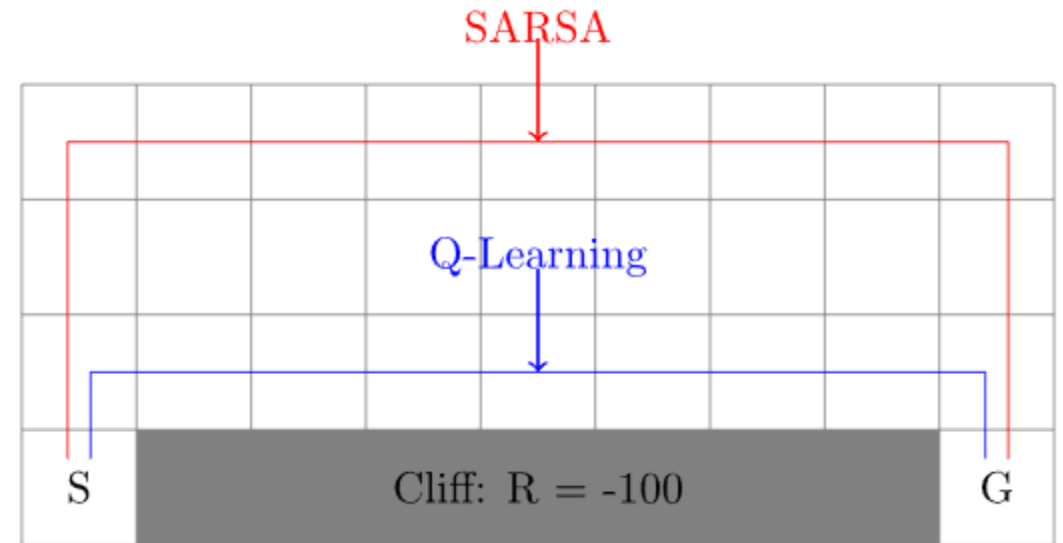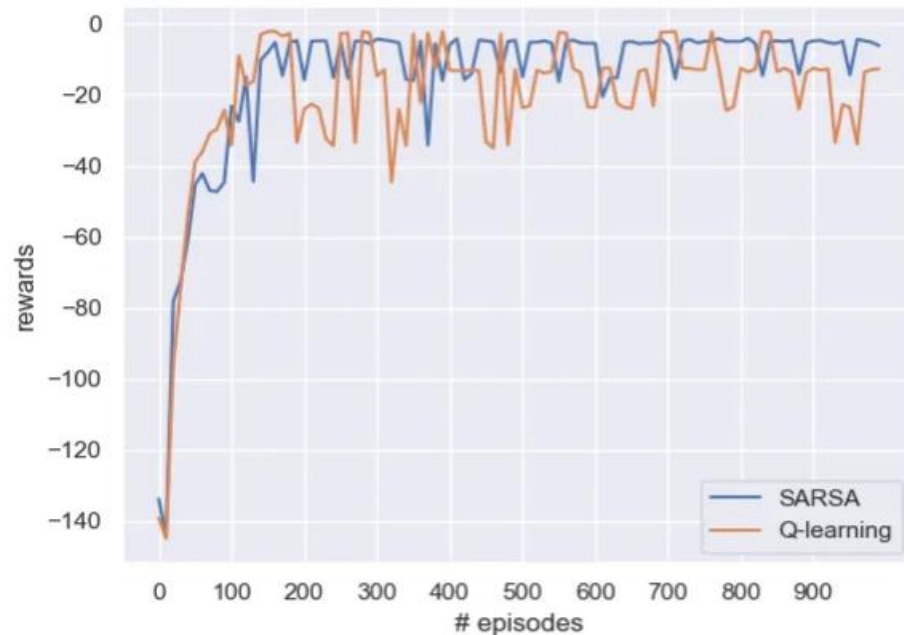
Experience Replay

- The standard practice:

  - Keep a finite-sized buffer of most recent transitions during learning.

  - Periodically sample from the buffer to reuse it for learning.

- Advantages:

  - Reduce the effect of the correlation between consecutive episodes on the learning process.

  - Reusing past transitions may be more efficient than running new episodes.

Deep Q-Learning

■ Use a neural network to represent the Q function.

■ The loss function (which leads to TD learning):

$$(1/2)\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]^2$$

■ The network weights are updated via backpropagation.

■ DeepMind used DQL to make agents that play many Atari games to top human levels (2016).

Case Study: TD-Gammon

- Two-ply game tree search.

- Classical neural network (single hidden layer) for the evaluation function.

- First version: Pure reinforcement learning (TD-lambda) of the evaluation function without human knowledge. ➔ Good but not top performance.

- Second version: The network learns a representation based on a set of expert-designed attributes. ➔ As good as human world champions.