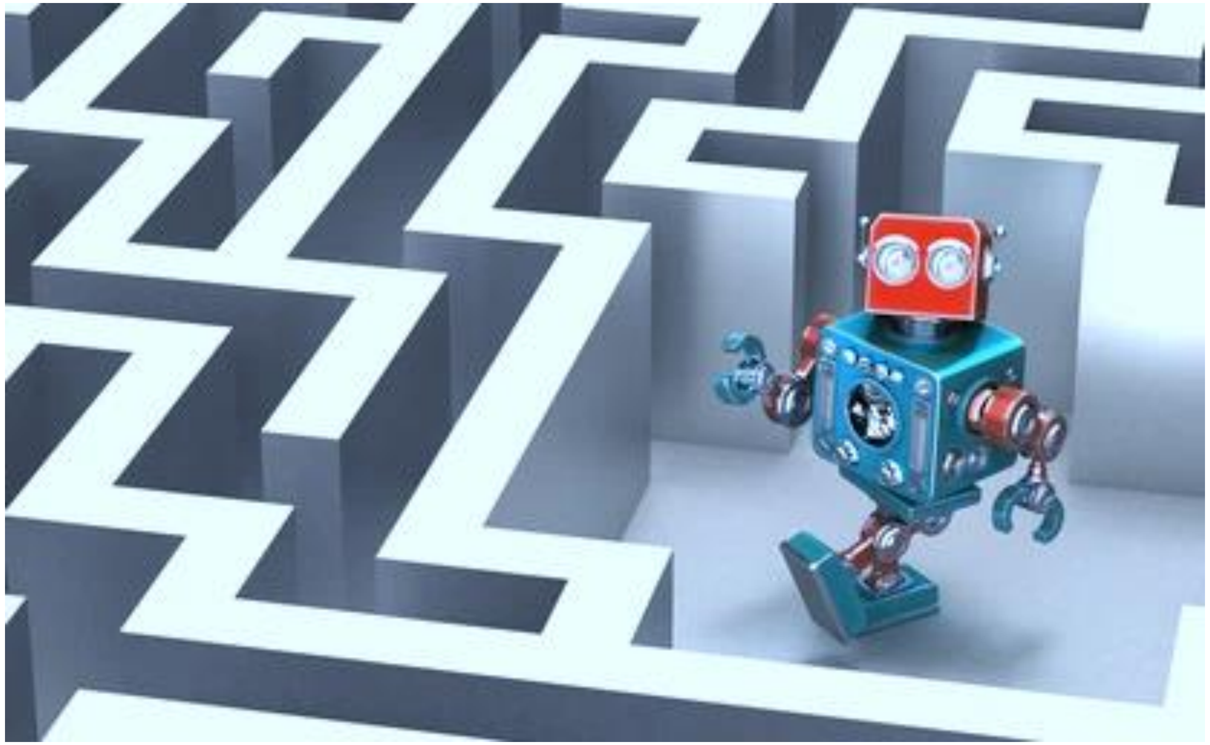
Reinforcement Learning (RL)

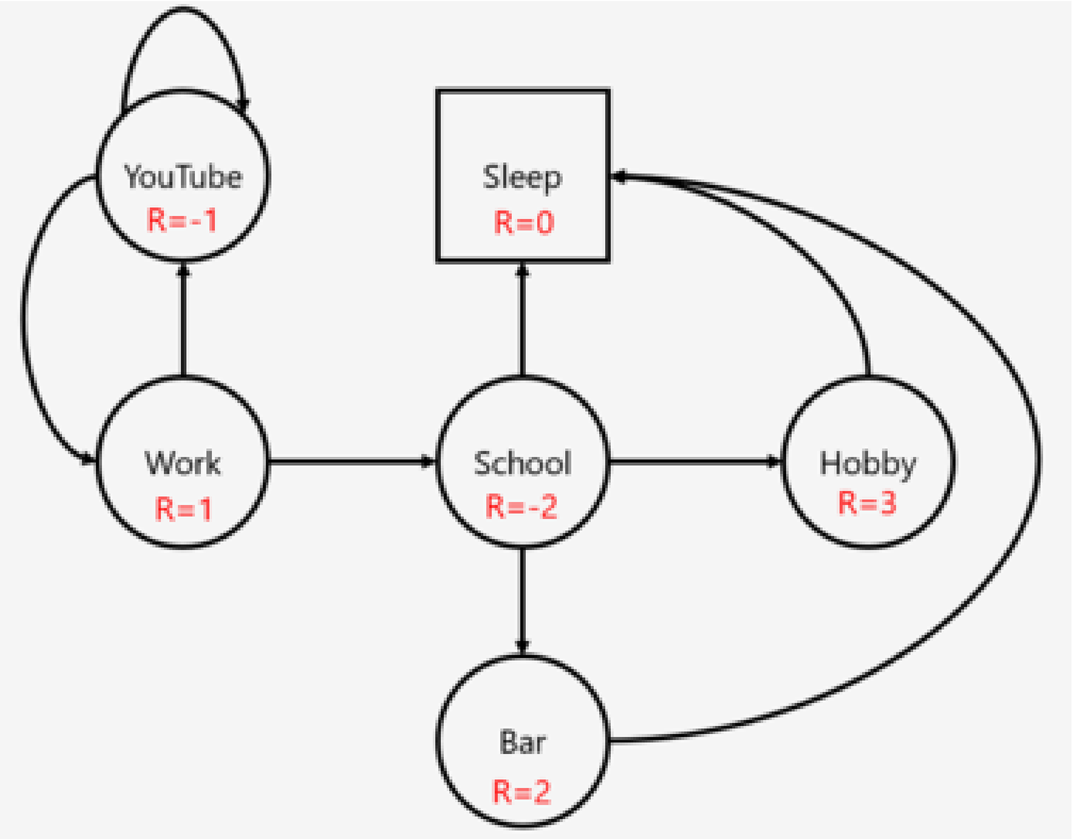


Why Reinforcement Learning?

- 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.
- What happens at a state does NOT depend on previous states.
- The content of a MDP consists of:
  - States.
  - Allowed actions of the states.
  - Immediate rewards of the states.
  - Probabilistic transition function $P(s'|s,a)$.



https://optimization.cbe.cornell.edu/index.php?title=Markov_decision_process

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 $s_0, s_1, s_2, \ldots$ 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

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

State-values

Discount Factor

Policy (action at s)

Bellman's equation (optimal policy):

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

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:

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

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 the MDP; such methods are considered **model-free**.
- Such learning is achieved by **sampling** paths through the state space in order to collect information.

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.

"Update" me with what you see!

I can see better from here!

Temporal Difference (TD) Learning

- A basic TD learning step:

$$V(s) \leftarrow V(s) + \underset{\text{Learning Rate}}{\alpha}\left[\overbrace{r + \underset{\text{Discount Factor}}{\gamma} V(s')}^{\text{Critic}} - \overbrace{V(s)}^{\text{Actor}}\right]$$

- 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.

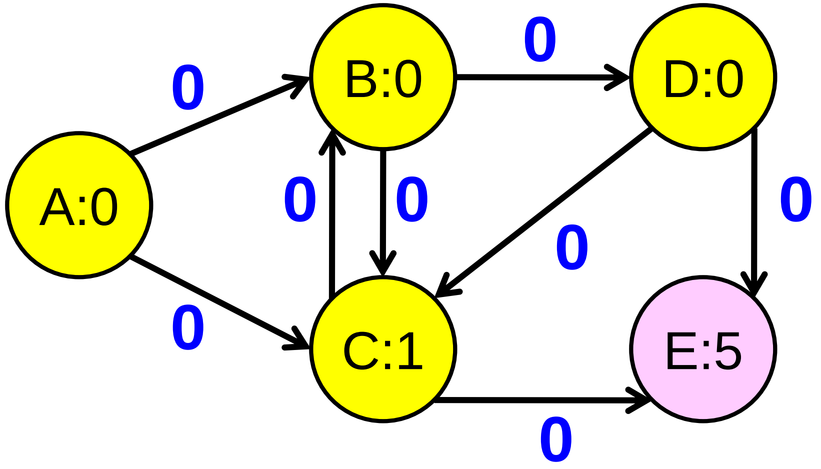
The Q-Learning Algorithm

- 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.)

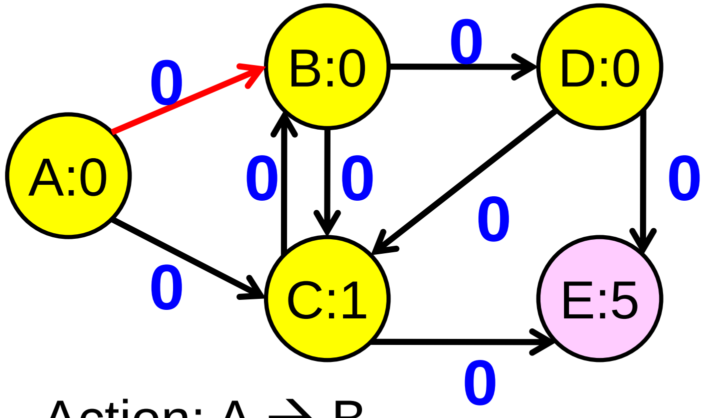
$$Q(s,a) \leftarrow Q(s,a) + \underbrace{\alpha}_{\text{Learning Rate}} \left[r + \underbrace{\gamma}_{\text{Discount Factor}} \max_{a'} Q(s',a') - Q(s,a) \right]$$

Q-Learning Example

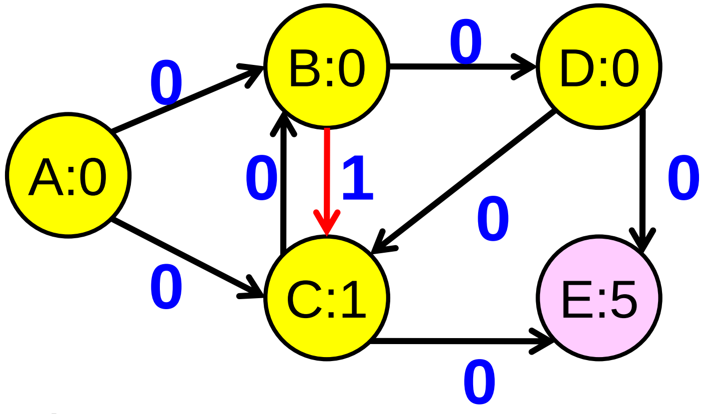
- For simplicity, we will consider only a deterministic environment here, and all the Q values are initialized to zero.
- Settings (state space given below): Start states = {A}. Learning rate = 1. Discount factor = 1. Terminal states = {E}.



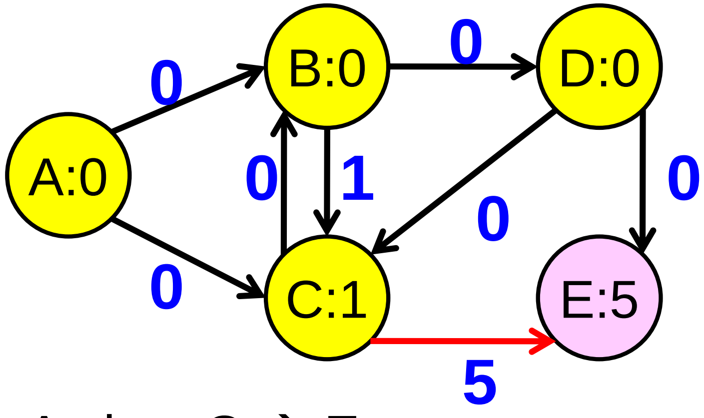
Q-Learning Example



Action: A → B
Q(A, A→B) ← 0

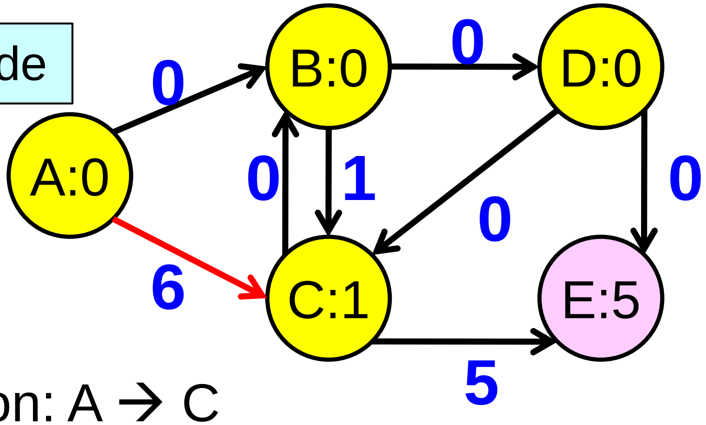


Action: B → C
Q(B, B→C) ← 1 + max(0,0) = 1



Action: C → E
Q(C, C→E) ← 5

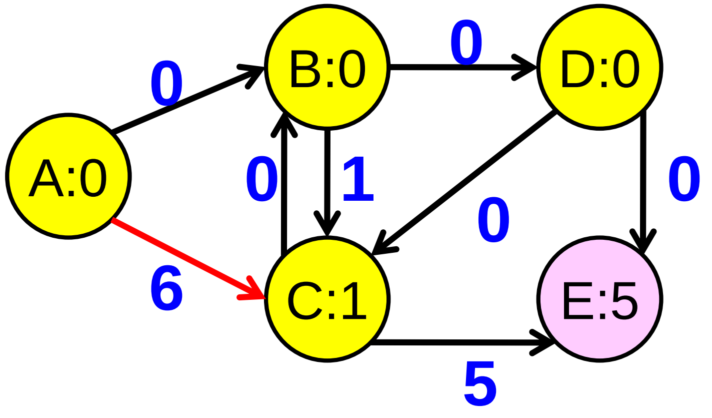
new episode



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 / discretization 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

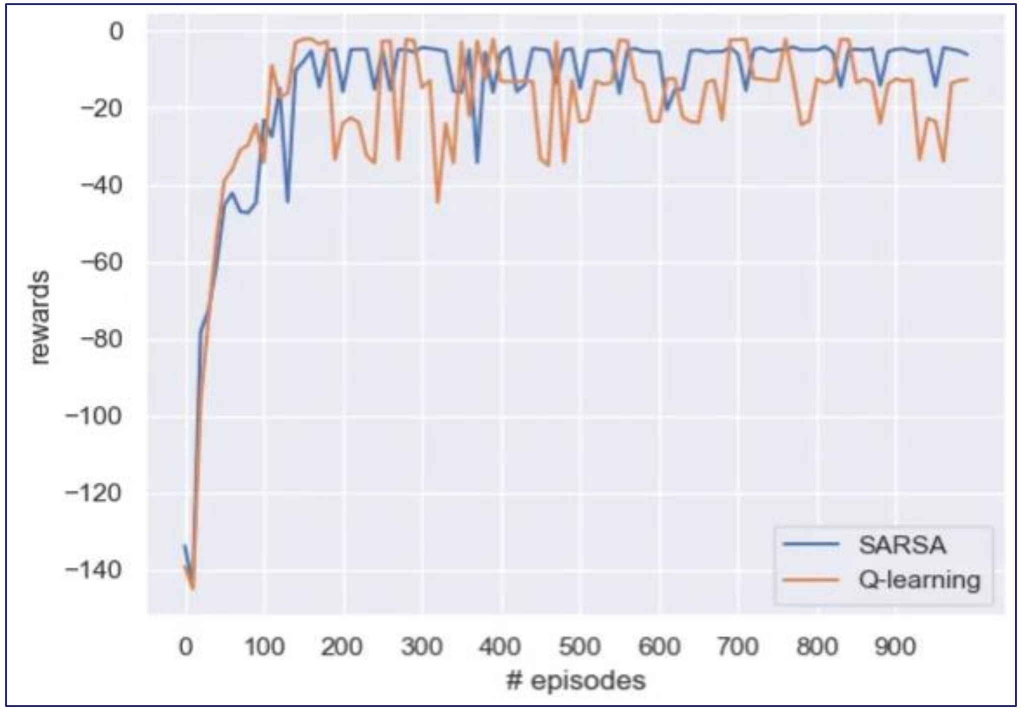
- Problem: During a training episode, how to choose the action from a state?
  - Best Q-value (greedy approach): Exploitation
  - Random action: Exploration
- ε-greedy: Use a probability (ε) to choose between the two. (More exploration initially, and more exploitation later to facilitate convergence.)
- (Optional) Adjustment of the discount factor: smaller initially (to avoid propagation of "noise") and larger later.

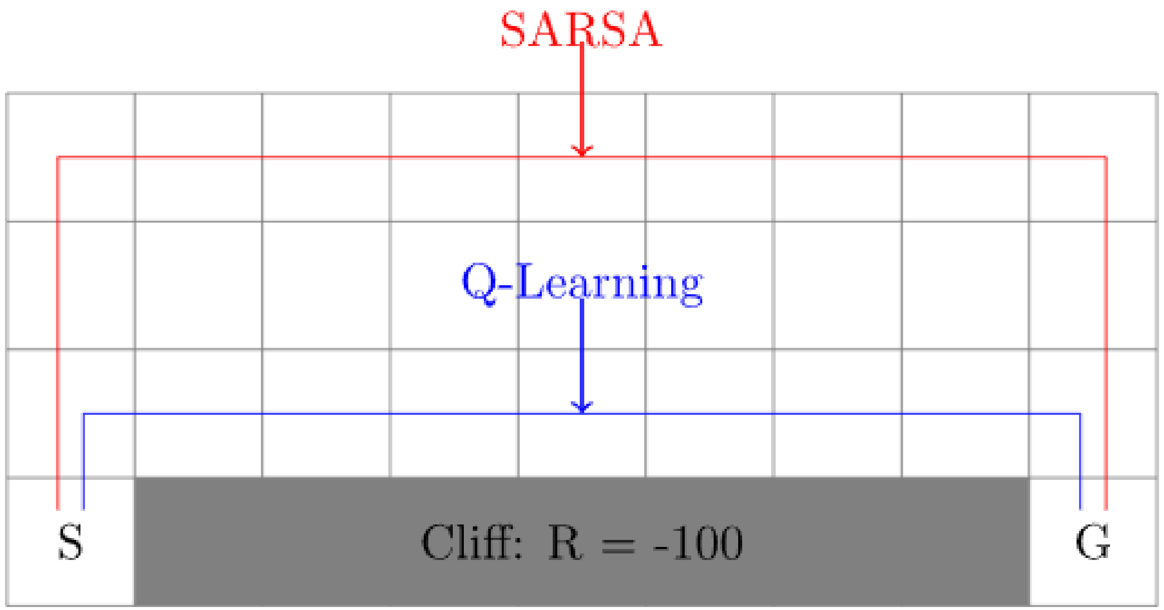
SARSA vs. Q-Learning

- **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):





https://towardsdatascience.com/walking-off-the-cliff-with-off-policy-reinforcement-learning-7fdbcdfe31ff https://www.baeldung.com/cs/q-learning-vs-sarsa

Experience Replay

- The standard practice:
  - Keep a finite-sized buffer of most recent transitions during learning.
  - Periodically sample a batch of recorded transitions from the buffer for learning.
- Advantages:
  - Reusing past transitions is likely more efficient than running new episodes.
  - Reduce the effect of the correlation between consecutive states on the learning process.
  - The use of randomly sampled mini-batches makes the learning more stable.

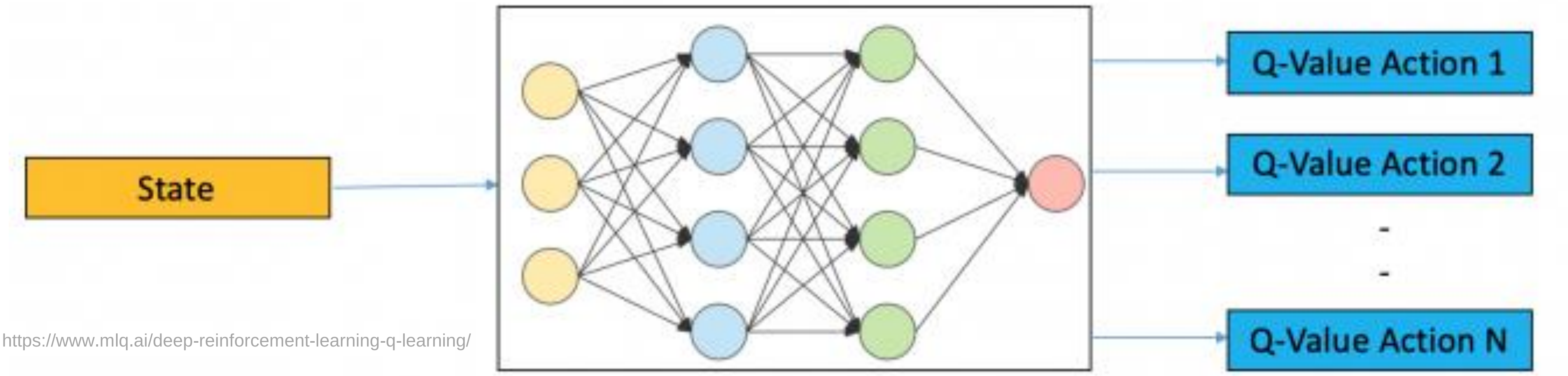
Deep Q-Learning

- 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 (2015/2016).

Deep Q-Learning



- A shared network that estimates the Q values of all the actions given a state.
- An action can be selected based on the maximum Q value.
- To allow for exploration during the learning process, ε-greedy or softmax can be used for the selection of actions.

Policy Parameterization

- Some limitations of Q-learning (and related methods like DQN):
  - Q-table based Q-learning works well for discrete states and discrete actions.
  - DQN (or Q-learning with parameterized Q functions) works with continuous states and discrete actions.
  - Tasks with continuous actions are still challenging. (Discretizing action spaces is heuristic and might not work well.)
- We can work directly with parameterized policies, $\pi_\theta(a|s)$, so that we can handle tasks with continuous actions naturally.
- **Policy gradient**: Update the policy according to how the reward of the sampled actions / trajectories compares to the expected reward of the current policy.
- There are also non-gradient based methods of policy optimization, such as those based on evolutionary computation or simulated annealing.

REINFORCE

- **REINFORCE**, also called **Monte-Carlo Policy Gradient**, is a gradient-ascend method that updates the policy parameters after a path (episode) is sampled.
- Updates are favored for the parameters that enhance the probabilities of actions along paths with larger accumulated rewards.
- Objective function (conceptually): $$U(\theta) = \sum_{\tau} P(\tau;\theta) R(\tau)$$

  summation over possible trajectories
- Gradient ascend: $$\theta \leftarrow \theta + \alpha \nabla_{\theta} U(\theta)$$
- Gradient estimation (from an episode):

$$\nabla_{\theta} U(\theta) = \sum_{t} \left[\nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t) G_t \right]$$

summation over steps in the episode

G_t: cumulative reward from step t till the end

Proximal Policy Optimization

- **Proximal policy optimization (PPO)** is considered the state-of-the-art of policy gradient methods.
- The optimization is based on the **Advantage function**, given as Q–V:
  - Q: State-action value of the action taken (from the sample).
  - V: The state value given by the current policy.
- Surrogate objective function to be maximized:

$$\mathrm{E}\left[r_t(\theta) A_t\right] \quad \text{where} \quad r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta(old)}(a_t \mid s_t)}$$

  - An action (a_t) that leads to positive (negative) advantage (A_t) will have its probability increased (decreased).

Proximal Policy Optimization

- It is well known that, for the training to be stable, the policy update should be gradual (i.e., the policies before and after an update should be somewhat similar).

- The solution proposed for PPO: To clip the action probability ratio. The adjusted surrogate objective function is

$$\mathrm{E}\left[\min\left(r_t(\theta)A_t,\ \ \mathrm{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t\right)\right]$$

  - Here ε is commonly set to 0.2.

Proximal Policy Optimization

PPO implementation, as in the original paper:

- Neural networks are used for (1) representing policies and (2) estimating state values. Use a shared network for both purposes.
- Objective function:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right]$$

policy gradient value function extra exploration

- Fixed-length trajectory segments (instead of episodes) used in learning.
- Estimate the advantage function (of the whole trajectory) using an actor-critic approach:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$
$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Proximal Policy Optimization

Pseudo-code, from the original paper:

Algorithm 1 PPO, Actor-Critic Style

for iteration$=1,2,\ldots$ **do**
  for actor$=1,2,\ldots,N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1,\ldots,\hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for