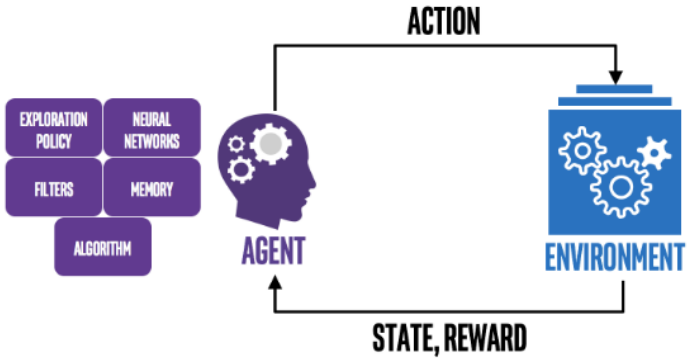
Reinforcement Learning

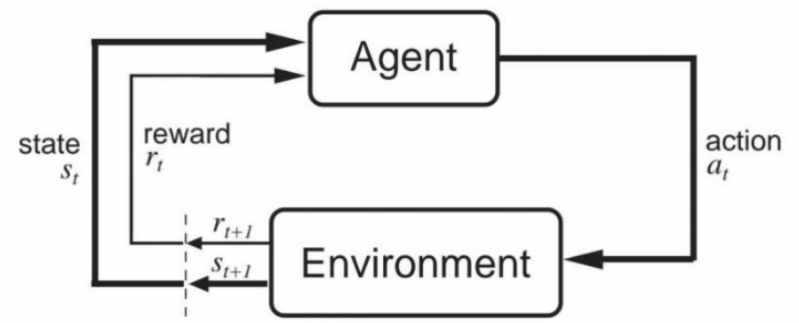
2021. 10

Yongjin Jeong, KwangWoon University

Reinforcement Learning

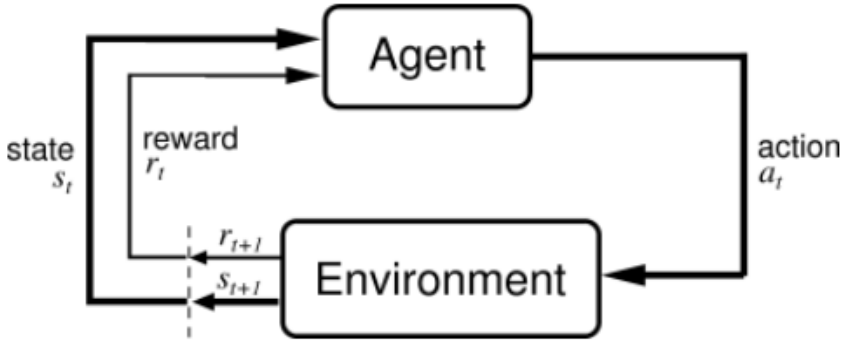
- **Reinforcement Leaning (RL)**
  - Concerned with sequential decision making process
  - Features an agent learning how to act or decide in an environment





- **Agent** – The program to train (Neural network)
- **Environment** – The world (real or virtual), in which the agent performs actions
- **State** – How the agent sees the environment
- **Action** – A move made by the agent, which causes a status change in the environment
- **Rewards** – The evaluation of an action, which can be positive or negative

Terminologies



Agent-environment interaction [Source]

- **Agent:** It is an assumed entity which performs actions in an environment to gain some reward (robots or human or program)
- **Environment (e):** A scenario that an agent has to face.
- **Reward (R):** An immediate return given to an agent when he or she performs specific action or task.
- **Action (A):** a set of actions which the agent can perform
- **State (s):** State refers to the current situation returned by the environment.
- **Policy (π):** It is a strategy which applies by the agent to decide the next action based on the current state (normally probability).

Terminologies

- **Value (V):** It is expected long-term return with discount
- **Value Function:** It specifies the value of a state that is the total amount of reward, which should be expected beginning from that state.
- **Q value or action value (Q):** Q value is quite similar to value. The only difference between the two is that it takes an additional parameter as a current action.
- **Model of the environment:** This mimics the behavior of the environment. By a model of the environment, we mean a function which predicts **state transitions and rewards**.
- **Model-Free vs. Model-Based RL**
  - **Model-based:** after learning, the agent can make predictions about what the next state and reward will be before it takes each action. **Dynamic Programming (policy iteration and value iteration)** use the model's predictions or distributions of next state and reward in order to calculate optimal actions.
  - **Model-free: no knowledge of MDP transitions and rewards - a ground-truth model of the environment is usually not available to the agent.** If an agent wants to use a model, it has to learn the model **purely from experience**. **Monte Carlo, Temporal Difference (TD), SARSA, Q-learning, Actor-Critic** rely on real samples from the environment.

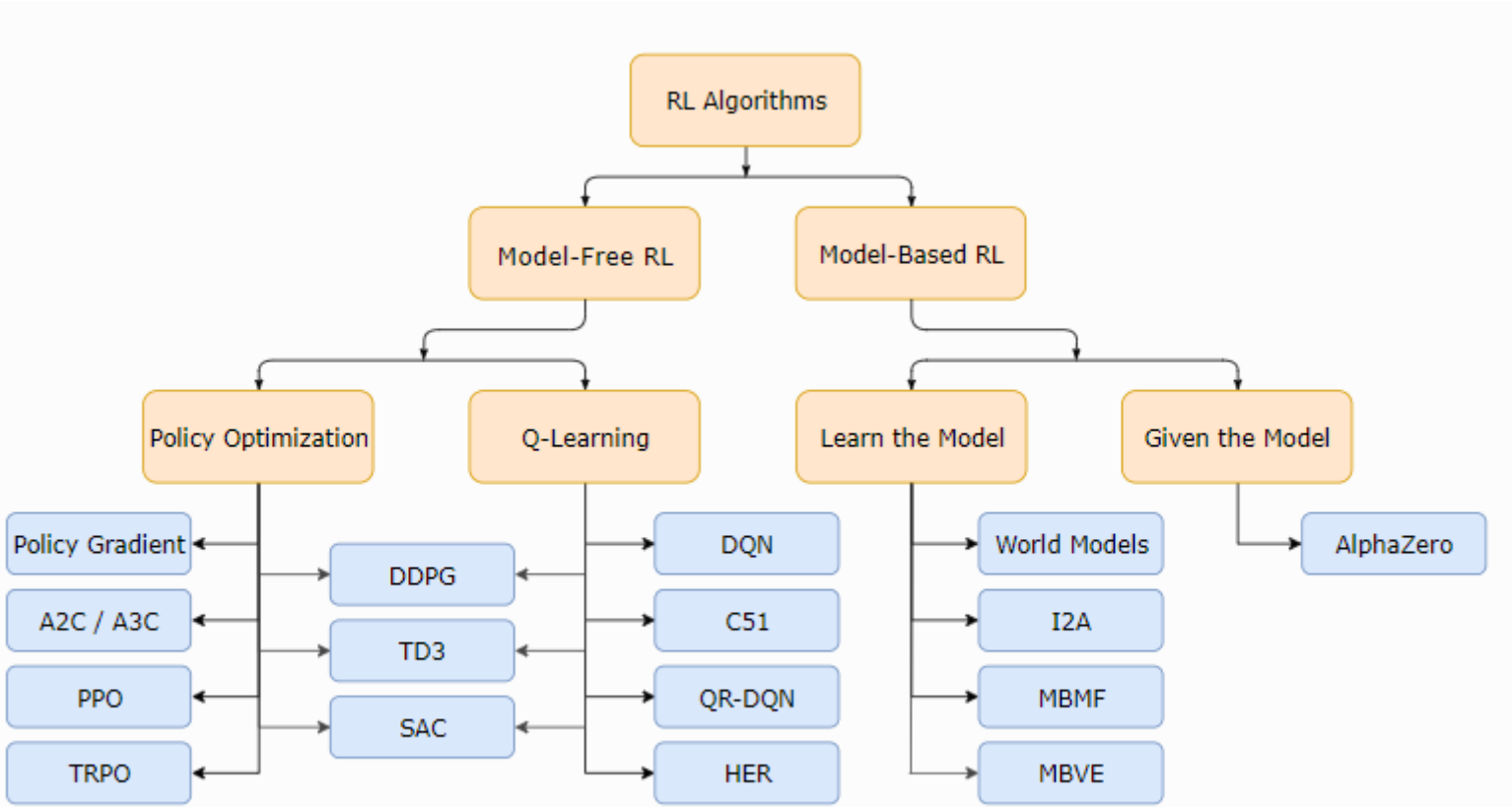
RL Methods

- **Dynamic programming (DP) —** good starting point to understand RL algorithms that can solve more complex problems
- **Monte Carlo methods** — A class of methods for learning of value functions, which estimates the value of a state by running many trials starting at that state, then averages the total rewards received on those trials.
- **Temporal Difference (TD) algorithms** — A class of learning methods, based on the idea of comparing temporally successive predictions. Possibly the single most fundamental idea in all of reinforcement learning.

- **What to Learn?**
  - **policies**, either stochastic or deterministic,
  - **action-value functions** (Q-functions),
  - **value functions**, (V)
  - and/or environment models.

Taxonomy

- **OpenAI** — a non-profit AI research company with the mission to build and share safe Artificial General Intelligence (AGI) — launched a program to "spin up" deep RL. (https://spinningup.openai.com/en/latest/index.html)
- **Taxonomy of RL algorithms**



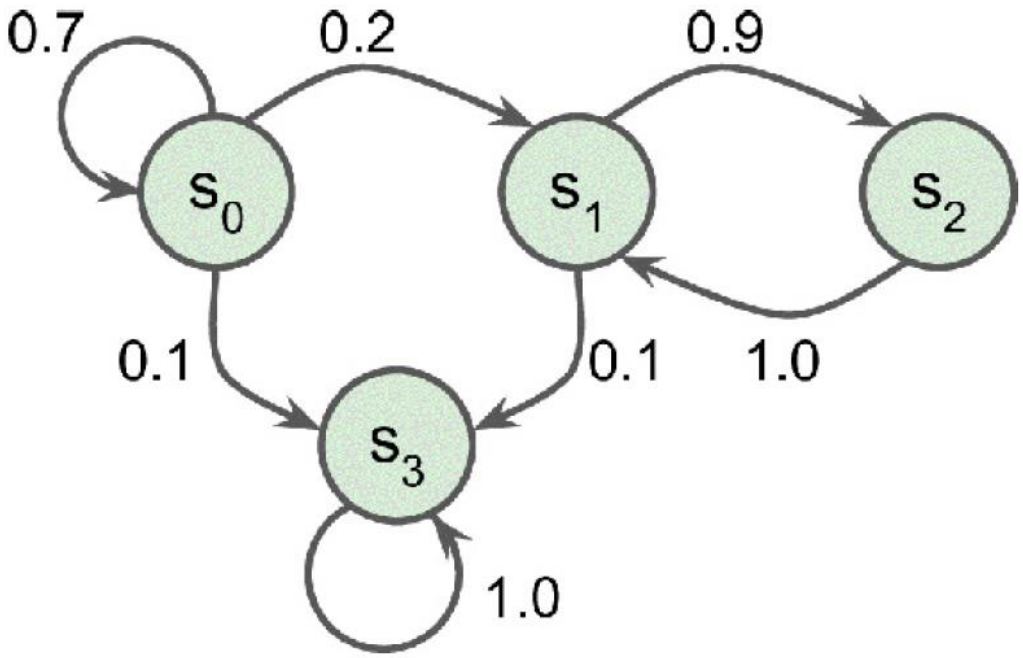
A non-exhaustive, but useful taxonomy of algorithms in modern RL. Citations below.

Real Life Applications

- **Self-driving cars:** trajectory optimization, motion planning, dynamic pathing, controller optimization, and scenario-based learning policies for highways.
- **Industry automation:** energy saving (Google), industrial robots
- **Trading and finance:** better stock exchange than time series model (supervised learning)
- **NLP:** text summarization, question answering, machine translation
- **healthcare:** medical treatment options for every disease stage
- **engineering:** optimize video streaming quality based on the state of video buffer, etc.
- **News recommendation:** track reader's return behavior
- **Gaming:** AlphaGo Zero
- **Marketing and advertisement:** real-time bidding with multi-agent RL
- **Robotics manipulation:** Boston Dynamics robots
- See https://neptune.ai/blog/reinforcement-learning-applications

Markov Chain

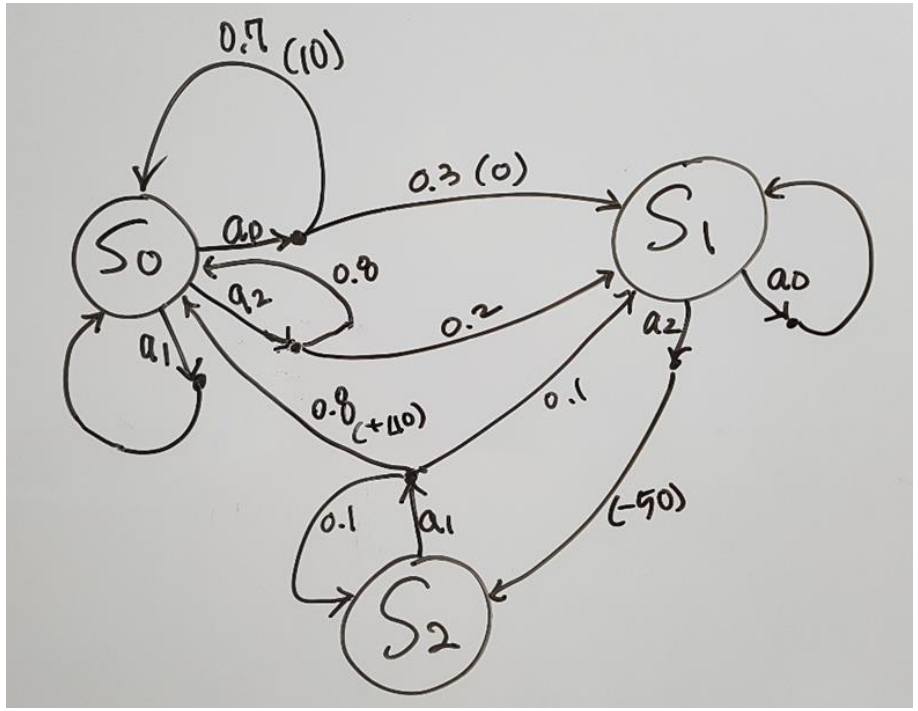
- Stochastic processes with no memory
- Has a fixed number of states
- Probability from a state s to s’ is fixed, and it depends only on the pair (s,s’), not on the past state (the system has no memory)




```
transition_probabilities = [
        [0.7, 0.2, 0.0, 0.1],
        [0.0, 0.0, 0.9, 0.1],
        [0.0, 1.0, 0.0, 0.0],
        [0.0, 0.0, 0.0, 1.0]]
```

Markov Decision Processes (MDP)

- They resemble Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and
- The state transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and
- The agent's goal is to find a policy that will maximize rewards over time.




```
transition_probabilities = [
        [[0.7, 0.3, 0.0],
         [1.0, 0.0, 0.0],
         [0.8, 0.2, 0.0]],

        [[0.0, 1.0, 0.0],
         None,
         [0.0, 0.0, 1.0]],

        [None,
         [0.8, 0.1, 0.1],
         None]]
```

```
rewards = [ # shape=[s, a, s']
        [[+10, 0, 0],
         [0, 0, 0],
         [0, 0, 0]],

        [[0, 0, 0],
         [0, 0, 0],
         [0, 0, -50]],

        [[0, 0, 0],
         [+40, 0, 0],
         [0, 0, 0]]]
```

```
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Policy and Value Functions

- **Policy: probability**

$$\text{policy} = \pi(a|s) \quad = P(A_t=a \mid S_t=s)$$

- **State-Value function**

"return": t 시점 이후의 모든 discount 된 reward 의 총합

$$G_t = R_{t+1} + \gamma R_{t+2} + r^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \middle| S_t = s \right] \end{aligned}$$

Value(가치) of state s

- **Action-Value function (Q-Function)**

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \middle| S_t = s, A_t = a \right] \end{aligned}$$

Value (가치) of state-action (s,a)

Bellman Equation

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[G_t \middle| S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \middle| S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \middle| S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s',r} \mathbb{P}(s', r|s, a)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \middle| S_{t+1} = s\right.\right. \\
&= \sum_a \pi(a|s) \sum_{s',r} \mathbb{P}(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \\
&= \sum_a \pi(a|s) q_\pi(s, a)
\end{aligned}
$$

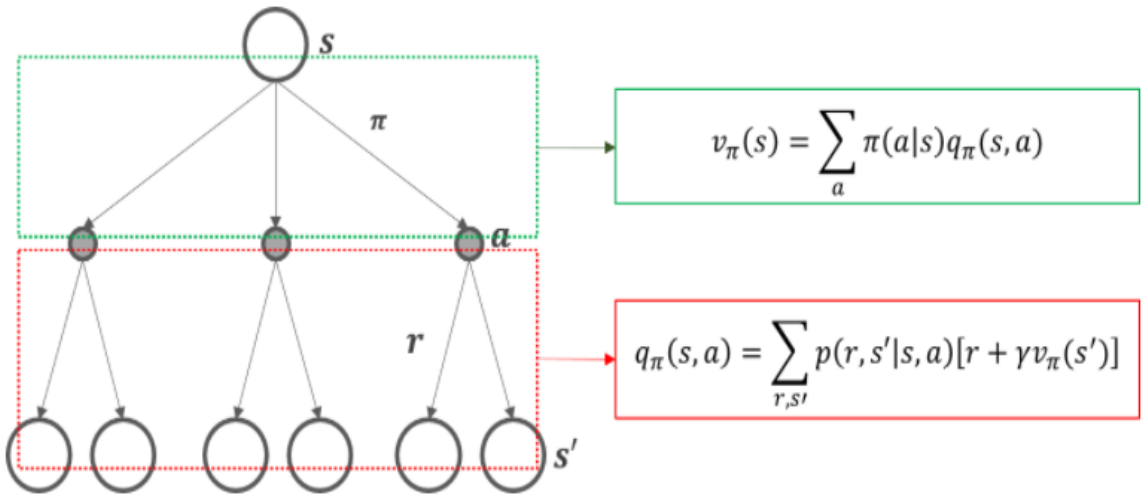


Figure 2: Bellman Equation Back-up Diagram

$$
\begin{aligned}
q_\pi(s, a) =& \mathbb{E}_\pi\left[G_t \middle| S_t = s, A_t = a\right] \\
=& \mathbb{E}_\pi\left[R_{t+1} + \gamma \mathbb{E}_\pi(G_{t+1} | S_{t+1} = s') \middle| S_t = s, A_t = a\right] \\
& \text{or } \mathbb{E}_\pi\left[R_{t+1} + \gamma \mathbb{E}(G_{t+1} | S_{t+1} = s', A_{t+1} = a') \middle| S_t = s, A_t = a\right] \\
=& \mathbb{E}\left[R_{t+1} + \gamma v_\pi(s') \middle| S_t = s, A_t = a\right] \\
& \text{or } \mathbb{E}\left[R_{t+1} + \gamma q_\pi(s', a') \middle| S_t = s, A_t = a\right] \\
=& \sum_{r,s'} \mathbb{P}(s', r|s, a)[r + \gamma v_\pi(s')] \\
& \text{or } \sum_{r,s',a'} \mathbb{P}(r, s', a'|s, a)[r + \gamma q_\pi(s', a')]
\end{aligned}
$$

Bellman Optimality Equation

- **$V^*(s)$: Expected Sum of all discounted future rewards** the agent can expect after it reaches a state *s*, assuming it acts optimally.

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V^*(s')] \quad \text{for all } s$$

  - $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a.
  - $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a.
  - γ is the discount rate.

- An algorithm to precisely estimate the optimal state value of every possible state:
  - Initialize all the state value estimates = 0
  - Iteratively update them using **Value Iteration algorithm**

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma . V_k(s')] \quad \text{for all } s$$

Dynamic programming

  - $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

Q-Value Iteration

- Knowing the optimal state values can be useful (to evaluate a policy), but it does not tell the agent explicitly what to do.
- A very similar algorithm to estimate the optimal state-action values (Q-values) of the state-action pair (*s,a*), noted Q*(*s,a*):

$$Q_*(s,a) = \max_{\pi} Q_{\pi}(s,a)$$

- **Precise estimation:**
  - Initialize all the Q-value estimates = 0
  - Iteratively update them using **Q-value Iteration algorithm**

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma . \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s,a)$$

- Once you have the optimal Q-values, the optimal policy π*(s) is:

$$\pi^*(s) = \operatorname{argmax}_{a} Q^*(s,a)$$

Temporal Difference (TD) Learning

- **Model-free**,
- RL problems with discrete actions can be modeled as MDP, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$).
- It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.
- Monte-Carlo vs. TD
  - Monte-Carlo: $V(S_t)$ can only be computed once a terminal state is reached

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

  - Time Difference TD(0): $V(S_t)$ is computed using only one step look ahead

$$V(S_t) \leftarrow V(S_t) + \alpha(\mathbf{R_{t+1}} + \gamma \mathbf{V(S_{t+1})} - V(S_t))$$

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}\big[\text{Target} - \text{OldEstimate}\big]$$

Temporal Difference Learning

- **Temporal Difference Learning (TD)**
  - Similar to Value Iteration algorithm, but, take into account the fact that the agent has only partial knowledge of the MDP. (In general we assume that the agent initially knows only the possible states and actions, and nothing more.)
  - The agent uses an exploration policy (for example, random policy) to explore the MDP, and progresses TD algorithm to update the estimates based on the transitions and rewards that are actually observed.

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma . V_k(s'))$$

- α is the learning rate (e.g., 0.01).

simply keep track of a running average of the rewards (immediate + future) at each state s

  - Many similarities with Stochastic Gradient Descent (SGD):
    - handles one sample at a time
    - converge if you gradually reduce the learning rate

Q-Learning algorithm

- **Q-Learning**
  - Adaptation of the Q-value iteration with no information about the transition probabilities and the rewards

$$Q_{k+1}(s, a) \leftarrow (1-\alpha)Q_k(s, a) + \alpha\left(r + \gamma.\max_{a'} Q_k(s', a')\right)$$

  - For each state-action pair (*s*, *a*), this algorithm keeps track of a running average of the rewards (immediate + future)
  - Since the target policy would act optimally, we take the maximum of the Q-Value estimates for the next state.

SARSA vs. Q-Learning

- **Q-Learning (off-policy)**

Considers the best possible case

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

```
Q[s,a] = Q[s,a] + alpha*(reward - Q[s,a]
          + gamma*np.max([Q[s_,a_] for a_ in range(possible_action)]))
```

- **SARSA (on-policy)**

follow the current policy

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

```
a_ = max_action(s_) if rand < (1-epsilon) else env.action_space.sample()
Q[s,a] = Q[s,a] + alpha*(reward - Q[s,a] + gamma*Q[s_,a_])
```

ε-greedy policy

SARSA vs. Q-Learning

- Policy:
  - e-greedy is applied to balance the exploration and exploration of reinforcement learning.
- Both are updated using the TD learning formula.
- Difference:
  - **Q-Learning (off-policy):** when passing the reward from the next state (s_, a_) to the current state, it takes the maximum possible reward of the new state (s_) and ignores whatever policy we are using.
  - **SARSA (on-policy):** we still follow the policy (e-greedy), compute the next state (a_), and pass the reward corresponding to that exact a_ back the previous step.
  - Hence, if our policy is greedy, SARSA and QL will be the same. But we are using e-greedy normally, so there is a slight difference.

SARSA vs. Q-Learning

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

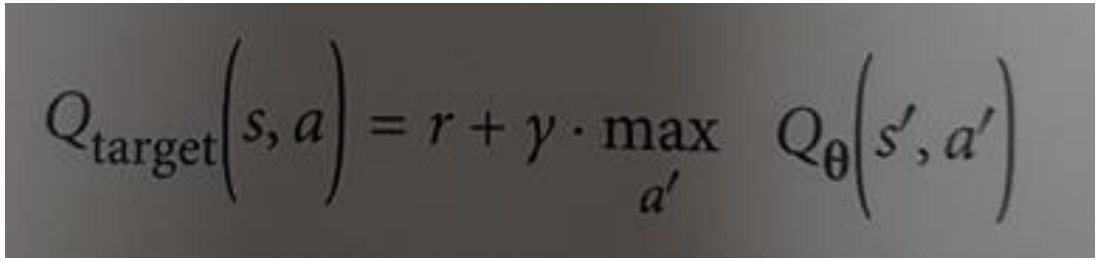
SARSA vs. Q-Learning

- **Some notes:**
  - Both approach work in a finite environment (or a discretized continuous environment)
  - QL directly learns the optimal policy while SARSA learns a "near" optimal policy. QL is a more aggressive agent, while SARSA is more conservative.
  - An example is walking near the cliff. QL will take the shortest path because it is optimal (with the risk of falling), while SARSA will take the longer, safer route (to avoid unexpected falling).
  - In practice, if you want to fast in a fast-iterating environment, QL should be your choice. However, if mistakes are costly (unexpected minimal failure — robots), then SARSA is the better option.
  - If your state space is too large, try exploring the deep q network.

Approximate Q-Learning (DQN)

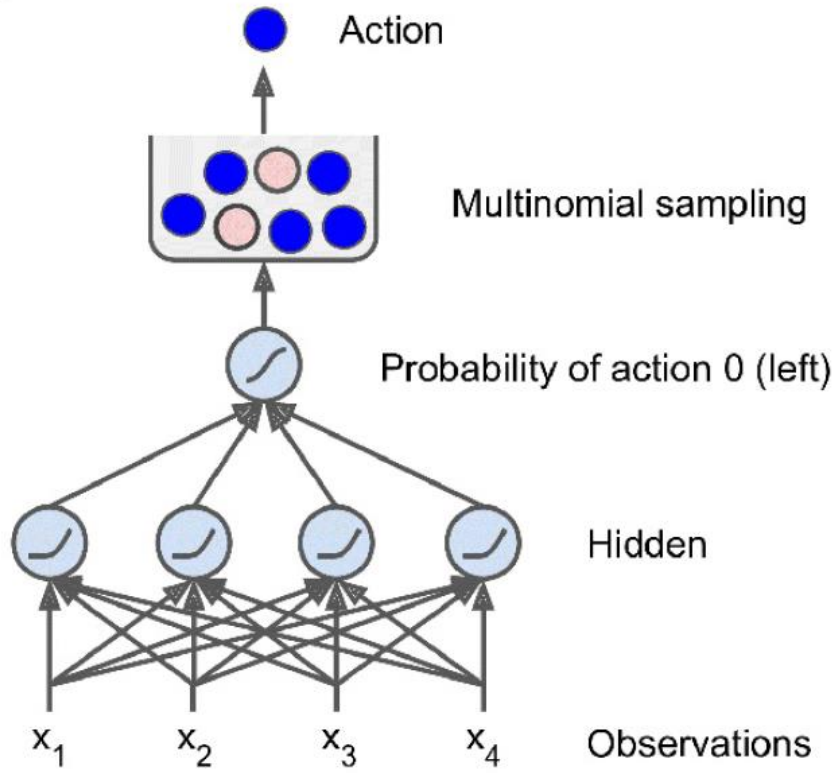
- **Problem in Q-Learning**
  - It does not scale well to large MDPs with many states and actions.
  - The solution is to find a function that approximates the Q-values using some parameters. (Approximate Q-Learning)
  - Uses DQN(deep Q-network) to estimate Q-Values (Deep Q-Learning)

- **How to train DQN?**
  - Thanks to Bellman, we want the approximate Q-value to be as close as to the reward **r** that we actually observe after playing action a in state s, plus the discounted value of playing optimally from then on.
  - Target Q-value:



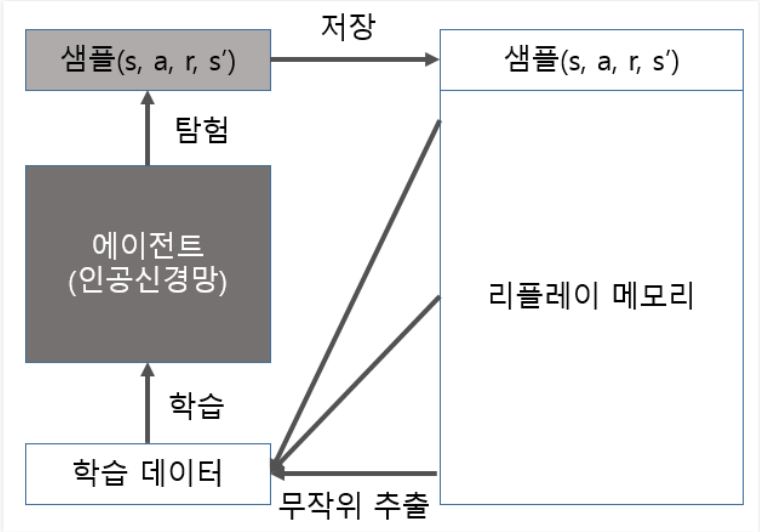
Approximate Q-Learning (DQN)

- Neural network: estimate a probability for each action
  - Input: an observation
  - Output: action to be taken



DQN

- **Experience (replay) Memory**
  - Stores experiences including state transitions, rewards and actions (that are necessary to perform Q learning)
  - Makes mini-batches to update neural networks.
  - Merits:
    - Reduces correlation between experiences in updating DNN
    - Increases learning speed with mini-batches
    - Reuses past transitions to avoid catastrophic forgetting
- **Double DQN**
  - Use two separate Q-networks (**Action Network** and **Target Network**)
  - Periodically update (by copying Action Network to Target Network)
- **Gradient Clipping**
  - forces the gradient values (element-wise) to a specific minimum or maximum value (+1 and -1)



Deep Q-learning

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode = 1, M **do**

Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1$,T **do**

With probability ε select a random action a_t

otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

Execute action a_t in emulator and observe reward r_t and image x_{t+1}

Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ

Every C steps reset $\hat{Q} = Q$

End For

End For

preprocess example: screen display interface for games

ε-greedy

Example code:

- Note that we do not clear out the memory after each episode, this enables us to recall and build batches of experiences from across episodes.
- https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c

Initialize network Q
Initialize target network $\hat{Q}$
Initialize experience replay memory D
Initialize the *Agent* to interact with the Environment
while *not converged* **do**

```
/* Sample phase
```

$\epsilon \leftarrow$ setting new epsilon with ϵ-decay
Choose an action a from state s using policy ϵ-greedy(Q)
Agent takes action a, observe reward r, and next state s'
Store transition $(s, a, r, s', done)$ in the experience replay memory D

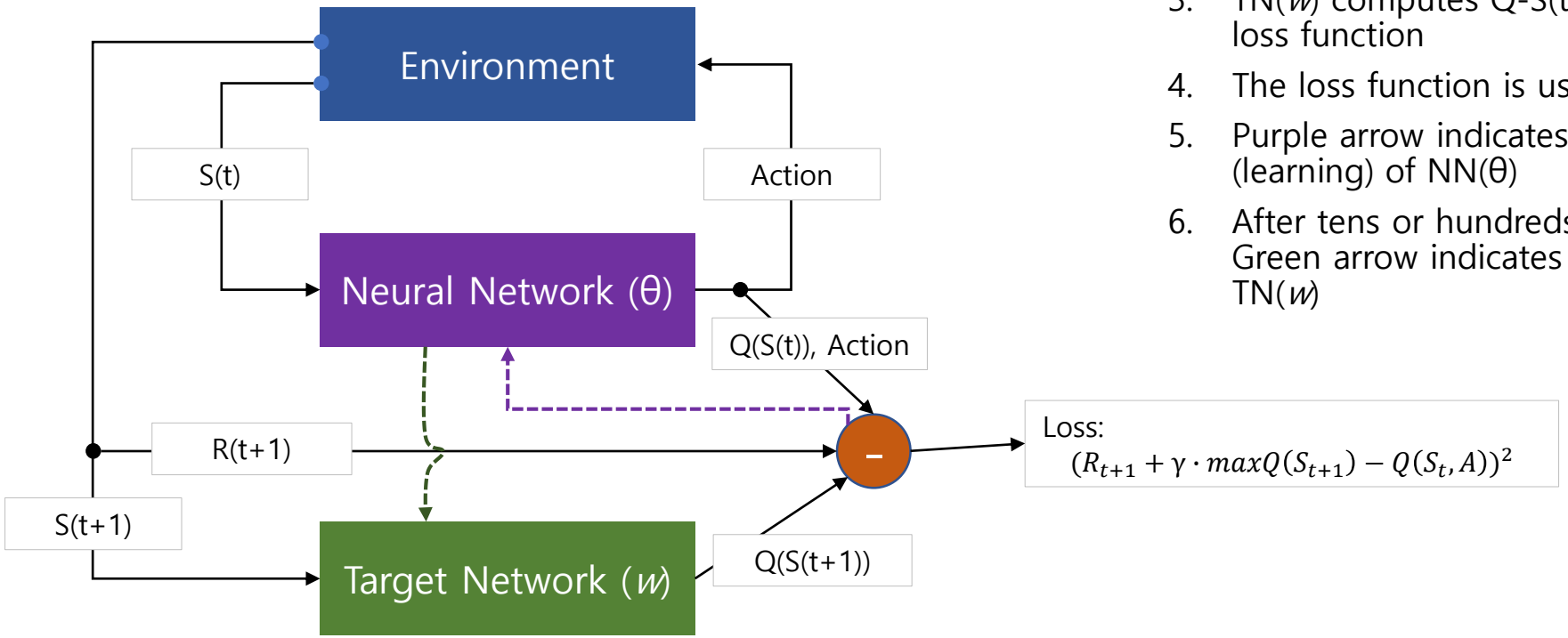
if *enough experiences in* D **then**

```
/* Learn phase
```

Sample a random *minibatch* of N transitions from D
for *every transition* $(s_i, a_i, r_i, s'_i, done_i)$ ***in*** *minibatch* **do**
if $done_i$ **then**
$y_i = r_i$
else
$y_i = r_i + \gamma \max_{a' \in \mathcal{A}} \hat{Q}(s'_i, a')$
end
end
Calculate the loss $\mathcal{L} = 1/\mathrm{N} \sum_{i=0}^{N-1} (Q(s_i, a_i) - y_i)^2$
Update Q using the SGD algorithm by minimizing the loss $\mathcal{L}$
Every C steps, copy weights from Q to $\hat{Q}$
end
end

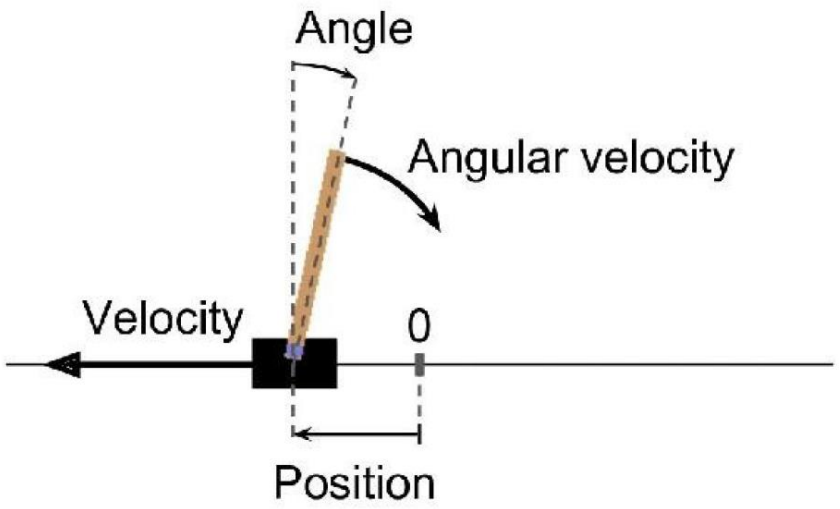
Deep Q-learning

- **General flow of Deep Q-Leaning**
  - Deep Q-Network (DQN) Perspective



1. Input S(t) to NN(θ) then outputs an action
2. From the Action, Environment gives S(t+1) and R(t+1)
3. TN(*w*) computes Q-S(t+1), then computes the loss function
4. The loss function is used to train NN(θ)
5. Purple arrow indicates the parameter update (learning) of NN(θ)
6. After tens or hundreds of NN(θ) update, Green arrow indicates parameters copy to TN(*w*)

CartPole example



Policy Gradients

- Parameterize policy and update those parameters directly
- Enables new kinds of policies: stochastic, continuous action s paces

$$Q_\theta(s,a) \qquad \pi(a|s) \rightarrow \pi_\theta(a|s)$$

- On policy learning → learn directly from your actions

$$\hat{\theta} = \arg\max_\theta J(\theta)$$

$$J(\theta) = \mathbb{E}_{a_t \sim \pi(\cdot|s_t;\theta), s_0 \sim \mathcal{T}_0}\left[G(s_0)\right]$$

Policy Gradients

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{a_t \sim \pi(\cdot|s_t;\theta)} \left[G(s_t, a_t) \right]$$

$$= \nabla_\theta \int \pi(a|s_t;\theta) G(s_t, a) \mathrm{d}a = \int \mathrm{d}a \; G(s_t, a) \nabla_\theta \pi(a|s_t;\theta)$$

$$= \int \mathrm{d}a \; G(s_t, a) \frac{\pi(a|s_t;\theta)}{\pi(a|s_t;\theta)} \nabla_\theta \pi(a|s_t;\theta) = \int \mathrm{d}a \; \pi(a|s_t;\theta) G(s_t, a) \frac{\nabla_\theta \pi(a|s_t;\theta)}{\pi(a|s_t;\theta)}$$

$$= \int \mathrm{d}a \; \pi(a|s_t;\theta) G(s_t, a) \nabla_\theta \ln \pi(a|s_t;\theta)$$

$$= \mathbb{E}_{a_t \sim \pi(\cdot|s_t;\theta)} \left[G(s_t, a_t) \nabla_\theta \ln \pi(a_t|s_t;\theta) \right]$$

→ Approximate expectation value from samples

REINFORCE

function REINFORCE
Initialise θ arbitrarily
for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
for $t = 1$ to $T - 1$ **do**
$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
end for
end for
return θ
end function

Sutton et al. (2000)

ACTOR CRITIC