

지능 시스템

Intelligent Systems

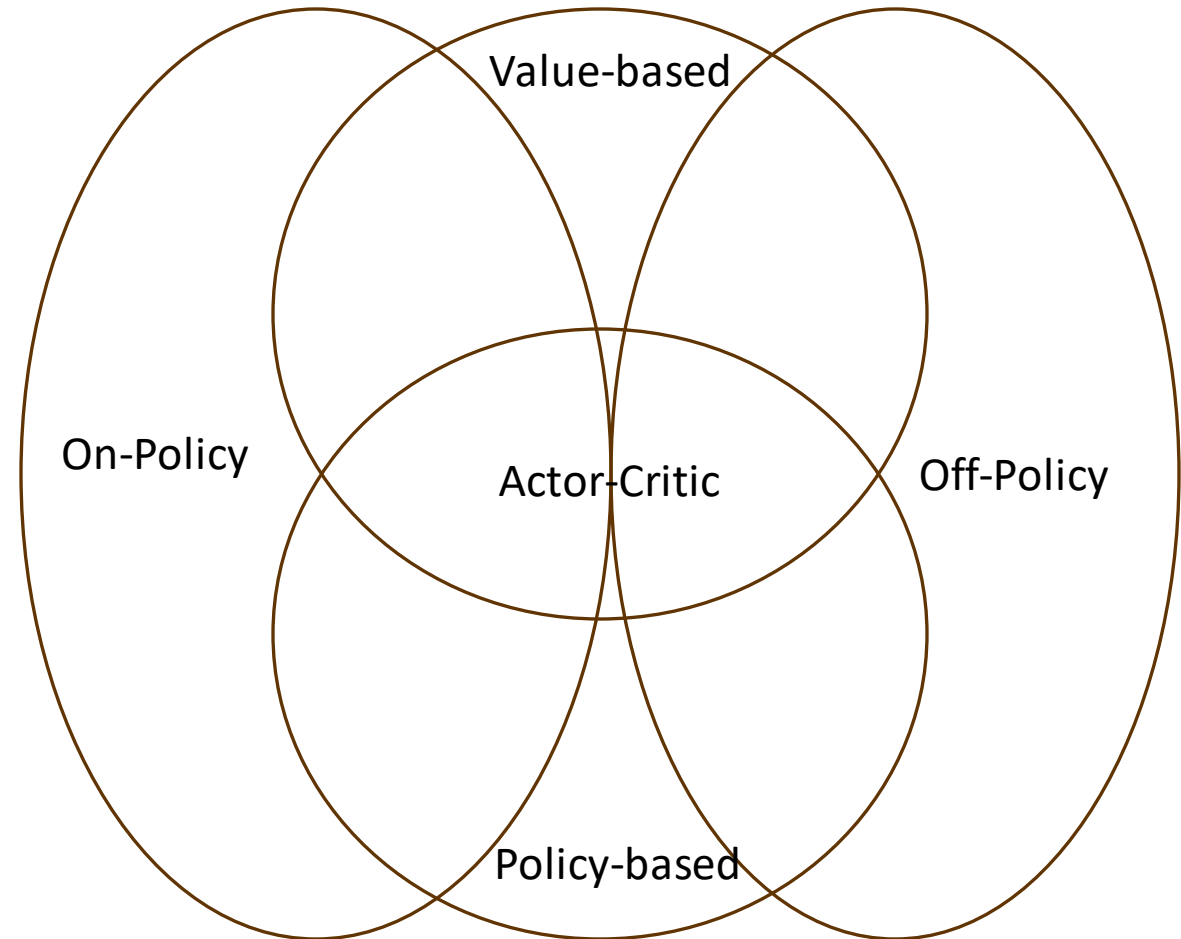
Lecture 3 – Monte-Carlo and Temporal Difference Prediction and Evaluation

Today's Contents

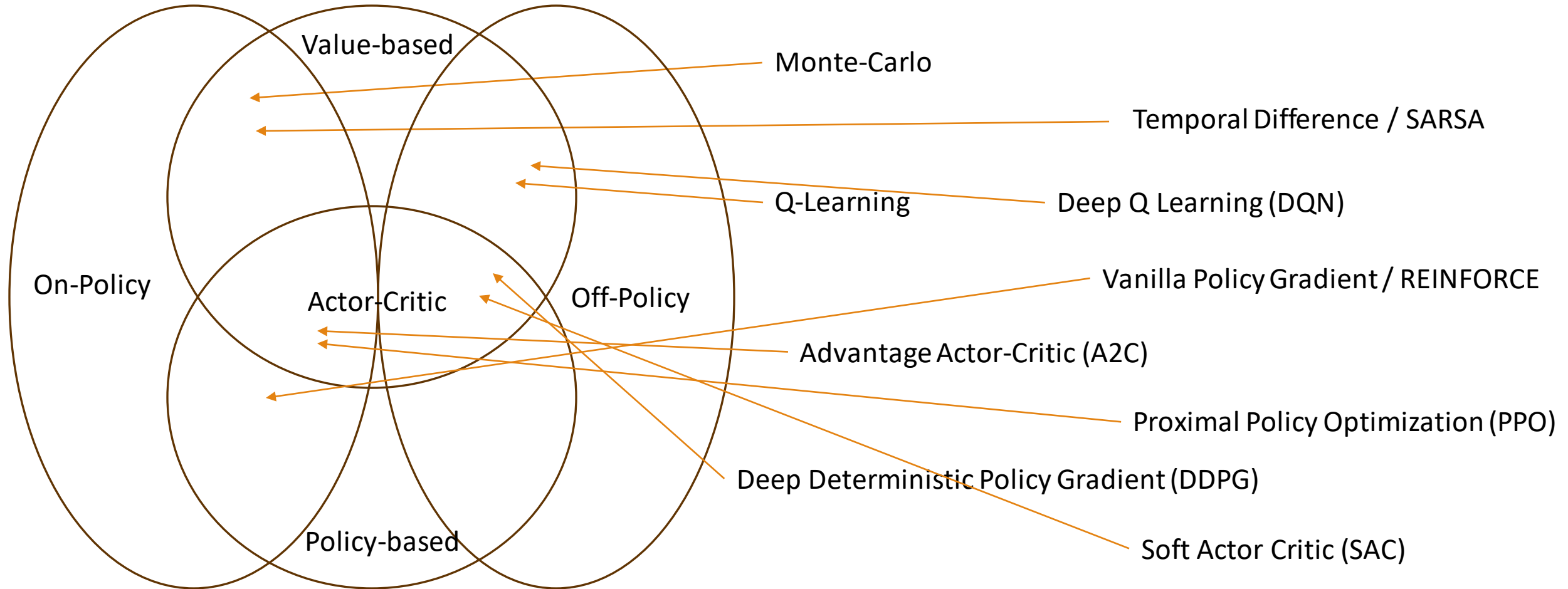
- Revision of Value / Action-Value functions
- Revision of Bellman Equation
- Concept of Learning V and Q
- Learning Methodologies
- Monte-Carlo Algorithm
- Temporal-Difference Algorithm

Reinforcement Learning Methodologies

Model-free
Does not require the
transition probability



Some Popular Algorithms



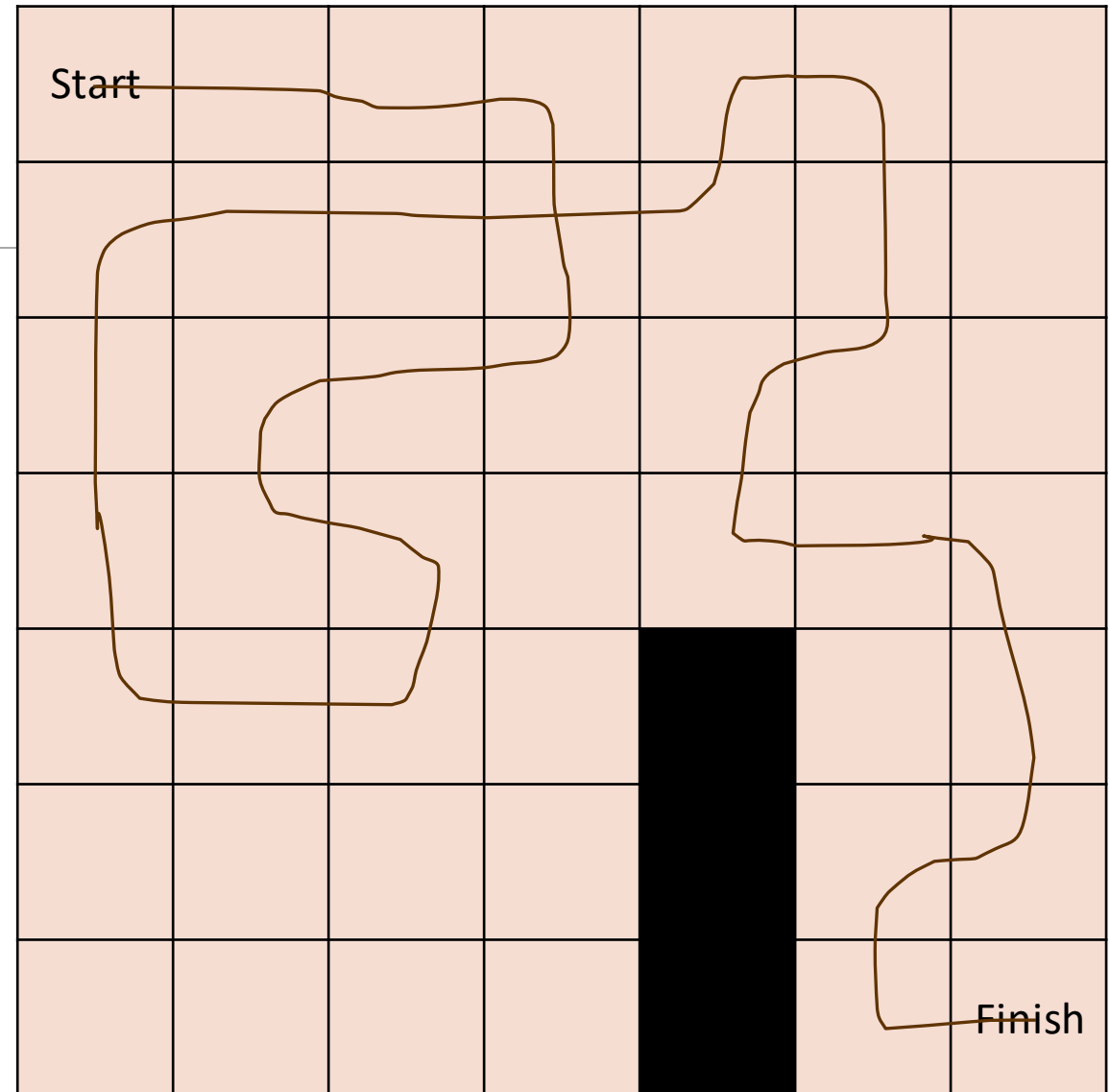
Value-based Method

- Finding Optimal Values / Action-value is the key in RL!
- Knowing which state is good, the policy can simply be set to gradually lead the agent to the good state.
- This is called **Value-based method**.

Maze Example

Example Algorithm for a Value-based Method:

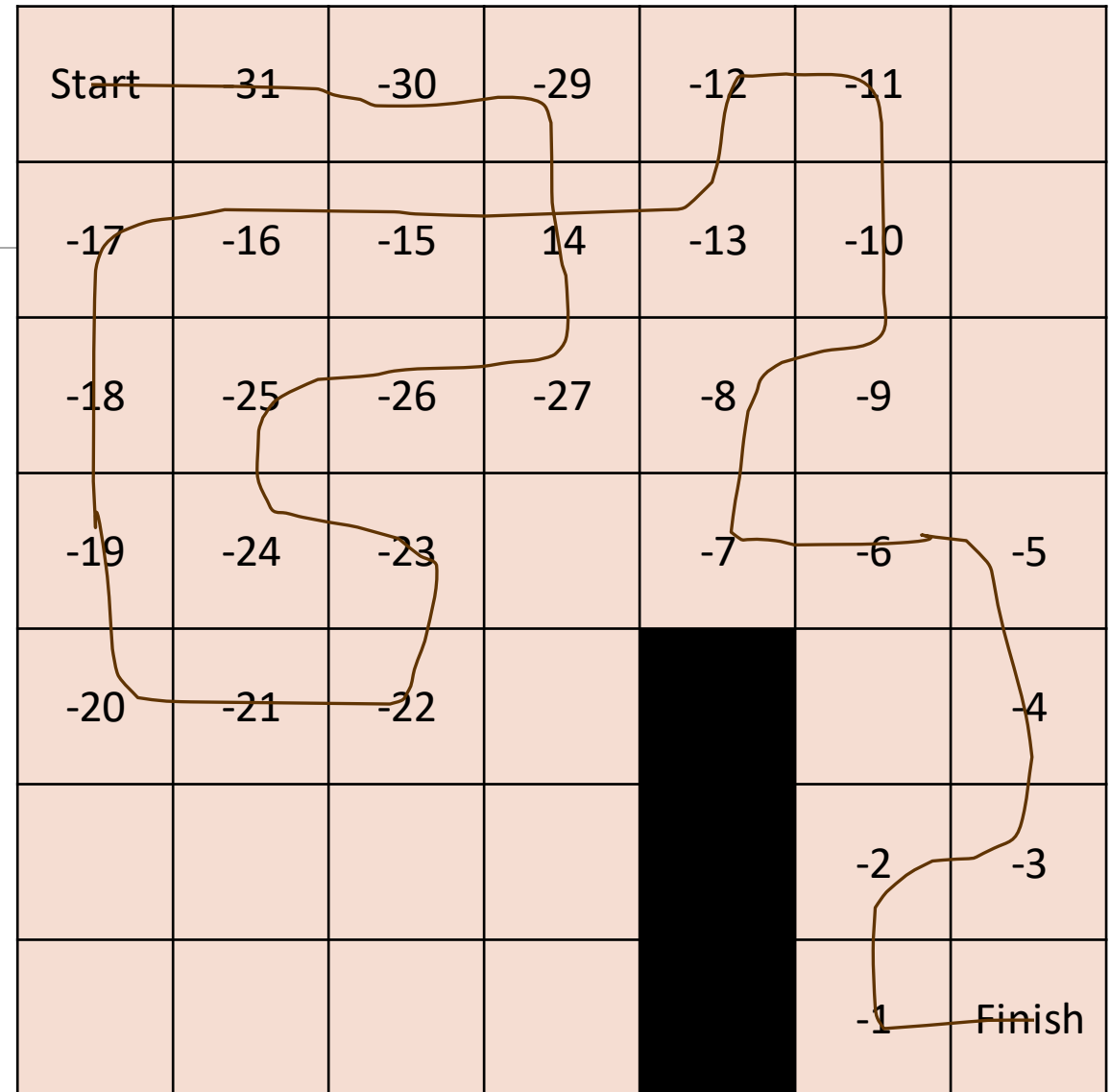
1. Start with an **Initial Policy**, π_0 , that is random.
2. Collect a **Sample Trajectory**
3. Use the collected sample to **compute Values** of the states the agent has visited.
4. Update the policy;
$$\pi_1 \leftarrow \text{UPDATE}(\pi_0)$$
5. Repeat until convergence



Maze Example

Example Algorithm for a Value-based Method:

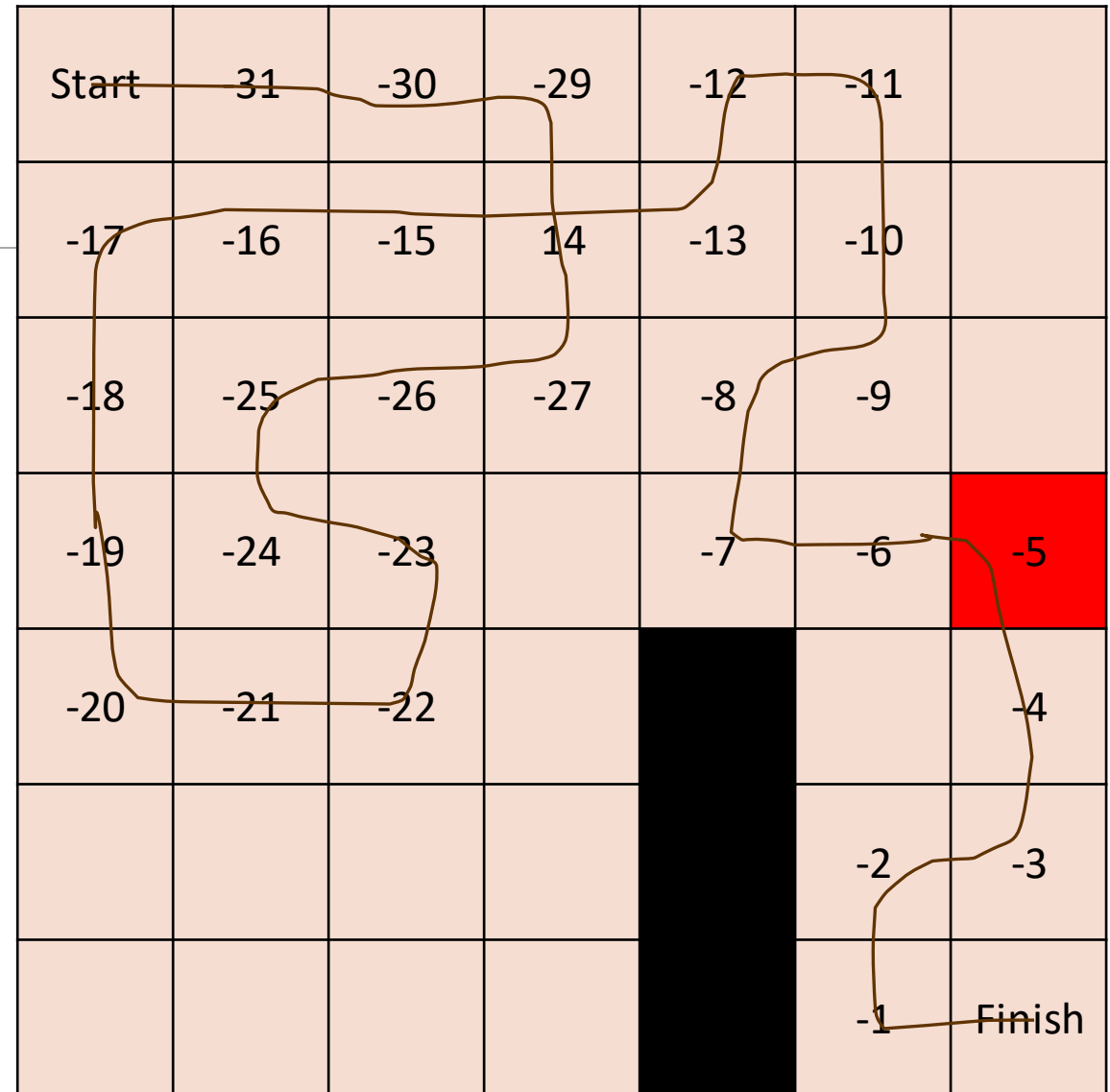
1. Start with an **Initial Policy**, π_0 , that is random.
2. Collect a **Sample Trajectory**
3. Use the collected sample to **compute Values** of the states the agent has visited.
4. Update the policy;
 $\pi_1 \leftarrow \text{UPDATE}(\pi_0)$
5. Repeat until convergence



Maze Example

Example Algorithm for a Value-based Method:

1. Start with an **Initial Policy**, π_0 , that is random.
2. Collect a **Sample Trajectory**
3. Use the collected sample to **compute Values** of the states the agent has visited.
4. Update the policy;
$$\pi_1 \leftarrow \text{UPDATE}(\pi_0)$$
5. Repeat until convergence



Maze Example

Example Algorithm for a Value-based Method:

1. Start with an **Initial Policy**, π_0 , that is random.
2. Collect a **Sample Trajectory**
3. Use the collected sample to **compute Values** of the states the agent has visited.
4. Update the policy;
 $\pi_1 \leftarrow \text{UPDATE}(\pi_0)$
5. Repeat until convergence

$\pi_0(\text{Red cell}) =$

UP	DOWN	LEFT	RIGHT
25%	25%	25%	25%

$\pi_1(\text{Red cell}) =$

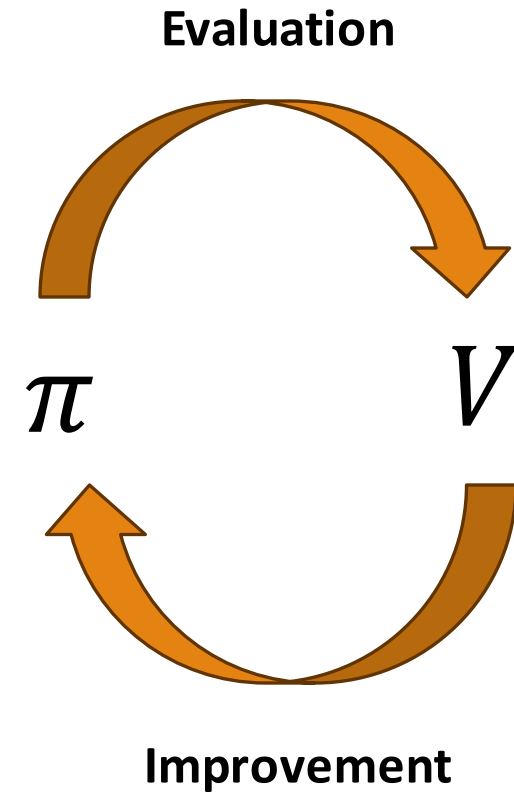
UP	DOWN	LEFT	RIGHT
0%	100%	0%	0%

Value-based Algorithm

Example Algorithm for a Value-based Method:



1. Start with an **Initial Policy**, π_0 , that is random.
2. Collect a **Sample Trajectory**
3. Use the collected sample to **compute Values** of the states the agent has visited.
4. Update the policy;
 $\pi_1 \leftarrow \text{UPDATE}(\pi_0)$

5. Repeat until convergence



Value-based Algorithm

Example Algorithm for a Value-based Method:

1. Start with an **Initial Policy**, π_0 , that is random.
2. Collect a **Sample Trajectory**
3. Use the collected sample to **compute Values** of the states the agent has visited.  For the second update and onwards, we take an **incremental mean**
4. Update the policy;
 $\pi_1 \leftarrow \text{UPDATE}(\pi_0)$  Typically use ϵ –**Greedy** update for exploration
5. Repeat until convergence

Monte-Carlo Policy Evaluation

Goal: Learn V_π from sample trajectories under policy π

$$s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_H \sim \pi$$

Recall that

$$V^\pi(s_t) = \mathbb{E}_{\tau_{a_t:a_H} \sim p_\pi(\tau|s_t)}[G_t|s_t]$$

Since we use samples instead of the distribution function, we use an **empirical mean**:

$$V^\pi(s_t) \cong \frac{1}{N} \sum_{i=1}^N G_{t,i}(s_t)$$

N denotes the number of samples.

Incremental Mean

The mean μ_1, μ_2, \dots of a sequence x_1, x_2, \dots can be computed incrementally,

$$\begin{aligned}\mu_k &= \frac{1}{k} \sum_{i=1}^k x_i \\ &= \frac{1}{k} \left(x_k + \sum_{i=1}^{k-1} x_i \right) \\ &= \frac{1}{k} (x_k + (k-1)\mu_{k-1}) \\ &= \mu_{k-1} + \frac{1}{k} (x_k - \mu_{k-1})\end{aligned}$$

Incremental Monte-Carlo Updates

Update V_π incrementally after an episode.

For each state, s_t , with return, G_t

$N(s_t)$ is the number of visits of the state, s_t $N(s_t) \leftarrow N(s_t) + 1$

$$V(s_t) \leftarrow V(s_t) + \frac{1}{N(s_t)} (G_t - V(s_t))$$

In many complicated cases, it is usually unnecessary to count the number of visits. We fix the fraction term to a small constant value, which we call it the “Learning Rate”

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

ϵ -Greedy Policy Update



- There are two doors in front of you
- You open the left door and get reward 0
 - $V(\text{left}) = 0$
- You open the right door and get reward +1
 - $V(\text{right}) = 1$
- You open the right door and get reward +2
 - $V(\text{right}) = 1.5$
- You open the right door and get reward +3
 - $V(\text{right}) = 2.25$
- Are you sure you have chosen the best door?
 - What if the left door provides a reward of 0 with only 10% of chance and provides a reward of 10 with 90% of chance?
 - We need to **Explore** more in order to estimate a solid **state transition probability**.

ϵ -Greedy Policy Update

- Simplest idea for ensuring continual exploration is the epsilon-greedy exploration.
- All m actions are tried with non-zero probability
- With probability $1 - \epsilon$ choose the greedy action
- With probability ϵ choose an action at random

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + (1 - \epsilon) & \text{if } a^* = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a) \\ \frac{\epsilon}{m} & \text{otherwise} \end{cases}$$

Maze Example

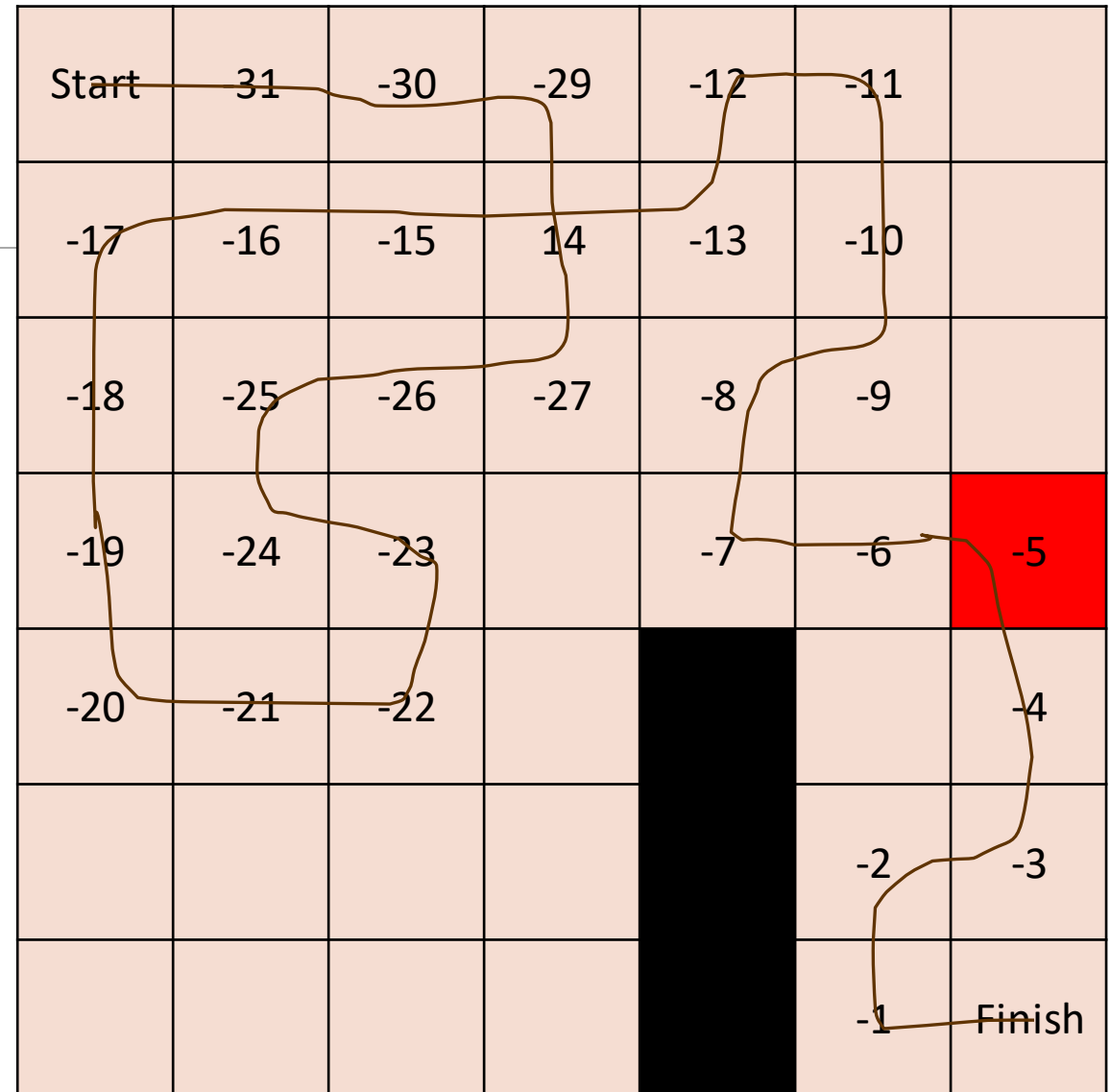
For an ε -greedy policy with $\varepsilon = 0.1$

$\pi_0(\text{Red cell}) =$

UP	DOWN	LEFT	RIGHT
25%	25%	25%	25%

$\pi_1(\text{Red cell}) =$

UP	DOWN	LEFT	RIGHT
2.5%	92.5%	2.5%	2.5%



Monte-Carlo (MC) Algorithm

Monte-Carlo Algorithm

1. Start with an **Initial Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)

$$S_0, a_0, r_0, S_1, a_1, r_1, \dots, S_H$$

3. Compute **mean returns** for each state

$$G_t \cong \frac{1}{k} \sum_{i=1}^k G_{t,i}$$

4. Update the **Value function**

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

5. Update the **ϵ -greedy policy**
6. Repeat until convergence

Monte-Carlo (MC) Algorithm

Problems with Monte-Carlo Algorithm

- The samples must be a full trajectory. An episode must be terminated.
 - What if the environment does not have a terminal state?
 - Even when the terminal states exist;
 - Computationally inefficient to run a full episode (CPU)
 - Computationally heavy to compute the return at every states (GPU)
- MC has high variance.
 - The return at each state depends on many random actions, transitions and rewards.

Monte-Carlo (MC) Algorithm

Monte-Carlo Algorithm

1. Start with an **Initial Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)

$$S_0, a_0, r_0, S_1, a_1, r_1, \dots, S_H$$

3. Compute **mean returns** for each state

$$G_t \cong \frac{1}{k} \sum_{i=1}^k G_{t,i}$$

4. Update the **Value function**

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

5. Update the **ϵ -greedy policy**
6. Repeat until convergence

We often do not want to collect a full trajectory.

But we need a full trajectory to compute the return.

Is there a way to estimate the return(or value) without reaching the terminal state?

Temporal Difference (TD)

We use the Bellman Equation to predict a value of the current step, V_t , by using the estimated value of the next step, V_{t+1} .

Bellman Equation of Value for one step bootstrapping

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi(a_t|s_t), s_{t+1} \sim p(s_{t+1}|s_t, a_t)} [r_t + \gamma V^\pi(s_{t+1}) | s_t]$$

We can write a simplified version of the above Bellman equation as follows;

$$V(s_t) \cong r_t + \gamma V(s_{t+1})$$

Note that we have depreciated the expectation symbol since in the upcoming case, we are only considering a single step

Temporal Difference (TD) Algorithm

Temporal Difference Algorithm

1. Start with an **Initial Policy**, π_0
2. Collect a **Sample Step**

$$s_t, a_t, r_t, s_{t+1}$$

Unlike MC, here we do not have to run the environment until the terminal state. Just 1 step is enough!

3. Compute **Temporal Difference Target** at the state

$$r_t + \gamma V(s_{t+1})$$

TD target

4. Update the **Value function**

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

5. Update the **ϵ -greedy policy**
6. Repeat until convergence

TD error

Temporal Difference (TD) Algorithm

Temporal Difference Algorithm Terminology

TD Target Value $r_t + \gamma V(s_{t+1})$

TD Target Error $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$

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

Is equivalent to:

$$V(s_t) \leftarrow V(s_t) + \alpha \delta_t$$

Policy Structure

An example of a policy in a maze environment

$\pi_0(\text{Red cell}) =$	UP	DOWN	LEFT	RIGHT
	25%	25%	25%	25%
$\pi_1(\text{Red cell}) =$	UP	DOWN	LEFT	RIGHT
	2.5%	92.5%	2.5%	2.5%

This is an informal way of defining the policy.

Formally, we define a Greedy(deterministic) Policy as;

$$\pi(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a)$$

Note that this is not a probability function anymore for a deterministic policy.

Considering the policy structure, it makes more sense to predict Q values instead of V

Monte-Carlo (MC) Algorithm

Monte-Carlo Algorithm – Q

1. Start with an **Initial Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)

$$S_0, a_0, r_0, S_1, a_1, r_1, \dots, S_H$$

3. Compute **mean returns** for each state

$$G_t \cong \frac{1}{k} \sum_{i=1}^k G_{t,i}$$

4. Update the **Q function**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(G_t - Q(s_t, a_t))$$

5. Update the **ϵ -greedy policy**
6. Repeat until convergence

Bellman Equation for Q

Bellman Equation of Q for one step bootstrapping

$$Q^\pi(s_t, a_t) = r_t + \mathbb{E}_{a_t \sim \pi(a_t|s_t), s_{t+1} \sim p(s_{t+1}|s_t, a_t)} [\gamma Q^\pi(s_{t+1}, a_{t+1}) | s_t, a_t]$$

We can write a simplified version of the above Bellman equation as follows;

$$Q(s_t, a_t) \cong r_t + \gamma Q(s_{t+1}, a_{t+1})$$

Note that we have depreciated the expectation symbol since in the upcoming case, we are only considering a single step

SARSA

$$Q(s_t, a_t) \cong r_t + \gamma Q(s_{t+1}, a_{t+1})$$

Data of 1 step

s_t, a_t, r_t, s_{t+1}

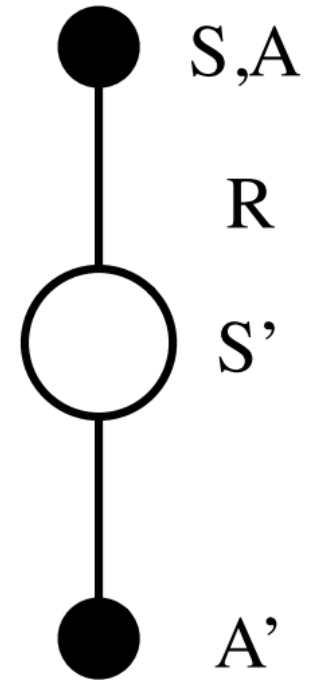
A usual 1 step sample is not enough!
We need the next action.

We can easily obtain the next action by
using our policy!

Next
state

Next
action

We call the algorithm that
uses this information a
SARSA



SARSA Algorithm

SARSA Algorithm

1. Start with an **Initial Policy**, π_0
2. Collect a **Sample Step**

$$s_t, a_t, r_t, s_{t+1}$$

3. Evaluate the current policy at state, s_{t+1}
4. Compute **Temporal Difference Target** at the state

$$r_t + \gamma Q(s_{t+1}, a_{t+1})$$

5. Update the **Q function**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

6. Update the **ϵ -greedy policy**
7. Repeat until convergence

SARSA Algorithm

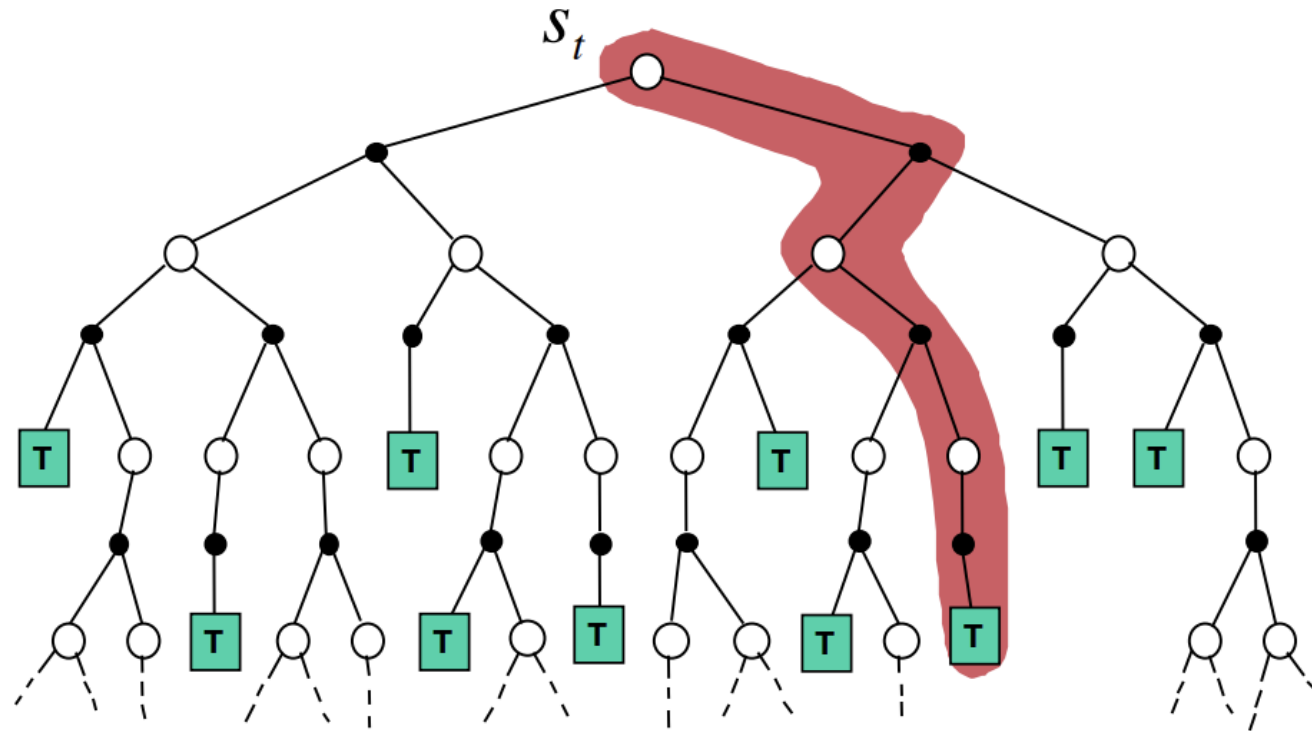
```
Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$   
Repeat (for each episode):  
    Initialize  $S$   
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)  
    Repeat (for each step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

From David Silver's lecture of Reinforcement Learning

<https://www.deepmind.com/learning-resources/introduction-to-reinforcement-learning-with-david-silver>

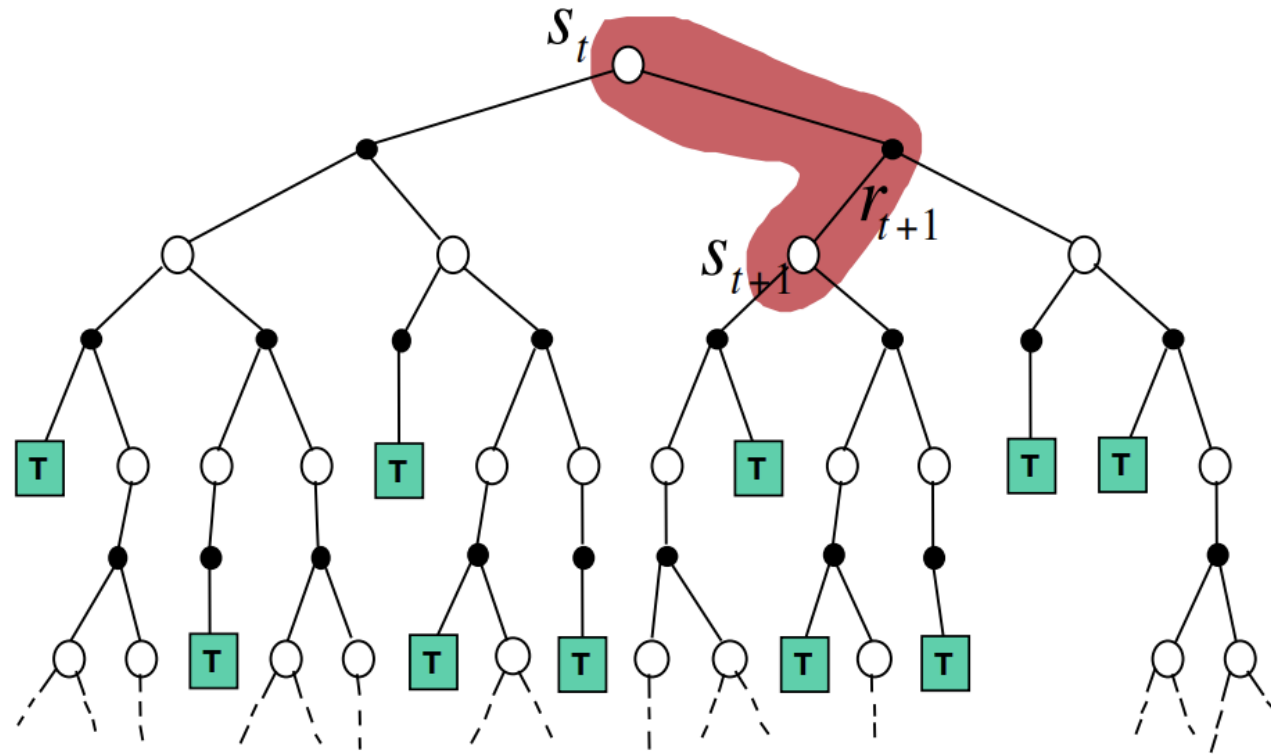
Monte-Carlo Update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(G_t - Q(s_t, a_t))$$



TD/SARSA Update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$



MC VS TD

Monte Carlo (MC)

Good:

- Zero Bias
- Good convergence properties (even with function approximation)
- Not very sensitive to initial value
- Very simple to understand and use

Bad:

- High Variance
- Must wait until an episode terminates
- Cannot be used for environments with no terminal states.

Temporal Difference (TD)

Good:

- Low Variance
- Usually more efficient than MC
- No need for terminal states.

Bad:

- Some Bias
- Does converge to the optimal V or Q (but not always with function approximation)
- More sensitive to initial value

Blackjack Example

- States (200 of them)
 - Current sum (12 ~ 21)
 - Dealer's showing card (ace ~ 10)
 - Do I have a "useable" ace? (yes or no)
- Action
 - Hit: take another card without replacement
 - Stick: Stop receiving cards and terminate
- Rewards for stick
 - +1 if my sum > dealer's sum
 - 0 if my sum = dealer's sum
 - -1 if my sum < dealer's sum
- Rewards for hit
 - -1 if my sum > 21 and terminate
 - 0 otherwise
- Transitions: automatically hit if my sum < 12



Blackjack Example

Q	Dealer showing 6									
	12	13	14	15	16	17	18	19	20	21
Hit	0.9	0.8	0.7	0.7	0.6	0	0	0	0	0
Stick	0.5	0.6	0.7	0.8	0.9	1	1	1	1	1

Q	Dealer showing 10									
	12	13	14	15	16	17	18	19	20	21
Hit	0.95	0.95	0.95	0.9	0.8	0.7	0.5	0.4	0.3	0
Stick	-0.95	-0.9	-0.9	-0.8	-0.7	0.2	0.5	0.7	0.8	1

MC for Blackjack

Monte-Carlo Algorithm for Blackjack

1. Start with an **Existing Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)

States				
Dealer's Card	10	10	10	10
My Card Sum	13	15	18	20
Usable ace?	No	No	No	No
Action Chosen	Hit	Hit	Hit	Stick
Reward	0	0	0	+1

MC for Blackjack

Monte-Carlo Algorithm for Blackjack

1. Start with an **Existing Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)
3. Compute **mean returns** for each state

$G([10,13,\text{No}]) = +1$

$G([10,15,\text{No}]) = +1$

$G([10,18,\text{No}]) = +1$

$G([10,20,\text{No}]) = +1$

States				
Dealer's Card	10	10	10	10
My Card Sum	13	15	18	20
Usable ace?	No	No	No	No
Action Chosen	Hit	Hit	Hit	Stick
Reward	0	0	0	+1

MC for Blackjack

Monte-Carlo Algorithm for Blackjack

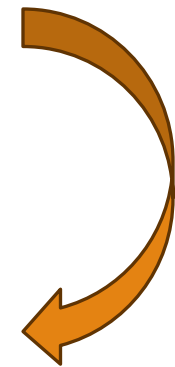
1. Start with an **Existing Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)
3. Compute **mean returns** for each state
4. Update the **Q function**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(G_t - Q(s_t, a_t))$$

$$\begin{aligned} G([10,13,\text{No}]) &= +1 \\ G([10,15,\text{No}]) &= +1 \\ G([10,18,\text{No}]) &= +1 \\ G([10,20,\text{No}]) &= +1 \end{aligned}$$

Q	Dealer showing 10									
	12	13	14	15	16	17	18	19	20	21
Hit	0.95	0.95	0.95	0.9	0.8	0.7	0.5	0.4	0.3	0
Stick	-0.95	-0.9	-0.9	-0.8	-0.7	0.2	0.5	0.7	0.8	1
Q	Dealer showing 10									
	12	13	14	15	16	17	18	19	20	21
Hit	0.95	0.95	0.95	0.9	0.8	0.7	0.5	0.4	0.3	0
Stick	-0.95	-0.88	-0.9	-0.78	-0.7	0.2	0.5	0.7	0.8	1

$$\alpha = 0.01$$



Update

States				
Dealer's Card	10	10	10	10
My Card Sum	13	15	18	20
Usable ace?	No	No	No	No
Action Chosen	Hit	Hit	Hit	Stick
Reward	0	0	0	+1

SARSA for Blackjack

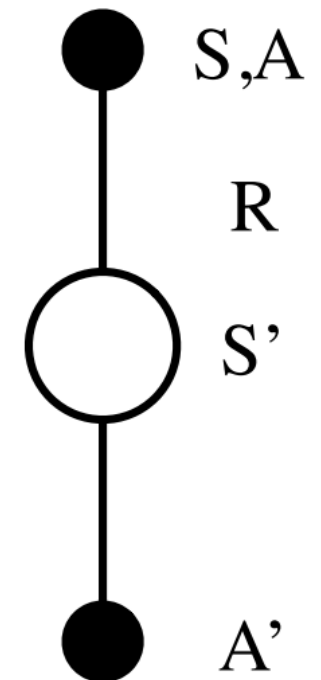
Monte-Carlo Algorithm for Blackjack

1. Start with an **Existing Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)

States		
Dealer's Card	10	10
My Card Sum	13	15
Usable ace?	No	No
Action Chosen	Hit	Hit
Reward	0	

Diagram illustrating the SARSA algorithm components and transitions:

- State**: Points to the first column of the table (Dealer's Card, My Card Sum, Usable ace?).
- Action**: Points to the 'Hit' action in the 'Action Chosen' row.
- Reward**: Points to the '0' reward in the 'Reward' row.
- Next State**: Points to the second column of the table (Dealer's Card, My Card Sum, Usable ace?).
- Next Action**: Points to the empty cell in the 'Action Chosen' row of the second column.



SARSA for Blackjack

Monte-Carlo Algorithm for Blackjack

1. Start with an **Existing Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)
3. Compute **Temporal Difference Target** at the state

$$r_t + \gamma Q(s_{t+1}, a_{t+1})$$

States		
Dealer's Card	10	10
My Card Sum	13	15
Usable ace?	No	No
Action Chosen	Hit	Hit
Reward	0	

Q	Dealer showing 10									
	12	13	14	15	16	17	18	19	20	21
Hit	0.95	0.95	0.95	0.9	0.8	0.7	0.5	0.4	0.3	0
Stick	-0.95	-0.9	-0.9	-0.8	-0.7	0.2	0.5	0.7	0.8	1

$$Q([10, 13, No], Hit) = 0 + 0.9 * Q([10, 15, No], Hit)$$

$$Q([10, 13, No], Hit) = 0 + 0.9 * 0.9 = 0.81$$

SARSA for Blackjack

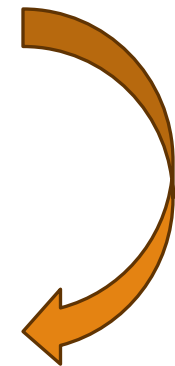
Monte-Carlo Algorithm for Blackjack

1. Start with an **Existing Policy**, π_0
2. Collect a **Sample Trajectory**(that is until the terminal step)
3. Compute **Temporal Difference Target** at the state
4. Update the **Q function**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Q	Dealer showing 10									
	12	13	14	15	16	17	18	19	20	21
Hit	0.95	0.95	0.95	0.9	0.8	0.7	0.5	0.4	0.3	0
Stick	-0.95	-0.9	-0.9	-0.8	-0.7	0.2	0.5	0.7	0.8	1
Q	Dealer showing 10									
	12	13	14	15	16	17	18	19	20	21
Hit	0.95	0.95	0.95	0.9	0.8	0.7	0.5	0.4	0.3	0
Stick	-0.95	-0.88	-0.9	-0.8	-0.7	0.2	0.5	0.7	0.8	1

$\alpha = 0.01$

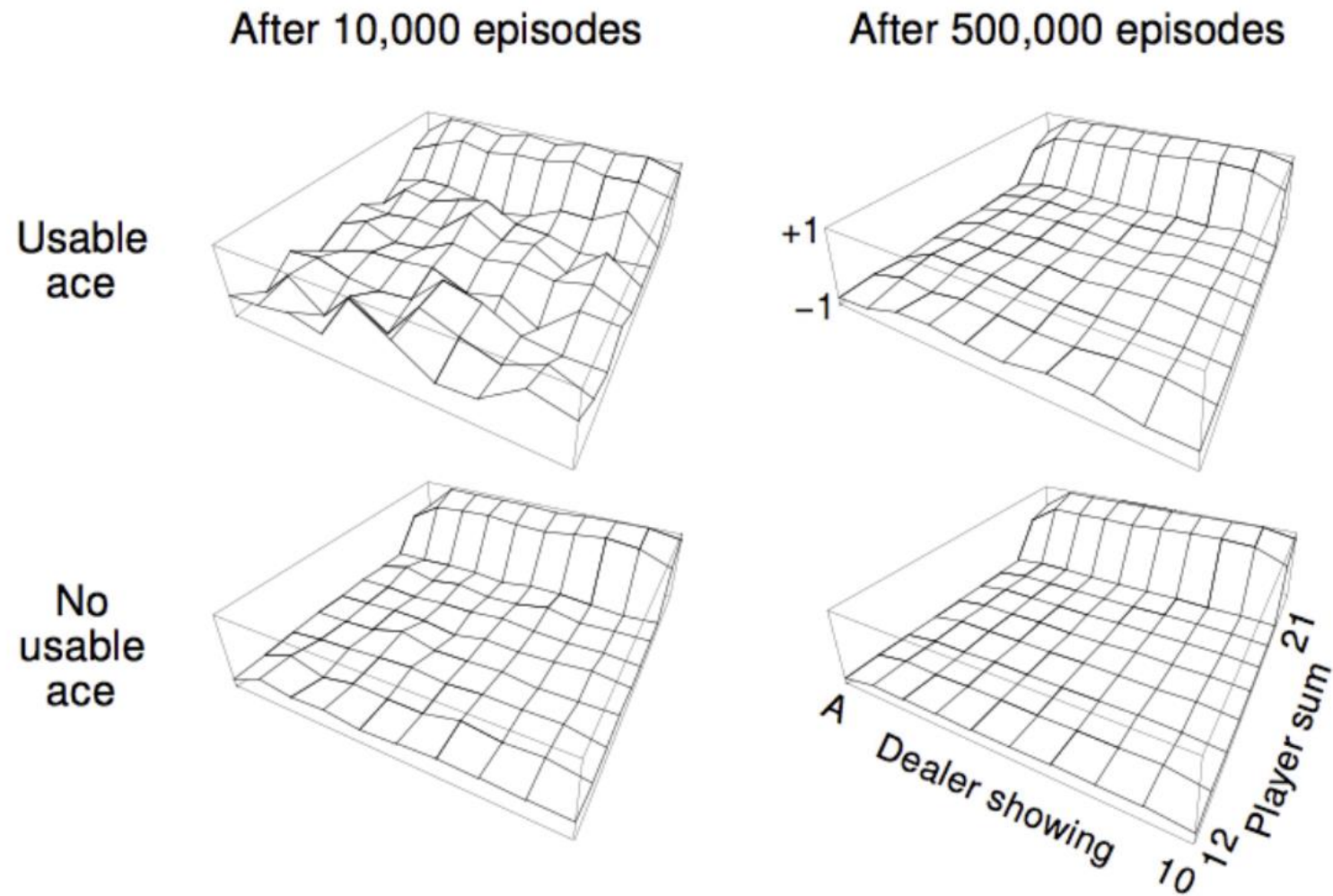


Update

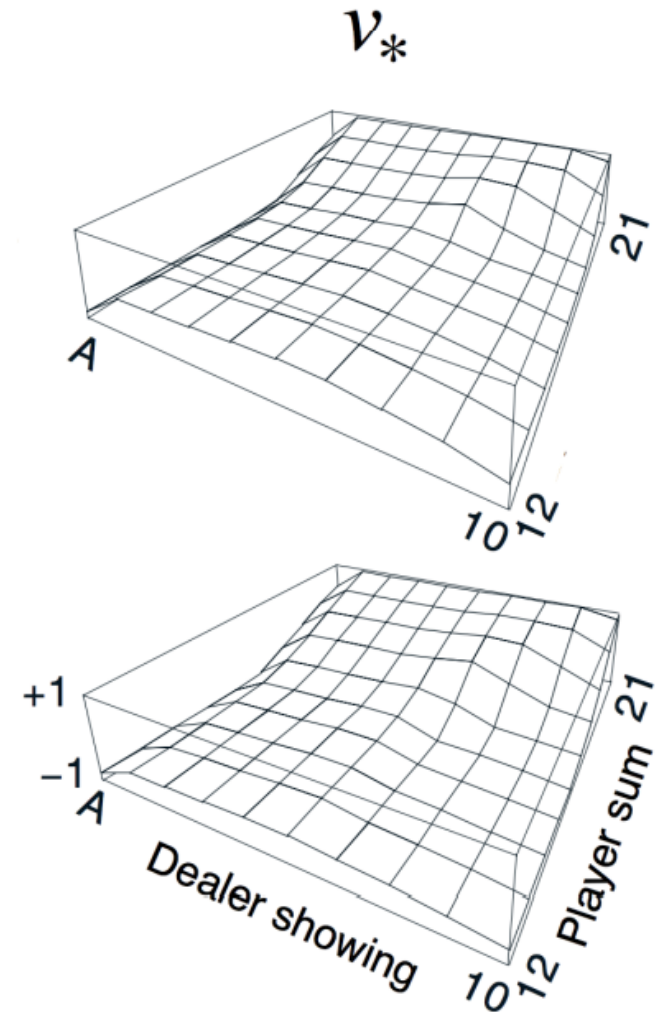
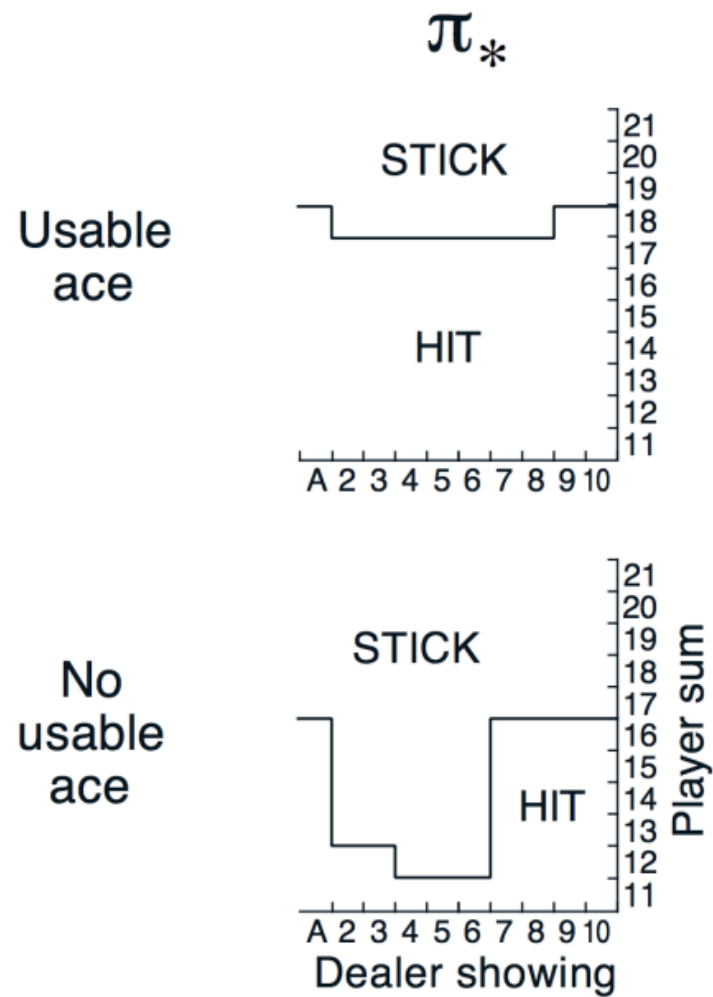
States		
Dealer's Card	10	10
My Card Sum	13	15
Usable ace?	No	No
Action Chosen	Hit	Hit
Reward	0	

$$Q([10, 13, No], Hit) = 0.81$$

Blackjack Example



Blackjack Example



On-Policy VS Off-Policy

On-Policy

- “Learn on the job”
- Learn about a policy from experiences sampled from that policy.

Off-Policy

- “Look over someone’s shoulder”
- Learn about a policy from experiences sampled from other policies.

Next Lecture

Q-Learning

Introduction to Deep Learning