

Introduction to Reinforcement Learning based Control and Deep Reinforcement Learning based control

MAYank Shekhar JHA

Associate Professor
(Maitre de Conférences)

Automatic Control, Reliability of Systems

mayank-shekhar.jha@univ-lorraine.fr



Research

Centre de Recherche en Automatique de Nancy
(CRAN) UMR 7039,
Faculté des Sciences et Technologies
Boulevard des Aiguillettes - BP 70239 - Bât.
1er cycle 54506 Vandoeuvre-lès-Nancy
Cedex France



Teaching

Polytech Nancy (ESSTIN),
2 Rue Jean Lamour 54509
Vandoeuvre-lès-Nancy,
Cedex France



JHA, mayank-shekhar.jha@univ-lorraine.fr

Introduction

Markov Decision process

Dynamic Programming for planning
and control

Model Free Prediction and Control
(SARSA and Q-learning)

Function Approximation (Use Deep Neural Networks)

Deep Q networks (deep learning for RL)

Deep Deterministic Policy Gradient Method

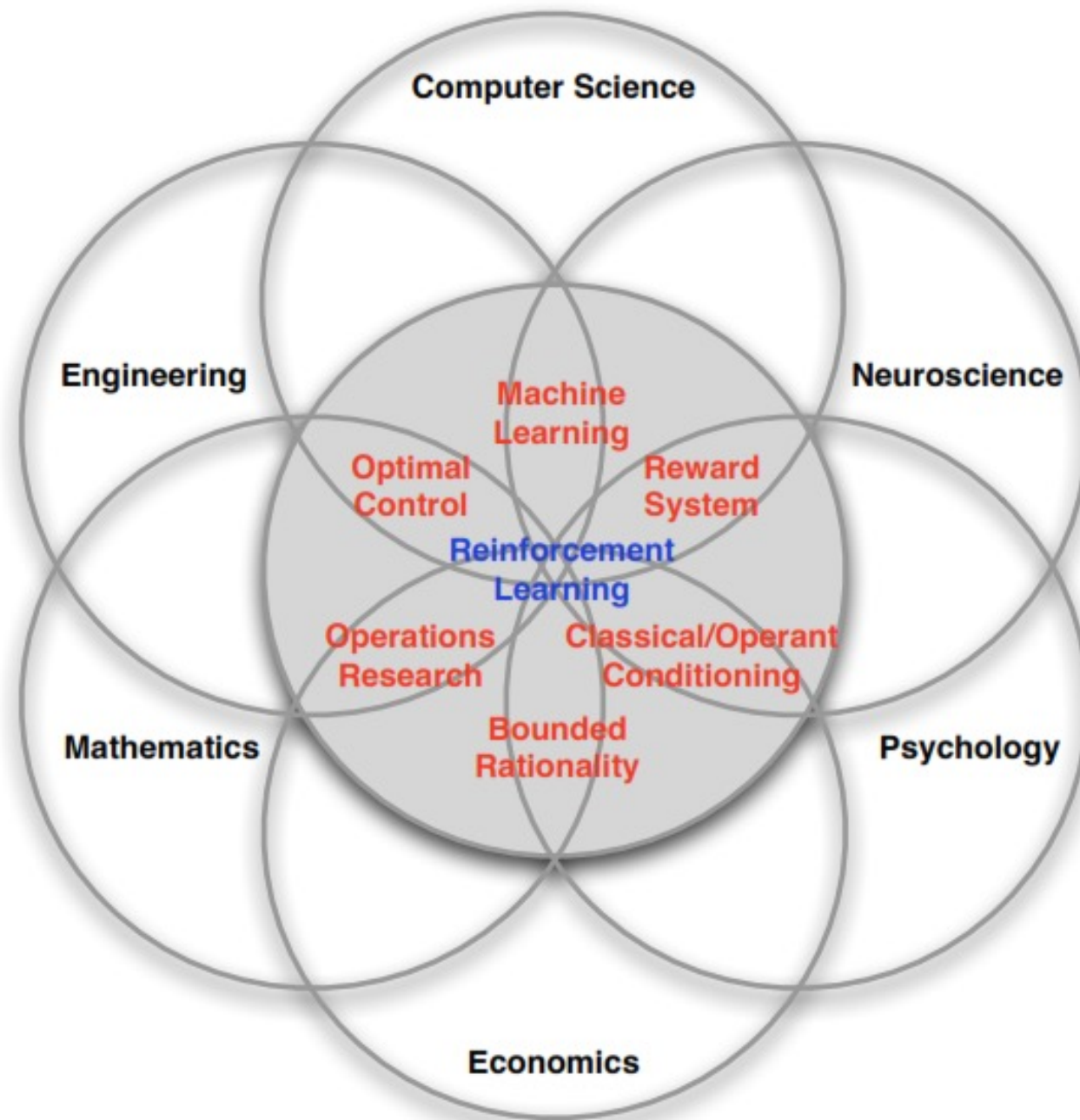
Introduction

Environment

Agent

Reward

Policy



Motivation

Reinforcement Learning: Towards human level :

control ((Finding the optimal way of doing a given task)

prediction

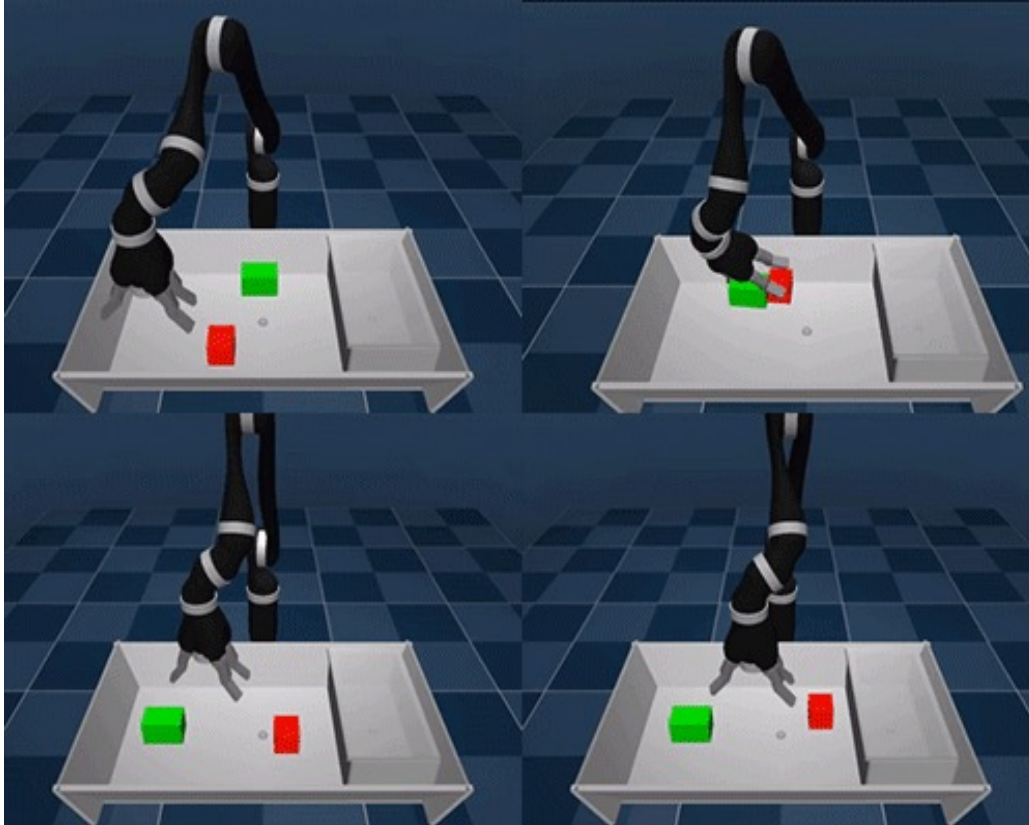
Adaptation (Robots That Can Adapt like Animals, *Nature*)



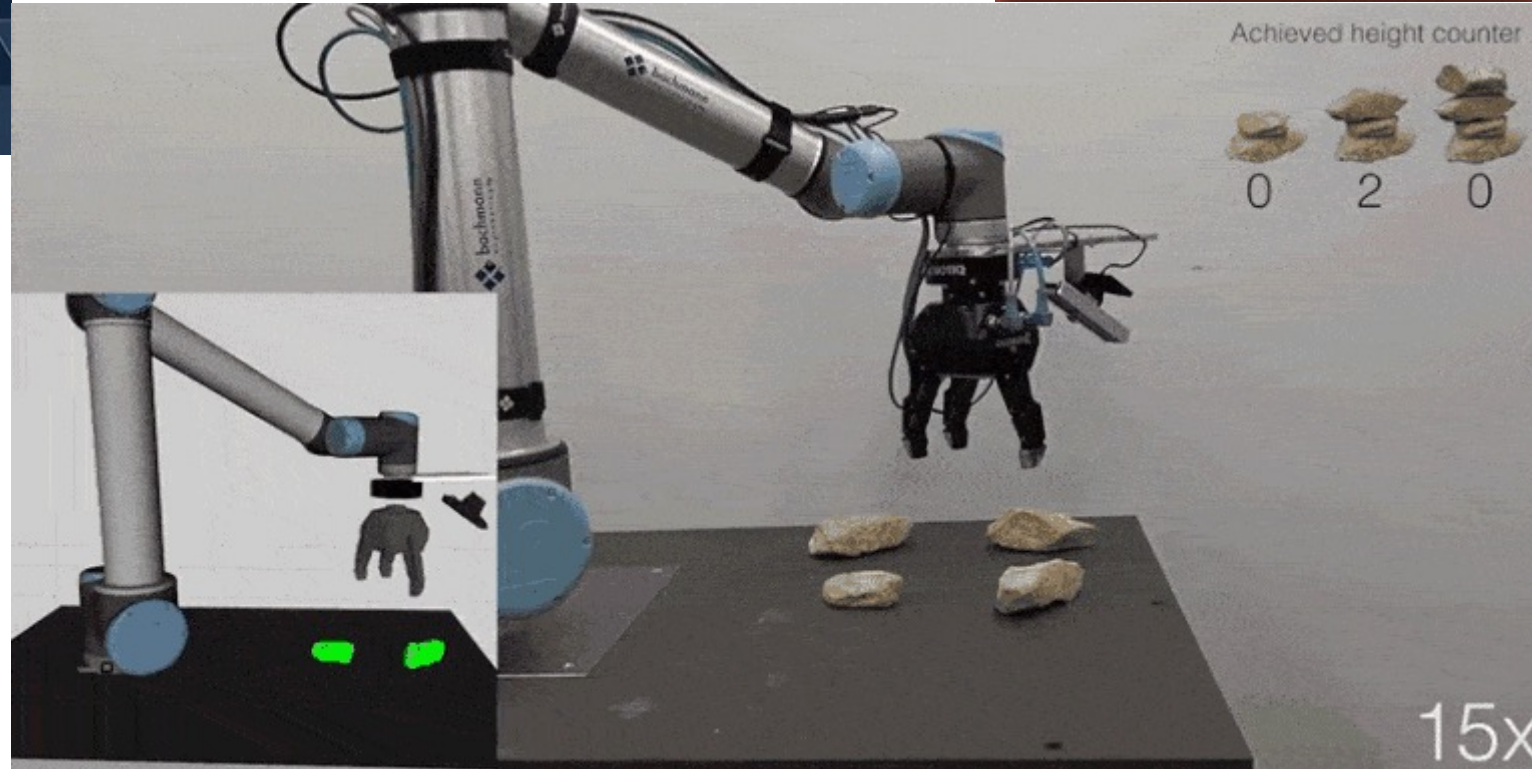
Built
new
moves



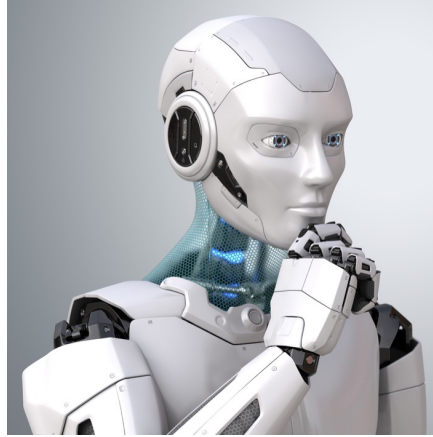
Some Applications



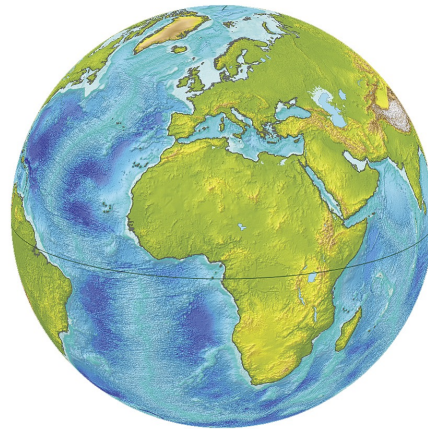
Source : Deep Mind



Agent and Environment



Agent/controller

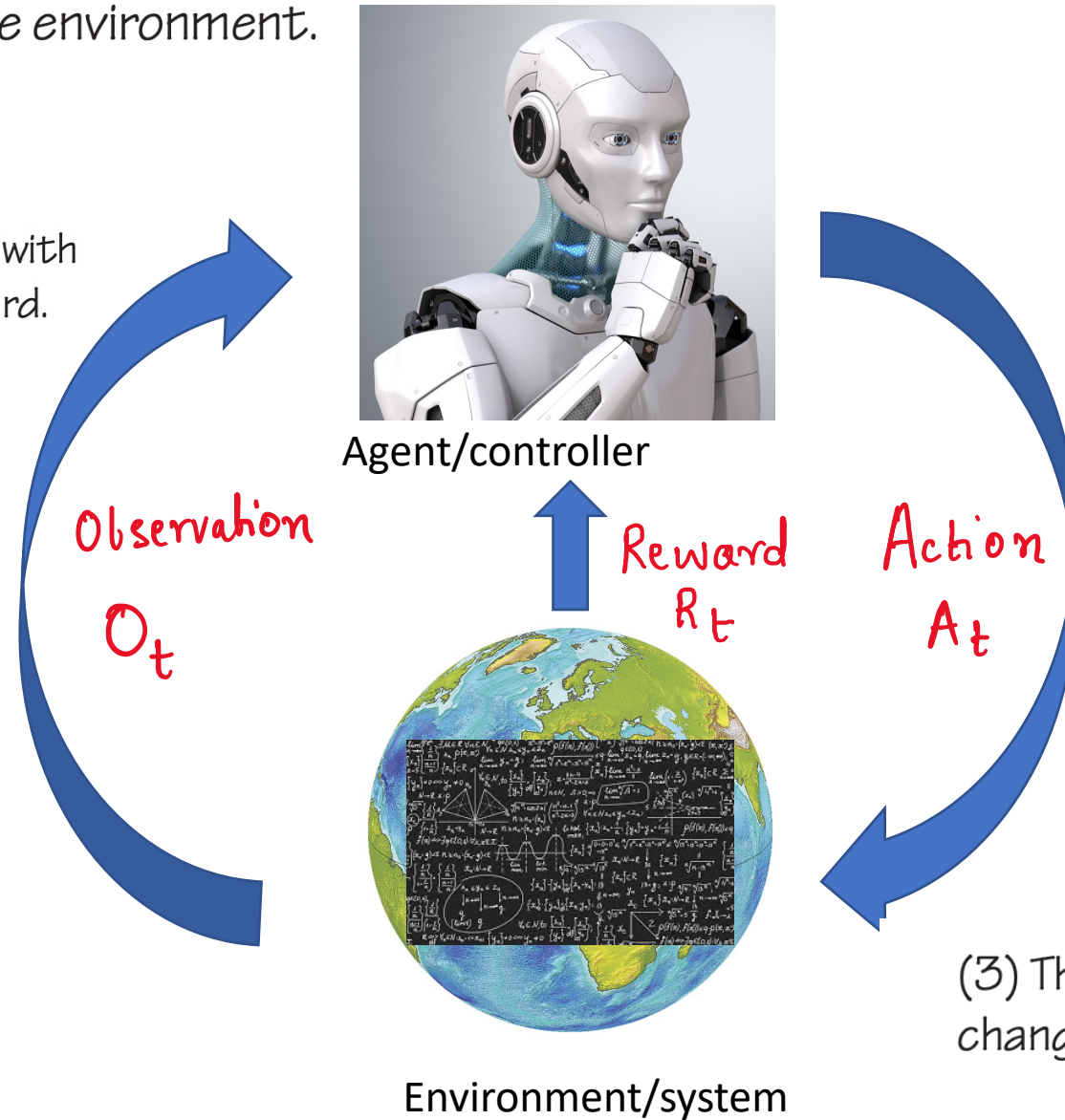


Environment/system

Agent and Environment Interaction

(1) Agent perceives the environment.

(4) The environment reacts with new observation, and a reward.



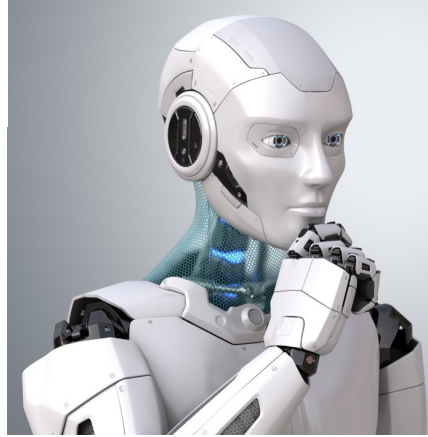
(2) Agent takes an action.

(3) The environment goes through internal state change as a consequence of the agent's action.

Agent and Environment Interaction

(1) The cycle begins with the agent observing the environment.

Observation
and Reward



Agent/controller

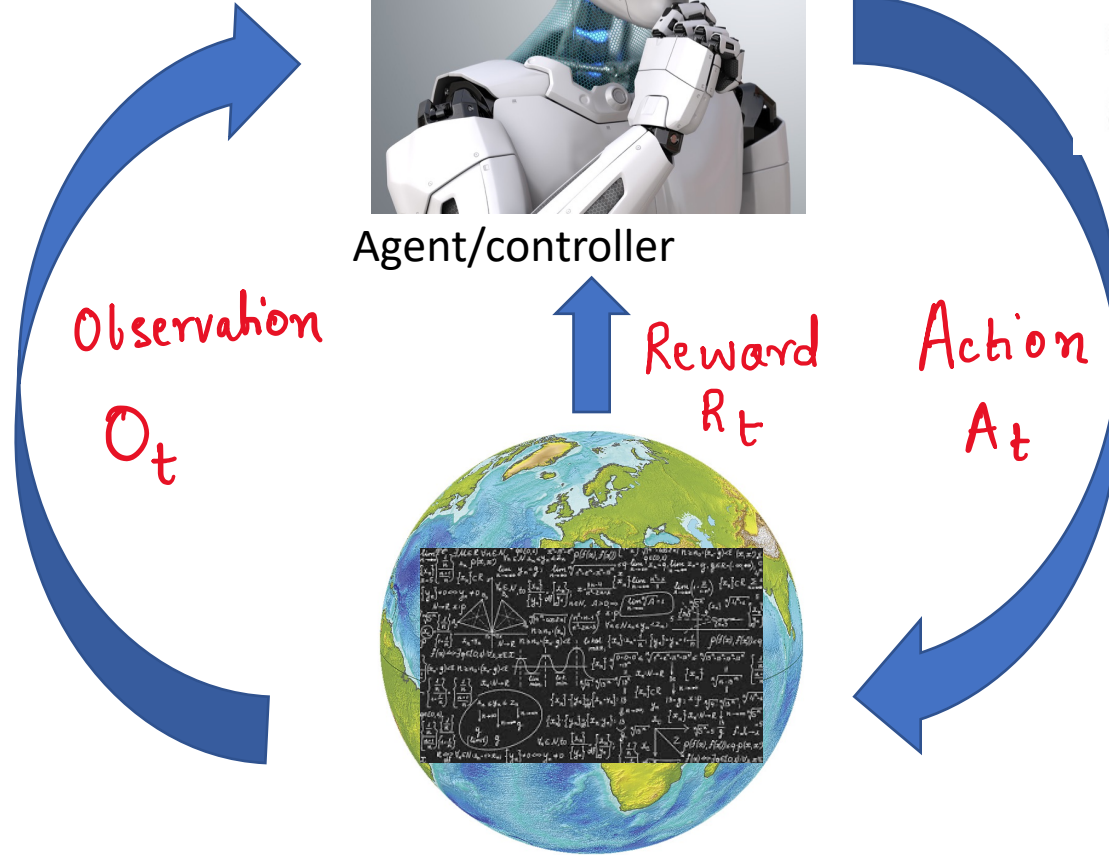
(2) The agent uses this observation and reward to attempt to improve at the task.

Observation
 O_t

Reward
 R_t

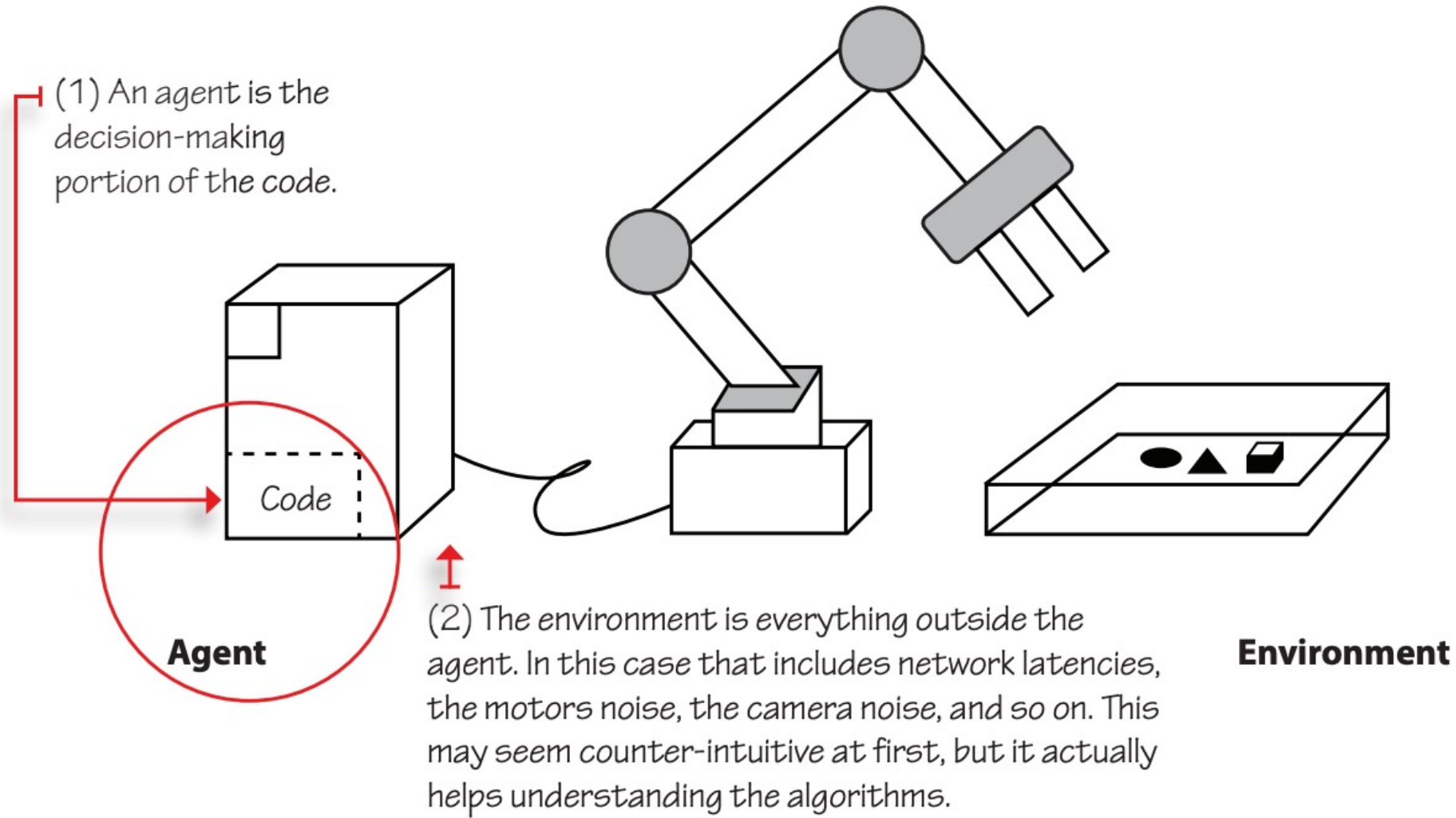
Action
 A_t

(3) It then sends an action to the environment in an attempt to control it in a favorable way.

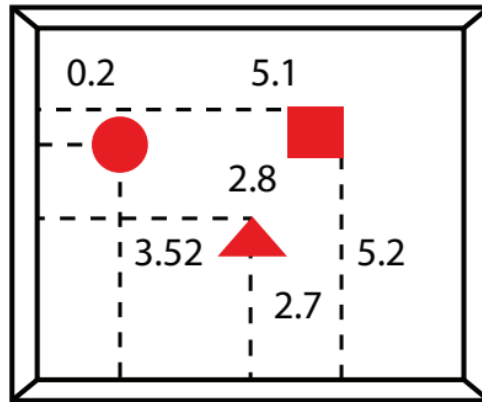


Environment/system

Boundary between Agent and Environment



State v/s Observations

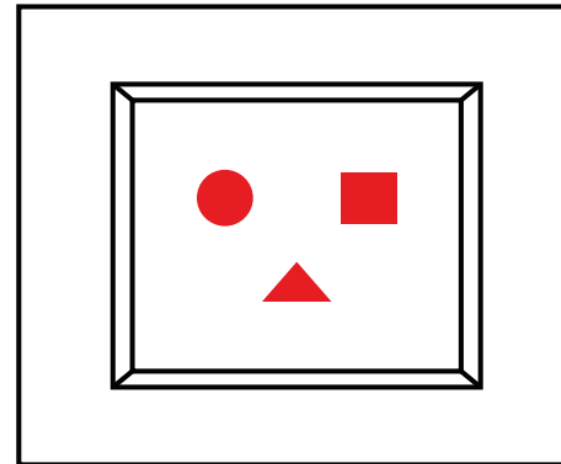


State: true locations

(1) States are the perfect and complete information related to the task at hand.

Observation: just an image

(2) While observations are the information the agent receives. This could be noisy or incomplete information.



Rewards and Reinforcement Learning

Rewards :

- functions-→ chosen by user, problem dependent.

- scalar feedback signal

- indicates how well the agent does

Agent's mission:

- maximize the cumulated reward

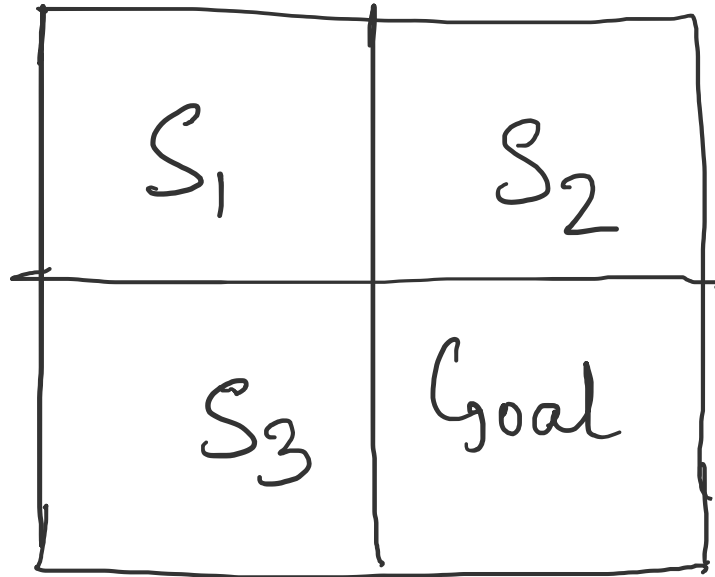
Reinforcement learning :

- finding best way (optimal way) to maximize the cumulative reward.

- finding optimal “policy” or strategy to achieve the goal in an optimal manner.

- sequential decision-making problem.

Grid World Example



What is the environment?

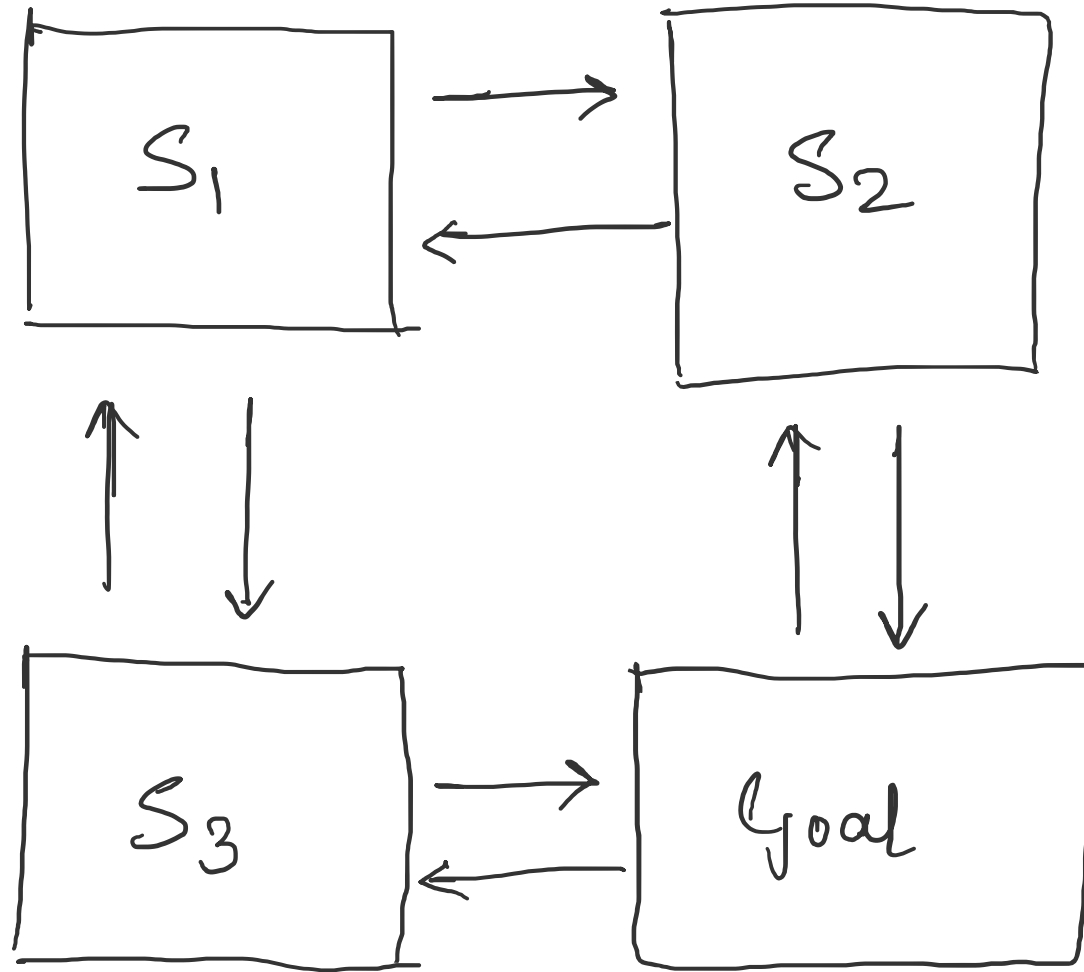
What are the states?

What is agent's role?

A small Grid world → Environment

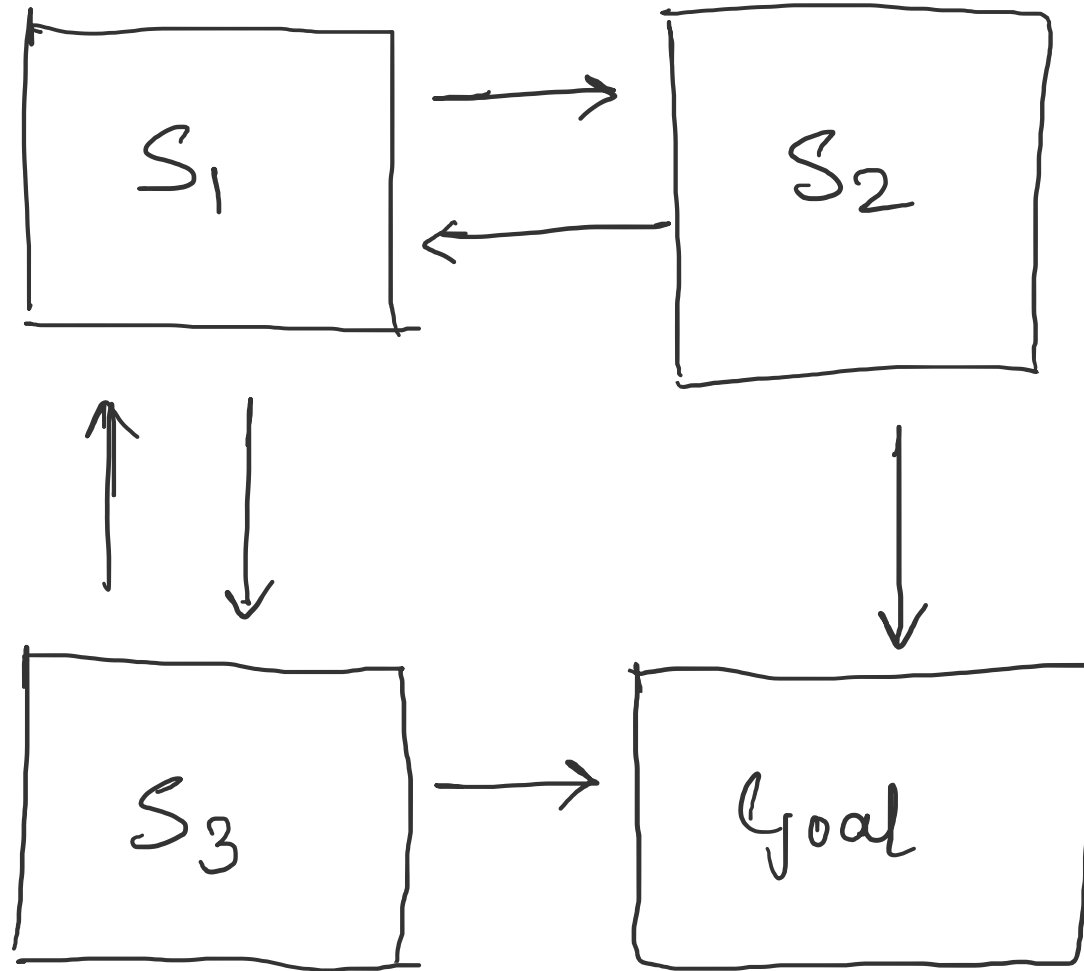
An agent lives in it → Agent

Grid World Example



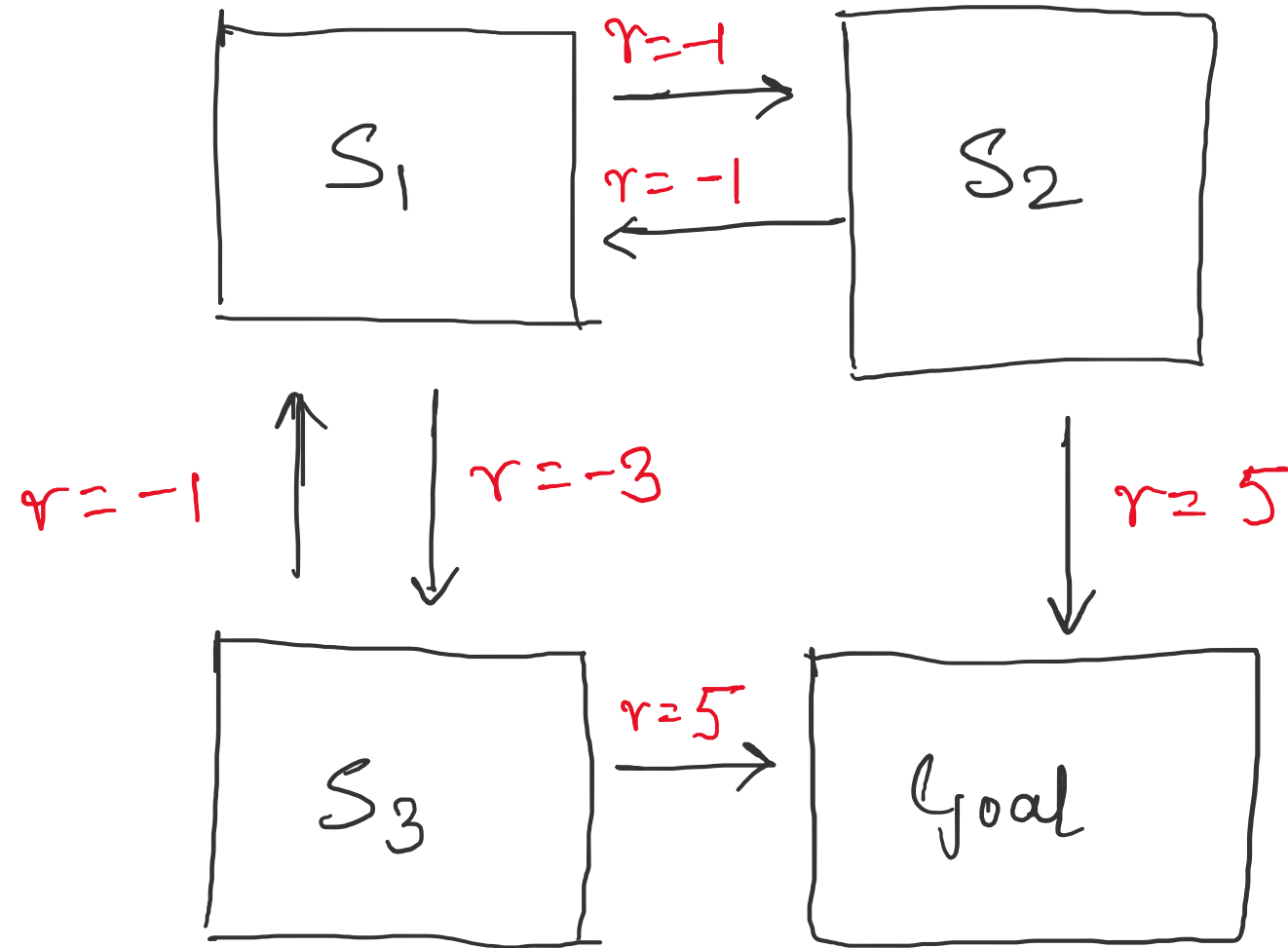
Agent moves.
But once reaches
goal state,
the objective is
accomplished!!

Grid World Example

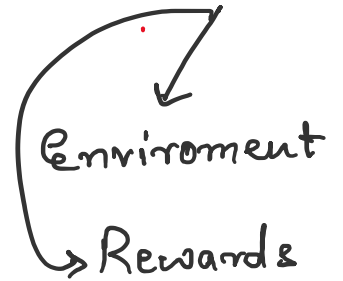


Once, Goal is reached,
No turning back!!

Grid World Example



Rewards are assigned.
Agent knows the MDP



What is the best way
to reach goal?
Optimal path?

Markov Decision Process (MDP)

Markov process

Markov Reward Process

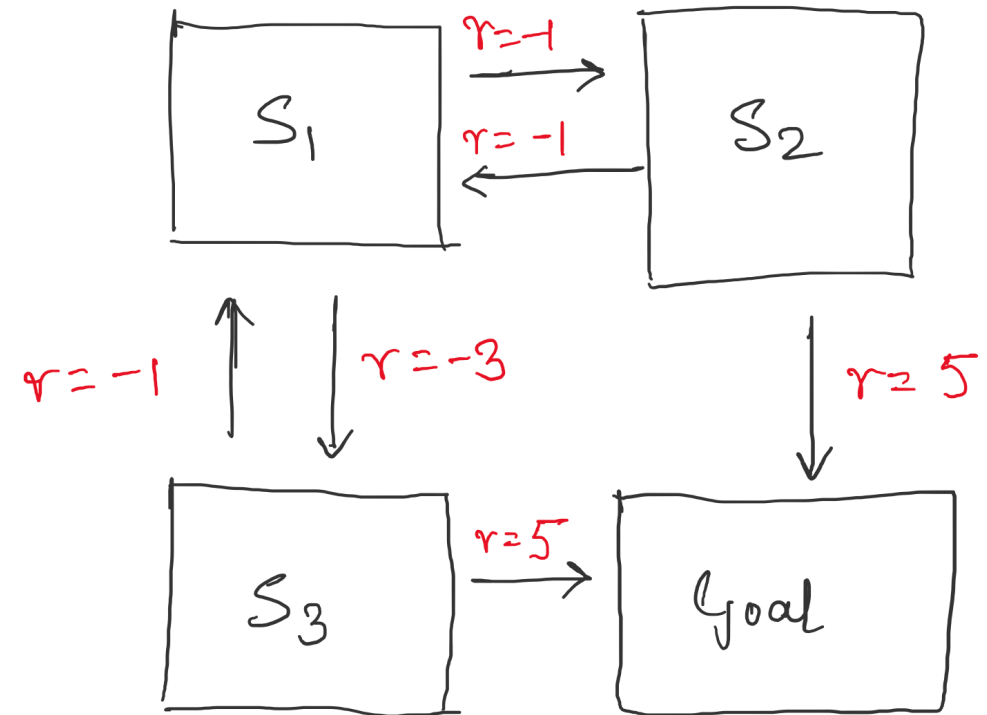
Markov Decision Process

Markov Decision Process (MDP)

An MDP characterizes the world/environment: $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$.

- $\mathcal{S}$: finite set of states
- $\mathcal{A}$: finite set of actions.
- $p(s, s') = P(S_{t+1} = s' \mid S_t = s, A_t = a)$.
- $r(s) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$.

All process $\rightarrow$ MDP! (Fully observable!)



Environment: Transition Function

An MDP characterizes the world/environment: $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$.

- $\mathcal{S}$: finite set of states
- $\mathcal{A}$: finite set of actions.
- $p(s, s') = P(S_{t+1} = s' \mid S_t = s, A_t = a)$.
- $r(s) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$.

(1) The probability of the next state.

(2) Given the current state and current action.

(3) Will be the same.

(4) As if you give it the entire history of interactions.

$$P(S_{t+1} | S_t, A_t) = P(S_{t+1} | S_t, A_t, S_{t-1}, A_{t-1}, \dots)$$

Environment: Transition Function

An MDP characterizes the world/environment: $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$.

- $\mathcal{S}$: finite set of states
- $\mathcal{A}$: finite set of actions.
- $p(s, s') = P(S_{t+1} = s' \mid S_t = s, A_t = a)$.
- $r(s) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$.

(1) The probability of the next state.

(2) Given the current state and current action.

(3) Will be the same.

(4) As if you give it the entire history of interactions.

$$P(S_{t+1} | S_t, A_t) = P(S_{t+1} | S_t, A_t, S_{t-1}, A_{t-1}, \dots)$$

(1) The transition function is defined.

(2) As the probability of transitioning to state s' at time step t .

(3) Given action a was selected on state s in the previous time step $t-1$.

(4) Given these are probabilities, we expect the sum of the probabilities across all possible next states to sum to 1.

$$p(s' | s, a) = P(S_t = s' | S_{t-1} = s, A_{t-1} = a)$$
$$\sum_{s' \in \mathcal{S}} p(s' | s, a) = 1, \forall s \in \mathcal{S}, \forall a \in A(s)$$

(5) That's true for all states s in the set of states $\mathcal{S}$, and all actions a in the set of actions available in state s .

Environment: Reward Function

An MDP characterizes the world/environment: $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$.

- $\mathcal{S}$: finite set of states
- $\mathcal{A}$: finite set of actions.
- $p(s, s') = P(S_{t+1} = s' \mid S_t = s, A_t = a)$.
- $r(s) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$.

(1) The reward function can be defined as follows.

(2) It can be defined as a function that takes in a state-action pair.

$$r(s, a) = \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a]$$

(3) And, it is the expectation of reward at time step t , given the state-action pair in the previous time step.

(4) But, it can also be defined as a function that takes a full transition tuple s, a, s' .

(5) And it is also defined as the expectation, but now given that transition tuple.

$$r(s, a, s') = \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s']$$

$$R_t \in \mathcal{R} \subset \mathbb{R}$$

(6) The reward at time step t comes from a set of all rewards $\mathcal{R}$, which is a subset of all real numbers.

Agent:

- Policy
- Value function

Agent: Policy

A (stationary) policy π is a distribution over actions given states:

$$\pi(a \mid s) = P(A_t = a \mid S_t = s).$$

Agent: Policy

A (stationary) policy π is a distribution over actions given states:

$$\pi(a \mid s) = P(A_t = a \mid S_t = s).$$

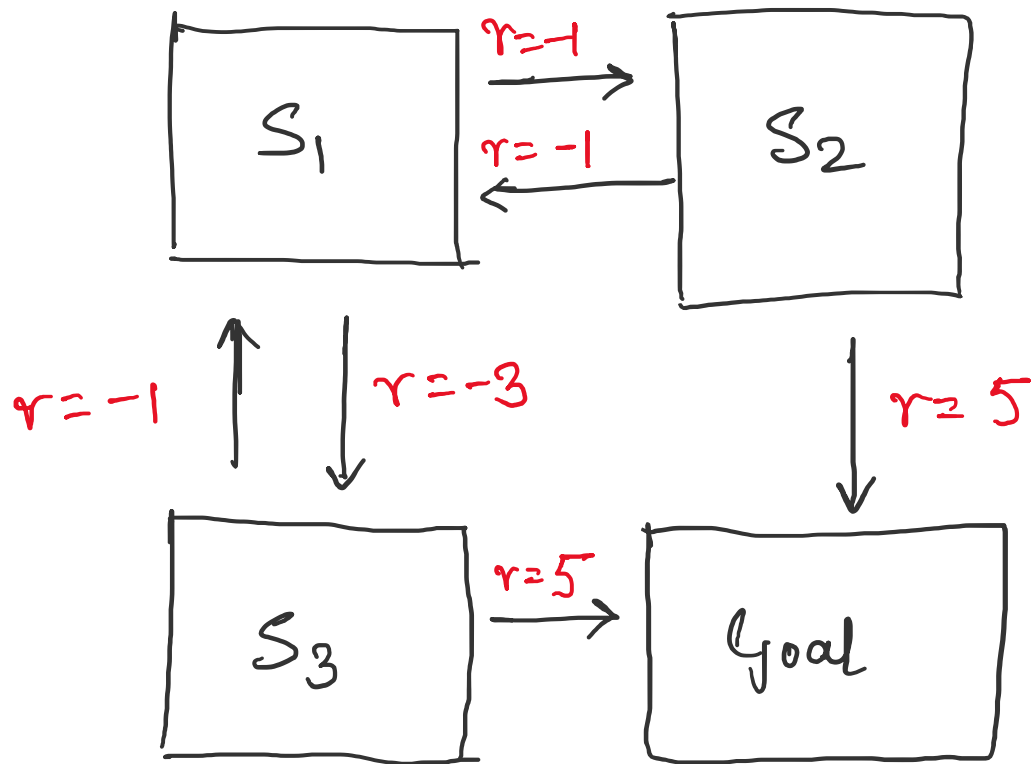
Deterministic Policy:

$$a = \pi(s)$$

Stochastic Policy:

$$\pi(a \mid s) = P(A_t = a \mid S_t = s).$$

Grid World Example



Policy Example: π

Actions are drawn from a uniform distribution.

$$\pi(\text{right} | s_1) = \frac{1}{2}$$

$$\pi(\text{down} | s_1) = \frac{1}{2}$$

$$\pi(\text{left} | s_2) = \frac{1}{2}$$

$$\pi(\text{down} | s_2) = \frac{1}{2}$$

$$\pi(\text{up} | s_2) = \frac{1}{2}$$

$$\pi(\text{right} | s_2) = \frac{1}{2}$$

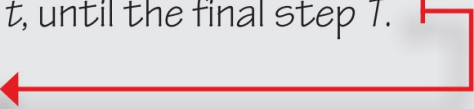
Actions are based on policy.

Policy depends on some probability

Agent: Value Function

- Return

(1) The return is the sum of rewards encounter from step t , until the final step T .

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T$$


Agent: Value Function

- Return

(1) The return is the sum of rewards encountered from step t , until the final step T .

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T$$

discount factor, gamma. This is then the *discounted* return, which prioritizes early rewards.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T$$

(3) We can simplify the equation and have a more general equation, such as this one.

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

$$G_t = R_{t+1} + \gamma G_{t+1}$$

(4) And stare at this recursive definition of G for a while.

Agent: Value Function

- Return

(1) The return is the sum of rewards encounter from step t , until the final step T .

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T$$

discount factor, gamma. This is then the *discounted* return, which prioritizes early rewards.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T$$

(3) We can simplify the equation and have a more general equation, such as this one.

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

$$G_t = R_{t+1} + \gamma G_{t+1}$$

(4) And stare at this recursive definition of G for a while.

- Value Function

(1) The value of a state s .

(3) Is the expectation over π .

(2) Under policy π .

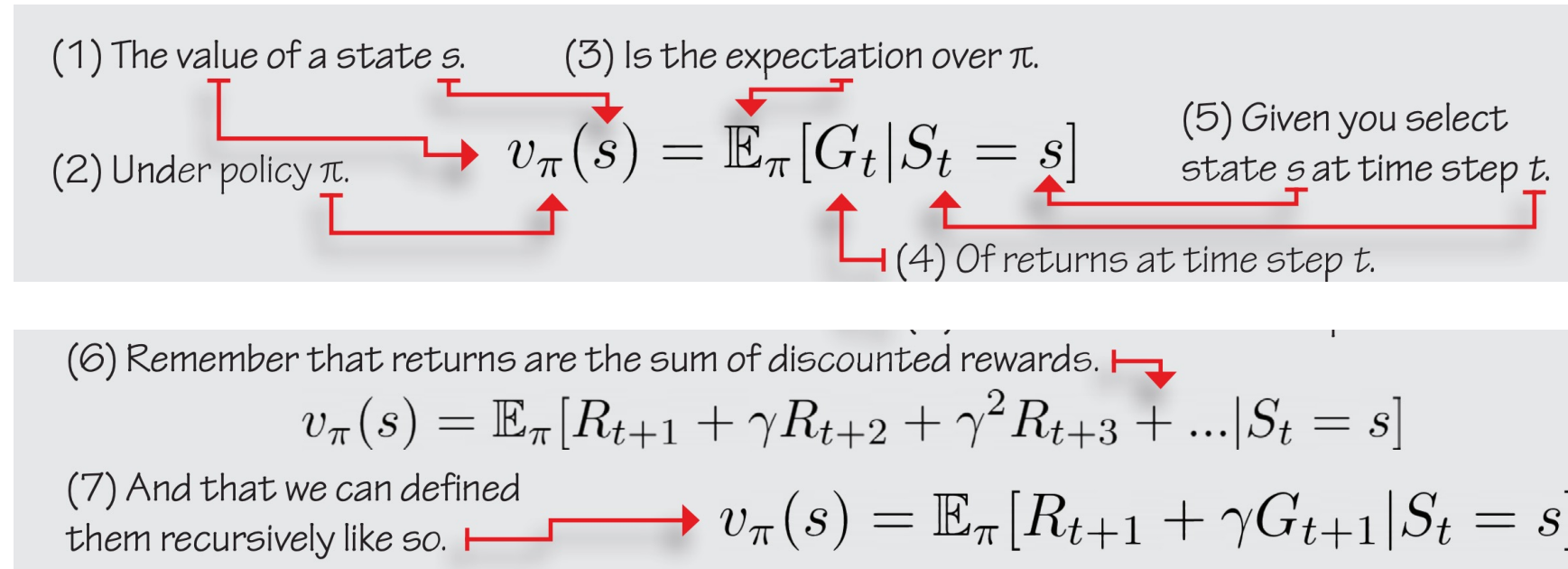
$$v_{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s]$$

(5) Given you select state s at time step t .

(4) Of returns at time step t .

Agent: Value Function

- Value function



Value Function:

Assessment of “Importance” of being in a state.

Helps to make a choice in taking actions while being in a state.

Dynamic Programming

Value Function

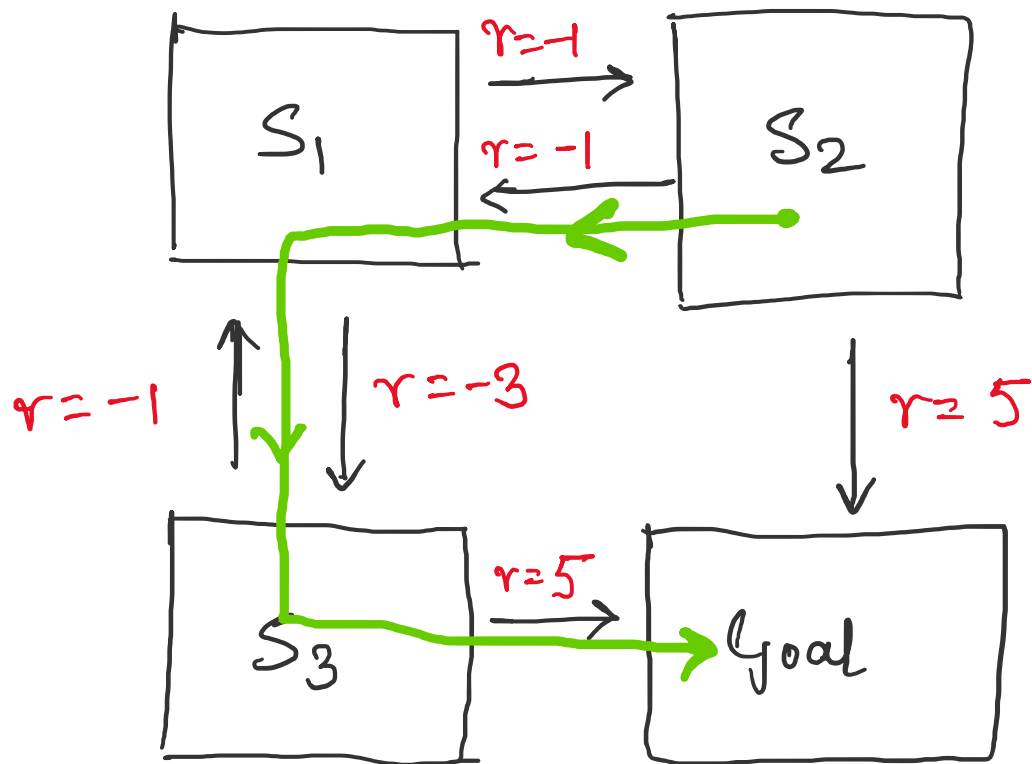
$$v(s) = \mathbb{E} [R_{t+1} + \gamma \cdot v(s_{t+1}) \mid s_t = s]$$

↑
Reward
at $t+1$.

for any $s \in S$
↓
Value for any $s \in S$ at time t

(This is the Iterative Equation)

Grid World Example



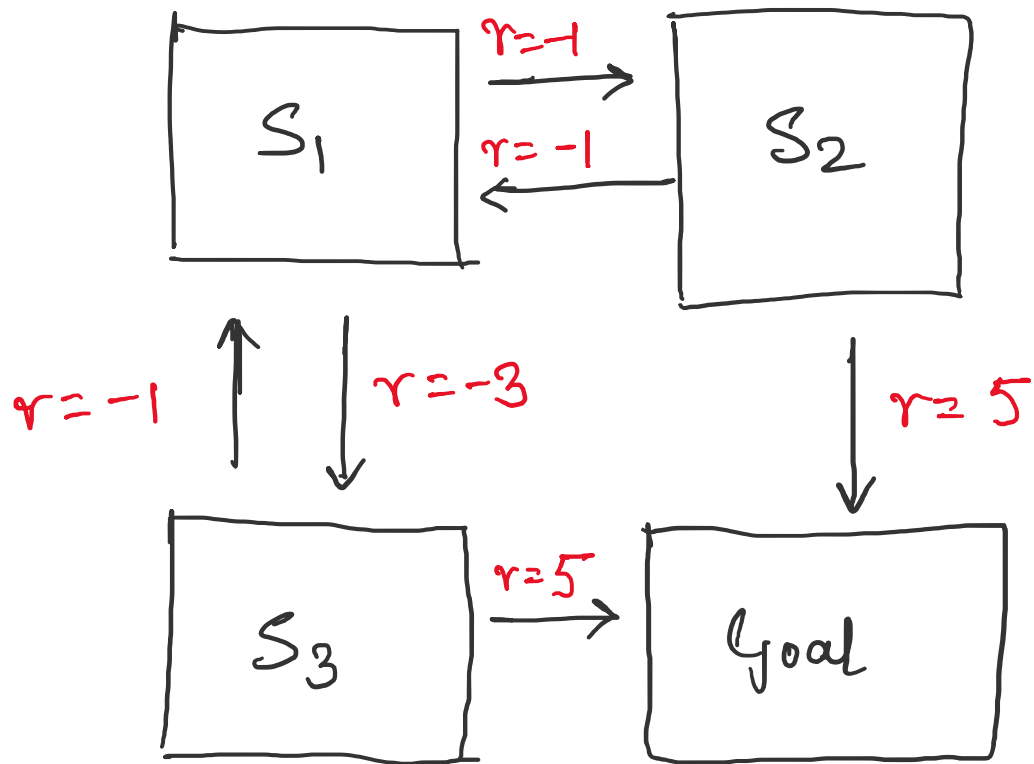
$$V_{\pi}(s) = \mathbb{E}_{\pi} [\underbrace{R_{t+1} + R_{t+2} + R_{t+3} + \dots}_{\text{future rewards}} \mid S_t = s]$$

Assume: A movement from S_2 as:

$$\begin{aligned} V_{\pi}(S_{2,t}) &= \mathbb{E}_{\pi} (-1 + -3 + 5 \mid S_t = S_2) \\ &= (-1 + -3 + 5) = 1 \end{aligned}$$

→ we consider only one π
if, $\pi = \{ \pi_1, \pi_2, \pi_3, \dots \}$
then average over all π .

Grid World Example



Given π ,
determine $V_\pi(s) \forall s \in S$ → for all

Policy Example: π

Actions are drawn from a uniform distribution.

$$\pi(\text{right} | s_1) = \frac{1}{2}$$

$$\pi(\text{down} | s_1) = \frac{1}{2}$$

$$\pi(\text{left} | s_2) = \frac{1}{2}$$

$$\pi(\text{down} | s_2) = \frac{1}{2}$$

$$\pi(\text{up} | s_2) = \frac{1}{2}$$

$$\pi(\text{right} | s_2) = \frac{1}{2}$$

Actions are based on policy.

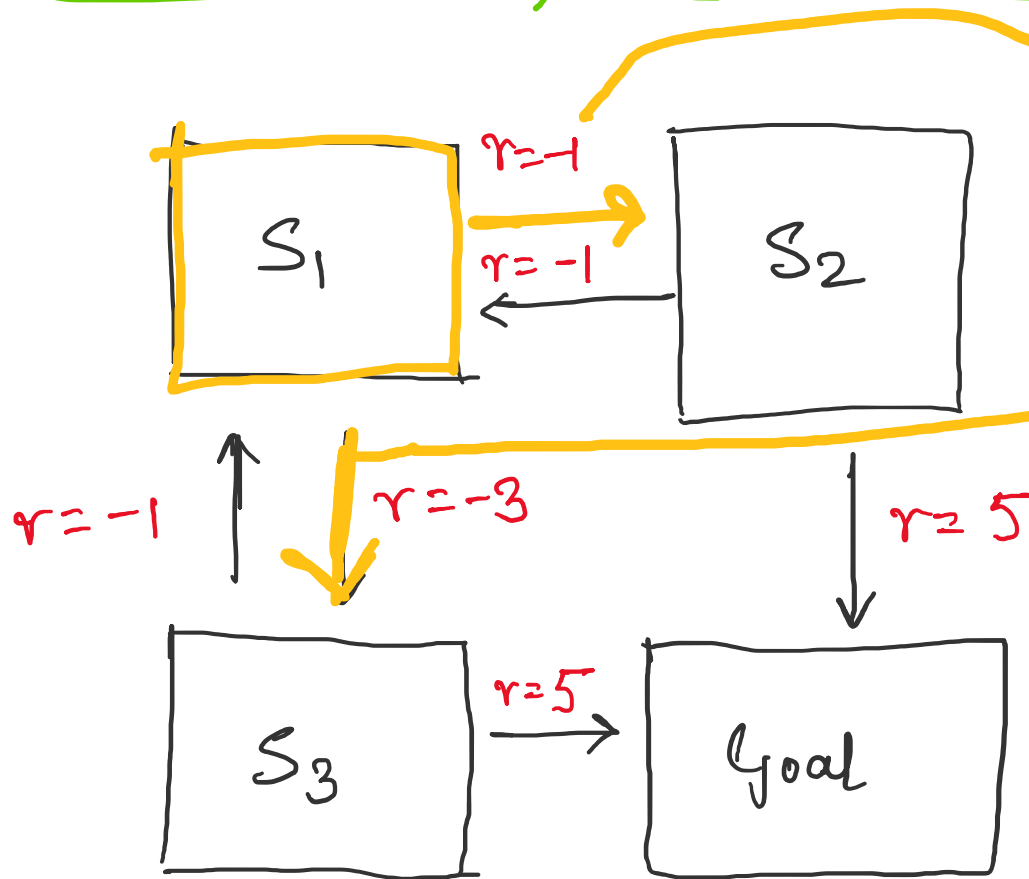
Policy depends on some probability

Grid World Example

Half time

other half
time

$$V_{\pi}(s_1) = \frac{1}{2} (-1 + V_{\pi}(s_2)) + \frac{1}{2} (-3 + V_{\pi}(s_3))$$

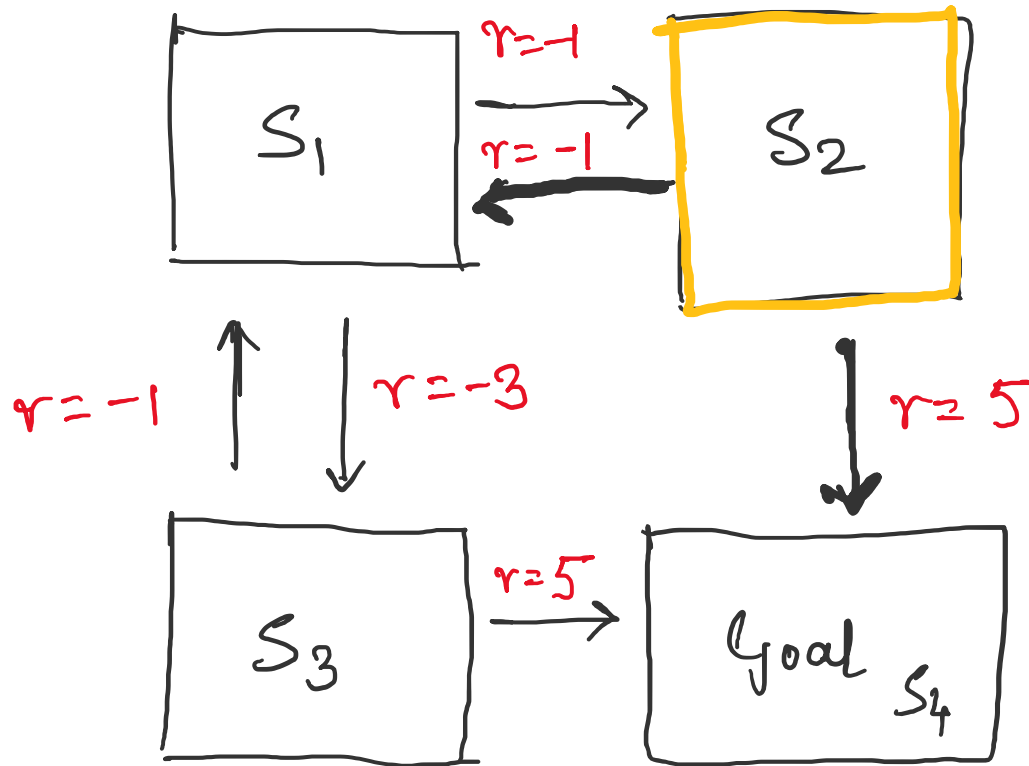


Given π , determine $V_{\pi}(s) \forall s \in S$ → Random Uniform Policy.

Grid World Example

Half time

other half
time



$$v_{\pi}(s_1) = \frac{1}{2} (-1 + v_{\pi}(s_2)) + \frac{1}{2} (-3 + v_{\pi}(s_3))$$

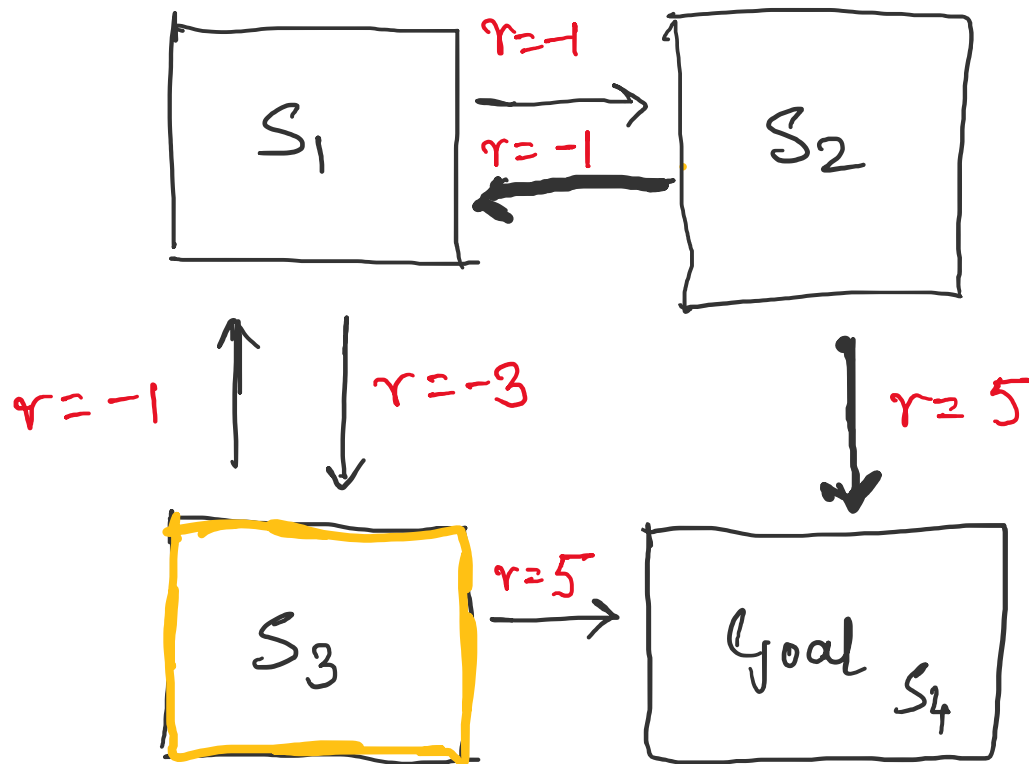
$$v_{\pi}(s_2) = \frac{1}{2} (-1 + v_{\pi}(s_1)) + \frac{1}{2} (5 + v_{\pi}(s_4))$$

Given π ,
determine $v_{\pi}(s) \forall s \in S$

Grid World Example

Half time

other half
time



$$V_{\pi}(S_1) = \frac{1}{2} (-1 + V_{\pi}(S_2)) + \frac{1}{2} (-3 + V_{\pi}(S_3))$$

$$V_{\pi}(S_2) = \frac{1}{2} (-1 + V_{\pi}(S_1)) + \frac{1}{2} (5 + V_{\pi}(S_4))$$

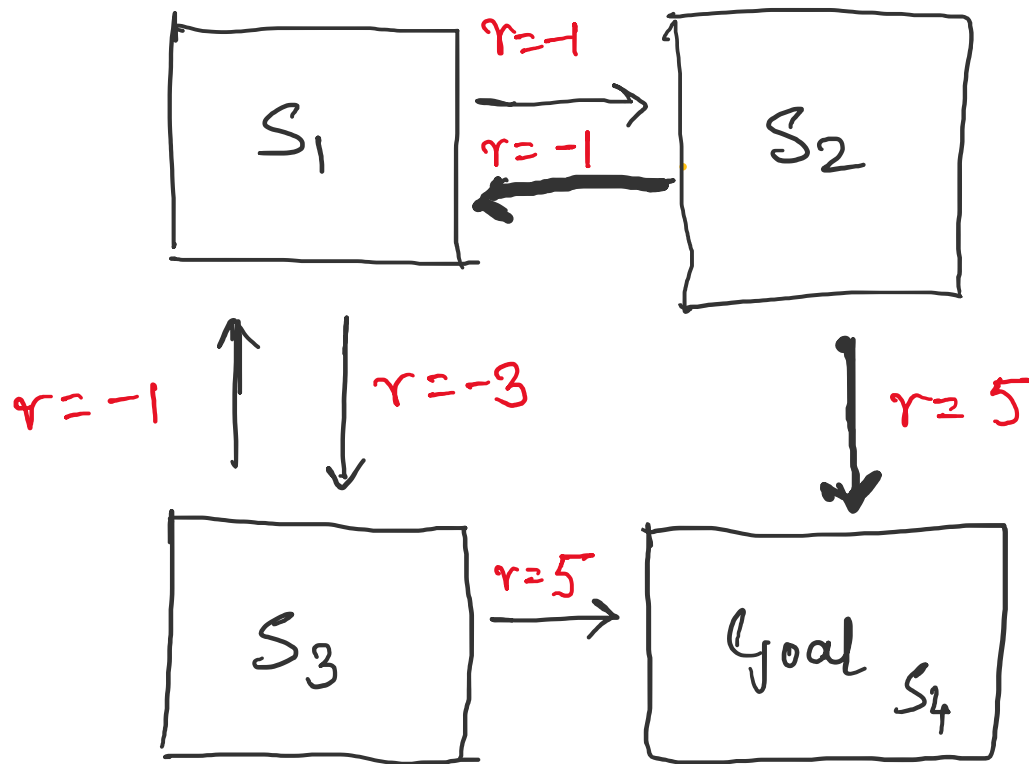
$$V_{\pi}(S_3) = \frac{1}{2} (-1 + V_{\pi}(S_1)) + \frac{1}{2} (5 + V_{\pi}(S_4))$$

Given π ,
determine $V_{\pi}(s) \forall s \in S$

Grid World Example

Half time

other half
time



$$V_{\pi}(s_1) = \frac{1}{2} (-1 + V_{\pi}(s_2)) + \frac{1}{2} (-3 + V_{\pi}(s_3))$$

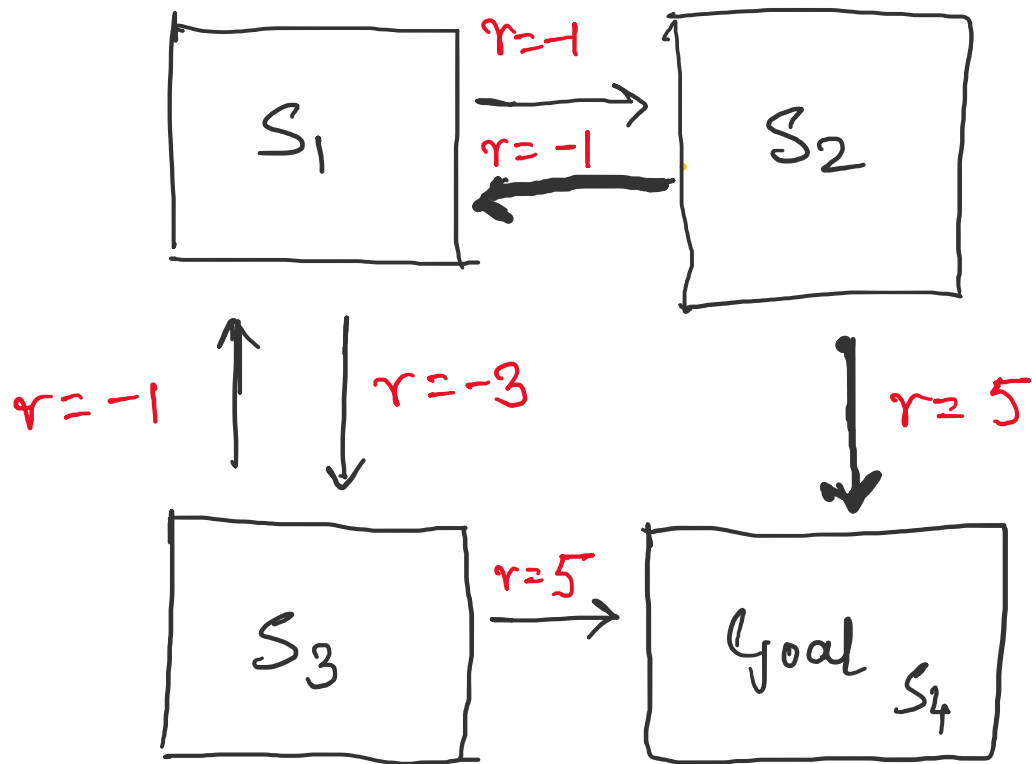
$$V_{\pi}(s_2) = \frac{1}{2} (-1 + V_{\pi}(s_1)) + \frac{1}{2} (5 + V_{\pi}(s_4))$$

$$V_{\pi}(s_3) = \frac{1}{2} (-1 + V_{\pi}(s_1)) + \frac{1}{2} (5 + V_{\pi}(s_4))$$

$$V_{\pi}(s_4) = 0$$

Given π ,
determine $V_{\pi}(s) \forall s \in S$

Grid World Example



Given π ,
determine $V_{\pi}(s) \forall s \in S$

$$V_{\pi}(s_1) = \frac{1}{2} (-1 + V_{\pi}(s_2)) + \frac{1}{2} (-3 + V_{\pi}(s_3))$$

$$V_{\pi}(s_2) = \frac{1}{2} (-1 + V_{\pi}(s_1)) + \frac{1}{2} (5 + V_{\pi}(s_4))$$

$$V_{\pi}(s_3) = \frac{1}{2} (-1 + V_{\pi}(s_1)) + \frac{1}{2} (5 + V_{\pi}(s_4))$$

$$V_{\pi}(s_4) = 0$$

4 equations, 4 unknown variables

Can be solved to give

$$V_{\pi}(s_1) = 0, V_{\pi}(s_4) = 0$$

$$V_{\pi}(s_2) = 2$$

$$V_{\pi}(s_3) = 2$$

Bellman Equation

$$v = R + \gamma \mathbf{P}v \quad (\text{using matrices})$$
$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix} + \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & & \\ P_{n1} & \cdots & P_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

Bellman Equation

$$v = R + \gamma \mathbf{P}v \quad (\text{using matrices})$$

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix} + \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & & \\ P_{n1} & \cdots & P_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

$$v(s) = \mathbb{E} [R_{t+1} + \gamma \cdot v(s_{t+1}) \mid s_t = s]$$

applied over all states and policies!!

Bellman Equation

$$v = R + \gamma \mathbf{P}v \quad (\text{using matrices})$$

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix} + \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & & \\ P_{n1} & \cdots & P_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

$$V_\pi(s) = \sum_{a \in A} \pi(a | s) \left(R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') V_\pi(s') \right)$$

Bellman Equation

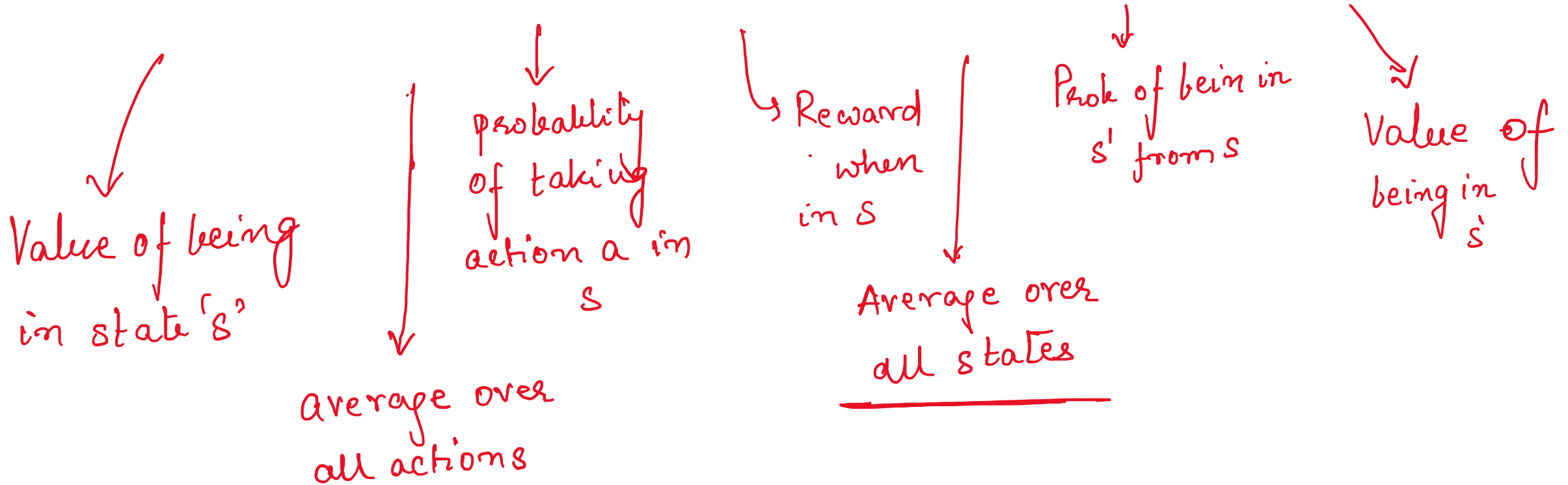
$$v = R + \gamma \mathbf{P}v \quad (\text{using matrices})$$

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} R_1 \\ \vdots \\ R_n \end{bmatrix} + \begin{bmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & & \\ P_{n1} & \cdots & P_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

$$V_\pi(s) = \sum_{a \in A} \pi(a | s) \left(R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') V_\pi(s') \right)$$

Bellman Equation

$$V_{\pi}(s) = \sum_{a \in A} \pi(a | s) \left(R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') V_{\pi}(s') \right)$$



Bellman Equation

$$V_{\pi}(s) = \sum_{a \in A} \pi(a | s) \left(R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') V_{\pi}(s') \right)$$

(8) This equation is called the Bellman equation and it tells us how to find the value of states.

$$v_{\pi}(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v_{\pi}(s')], \forall s \in S$$

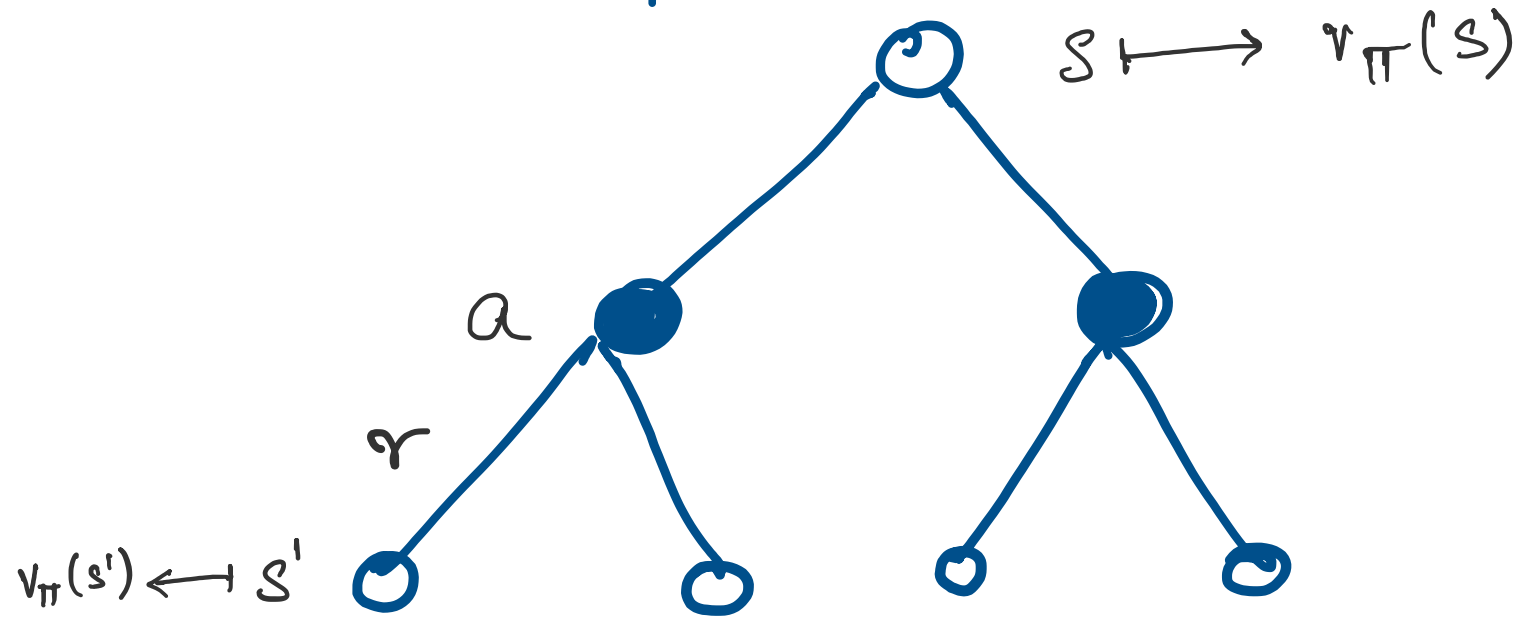
(9) We get the action (or actions, if the policy is stochastic) prescribed for state s . And do a weighted sum...

(10) We also weight the sum over the probability of next states and rewards.

(11) We add the reward and the discounted value of the landing state, then weight that by the probability of that transition occurring.

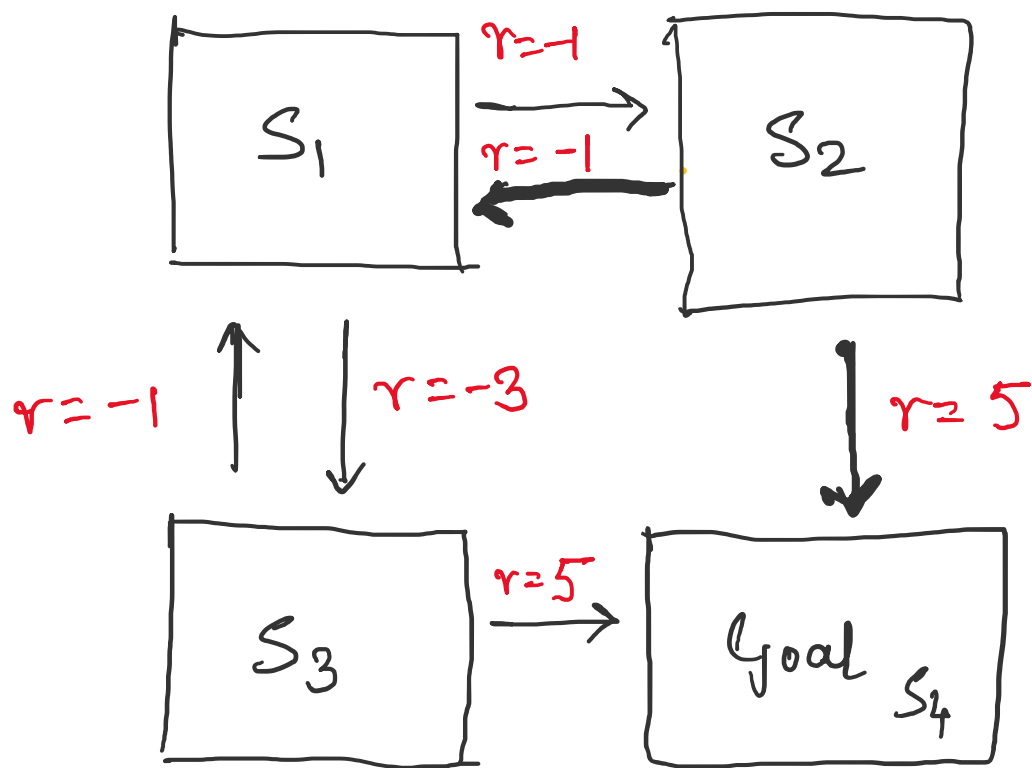
(12) Do this for all states in the state space.

Bellman Equation (Expectation) for v_π



$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') v_\pi(s') \right)$$

Grid World Example



Given π ,
determine $V_\pi(s) \forall s \in S$

$$V_\pi(s_1) = \frac{1}{2} (-1 + V_\pi(s_2)) + \frac{1}{2} (-3 + V_\pi(s_3))$$

$$V_\pi(s_2) = \frac{1}{2} (-1 + V_\pi(s_1)) + \frac{1}{2} (5 + V_\pi(s_4))$$

$$V_\pi(s_3) = \frac{1}{2} (-1 + V_\pi(s_1)) + \frac{1}{2} (5 + V_\pi(s_4))$$

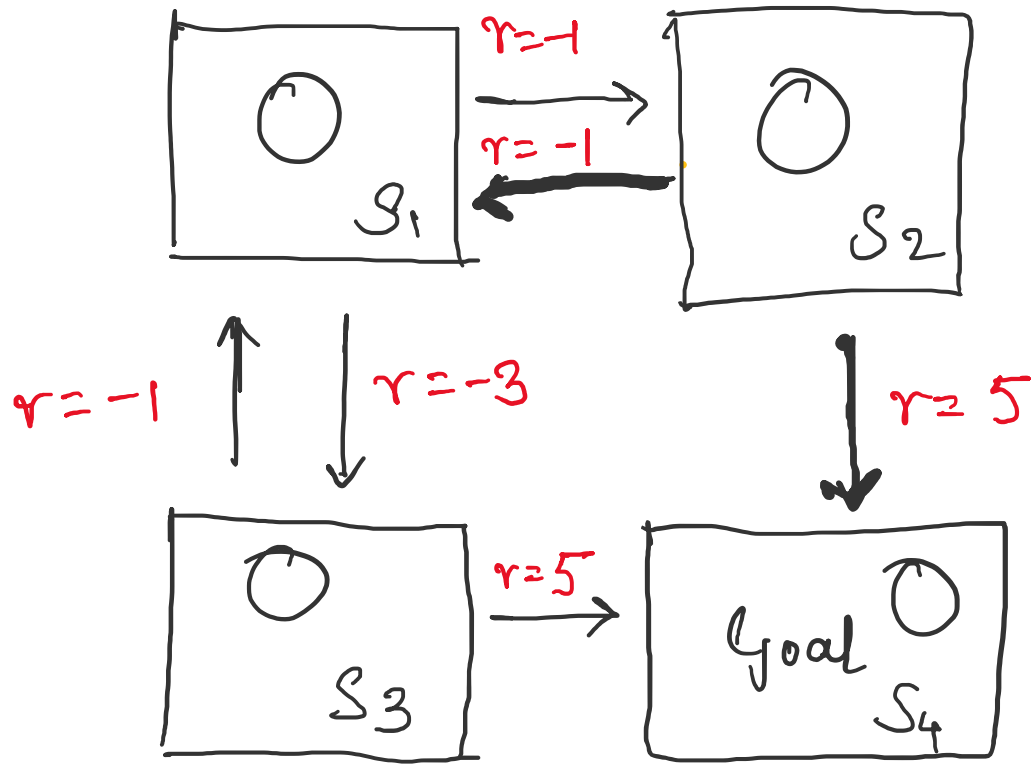
$$V_\pi(s_4) = 0$$

How to handle large state space?

How to optimise under several policies / large states??

Iterative Approach.

Grid World Example (Iterative Approach)



Given π ,
determine $V_\pi(s) \forall s \in S$

→ start with a guess
 $\{S_1=0, S_2=0, S_3=0, S_4=0\}$

→ Consider one state S_1

→ Improve the guess with respect to that state

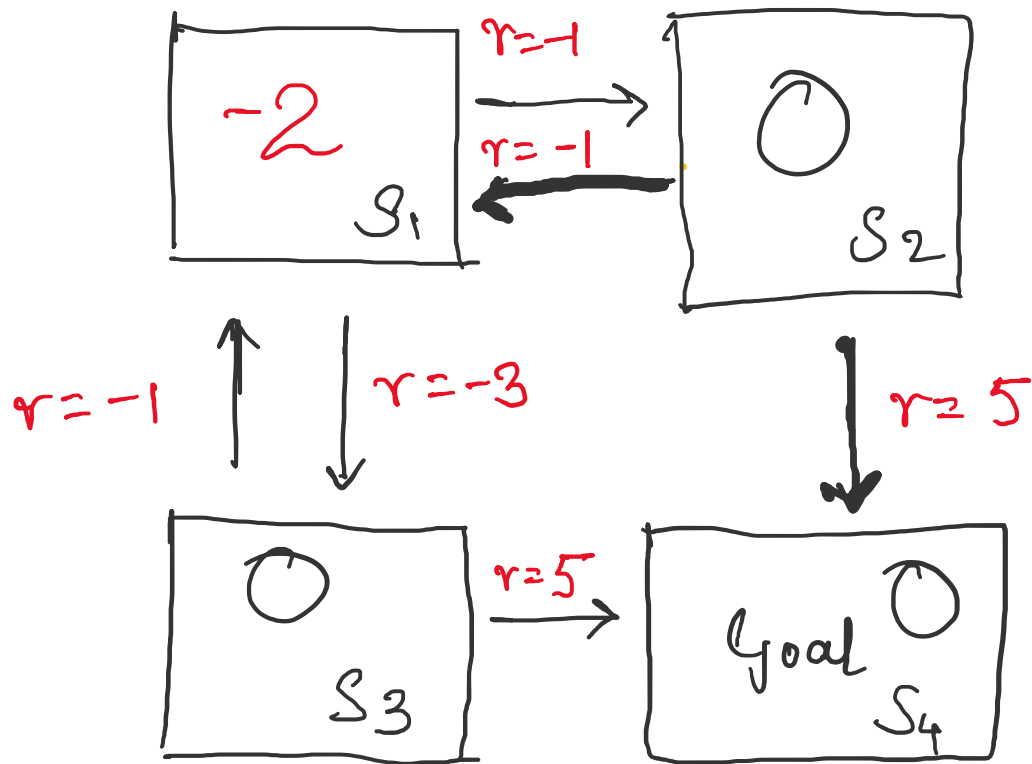
$$V_\pi(S_1) = \frac{1}{2}(-1 + V_\pi(S_2)) + \frac{1}{2}(-3 + V_\pi(S_3))$$

Use this to update the guess $V_\pi(S_1)$

update guess with a guess

$$V_\pi(S_1) \leftarrow \frac{1}{2}(-1 + 0) + \frac{1}{2}(-3 + 0) = -2$$

Grid World Example (Iterative Approach)

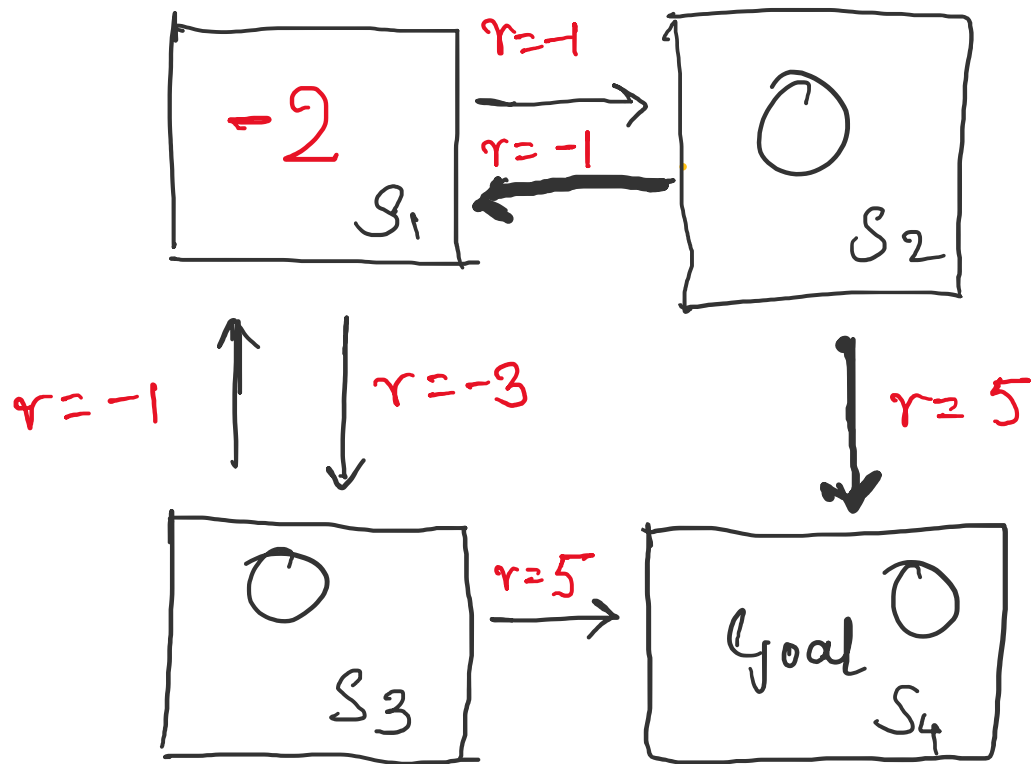


Use $V_\pi(s_1)$ to update value of s_2

$$V_\pi(s_2) \leftarrow \frac{1}{2}(-1 + V_\pi(s_1)) + \frac{1}{2}(5)$$

Given π ,
determine $V_\pi(s) \forall s \in S$

Grid World Example (Iterative Approach)



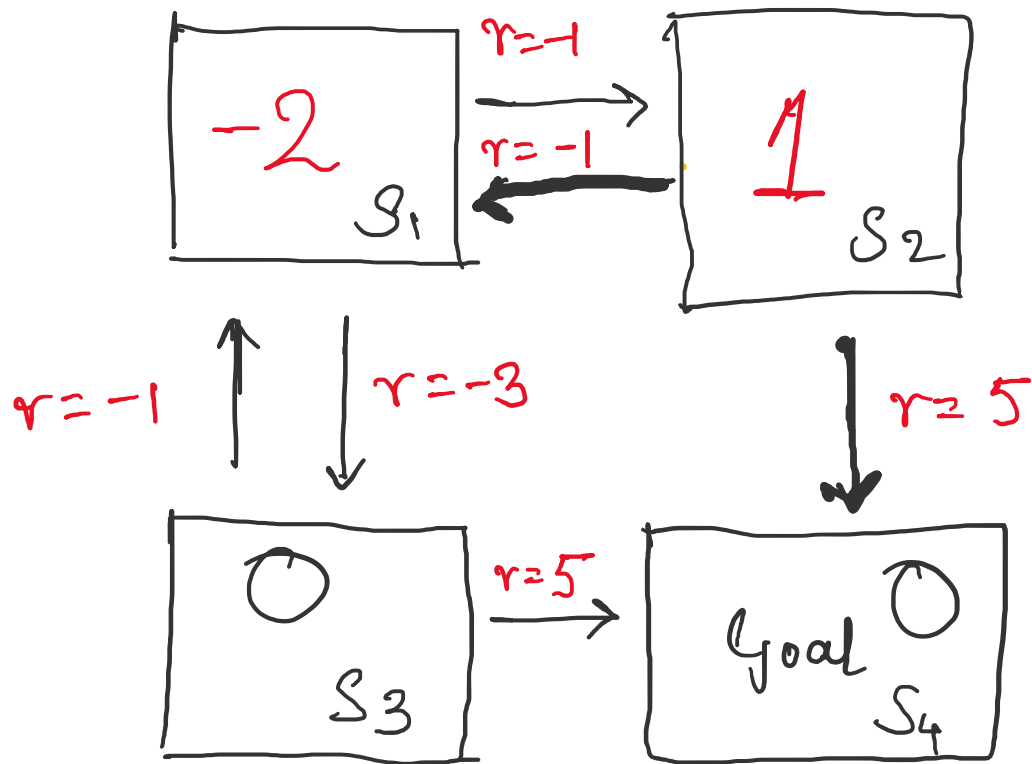
Use $V_\pi(s_1)$ to update value of s_2

$$V_\pi(s_2) \leftarrow \frac{1}{2}(-1 + V_\pi(s_1)) + \frac{1}{2}(5)$$

$$V_\pi(s_2) \leftarrow 1$$

Given π ,
determine $V_\pi(s) \forall s \in S$

Grid World Example (Iterative Approach)



Given π ,
determine $V_\pi(s) \forall s \in S$

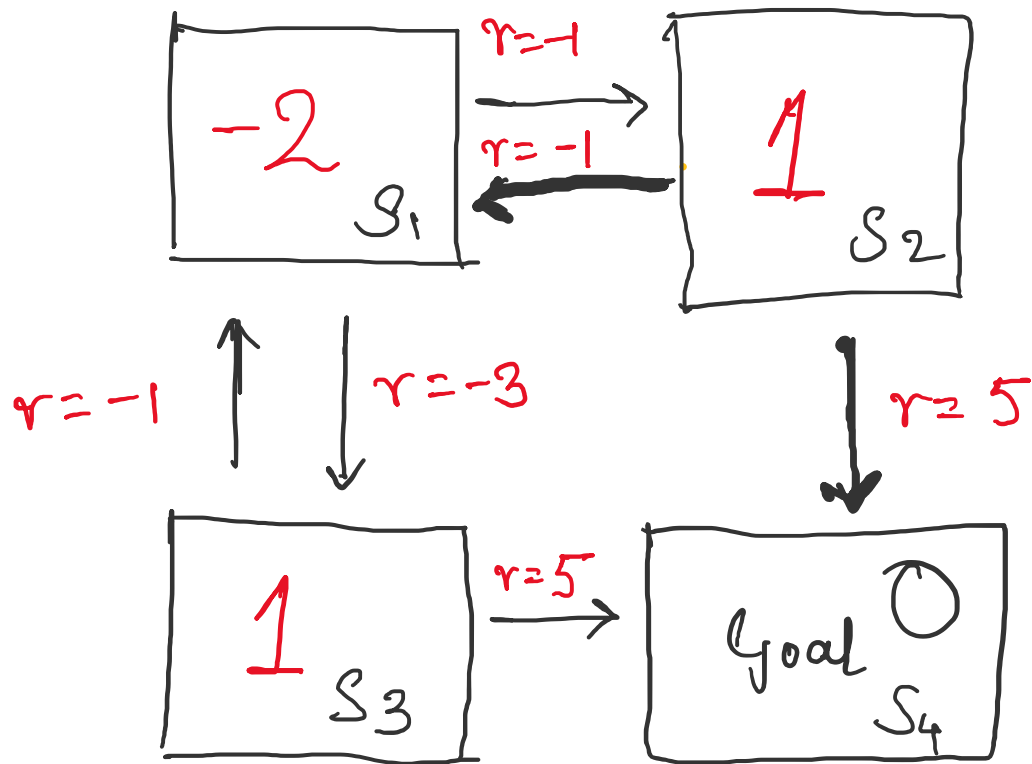
Use $V_\pi(s_1)$ to update value of s_2

$$V_\pi(s_2) \leftarrow \frac{1}{2}(-1 + V_\pi(s_1)) + \frac{1}{2}(5)$$

$V_\pi(s_2) \leftarrow 1$ record
keep track and update $V_\pi(s_3)$

$$\begin{aligned} V_\pi(s_3) &\leftarrow \frac{1}{2}(-1 + V_\pi(s_1)) + \frac{1}{2}(5) \\ &= \frac{1}{2}(-1 + -2) + \frac{1}{2}(5) = 1 \end{aligned}$$

Grid World Example (Iterative Approach)



Given π ,
determine $V_\pi(s) \forall s \in S$

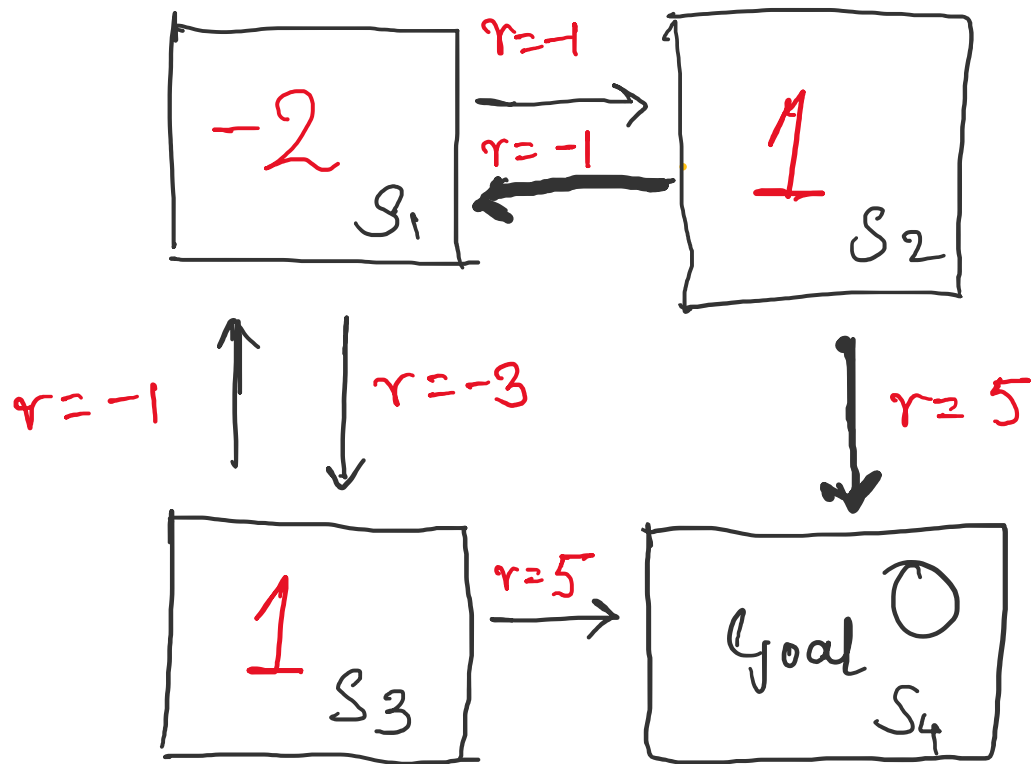
Use $V_\pi(s_1)$ to update value of s_2

$$V_\pi(s_2) \leftarrow \frac{1}{2}(-1 + V_\pi(s_1)) + \frac{1}{2}(5)$$

$V_\pi(s_2) \leftarrow \underline{1}$ record
keep track and update $V_\pi(s_3)$

$$\begin{aligned} V_\pi(s_3) &\leftarrow \frac{1}{2}(-1 + V_\pi(s_1)) + \frac{1}{2}(5) \\ &= \frac{1}{2}(-1 + -2) + \frac{1}{2}(5) = \underline{1} \end{aligned}$$

Grid World Example (Iterative Approach)



Given π ,
determine $V_\pi(s) \forall s \in S$

- Repeat this procedure
- Visit the states over and over
- Converge to True values of states!

Idea behind → Iterative Policy Evaluation

Dynamic Programming :

Iterative Policy Evaluation

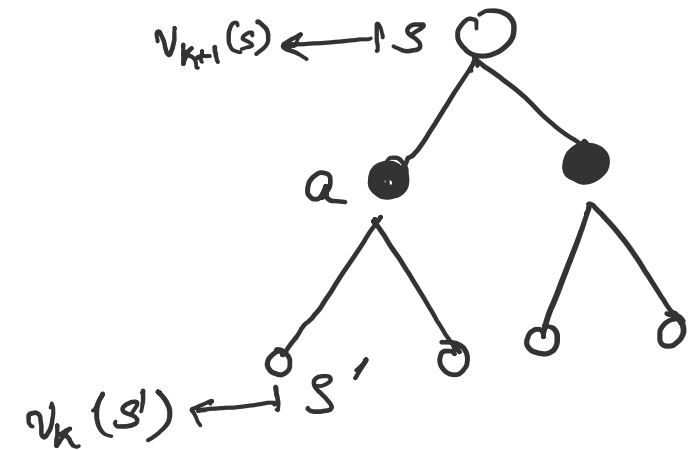
Iterative Policy Evaluation

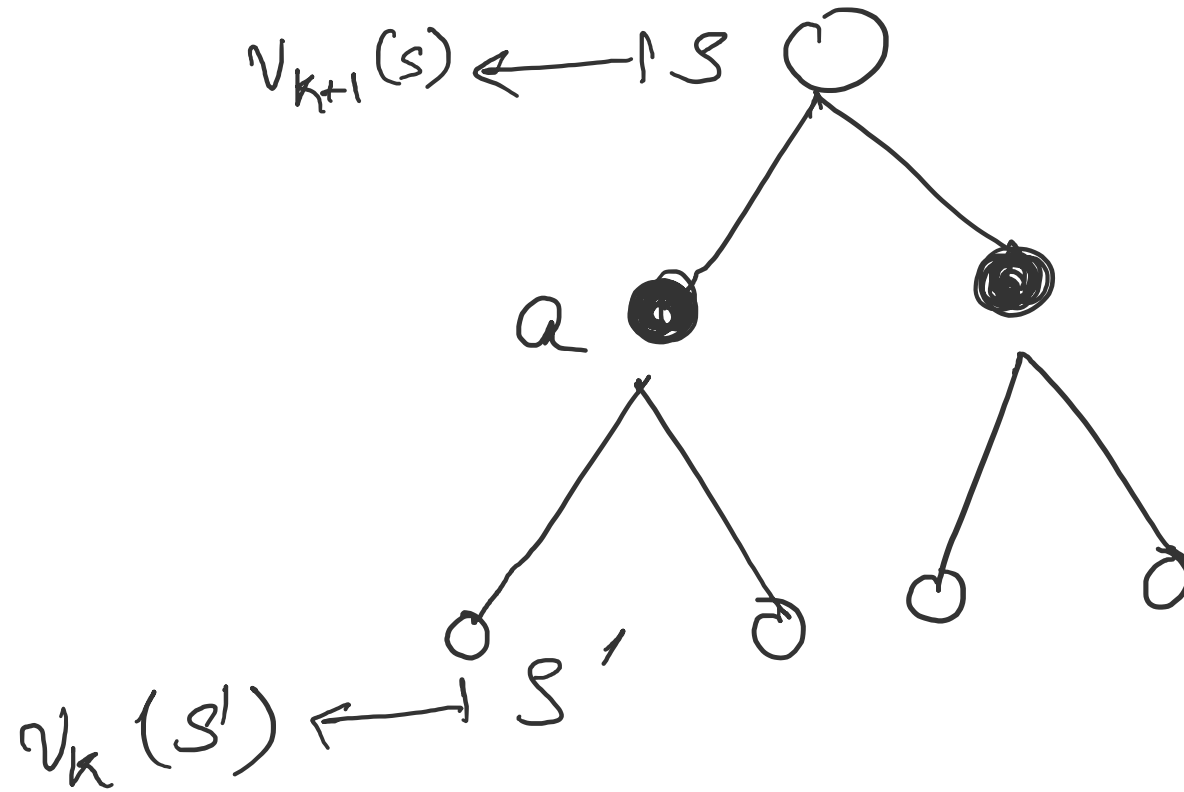
→ Assumes Agent knows MDP, rewards that characterise environment.

Algorithm: to evaluate a given policy π

(iterative application of Bellman expectation Backup)

- $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_\pi$
- Use synchronous back-ups (storage)
 - At iteration $k+1$
 - For all states $s \in S$
 - Update $v_{k+1}(s)$ from $v_k(s')$





$$\underline{V}_{k+1}(\bar{s}) = \sum_{a \in A(s)} \pi(a|s) \sum_{s' \in S, r \in R} p(s', r | s, a) (\gamma + \gamma \underline{V}_k(\underline{s'}))$$

for all $s \in S$.

Iterative Policy Evaluation

Iterative algorithm adapts the Bellman Eq to be used as update rule:

$$V(s) \leftarrow \sum_{a \in A} \pi(a/s) \sum_{s' \in S, r \in R} P(s', r/s, a) (r + \gamma V(s'))$$

↑
Value function

↑
Value function.

Iterative Policy Evaluation

Iterative algorithm

Input: MDP, policy π

Output: V-function (V_π)

$V(s) \leftarrow 0$ for all $s \in S$

repeat:
for $s \in S$

$$V(s) \leftarrow \sum_{a \in A} \pi(a/s) \sum_{s' \in S, r \in R} p(s', r/s, a) (r + \gamma V(s'))$$

Iterative Policy Evaluation

Iterative algorithm

Input: MDP, policy π

Output: V-function (V_π)

$V(s) \leftarrow 0$ for all $s \in S$ | begin with initial guess

repeat:

for $s \in S$ | loop over all states

| update the values

$$V(s) \leftarrow \sum_{a \in A} \pi(a/s) \sum_{s' \in S, r \in R} p(s', r/s, a) (r + \gamma V(s'))$$

If value fn $\rightarrow$ contraction mapping, state $\rightarrow$ compact set
guaranteed to converge to V_π .

Iterative Policy Evaluation

- Guaranteed to converge to V_{π} in infinite sense.
- But how to assess if convergence is reached in approximate sense?
- Introduce a θ

Iterative Policy Evaluation

Input: MDP, policy π , small positive θ

Output: V-function (V_π)

$V(s) \leftarrow 0$ for all $s \in S$

| begin with initial guess

repeat untill $\Delta < \theta$:

$\Delta \leftarrow 0$

for $s \in S$ | loop over all states

$v \leftarrow V(s)$

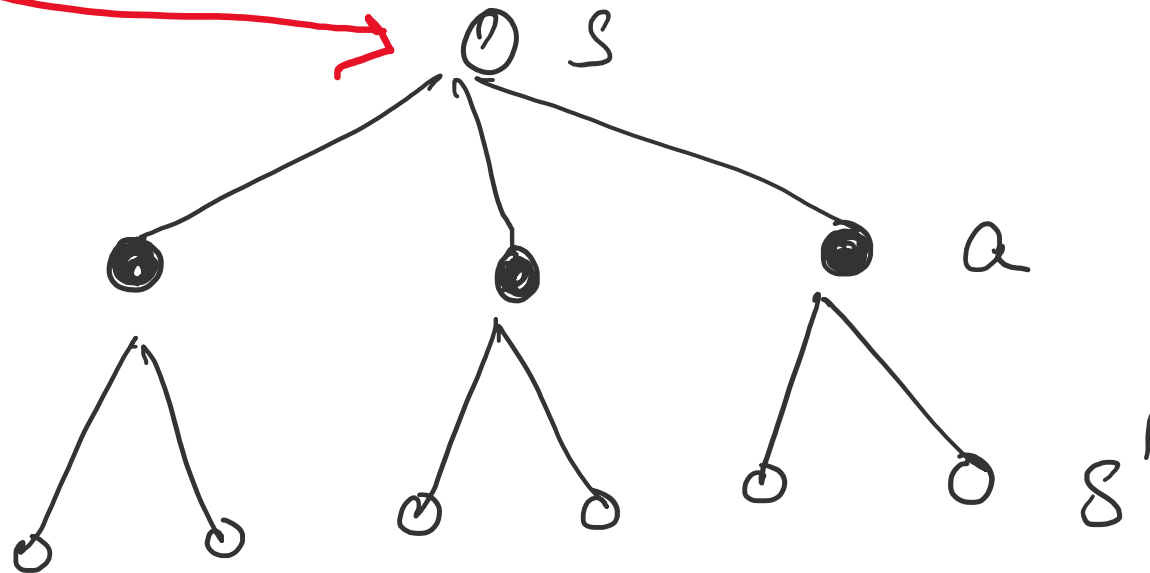
$$V(s) \leftarrow \sum_{a \in A} \pi(a/s) \sum_{s' \in S, r \in R} p(s', r/s, a) (r + \gamma V(s'))$$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

| how much values of each state changed
if not changed, stop.

return V

$$V(s) \leftarrow \sum_{a \in A} \pi(a/s) \sum_{s' \in S, r \in R} p(s', r/s, a) (r + \gamma V(s'))$$



Iterative Policy Evaluation

Policy π



Value function V_π

We have V_π



But, optimal V -function
still not found!!

Policy Improvement

How to improve a Policy

Policy Evaluation

policy π



value function
 v_π

Policy Improvement

value function v_π



policy π' (with $\pi' \geq \pi$)

policy π



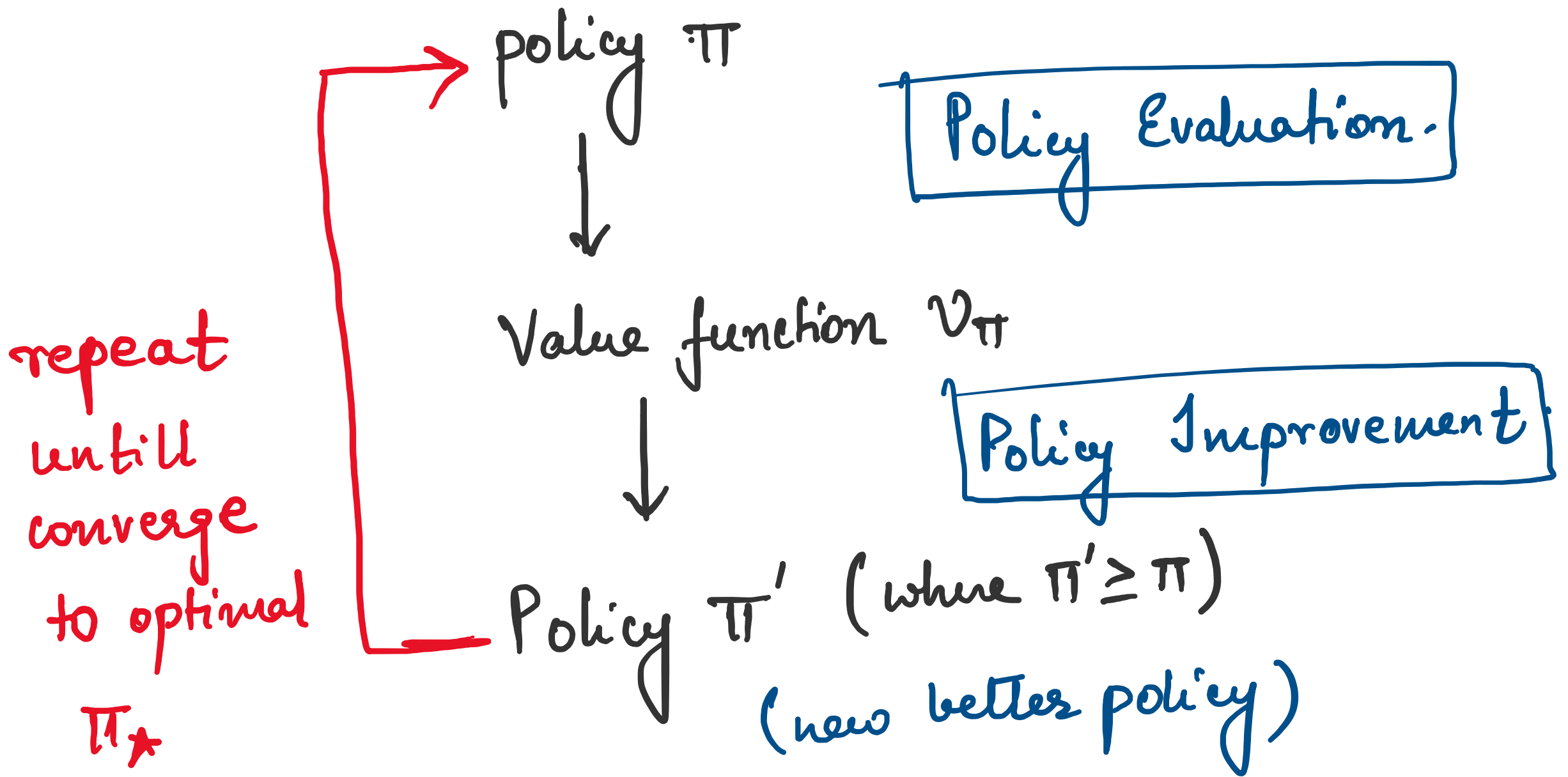
Value function V_π

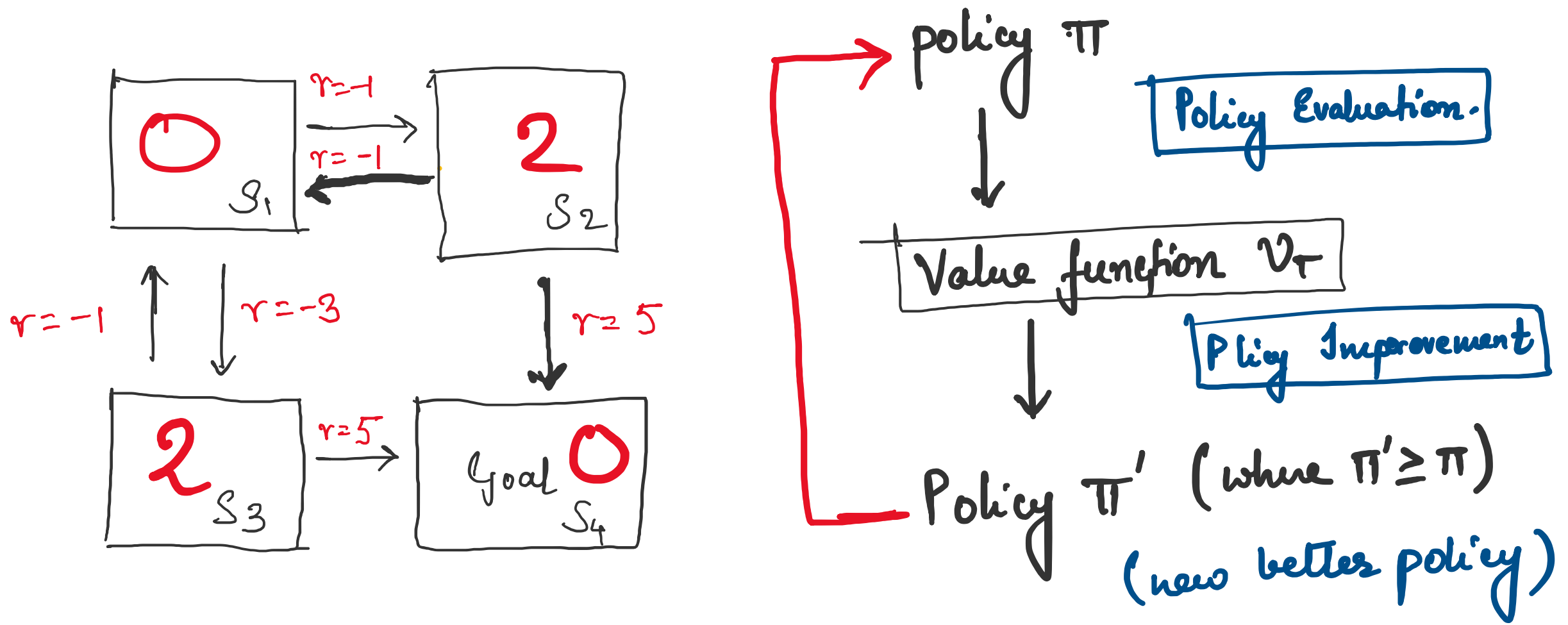


Policy π' (where $\pi' \geq \pi$)
(new better policy)

Policy Evaluation

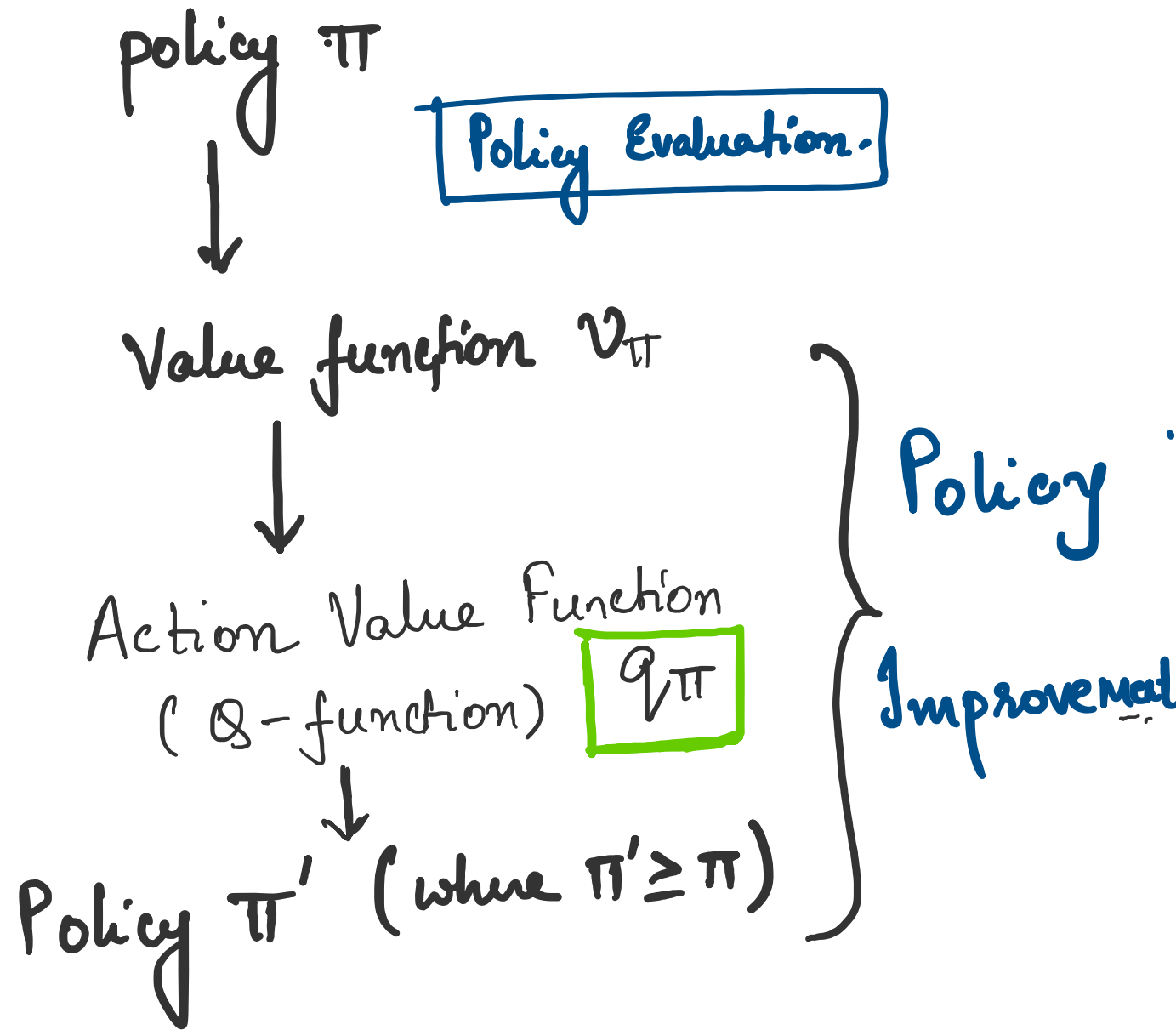
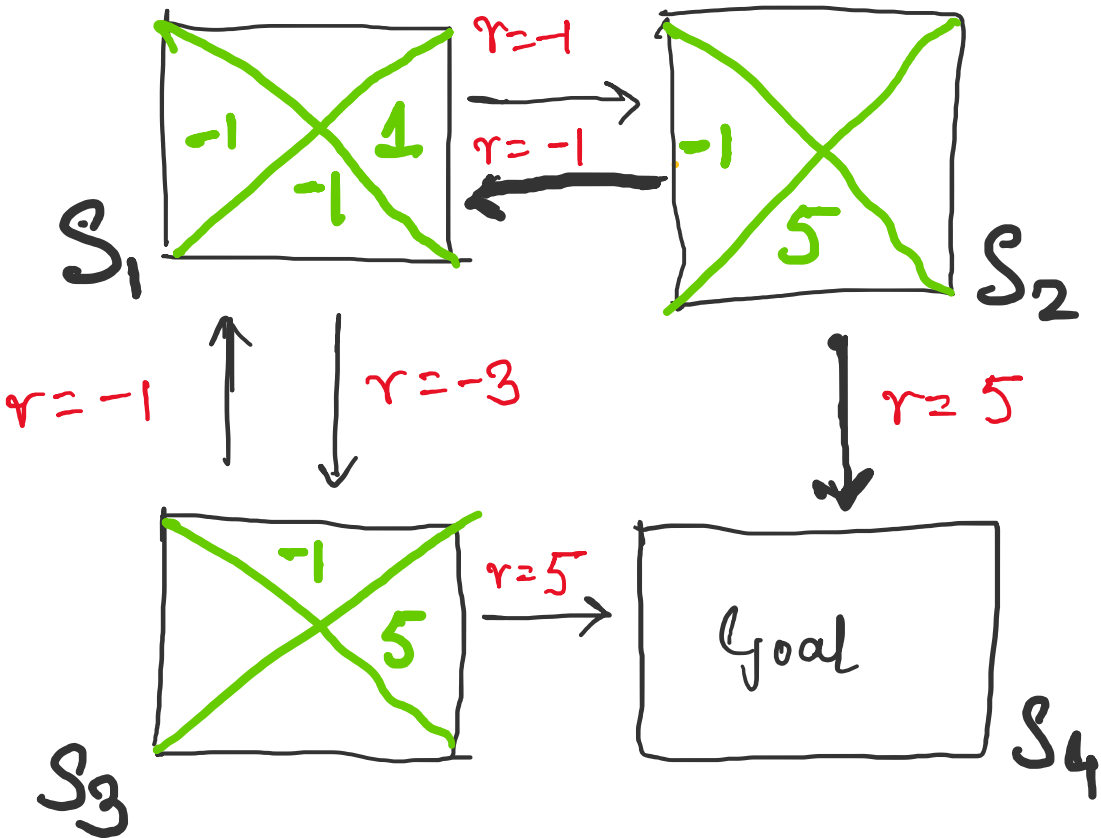
Policy Improvement





Now, how to improve policy.

choose policy $\pi' = \text{greedy}(\pi)$ → gives better value for all V_π .



Value Functions → V-functions and Q-functions

Value Function (V-function and Q-function)

State Value Function (V function)

V-function $V_{\pi}(s) \rightarrow$ expected 'return' from state 's' and following policy π

$$V_{\pi}(s) = \mathbb{E}_{\pi} [G_t \mid S_t = s]$$

(Remember:
 $G_t = R_{t+1} + \gamma R_{t+2} + \dots$
 $= \sum_{k=0}^{\infty} \gamma^k \cdot R_{t+k+1}$)

Action Value Function (Q-function)

Q-function $\rightarrow$ expected return starting from state 's', taking action 'a', and follow policy π .

$$q_{\pi}(s, a) = \mathbb{E}_{\pi} [G_t \mid S_t = s, A_t = a]$$

V-function can be decomposed $\rightarrow$ immediate reward
+ all future states/rewards

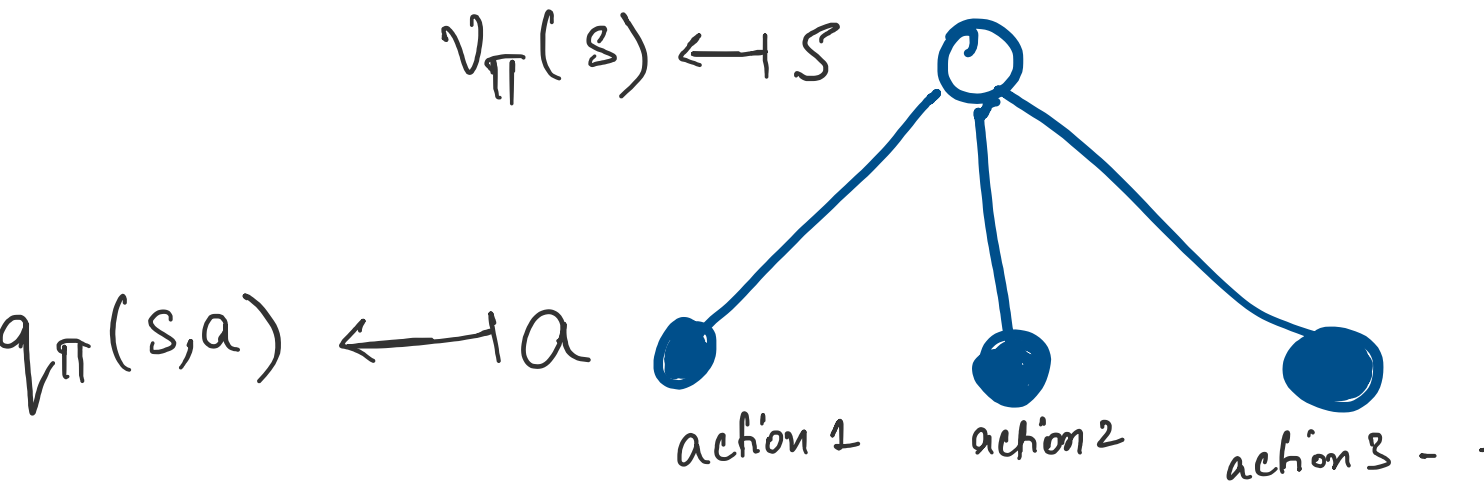
$$v_{\pi}(s) = \mathbb{E}_{\pi} \left(R_{t+1} + \gamma v_{\pi}(s_{t+1}) \mid S_t = s \right)$$

Q-function can also be decomposed.

$$q_{\pi}(s, a) = \mathbb{E}_{\pi} \left(R_{t+1} + \gamma q_{\pi}(s_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right)$$

- $\rightarrow$ take some action from state s
- $\rightarrow$ get immediate reward for that action
- $\rightarrow$ ask Q-value of next state we end up in,
under the next action chosen from that time.

How are V-function and Q-function related?



$$V_{\pi}(s) = \sum_{a \in A} \pi(a/s) \cdot q_{\pi}(s,a)$$

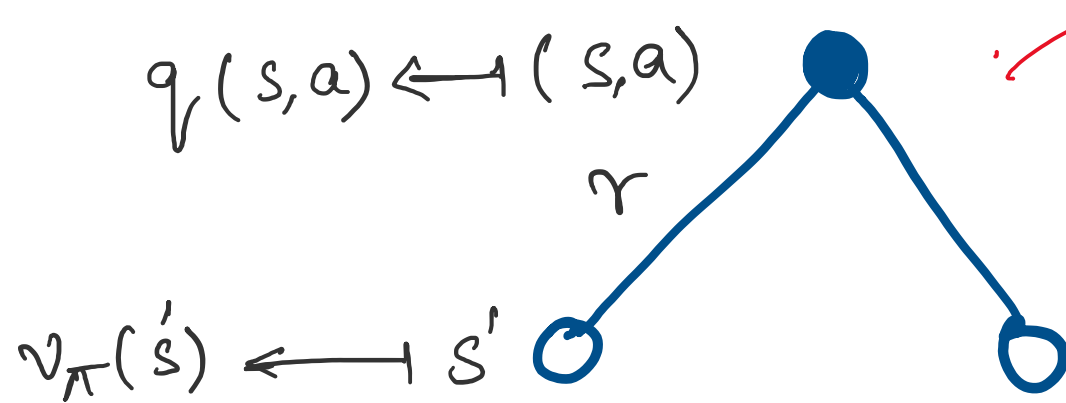
→ from state 's' under policy π

→ three actions taken.

→ each action has action value.

→ average over all such values → value $V_{\pi}(s)$ of being in state 's'.
(at top)

How to interpret Q^π



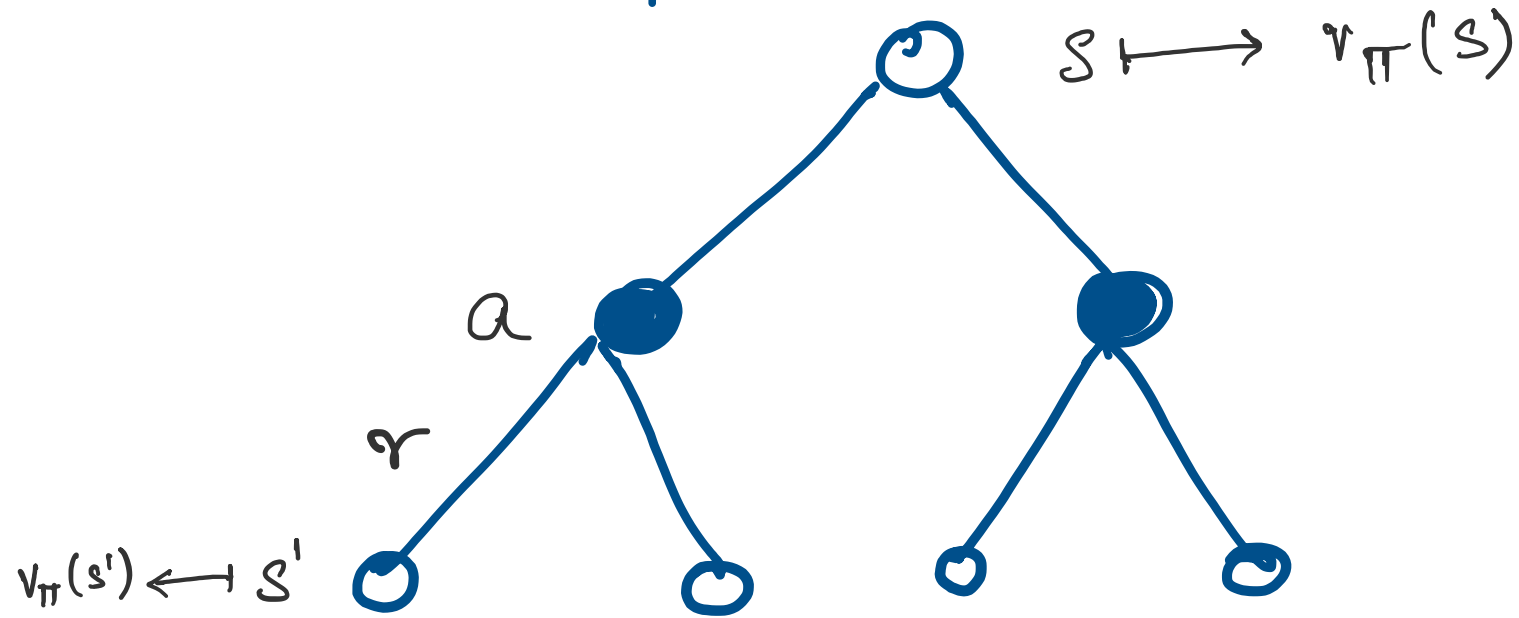
→ When in state 's'
how good is it to take
action 'a'.

after transition
with reward 'r'
how good to
be in that state.

$$q_\pi(s, a) = R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') \cdot v_\pi(s')$$

Bellman Equation

Bellman Equation (Expectation) for v_π



$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') v_\pi(s') \right)$$

Bellman Equation (Expectation) for V_π

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') V_\pi(s') \right)$$

Value of being
in state 's'

Average over
all actions

probability
of taking
action a in
s

Reward
when
in s

Average over
all states

Prob of being in
s' from s

Value of
being in
s'

Bellman Equation (Expectation) for V_π

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \left(R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') V_\pi(s') \right)$$

Value of being
in state 's'

at time 't'

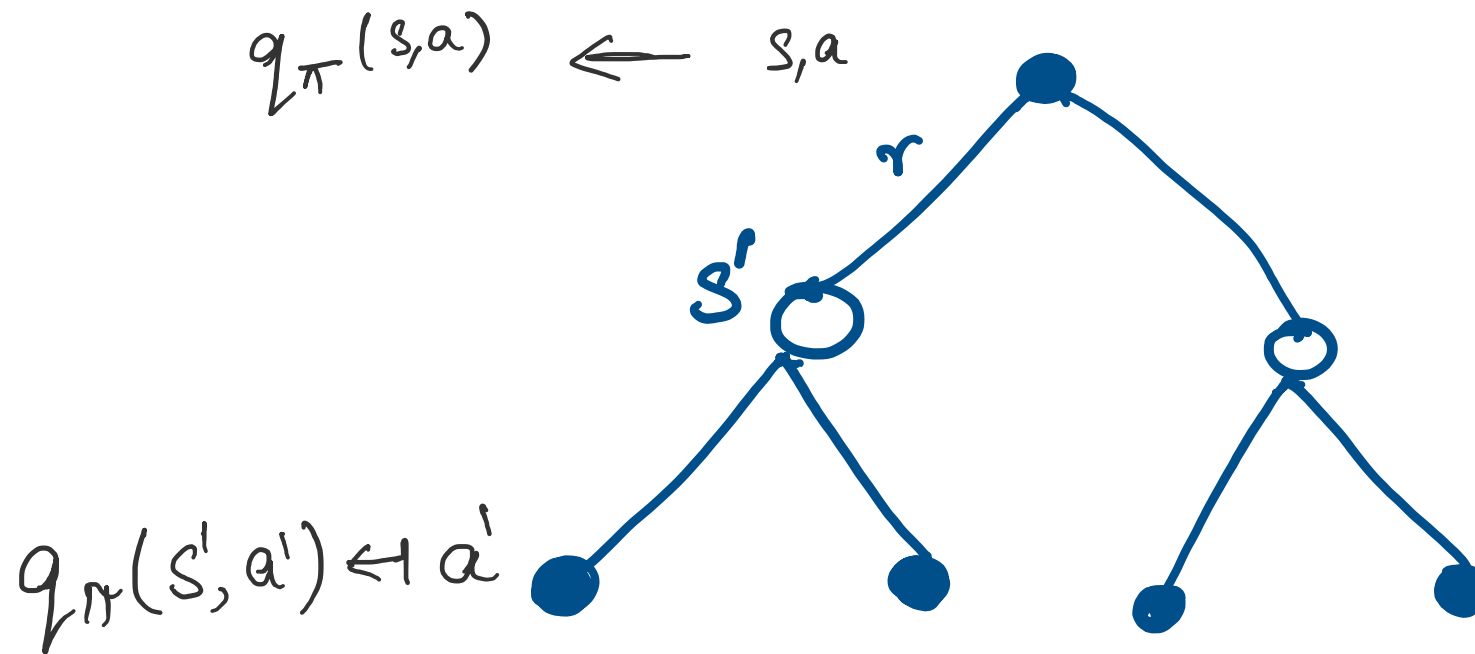
sum over
all actions

sum over
all states!!

Value of being
in state

s' at t+1

Bellman Equation for Q-function.



$$V_{\pi}(s) = \sum_{a \in A} \pi(a|s) \cdot q_{\pi}(s,a)$$

$$q_{\pi}(s,a) = R_a(s) + \gamma \sum_{s' \in S} P_a(s,s') \sum_{a' \in A} \pi(a'|s') q_{\pi}(s',a')$$

Bellman Equation for Q-function.

$$q_{\pi}(s, a) = R_a(s) + \gamma \sum_{s' \in S} P_a(s, s') \sum_{a' \in A} \pi(a' | s') q_{\pi}(s', a')$$

Value of being
in state s
with respect to
action a

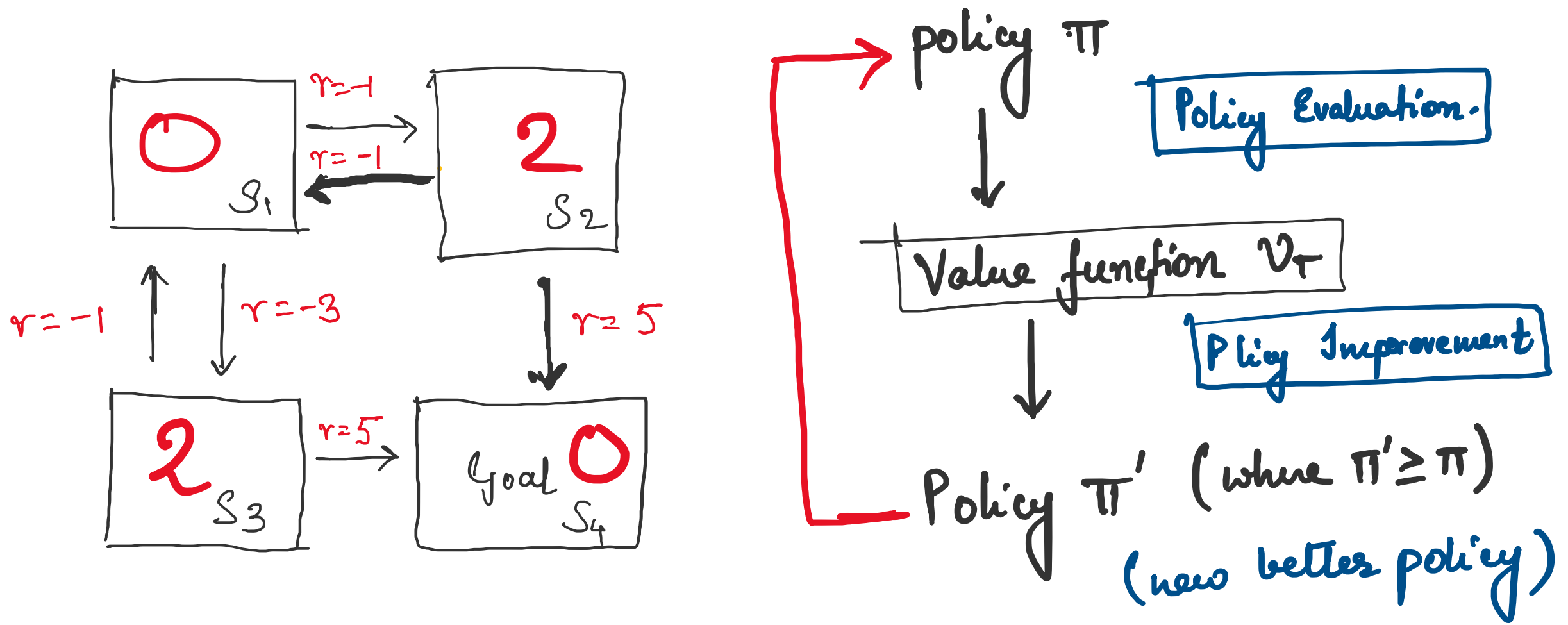
sum over all
states

Probability of
passing from
s to s'

Prob of
action a'
when in
state s'

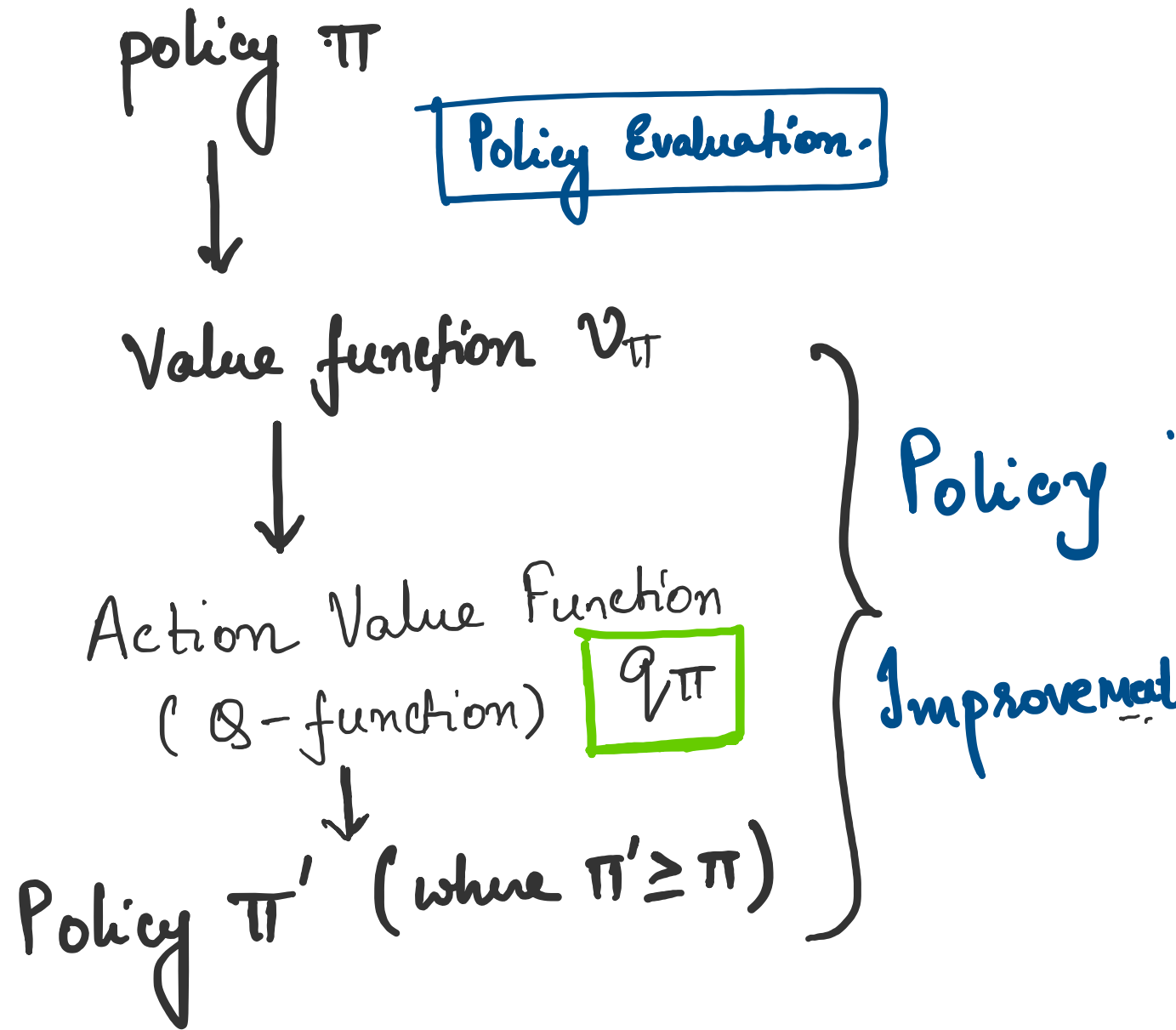
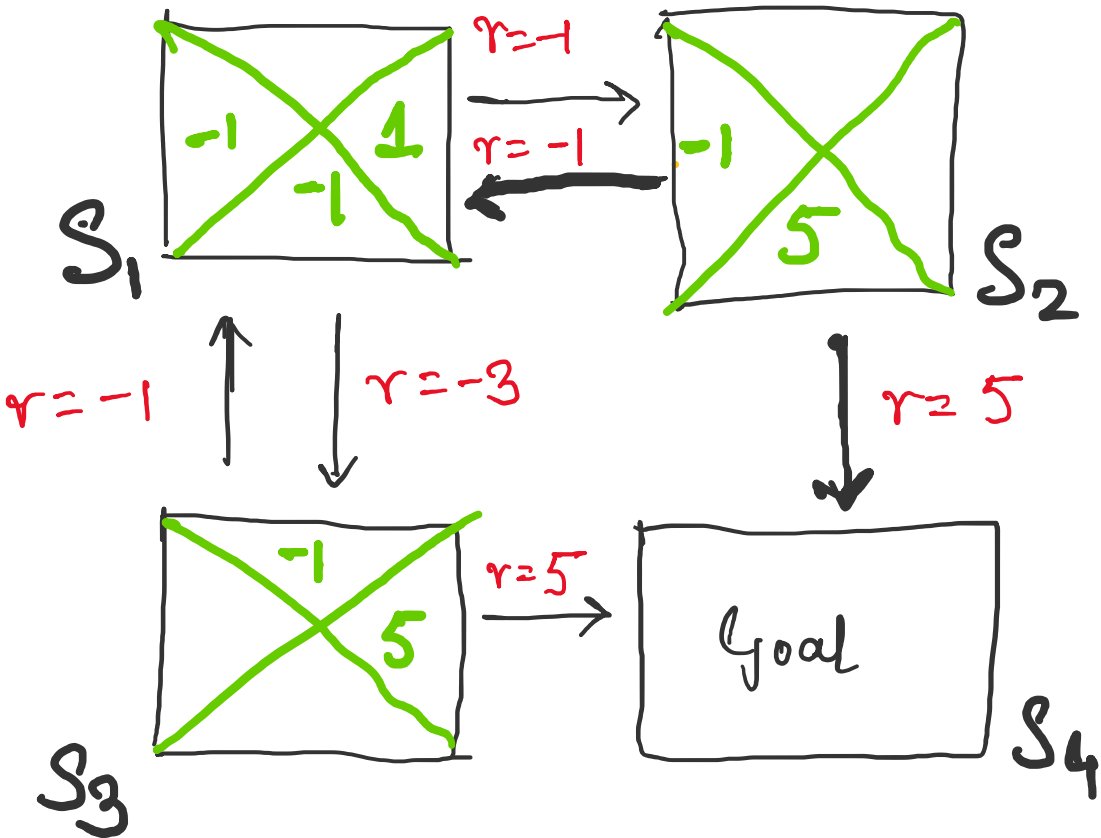
Value of being
in s' with
respect to
a'.

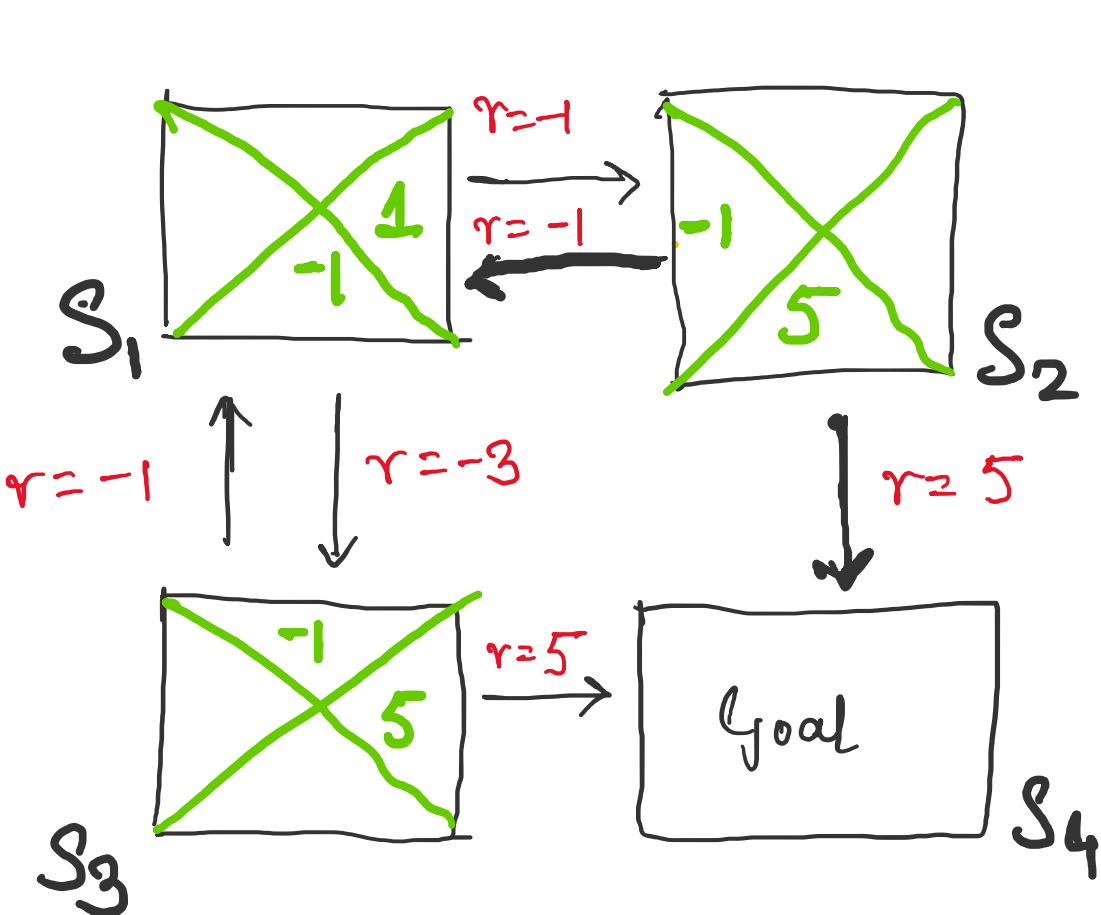
s → at time t
s' → at time t+1.



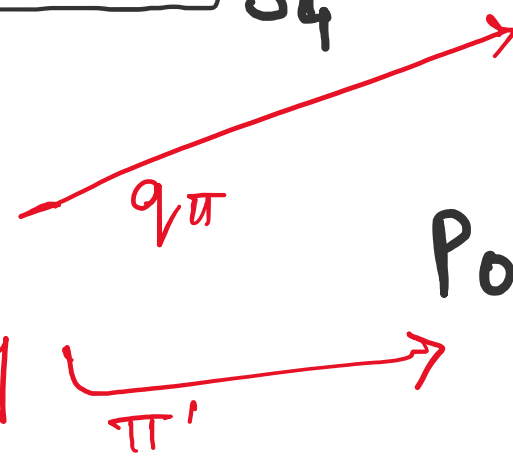
Now, how to improve policy.

choose policy $\pi' = \text{greedy}(\pi) \rightarrow$ gives better value for all V_π .





Focus: How to use
to choose better policy



policy π

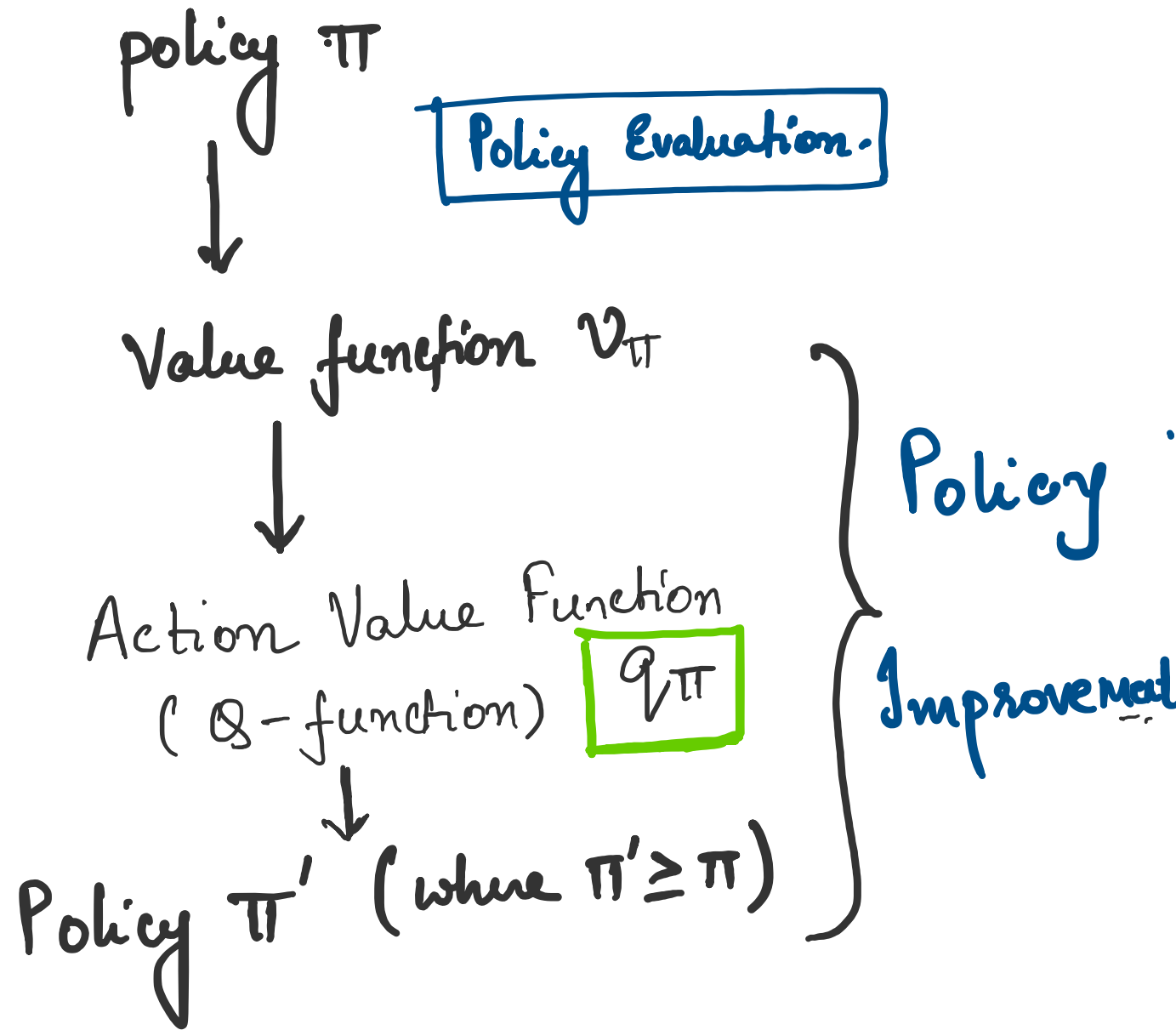
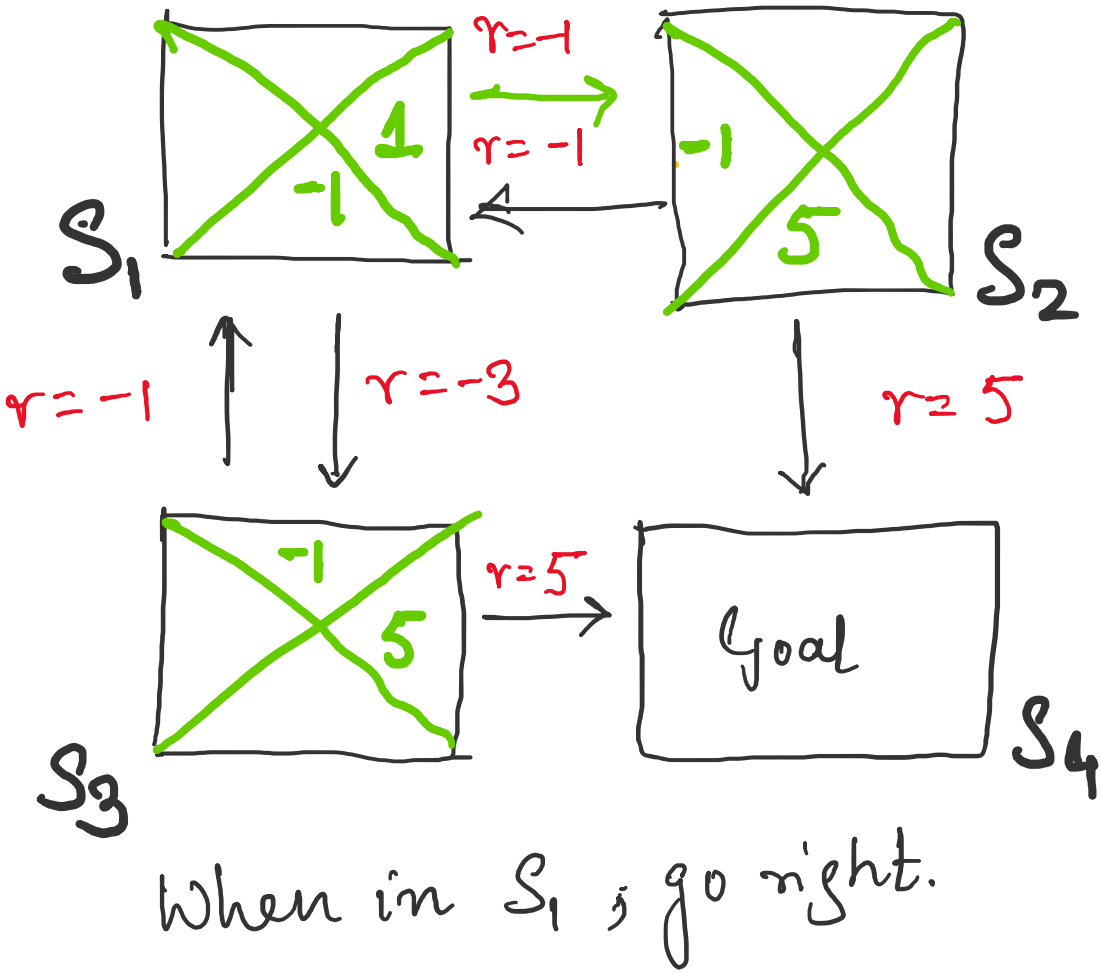
Policy Evaluation.

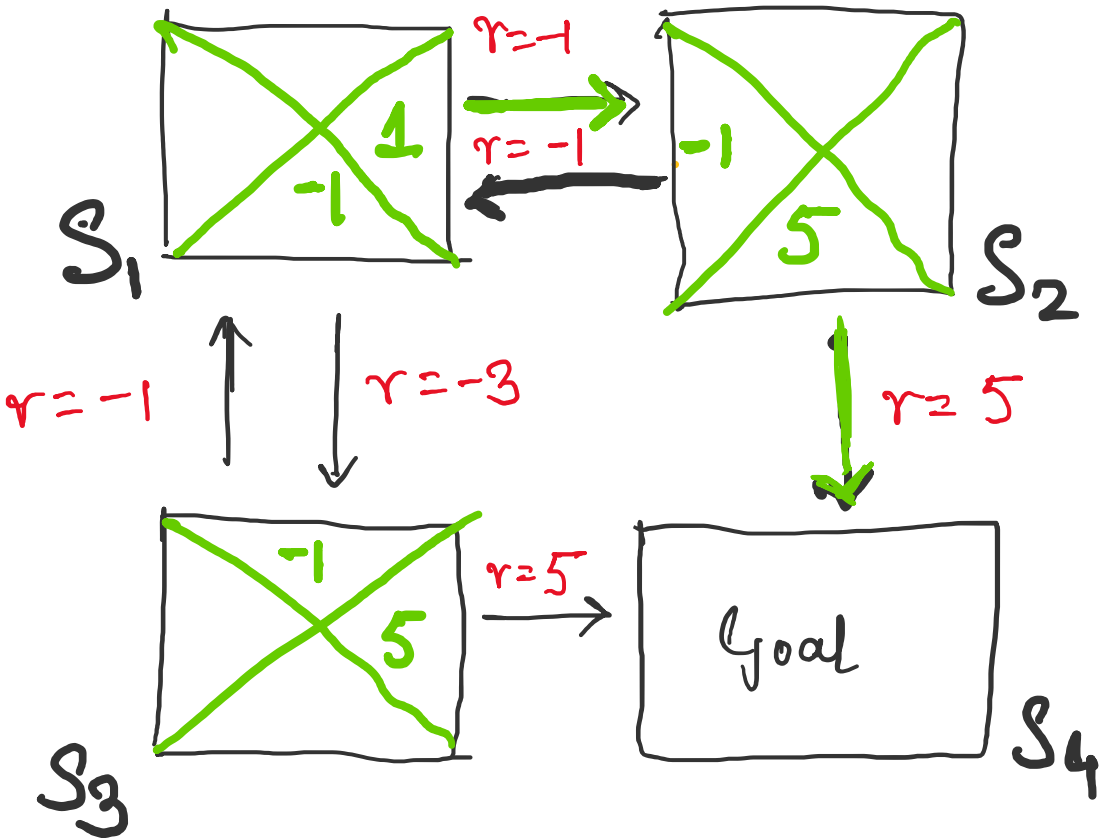
Value function V_π

Action Value Function
(Q-function) Q_π

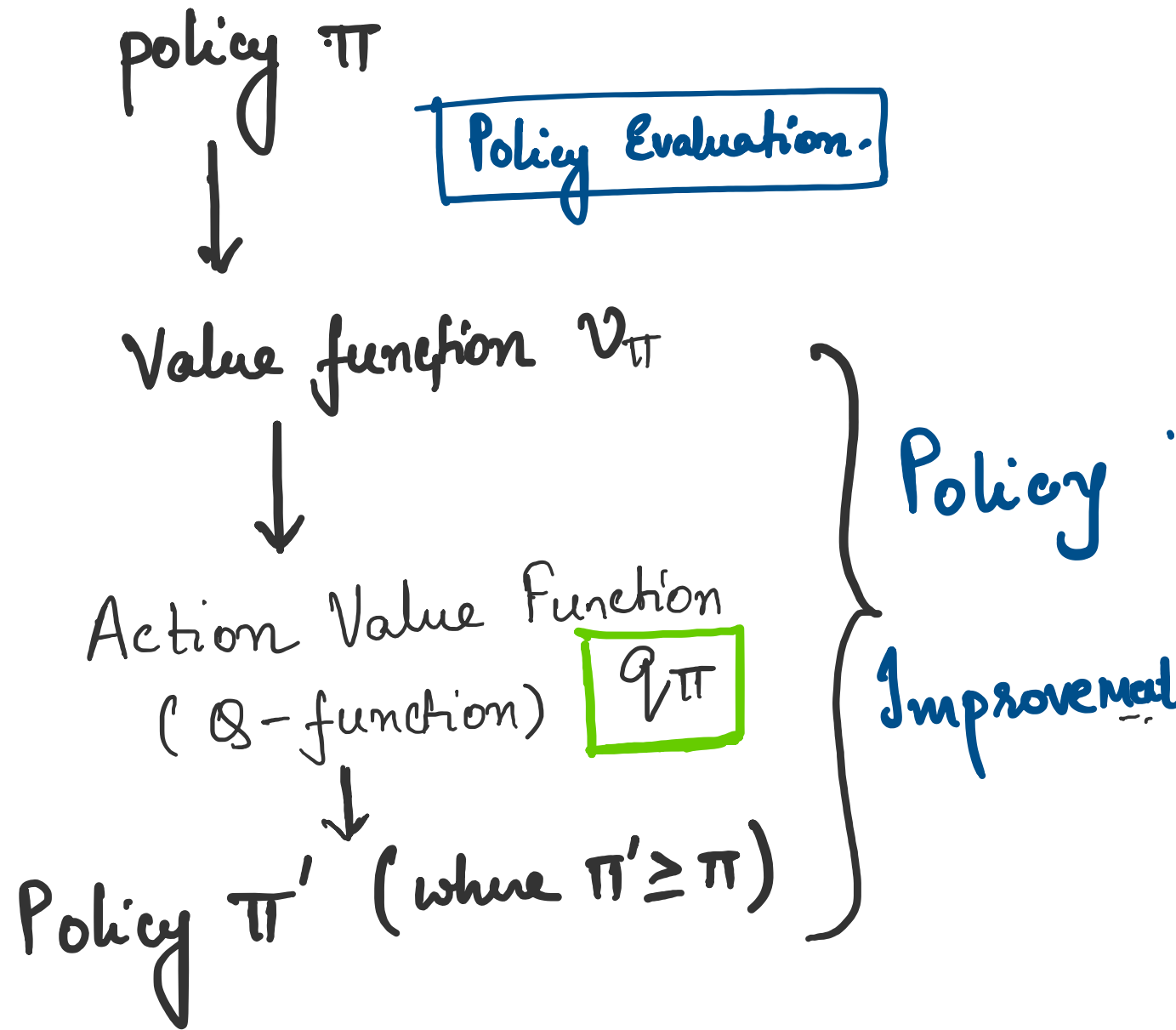
Policy π' (where $\pi' \geq \pi$)

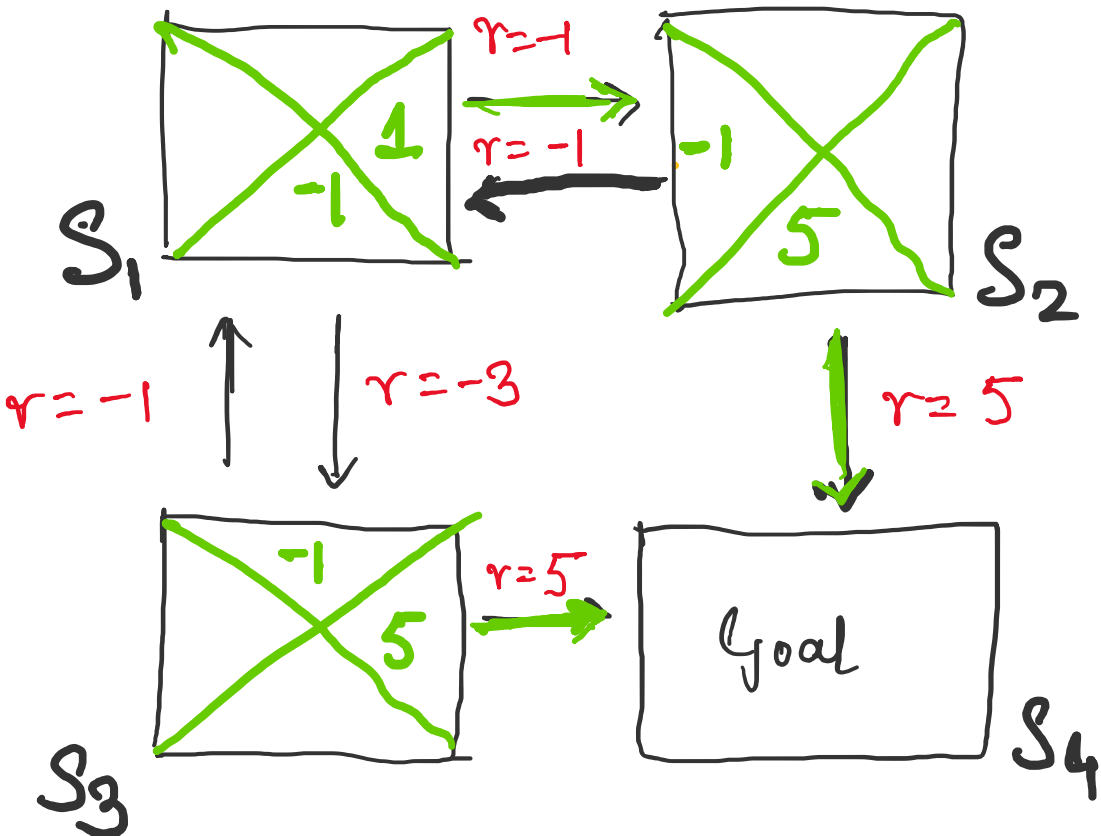
Policy Improvement





When in S_2 , go down.





When in S_3 , go right.

Optimal policy in Green. ←

policy π

Policy Evaluation.

Value function V_π

Action Value Function
(Q-function) Q_π

Policy π' (where $\pi' \geq \pi$)

(if and only if, $V_{\pi'}(s) \geq V_\pi(s)$ for all $s \in S$)

Policy

Improvement

Policy Improvement.

- Consider a policy $a = \pi(s)$
- Improve the policy by greedy approach.

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

- This improves value from any state 's' over one step.

$$q_{\pi}(s, \pi'(s)) = \max_{a \in A} q_{\pi}(s, a) \geq q_{\pi}(s, \pi(s)) = v_{\pi}(s)$$

Policy Improvement.

Value function $V_{\pi'}(s) \geq V_{\pi}(s)$
is improved.

$$\begin{aligned} V_{\pi}(s) &\leq q_{\pi}(s, \pi'(s)) = \mathbb{E}_{\pi'}(R_{t+1} + \gamma V_{\pi}(S_{t+1}) \mid S_t = s) \\ &\leq \mathbb{E}_{\pi'}(R_{t+1} + \gamma q_{\pi}(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s) \\ &\leq \mathbb{E}_{\pi'}(R_{t+1} + \gamma R_{t+2} + \gamma^2 q_{\pi}(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s) \\ &\leq \mathbb{E}_{\pi'}(R_{t+1} + \gamma R_{t+2} + \dots \mid S_t = s) = V_{\pi'}(s) \end{aligned}$$

Thus, $V_{\pi}(s) \leq V_{\pi'}(s)$

Policy Improvement

→ If improvements stop,

$$q_{\pi'}(s, \pi'(s)) = \max_{a \in A} q_{\pi}(s, a) = q_{\pi}(s, \pi(s)) = v_{\pi}(s)$$

→ Then, Bellman Optimality is stopped;

$$v_{\pi}(s) = \max_{a \in A} q_{\pi}(s, a)$$

→ Therefore, $v_{\pi}(s) = v_{*}(s)$ for all $s \in S$

π is optimal policy

Policy iteration algorithm

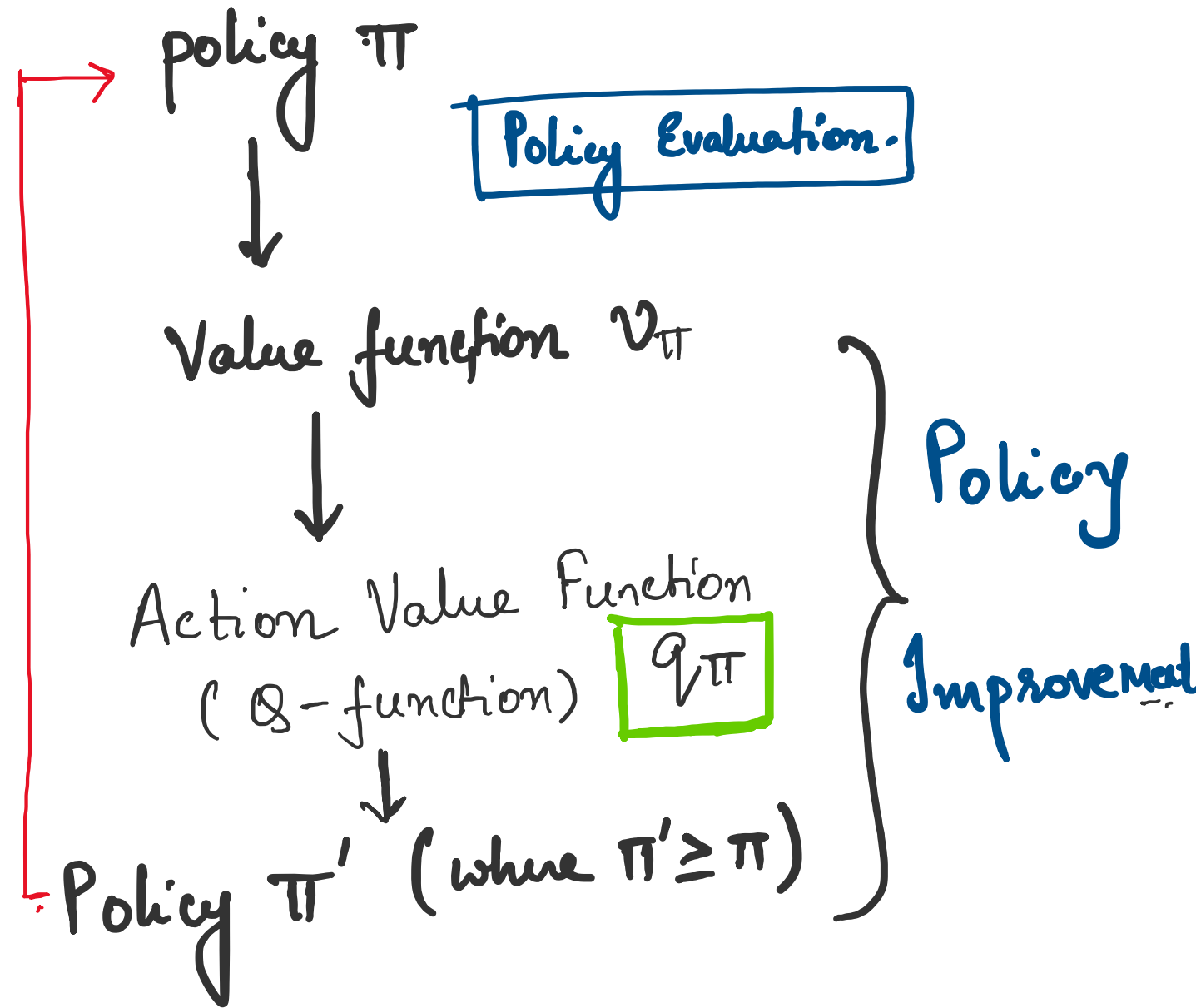
=

Policy evaluation

+

Policy improvement

Policy Iteration



Policy Iteration

=

Policy Evaluation

for $s \in S$

$$V(s) \leftarrow \sum_{s' \in S, r \in R} p(s', r | s, \pi(s)) (r + \gamma V(s'))$$

+

Policy Improvement

for $s \in S$

for $a \in A(s)$

$$Q(s, a) \leftarrow \sum_{s' \in S, r \in R} p(s', r | s, a) (r + \gamma V(s'))$$

$$\pi'(s) \leftarrow \arg \max_{a \in A(s)} Q(s, a)$$

Policy Iteration Algorithm

Input: MDP, small positive number θ

Output: policy $\pi = \pi^*$

$$\pi(a/s) = \frac{1}{|A(s)|} \text{ for all } s \in S \text{ and } a \in A(s)$$

begin with equiprobable random policy

repeat:

$V \leftarrow$ Policy-Evaluation (MDP, π , θ)

obtain V-fun wrt π

$\pi' \leftarrow$ Policy-Improvement (MDP, V)

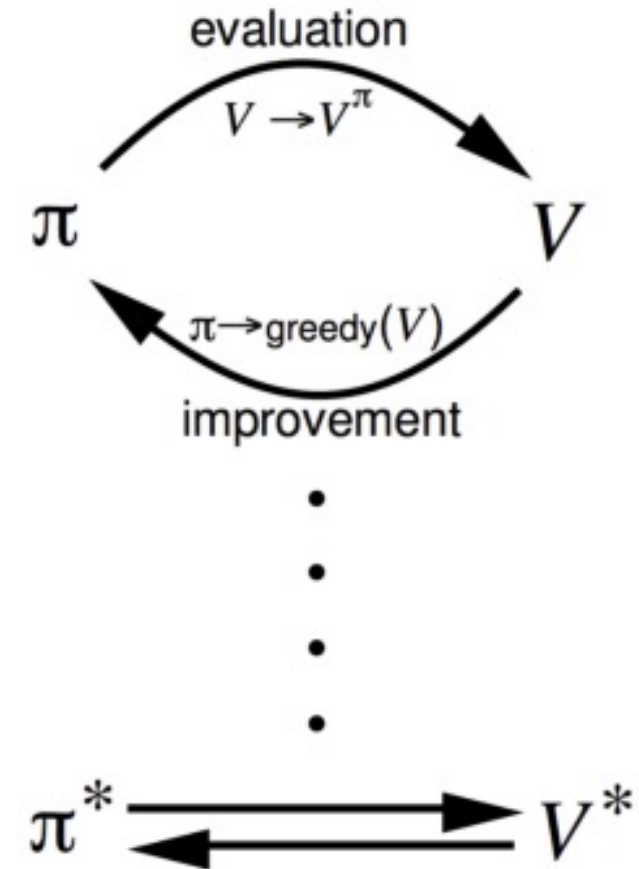
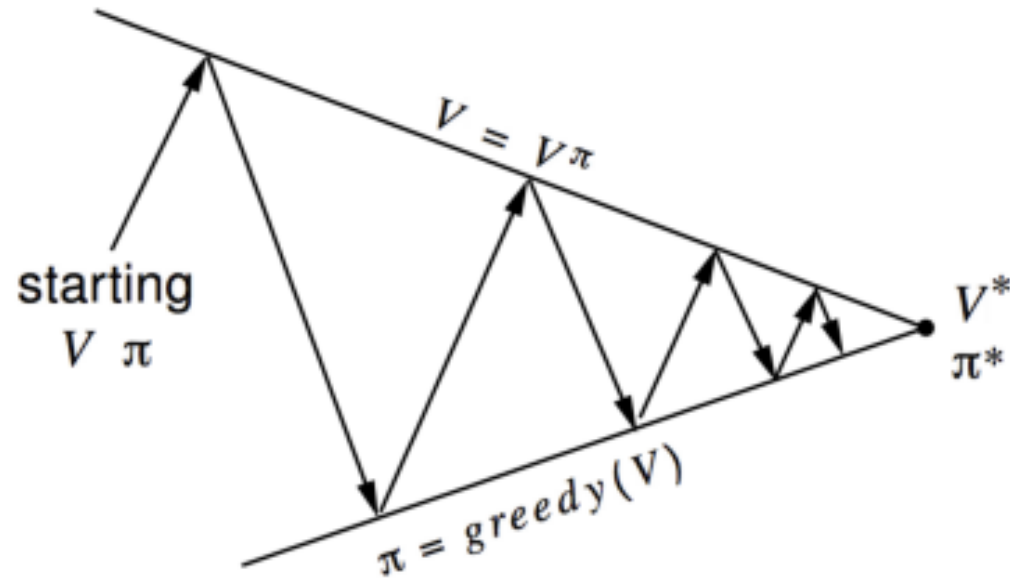
obtain better or equal policy from V-function

if $\pi = \pi'$ then Break

$\pi \leftarrow \pi'$

return π

Policy Iteration



Policy Evaluation : Estimate V^π

Policy Improvement : Greedy ($\pi' \geq \pi$)

Principle of Optimality

Optimal policy = optimal first action A^* + optimal policy from successor state s' .

Theorem: A policy $\pi(a/s)$ achieves optimal value from state ' s ',

$V_\pi(s) = V_{A^*}(s)$ if and only if

- any state s' is reachable from s

- π achieves optimal value from s'

or $V_\pi(s') = V_{A^*}(s')$

Introduction

Markov Decision process

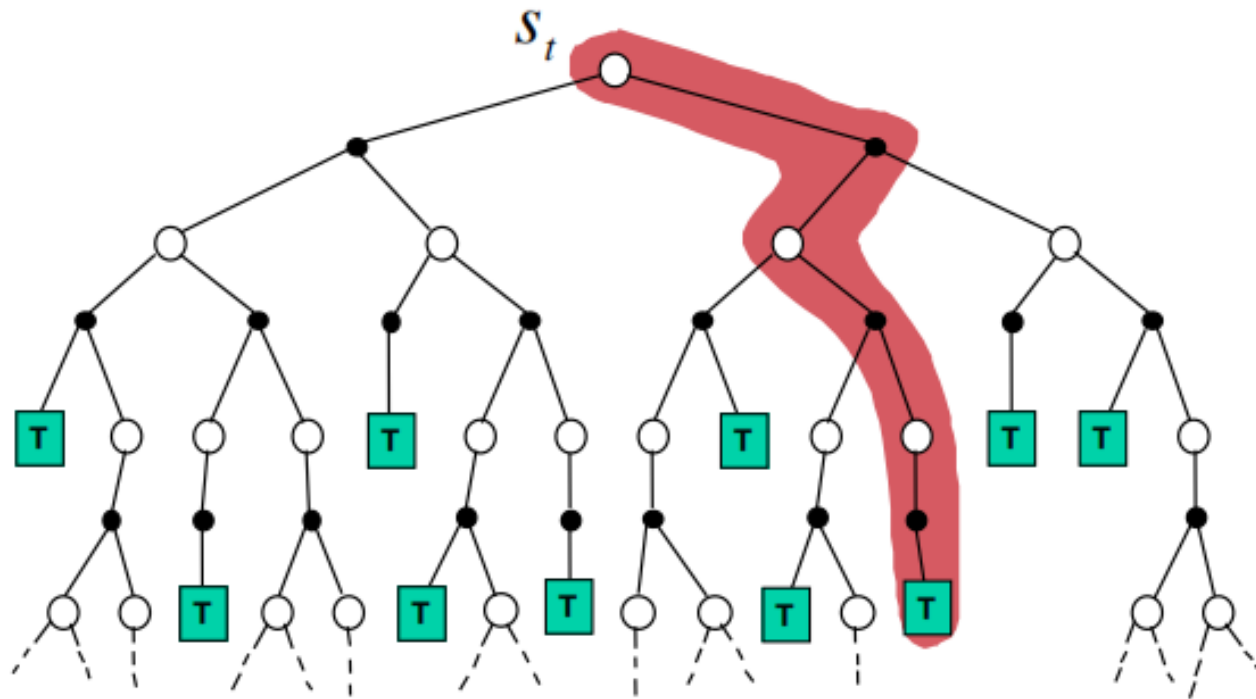
Dynamic Programming for planning
and control

Model Free Prediction and Control
(SARSA and Q-learning)

Model-free Prediction

Temporal Difference (TD)

Learning : on the fly



(1) We again start from the definition of the state-value function. $v_{\pi}(s) = \mathbb{E}_{\pi}[G_{t:T} \mid S_t = s]$

(2) And the definition of the return.

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

(3) From the return, we can rewrite the equation by grouping up some terms. Check it out.

$$\begin{aligned} G_{t:T} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots + \gamma^{T-2} R_T) \\ &= R_{t+1} + \gamma G_{t+1:T} \end{aligned}$$

(4) Now, the same return has a recursive style.

(5) We can use this new definition to also rewrite the state-value function definition equation.

$$v_{\pi}(s) = \mathbb{E}_{\pi}[G_{t:T} \mid S_t = s]$$

$$= \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1:T} \mid S_t = s]$$

$$= \mathbb{E}_{\pi}[R_{t+1} + \gamma v_{\pi}(S_{t+1}) \mid S_t = s]$$

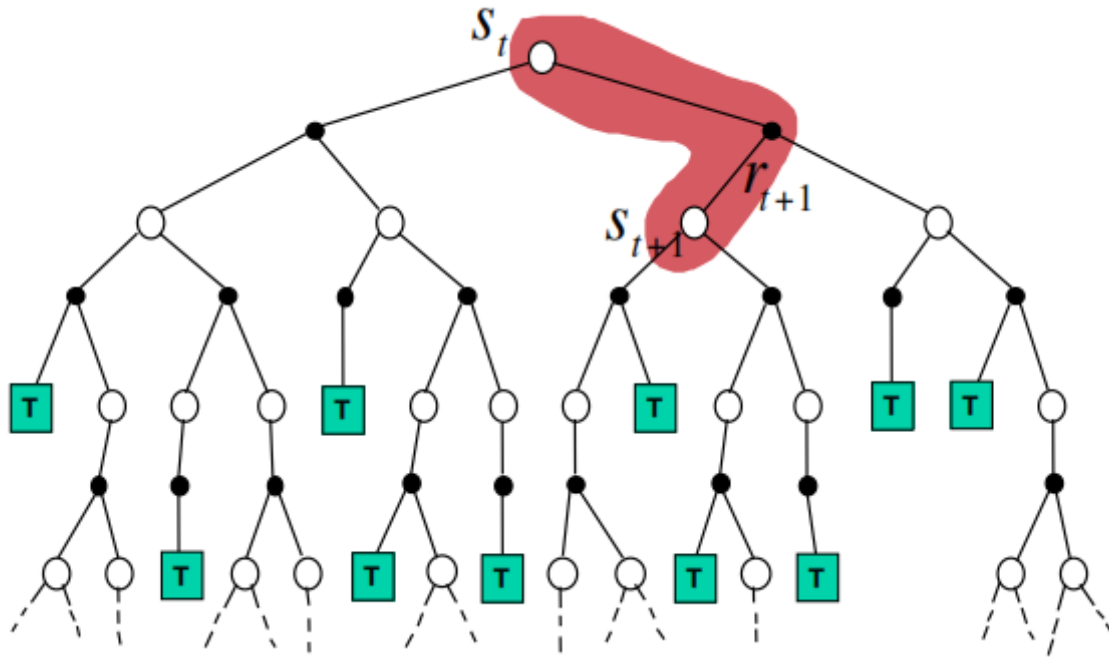
(6) And because the expectation of the returns from the next state is simply the state-value function of the next state, we get.

(7) This means we could estimate the state-value function on every time step.

$$S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{t:t+1}$$

(8) We roll out a single interaction step.

Temporal Difference TD (0) Algorithm



(1) We again start from the definition of the state-value function. $v_\pi(s) = \mathbb{E}_\pi[G_{t:T} \mid S_t = s]$

(2) And the definition of the return.

$$G_{t:T} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

(3) From the return, we can rewrite the equation by grouping up some terms. Check it out.

$$\begin{aligned} G_{t:T} &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots + \gamma^{T-2} R_T) \\ &= R_{t+1} + \gamma G_{t+1:T} \end{aligned}$$

(4) Now, the same return has a recursive style.

(5) We can use this new definition to also rewrite the state-value function definition equation.

$$v_\pi(s) = \mathbb{E}_\pi[G_{t:T} \mid S_t = s]$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1:T} \mid S_t = s]$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$$

(6) And because the expectation of the returns from the next state is simply the state-value function of the next state, we get.

(7) This means we could estimate the state-value function on every time step.

$$S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{t:t+1}$$

(8) We roll out a single interaction step.

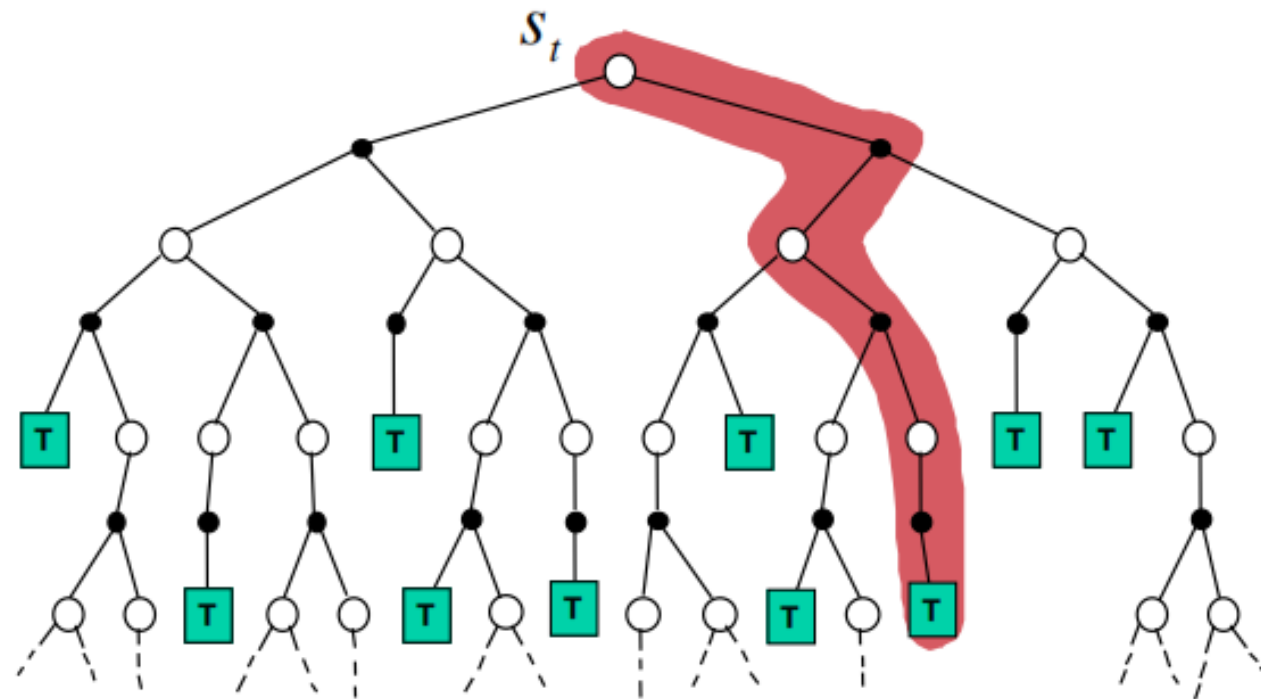
(9) And can obtain an estimate $V(s)$ of the true state-value function $v_\pi(s)$ a different way than with MC.

(10) The key difference to realize is we are now estimating $v_\pi(s_t)$ with an estimate of $v_\pi(s_{t+1})$. We are using an estimated, not an actual return.

$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \left[\underbrace{R_{t+1} + \gamma V_t(S_{t+1})}_{\text{TD target}} - \underbrace{V_t(S_t)}_{\text{TD error}} \right]$$

(11) A big win is we can now make updates to the state-value function estimates $V(s)$ every time step.

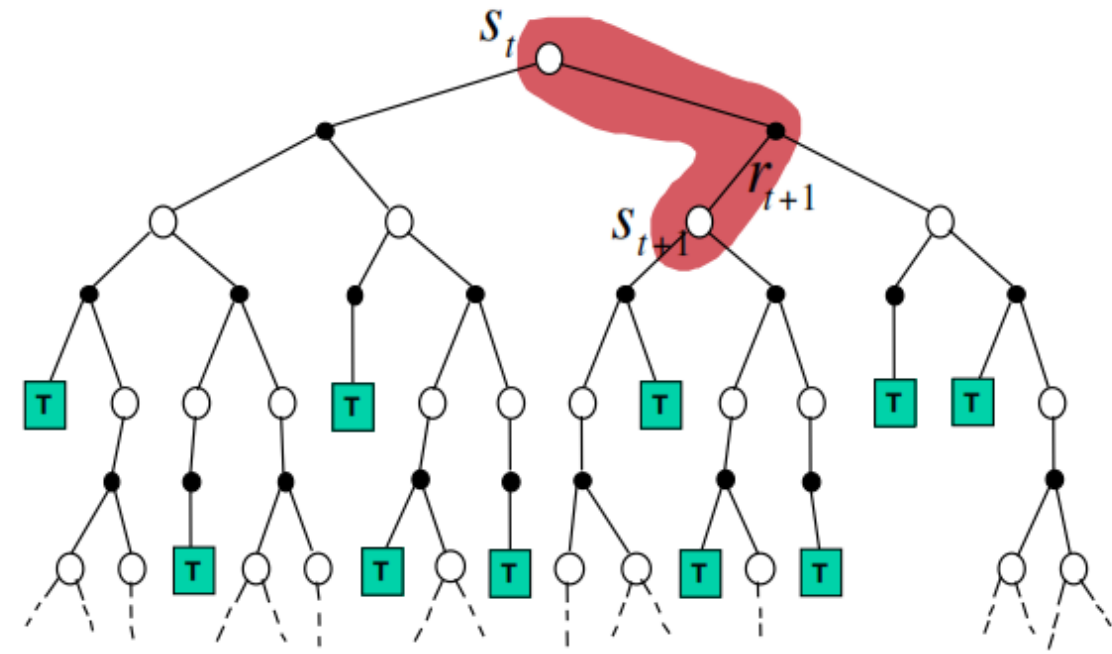
Monte Carlo Learning



$$V_T(S_t) = V_{T-1}(S_t) + \alpha_t \left[\underbrace{G_{t:T}}_{\text{MC target}} - \underbrace{V_{T-1}(S_t)}_{\text{MC error}} \right]$$

(1) If we get a close-up (log-scale plot) these trends, you will see what's happening. MC estimates jump around the true values. This is because of the high variance of the MC targets.

Temporal Difference TD (0) Algorithm



$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \left[\underbrace{G_{t:t+1}}_{\text{TD target}} - \underbrace{V_t(S_t)}_{\text{TD error}} \right]$$

(3) TD estimates are off target most of the time, but they are less jumpy. This is because TD targets are low variance, though biased. They use an estimated return for target.

Differences between TD (0) and MC methods and other variants

1. TD can learn from episodes of experiences
2. TD is model free: No knowledge of MDP required.
3. TD learns by bootstrapping
4. TD can learn from incomplete episodes



Learn V_{π} from experience , under policy π

Dynamic Programming Backup

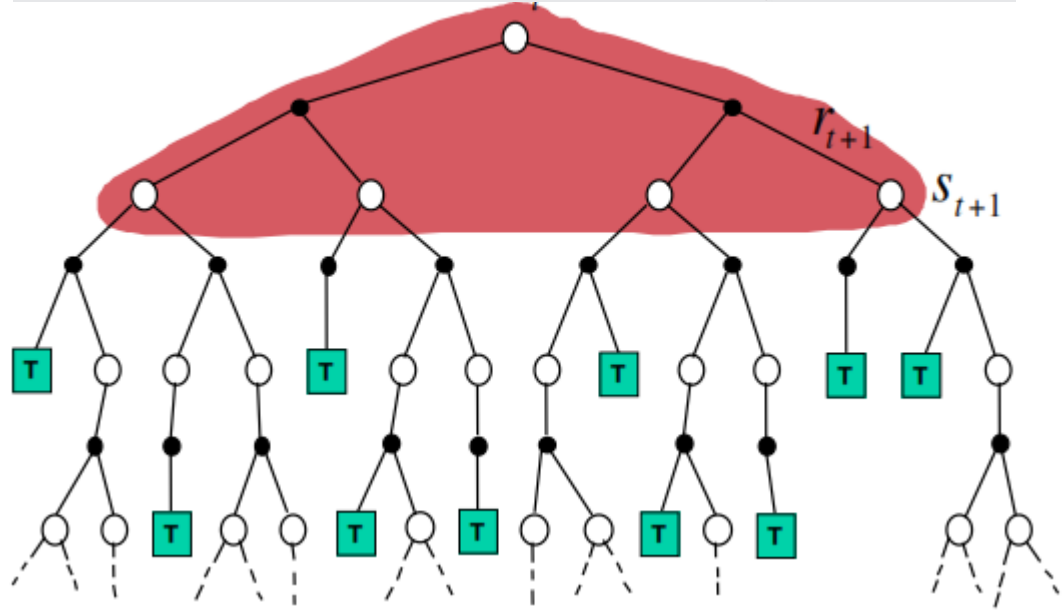
(9) And can obtain an estimate $V(s)$ of the true state-value function $v_{\pi}(s)$ a different way than with MC.

(10) The key difference to realize is we are now estimating $v_{\pi}(s_t)$ with an estimate of $v_{\pi}(s_{t+1})$. We are using an estimated, not an actual return.

$$V_{t+1}(S_t) = V_t(S_t) + \alpha_t \left[\underbrace{R_{t+1} + \gamma V_t(S_{t+1})}_{\text{TD target}} - V_t(S_t) \right]$$

TD error

(11) A big win is we can now make updates to the state-value function estimates $V(s)$ every time step.



Model-free Control

TD control (On-policy SARSA)

TD control (Off-Policy) Q-learning

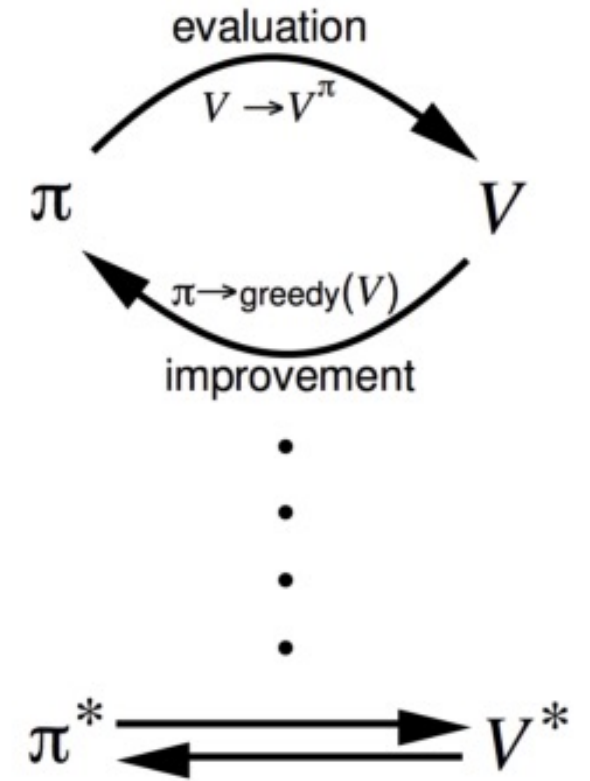
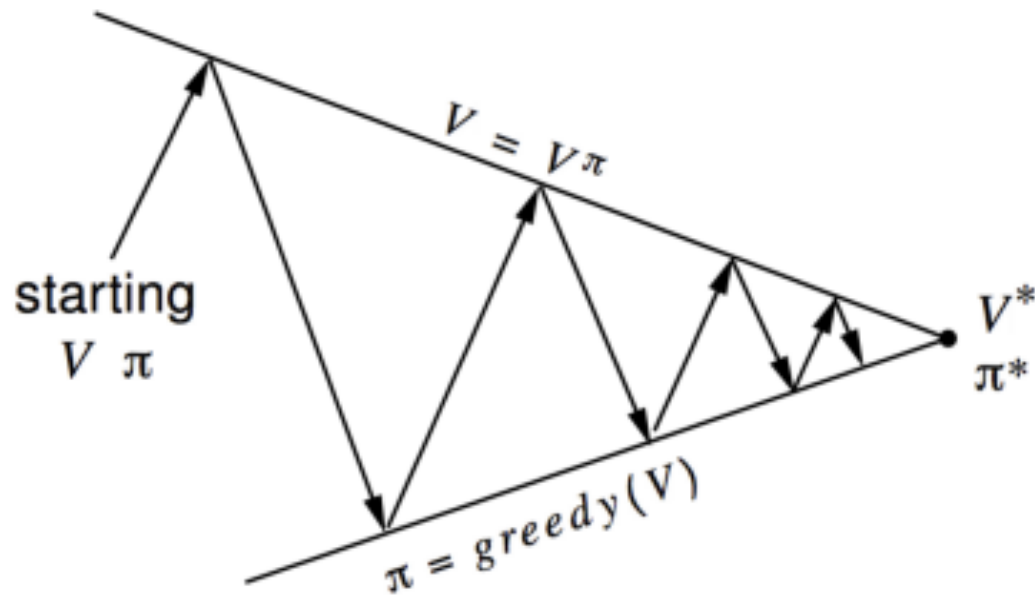
Model Free Control

→ MDP model is unknown,
experience sampling is possible!

On-Policy learning: learn while on 'RUN'
learn about policy π , from experience
sampled from π

Off-Policy learning: learn while off-line.
learn about policy π from experience
sampled from μ .

Generalised Policy Iteration



Policy Evaluation: Evaluate V_π (ϵ_n : Iterative Policy Evaluation)

Policy Improvement: Generate $\pi' \geq \pi$ (ϵ_n : greedy Policy Improvement)

what is greedy strategy ??

Greedy Policy Improvement Example:

$$\pi'(s) = \arg \max_{a \in A} Q(s, a)$$

choose the action that maximises the Q -fun value.

the action chosen is greedy with respect to our objective

Chooses action
that gives max

maximise
the
value!

What is limitation of greedy policy → can get stuck at local minimas!
Imagine being greedy but not exploring all state space!

Exploration

ϵ -Greedy satisfies the following conditions

ϵ -Greedy Exploration.

→ Choose greedy action : with probability $(1-\epsilon)$

→ choose random action : with probability ϵ

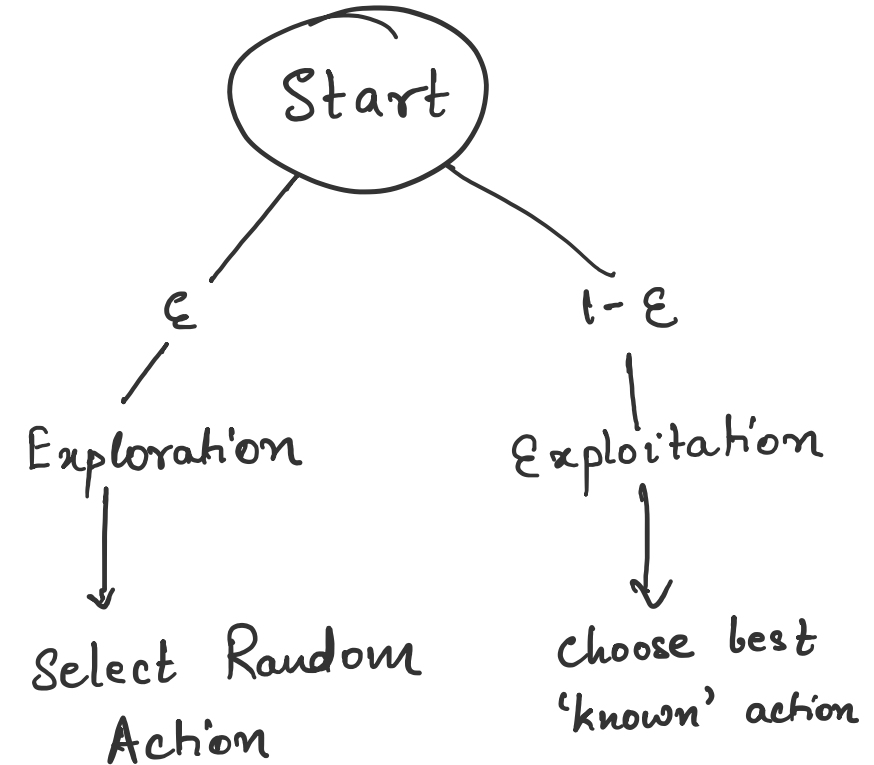
Greedy in Limit with Infinite Exploration (GLIE)

→ all state-action pairs are explored infinitely many times.

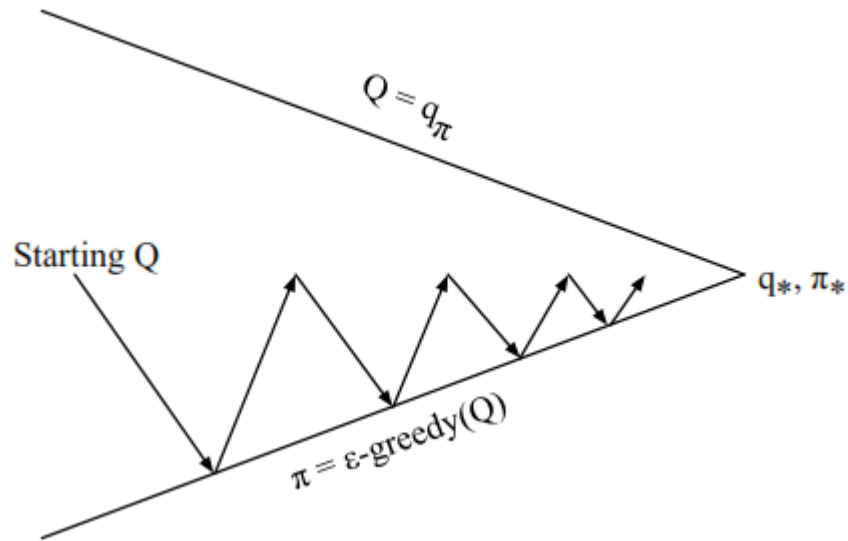
→ Policy converges on a greedy policy in asymptotic sense

$$\lim_{k \rightarrow \infty} \pi_k(a|s) = 1 \quad (a = \underset{a' \in A}{\operatorname{argmax}} Q_k(s, a'))$$

Note: ϵ -greedy with $\epsilon_k = 1/k$ is GLIE.



Policy Iteration (TD control)



Idea:

- 1) Apply TD to update $Q(s,a)$
- 2) Use ϵ -greedy policy improvement
- 3) Update every time step!

Every time step:

- Policy Evaluation: Use TD to update $Q(s,a)$
$$Q(s,a) \leftarrow Q(s,a) + \alpha (R + \gamma Q(s',a') - Q(s,a))$$
- Policy Improvement: ϵ -greedy policy improvement

TD Control (Update Eqn)

$$Q(s_t, A_t) \leftarrow Q(s_t, A_t) + \alpha \left(\underbrace{R_{t+1} + \gamma Q(s_{t+1}, A_{t+1})}_{\text{TD target}} - \underbrace{Q(s_t, A_t)}_{\text{current estimate}} \right)$$



V-function is good for theory development, but policy extraction requires MDP!
(model knowledge)

So, we use Q-function , in place of V-function!

TD Control (Algorithm)

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

Initialize : $Q(s, a) \forall s \in S, a \in A$. $\rightarrow$ initialize with all action-values to zero.

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

$\rightarrow$ Choose an initial policy

s_0, a_0, R_1

$\rightarrow$ interact with environment; take action a_0
 $\rightarrow$ obtain the reward.

TD Control (Algorithm)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(R_{t+1} + \gamma Q(s_{t+1}, A_{t+1}) - Q(s_t, A_t) \right)$$

Initialize : $Q(s, a) \forall s \in S, a \in A$. $\rightarrow$ initialize with all action-values to zero.

$\pi \leftarrow \epsilon\text{-greedy}(Q)$ $\rightarrow$ Choose an initial policy

S_0, A_0, R_1, S_1, A_1

$\rightarrow$ next state is generated

$\rightarrow$ Use same π to generate next action A_1

TD Control (Algorithm)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\underbrace{R_{t+1} + \gamma Q(s_{t+1}, A_{t+1})}_{\text{TD target}} - \underbrace{Q(s_t, A_t)}_{\text{current estimate}} \right)$$

Initialize : $Q(s, a) \forall s \in S, a \in A$. $\longrightarrow$ initialize with all action-values to zero.

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

$| S_0 \ A_0 \ R_1 \ S_1 \ A_1 | \ R_2 \ S_2 \ A_2 \ \dots \ S_t \ A_t \ R_{t+1} \ S_{t+1} \ A_{t+1}$

Update $Q(s_0, A_0)$: $Q(s_0, A_0) \leftarrow Q(s_0, A_0) + \alpha (R_1 + \gamma Q(s_1, A_1) - Q(s_0, A_0))$
(update the value policy-evaluation)

$\pi \leftarrow \epsilon\text{-greedy}(Q)$ $\rightarrow$ choose $a = \arg\max_{a \in A} Q(s_0, A_0)$ $\rightarrow$ policy improvement.

TD Control (Algorithm)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\underbrace{R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})}_{\text{TD target}} - \underbrace{Q(s_t, a_t)}_{\text{current estimate}} \right)$$

Initialize : $Q(s, a) \forall s \in S, a \in A$.

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

$| S_0 \ A_0 \ R_1 \ S_1 \ A_1 | \ R_2 \ S_2 \ A_2 | \dots\dots\dots S_t \ A_t \ R_{t+1} \ S_{t+1} \ A_{t+1}$

Update $Q(s_0, a_0)$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

Update $Q(s_1, a_1)$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

TD Control (Algorithm) SARSA (on-policy)

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(\underbrace{R_{t+1} + \gamma Q(s_{t+1}, a_{t+1})}_{\text{TD target}} - \underbrace{Q(s_t, a_t)}_{\text{current estimate}} \right)$$

Initialize : $Q(s, a) \forall s \in S, a \in A$.

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

$| S_0 \ A_0 \ R_1 \ S_1 \ A_1 | \ R_2 \ S_2 \ A_2 | \dots\dots\dots S_t \ A_t \ R_{t+1} \ S_{t+1} \ A_{t+1} |$

Update $Q(S_0, A_0)$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

Update $Q(S_1, A_1)$
 $\pi \leftarrow \epsilon\text{-greedy}(Q)$

Update $Q(S_t, A_t)$
 $\pi \leftarrow \epsilon\text{-greedy}(Q)$

SARSA

off-policy Learning

Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$
while following another policy $\mu(a|s)$

Advantages:

- ① Learn about optimal policy while following exploratory policy.
- ② Learn about multiple policies while following one policy.
- ③ Re-use experience generated from old policies $\pi_1, \pi_2, \pi_3, \dots$
- ④ Learn by observing other agents.

TD Control (off-policy)

Q-Learning

→ Off-policy learning of $Q(s, a)$ [Q-function]

→ Next action is chosen using a behaviour policy $A_{t+1} \sim p(\cdot | s_t)$

→ But, best successor action chosen $a' \sim \pi(\cdot | s_t)$

→ Update $Q(s_t, A_t)$ towards value given by alternative action!

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

best alternative
action

Q-learning (Algorithm).

→ Consider two policies $\left\{ \begin{array}{l} \text{Behaviour} \rightarrow \text{used for exploration} \\ \text{Target} \rightarrow \text{that is learnt} \end{array} \right.$ → improve!!

→ Behaviour policy: ϵ -greedy wrt $Q(s,a)$
explores
exploits

→ Target policy: greedy wrt $Q(s,a)$ always selects 'max'
 $\pi(S_{t+1}) = \arg \max_a Q(S_{t+1}, a)$

→ Q-learning target becomes:

$$R_{t+1} + \gamma Q(S_{t+1}, A')$$

$$= R_{t+1} + \gamma Q(S_{t+1}, \arg \max_{a' \in A} Q(S_{t+1}, a'))$$

$$= R_{t+1} + \max_{a' \in A} \gamma Q(S_{t+1}, a')$$

SARSA (on-policy) Reminder

Initialize $Q(s,a) = 0$ for all $s \in S, a \in A$

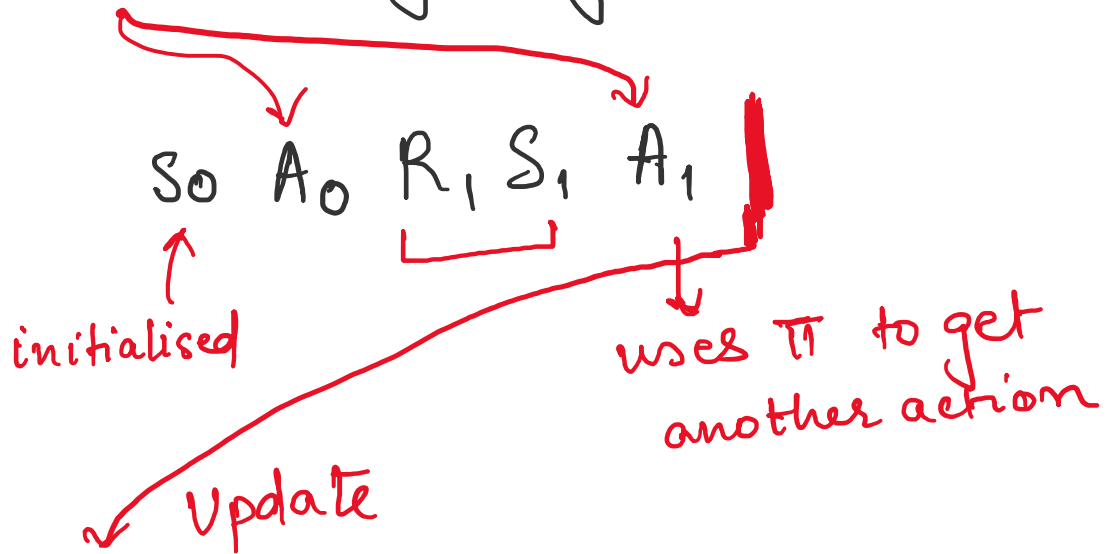
$\pi \leftarrow \epsilon\text{-greedy}(Q)$



SARSA (on-policy) Reminder

Initialize $Q(s,a) = 0$ for all $s \in S, a \in A$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$



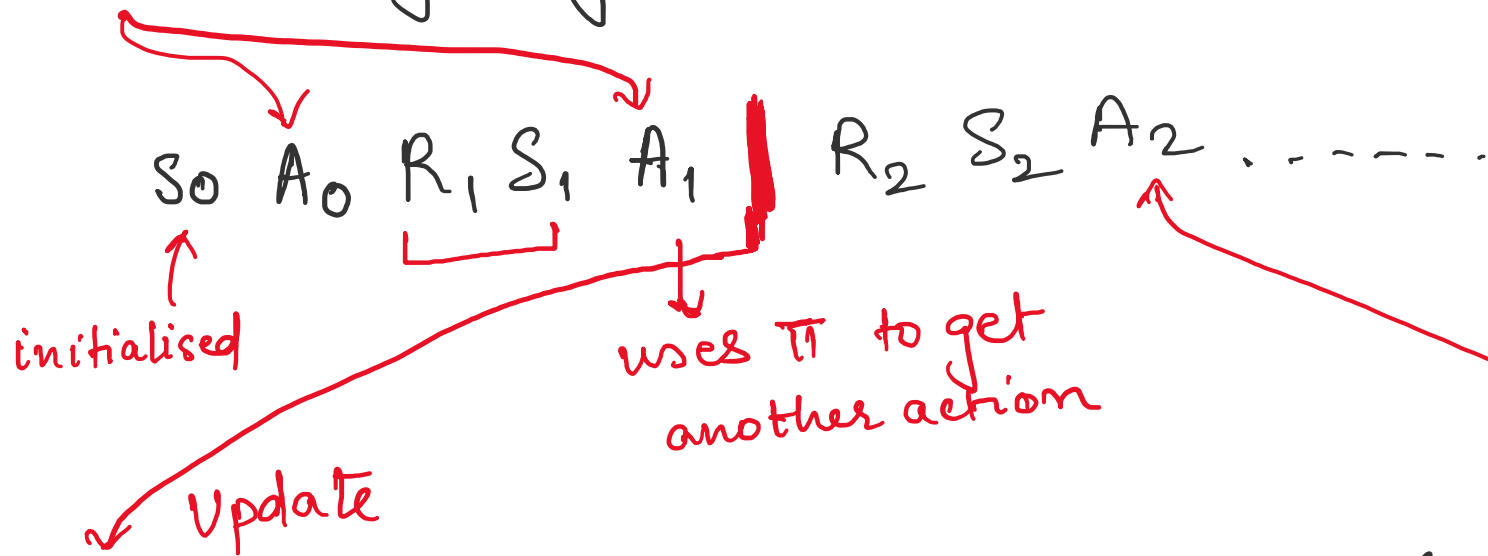
$$Q(S_0, A_0) \leftarrow Q(S_0, A_0) + \alpha (R_1 + \gamma Q(S_1, A_1) - Q(S_0, A_0)) :$$

and improve to most recent update
 $\pi \leftarrow \epsilon\text{-greedy}(Q)$

SARSA (on-policy) Reminder

Initialize $Q(s,a) = 0$ for all $s \in S, a \in A$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$



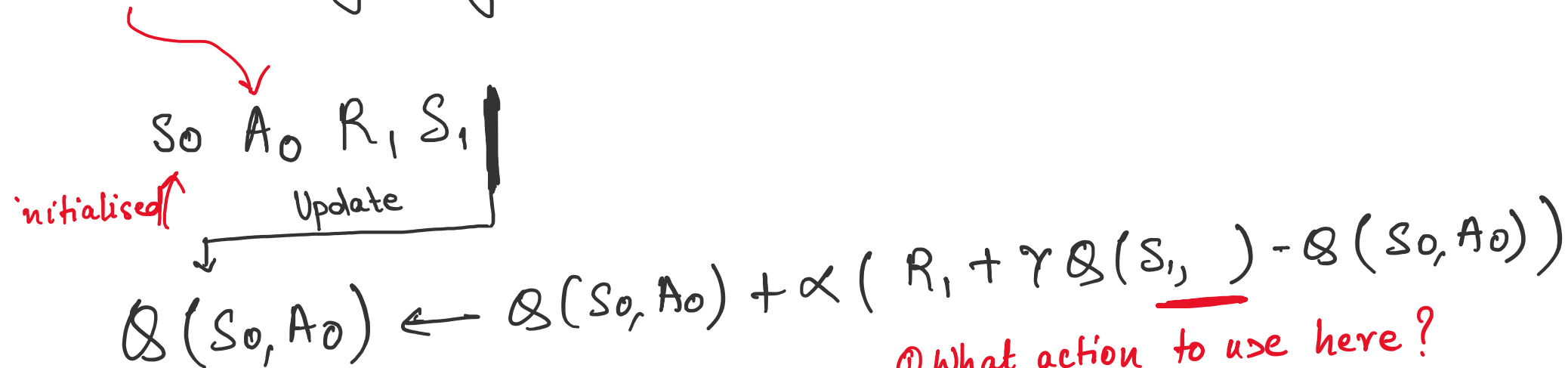
$$Q(S_0, A_0) \leftarrow Q(S_0, A_0) + \alpha (R_1 + \gamma Q(S_1, A_1) - Q(S_0, A_0))$$

and improve to most recent update
 $\pi \leftarrow \epsilon\text{-greedy}(Q)$

Q-learning = SARSA (max)

Initialize $Q(s, a) = 0$ for all $s \in S, a \in A$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$



① What action to use here?

② SARSA uses $\epsilon\text{-greedy}$

③ Here, we could maximise directly
keep $\pi \sim \arg \max_{a \in A} Q(s, a)$

ie. use greedy policy.

Q-learning = SARSA (max)

Initialize $Q(s, a) = 0$ for all $s \in S, a \in A$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

s_0, a_0, R, s_1

initialised $\uparrow$ $\downarrow$ *Update*

$$Q(s_0, a_0) \leftarrow Q(s_0, a_0) + \alpha (R_1 + \gamma \underbrace{Q(s_1, \arg \max_{a' \in A} Q(s_1, a'))}_{\text{keep } \pi \sim \arg \max_{a' \in A} Q(s, a')}} - Q(s_0, a_0))$$

keep $\pi \sim \arg \max_{a' \in A} Q(s, a')$
ie. use greedy policy.

Q-learning = SARSA (max)

Initialize $Q(s, a) = 0$ for all $s \in S, a \in A$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

s_0, a_0, R, s_1

initialised $\uparrow$ Update $\downarrow$

$$Q(s_0, a_0) \leftarrow Q(s_0, a_0) + \alpha \left(R + \gamma \max_{a \in A} Q(s_1, a) - Q(s_0, a_0) \right)$$

keep $\pi \sim \arg \max_{a \in A} (Q(s, a))$
ie. use greedy policy.

Q-learning = SARSA (max)

Initialize $Q(s, a) = 0$ for all $s \in S, a \in A$

$\pi \leftarrow \epsilon\text{-greedy}(Q)$

$s_0, a_0, R, s_1, \text{ } | \text{ } a_1, R, s_1$

initialised $\swarrow$ $\xrightarrow{\text{Update}}$

$$Q(s_0, a_0) \leftarrow Q(s_0, a_0) + \alpha \left(R_1 + \gamma \max_{a \in A} Q(s_1, a) - Q(s_0, a_0) \right)$$

$\pi \leftarrow \epsilon\text{-greedy}(\text{updated } Q)$

Q-learning $\rightarrow$ attempts to directly estimate the optimal value at every time step !!
(max)

(1) The only difference between Sarsa and Q-learning is the action used in the target.

(2) This is Sarsa update equation.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_t \left[\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}_{\text{Sarsa target}} - Q(S_t, A_t) \right]$$

Sarsa error

(3) It uses the action actually taken in the next state to calculate the target.

(4) This one is Q-learning's.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha_t \left[\underbrace{R_{t+1} + \gamma \max_a Q(S_{t+1}, a)}_{\text{Q-learning target}} - Q(S_t, A_t) \right]$$

Q-learning error

(5) Q-learning uses the action with the maximum estimated value in the next state, despite the action actually taken.

Introduction

Markov Decision process

Dynamic Programming for planning
and control

Model Free Prediction and Control
(SARSA and Q-learning)

Function Approximation (Use Deep Neural Networks)

Discrete vs Continuous states

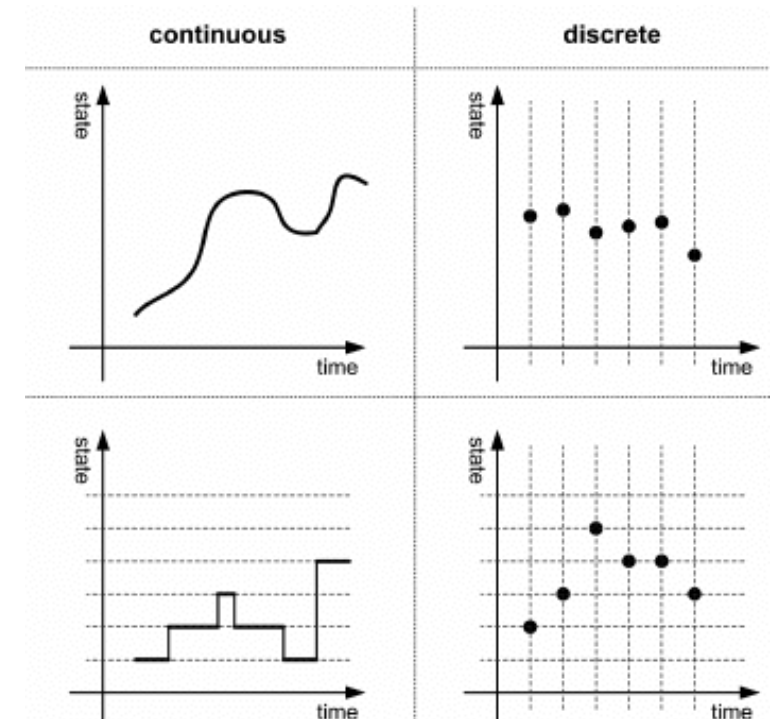
Temporal Difference (TD)

Discrete Space vs Continuous Space

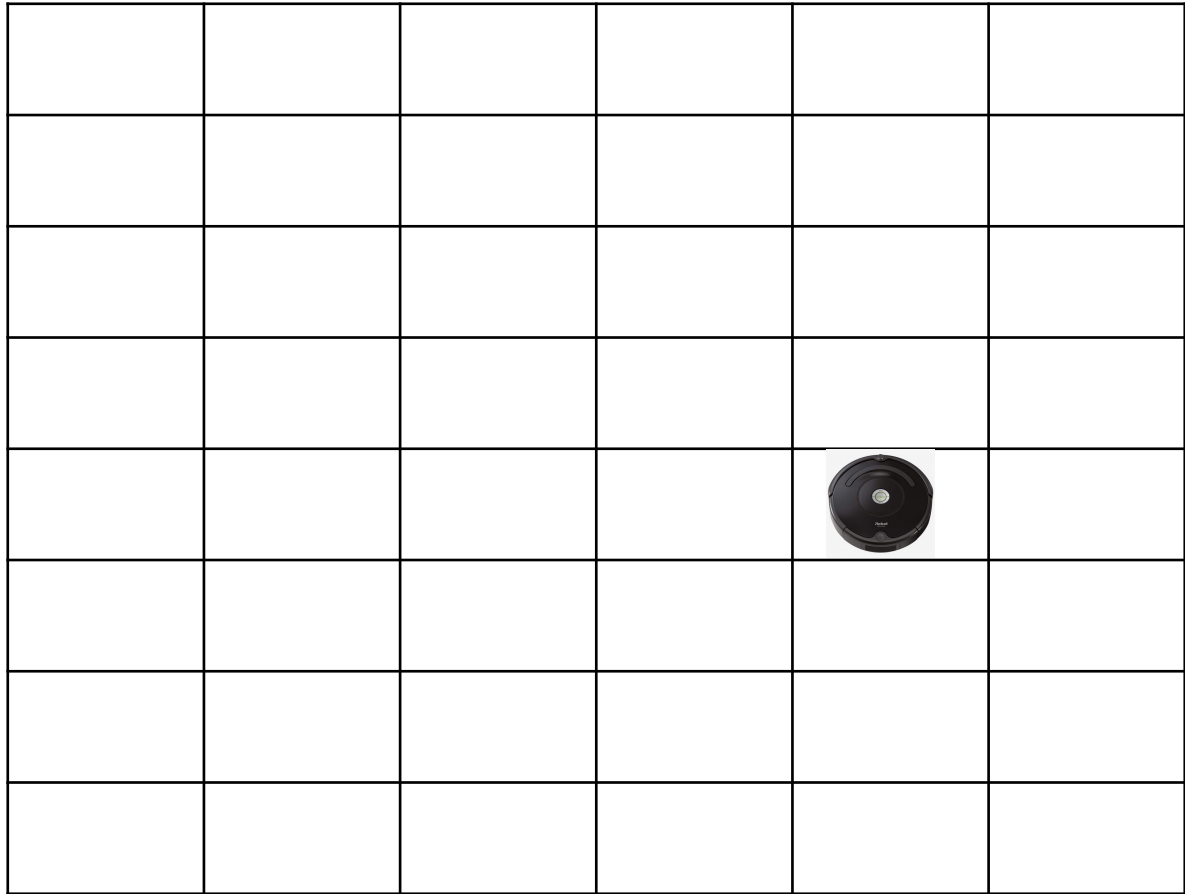


Discrete Space
States: $s \in \{s_0, s_1, \dots, s_m\}$
Actions: $a \in \{a_0, a_1, \dots, a_m\}$

Continuous Spaces
States: $s \in \mathbb{R}^n$
Actions: $a \in \mathbb{R}^n$



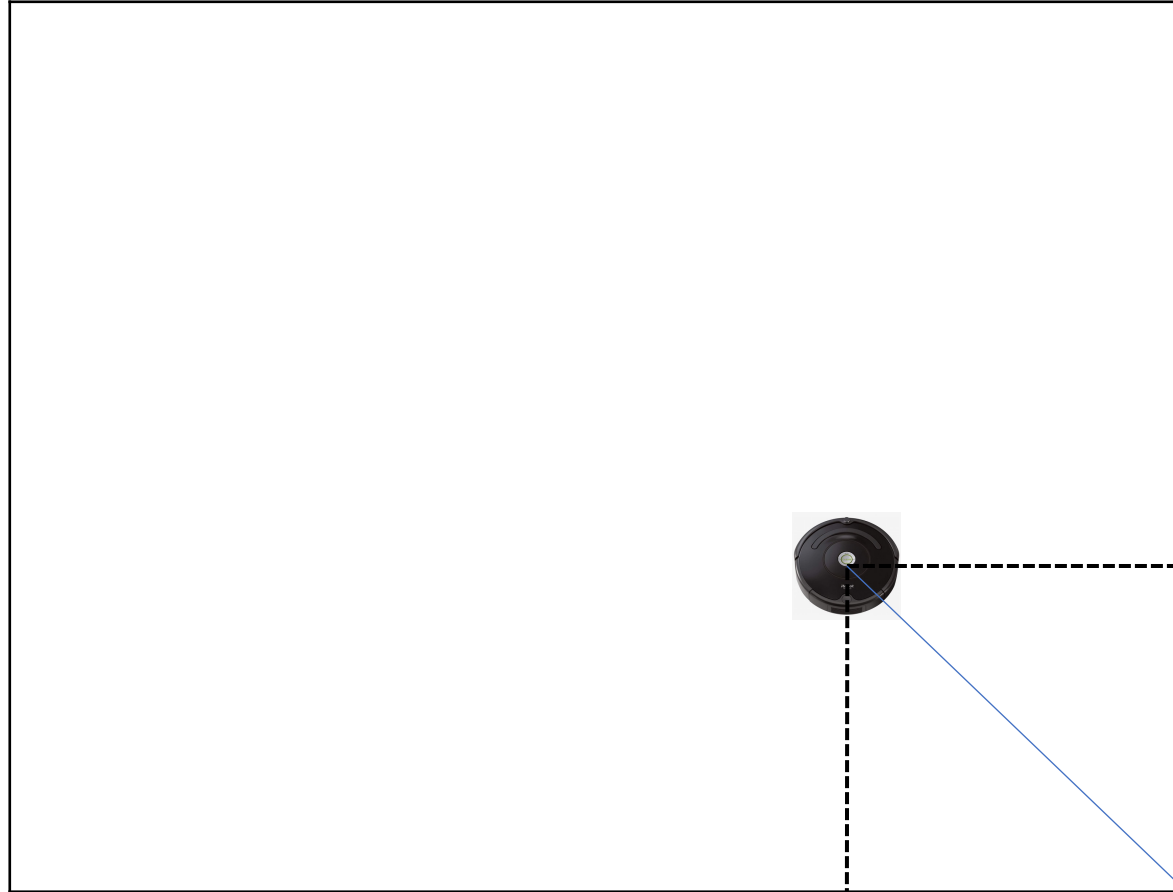
Discrete Space vs Continuous Space



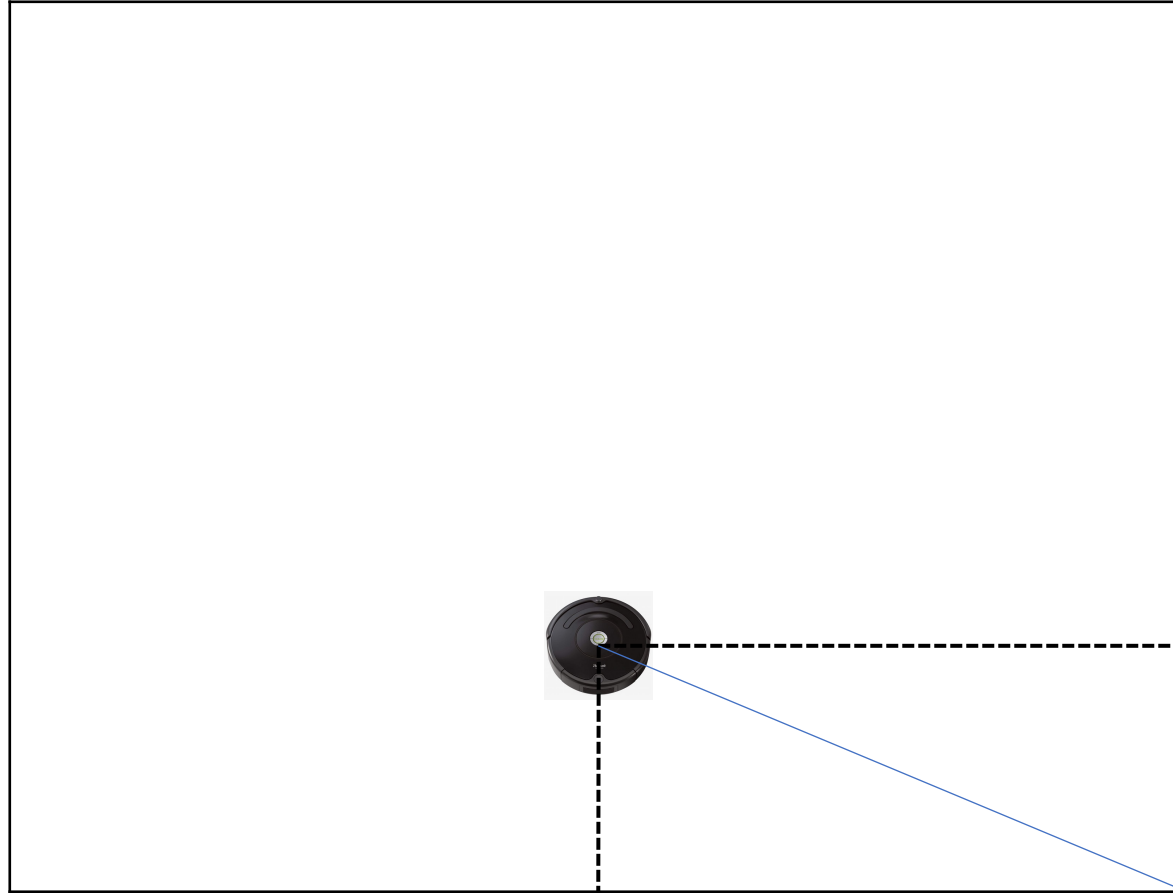
→ In real world,
position
speed
:
:
}

continuous variables.

Discrete Space vs Continuous Space



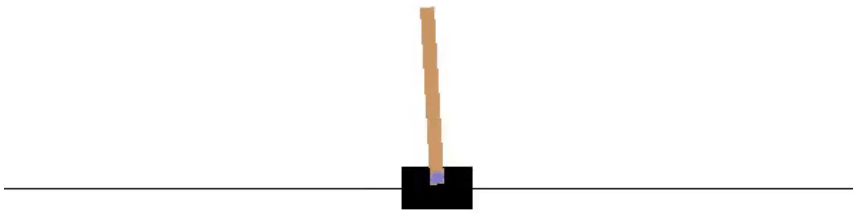
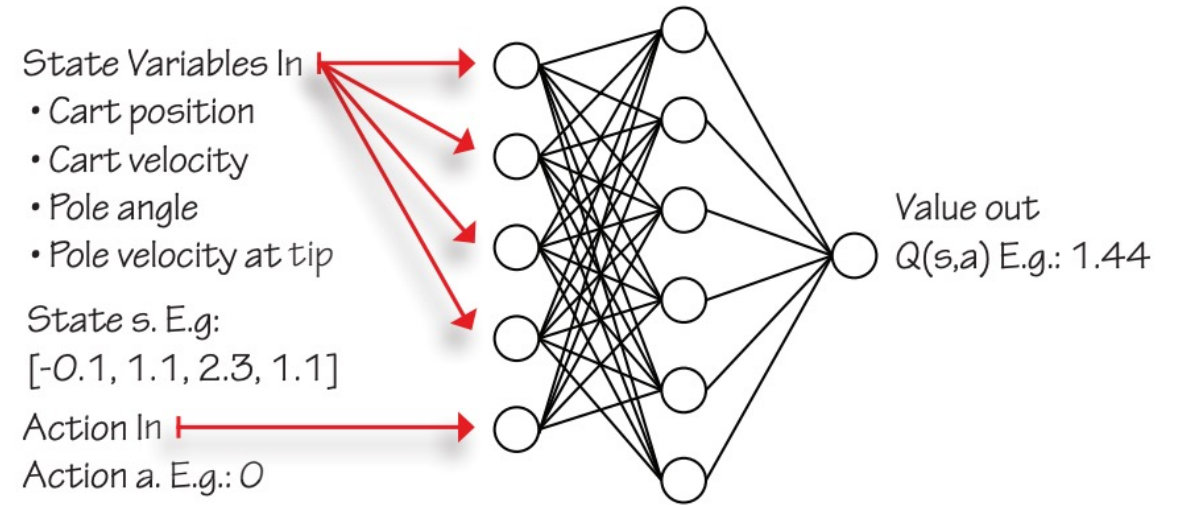
Discrete Space vs Continuous Space



Approximating States/Action....using Neural Networks!



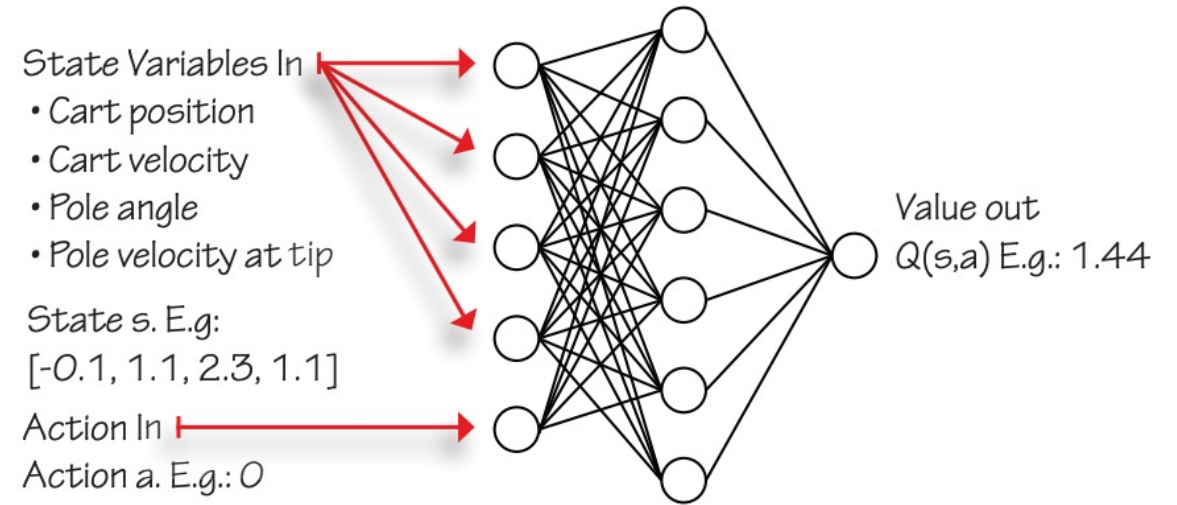
State-action-in-value-out architecture



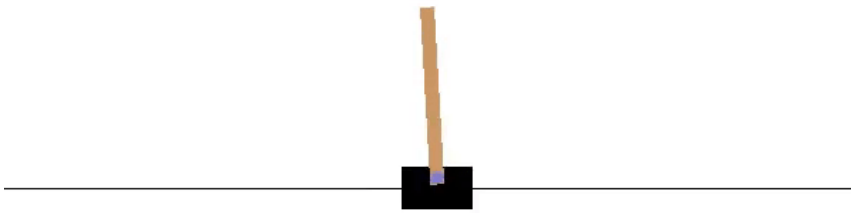
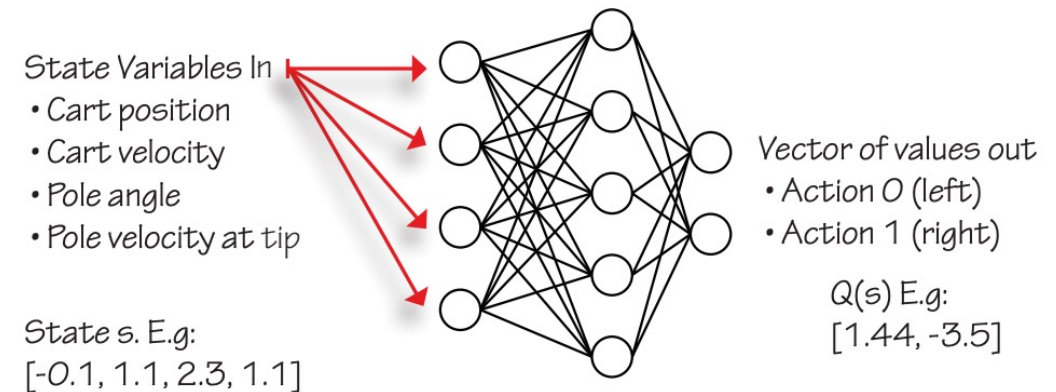
Approximating States/Action...using Neural Networks!



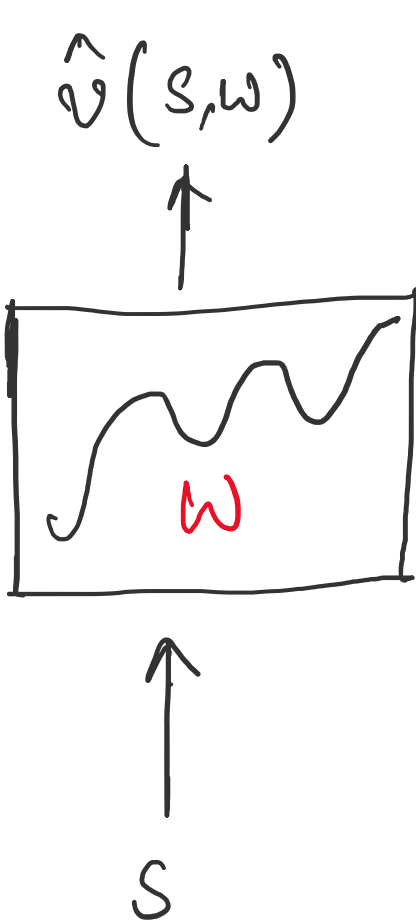
State-action-in-value-out architecture



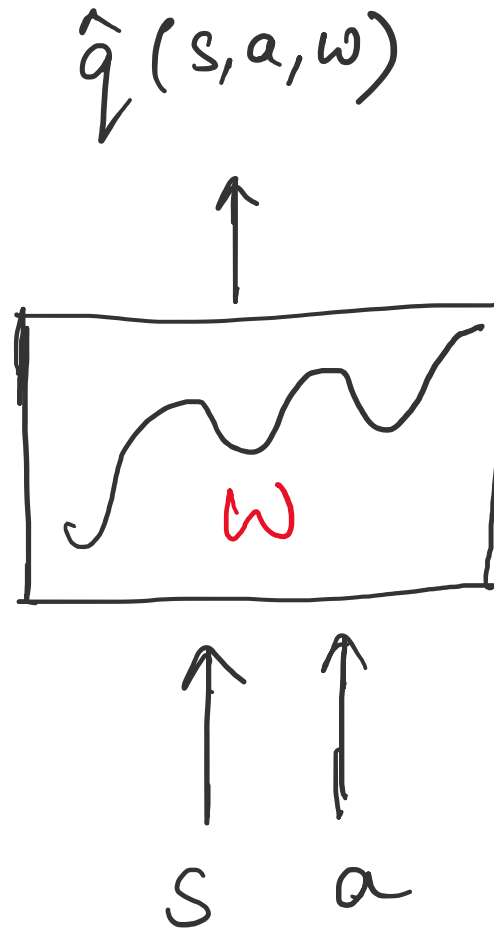
State-in-values-out architecture



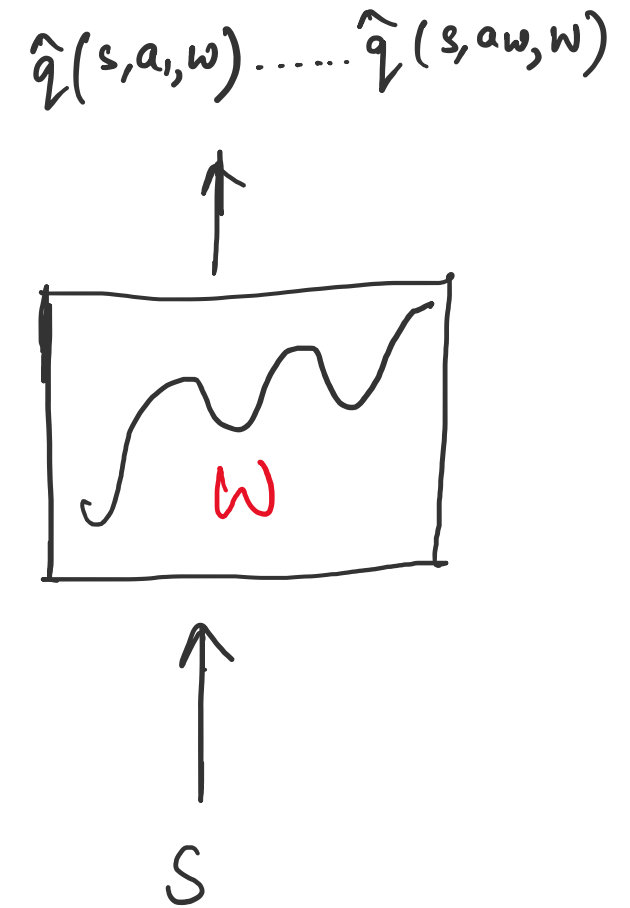
Function Approximation States/Action....using Neural Networks!



State Value Approximation



State-Action Value Approximation



State to different Q Values Values

Function Approximation States/Action....using Neural Networks!

What we want: Close approximations of true functions

$$\hat{v}(s) \approx v_{\pi}(s)$$

$$\hat{q}(s,a) \approx q_{\pi}(s,a)$$

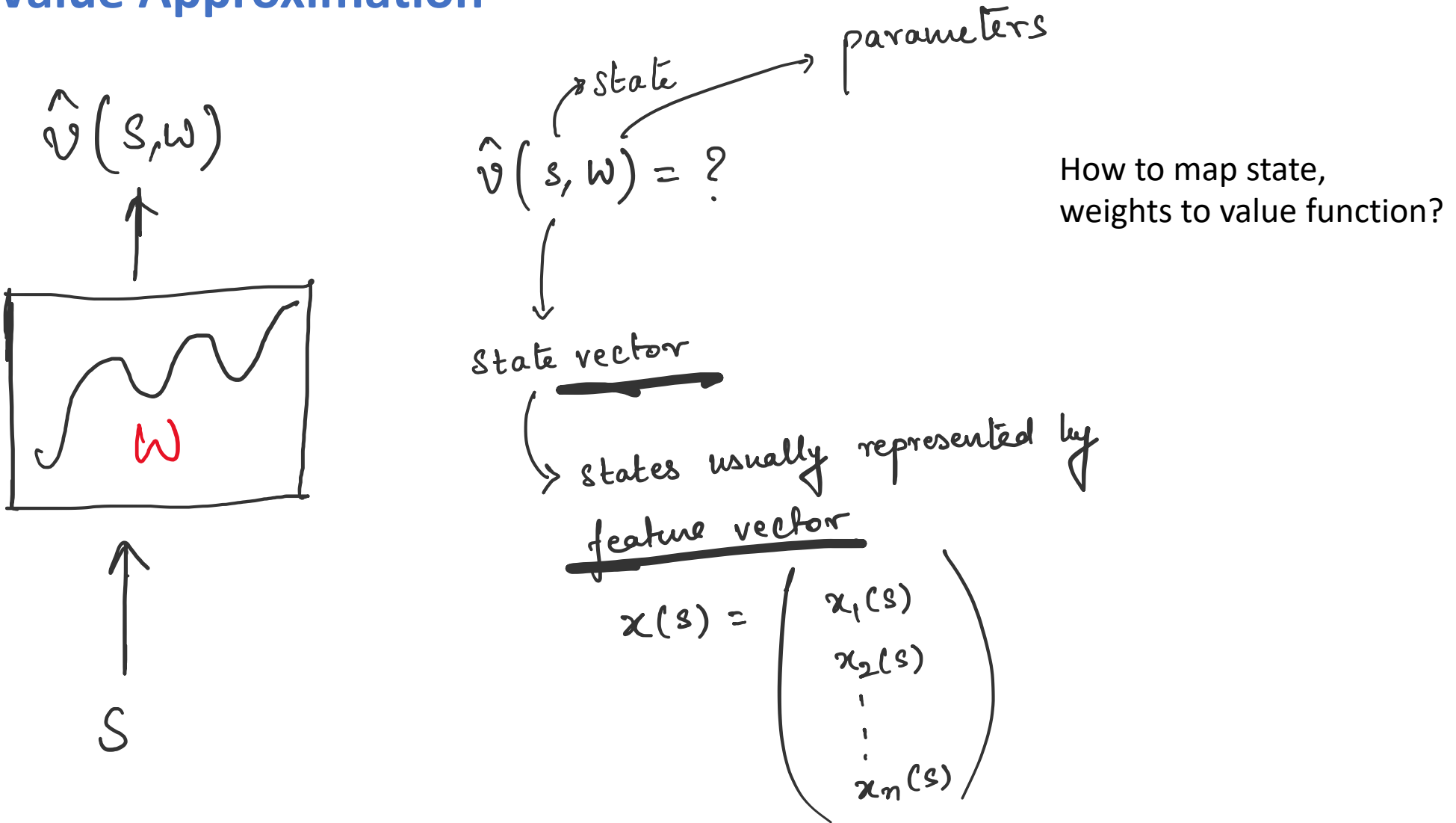
Parametrized by weights w

$$\hat{v}(s, \underline{w}) \approx v_{\pi}(s)$$

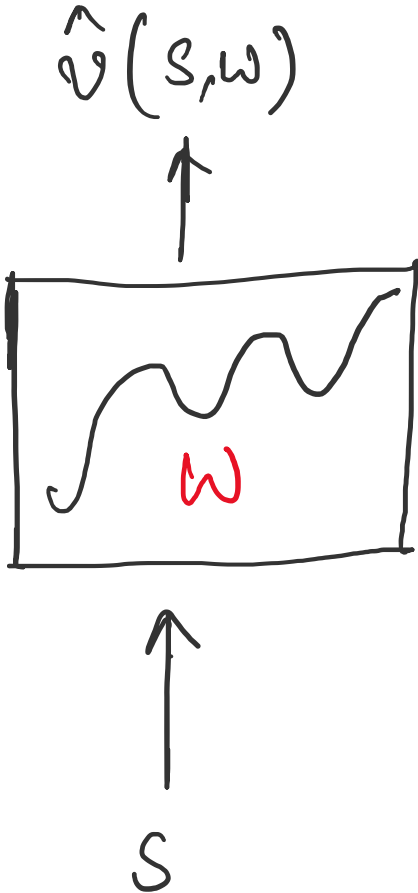
$$\hat{q}(s, a, \underline{w}) \approx q_{\pi}(s, a)$$

Find good ' w ' iteratively !!

State Value Approximation



State Value Approximation



$$\hat{v}(s, w) = \overbrace{X(s)^T \cdot w}^{\text{simplest approach.}}$$

$$= x_1(s) \cdot w_1 + x_2(s) \cdot w_2 + \dots + x_n(s) \cdot w_n$$

state vector

states usually

$$= \sum_{j=1}^n x_j(s) \cdot w_j$$

Linear
function
Approximation

State Value Approximation



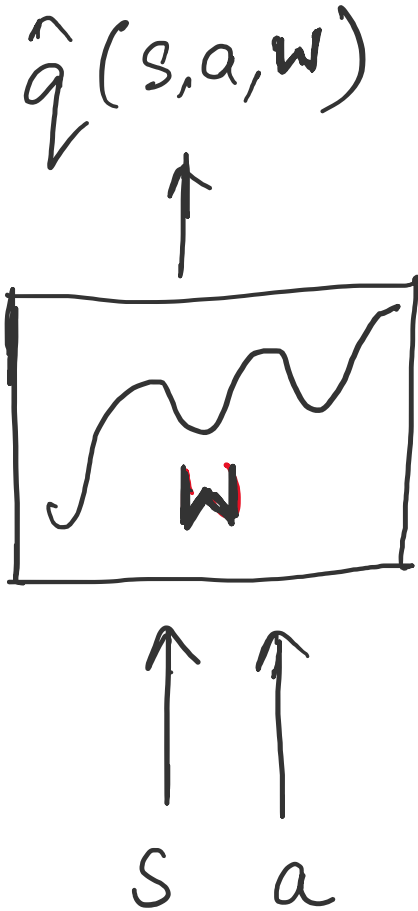
Value Function $\Rightarrow \hat{v}(s, w) = x(s)^T \cdot w$

Minimize Error $\Rightarrow J(w) = \mathbb{E}_{\pi} \left[(v_{\pi}(s) - x(s)^T w)^2 \right]$

Error Gradient $\Rightarrow \nabla_w J(w) = -2 \left(v_{\pi}(s) - x(s)^T \cdot w \right) x(s)$

Update Rule $\Rightarrow \Delta w = -\alpha \frac{1}{2} \nabla_w J(w)$
 $= \alpha \left(v_{\pi}(s) - x(s)^T \cdot w \right) x(s)$

Q-Function Approximation



$$\hat{q}(s, a, w) = ?$$

Action-Value
Function

→ Use Gradient Descent
for weight adaptation !!

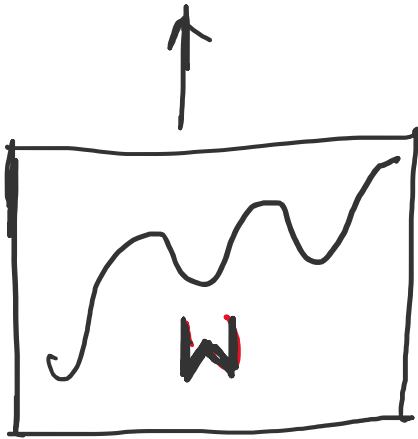
Action vector approximation

$$\chi(s, a) = \begin{pmatrix} \chi_1(s, a) \\ \chi_2(s, a) \\ \vdots \\ \chi_n(s, a) \end{pmatrix}$$

State/Action
feature vector

Q-Function Approximation

$$\hat{q}(s, \underline{a_1}, w), \dots, \hat{q}(s, \underline{a_m}, w)$$



s

$$\hat{q}(s, a_1, w) = ?$$

...

$$\hat{q}(s, a_m, w) = ?$$

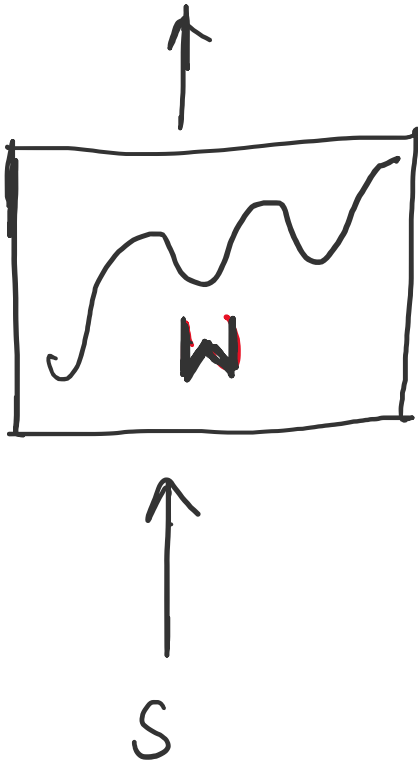
Action vector approximation

$$X(s, a) = \begin{pmatrix} x_1(s, a) \\ x_2(s, a) \\ \vdots \\ x_n(s, a) \end{pmatrix}$$

Q-Function Approximation

Action vector approximation

$$\hat{q}(s, \underline{a_1}, w), \dots, \hat{q}(s, \underline{a_m}, w)$$



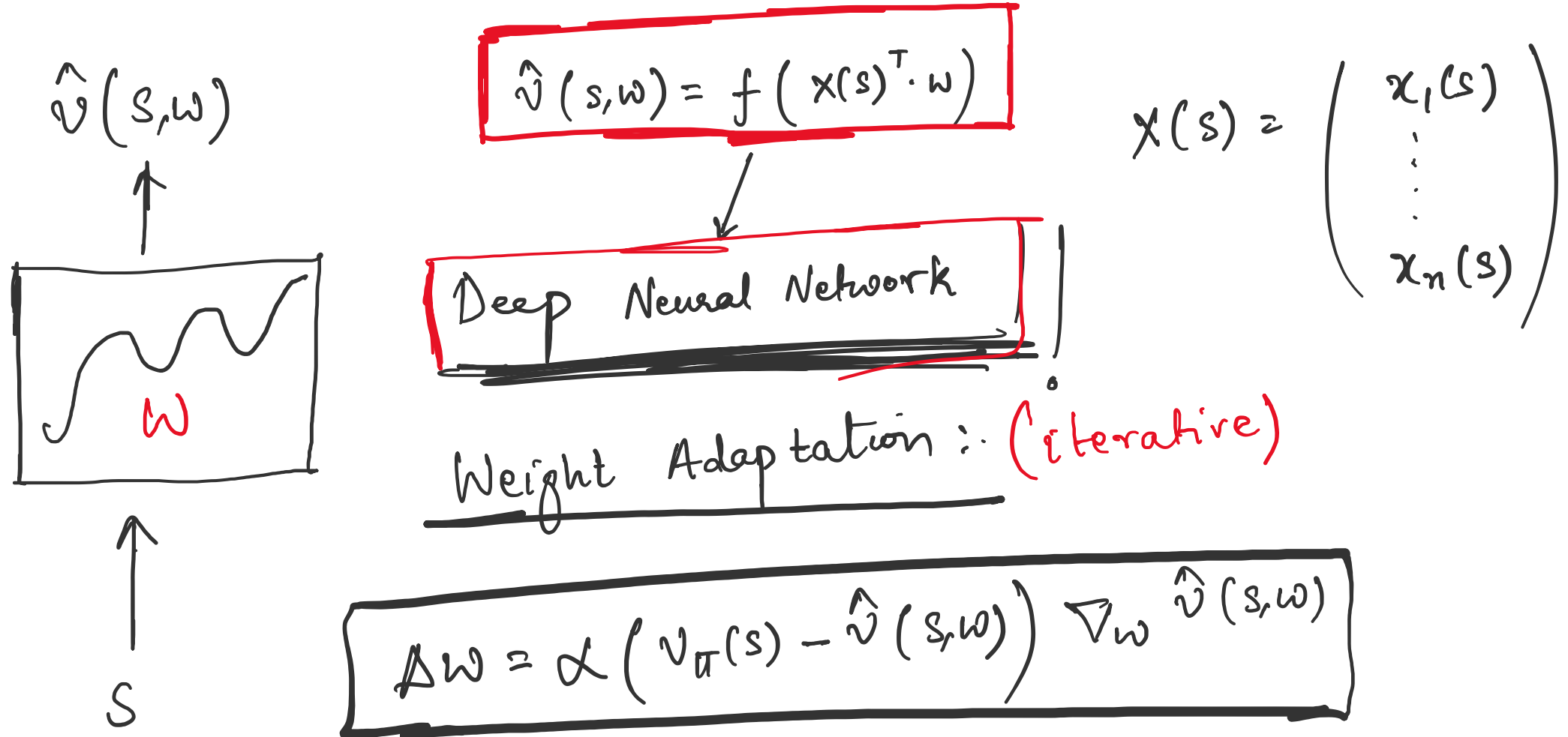
$$\hat{q}(s, a_1, w)$$

$$= \begin{pmatrix} x_1(s, a) & \dots & x_n(s, a) \end{pmatrix} \cdot$$

$$\begin{pmatrix} w_{11} & \dots & w_{1m} \\ \vdots & & \vdots \\ w_{n1} & \dots & w_{nm} \end{pmatrix}$$

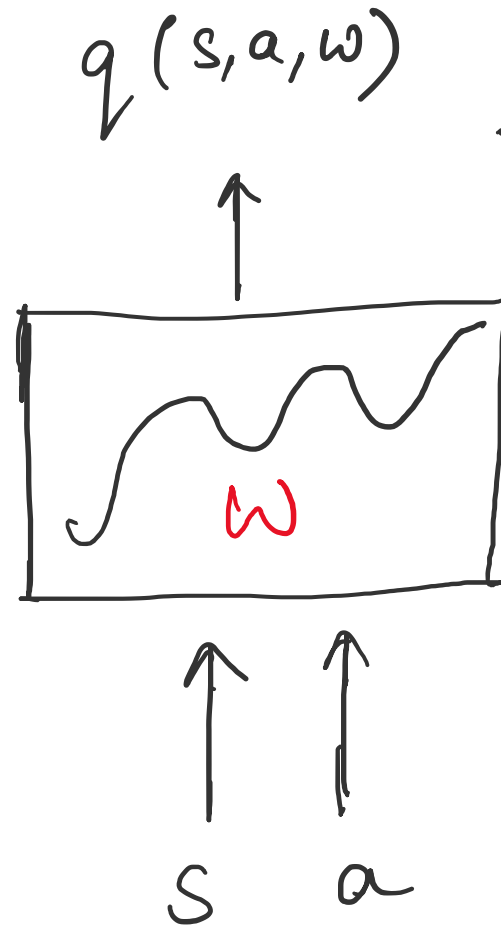
$$= \begin{pmatrix} \hat{q}(s, a_1, w) & \dots & \hat{q}(s, a_m, w) \end{pmatrix}$$

Any Non-Linear Function Approximation

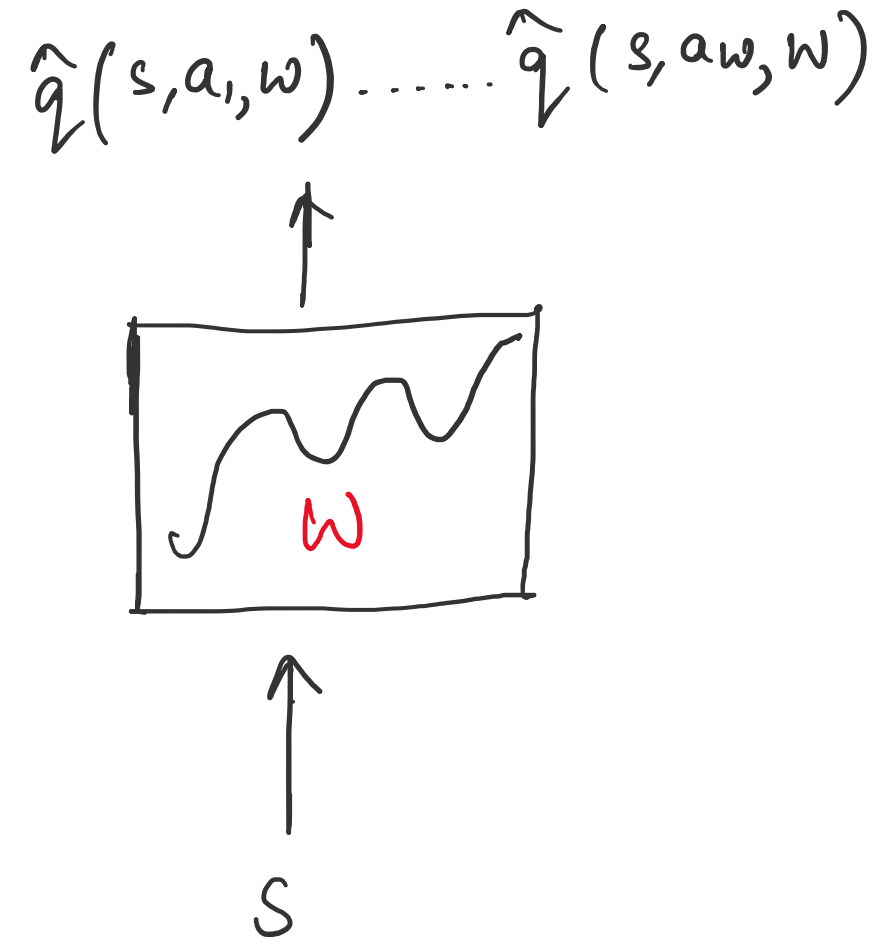


Q-Networks

Generalise from seen states
to Unseen states :



State - Action Value
Approx



State to different Q-values.

Q-Networks

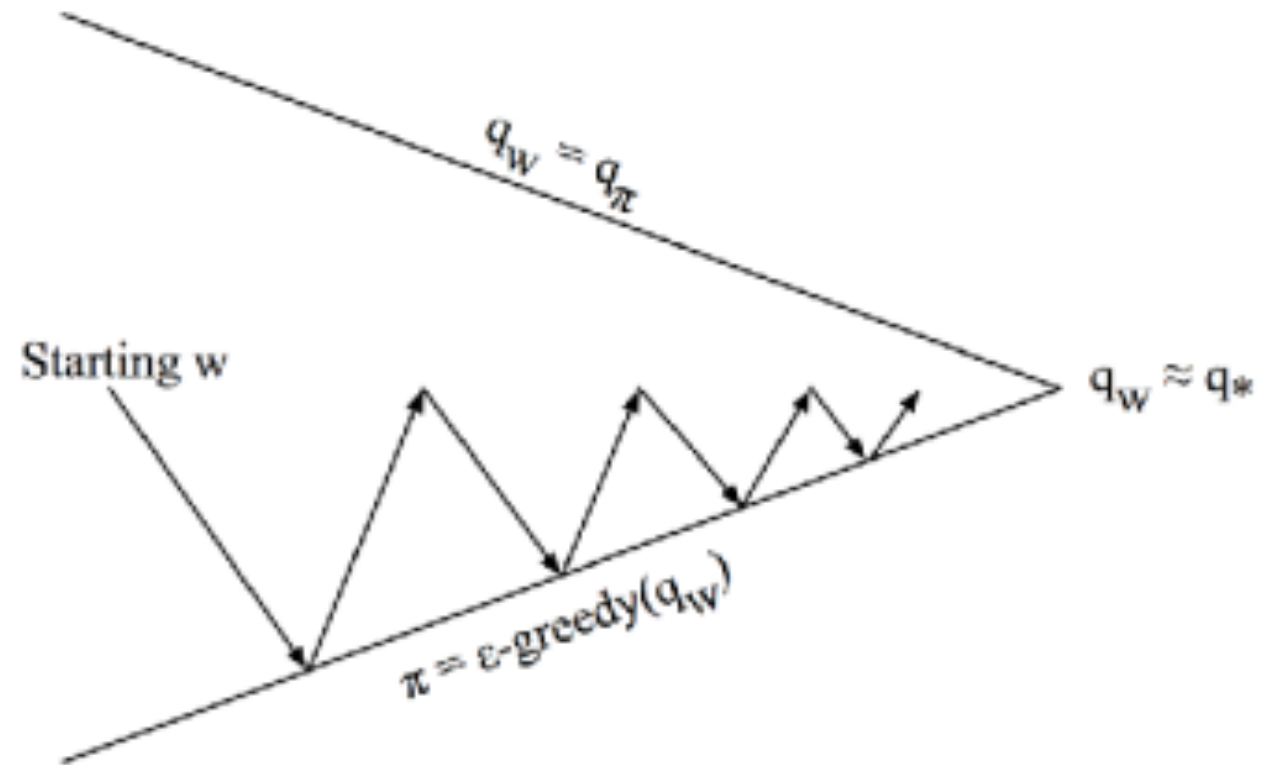
Goal: find parameter vector ' W ' to minimize MSE between
approximate value function
 $\updownarrow$
true value function.

Use TD(0) :

TD target : $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

Weight update:

$$\Delta W = \alpha \left(\underbrace{R_{t+1} + \gamma \hat{Q}(S_{t+1}, A_{t+1}, W)} - \hat{Q}(S_t, A_t, W) \right) \nabla_W \hat{Q}(S_t, A_t, W)$$



Q-Networks

- Optimal Q -values should obey Bellman Equation.

$$Q^*(s,a) = \mathbb{E}_{s'} \left[r + \gamma \max_{a'} Q(s',a')^* \mid s,a \right]$$

- Target (RHS as target)

$$r + \gamma \max_{a'} Q(s',a',w)$$

- Minimize MSE loss by SGD.

$$l = \left(\underbrace{r + \gamma \max_a Q(s',a',w)}_{\text{target}} - Q(s,a,w) \right)^2$$

- Minimize MSE loss by SGD.

$$l = \left(\underbrace{r + \gamma \max_a Q(s', a', w)}_{\text{target}} - Q(s, a, w) \right)^2$$

↳ Converges to Q^* using look up table.

↳ Diverges using Neural networks

→ Correlation between samples

→ Non-stationary targets.

Introduction

Markov Decision process

Dynamic Programming for planning
and control

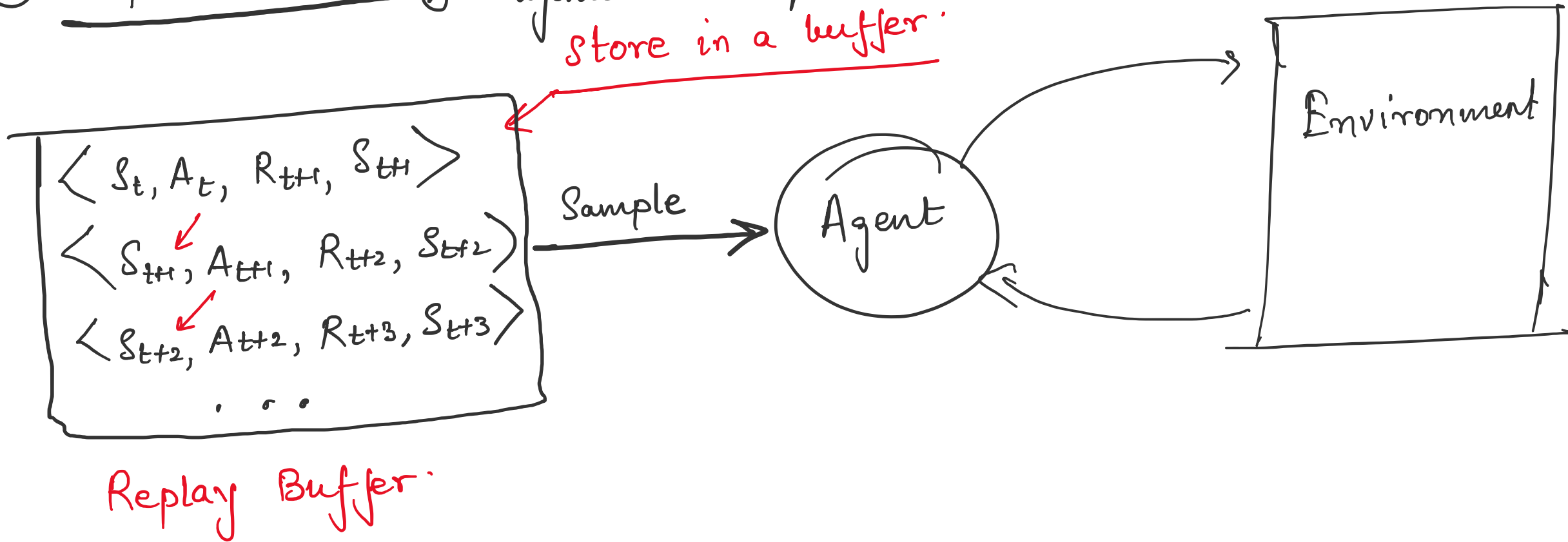
Model Free Prediction and Control
(SARSA and Q-learning)

Function Approximation (Use Deep Neural Networks)

Deep Q networks (Deep learning for RL)

Deep Q-Networks (DQN)

① Experience Replay: Build data set from agent's own experience.
store in a buffer.



Deep Q-Networks (DQN)

① Experience Replay: Build data set from agent's own experience.

$\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$
 $\langle S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2} \rangle$
 $\langle S_{t+2}, A_{t+2}, R_{t+3}, S_{t+3} \rangle$
...

Replay Buffer

Sample
Randomly

Agent

Environment

Breaks correlation.
(between consecutive tuples)

Deep Q-Networks (DQN)

②. Fixed Q-targets.

$$\Delta w = \alpha \left(r + \gamma \max_a \hat{q}(s', a, w) - \hat{q}(s, a, w) \right) \nabla_w \hat{q}(s, a, w)$$

$\nwarrow q_\pi(s, a)$

$$= \alpha \left(q_\pi(s, A) - \hat{q}(s, A, w) \right) \nabla_w \hat{q}(s, A, w)$$

↳ updating ^asame network by same network !!

↳ The target is moving !!

Deep Q-Networks (DQN)

② Fixed Q-targets.

$$\Delta W = \alpha \left(r + \gamma \max_a \hat{Q}(s', a, w) - \hat{Q}(s, a, w) \right) \nabla_w \hat{Q}(s, a, w)$$

$$\Delta W = \alpha \left(Q_{\pi}(s, a) - \hat{Q}(s, a, w) \right) \nabla_w \hat{Q}(s, a, w)$$

Target keeps moving with changing weights!

Deep Q-Networks (DQN)

② Fixed Q-targets.

$$\Delta w = \alpha \left(r + \gamma \max_a \hat{q}(s', a, \bar{w}) - \hat{q}(s, a, w) \right) \nabla_w \hat{q}(s, a, w)$$

fixed (hold weights fixed for some updates)



Use a Target Q network

Decouple update and target TD

$$\Delta w \quad \quad r + \gamma \max_a \hat{q}(s', a, \bar{w})$$

Deep Q-Networks (DQN) Algorithm

- Initialize replay memory D with capacity N .
- Initialize action-value function $\hat{q}$ with random weights $\underline{w}$.
- Initialize targets action-value weights $\bar{w} \leftarrow w$.
- for the episode $e \leftarrow 1$ to M :
 - initial input state x_1
 - prepare initial state: $s \leftarrow \phi(\langle x_1 \rangle)$
 - for time step $t \leftarrow 1$ to T :
 - choose action A from state s using policy $\pi \leftarrow \epsilon\text{-greedy}(\hat{q}(s, A, w))$
 - Take action A , observe reward r , next state x_{t+1}
 - Prepare next state: $s' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
 - Store experience tuple (s, A, R, s') in replay memory D
 - $s \leftarrow s'$

Deep Q-Networks (DQN) Algorithm

- Initialize replay memory $\mathcal{D}$ with capacity N .
- Initialize action-value function $\hat{q}$ with random weights $\underline{w}$.
- Initialize targets action-value weights $\bar{w} \leftarrow w$.
- for the episode $e \leftarrow 1$ to M :

- initial input state x_1
- prepare initial state: $s \leftarrow \phi(\langle x_1 \rangle)$
- for time step $t \leftarrow 1$ to T :

Sample

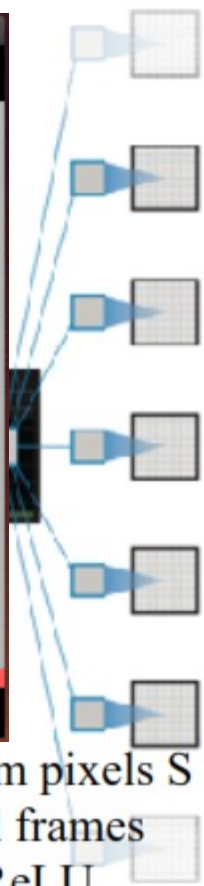
- choose action A from state s using policy $\pi \leftarrow \epsilon\text{-greedy}(\hat{q}(s, A, w))$
- Take action A , observe reward r , next state x_{t+1}
- Prepare next state: $s' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
- Store experience tuple (s, A, R, s') in replay memory $\mathcal{D}$
- $s \leftarrow s'$

Learn

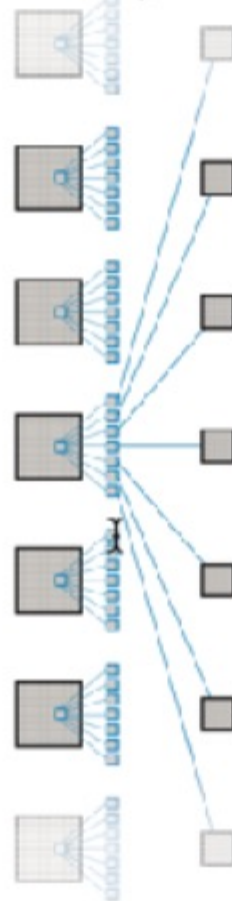
- Obtain random minibatches of tuples (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
- Set target $y_j = r_j + \max_a \hat{q}(s_{j+1}, a, w)$
- Update: $\Delta w = \alpha (y_j - \hat{q}(s_j, a_j, w)) \nabla_w \hat{q}(s_j, a_j, w)$
- Every 'C' steps, reset: $\bar{w} \leftarrow w$.



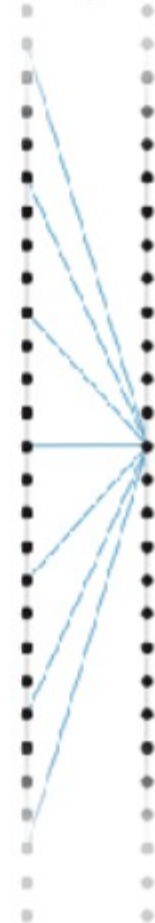
Convolution



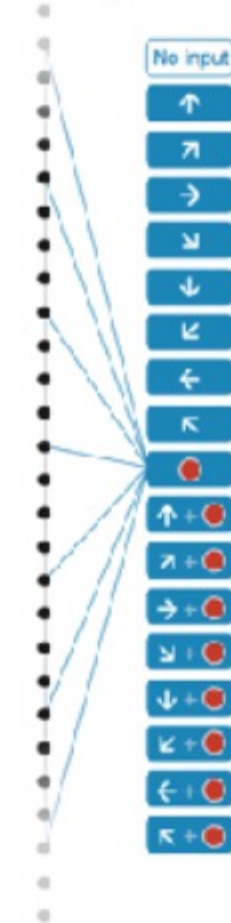
Convolution



Fully connected



Fully connected



end-to-end learning $Q(S,a)$ from pixels S
input state: 4 stacked raw pixel frames
32 8x8 filters with stride 4 + ReLU
64 4x4 filters with stride 2 + ReLU
64 3x3 filters with stride 1 + ReLU
fully connected 512 units + ReLU
fully connected output units, one per action

[Source: Mnih et al., Nature 2015 (DeepMind)]

Introduction

Markov Decision process

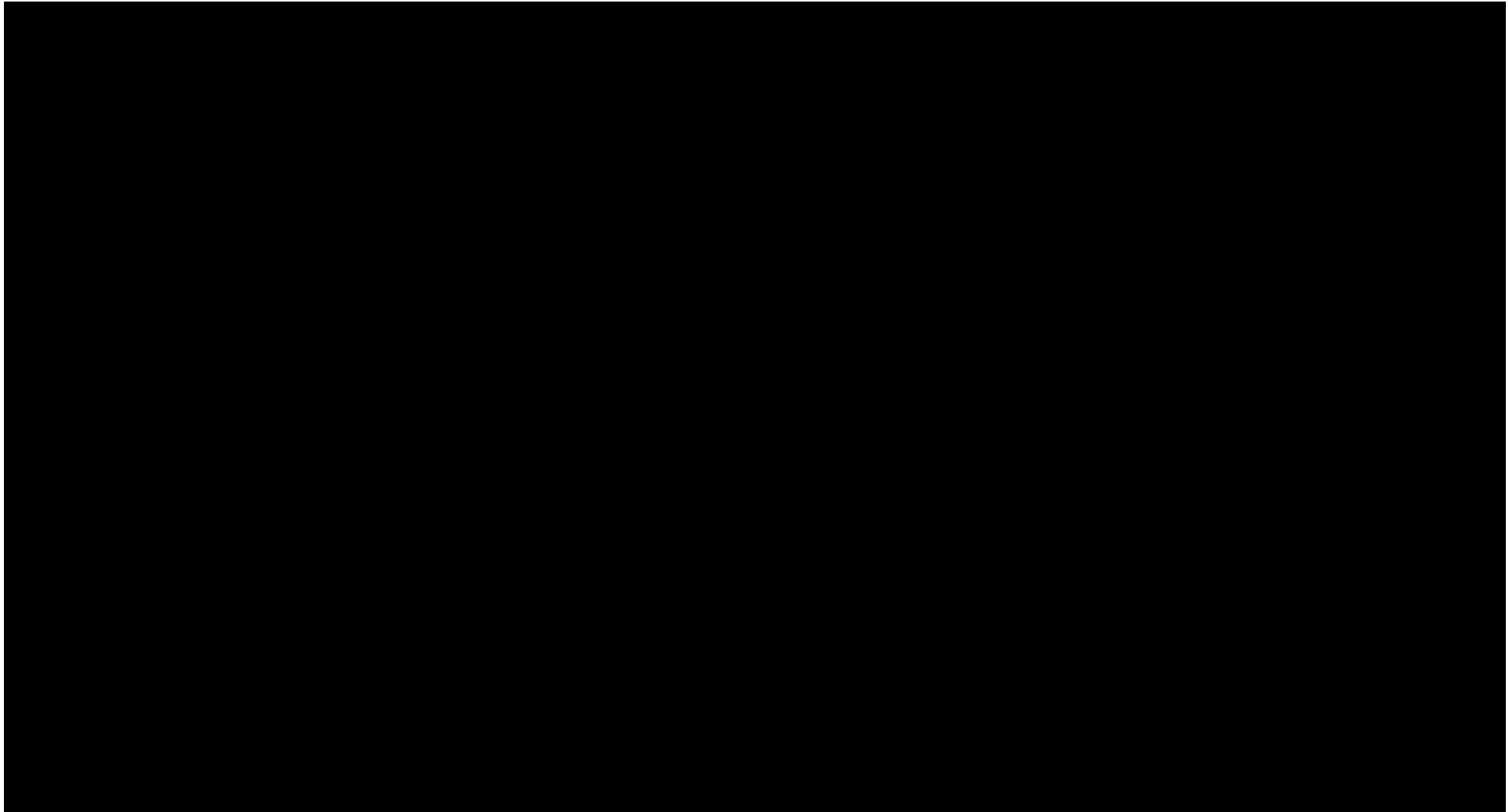
Dynamic Programming for planning
and control

Model Free Prediction and Control
(SARSA and Q-learning)

Function Approximation (Use Deep Neural Networks)

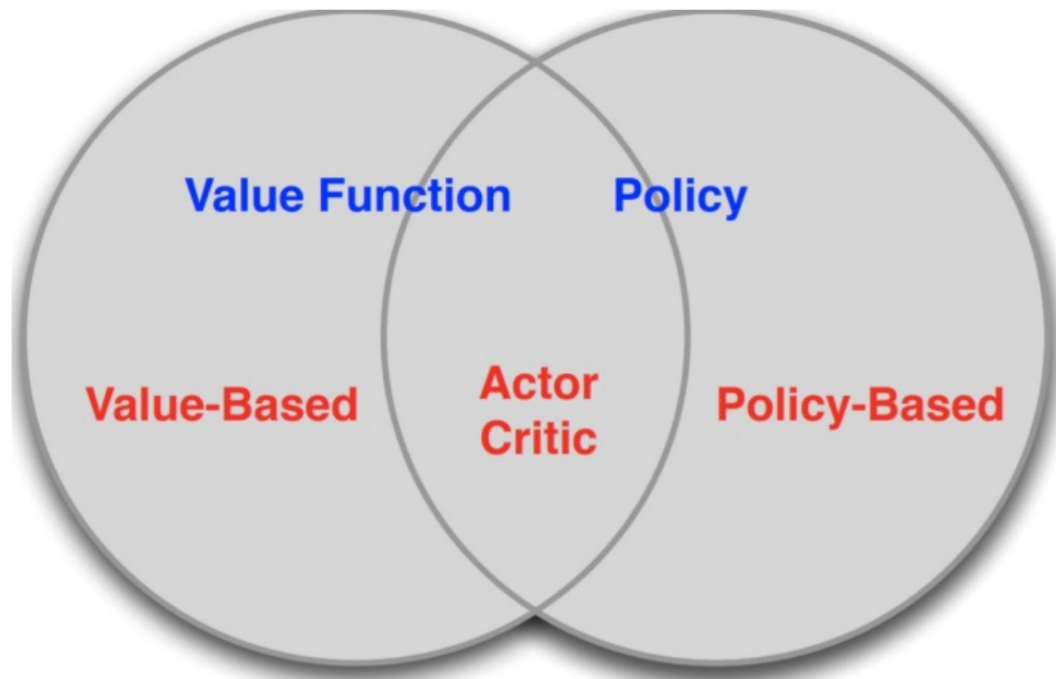
Deep Q networks (deep learning for RL)

Deep Deterministic Policy Gradient Method (DDPG)





Actor Critic Approach to Solve RL using Q-functions



- Value Based
 - Learn Value function
 - Implicit policy
- Policy Based
 - No Value function
 - Learn policy (control law)
- Actor Critic
 - Learn Value function
 - guides learn of control policy
 - Learn control policy

Neural Networks: To approximate Value and Control Law

Actor-critic algorithms maintain two sets of parameters.

- **Critic NN** used to estimate the action-value function:

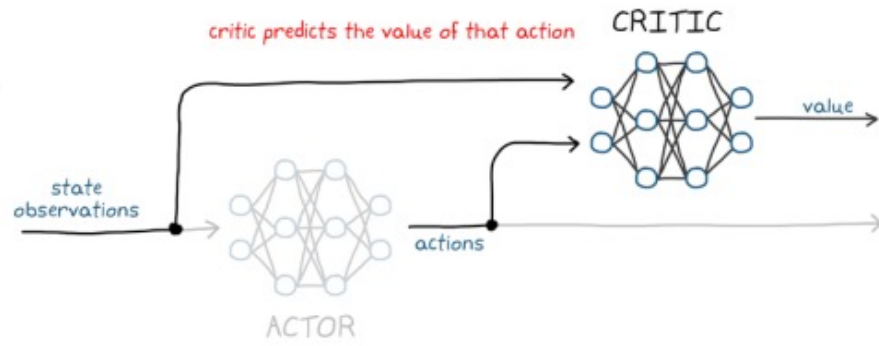
$$Q^w(s, a) \approx Q^{\pi_\theta}(s, a)$$

- **Actor NN** Updates policy parameters θ , in direction suggested by critic.

Actor-critic algorithms follow an approximate policy gradient

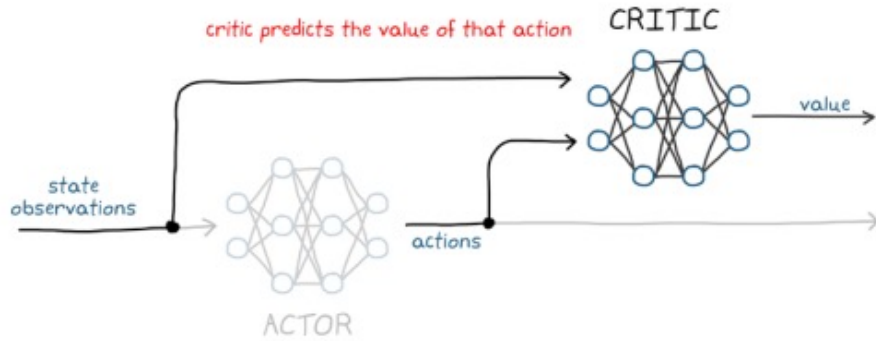
$$\nabla_\theta J(\theta) \approx E_{\pi_\theta} [\nabla_\theta \log \pi_\theta(x, u) Q^w(x, u)]$$

1

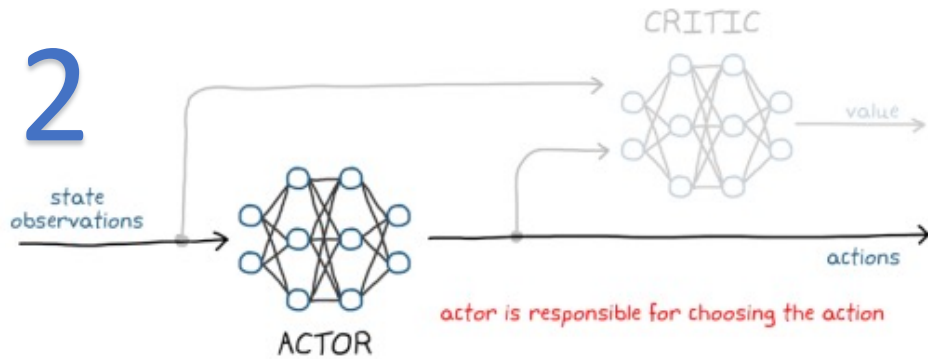


Actor –Critic approach

1



2



Actor –Critic approach

Critic Loss:

$$L(\phi) = \mathbb{E} [(y - Q(s, a|\phi))^2]$$

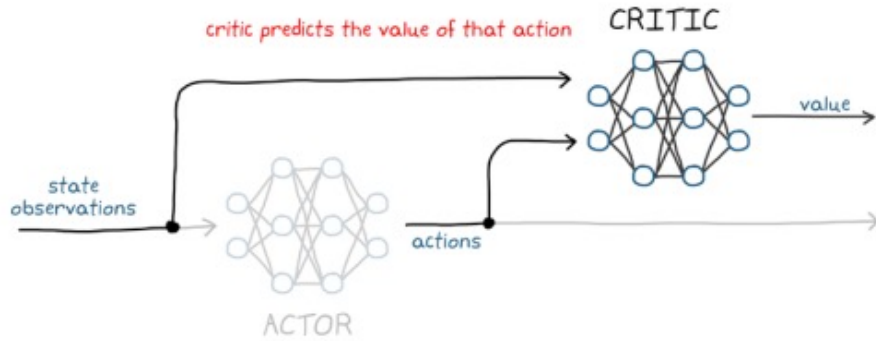
where

$$y = r + \gamma Q'(s', \mu'(s')|\phi')$$

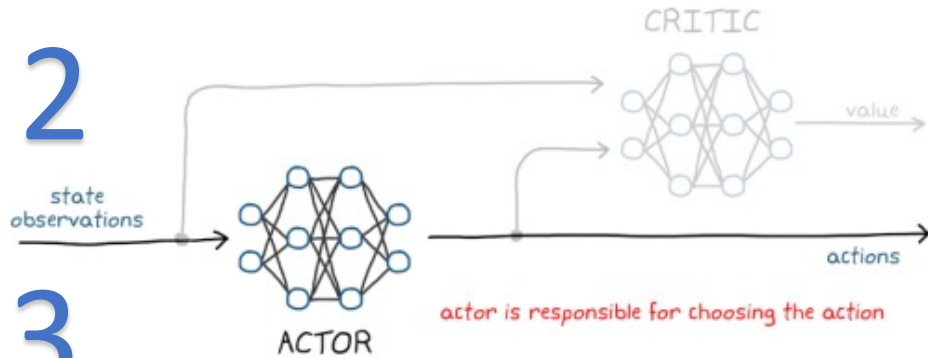
Policy Gradient (Actor):

$$\nabla_{\theta} J \approx \mathbb{E} [\nabla_{\theta} \mu(s|\theta) \nabla_a Q(s, a|\phi)|_{a=\mu(s)}]$$

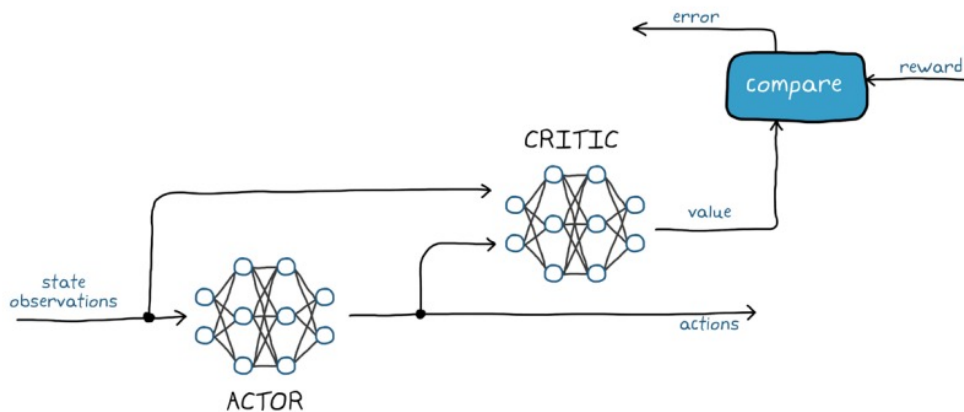
1



2



3



Actor –Critic approach

Critic Loss:

$$L(\phi) = \mathbb{E} [(y - Q(s, a|\phi))^2]$$

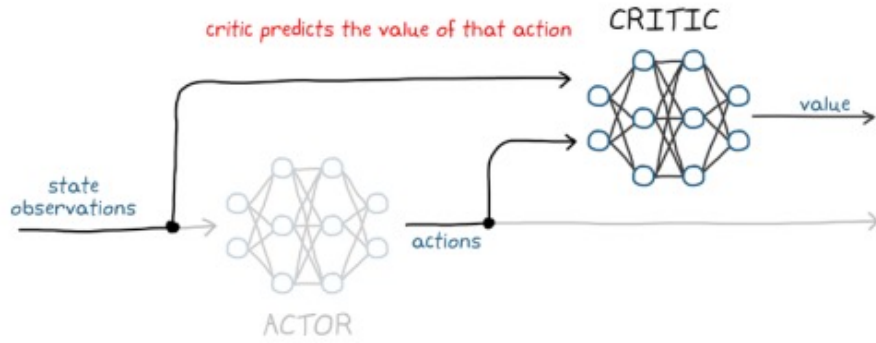
where

$$y = r + \gamma Q'(s', \mu'(s')|\phi')$$

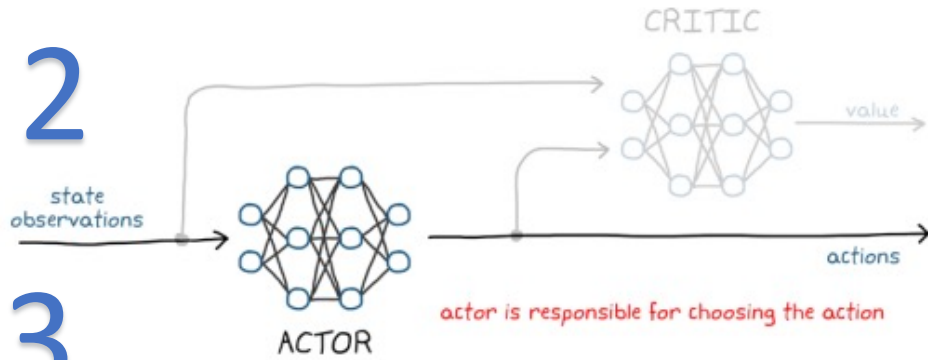
Policy Gradient (Actor):

$$\nabla_{\theta} J \approx \mathbb{E} [\nabla_{\theta} \mu(s|\theta) \nabla_a Q(s, a|\phi)|_{a=\mu(s)}]$$

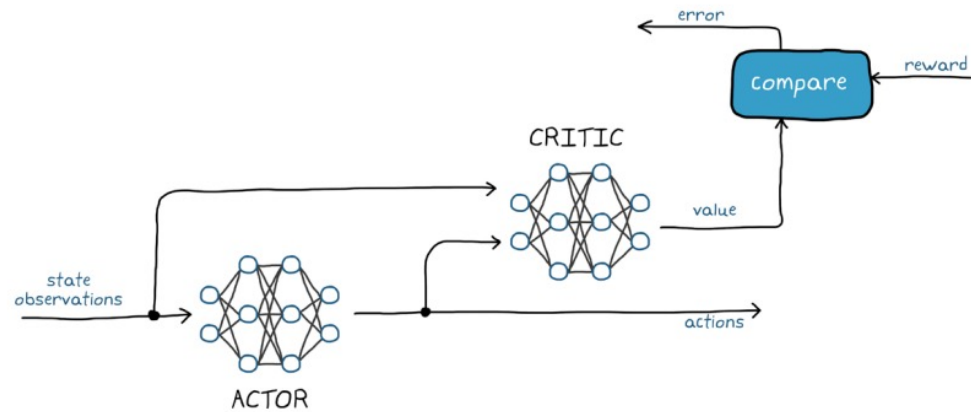
1



2

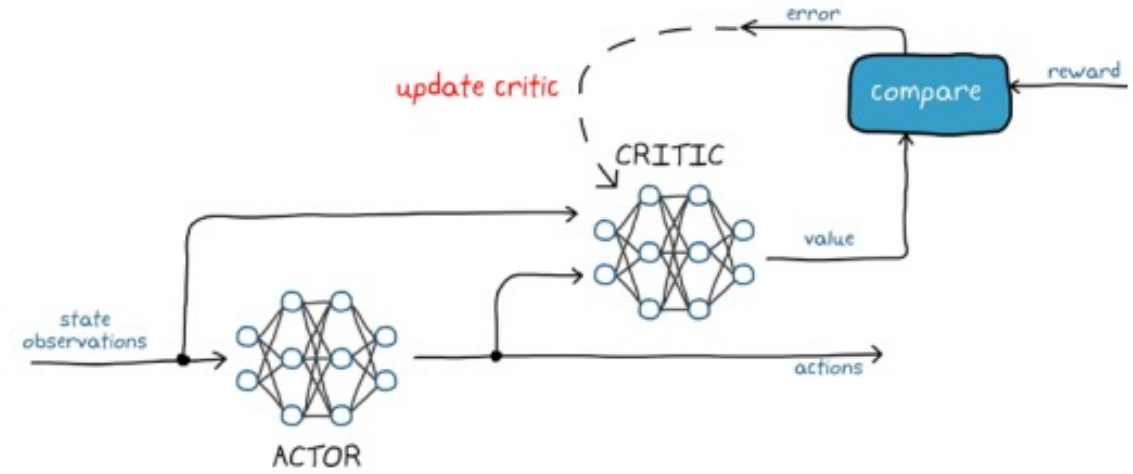


3

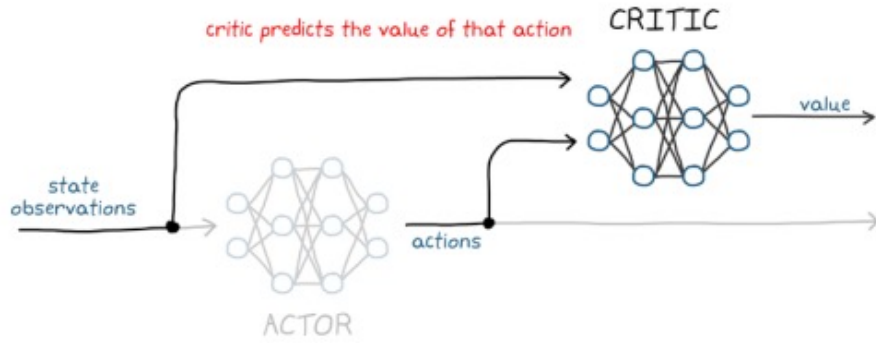


Actor –Critic approach

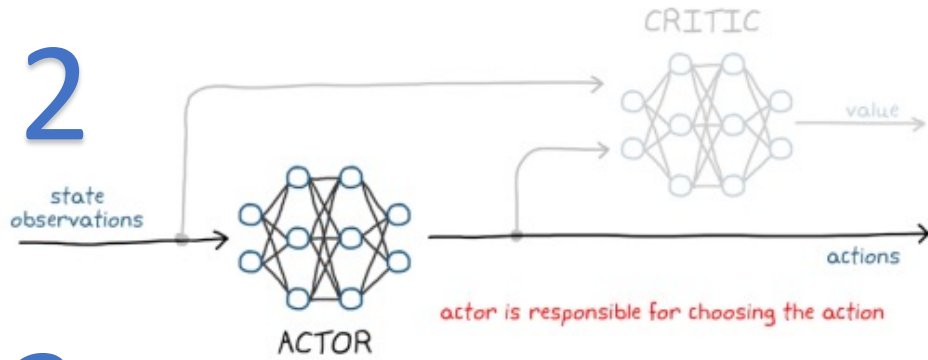
4



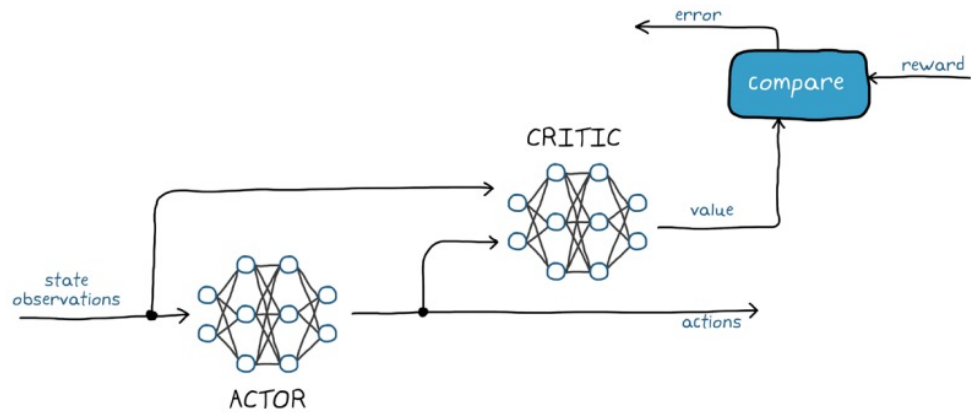
1



2

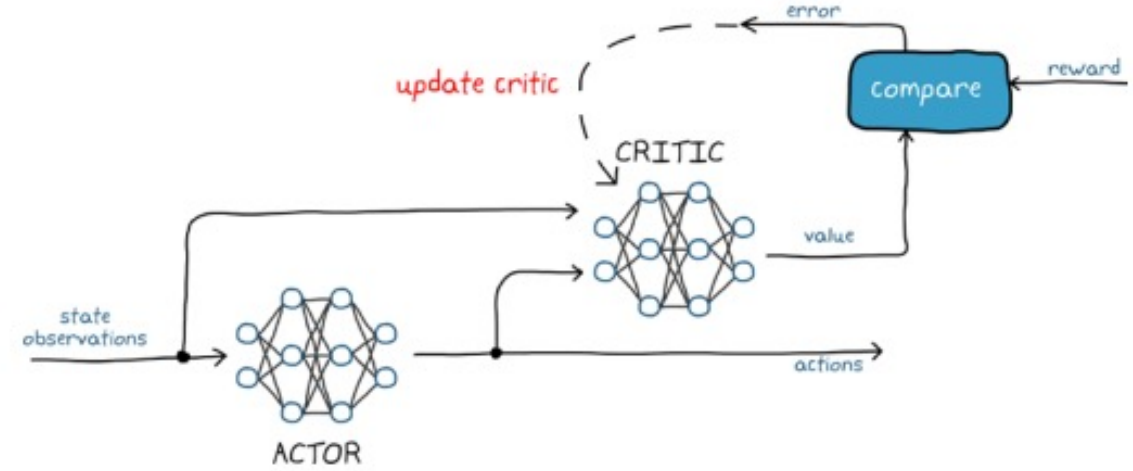


3

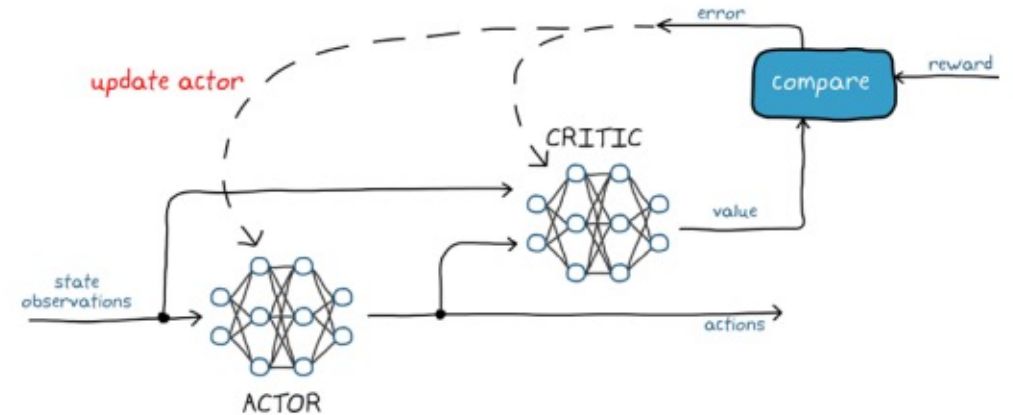


4

Actor –Critic approach



5

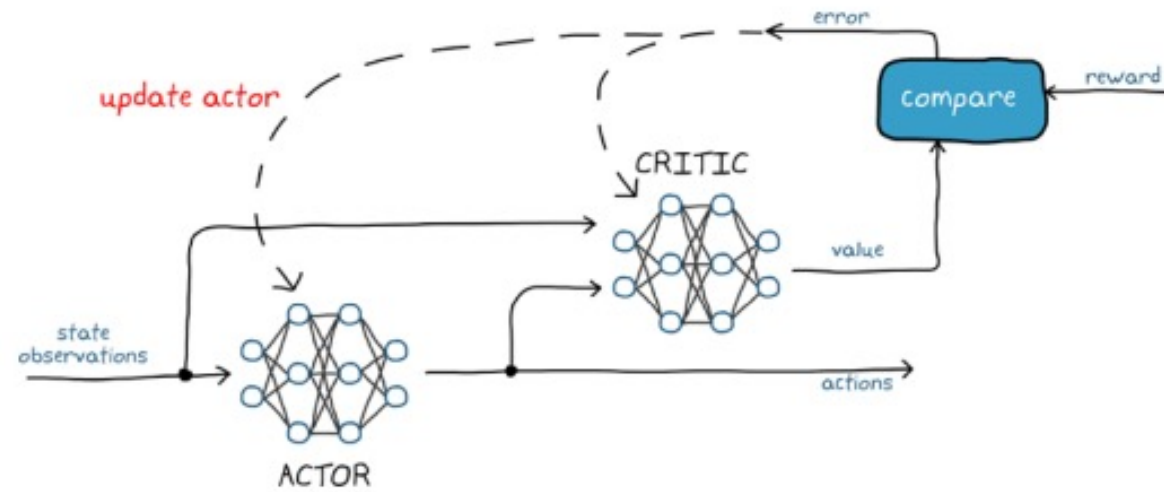


Deep Deterministic Policy Gradient (DDPG)

- **Actor:** Approximates policy function, deterministic $\mu_{\theta}(s)$.
- **Critic:** Approximates value function, $Q(s, a|\phi)$.

Core Idea:

- Actor improves policy based on Critic's evaluation.
- Critic provides gradient to guide policy improvement.



Deep Deterministic Policy Gradient (DDPG)

Critic Loss:

$$L(\phi) = \mathbb{E} [(y - Q(s, a|\phi))^2]$$

where

$$y = r + \gamma Q'(s', \mu'(s')|\phi')$$

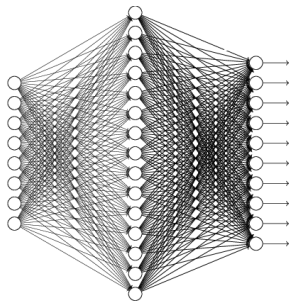
Policy Gradient (Actor):

$$\nabla_{\theta} J \approx \mathbb{E} [\nabla_{\theta} \mu(s|\theta) \nabla_a Q(s, a|\phi)|_{a=\mu(s)}]$$

Basic Steps of DDPG

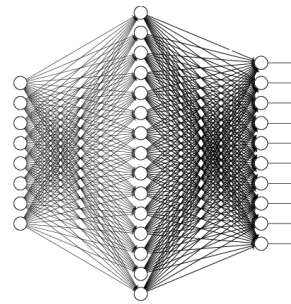
- 1 Initialize Actor and Critic networks.
- 2 Initialize target networks with same parameters.
- 3 Repeat for each episode:
 - Observe state s and select action $a = \mu(s|\theta) + \text{noise}$.
 - Execute a , observe reward r and new state s' .
 - Store transition (s, a, r, s') in replay buffer.
 - Sample batch from buffer to update Critic and Actor.
 - Update target networks slowly:

Regular Network



Actor or Critic

Target Network.



Actor or Critic

Every step, blend in
0.01%
of regular weights
with target
weights.

$$\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$$

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

- **Experience Replay:** Stabilizes learning.
- **Ornstein-Uhlenbeck Noise:** Encourages exploration in continuous space.
- **Target Networks:** Ensure stability.
- **Batch Normalization:** May enhance training.

Algorithm DDPG

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .

Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$ ————— Use target networks to do off-policy updates

Initialize replay buffer R

for episode = 1, M **do**

 Initialize a random process $\mathcal{N}$ for action exploration

 Action selected using Deterministic Actor as explained before

 Receive initial observation state s_1

for t = 1, T **do**

 Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise

 Execute action a_t and observe reward r_t and observe new state s_{t+1}

 Store transition (s_t, a_t, r_t, s_{t+1}) in R

 Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R ————— Experience Replay

 Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

 Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

 Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$
 ————— Policy Gradient

 Update the target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

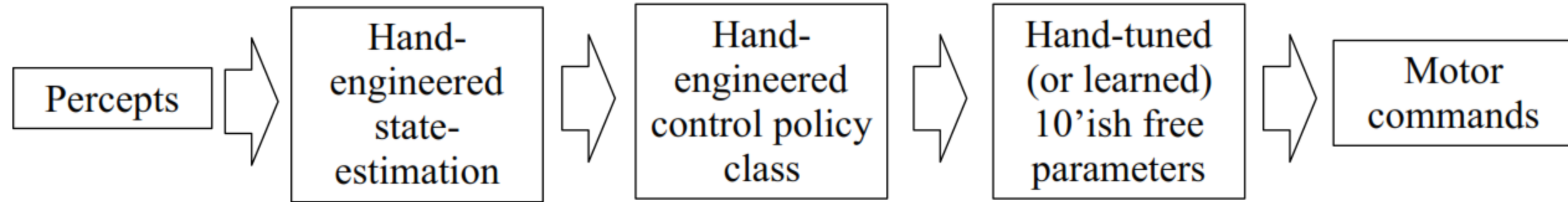
$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

$\tau \ll 1$

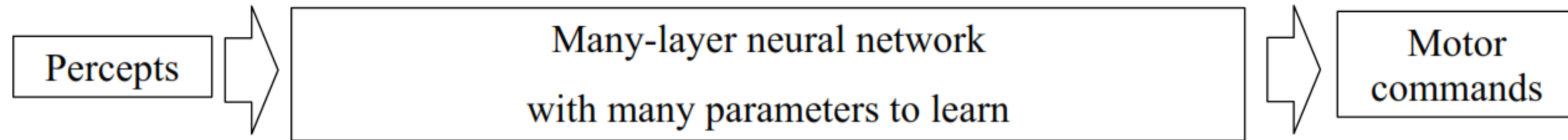
Slow updates,
but highly stable

end for

- **Current state-of-the-art robotics**



- **Deep reinforcement learning**

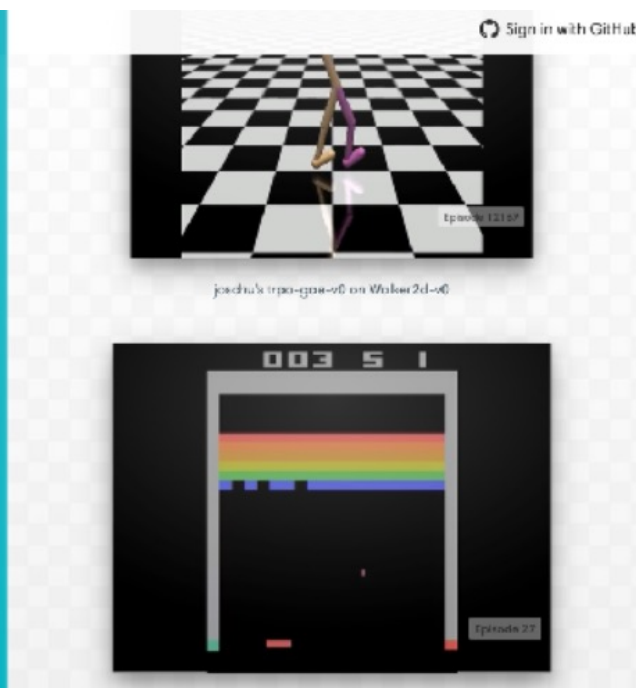


Environments Docs Chat

OpenAI Gym BETA

A toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Go.

[Read the launch blog post >](#)
[View documentation >](#)
[View on GitHub >](#)



- Environments Documentation
- Algorithms
 - Atari
 - Box2D
 - Classic control
 - MuJoCo**
 - Robotics
 - Toy text EASY
 - Third party environments [↗](#)

MuJoCo

Continuous control tasks, running in a fast physics simulator.

Ant-v2 Make a 3D four-legged robot walk.	HalfCheetah-v2 Make a 2D cheetah robot run.	Hopper-v2 Make a 2D robot hop.
Humanoid-v2 Make a 3D two-legged robot walk.	HumanoidStandup-v2 Make a 3D two-legged robot standup.	InvertedDoublePendulum-v2 Balance a pole on a pole on a cart.
InvertedPendulum-v2 Balance a pole on a cart.	Reacher-v2 Make a 2D robot reach to a randomly located target.	Swimmer-v2 Make a 2D robot swim.
Walker2d-v2 Make a 2D robot walk.		