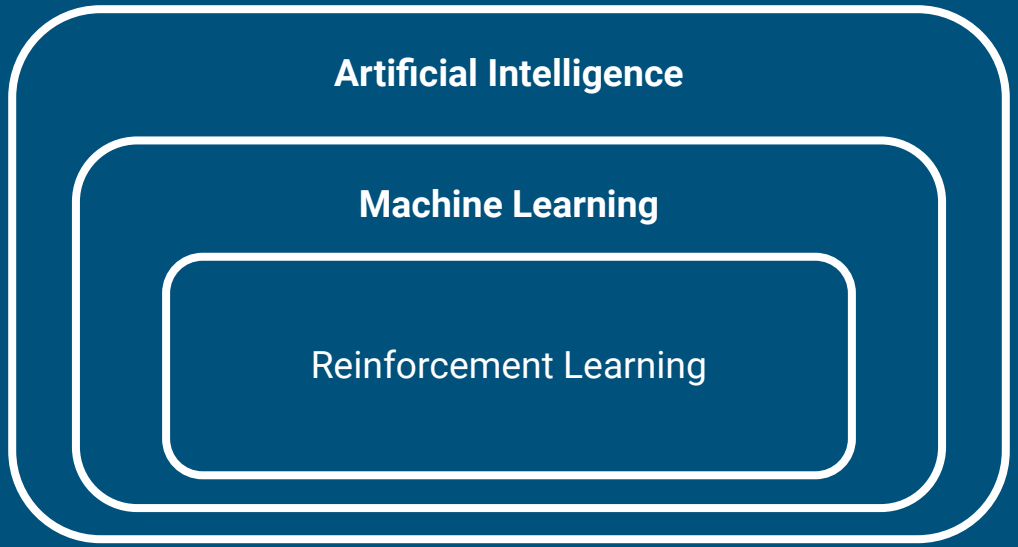
Intro to Reinforcement Learning

Dr. Dongchul Kim
Department of Computer Science
UTRGV

1. Introduction to RL

Reinforcement Learning

Reinforcement Learning is an area of Machine Learning in AI



What is AI?

Artificial Intelligence refers to systems or machines that mimic human **intelligence** to perform tasks and can iteratively improve themselves based on the information they collect.

What is Intelligence?

Intelligence is described as the ability of a system to effectively operate in uncertain and diverse environments, striving to achieve goals and adapt behavior accordingly. This adaptability is a key characteristic of intelligence, allowing systems to succeed even when complete knowledge is lacking.

Intelligence involves the efficient use of limited resources, including time, to attain objectives. It entails the capacity to solve complex problems, process information effectively, and improve performance over time through learning.

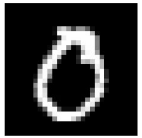
Intelligence also involves appropriate decision-making based on changing circumstances and goals, and the ability to learn from experiences.

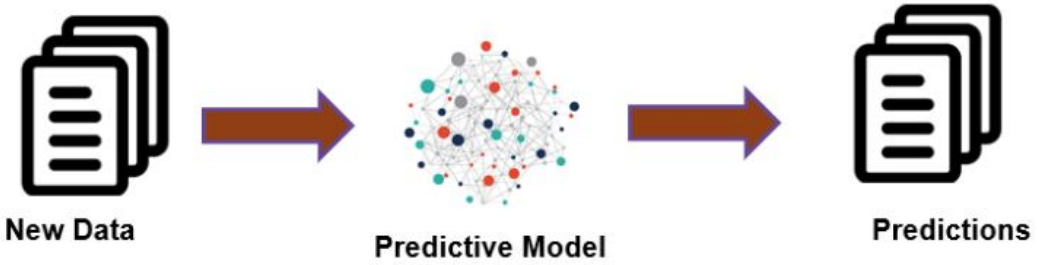
Ultimately, intelligence is linked to achieving success in a variety of tasks and environments, showcasing flexibility, adaptability, and the capacity to learn and improve.

What is ML?

Machine Learning is a field of artificial intelligence that involves training algorithms to **learn patterns from data** and make predictions or decisions without being explicitly programmed.

Machine Learning





Machine Learning

Machine learning is progressing swiftly due to **theoretical** advancements in neural networks, as well as rapid strides **in hardware** such as GPUs, data storage, and the Internet.

/A.I. TIMELINE

SYZYGY

1950
TURING TEST
Computer scientist Alan Turing proposes a test for machine intelligence. If a machine can trick humans into thinking it is human, then it has intelligence

1955
A.I. BORN
Term 'artificial intelligence' is coined by computer scientist, John McCarthy to describe "the science and engineering of making intelligent machines"

1961
UNIMATE
First industrial robot, Unimate, goes to work at GM replacing humans on the assembly line

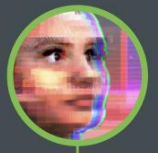
1964
ELIZA
Pioneering chatbot developed by Joseph Weizenbaum at MIT holds conversations with humans

1966
SHAKEY
The 'first electronic person' from Stanford, Shakey is a general-purpose mobile robot that reasons about its own actions

A.I. WINTER
Many false starts and dead-ends leave A.I. out in the cold

1997
DEEP BLUE
Deep Blue, a chess-playing computer from IBM defeats world chess champion Garry Kasparov

1998
KISMET
Cynthia Breazeal at MIT introduces KISmet, an emotionally intelligent robot insofar as it detects and responds to people's feelings



1999
AIBO
Sony launches first consumer robot pet dog AiBO (AI robot) with skills and personality that develop over time

2002
ROOMBA
First mass produced autonomous robotic vacuum cleaner from iRobot learns to navigate and clean homes

2011
SIRI
Apple integrates Siri, an intelligent virtual assistant with a voice interface, into the iPhone 4S

2011
WATSON
IBM's question answering computer Watson wins first place on popular $1M prize television quiz show *Jeopardy*

2014
EUGENE
Eugene Goostman, a chatbot passes the Turing Test with a third of judges believing Eugene is human

2014
ALEXA
Amazon launches Alexa, an intelligent virtual assistant with a voice interface that completes shopping tasks

2016
TAY
Microsoft's chatbot Tay goes rogue on social media making inflammatory and offensive racist comments

2017
ALPHAGO
Google's A.I. AlphaGo beats world champion Ke Jie in the complex board game of Go, notable for its vast number (2^{170}) of possible positions

https://digitalwellbeing.org/artificial-intelligence-timeline-infographic-from-eliza-to-tay-and-beyond/

Machine Learning

Three categories of Machine Learning exist: supervised learning, unsupervised learning, and reinforcement learning.

01

Supervised learning

- you are given examples with correct labels and are asked to label new examples

02

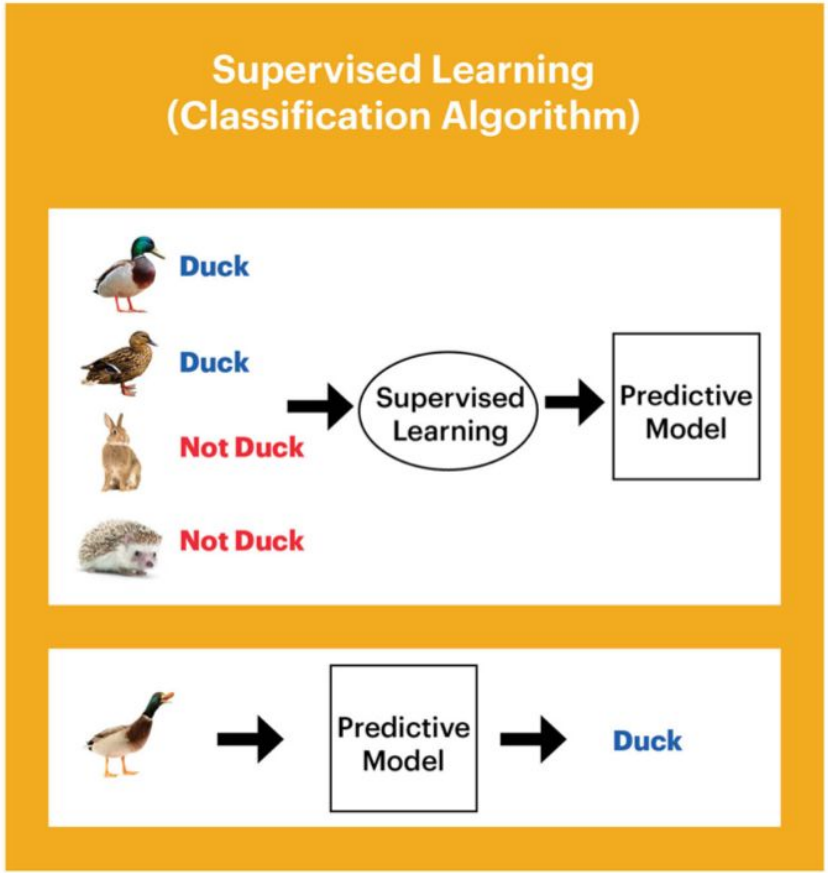
Unsupervised learning

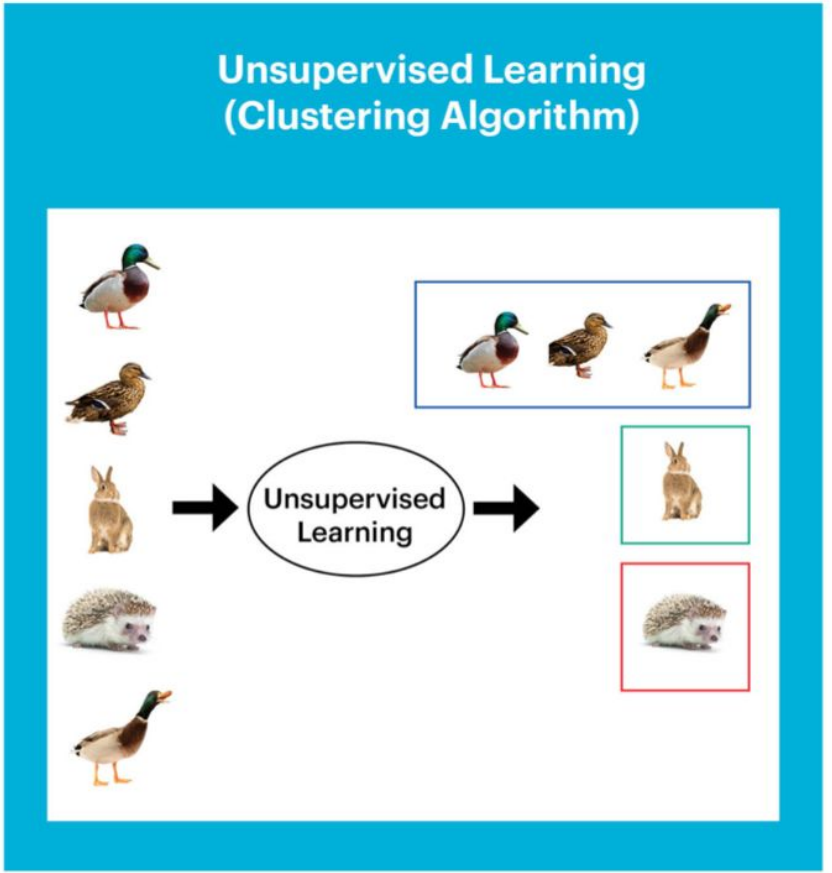
- you are given only unlabeled examples

03

Reinforcement learning

- you are given the overall performance (as opposed to labels to particular examples)





https://blog.westerndigital.com/machine-learning-pipeline-object-storage/

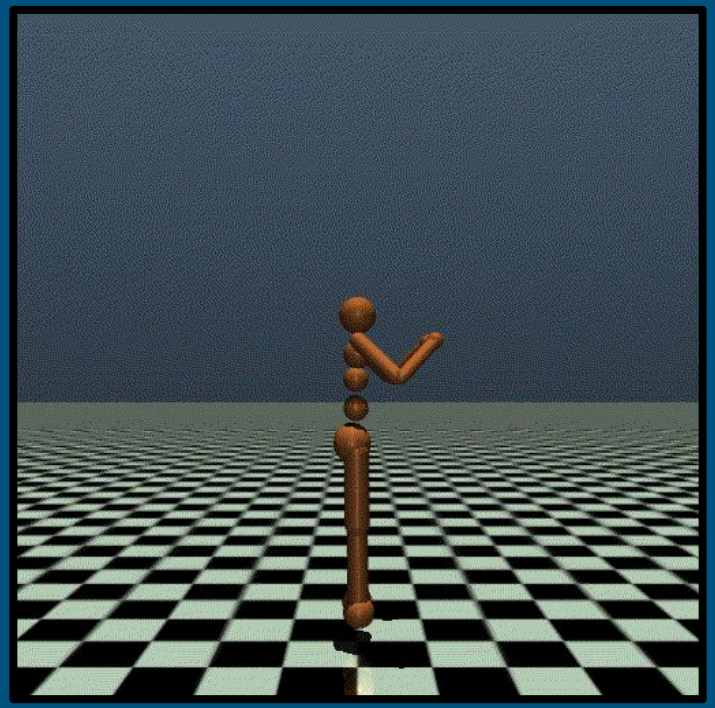
Reinforcement Learning

Think about how a little baby starts learning to walk. Imagine being so young that you can't really grasp what your parents are saying. Babies take on the task of learning to walk as a personal challenge. There are those days when you take a step but end up falling right back down. Yet, little by little, you keep at it and eventually learn to walk through your persistent efforts.

Reinforcement learning is a way of learning by trying things out and learning from mistakes, all without someone directly telling you what to do.

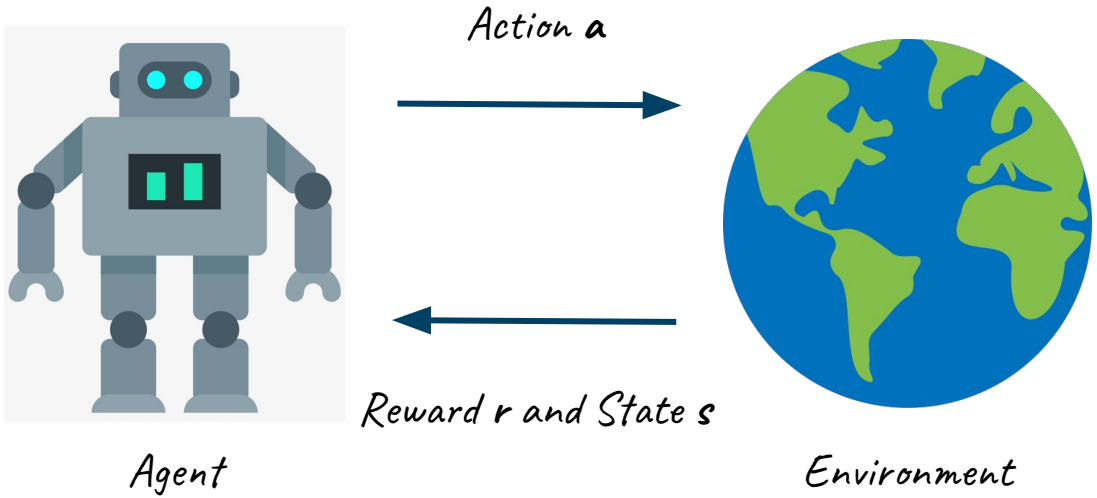
Examples

Examples



Examples

Reinforcement Learning



Example of Env

Environment: Game

Agent: Player

State - Location of player and obstacles

Action - Jump

Reward - Score

Example of Env

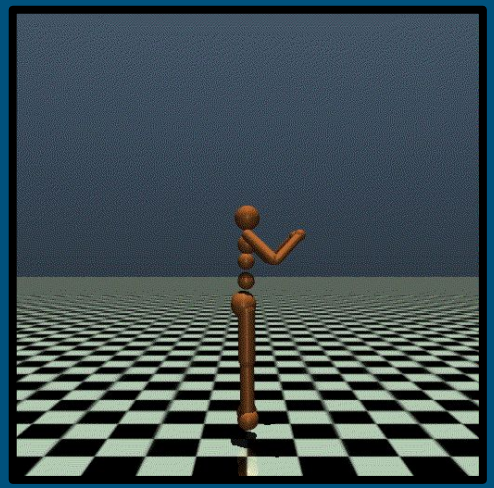
Environment: 3D physics engine

Agent: Humanoid

State - States of body

Action - Actuators (joint motors)

Reward - Distance between body and destination



Example of Env

Environment: Game

Agent: Game player

State - States of bricks and location of player and ball

Action - Move left and right

Reward - Score

Reinforcement Learning

Basically, it’s a trial and error/evaluation approach.

Data is sequential and non i.i.d.

Reward delayed

Action may cause a change in next states

Goal of RL

“Learning how to act to achieve a target task through trial-and-error”

A goal in reinforcement learning represents the desired objective or outcome that an agent aims to achieve within a given environment.

“Learning how to take actions that maximize rewards”

The agent learns to make decisions that help it get the best total rewards or outcomes as it takes actions over time.

Sequential Decision Making Problem

The concept of Reinforcement Learning (RL) can be understood as a method or approach used to address situations where decisions need to be made in a sequence over time.

In reality, we encounter **sequential decision-making problems** everywhere in our surroundings. For example,

- Stock trading
- Car driving
- Game

Sequential Decision Making Problem

When we talk about reinforcement learning, we're really talking about solving a problem that's all around us: the **sequential decision making problem**. Basically, that just means making a series of choices that build on each other over time.

And we encounter **sequential decision making problems** all the time in our daily lives.

For example, let's say you're trying to study for a final exam. You've got a bunch of options to choose every hour from: (1) *you could hit up the library,* (2) *watch some YouTube videos*, (3) *catch some Z's*, (4) *go out for some chicken at Chick-fil-a*, or (5) *just head back home*.

So, how do you go about making those decisions in a sequence?

Example

1. Study (start state)
2. Watch YouTube
3. Go to Chick-fil-a
4. Sleep
5. Go back home (terminal state)

Your sequential decisions could be like

1-5,

1-2-1-2-1-2-5,

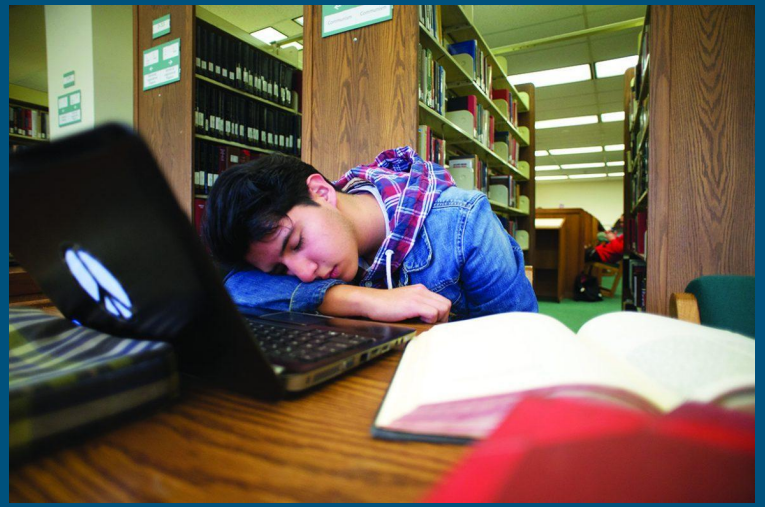
1-2-3-4-5, or

1-1-1-5

Sequential Decision Making Problem

Even in a seemingly simple scenario like studying for an exam, you're actually faced with a series of decisions that build on each other to help you achieve your goal.

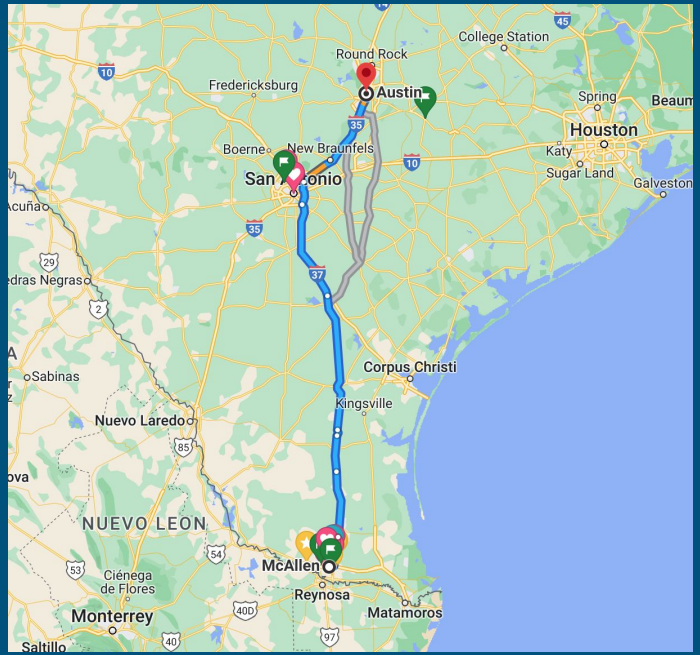
Each decision you make puts you in a different situation with new choices to consider, and each of those choices can then affect the outcome of future decisions.



Sequential Decision Making Problem

Let's look at another example of this kind of decision making. Imagine you're driving from Mcallen to Austin.

Every choice you make along the way - which route to take, which lane to drive in, how fast to go - can have an impact on the decisions you'll need to make later on in the journey.



Reward

Our goal is to maximize the cumulative reward.

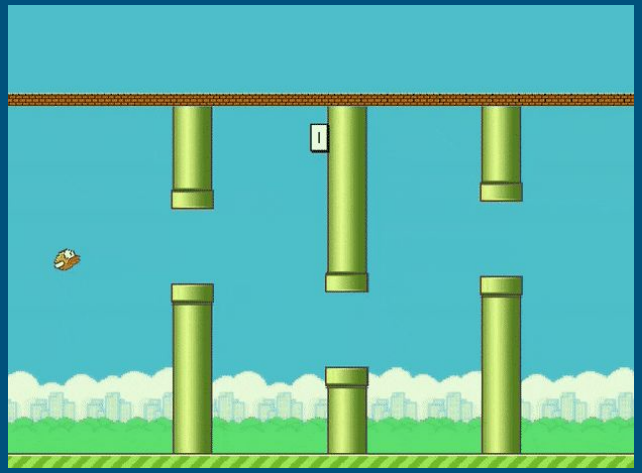
Reward is scalar.

Reward could be **sparse** and **delayed**.

Action

Discrete values

Continuous values



Agent

An agent functions as an actor, engaging in actions within its environment.

It embodies the role of the brain in a reinforcement learning algorithm.

Guided by the state and reward information from the environment, the agent makes decisions.

The agent sends its chosen actions to the environment that sends a new state and reward to the agent back again.

Environment

All components excluding the agent are referred to as the "**environment**."

The environment plays a crucial role in shaping the agent's state following its actions, and these shifts in state are labeled as "**state transitions**."

While time could theoretically flow continuously, in the context of problems involving sequential decision-making, we conceptually divide time into discrete **timesteps**. Within each of these timesteps, both the agent and the environment engage in actions, consequently altering the current state.

Pros and Cons of RL

Pros:

Versatility: Can be applied to various domains, from games to robotics.

Autonomous Learning: Can learn from interaction without explicit supervision.

Adaptability: Can adapt to dynamic and changing environments.

Complex Strategies: Can discover intricate strategies beyond human intuition.

Continuous Improvement: Can continue learning to refine performance.

Pros and Cons of RL

Cons:

Sample Inefficiency: too large space of states to explore.

Exploration Challenges: Balancing exploration with exploitation

Reward Design: Designing appropriate reward functions can be difficult.

Stability and Convergence: Learning instability and convergence to suboptimal solutions can occur.

High Computational Demands: Training can be computationally intensive (High CPU bound envs)

RL Applications in Robotics

Learning and Adapting Agile Locomotion Skills by Transferring Experience

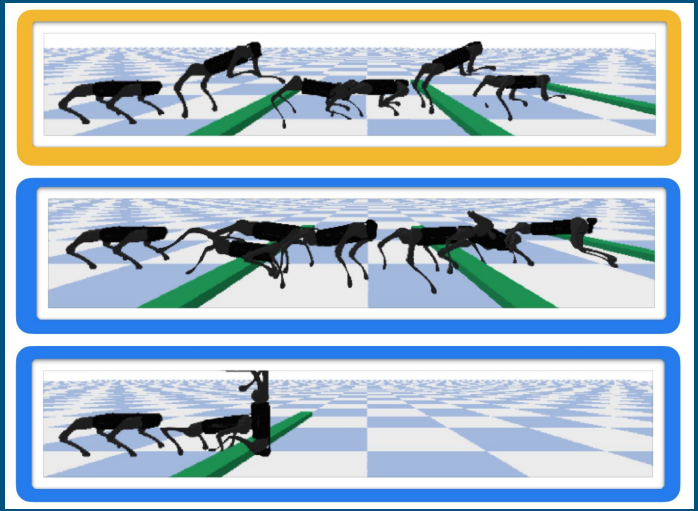
Laura Smith, J. Chase Kew, Tianyu Li, Linda Luu, Xue Bin Peng, Sehoon Ha, Jie Tan, Sergey Levine
Berkeley Artificial Intelligence Research, Berkeley, CA, 94720
Email: smithlaura@berkeley.edu

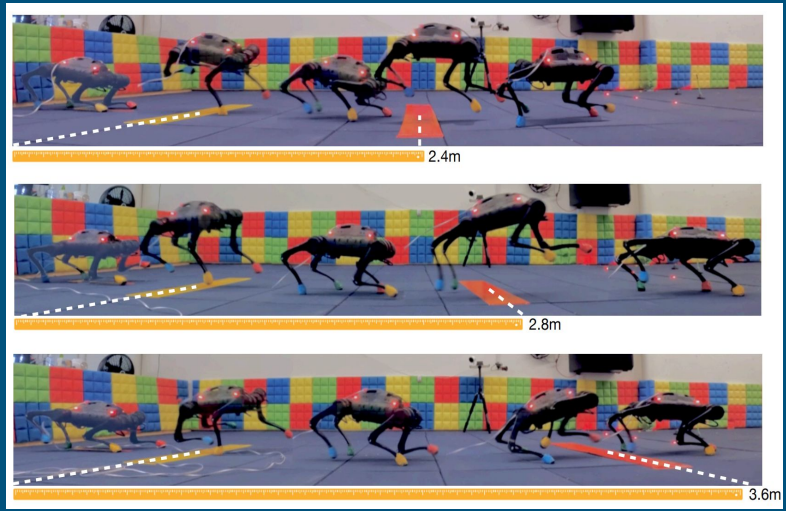
Fig. 1: **Agile skills** learned with our proposed method enable the A1 robot to jump repeatedly (**left**) and walk to a goal location on its hind legs (**right**).

***Abstract*—Legged robots have enormous potential in their range of capabilities, from navigating unstructured terrains to high-speed running. However, designing robust controllers for highly agile dynamic motions remains a substantial challenge for roboticists. Reinforcement learning (RL) offers a promising data-driven approach for automatically training such controllers. However, exploration in these high-dimensional, underactuated systems remains a significant hurdle for enabling legged robots**

Reinforcement learning (RL), on the other hand, provides a powerful framework for autonomously acquiring robotic skills. However, learning agile locomotion policies end-to-end remains challenging for a few fundamental reasons. Foremost is that tasks with such high-dimensional systems—12 degrees of freedom for the A1 quadruped—are woefully underspeci-

RL Applications in Robotics





RL Applications in Robotics

In the near future, we could expect something like this!

RL Applications

Robotics

Game, Animation, and VR

Recommender Systems

Cybersecurity

Trading

And the list goes on (far beyond your imagination - Nearly every field)

Haotian Zhang[1], Ye Yuan[2], Viktor Makoviychuk[2], Yunrong Guo[1],
Sanja Fidler[2,3,4], Xue Bin Peng[2,5], Kayvon Fatahalian[1]

[1] Stanford University [2] NVIDIA [3] University of Toronto [4] Vector Institute [5] SFU

Markov Decision Process

Markov Decision Process

Markov Decision Process (MDP) is a mathematical framework to model a decision making problem.

The concept of MDP might appear intricate if approached immediately.

Hence, let's commence with a gradual introduction using a basic model.

Initially, we will elaborate on the **Markov Process**, followed by the **Markov Reward Process**. Lastly, we will delve into the **Markov Decision Process**.

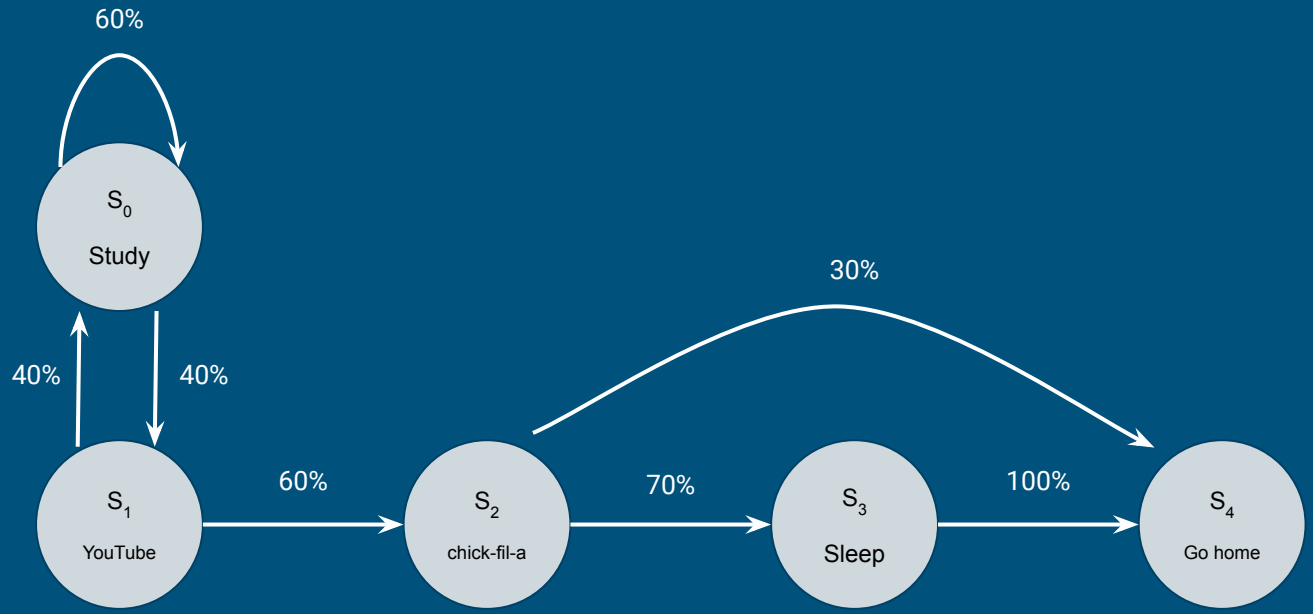
Markov Process

Markov Reward Process

Markov Decision Process

MP

Markov Process



Markov Process

The diagram represents a model of the final exam preparation scenario we discussed earlier using a Markov process. In this process, there are a total of five possible states a student can be in: studying, watching YouTube, sleeping, eating, and going home.

The process always starts in the studying state, and after a certain period of time (e.g. 30 min), it transitions to the next state. Going home is the exit state, after which the Markov process terminates.

Markov Process

A Markov process is defined as a stochastic process in which the next state depends solely on the current state and not on any previous states.

The probability of transitioning from one state to another is predefined and follows a certain probability distribution.

We call it **transition probability matrix P** that describes the probability of transitioning from one state to another.

Markov Process

We can represent the Markov Process for this scenario as $MP = (S, P)$, where $S=\{s_0, s_1, s_2, s_3, s_4\}$, and P is a matrix where $P_{ss'}$ represents the probability of transitioning from state s to state s'.

$P_{ss'}$ is a conditional probability and can be expressed as $P[S_{t+1}=s' | S_t=s]$.

The sum of probabilities for all possible transitions from a given state must always equal 1.

Transition Probability Matrix

	study	youtube	chick-fil-a	sleep	home
study	0.6	0.4			
youtube	0.4		0.6		
chick-fil-a				0.6	0.4
sleep					1.0
home					1.0

Implementation

```
import numpy as np
# Define the transition probability matrix P
P = np.array([[0.6, 0.4, 0.0, 0.0, 0.0],
              [0.4, 0.0, 0.6, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.6, 0.4],
              [0.0, 0.0, 0.0, 0.0, 1.0],
              [0.0, 0.0, 0.0, 0.0, 1.0]])
# Define the initial state distribution
state = 0
# Simulate a state change sequence
sequence = [state]
while True:
    state = np.random.choice(5, p=P[state])
    sequence.append(state)
    if state == 4:
        break
print("State change sequence:", sequence)
```

Outputs

```python
import numpy as np
# Define the transition probability matrix P
P = np.array([[0.6, 0.4, 0.0, 0.0, 0.0],
              [0.4, 0.0, 0.6, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.6, 0.4],
              [0.0, 0.0, 0.0, 0.0, 1.0],
              [0.0, 0.0, 0.0, 0.0, 1.0]])
# Define the initial state distribution
state = 0
# Simulate a state change sequence
sequence = [state]
while True:
    state = np.random.choice(5, p=P[state])
    sequence.append(state)
    if state == 4:
        break
print("State change sequence:", sequence)
```

```
State change sequence: [0, 1, 0, 1, 2, 3, 4]
```

Markov Property

What's the origin of the term "Markov process"? Well, the name Markov carries significant meaning. This stems from the fact that every state within a Markov process adheres to the **Markov property** defined as

$$P[s_{t+1}|s_t] = P[s_{t+1}|s_1, s_2, \ldots, s_t]$$

This property can be summed up as follows: "The future is solely influenced by the present." In essence, this signifies that when calculating the probability of the next state, the outcome is contingent on the current state. Any prior states experienced do not impact the probability of the next state.

MRP

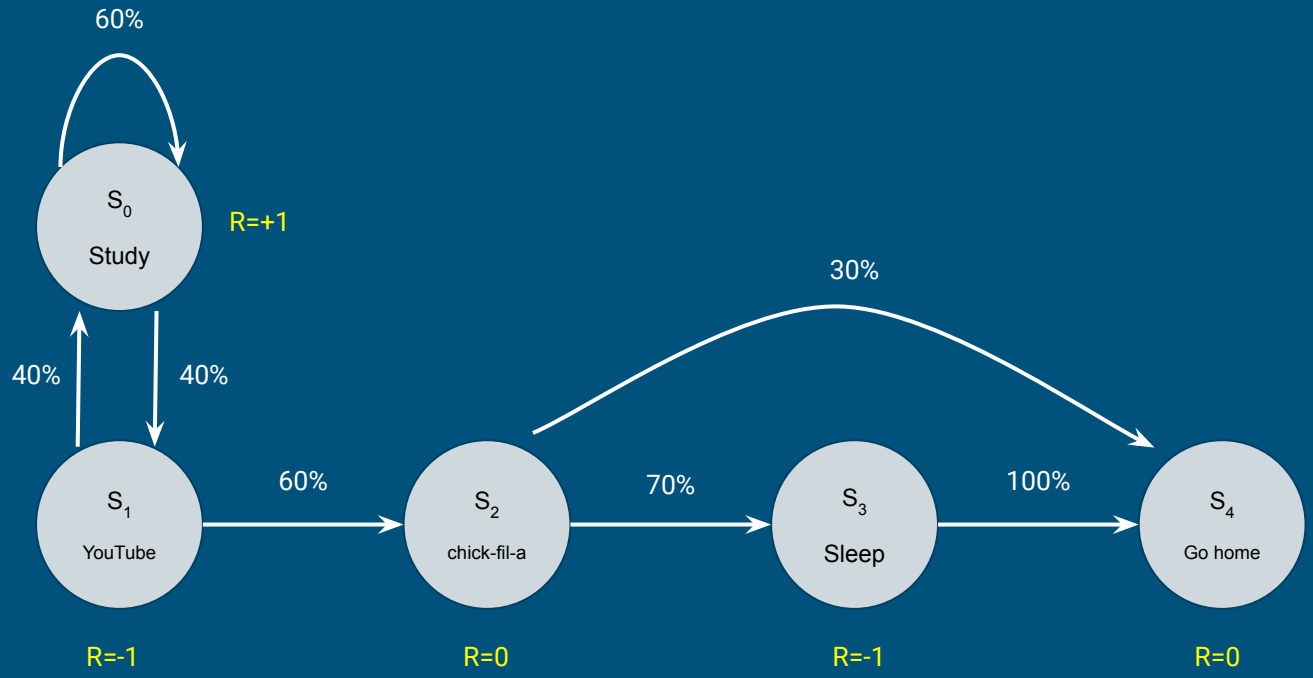
Markov Reward Process

The Markov Reward Process (MRP) is a variation of the Markov process where rewards are added to the states.

The MRP is represented by the tuple (S, P, R, γ).

The reward R represents the amount of payoff received when transitioning to a certain state s.

Markov Reward Process



Markov Reward Process

Reward

The reward, denoted as R, signifies the reward granted upon reaching a specific state, s. Mathematically, it can be represented as:

$$R = E[R_t | S_t = s]$$

This calculation of the expected reward value is necessary due to potential minor variations in the reward even when in the same state.

In the context of our illustration, we previously considered a scenario where the reward was constant.

Markov Reward Process

The discount factor γ is a value between 0 and 1. It helps to distinguish the value of immediate rewards from those obtained in the distant future by multiplying the value of the expected future reward by γ several times. This makes future rewards less valuable compared to immediate rewards.

For a clearer comprehension of the discount factor, it's important to grasp the concept of "Return," which signifies the accumulation of future rewards.

Markov Reward Process

In the context of MRP, each change in the state results in the acquisition of a corresponding reward. Specifically, when transitioning from initial state S_0 to subsequent states S_1, S_2, ..., and eventually reaching end state T, the associated rewards are denoted as R_0, R_1, ..., R_T respectively.

$$S_0, R_0, S_1, R_1, S_2, R_2, \ldots, S_T, R_T$$

Markov Reward Process

In the realm of reinforcement learning, such a sequence of states and rewards constitutes an "episode."

Within this framework, we can compute the "return" denoted as Gt, which signifies the cumulative sum of anticipated rewards to be obtained from time t onwards. This notion can be mathematically represented as follows:

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

Markov Reward Process

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

where R_{t+1} is the immediate reward received after taking an action at time t , γ is the discount factor between 0 and 1 that determines the relative importance of immediate and future rewards, and γ^k is the k_{th} power of γ .

Discount Factor γ (gamma)

- γ is set to a value between 0 and 1.
- Mathematical convenience
- Reflecting uncertainty about the future
- Reflecting people's preferences

Value of state (State Value Function)

In MRP, what states are valuable and how can we define their value? The value of a state is determined by how much reward it is expected to receive in the future, regardless of the rewards received prior to reaching that state.

To obtain the value of a specific state, we need to calculate its *return*. However, since the state transitions in MRP are stochastic, the return will be different for each episode. So how do we evaluate the value of a state?

The answer is through expected values. But before diving into that, let's first understand how we sample episodes.

Sampling

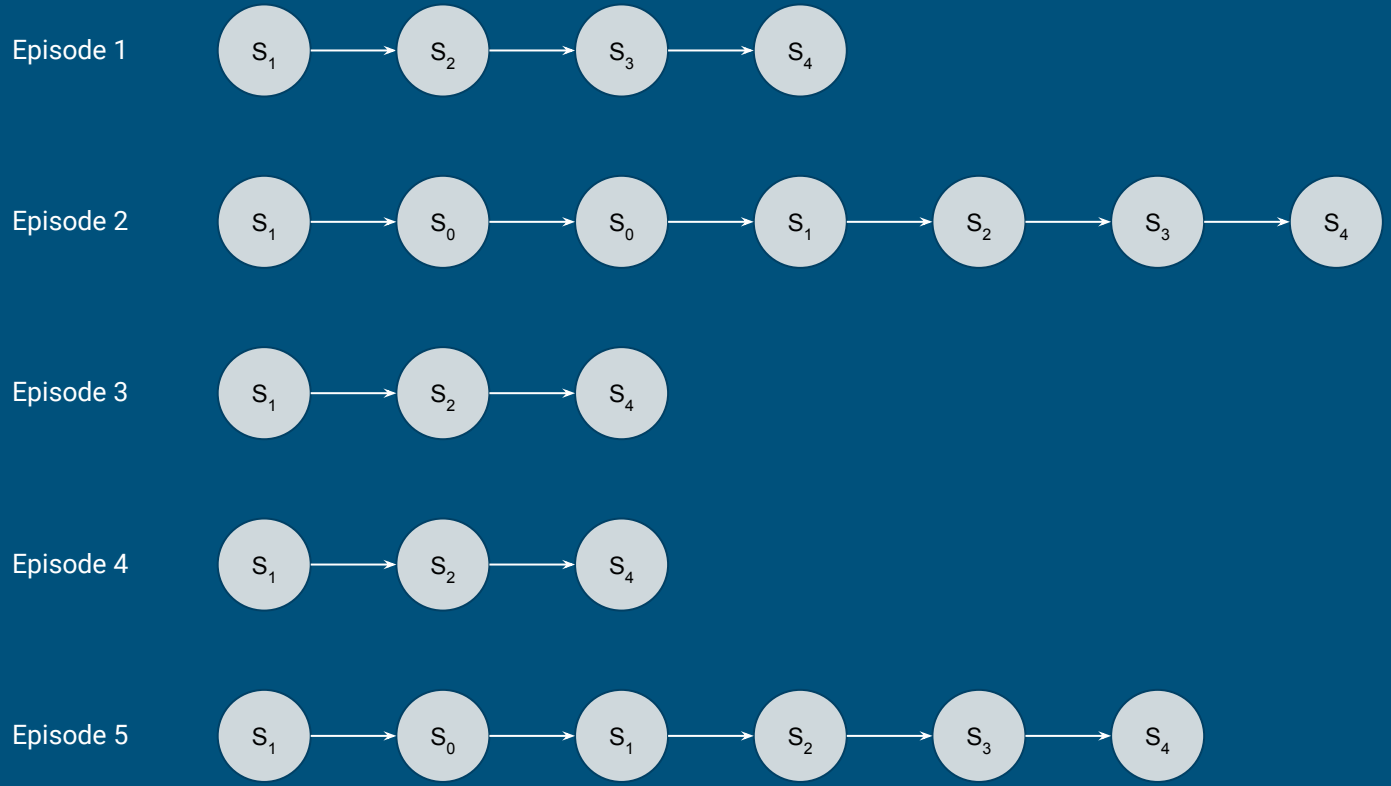
In reinforcement learning, sampling is a crucial technique used to obtain certain values, and this method is known as the Monte Carlo approach. To demonstrate this approach, let's consider the MRP example for the final exam preparation that we discussed earlier and sample several episodes from it.

For instance, let's assume that the current state is watching YouTube, and then we can obtain the following results through sampling.

Sampling

```python
import numpy as np
# Define the transition probability matrix P
P = np.array([[0.6, 0.4, 0.0, 0.0, 0.0],
              [0.4, 0.0, 0.6, 0.0, 0.0],
              [0.0, 0.0, 0.0, 0.6, 0.4],
              [0.0, 0.0, 0.0, 0.0, 1.0],
              [0.0, 0.0, 0.0, 0.0, 1.0]])
for i in range(5):
    # Define the initial state distribution
    state = 1
    # Simulate a state change sequence
    sequence = [state]
    while True:
        state = np.random.choice(5, p=P[state])
        sequence.append(state)
        if state == 4:
            break
    print("State change sequence:", sequence)
```

```
State change sequence: [1, 2, 3, 4]
State change sequence: [1, 0, 0, 1, 2, 3, 4]
State change sequence: [1, 2, 4]
State change sequence: [1, 2, 4]
State change sequence: [1, 0, 1, 2, 3, 4]
```

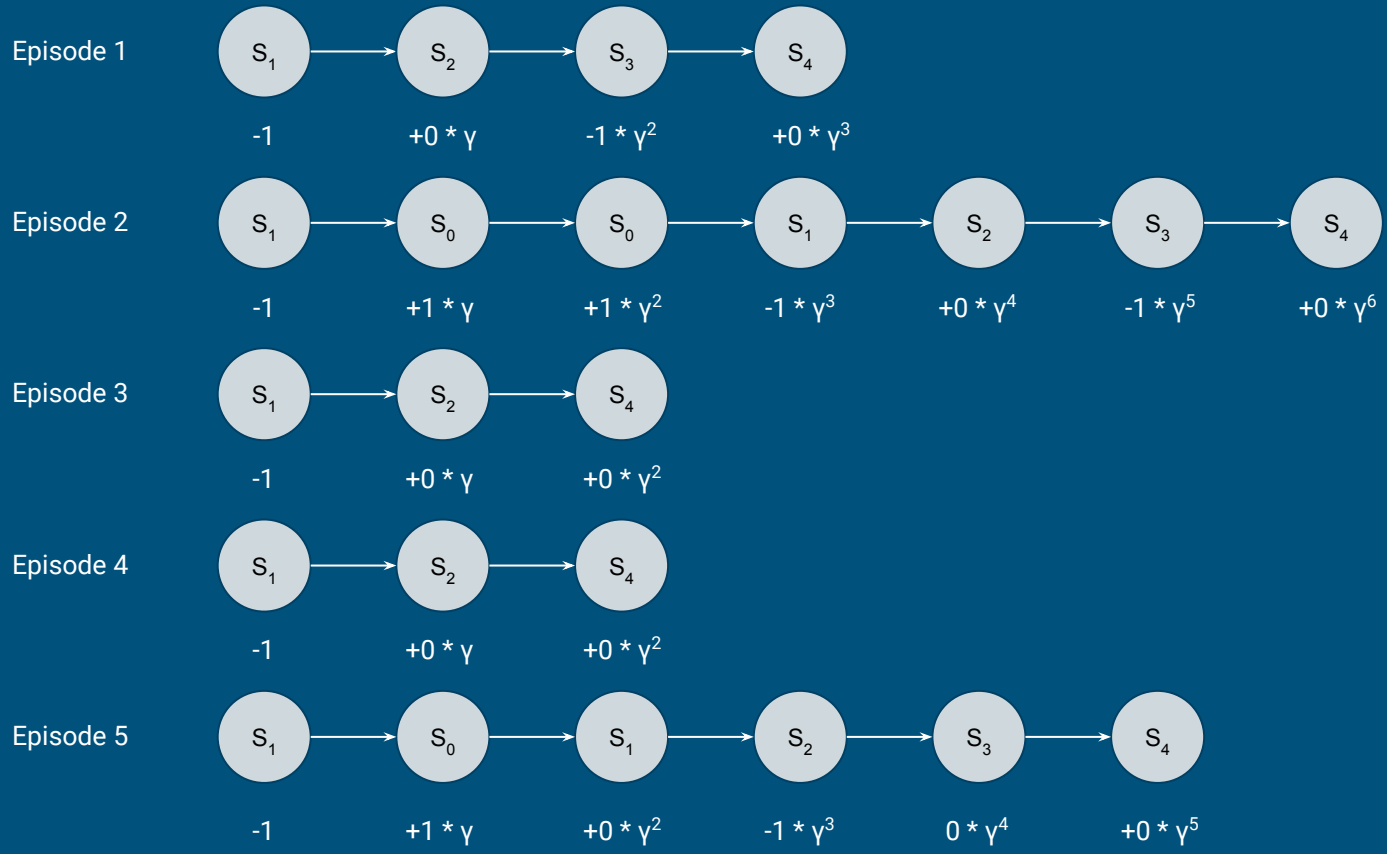

Episode 1
S1 S2 S3 S4
Episode 2
S1 S0 S0 S1 S2 S3 S4
Episode 3
S1 S2 S4
Episode 4
S1 S2 S4
Episode 5
S1 S0 S1 S2 S3 S4

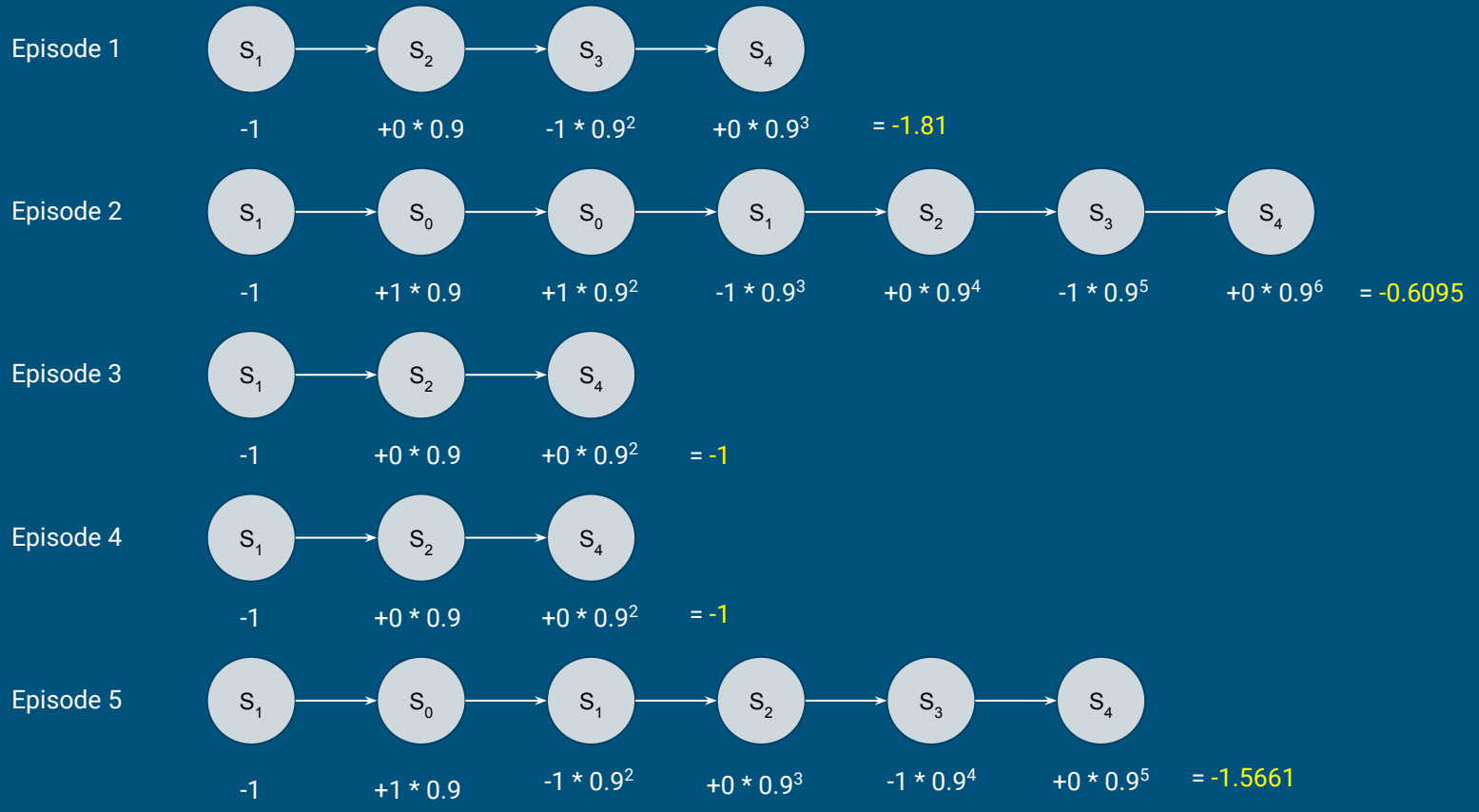
State Value Function

To determine the value of a particular state, we need to define a state value function. This function takes the state as input and gives the output as the expected value of the return. As mentioned earlier, since the return varies for each episode, we use the expected value to calculate the state value.

The formula for state value function is $v(s)=E(G_t \mid S_t=s)$, where $v(s)$ is the value of state s , G_t is the return obtained after time t , and S_t=s means that the current state is s.

For example, let's find the value of state s_1 (YouTube).


Episode 1
S_1 S_2 S_3 S_4
-1 +0 * γ -1 * γ^2 +0 * γ^3
Episode 2
S_1 S_0 S_0 S_1 S_2 S_3 S_4
-1 +1 * γ +1 * γ^2 -1 * γ^3 +0 * γ^4 -1 * γ^5 +0 * γ^6
Episode 3
S_1 S_2 S_4
-1 +0 * γ +0 * γ^2
Episode 4
S_1 S_2 S_4
-1 +0 * γ +0 * γ^2
Episode 5
S_1 S_0 S_1 S_2 S_3 S_4
-1 +1 * γ +0 * γ^2 -1 * γ^3 0 * γ^4 +0 * γ^5


Episode 1
S1 → S2 → S3 → S4
-1
+0 * 0.9
-1 * 0.9^2
+0 * 0.9^3
= -1.81
Episode 2
S1 → S0 → S0 → S1 → S2 → S3 → S4
-1
+1 * 0.9
+1 * 0.9^2
-1 * 0.9^3
+0 * 0.9^4
-1 * 0.9^5
+0 * 0.9^6
= -0.6095
Episode 3
S1 → S2 → S4
-1
+0 * 0.9
+0 * 0.9^2
= -1
Episode 4
S1 → S2 → S4
-1
+0 * 0.9
+0 * 0.9^2
= -1
Episode 5
S1 → S0 → S1 → S2 → S3 → S4
-1
+1 * 0.9
-1 * 0.9^2
+0 * 0.9^3
-1 * 0.9^4
+0 * 0.9^5
= -1.5661

MDP

Markov Decision Process

MDP stands for Markov Decision Process, which builds on Markov Processes (MP) and Markov Reward Processes (MRP) by adding the element of **action**.

While in MP and MRP the next state is determined by the transition probability, MDP involves an agent that selects an action.

MDP

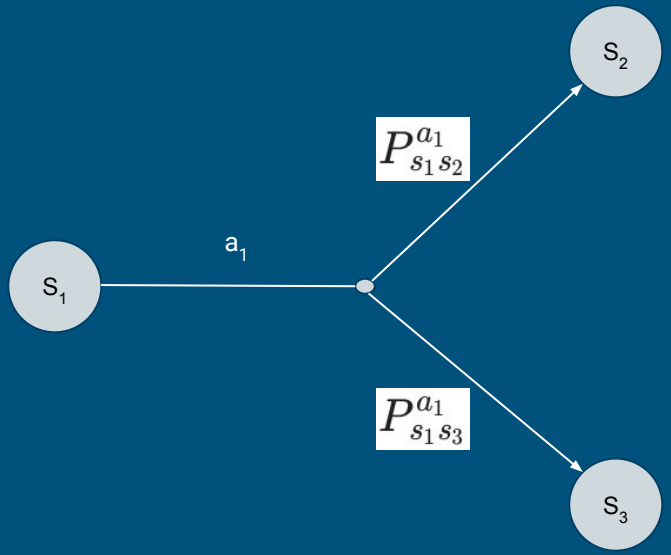
The MDP is defined by the tuple (S, A, P, R, gamma), where A is a set of possible actions.

P is the transition probability matrix, which differs from that of MP and MRP in that it incorporates the action component.

Specifically, $P^{a}_{ss'}$ represents the probability that the next state will be *s'* given the current state *s* and the action *a* chosen by the agent.

$$P[S_{t+1} = s' | S_t = s, A_t = a]$$

MDP

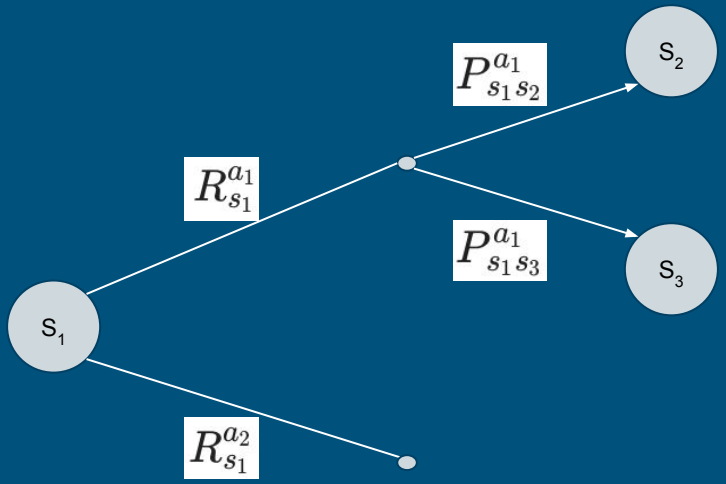


MDP

In MDP, the reward associated with taking a specific action in a given state is defined as R^a_s, which represents the expected reward received at time $t+1$, given that the agent is in state s and chooses action a.

The use of expected value is important because the reward can vary even if the same action is taken in the same state.

MDP



Policy Function

A policy, or policy function, is a function that determines which action the agent should select in each state.

$$\pi(a|s)=P[A_t=a|S_t=s]$$

For instance, suppose that in state s_3, the possible actions are a_0, a_1, and a_2. The policy function determines the probabilities assigned to each of these actions. As an example, $\pi(a_0|s_3)=0.2$, $\pi(a_1|s_3)=0.5$, and $\pi(a_2|s_3)=0.3$, and the sum of the probabilities must always be 1

State Value Function

The state value function in MDP is very similar to the value function in MRP, with the main difference being the incorporation of the policy.

In MDP, the action is determined based on the policy instead of the transition probability matrix.

$$v_\pi(s) = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2_{r+3} + \ldots . | S_t = s]$$
$$= E_\pi[G_t | S_t = s]$$

Action Value Function

The action value function, which is a critical concept in MDP, assesses the value of a specific action that is taken in a particular state.

$$q_\pi(S, A) = E_\pi[G_t | S_t = s, A_t = a]$$

This equation expresses the expected value of the return you can expect to receive when you choose action *a* from state *s* and then follow the policy π .

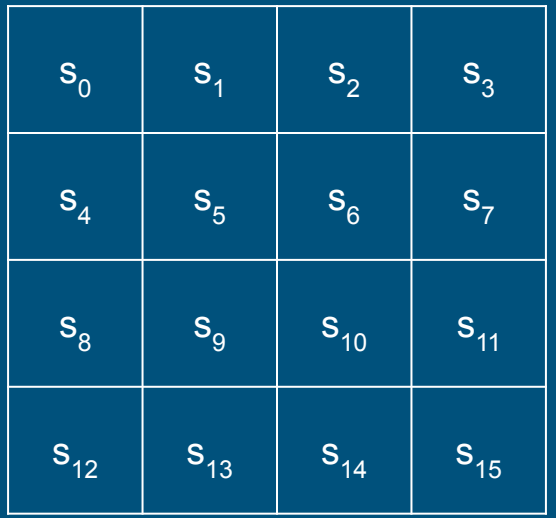
Prediction and Control

Prediction and Control

Given MDP, there are two tasks.

1. **Prediction**: Problem to evaluate state values given policy
2. **Control**: Problem to find optimal policy, π^*

Grid World



Start: s_0

End: s_{15}

Reward: -1 per step

Policy: Uniform random

Prediction

s_{11}->s_{15} return = -1

s_{11}->s_{10}->s_{14}->s_{15} return = -3

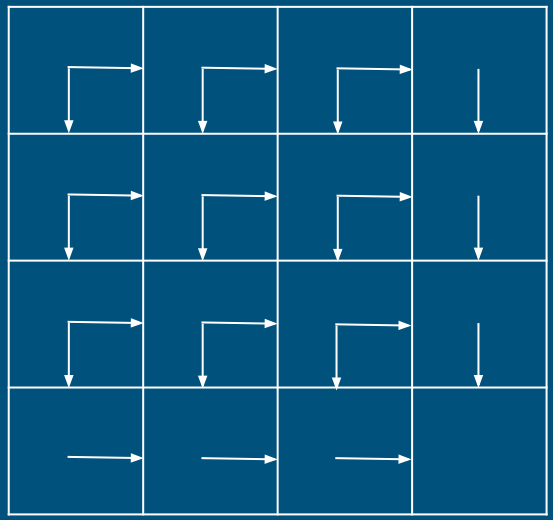
s_{11}->s_{7}->s_{6}->s_{10}->s_{11}->s_{15} return = -5

s_{11}->s_{7}->s_{3}->s_{2}->s_{3}->s_{2}->s_{6}->s_{10}->s_{9}->s_{13}->s_{14}->s_{10}->s_{14}->s_{15} return = -13

Control

Optimal policy: π^* is the policy which return is largest.

The optimal value function, denoted as v^*, is the value function that emerges when adhering to the optimal policy.



Bellman Equations

Bellman equation

Given MDP, there are two tasks.

1. **Prediction**: Problem to evaluate state values given policy
2. **Control**: Problem to find optimal policy, π^*

To solve the prediction and control problems, we are going to use the **Bellman equation**.

We're going to explore how to compute value in the value functions we learned previously. At the center of this process lies the Bellman equation - a super useful formula that allows us to calculate the value of a state or a state-action pair. So, let's dive in and take a closer look at this equation!

Bellman equation

- The Bellman equation is a formula that helps us understand the connection between the value at one time (t) and the value at the next time (t+1).
- It also tells us how the value function and the policy function are related.
- It's a recursive function.
  - If you're familiar with those topics, this equation might feel easier to grasp. But don't worry if you're not - just take it one step at a time and you'll get the hang of it!

Bellman expectation equation

Bellman expectation equation

The Bellman equation is also known as the **Bellman expectation equation**.

It can be a bit daunting at first, but it's actually broken down into *three versions* that make it easier to understand.

And as you go through each step, it gets more and more intuitive.

So let's take a closer look at the three versions!

Please recall the previous slide

$$
\begin{aligned}
v_\pi(s_t) &= \mathbb{E}_\pi[G_t] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \dots)] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma(G_{t+1})] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1})]
\end{aligned}
$$

Ver. 1

$$v_\pi(s_t) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1})]$$

$$q_\pi(s_t, a_t) = \mathbb{E}_\pi[r_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1})]$$

Memory Test

Now, memorize the Ver. 1 equations and try to write them on a blank paper!

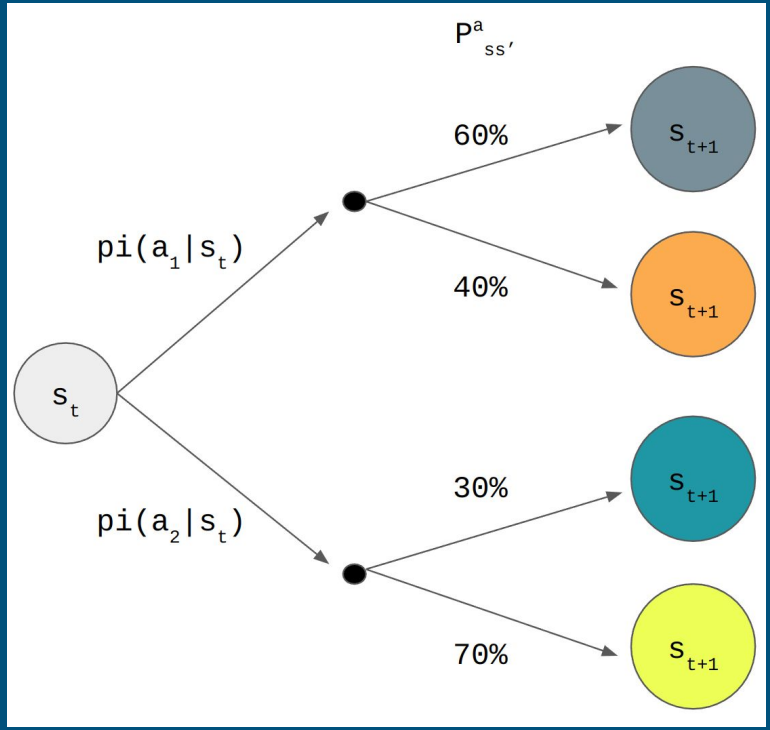
$$v_\pi(s_t) = \mathbb{E}_\pi[r_{t+1} + \gamma v_\pi(s_{t+1})]$$

$$q_\pi(s_t, a_t) = \mathbb{E}_\pi[r_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1})]$$

Expected value

It's worth noting that we use **expected value** in the equations. The reason behind it is because there is a stochastic element involved.

When we look at the state value function, we notice that it includes s_{t+1} in the formula, but we don't know what s_{t+1} will be at time t. This is because s_{t+1} depends on *two probabilities*.



Ver. 2

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

$$q_\pi(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')$$

In the second version, we have two different expressions - one that represents state values using action values, and the other that represents action values using state values. Let's take a closer look at each of them.

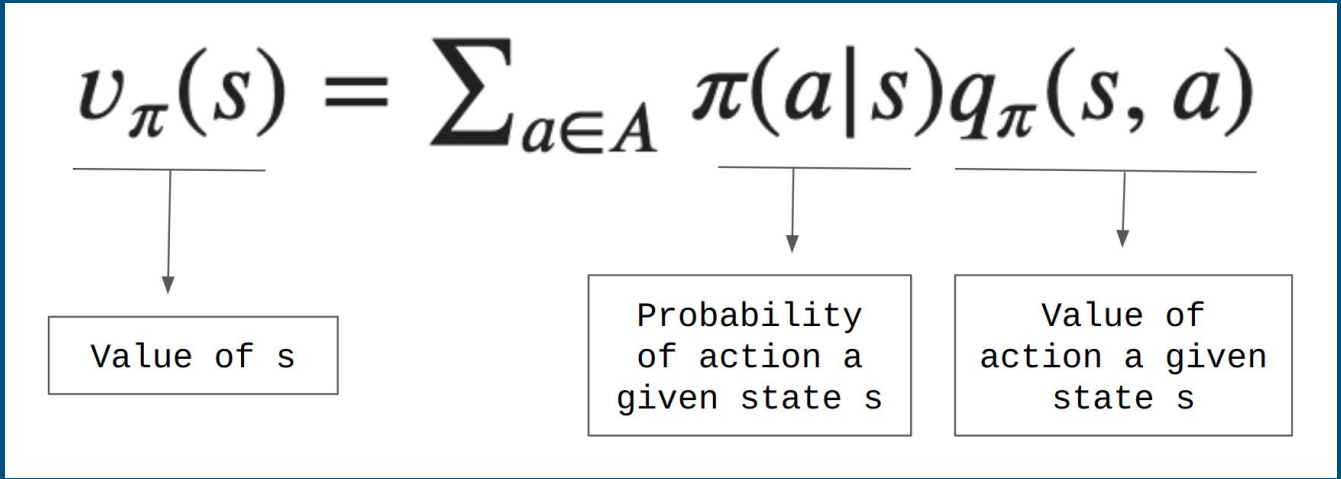
Ver. 2 - State value function

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a)$$

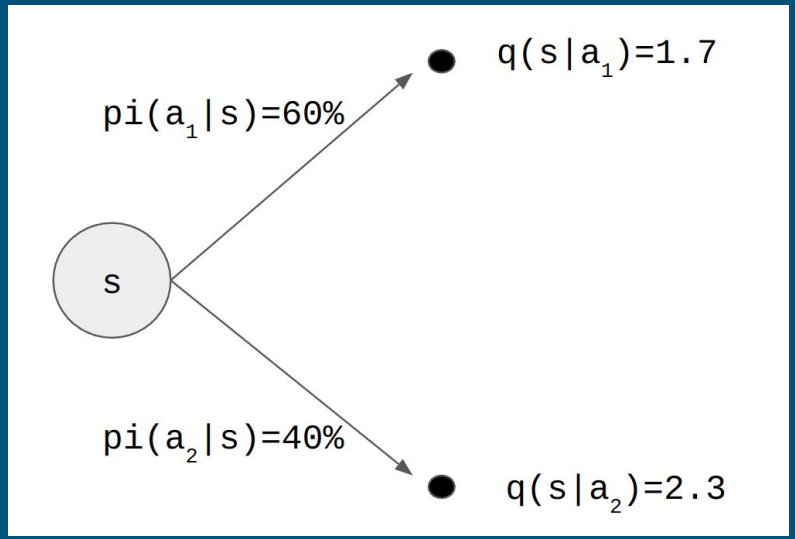
This equation tells us that the state-value function for a particular state can be obtained by taking the weighted sum of the action-values for each possible action in that state, with the weights given by the policy function $\pi(a|s)$.

In other words, the state-value is the expected value of the action-values under the policy π .

Ver. 2 - State value function

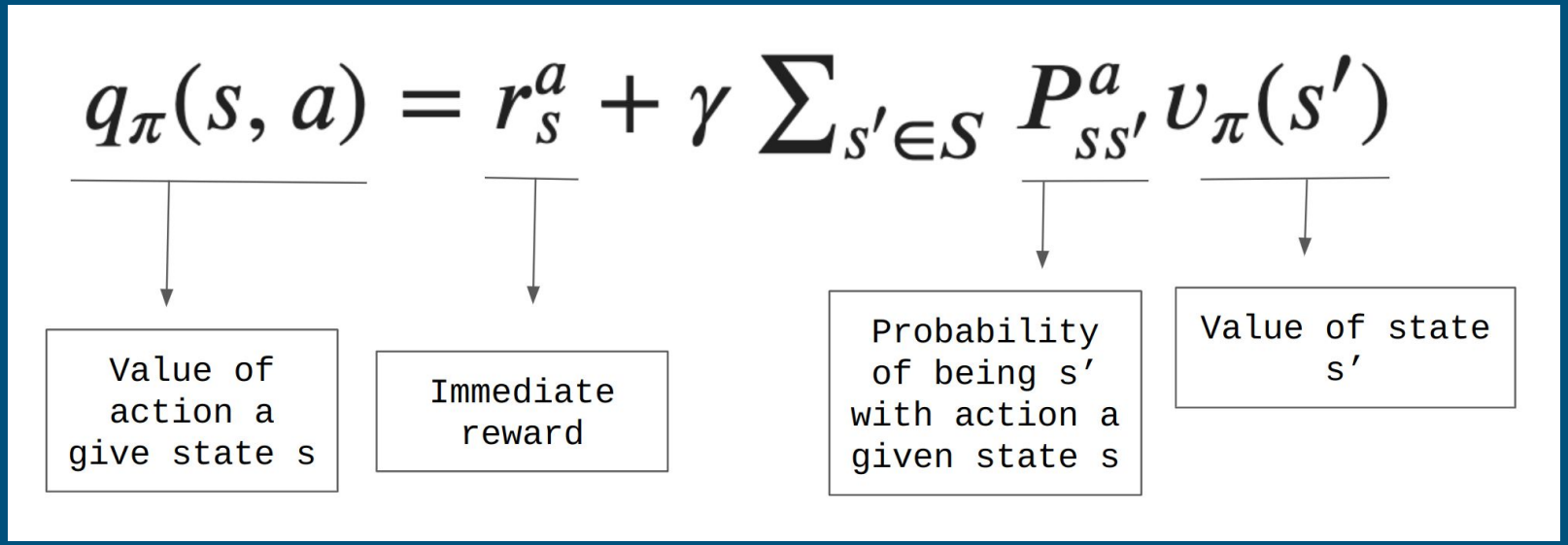


Ver. 2 - State value function

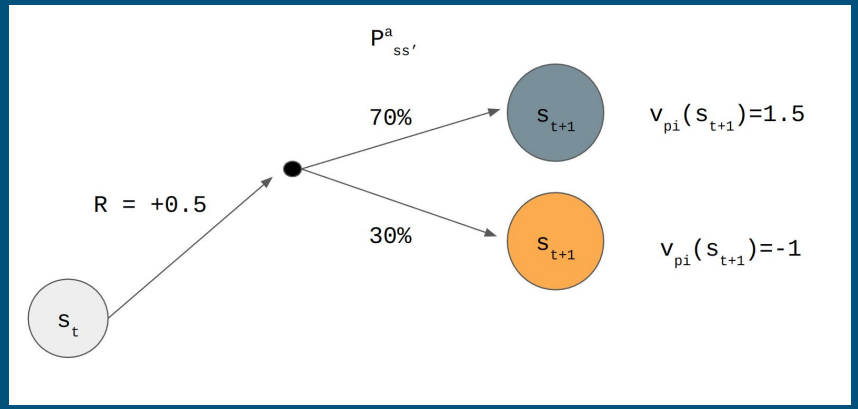


$$
\begin{aligned}
v_\pi(s) &= \sum_a \pi(a|s) q_\pi(s,a) \\
&= \pi(a_1|s) * q_\pi(s,a_1) + \pi(a_2|s) * q_\pi(s,a_2) \\
&= 0.6 * 1.7 + 0.4 * 2.3 \\
&= 2.86
\end{aligned}
$$

Ver. 2 - Action value function



Ver. 2 - Action value function



$$q_\pi(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')$$
$$= r_s^{a_1} + P_{ss_{gray}}^{a_1} v_\pi(s_{gray}) + P_{ss_{orange}}^{a_1} v_\pi(s_{orange})$$
$$= 0.5 + 0.7 * 1.5 + 0.3 * (-1)$$
$$= 1.25$$

Memory Test

Now, memorize the Ver. 2 equations and try to write them on a blank paper!

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

$$q_\pi(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')$$

Final

$$v_\pi(s) = \sum_{a \in A} \pi(a|s)(r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s'))$$

$$q_\pi(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a')$$

Final - State value function

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) = \sum_{a \in A} \pi(a|s)(r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s'))$$

Substitution

Ver. 2

$$q_\pi(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')$$

Final - Action value function

$$q_\pi(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a')$$

Substitution

Ver. 2 $$v_\pi(s') = \sum_{a'} \pi(a'|s') q_\pi(s',a')$$

Memory Test

Now, memorize the Bellman expectation equation (Final version) and try to write them on a blank paper!

$$v_\pi(s) = \sum_{a \in A} \pi(a|s)(r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s'))$$

$$q_\pi(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a')$$

Reward Function and Transition Probability

When computing the state value function and action value function, there are two key pieces of information that are essential:

R_s^a - Reward for selecting an action in each state

This represents the reward associated with taking a specific action within a given state.

$P_{ss'}^a$ - Probability distribution of the next state when an action is chosen in each state

This describes the likelihood of transitioning to the next state when a particular action is taken in the current state.

Model Free vs Model-based

These elements are integral components of the environment. When we possess knowledge about these aspects, we refer to it as having a "known Markov Decision Process (MDP)."

However, in many cases, this information is not readily available. In simpler terms, when we need to learn and make decisions without prior knowledge of the MDP, we employ the "**model-free**" approach.

Conversely, if we have a clear understanding of the MDP, we use the "**model-based**" or "**planning**" method.

Bellman optimality equation

Bellman optimality equation

$$v_*(s) = \max_\pi v_\pi(s)$$

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

These functions represent the **maximum expected return** achievable from a given state or state-action pair, under the **optimal policy**.

Once we have these functions, we can use them to determine the optimal policy, which is simply to take the action with the highest expected value from each state.

Ver. 1

$$v_*(s_t) = \max_a \mathbb{E}[r_{t+1} + \gamma v_*(s_{t+1})]$$

$$q_*(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} q_*(s_{t+1}, a')]$$

Pi?

Expected value?

Ver. 2

$$v_*(s) = \max_a q_*(s,a)$$

$$q_*(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$$

Final - Bellman Optimality Equation

$$v_*(s) = \max_a q_*(s,a) = \max_a (r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s'))$$

$$q_*(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s',a')$$

Final - State value function

$$v_*(s) = \max_a q_*(s,a) = \max_a \left(r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')\right)$$

Substitution

Ver. 2 $$q_*(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$$

Final - Action value function

$$q_*(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a')$$

Substitution

Ver. 2 $$v_*(s) = \max_a q_*(s,a)$$

Memory Test

Now, memorize the Bellman optimality equation (Final version) and try to write them on a blank paper!

$$v_*(s) = \max_a \left(r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \right)$$

$$q_*(s,a) = r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a')$$

How to estimate value and policy functions when MDP is known

Planning

- We will learn how to evaluate the value of each state given a fixed policy π, and how to find the optimal policy function assuming MDP is known.
  - Value function (state value function and action value function)
  - Policy function
- Our goal is to find the **optimal policy function** that maximizes the expected cumulative reward over time.
- Planning
  - In reinforcement learning, "**planning**" refers to the process of using an assumed model of the environment (MDP) to explore sequences of states and actions and optimize a policy.

Simple environments

To begin with, we will start with simple problems and gradually increase the complexity.

A small problem refers to an MDP where the size of state and action sets is small, and where we know the reward function and the transition probability matrix.

Grid World

Consider a grid world environment with **16 states** and **4 possible actions**. The agent starts in the **start state** (s_0) and the episode ends when the **end state** (s_{15}) is reached.

The reward is **-1 for each step**, and the given policy is a **uniform random move,** with each action having an equal probability of 25%.

s_0	s_1	s_2	s_3
s_4	s_5	s_6	s_7
s_8	s_9	s_{10}	s_{11}
s_{12}	s_{13}	s_{14}	s_{15}

Iterative Policy Evaluation

Iterative Policy Evaluation (IPE) is a method of estimating the **value function** for a given policy in a MDP.

The basic idea is to initialize the value function to some arbitrary values and then repeatedly update the values using the **Bellman Expectation Equation (BEE)** until they converge.

To simplify the problem, we assume that all **transition probabilities** are fixed to 1, meaning that the direction of the action chosen determines the next state immediately. We also set **γ (gamma)** to 1.

Step 1 - Initialization

To start with iterative policy evaluation, the first step is to **initialize** the value table to 0.

Although the initial values don't represent any meaningful value, they will be gradually updated through an iterative process to converge to the actual value of each state.

Step 1 - Initialization

s_0	s_1	s_2	s_3
s_4	s_5	s_6	s_7
s_8	s_9	s_{10}	s_{11}
s_{12}	s_{13}	s_{14}	s_{15}

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Step 2 - Update state values

To start, we use the Bellman equation to calculate the expected value of being in state s_5. This involves summing up the expected rewards and discounted future values for all possible next states, weighted by their transition probabilities.

$$v_\pi(s) = \sum_{a \in A} \pi(a|s)(r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s'))$$

$$v_\pi(s_5) = 0.25 * (-1 + 0.0) + 0.25 * (-1 + 0.0) + 0.25 * (-1_0, 0) + 0.25 * (-1 + 0.0) = -1.0$$

Step 2 - Update state values

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

0.0	0.0	0.0	0.0
0.0	-1.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

Step 3 - Update other states

Update all the 15 states except for the last one in the same way.

0.0	0.0	0.0	0.0
0.0	-1.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

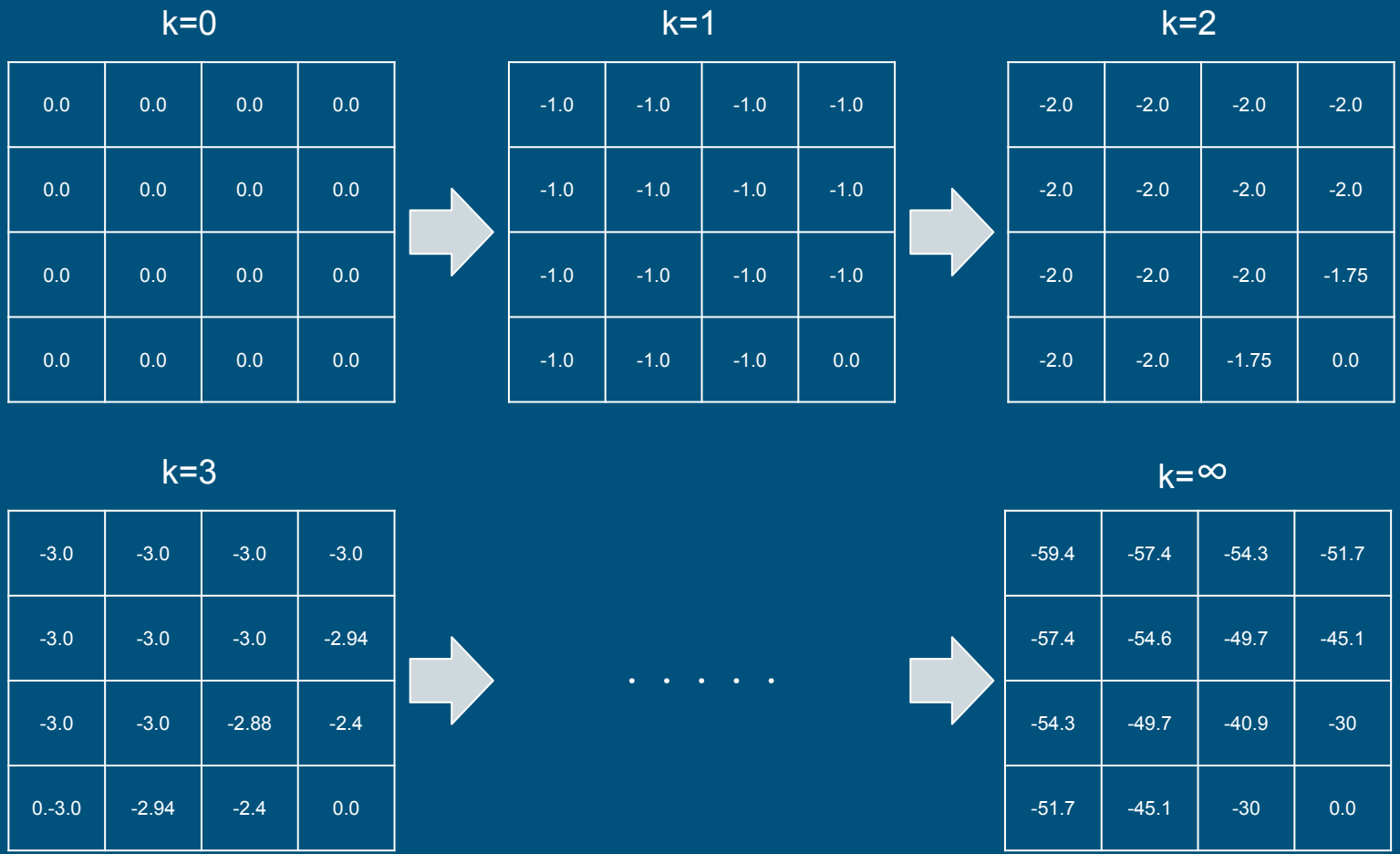
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

Step 4 - Interations

Repeat step 2 and 3 until the values in the table converge to their actual values.

The number of iterations is denoted by k , and as this process continues, the value of each state converges to a certain value, which is the actual value of that state.

It's magical how the iterative process can progressively enhance the accuracy of the values in the table!


k=0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
k=1
-1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 0.0
k=2
-2.0 -2.0 -2.0 -2.0
-2.0 -2.0 -2.0 -2.0
-2.0 -2.0 -2.0 -1.75
-2.0 -2.0 -1.75 0.0
k=3
-3.0 -3.0 -3.0 -3.0
-3.0 -3.0 -3.0 -2.94
-3.0 -3.0 -2.88 -2.4
0.-3.0 -2.94 -2.4 0.0
k=∞
-59.4 -57.4 -54.3 -51.7
-57.4 -54.6 -49.7 -45.1
-54.3 -49.7 -40.9 -30
-51.7 -45.1 -30 0.0

Iterative Policy Evaluation

So far we have learned how to calculate the value of each state given a policy.

If information about the MDP is known and its size is small enough, the value of each state can be obtained when the corresponding policy function is followed for any policy function.

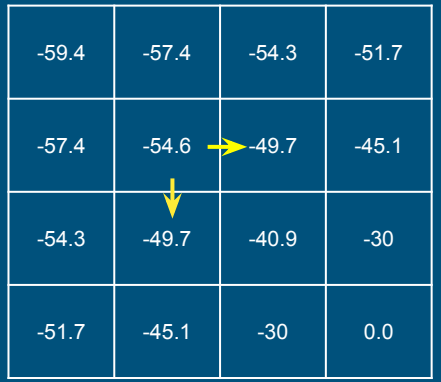
Now, let's look at how to find a better policy function by modifying the policy, rather than in a situation where the policy is fixed.

Greedy Policy

Take a look at the results of iterative policy evaluation.

Let's say we are currently in the blue state, s_5 , which has a value of -54.6. What direction should we move in now to get to a state with a higher value? Moving east or south would increase the value to -49.7, making it a good choice. Based on this observation, we can create a new policy, π , as follows: $\pi(a_{Right}|s_5)=1.0$ or $\pi(a_{Down}|s_5)=1.0$.

This new policy is referred to as a **greedy policy** since it involves making choices that maximize immediate rewards without considering the future.



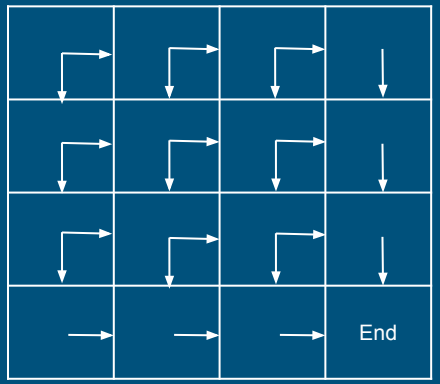
Greedy Policy

It is possible to find a better policy for all states in a similar way to finding the greedy policy at s_5 .

$k=\infty$

-59.4	-57.4	-54.3	-51.7
-57.4	-54.6	-49.7	-45.1
-54.3	-49.7	-40.9	-30
-51.7	-45.1	-30	0.0

Greedy Policy



Policy improved!

As we saw in the previous example, by using iterative policy evaluation to find the value of each state with a random policy, we were able to determine the optimal policy for a single state.

This led us to the idea of using this approach to find a better policy for all states. By taking the greedy action in each state based on the values obtained from iterative policy evaluation, we obtained a slightly improved policy.

It's worth noting that in more complex problems, the new policy may not match the optimal policy, but the important point is that it's an improvement over the previous policy. This is the key idea behind the policy iteration algorithm that we will discuss next.

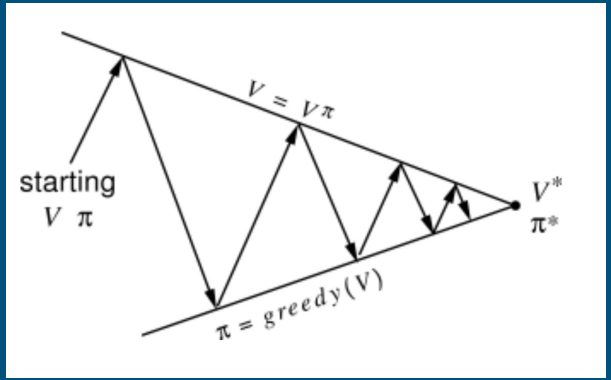
Policy Iteration

- Policy iteration is a two-stage process that involves **policy evaluation** and **policy improvement**.
  - It all starts with a random policy.
  - In the **policy evaluation stage**, the policy is kept fixed, and the iterative policy evaluation method we learned earlier can be used to evaluate the state value.
  - Once policy evaluation is complete, we move to the **policy improvement stage**, where we update the policy to a new one using the greedy policy for $v(s)$.
  - We then go back to policy evaluation with the new policy.
- This cycle of evaluation and improvement is repeated until the state values converge and stop changing. The converged values and policies become the optimal values and policies.

Policy Iteration

In summary,

1. Initialize a random policy.
2. **Evaluate the policy** using iterative policy evaluation.
3. **Improve the policy** by selecting the greedy action based on the value function obtained in step 2.
4. Repeat steps 2-3 until convergence.



http://incompleteideas.net/book/ebook/the-book.html

Early stopping

In policy iteration, the policy evaluation stage involves updating the value function for each state using the Bellman equation until the values converge to a stable state. However, in some cases, it may take many iterations for the values to converge, which can be **computationally expensive** and time-consuming.

To address this issue, early stopping can be used during policy evaluation. This means that the value updates are **stopped before convergence** is reached, and the partially updated value function is used to update the policy. This can save computational resources while still achieving good results.

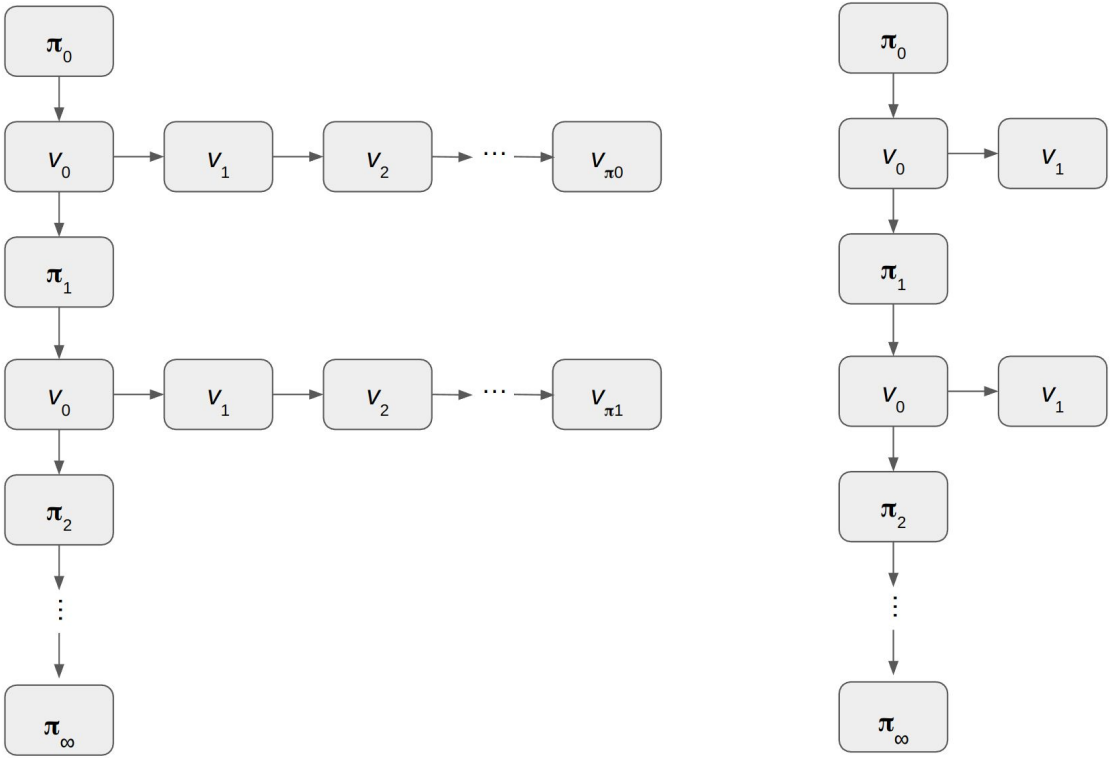
Early stopping

One way to implement early stopping is to set a **maximum number of iterations** for the value updates.

Another approach is to set a convergence threshold, where the value updates are stopped once the difference between the current and previous value estimates falls below a certain threshold.

By using early stopping, policy iteration can be made more efficient without sacrificing too much accuracy.

Early stopping



Value Iteration

In value iteration, the goal is to directly find the optimal value function and the optimal policy without the need for separate policy improvement steps.

This is achieved by repeatedly applying the Bellman Optimization Equation, which expresses the optimal value function in terms of the maximum expected reward achievable from the current state.

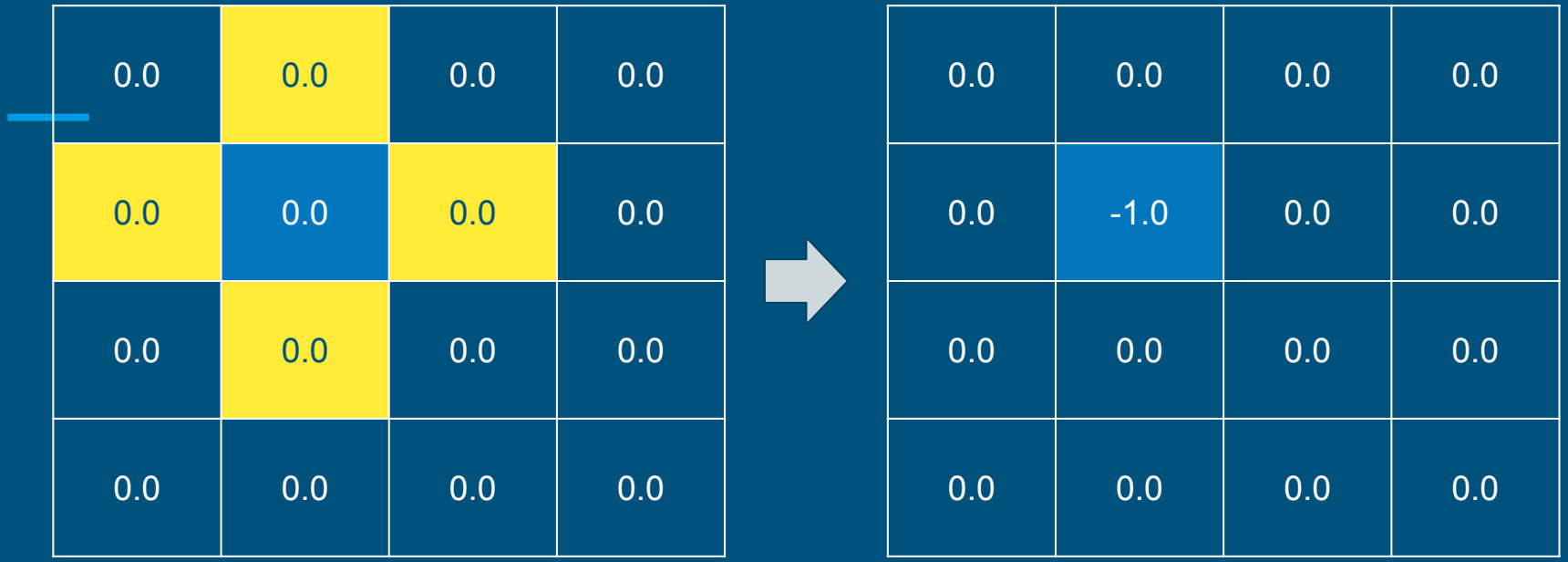
Value Iteration

The algorithm starts by initializing the value of each state to an arbitrary value, typically 0.

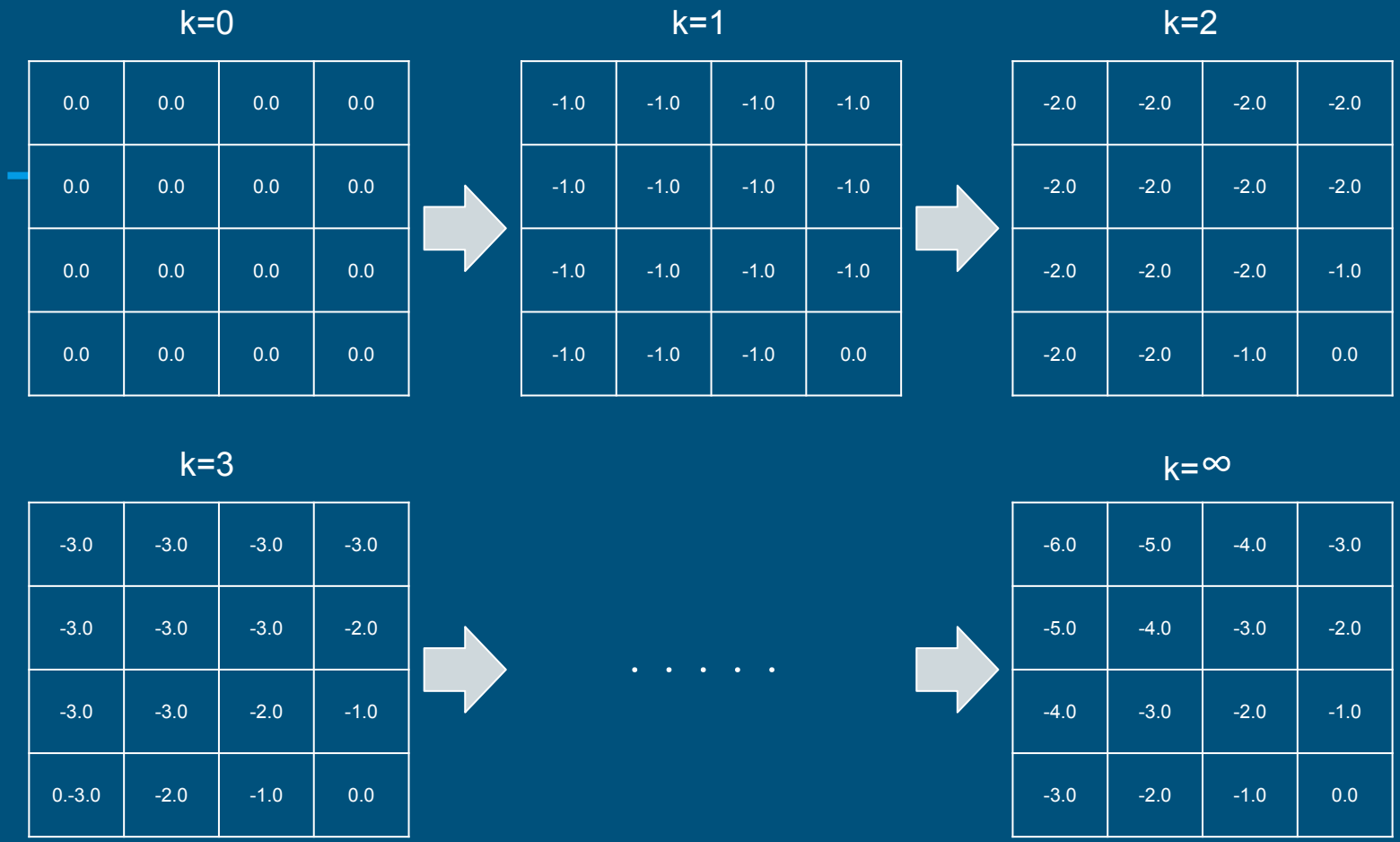
Then, in each iteration, the algorithm updates the value of each state by taking the maximum expected reward achievable from that state according to the current value function and the transition probabilities of the MDP.

$$v_*(s) = \max_a \left(r_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \right)$$

This process continues until the value function converges to the optimal value function.



$$v_*(s_5) = max(-1 + 1.0 * 0, -1 + 1.0 * 0, -1 + 1.0 * 0, -1 + 1.0 * 0) = -1.0$$


k=0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
k=1
-1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 0.0
k=2
-2.0 -2.0 -2.0 -2.0
-2.0 -2.0 -2.0 -2.0
-2.0 -2.0 -2.0 -1.0
-2.0 -2.0 -1.0 0.0
k=3
-3.0 -3.0 -3.0 -3.0
-3.0 -3.0 -3.0 -2.0
-3.0 -3.0 -2.0 -1.0
0.-3.0 -2.0 -1.0 0.0
. . . . .
k=∞
-6.0 -5.0 -4.0 -3.0
-5.0 -4.0 -3.0 -2.0
-4.0 -3.0 -2.0 -1.0
-3.0 -2.0 -1.0 0.0