

CST115

POLICY GRADIENT FOR REINFORCEMENT LEARNING

University Of Information Technology

TABLE OF CONTENTS

CHAPTER I REINFORCEMENT LEARNING

CHAPTER II POLICY

CHAPTER III POLICY GRADIENT

CHAPTER IV CODING MATERIALS

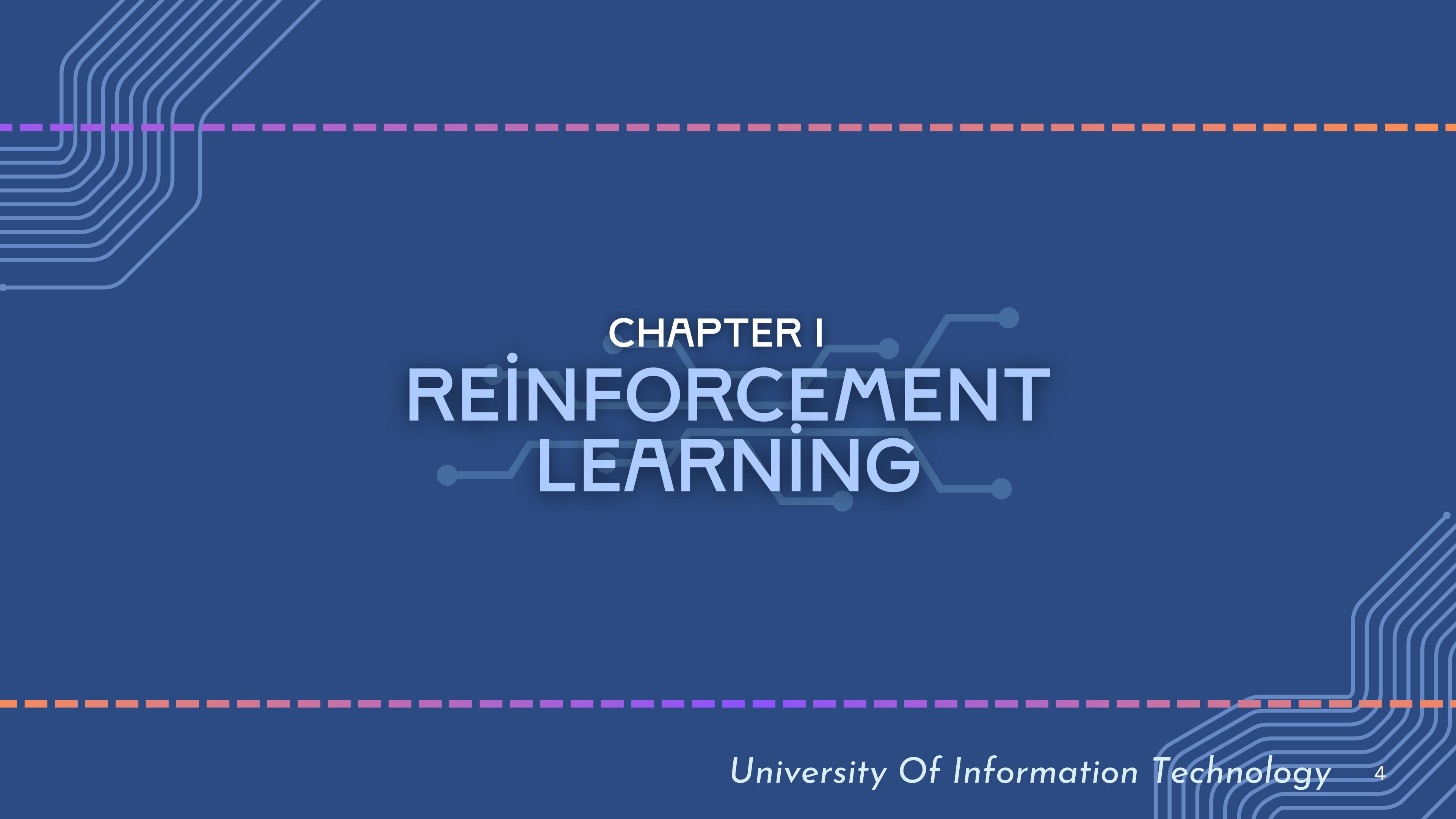
PROJECT BY GROUP 12

1 PHẠM TÂI LỘC - 23520865

2 LÊ PHÚ QUÝ - 23521316

3 TRẦN TUẤN KIẾT - 23520822

4 VÕ ANH KIẾT - 23520825



CHAPTER I

REINFORCEMENT

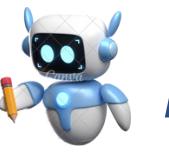
LEARNING



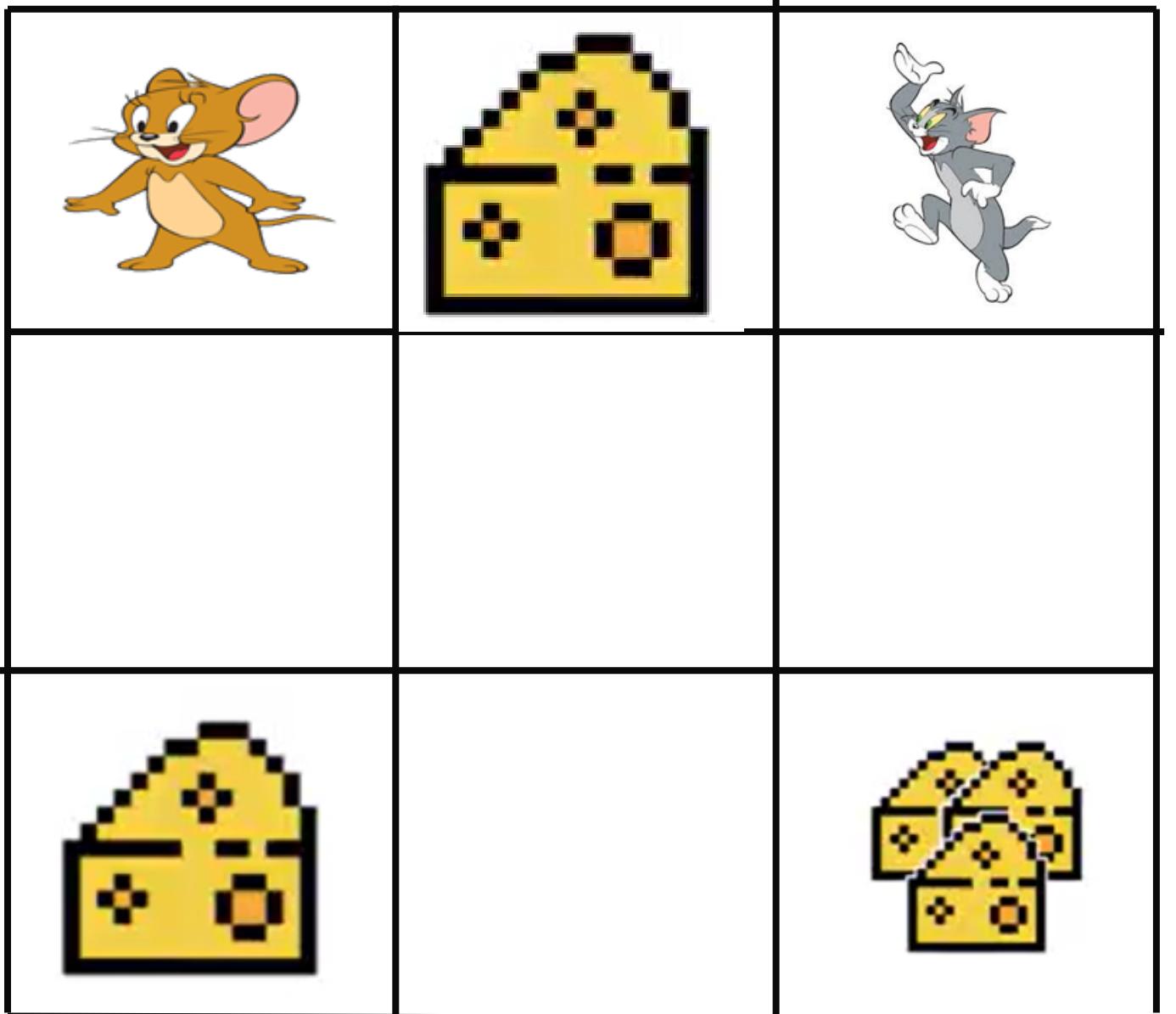
What is Reinforcement Learning (RL)?



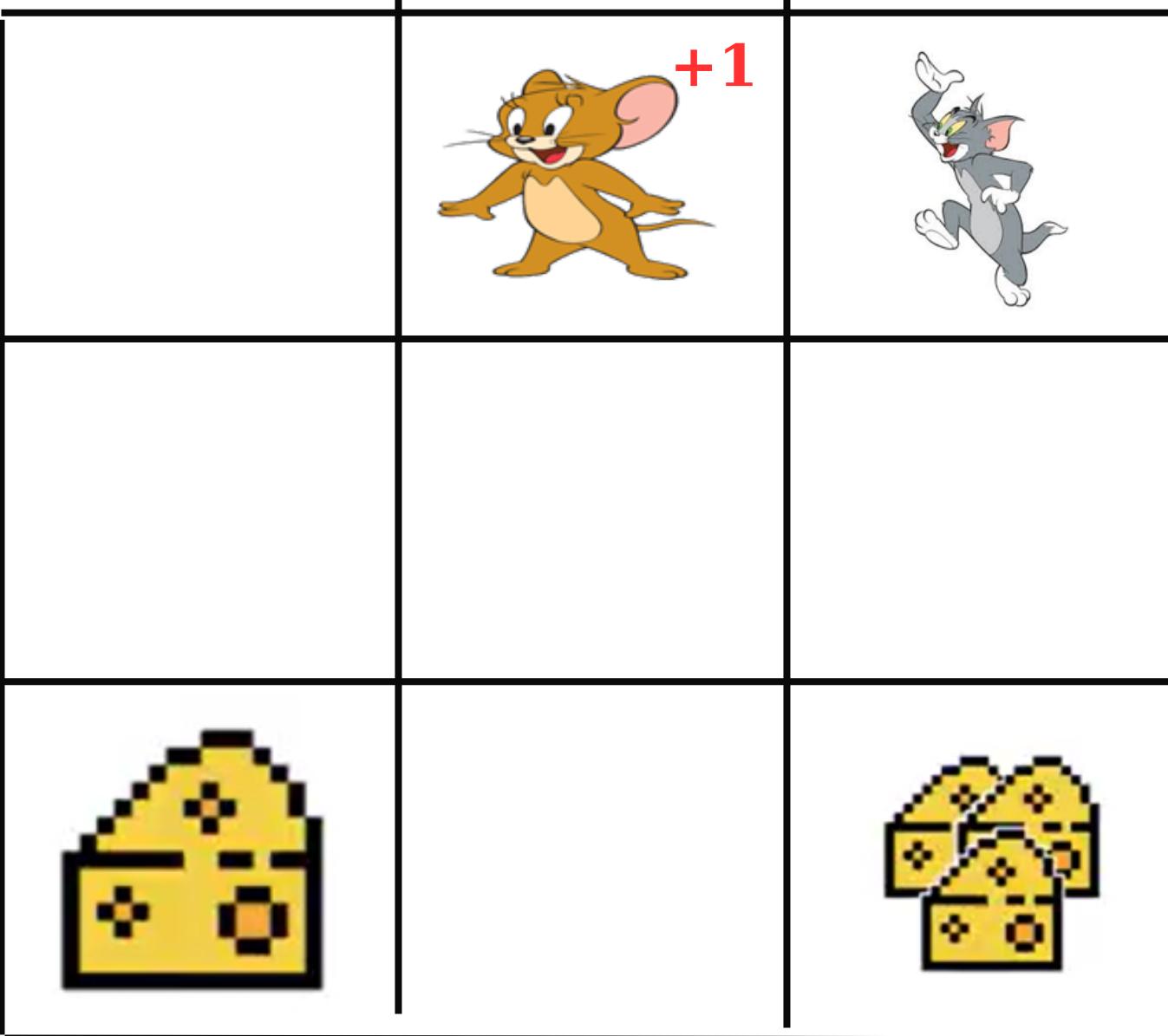
RL: a branch of machine learning where an agent makes decisions by interacting with an environment. Through this interaction , the agent receives rewards or penalties based on its actions , guiding it to learn an optimal policy for maximizing cumulative rewards over time

 Reinforcement learning idea

Points:0



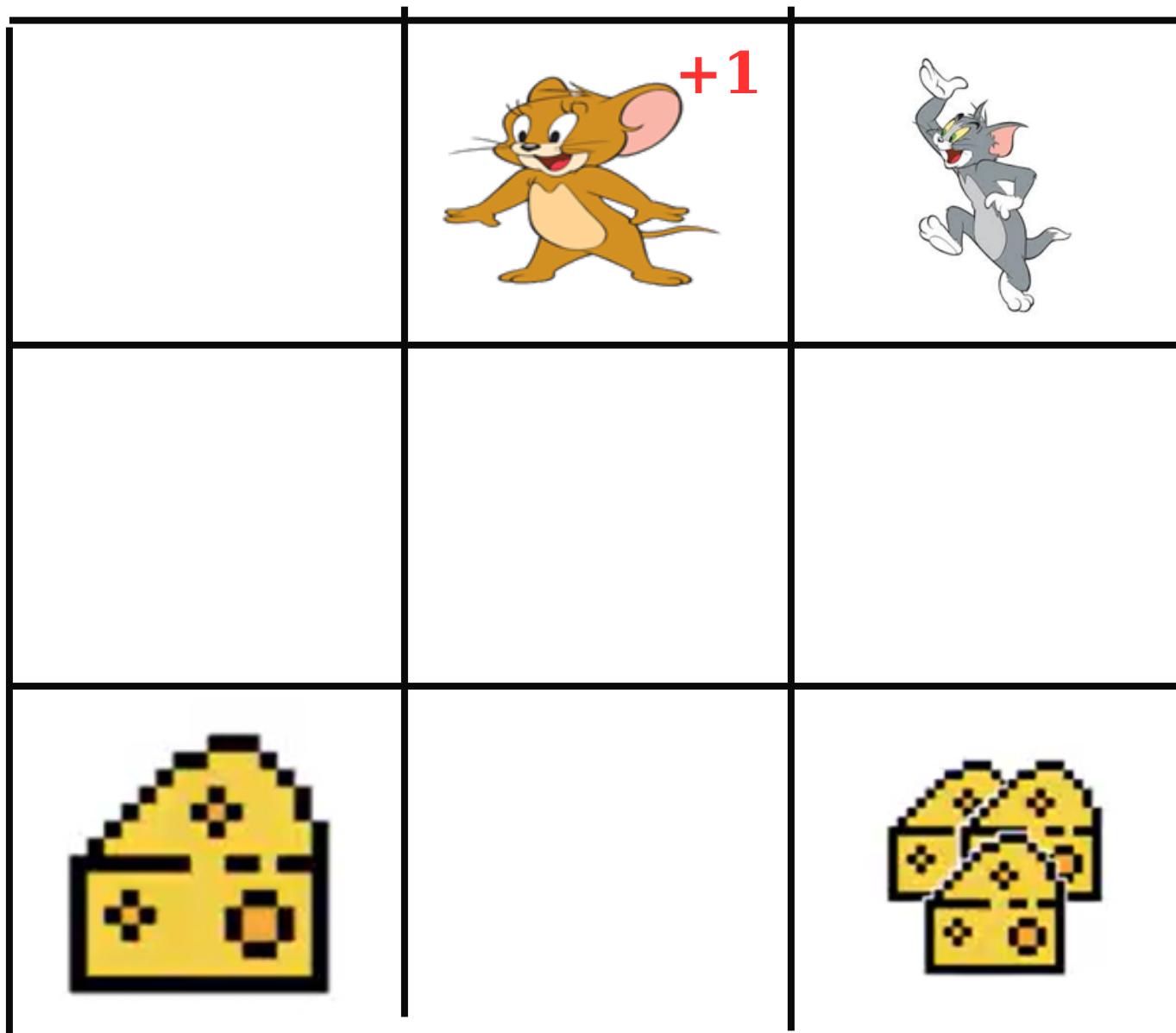
Points:1



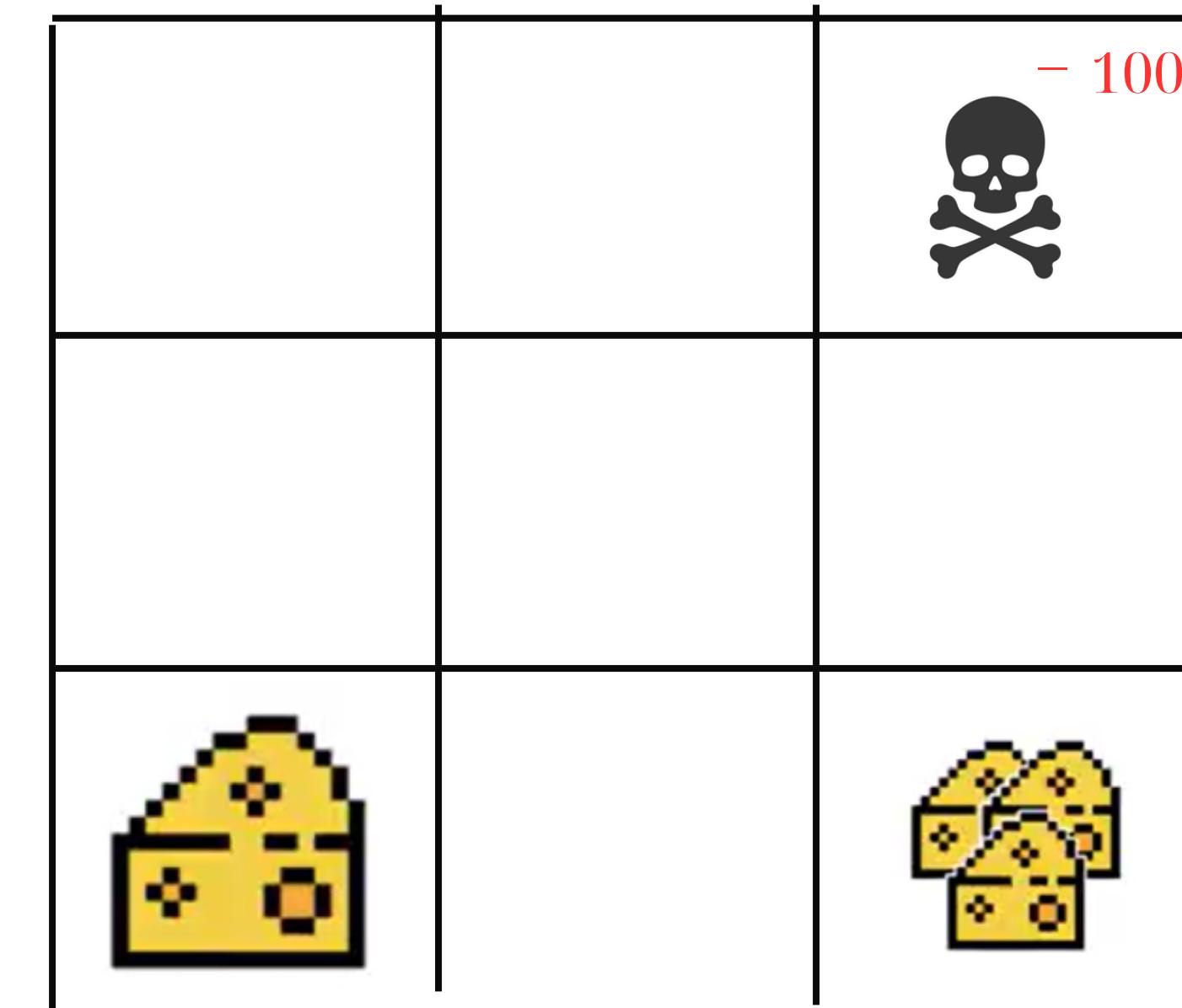


Reinforcement learning idea

Points: 1



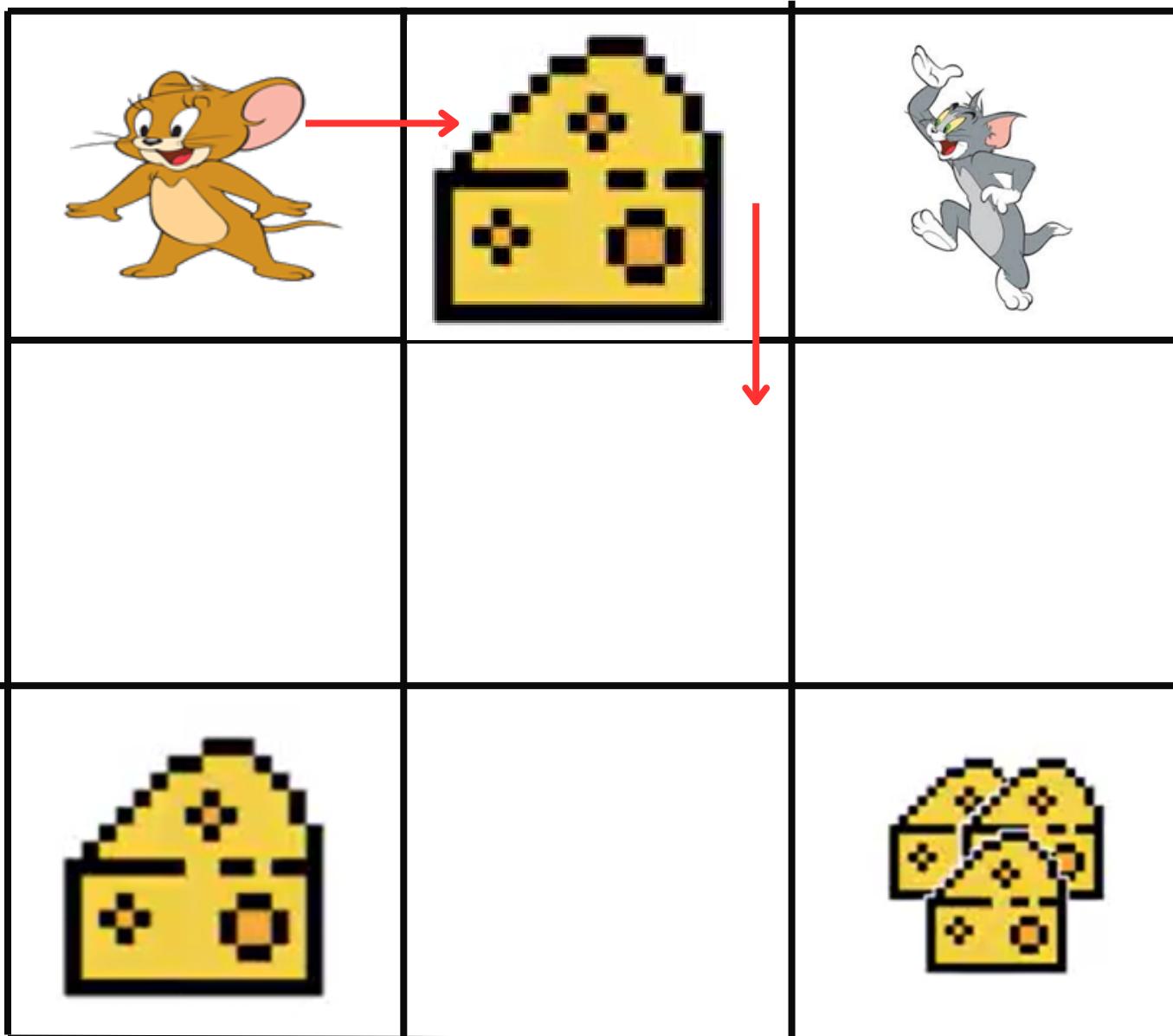
Points: -99



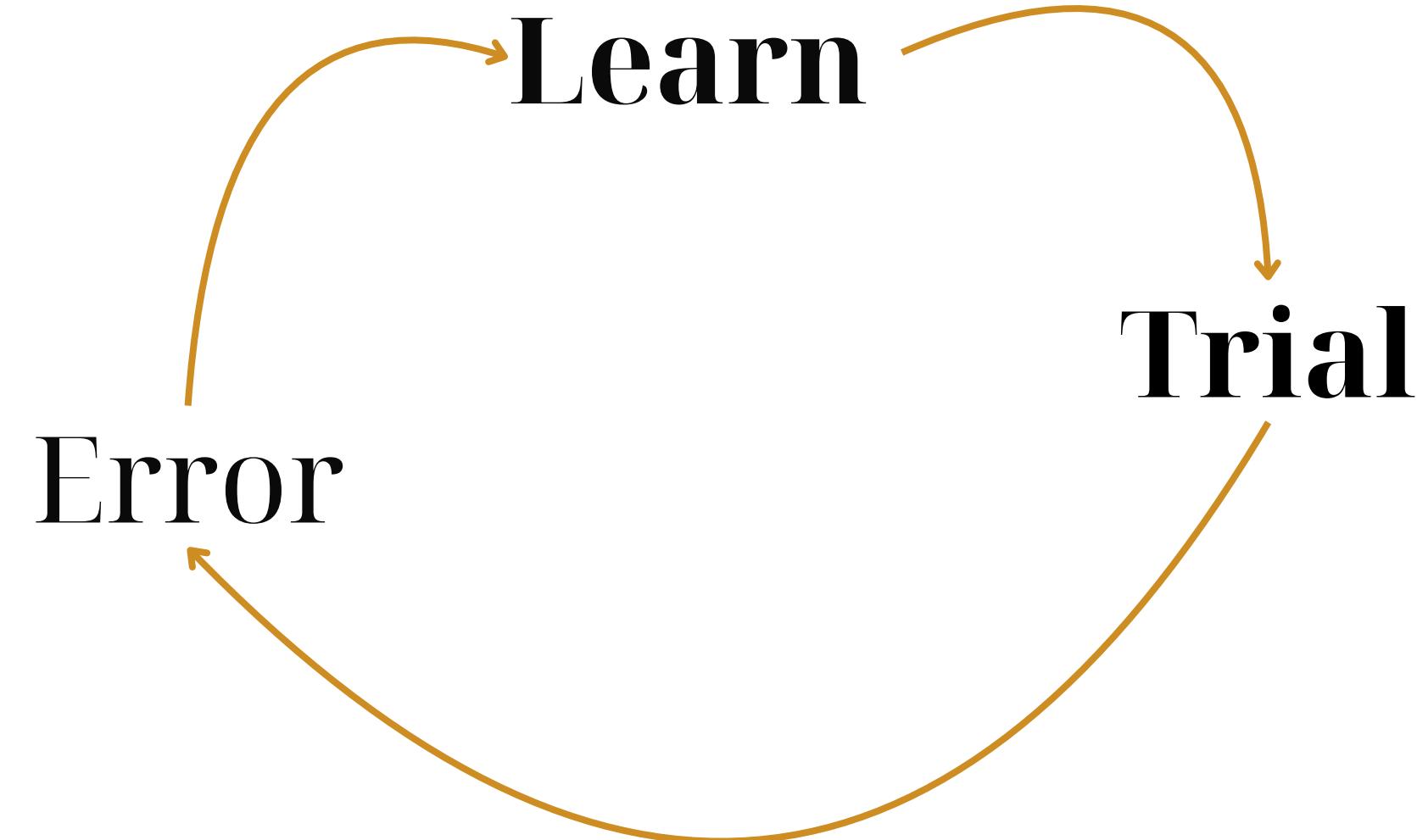


Reinforcement learning idea

Points:0



Create an agent that could interact with the environment, learn to reach the goal and obtain maximum rewards



Reset game and play again with new knowledge

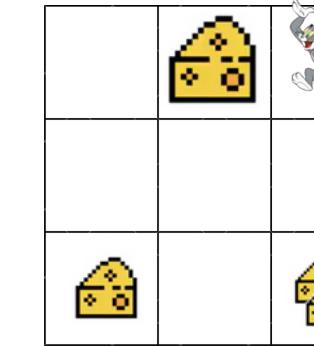


Reinforcement Learning Framework



Agent

The decision-making entity that learns to perform actions based on the current state of the environment.



Environment

The context or setting in which the agent operates, providing feedback to the agent in the form of rewards or penalties.

State

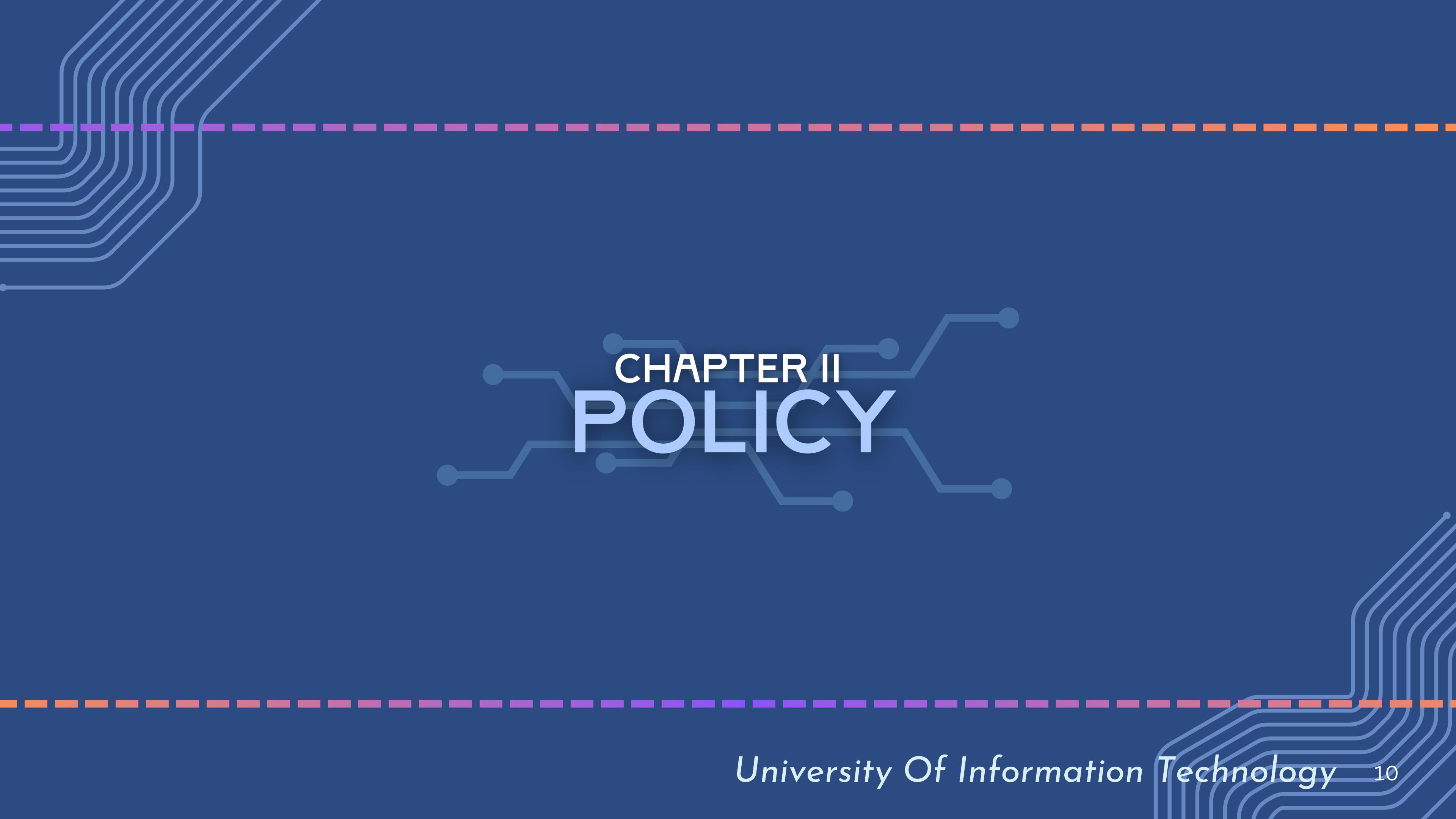
Represents the specific situation or configuration the agent encounters in the environment

Action

The set of possible moves or decisions the agent can take in response to the current state

Reward

A scalar value that quantifies the desirability of an action given a particular state, guiding the agent's learning process

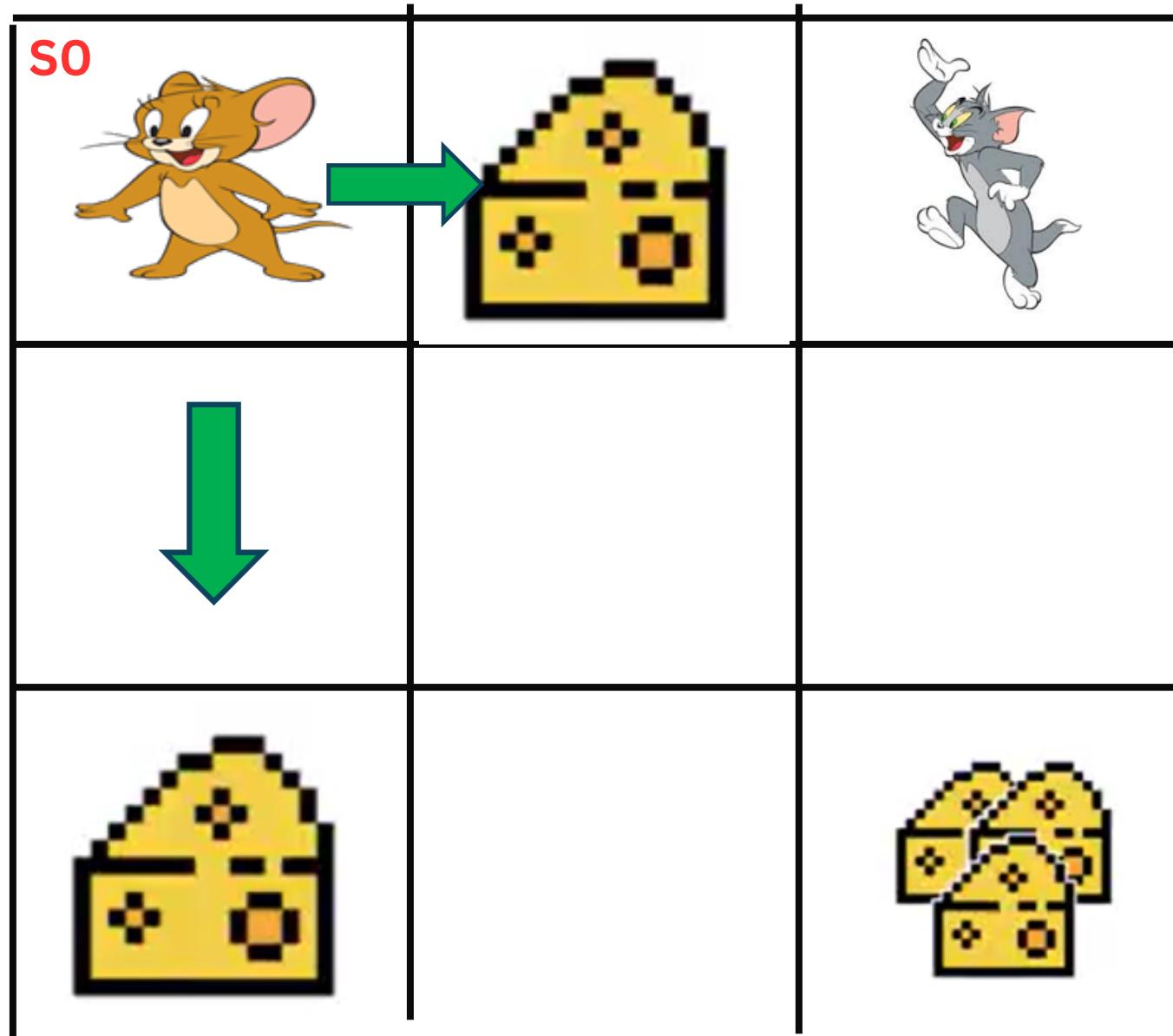


CHAPTER II

POLICY



Policy



Possible actions at S0:
Right, Down

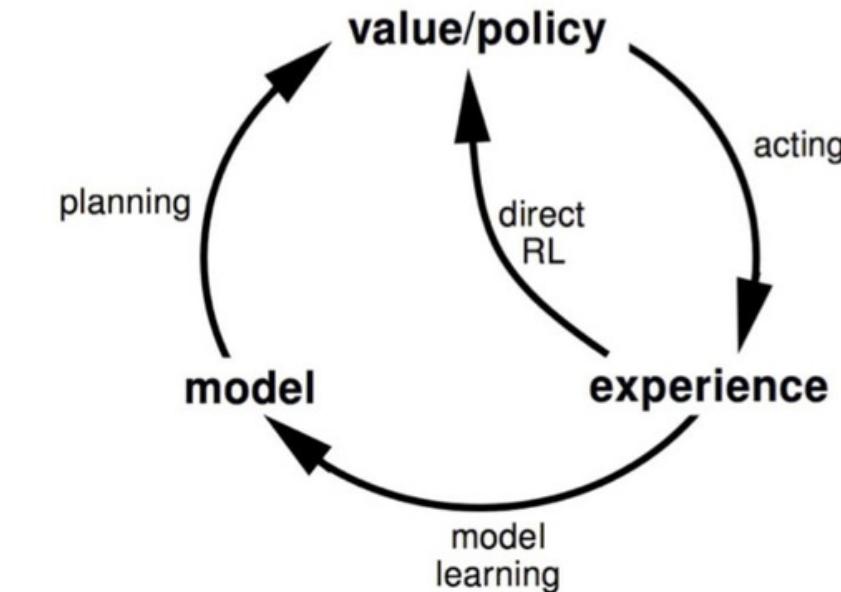
Given state S, our agent
will have **many possible
actions A.**

$$R_T = \sum_{i=0}^T r_{t+1} = r_t + r_{t+1} + \dots + r_T$$

- In RL, we attempt to maximize the expected cumulative reward.

Need a way so that at every state, the agent could be able to choose action that leads to the highest expected cumulative reward.

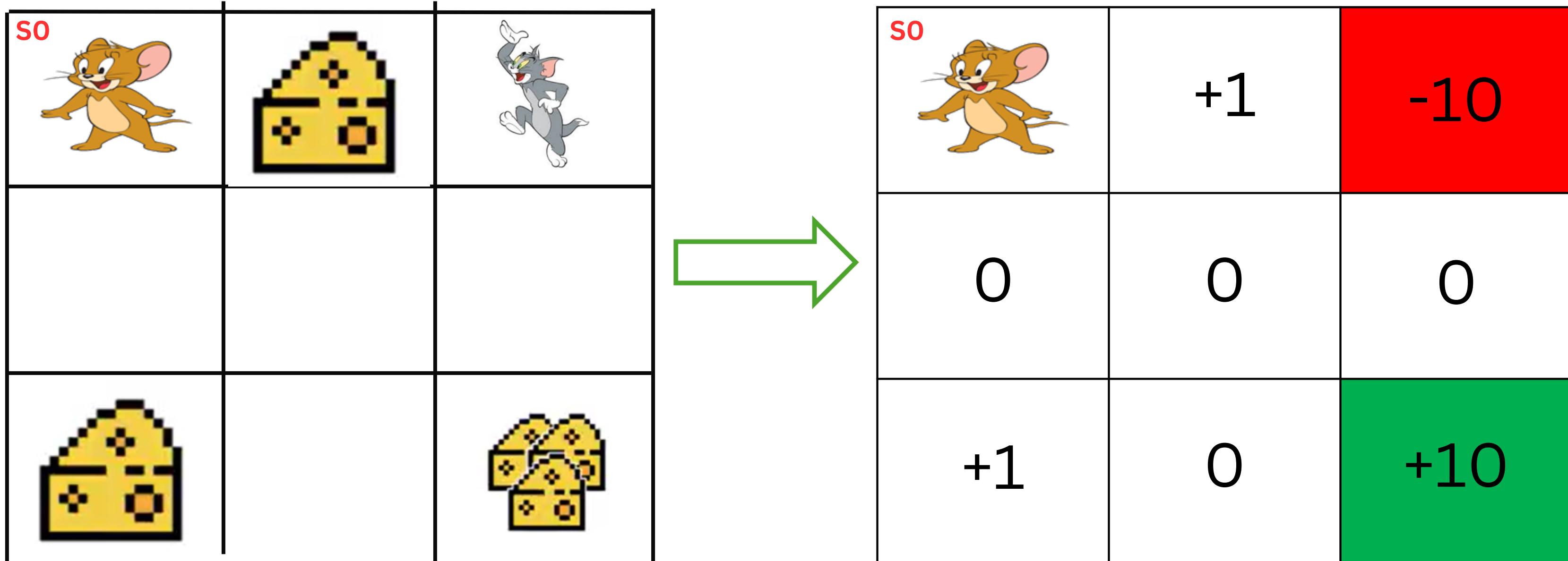
π
Policy



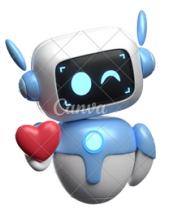
29



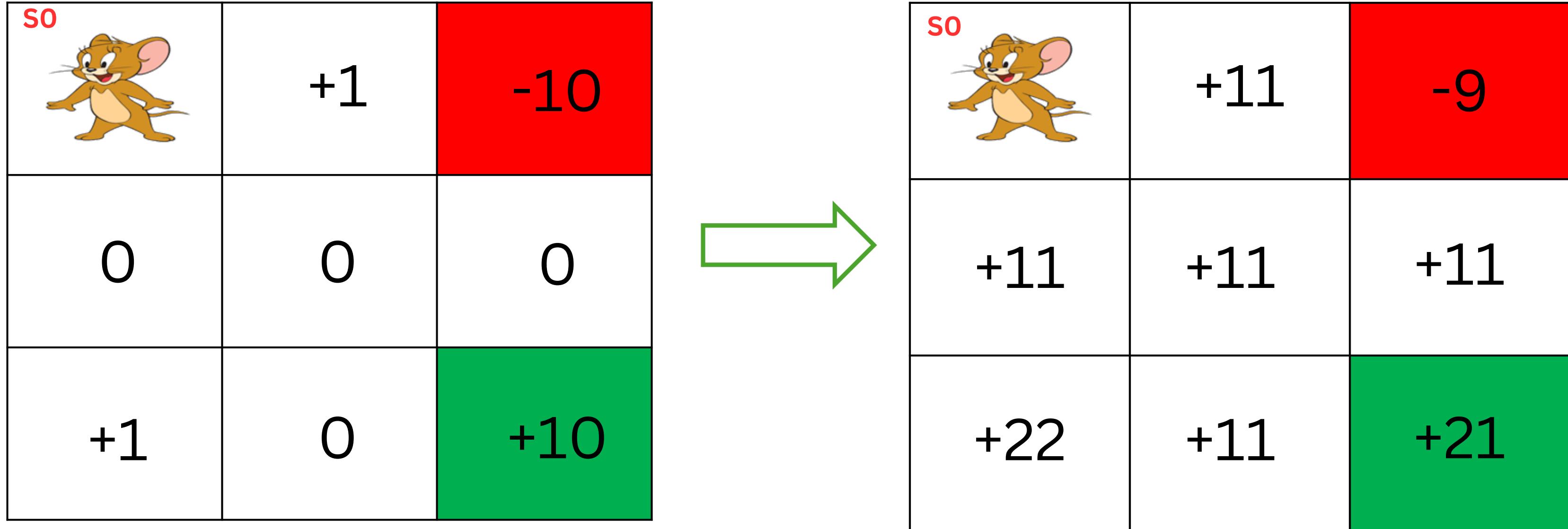
Policy



The agent must avoid the penalty (-10) while seeking the highest reward (+10) to achieve an optimal solution



Policy

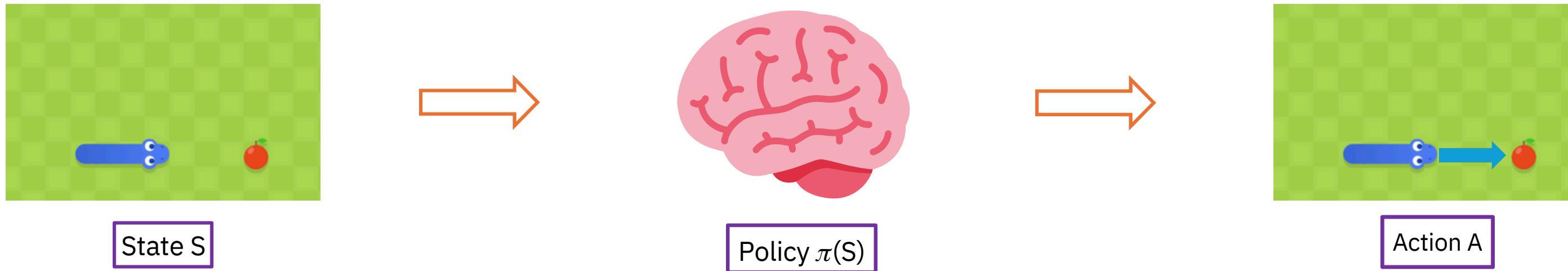


State-value function: Expected cumulative return the agent can get if it starts at that state, and act according to the policy

$$V^\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s]$$

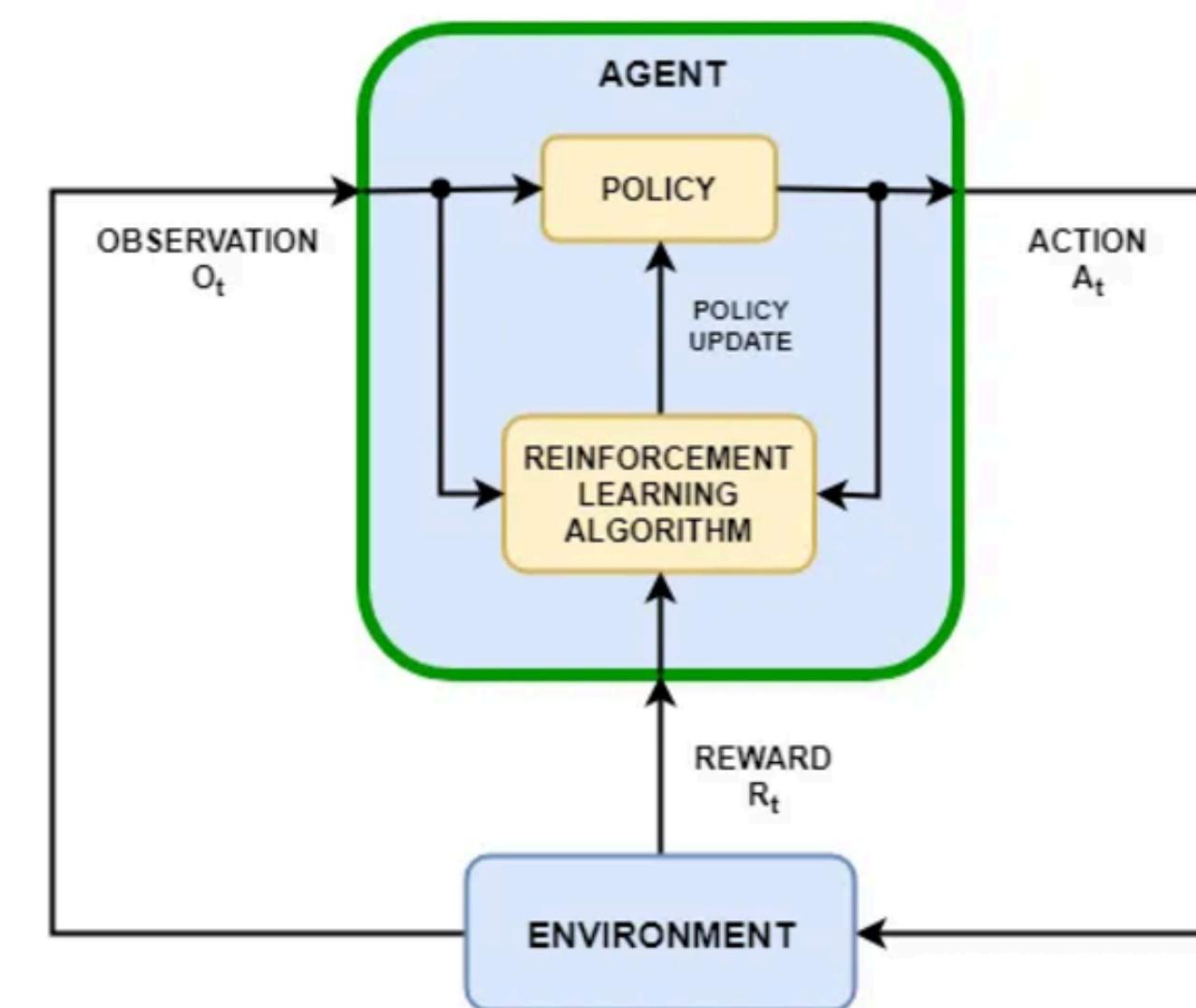


Policy



Policy π : the agent's behavior, define how agent chooses action in response to the current state.

In reinforcement learning, the agent refines its policy through interaction with the environment.



Typical RL loop (image from mathworks.com)



Deterministic Policy

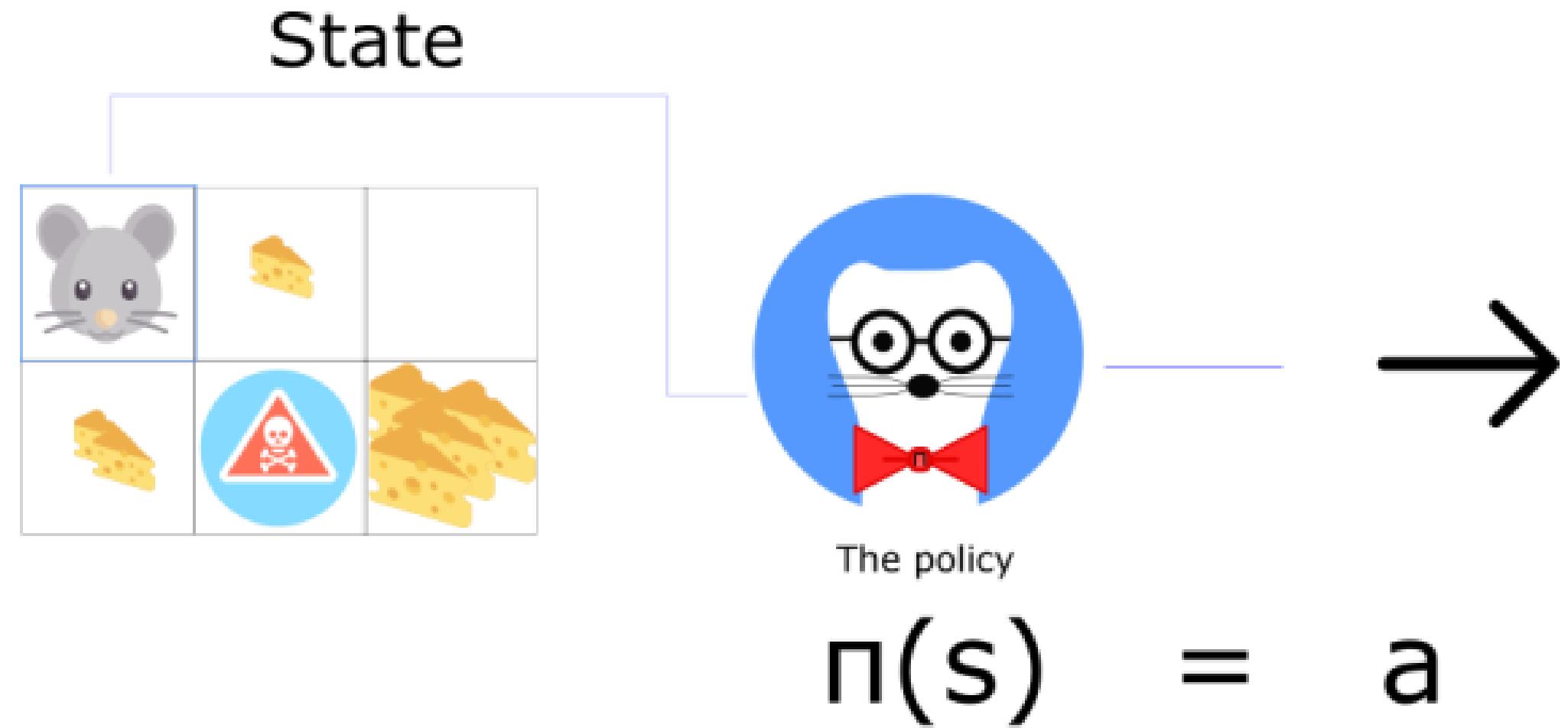
Deterministic Policy

A deterministic policy maps each state to a single action with certainty, ensuring the agent always takes the same action for a given state.

$$a = \pi(s)$$

Advantages:

- + Easy to interpret and implement.
- + Ideal for tasks requiring consistent actions for the same state



Represented as $\pi: S \rightarrow A$, it assigns a specific action $a \in A$ to each state $s \in S$.



Stochastic Policy

Stochastic Policy

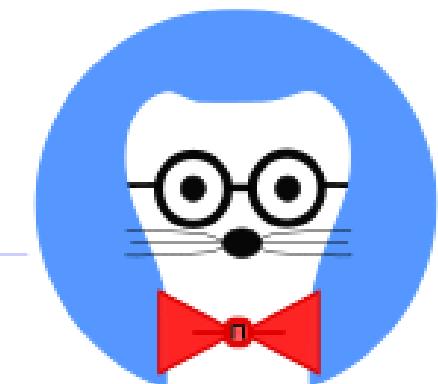
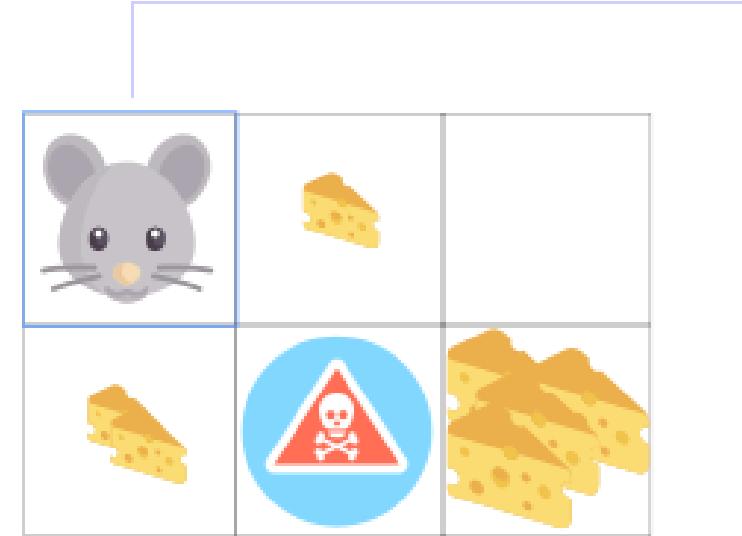
A stochastic policy maps each state to a probability distribution over actions. The agent selects an action randomly based on these probabilities.

$$\pi(a|s) \models P[A|s]$$

Advantages:

- + Handles uncertainty in the environment.
- + Prevents exploitability in repeated play

State and action →



The policy

→ 0.7

Probability of taking that action at that state

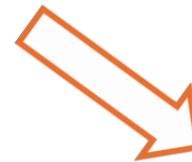
$$\Pi(a|s) = P(at|st)$$

Represented as $\pi:S \times A \rightarrow [0,1]$, where $\pi(s,a)$ is the probability of taking action a in state s .

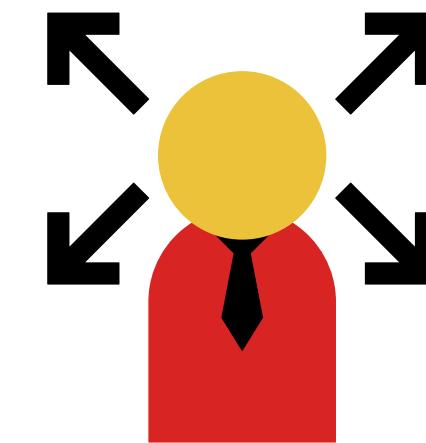


How to find optimal policy?

Value-based



Policy-based



Goal: Maximize the expected cumulative reward.



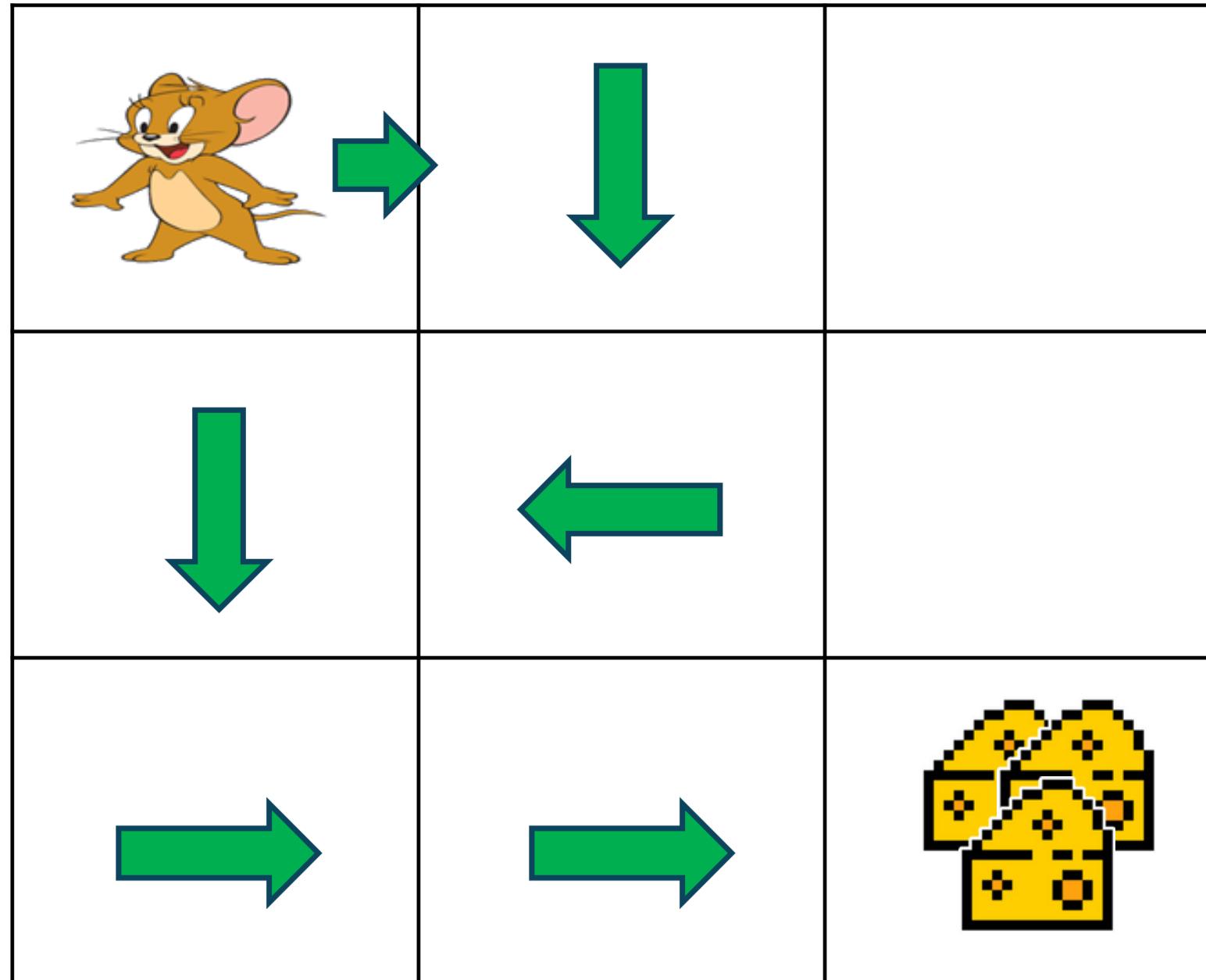
Value-based methods

	-3	-10
-3	-2	-1
-2	-1	

Value-based methods: train the agent to learn which state is more valuable and take the action that leads to it.



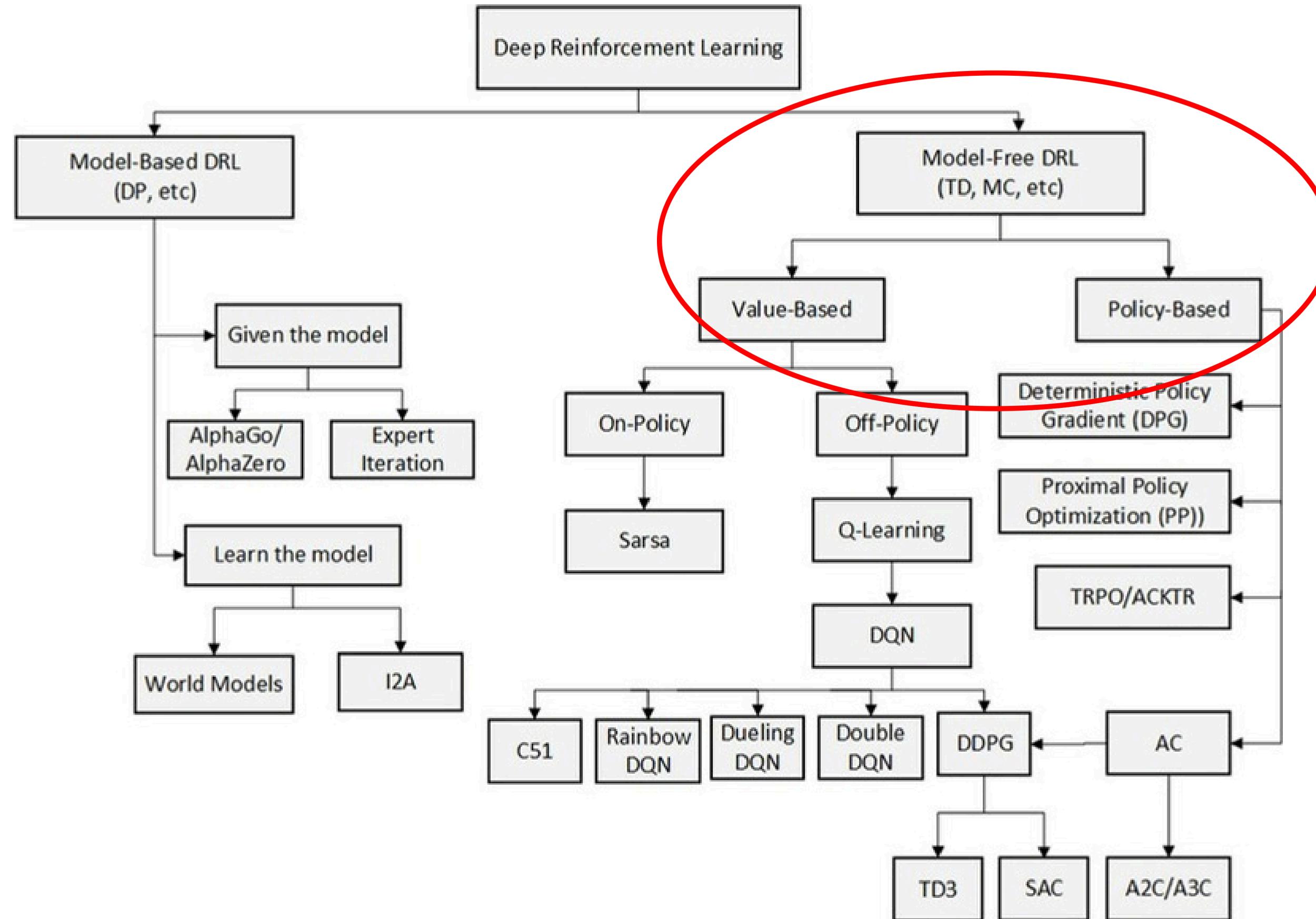
Policy-based methods

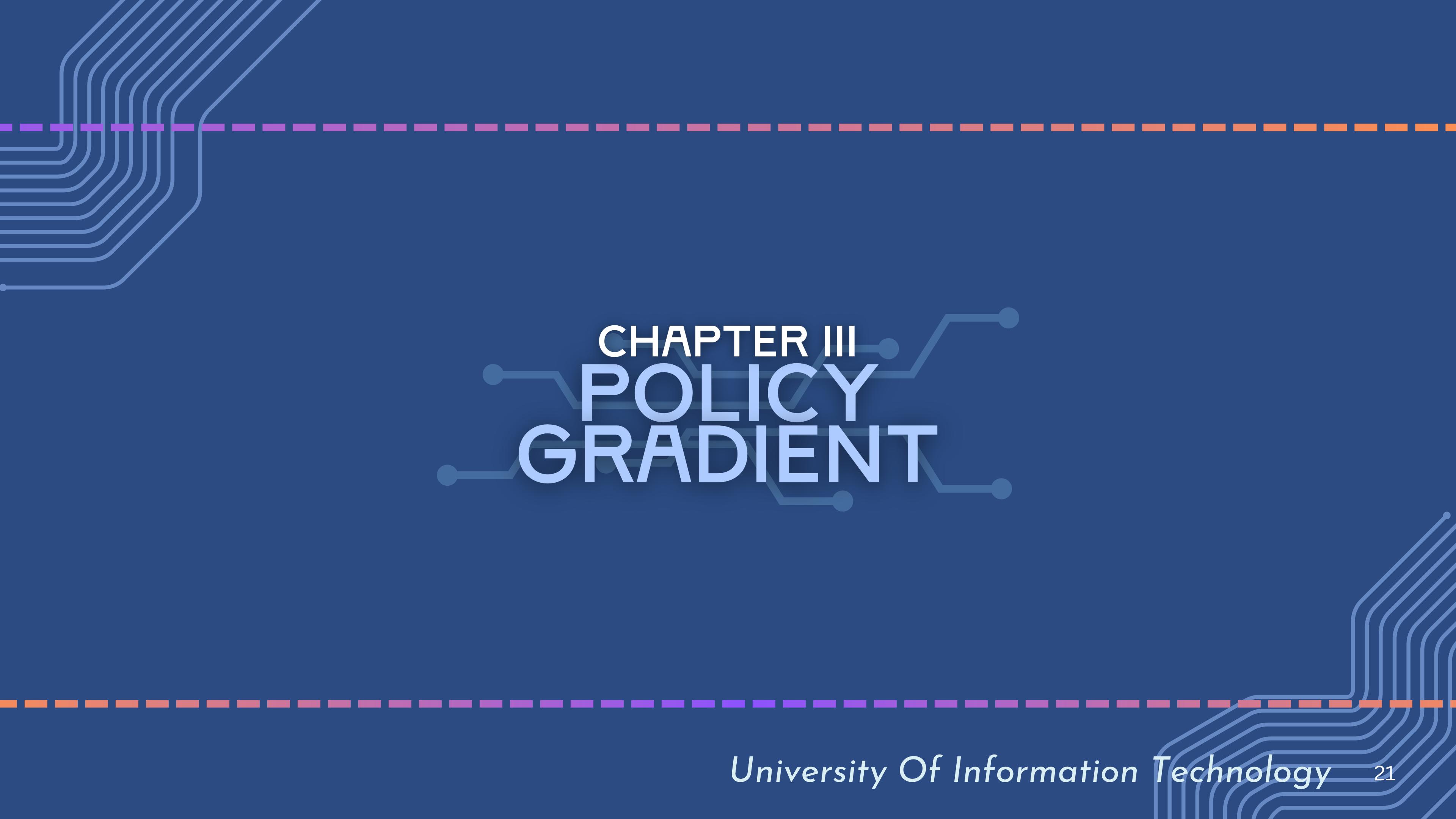


Policy-based methods: train the agent to learn which action to take, given a state.



Policy-based methods



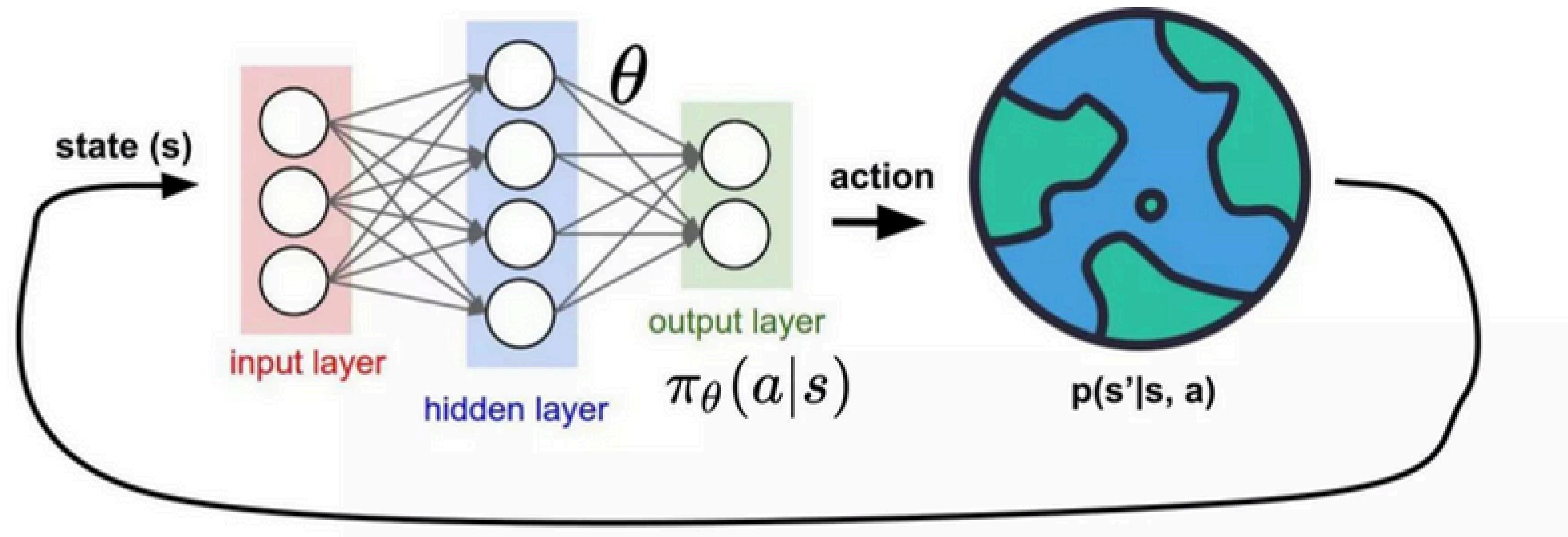


CHAPTER III

POLICY GRADIENT



Policy Gradient Approach





Makrov Decision Process (MDP)

Definition:

A model for sequential decision making when outcomes are uncertain.

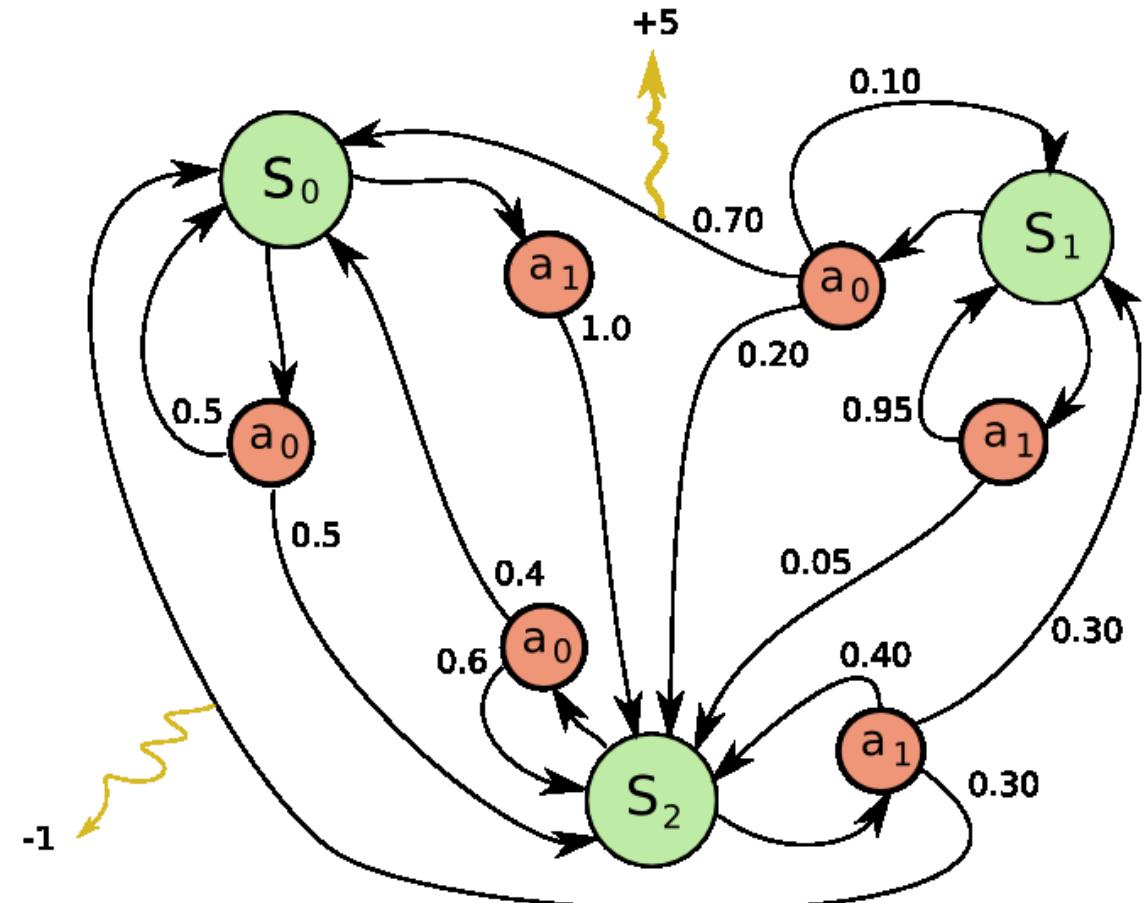
A Markov decision process is a 4-tuple (S, A, P_a, R_a)

S is a set of states called the state space.

A is a set of actions called the action space.

P_a is, on an intuitive level, the probability that action in state at time t will lead to state at time t+1

R_a is the immediate reward (or expected immediate reward) received after transitioning from state to state , due to action.



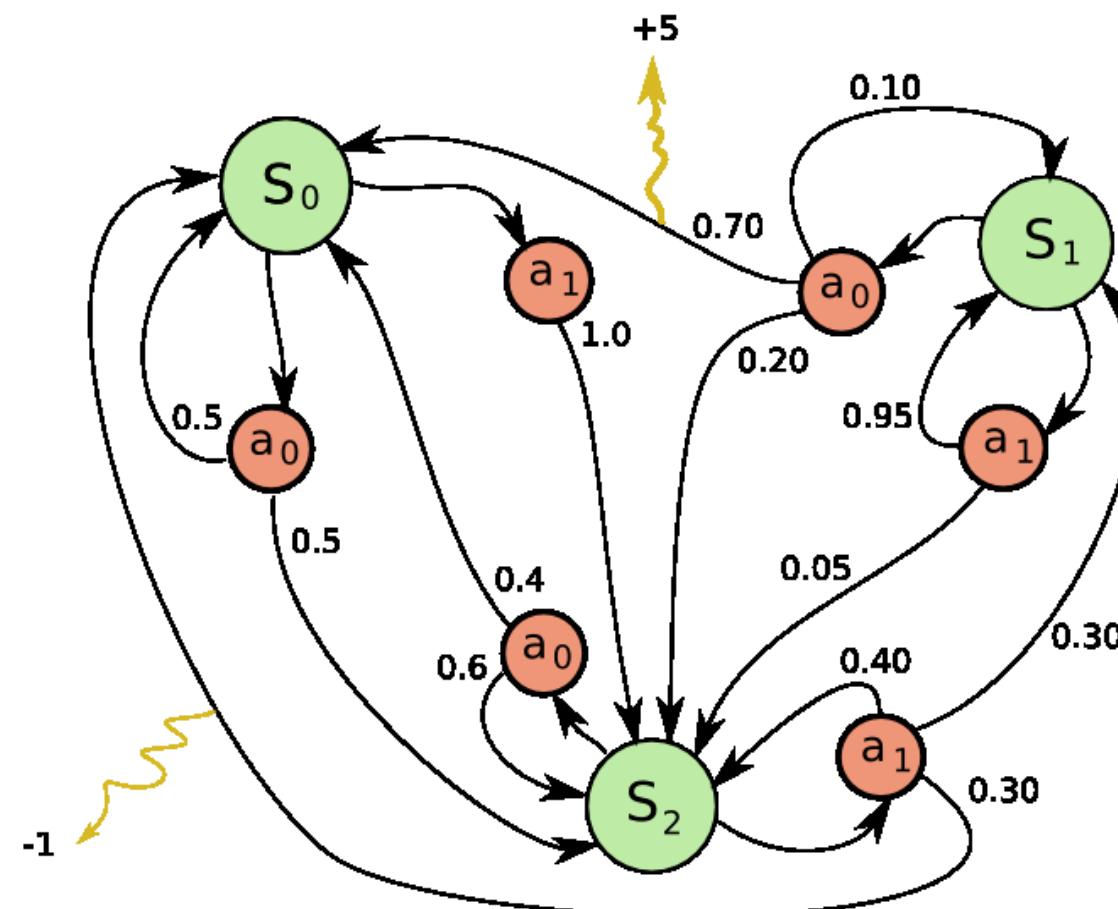
H2. Markov Decision Process (MDP).

Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows)

The process is called a "decision process" because it involves making decisions that influence these state transitions, extending the concept of a Markov chain into the realm of decision-making under uncertainty.



Makrov Decision Process (MDP)



A Markov chain is a mathematical system that experiences transitions from one state to another according to certain probabilistic rules. The defining characteristic of a Markov chain is that no matter how the process arrived at its present state, the possible future states are fixed.

→ **The probability of transitioning to any particular state is dependent solely on the current state and time elapsed**

H2. Markov Decision Process (MDP).
Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows)

A Markov chain is a stochastic process, but it differs from a general stochastic process in that a Markov chain must be "**memory-less**". That is, (the probability of) future actions are not dependent upon the steps that led up to the present state.

→ This is called **the Markov property**.



Makrov Decision Process (MDP)

A state S_t is Markov if and only if,

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, S_2, \dots, S_t]$$

A Markov Process is defined by (S, P) where S are the states, and P is the state-transition probability. It consists of a sequence of random states S_1, S_2, \dots where all the states obey the Markov Property.

$$P = \begin{bmatrix} p_{11} & p_{12} & \dots & p_{1n} \\ p_{21} & p_{22} & \dots & p_{2n} \\ \dots & \dots & & \\ p_{n1} & p_{n2} & \dots & p_{nn} \end{bmatrix}$$

State Transition Probability Matrix

Each row in the matrix represents the probability from moving from our original or starting state to any successive state. Sum of each row is equal to 1.

Stationary Distribution of a Markov Chain is a probability distribution that remains unchanged in the Markov chain as time progresses. Typically, it is represented as a row vector π whose entries are probabilities summing to 1, and given transition matrix P , it satisfies:

$$\pi = \pi P$$



Makrov Decision Process (MDP)

Makrov Reward Process:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

$$\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s]$$

The state reward R_t is the expected reward over all the possible states that one can transition to from state s . This reward is received for being at the state S_t . By convention, it's said that the reward is received after the agent leaves the state, and is hence regarded as R_{t+1} .

Makrov Decision Process:

$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

This is essentially MRP with actions. Introduction to actions elicits a notion of control over the Markov process. Previously, the state transition probability and the state rewards were more or less stochastic (random). However, now the rewards and the next state also depend on what action the agent picks. Basically, the agent can now control its own fate (to some extent).



Notations

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

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q^\pi(s, a) = E_\pi[G_t | S_t = s]$$

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



Policy Objective Function

Goal: given policy $\pi_\theta(s, a)$ with parameters θ , find best θ

But how do we measure the quality of a policy π_θ ?

In episodic environments we can use the **start value**

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta} [v_1]$$

In continuing environments we can use the **average value**

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

Or the **average reward per time-step**

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \mathcal{R}_s^a$$

where $d^{\pi_\theta}(s)$ is **stationary distribution** of Markov chain for π_θ



Policy Gradient

Let $J(\theta)$ be any policy objective function

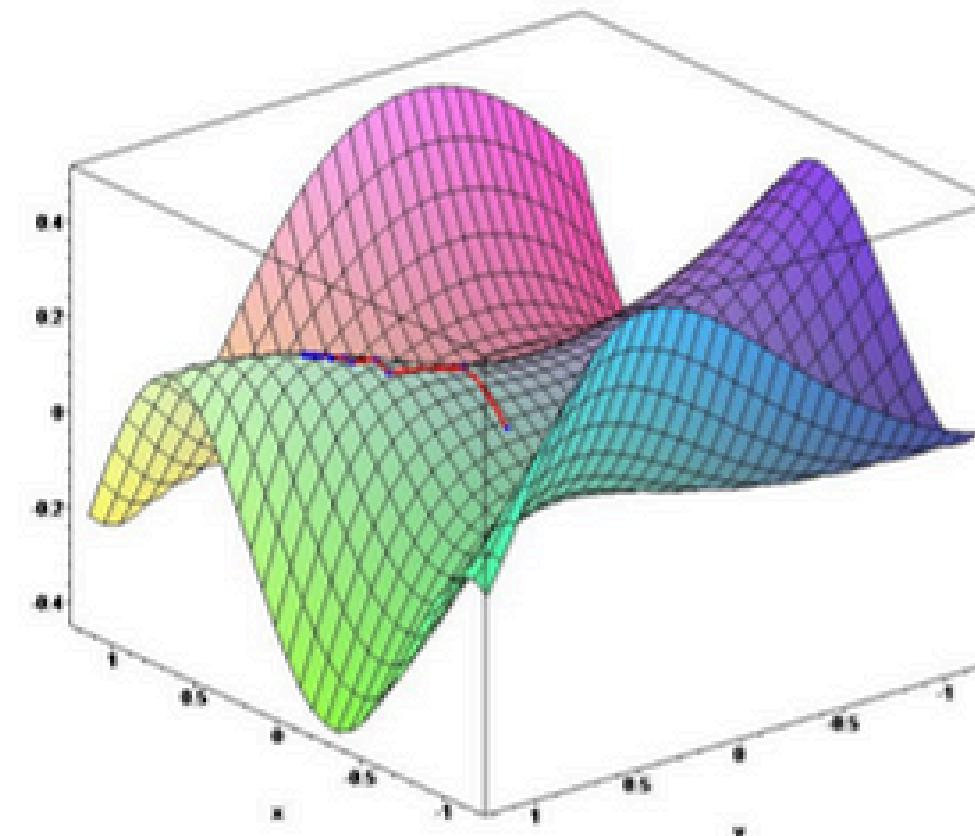
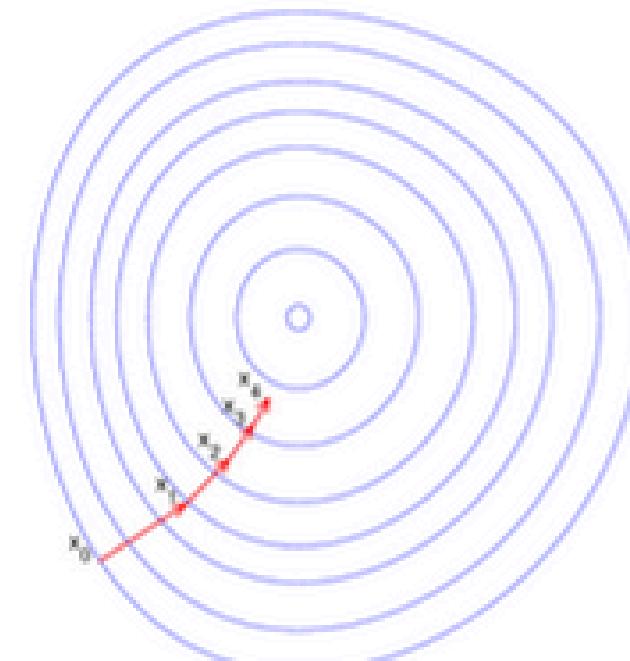
Policy gradient algorithms search for a *local* maximum in $J(\theta)$ by ascending the gradient of the policy, w.r.t. parameters θ

$$\Delta\theta = \alpha \nabla_{\theta} J(\theta)$$

Where $\nabla_{\theta} J(\theta)$ is the **policy gradient**

$$\nabla_{\theta} J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$$

and α is a step-size parameter





One-step Markov Decision Process

Consider a simple class of one-step MDPs

- Starting in state $s \sim d(s)$
- Terminating after one time-step with reward $r = \mathcal{R}_{s,a}$

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta} [r] \\ &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \mathcal{R}_{s,a} \end{aligned}$$

$$\begin{aligned} \nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \end{aligned}$$

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) r] \end{aligned}$$



Policy Gradient theorem

Replaces instantaneous reward r with long-term value $Q^\pi(s, a)$

Policy gradient theorem applies to start state objective,
average reward and average value objective

Theorem

*For any differentiable policy $\pi_\theta(s, a)$,
for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma} J_{avV}$,
the policy gradient is*

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)]$$



Monte-Carlo Policy Gradient (REINFORCE)

Update parameters by stochastic gradient ascent

Using policy gradient theorem

Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

function REINFORCE

 Initialise θ arbitrarily

for each episode $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**

for $t = 1$ to $T - 1$ **do**

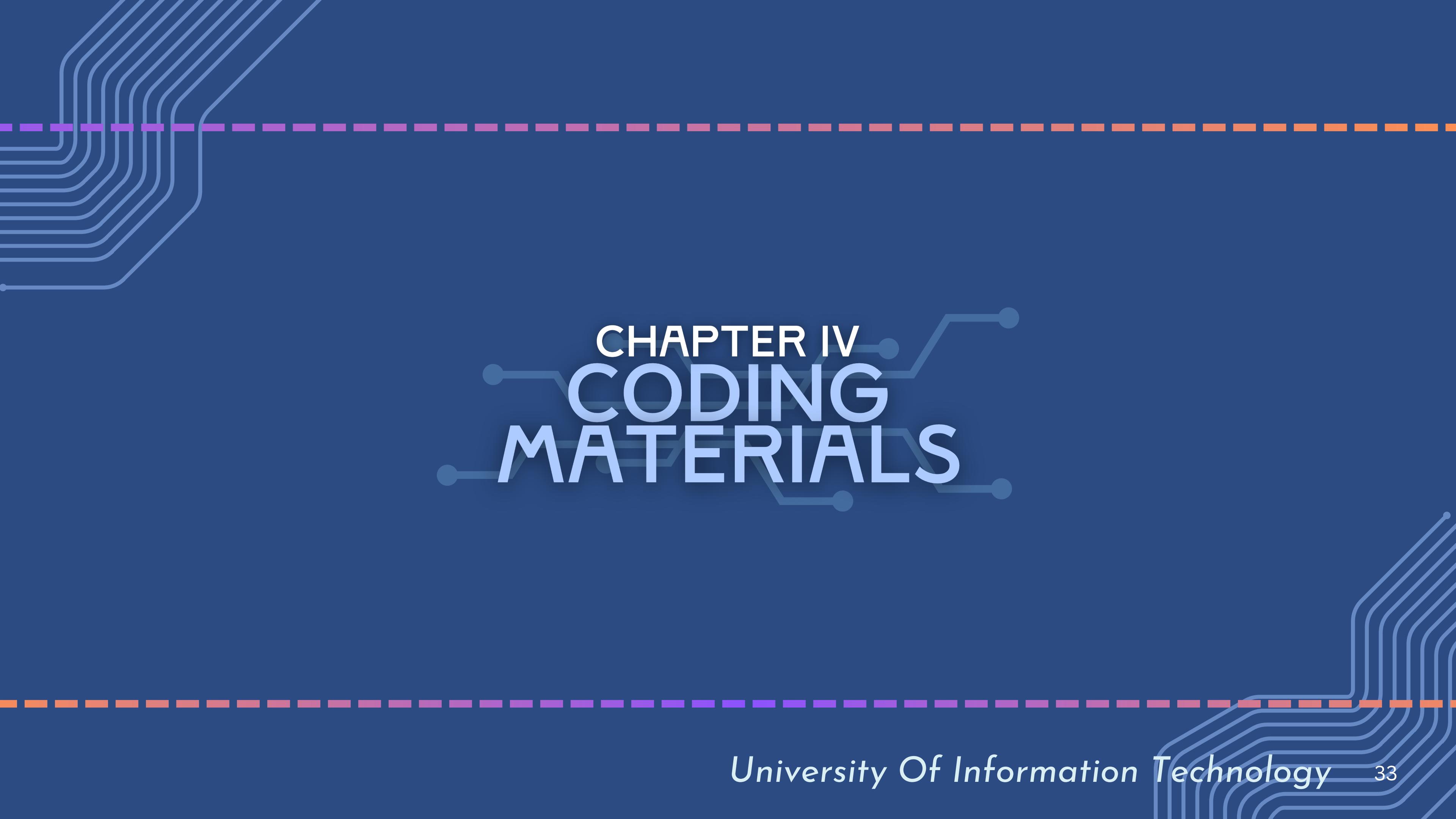
$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$

end for

end for

return θ

end function

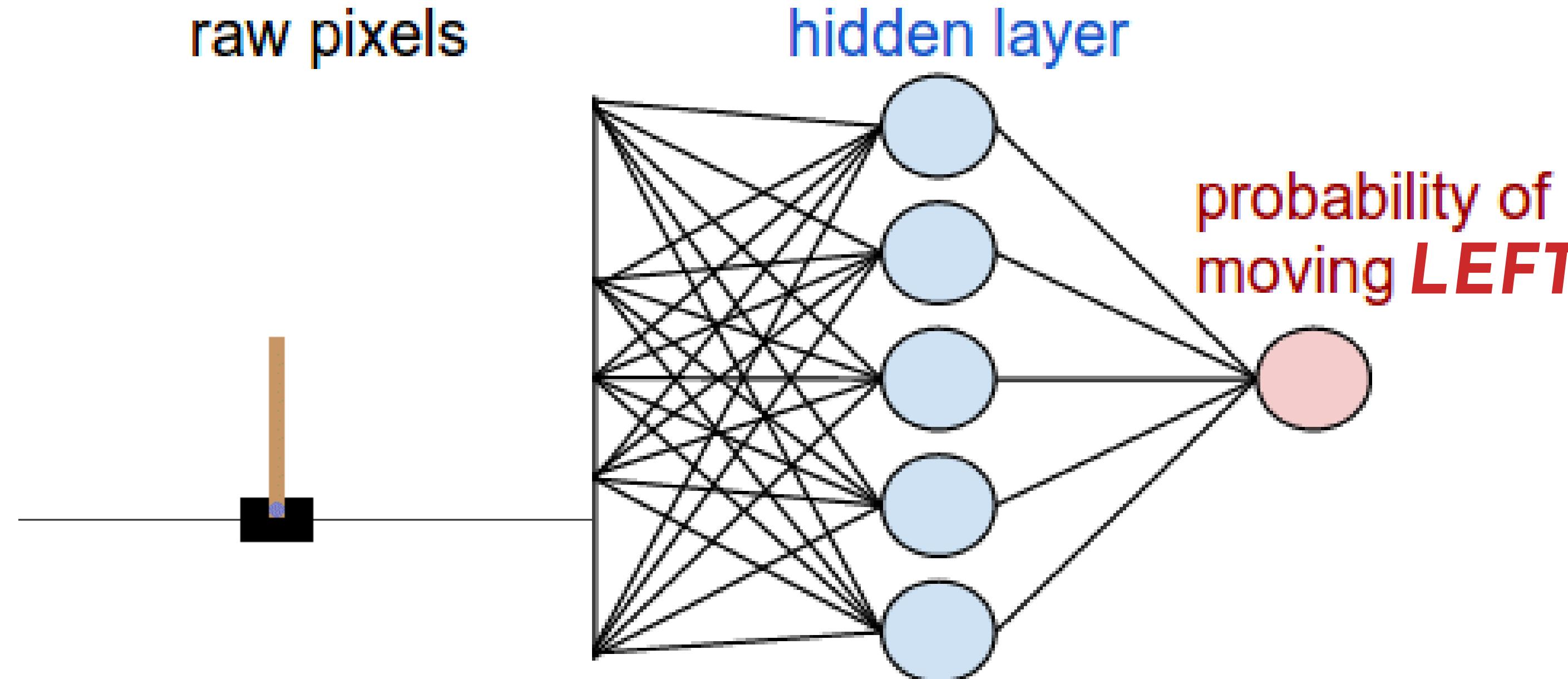


CHAPTER IV

CODING MATERIALS



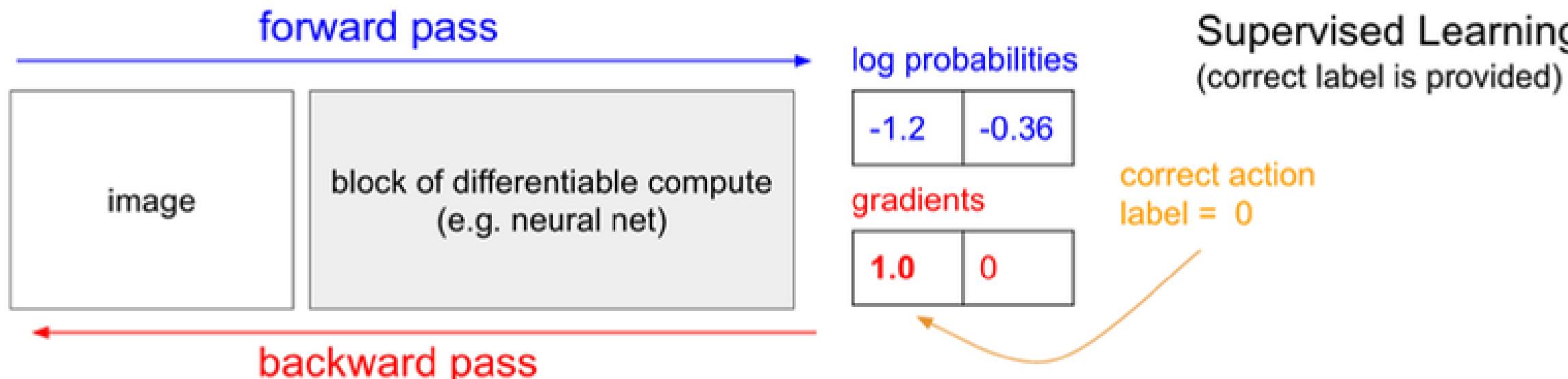
Reinforcement Training Models





Reinforcement Training Models

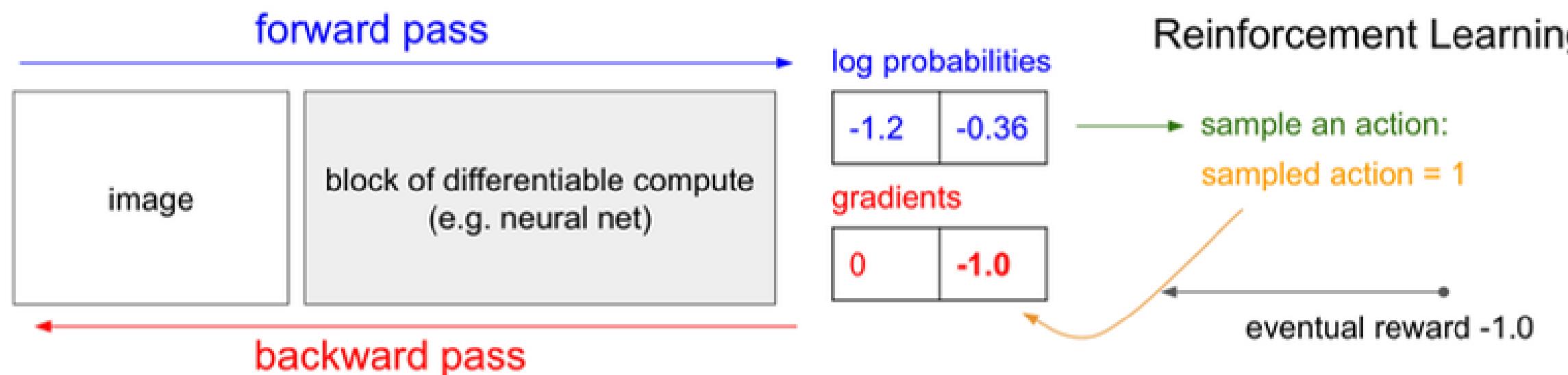
Supervised Learning (Classification):



$$\nabla_W \log p(y = \text{LEFT} | x)$$

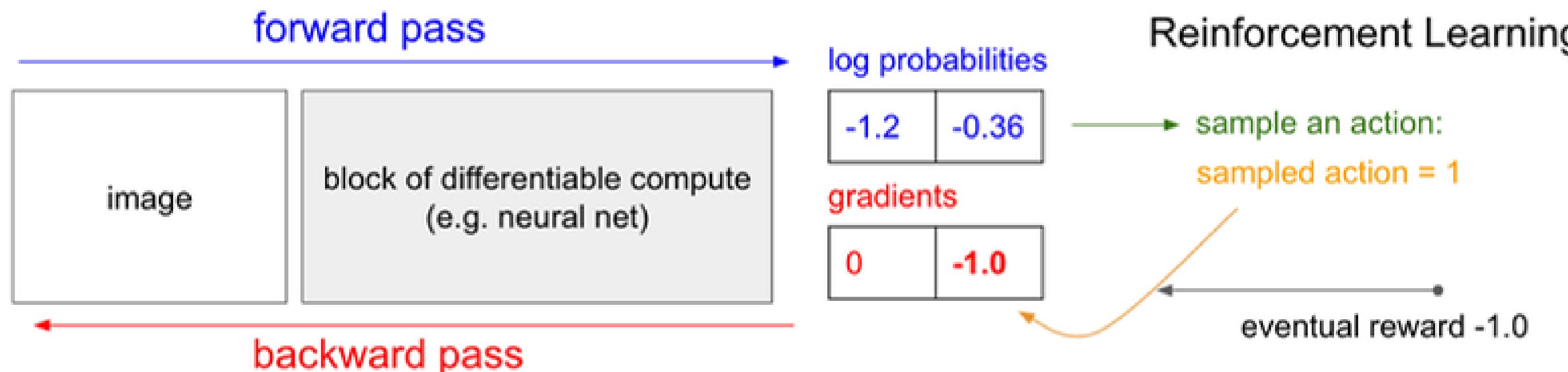
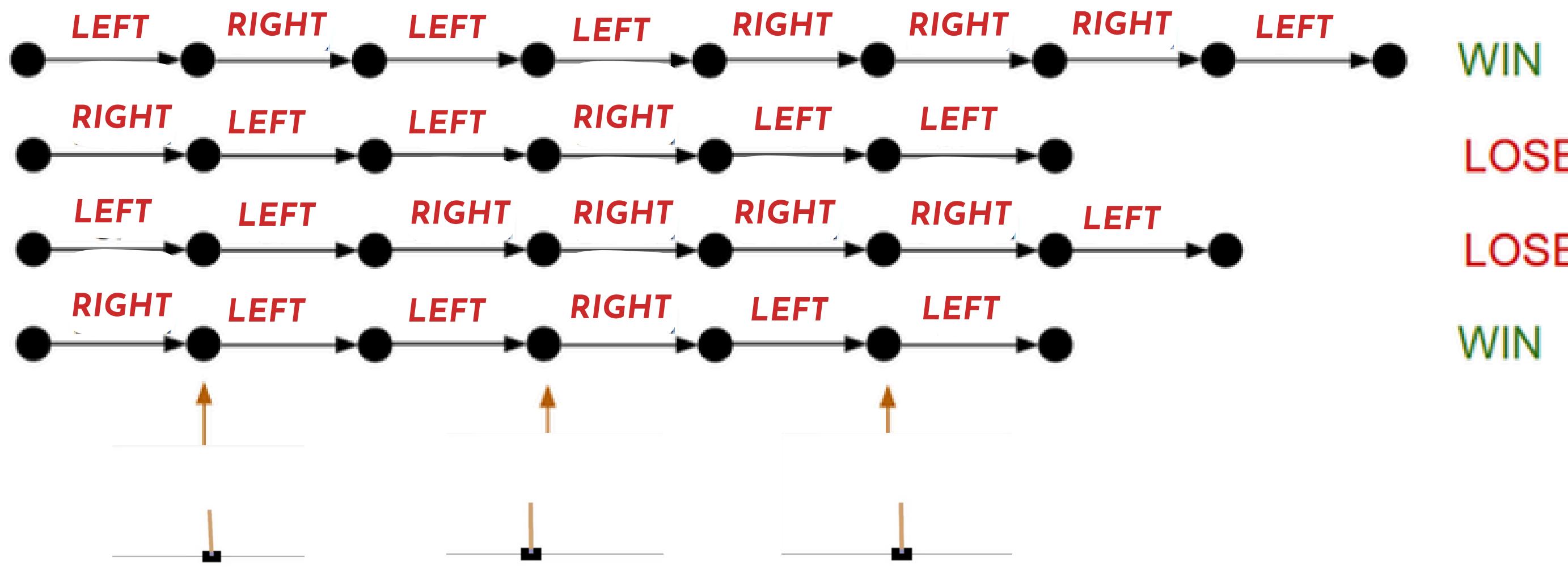
This gradient would tell us how we should change every one of our million parameters to make the network slightly more likely to predict UP.

Reinforcement Learning - Policy Gradients:





Reinforcement Training Models





Coding Materials

Install and Import The Essentials

```
import argparse
import gym
import numpy as np
from itertools import count
from collections import namedtuple

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical
```

Initialise The Environment

```
# Cart Pole

parser = argparse.ArgumentParser(description='PyTorch actor-critic example')
parser.add_argument('--gamma', type=float, default=0.99, metavar='G',
                    help='discount factor (default: 0.99)')
parser.add_argument('--seed', type=int, default=543, metavar='N',
                    help='random seed (default: 543)')
parser.add_argument('--render', action='store_true',
                    help='render the environment')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='interval between training status logs (default: 10)')
args = parser.parse_args()

env = gym.make('CartPole-v1', render_mode='human')
env.reset(seed=args.seed)
torch.manual_seed(args.seed)

SavedAction = namedtuple('SavedAction', ['log_prob', 'value'])
```



Coding Materials

```
class Policy(nn.Module):
    """
    implements both actor and critic in one model
    """

    def __init__(self):
        super(Policy, self).__init__()
        self.affinel = nn.Linear(4, 128)

        # actor's layer
        self.action_head = nn.Linear(128, 2)

        # critic's layer
        self.value_head = nn.Linear(128, 1)

        # action & reward buffer
        self.saved_actions = []
        self.rewards = []

    def forward(self, x):
        """
        forward of both actor and critic
        """

        x = F.relu(self.affinel(x))

        # actor: chooses action to take from state s_t
        # by returning probability of each action
        action_prob = F.softmax(self.action_head(x), dim=-1)

        # critic: evaluates being in the state s_t
        state_values = self.value_head(x)

        # return values for both actor and critic as a tuple of 2 values:
        # 1. a list with the probability of each action over the action space
        # 2. the value from state s_t
        return action_prob, state_values
```

Initialise The Model

```
model = Policy()
optimizer = optim.Adam(model.parameters(), lr=3e-2)
eps = np.finfo(np.float32).eps.item()
```



Coding Materials

```
def select_action(state):
    state = torch.from_numpy(state).float()
    probs, state_value = model(state)

    # create a categorical distribution over the list of probabilities of actions
    m = Categorical(probs)

    # and sample an action using the distribution
    action = m.sample()

    # save to action buffer
    model.saved_actions.append(SavedAction(m.log_prob(action), state_value))

    # the action to take (left or right)
    return action.item()
```



Coding Materials

```
def finish_episode():
    """
    Training code. Calculates actor and critic loss and performs backprop.
    """

    R = 0
    saved_actions = model.saved_actions
    policy_losses = [] # list to save actor (policy) loss
    value_losses = [] # list to save critic (value) loss
    returns = [] # list to save the true values

    # calculate the true value using rewards returned from the environment
    for r in model.rewards[::-1]:
        # calculate the discounted value
        R = r + args.gamma * R
        returns.insert(0, R)

    returns = torch.tensor(returns)
    returns = (returns - returns.mean()) / (returns.std() + eps)

    for (log_prob, value), R in zip(saved_actions, returns):
        advantage = R - value.item()

        # calculate actor (policy) loss
        policy_losses.append(-log_prob * advantage)

        # calculate critic (value) loss using L1 smooth loss
        value_losses.append(F.smooth_l1_loss(value, torch.tensor([R])))

    # reset gradients
    optimizer.zero_grad()

    # sum up all the values of policy_losses and value_losses
    loss = torch.stack(policy_losses).sum() + torch.stack(value_losses).sum()

    # perform backprop
    loss.backward()
    optimizer.step()

    # reset rewards and action buffer
    del model.rewards[:]
    del model.saved_actions[:]
```



Coding Materials

```
def main():
    running_reward = 10

    # run infinitely many episodes
    for i_episode in count(1):

        # reset environment and episode reward
        state, _ = env.reset()
        ep_reward = 0

        # for each episode, only run 9999 steps so that we don't
        # infinite loop while learning
        for t in range(1, 10000):

            # select action from policy
            action = select_action(state)

            # take the action
            state, reward, done, _, _ = env.step(action)

            if args.render:
                env.render()

            model.rewards.append(reward)
            ep_reward += reward
            if done:
                break

            # update cumulative reward
            running_reward = 0.05 * ep_reward + (1 - 0.05) * running_reward

            # perform backprop
            finish_episode()

            # log results
            if i_episode % args.log_interval == 0:
                print('Episode {}\tLast reward: {:.2f}\tAverage reward: {:.2f}'.format(
                    i_episode, ep_reward, running_reward))

            # check if we have "solved" the cart pole problem
            if running_reward > env.spec.reward_threshold:
                print("Solved! Running reward is now {} and "
                      "the last episode runs to {} time steps!".format(running_reward, t))
                break

    if __name__ == '__main__':
        main()
```

CS115

POLICY GRADIENT
FOR REINFORCEMENT LEARNING

Thanks For Following!

University Of Information Technology