Reinforcement Learning

Deep Reinforcement Learning

Learning Objectives

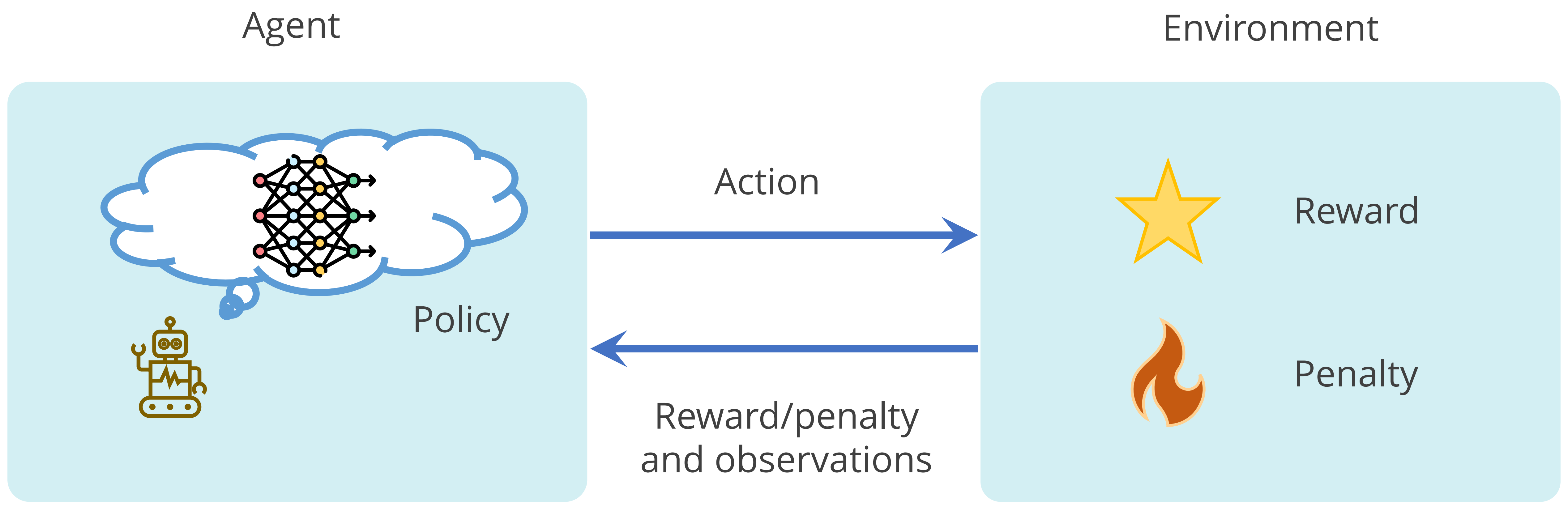
By the end of this lesson, you will be able to:

- Describe neural network policy
- Elaborate the policy gradient and REINFORCE algorithms
- Implement Markov chain model and Markov decision process for reinforcement learning
- Discuss Bellman optimality equation

Neural Network Policy

Deep Reinforcement Learning

Deep reinforcement learning combines deep learning with reinforcement learning principles to create efficient algorithms.



Why Neural Networks in RL?

In many real-world problems, the state and/or action spaces are so large or continuous that it becomes impractical to represent the value function or policy function explicitly.

Function approximation

Deep neural networks, are powerful function approximators that can compactly represent these functions.

Why Neural Networks in RL?

It is challenging to achieve generalization in dynamic environments where the agent continually encounters new situations.

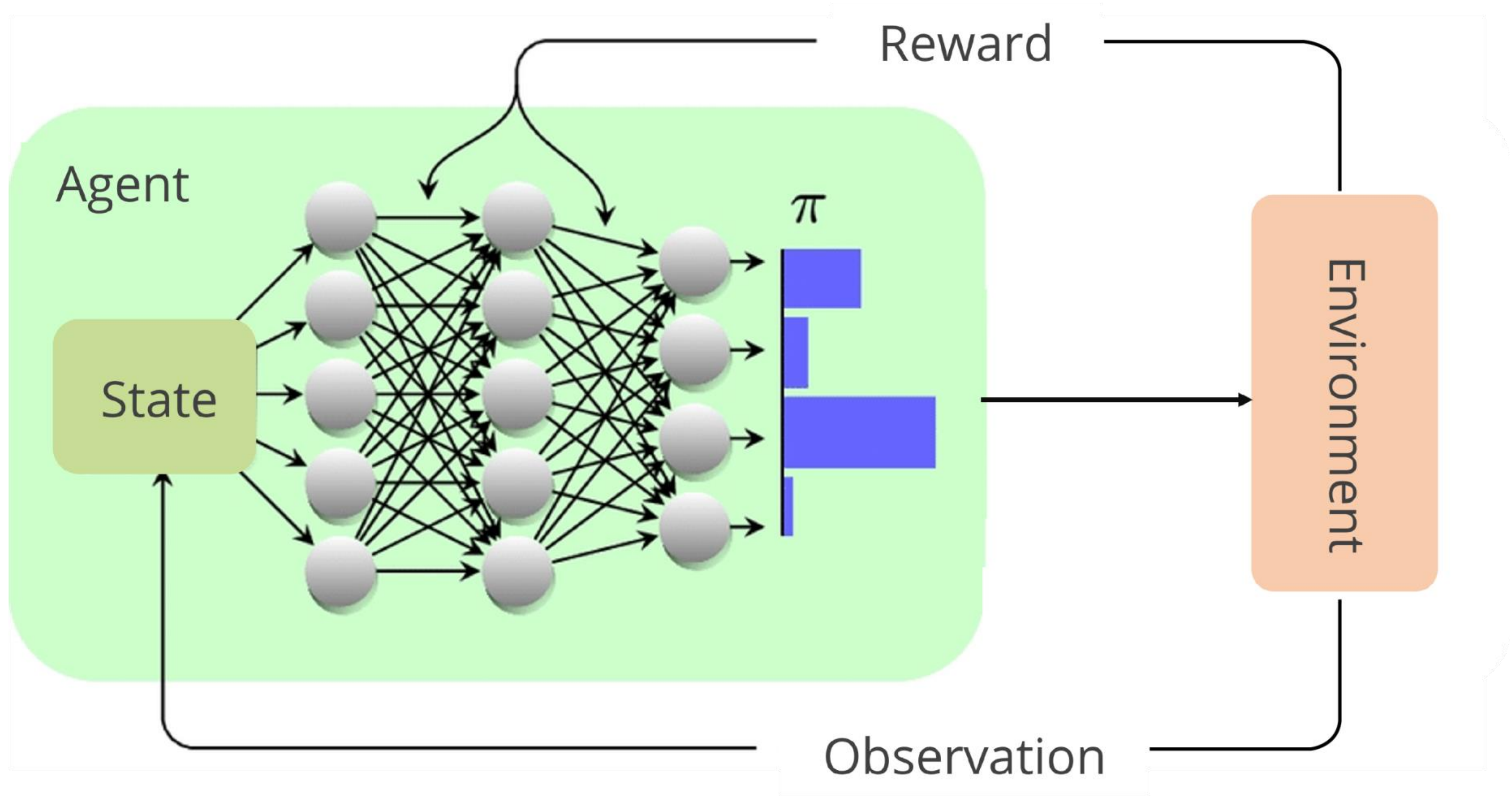
Generalization

Neural networks can generalize from seen to unseen states, making them very effective in dynamic environments.

Neural Network Policy

Instead of hardcoding, a deep neural network can be employed as a policy.

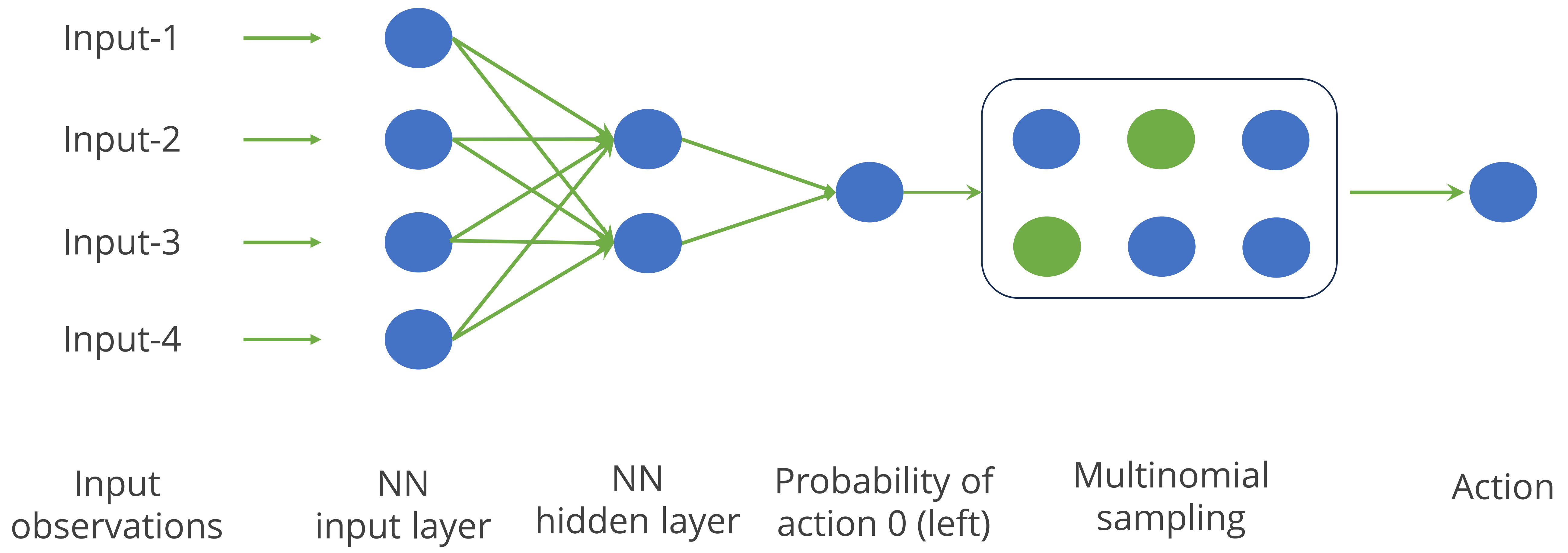
This neural network will use an observation as input and provide the actions to take as output.



Neural Network Policy in Cart-Pole Environment

Precisely, in the cart-pole environment – it will output the probability of one action (left) as ***p***, and the probability of other action (right) will obviously be ***1 – p***.

If the probability of action left is 70% from NN then probability of right will be 30%.



Neural Network Policy – Creating Model

Creating this neural network policy using keras:

```
# Creating this neural network policy using
keras:

import tensorflow as tf
from tensorflow import keras

n_inputs = 4

model = keras.models. Sequential([
     keras.layers.Dense(4, activation="relu",
input_shape=[n_inputs]),
      keras.layers.Dense(2, activation="relu"),
      keras.layers. Dense(1,
activation="sigmoid"),])
```

- Define the number of inputs.
- Cart-pole problem has total 4 observations hence number of input are 4.

Neural Network Policy – Creating Model

Creating this neural network policy using keras:

```
# Creating this neural network policy using
keras:

import tensorflow as tf
from tensorflow import keras

n_inputs = 4

model = keras.models. Sequential([
      keras.layers.Dense(4, activation="relu",
input_shape=[n_inputs]),
      keras.layers.Dense(2, activation="relu"),
      keras.layers. Dense(1,
activation="sigmoid"),])
```

Use simple sequential model from keras to create a neural network.

Neural Network Policy – Creating Model

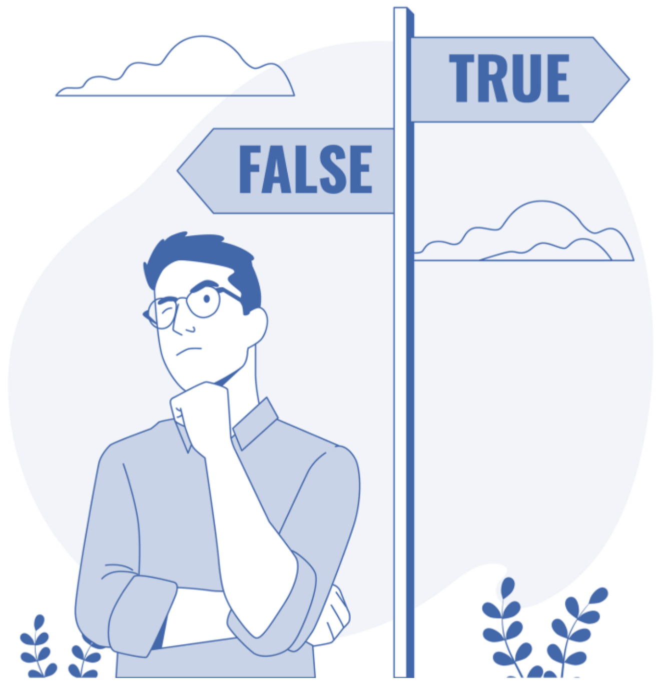
Creating this neural network policy using keras:

```
# Creating this neural network policy using
keras:

import tensorflow as tf
from tensorflow import keras

n_inputs = 4

model = keras.models. Sequential([
      keras.layers.Dense(4, activation="relu",
input_shape=[n_inputs]),
      keras.layers.Dense(2, activation="relu"),
      keras.layers. Dense(1,
activation="sigmoid"),])
```

Output layer uses the sigmoid activation function because Cart-Pole problem has only two possible output.

Neural Network Policy Evaluation - Credit Assignment Problem

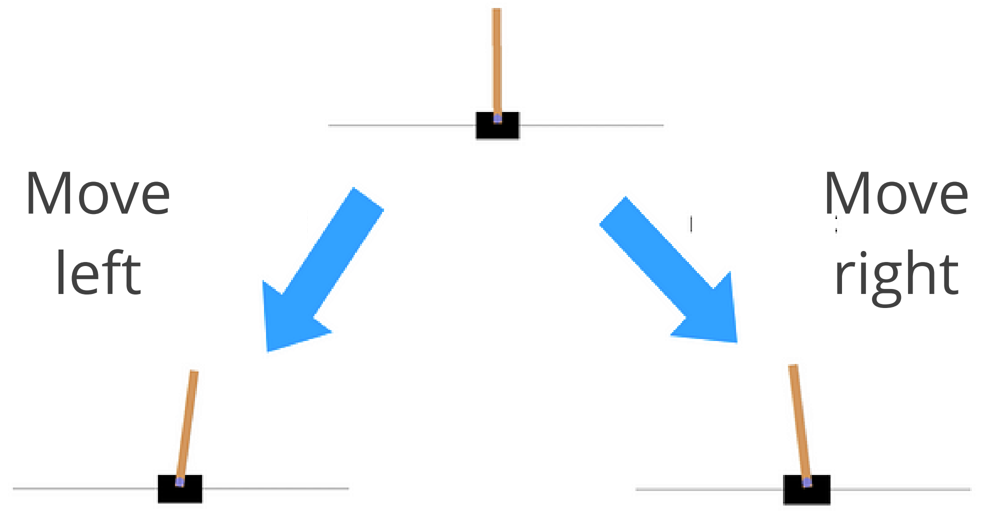
In cart-pole problem, or with reinforcement learning in general, there are no ground truth labels.



- The only guidance that an agent gets is through the rewards.
- And rewards are generally sparse and delayed.

Credit Assignment Problem

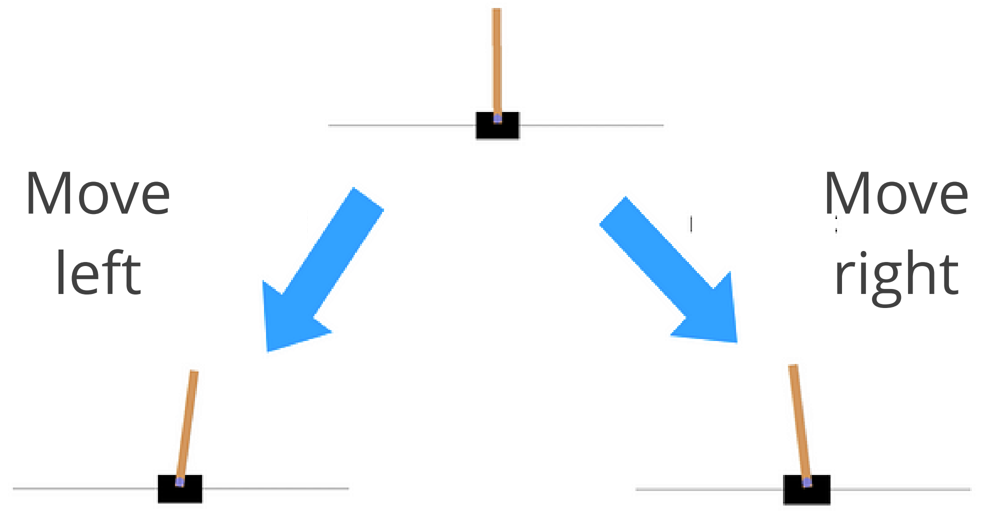
Consider the scenario where the pole is balanced by the agent for 100 steps.



How can it be determined which actions among the 100 were beneficial and which were detrimental?

Credit Assignment Problem

Consider the scenario where the pole is balanced by the agent for 100 steps.

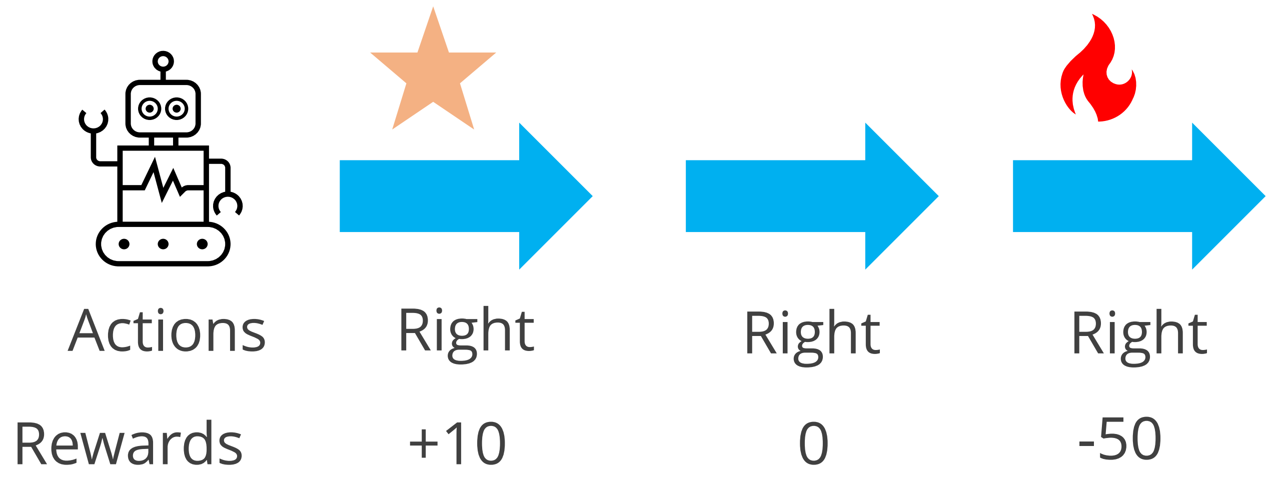


- The agent's only knowledge is that the pole fell after the last (100th) step.
- But it is evident that this final step alone is not solely responsible.

This situation is recognized as the **credit assignment problem.**

Solution to Credit Assignment Problem

The general strategy to tackle the credit assignment problem is to assess an action based on the total rewards received after it by applying a discount factor (γ) at each step.

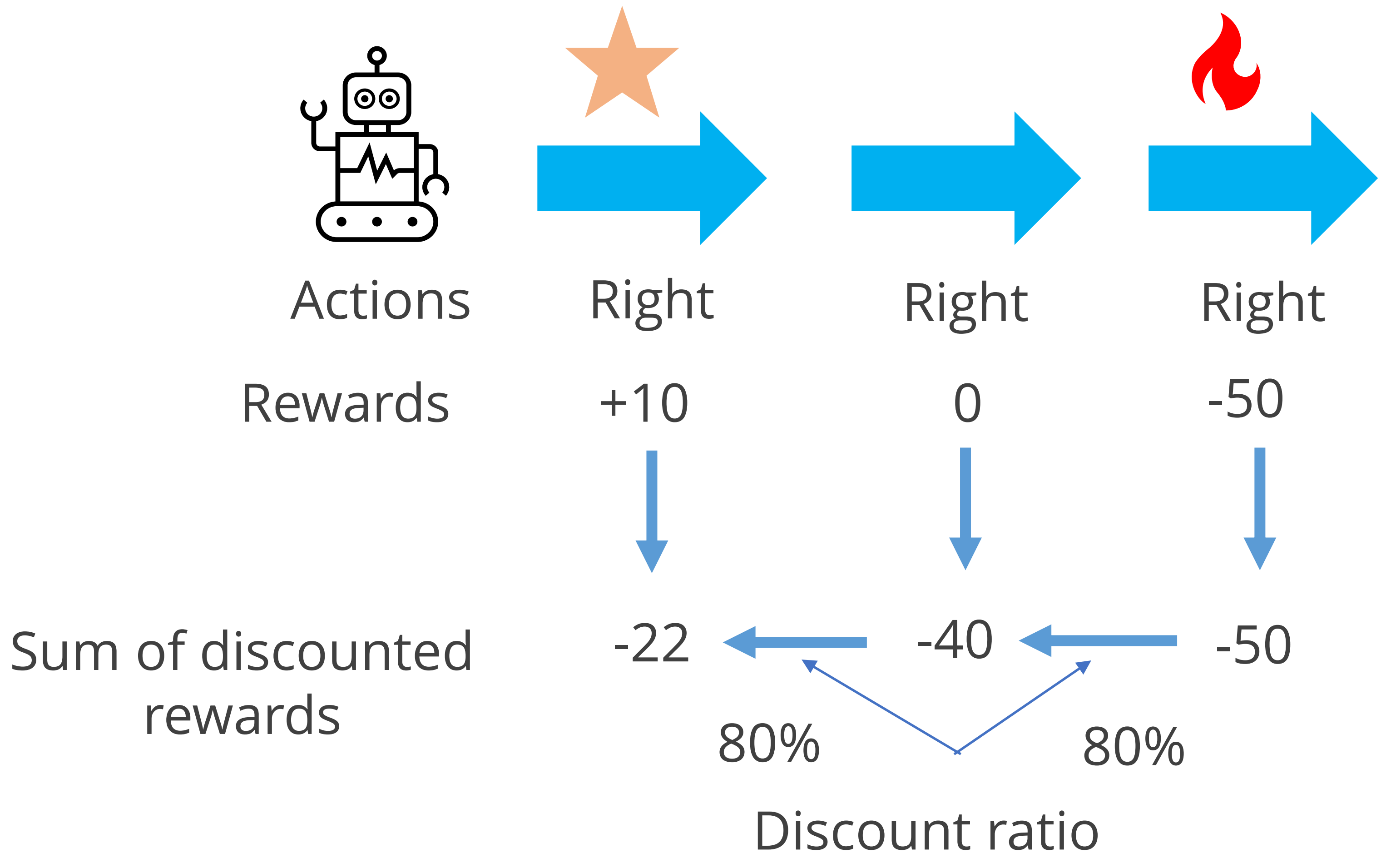


In this image, an agent decides to go right for three steps in a row and receives a reward of +10 after first step, no reward in second step and a penalty of -50 in the last step.

Solution to Credit Assignment Problem

Consider the discount factor, γ of 0.8, the return of the first action can be calculated as:

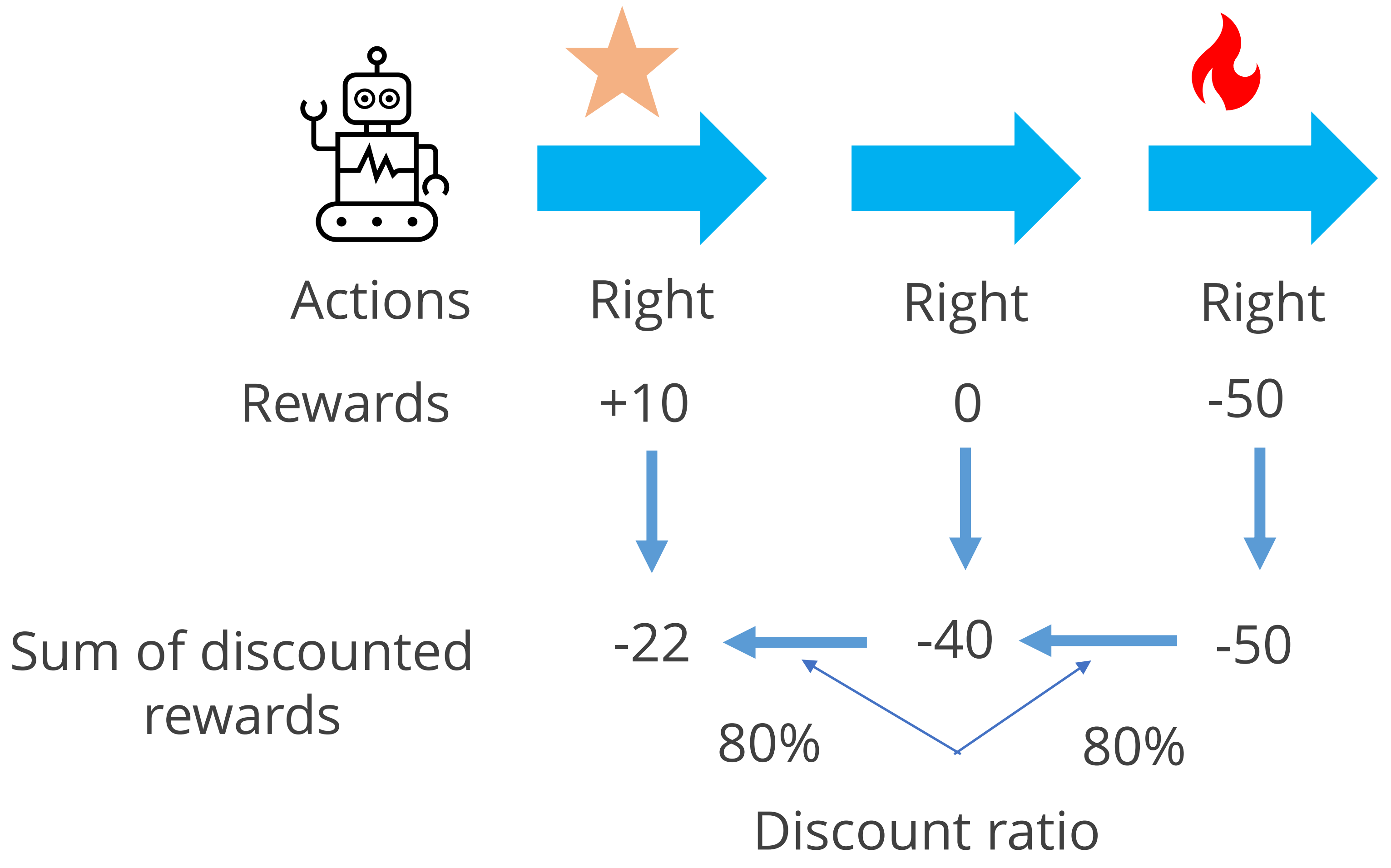
$$10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$$



Solution to Credit Assignment Problem

If γ is near zero, future rewards won't weigh heavily against immediate rewards.

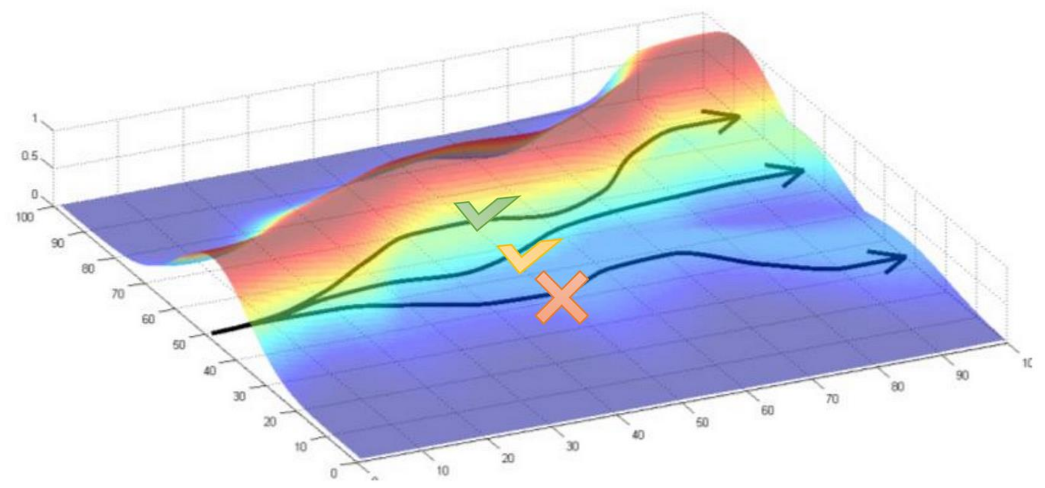
Alternatively, if γ is near 1, future rewards will be as significant as immediate rewards.



Policy Gradient

Policy Gradients

The neural network, serving as a policy, assists in determining the next action; however, to achieve the optimal output, its parameters must be optimized through training.



Policy gradient methods enhance policy parameters by navigating the gradients towards higher rewards.

A renowned example of this approach is the **REINFORCE** algorithm, established in 1992, which has inspired various adaptations over time.

REINFORCE Algorithm

REINFORCE algorithm consists the following:

Game iterations and gradient calculation

- Multiple game iterations are undergone by the neural network.
- At each step, gradients are calculated to enhance the probability of the selected action.
- These gradients, once computed, are not immediately applied.

REINFORCE Algorithm

REINFORCE algorithm consists the following:

Advantage calculation

- The advantage of each action is computed after several episodes.
- Based on previous discussions, this metric determines the relative quality of actions.

REINFORCE Algorithm

REINFORCE algorithm consists the following:

Gradient application based on advantage	
	• If an action's advantage is positive (indicating a beneficial action), gradients are applied to encourage future selection of this action. • Conversely, for negative advantages indicating less favourable actions, opposite gradients are applied to discourage these actions. • This is achieved by simply multiplying each gradient vector by the corresponding action advantage.

REINFORCE Algorithm

REINFORCE algorithm consists the following:

Gradient application based on advantage

- Finally, the mean of all resulting gradient vectors is computed.
- It is then used to perform a gradient descent step, adjusting the neural network parameters for improvement.

Policy Gradients - Implementation

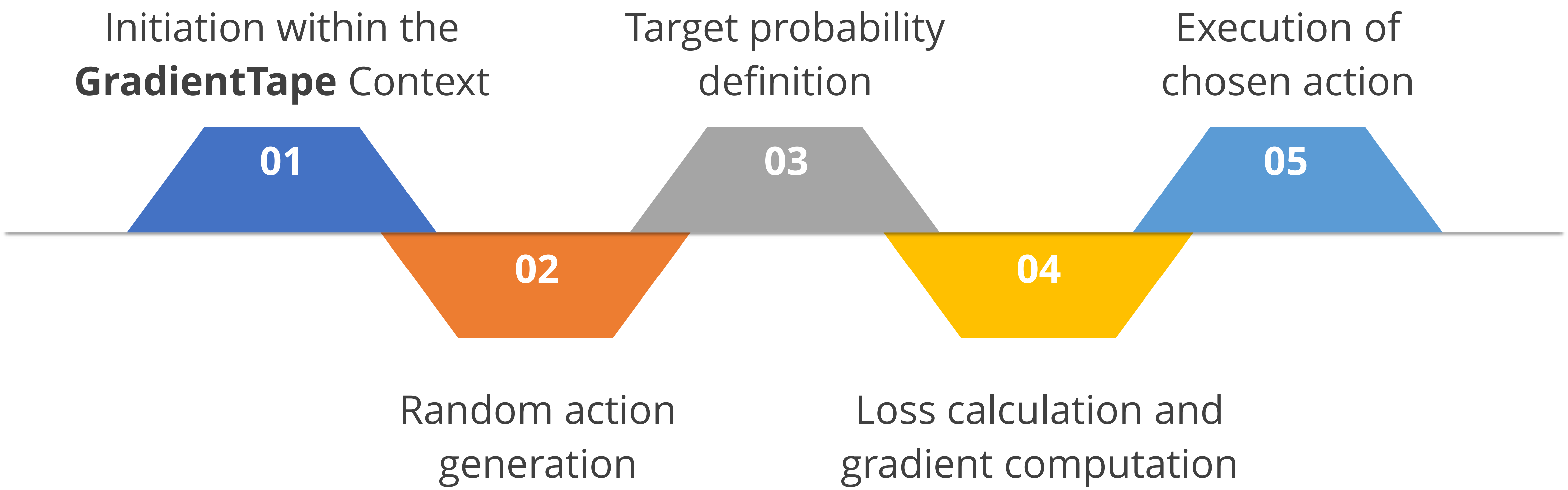
Train the neural network policy we built earlier so that it learns to balance the pole on the cart.

The function defined below will execute one step.

```
def execute_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model (obs [np.newaxis])
        action = (tf.random.uniform ([0, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean (loss_fn (y_target, left_proba))
    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, info = env.step(int(action [0, 0].numpy()))
    return obs, reward, done, grads
```

Policy Gradients - Implementation

The key steps the function performs are as below:


Initiation within the **GradientTape** Context
01
02
Random action generation
Target probability definition
03
04
Loss calculation and gradient computation
Execution of chosen action
05

Policy Gradients - Implementation

The function defined in the first step only executes the one-step iteration!

The function below, however, executes multiple episodes and returns all the rewards and gradients for each episode and each step.

```
def execute_multiple_episodes (env, n_episodes, n_max_steps, model,
loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
       current_rewards = []
       current_grads = []
       obs env.reset()
       for step in range(n_max_steps):
          obs, reward, done, grads = execute_one_step(env, obs, model,
loss_fn)
          current_rewards.append(reward)
          current_grads.append(grads)
          if done:
             break
       all_rewards.append(current_rewards)
       all_grads.append(current_grads)
    return all_rewards, all_grads
```

Policy Gradients - Implementation

The function defined previously produces an array of reward array.

- Each inner array corresponding to the rewards collected at each step of an episode.
- Additionally, it also generates an array of arrays of gradient tuples.
- In this array, each outer array pertains to an episode, and each inner tuple contains the gradient tensors for every trainable variable at each step.

Policy Gradients - Implementation

The algorithm will use the *execute_multiple_episodes()* function to run the game several times ,then it will go back and look at all the rewards, discount them, and normalize them.

Two additional functions are necessary to accomplish that.

discount_rewards()

discount_and_
normalize_rewards()

Policy Gradients - Implementation

The *discount_rewards()* function computes the sum of future discounted rewards at each step.

The function is defined as below:

```
def discount_rewards (rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted [step] += discounted [step + 1] *
discount_factor
    return discounted
```

Policy Gradients - Implementation

The *discount_and_normalize_rewards* function normalizes all these discounted rewards (returns) across many episodes.

```
def discount_and_normalize_rewards (all_rewards,
discount_factor):
    all_discounted_rewards =
[discount_rewards(rewards,discount_factor)
                          for rewards in all_rewards]
    flat_rewards = np.concatenate (all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
           for discounted_rewards in all_discounted_rewards]
```

Policy Gradients - Implementation

Define hyperparameters, optimizer, loss function, and training loop.

Hyperparameters

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

- The model will undergo 150 training iterations.
- Each iteration comprises of 10 episodes.
- Each episode capped at a maximum of 200 steps.
- A discount factor of 0.95 will be applied for calculating the future reward.

Policy Gradients - Implementation

Define hyperparameters, optimizer, loss function, and training loop.

Optimizer and loss function

```
optimizer = keras.optimizers.Adam(lr = 0.01)
loss_fn = keras.losses.binary_crossentropy
```

- The training will utilize the Adam optimizer with a learning rate of 0.01.
- The binary cross-entropy loss function will be used.
- This will be used to fit the binary decision-making scenario i.e.:
    - choosing to go left or right.

Policy Gradients - Implementation

Define hyperparameters, optimizer, loss function, and training loop.

Training iteration

```
for iteration in range(n_iterations):
    all_rewards, all_grads = execute_multiple_episodes (
          env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards (all_rewards,
                                                        discount_factor)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean (
             [final_reward * all_grads [episode_index] [step] [var_index]
             for episode_index, final_rewards in enumerate (all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)
    optimizer.apply_gradients (zip(all_mean_grads, model.trainable_variables))
```

Policy Gradients - Implementation

Define hyperparameters, optimizer, loss function, and training loop.

Training iteration

- At every training iteration, the *execute_multiple_episodes()* function is invoked.
- This function runs the game for 10 episodes, returning all rewards and gradients for each episode and step.

Policy Gradients - Implementation

Define hyperparameters, optimizer, loss function, and training loop.

Advantage calculation

- The discount_and_normalize_rewards() function is subsequently called to calculate the normalized advantage for each action (referred to as final reward in the code).
- This calculation indicates the relative quality of each action taken.

Policy Gradients - Implementation

Define hyperparameters, optimizer, loss function, and training loop.

Gradients weighted average calculation

- The process iterates over each trainable variable.
- It computes the weighted average of the gradients for that variable across all episodes and steps, with weighting determined by the final_reward().

Policy Gradients - Implementation

Define hyperparameters, optimizer, loss function, and training loop.

Gradient application

- Lastly, these average gradients are applied through the optimizer to adjust the model's trainable variables.
- This aims to incrementally improve the policy.

Policy Gradients - Implementation

The code defined previously will train the neural network policy, and it will successfully learn to balance the pole on the cart.

The mean reward per episode will get very close to 200 (which is the maximum by default with this environment).

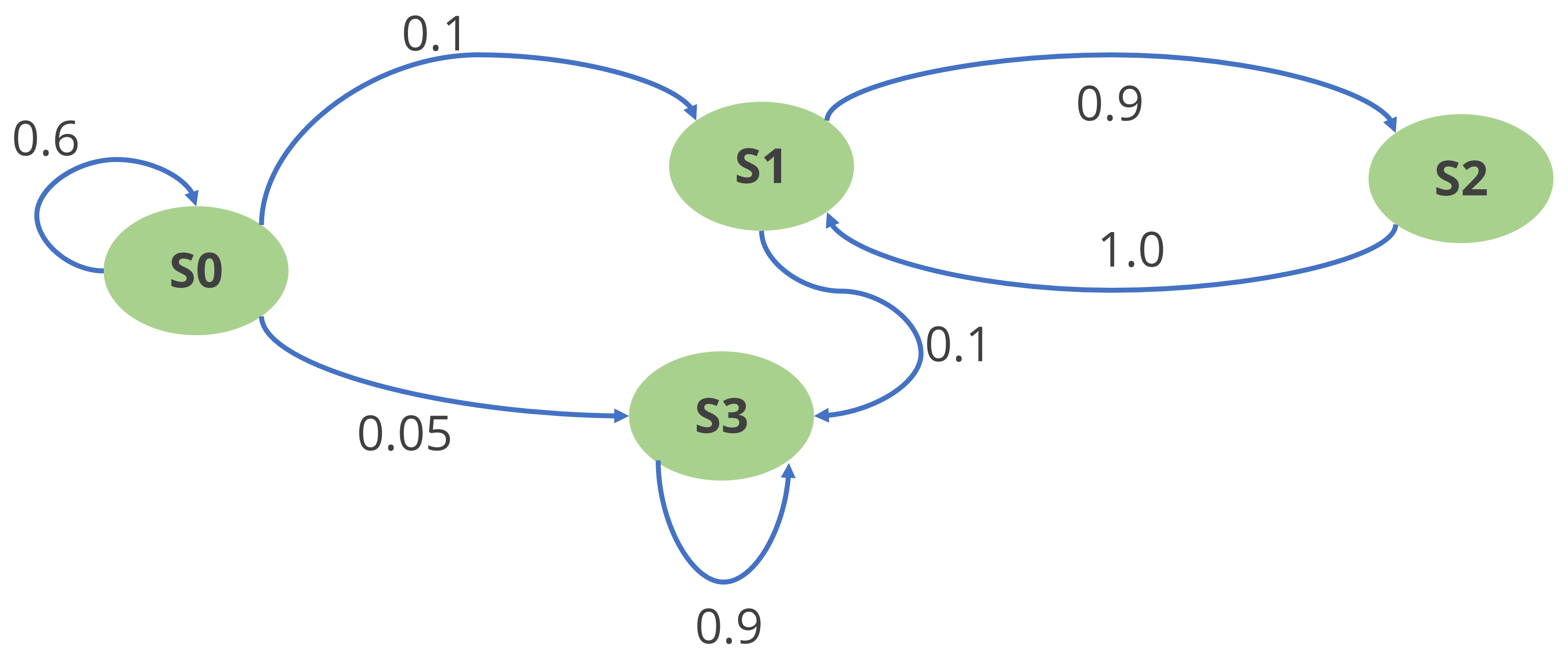
The neural network policy has effectively resolved the cart-pole problem, outperforming the hardcoded policy search discussed in previous sections.

Markov Decision Process

Markov Chain

A Markov chain is a stochastic model describing a sequence of possible events where the probability of each event depends only on the state attained in the previous event and not on the past states/events.

This is why it’s called memoryless process



Example of Markov chain with 4 states

Key Features of Markov Chains

The key features of Markov's Chain model include:

Memoryless property

- The core property of a Markov Chain is its lack of memory, formally known as the Markov Property.
- It implies that the future state depends only on the current state and not on the sequence of events that preceded it.

Key Features of Markov Chains

The key features of Markov's Chain model include:

Transition matrix

- In a Markov Chain, the probabilities of moving from one state to another are represented in a matrix known as the transition matrix.
- Each entry in the matrix *T(s, s')* represents the probability of transitioning from state ***i*** to state ***j***.

Key Features of Markov Chains

The key features of Markov's Chain model include:

States

- Markov Chains have a fixed number of states.
- These states can be anything from physical locations, a set of measurements, or even abstract conditions.

Markov Chains: Example

Consider a simple board game like "snakes and ladders”.

In this game, each roll of the dice determines your next position based solely on your current position, not how you arrived there.

Markov Decision Process

Markov decision process

A Markov decision process (MDP) is a mathematical framework used in reinforcement learning to describe an environment in terms of states, actions, rewards, and transitions.

Its like Markov chain but include a slight change:

- At each step, an agent can choose one of the several possible actions.
- And transition probabilities depends on the chosen action.

Some state transitions return rewards (positive or negative), and agent’s goal is finding a policy that will maximize reward over time.

Components of an MDP

The key features of Markov decision process include:

State (S)

A finite set of states of the environment. At any given time, the environment is in one state.

Actions (A)

A finite set of actions available to the agent. In response to an action, the environment transitions from one state to another.

Components of an MDP

The key features of Markov decision process include:

Transition probability

- A probability distribution that defines the likelihood of transitioning from one state to another given a particular action.
- T(s, a, s′) represents the probability of transitioning to state **s′** from state **s** after taking action **a**.

Components of an MDP

The key features of Markov decision process include:

Rewards (R)

- A reward function that gives the immediate reward received after transitioning from one state to another state via an action.
- R (s, a, s') is the reward received after moving to state s’ from state s by taking action a.

Components of an MDP

The key features of Markov decision process include:

Rewards (R)

- A factor between 0 and 1 that represents the present value of future rewards.
- This indicates how much the agent values immediate rewards over future rewards.

How MDP Works in Reinforcement Learning?

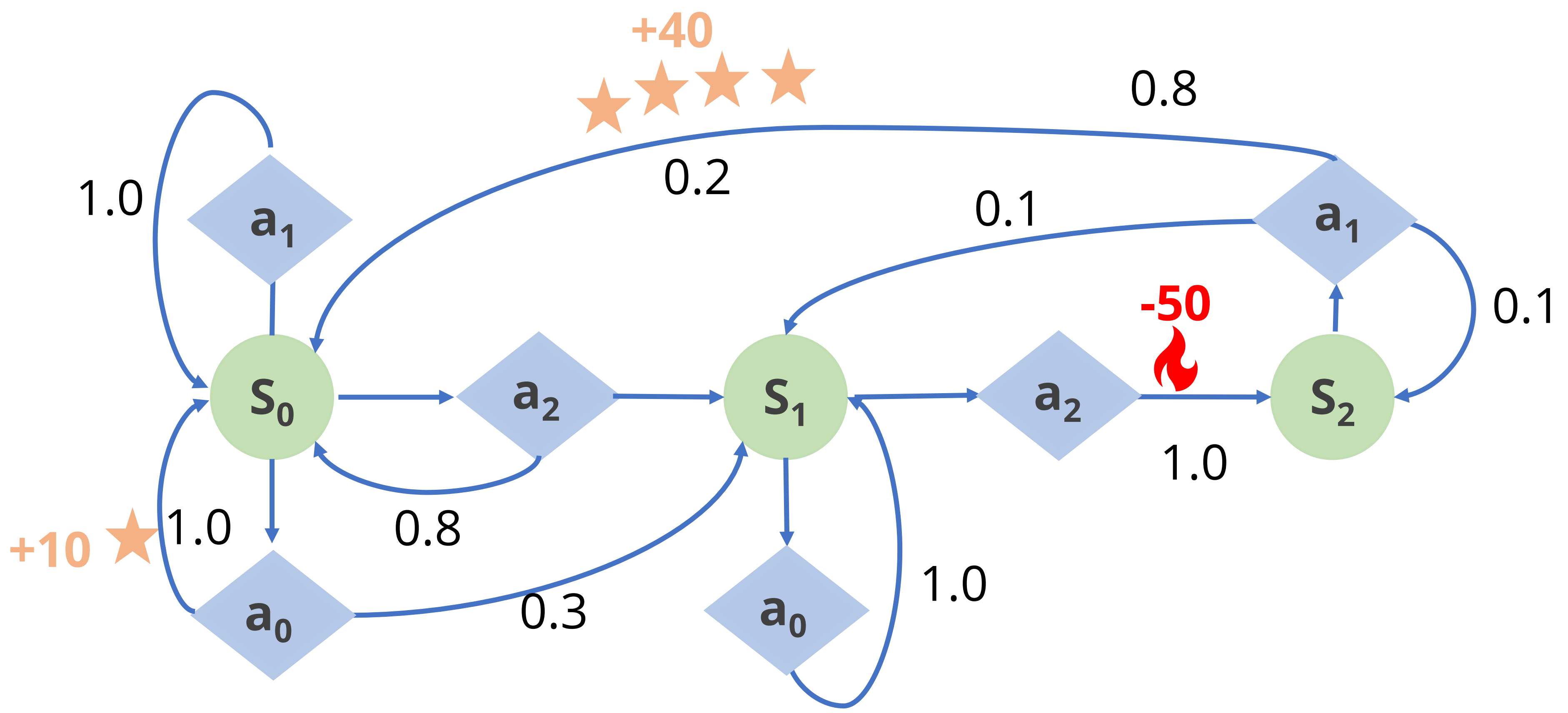
In reinforcement learning, an agent interacts with an environment in a sequence of steps.

At each step:

- The agent observes the current state ‘s’.
- Based on the state, the agent selects an action ‘a’ from a set of possible actions.
- The environment responds by presenting a new state s' and giving a reward R.
- The goal of the agent is to learn a policy (a strategy) that dictates the best action to take in each state to maximize cumulative rewards over time.

Markov Decision Process

Observing the MDP below, can you deduce which strategy will accumulate the highest rewards over time?

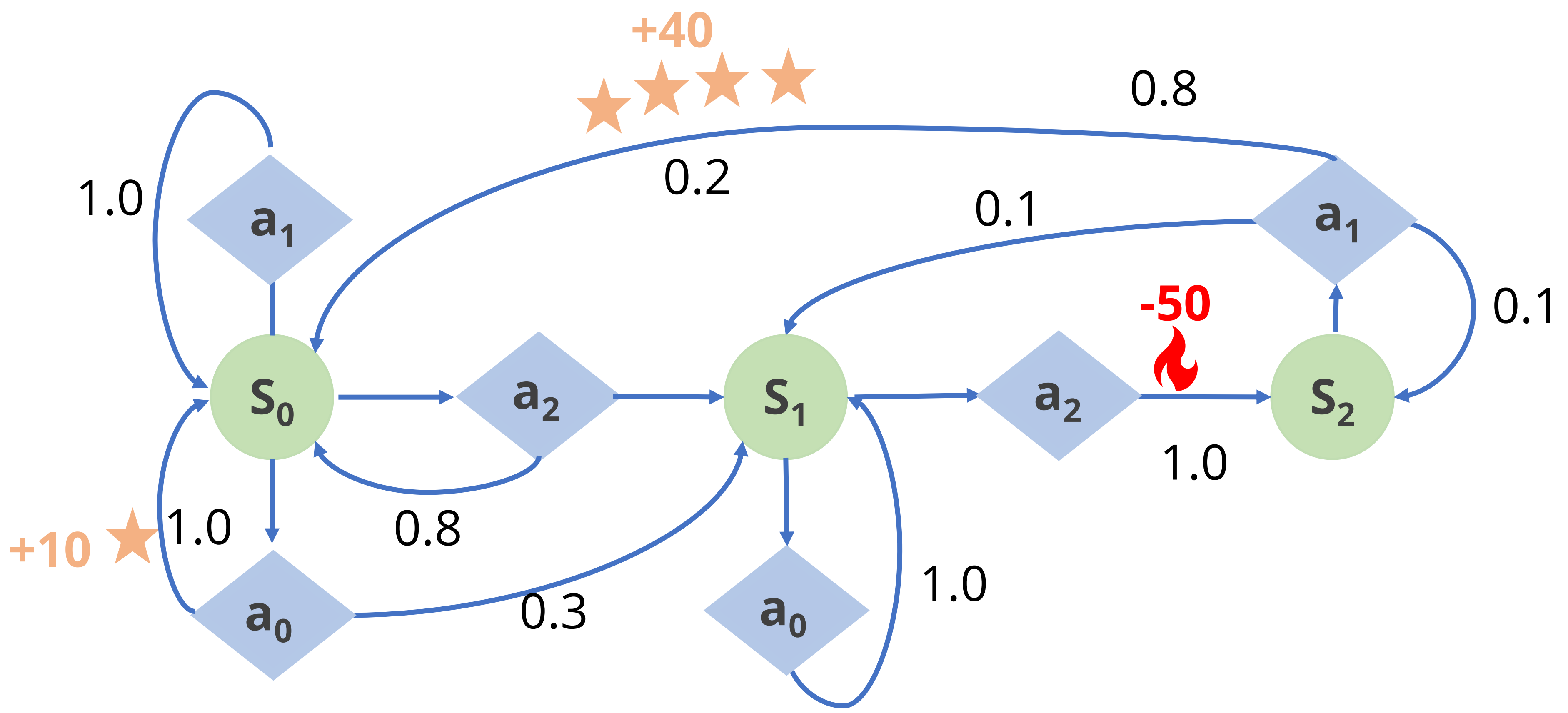


Example of a Markov Decision Process

- It's evident in state **S_0** that action **a_0** is the preferable choice.
- While in state ***S_2***, the agent is constrained to select action **a_1**.

Markov Decision Process

Observing the MDP below, can you deduce which strategy will accumulate the highest rewards over time?



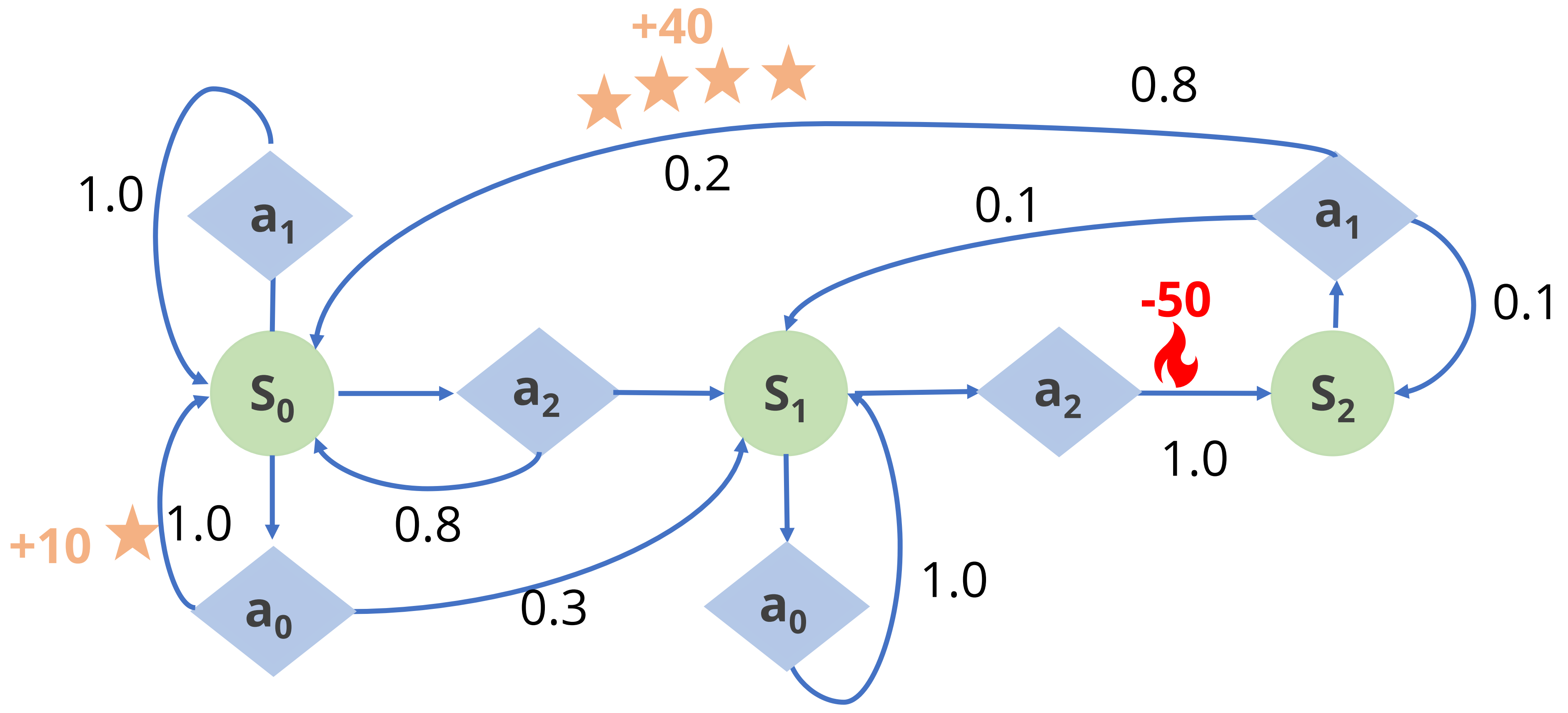
Example of a Markov Decision Process

- However, the decision becomes less clear in state $\mathbf{S}_1$.
- It's uncertain here, whether the agent should:
  - Remain in place ($\mathbf{a}_0$), or
  - Proceed through the hazardous path ($\mathbf{a}_2$).

Bellman Optimality Equation

Optimal State Value

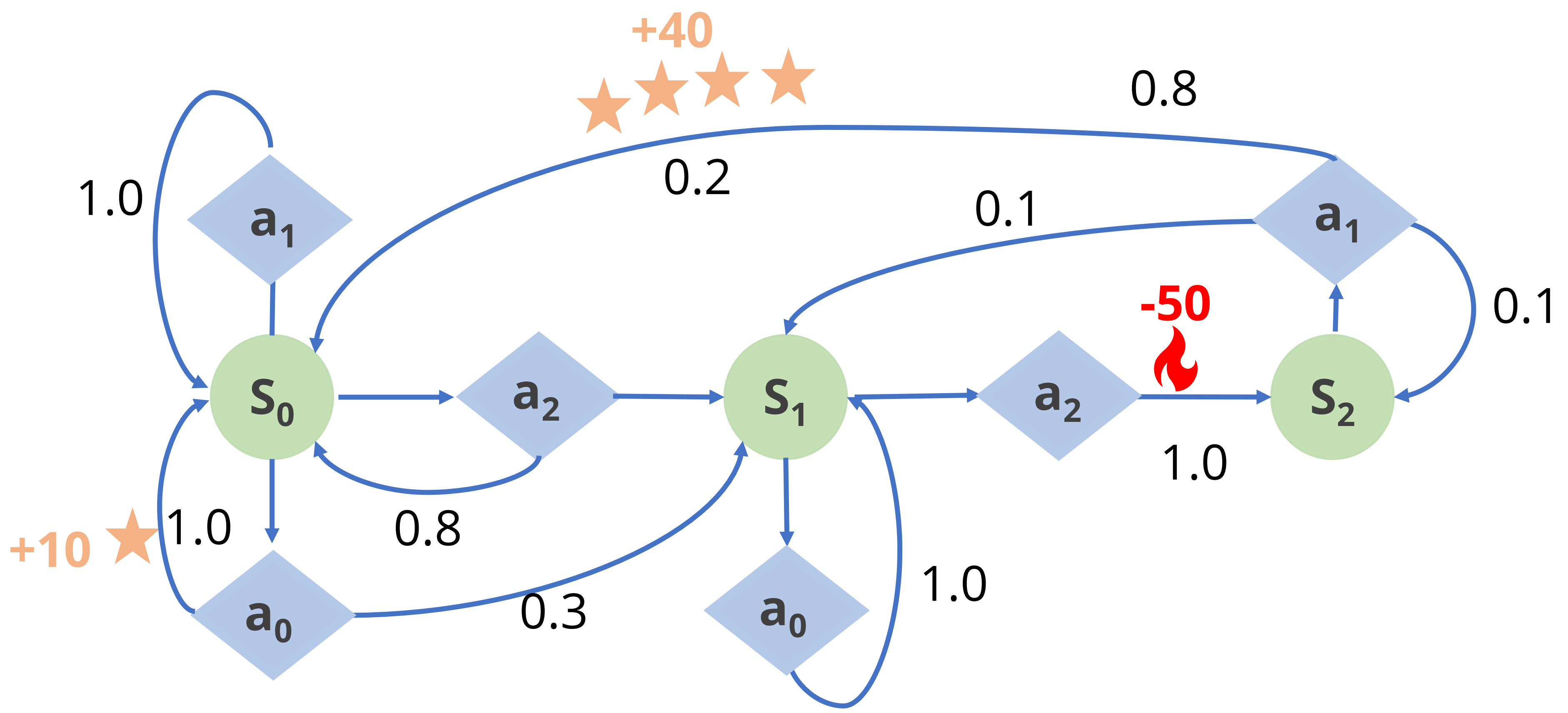
Richards Bellman figured out a way to estimate the **optimal state value** of any state.



Bellman's work includes the concept of the optimal state value **V*(s)**.

Optimal State Value

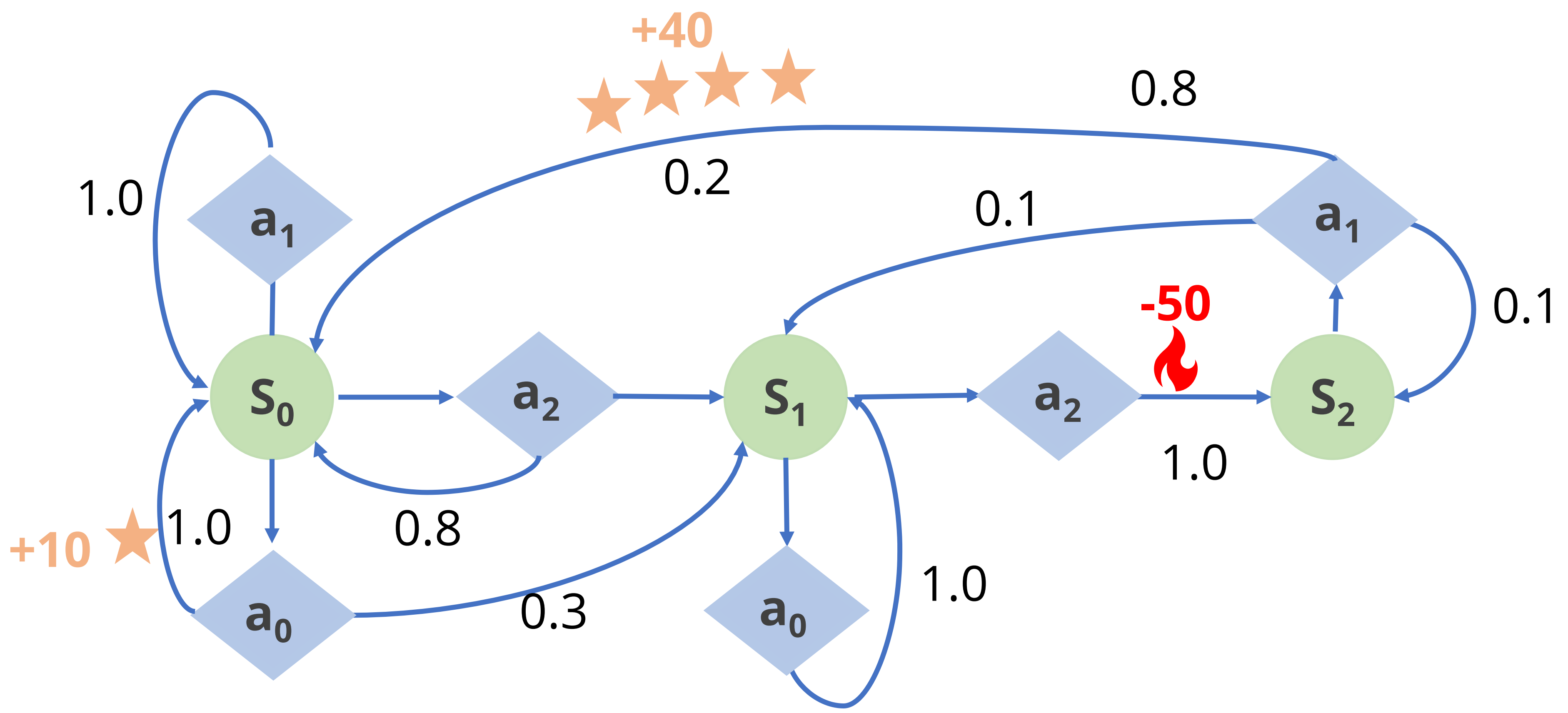
Richards Bellman figured out a way to estimate the **optimal state value** of any state.



This is sum of all discounted future rewards that agent can expect on average after it reaches a state ***s***, if it acts optimally.

Optimal State Value

Richards Bellman figured out a way to estimate the **optimal state value** of any state.



Its maximum expected return (or total reward) that can be obtained starting from a given state and following the best policy thereafter.

Bellman Optimality Equation

The Bellman optimality equation serves as a foundation for an algorithm capable of accurately determining the optimal state value for every possible state.

For all states (s)

$$V^*(s) = max_a \sum_s T(s, a, s') \left[R(s, a, s') + \gamma . V^*(s')\right]$$

In this equation:

- **T(s, a, s')** is the transition probability from state **s** to state **s',** given that the agent chose action ***a***.
- For example, in the MDP sample in last section , T(s2, a1, s0) = 0.8.

Bellman Optimality Equation

The Bellman optimality equation serves as a foundation for an algorithm capable of accurately determining the optimal state value for every possible state.

For all states (s)

$$V^*(s) = max_a \sum_s T(s,a,s')\,[R(s,a,s') + \gamma . V^*(s')]$$

In this equation:

- **R(s, a, s')** is the reward that the agent gets when it goes from state **s** to state **s'**, given that the agent chose action ***a*.**
- For example, in the MDP sample in last section, R(s2, a1, s0) = +40.

Bellman Optimality Equation

The Bellman optimality equation serves as a foundation for an algorithm capable of accurately determining the optimal state value for every possible state.

For all states (s)

$$V^*(s) = max_a \sum_s T(s,a,s') \, [R(s,a,s') + \gamma . V^*(s')]$$

In this equation:

- **γ** is the discount factor.

Value Iteration Algorithm

Value iteration is an iterative process that uses the Bellman optimality equation to update state values to converge to the optimal values.

Initialization

- Initialize the value of each state arbitrarily or set them to some initial values: ***V*(*s*)**.

Value Iteration Algorithm

Value iteration is an iterative process that uses the Bellman optimality equation to update state values to converge to the optimal values.

Iteration

- Repeat until convergence (or for a fixed number of iterations) utilizing the Bellman ’s equation.
- The update involves selecting the action that maximizes the expected cumulative reward.

With sufficient iterations, it's assured that these estimates will align with the true optimal state values, reflecting the strategy of the optimal policy.

Value Iteration Algorithm

Value Iteration is an iterative process that uses the Bellman optimality equation to update state values to converge to the optimal values.

The update rule for the value iteration algorithm is as below:

$$\mathrm{V}_{\mathrm{k+1}}(\mathrm{s}) \leftarrow \max_{\mathrm{a}} \sum_{s'} \mathrm{T}(\mathrm{s},\mathrm{a},\mathrm{s}')[\mathrm{R}(\mathrm{s},\mathrm{a},\mathrm{s}') + \gamma . \mathrm{V}_{\mathrm{k}}(\mathrm{s}')] \ \ for\ all\ s$$

In this algorithm, $\mathrm{V}_{\mathrm{k}}(s)$ is the estimated value of the state s at the k^{th} iteration of the algorithm.

Value Iteration Algorithm

Value iteration is an iterative process that uses the Bellman optimality equation to update state values to converge to the optimal values.

Convergence check

- Check for convergence by comparing the new values with the previous values.
- If the values have converged (or the change is below a certain threshold), stop the iteration.

Value Iteration Algorithm

Value iteration is an iterative process that uses the Bellman optimality equation to update state values to converge to the optimal values.

Output

- The resulting ***V*(*s*)** values after convergence represent the optimal state values.

Value Iteration Algorithm

Value iteration is an iterative process that uses the Bellman optimality equation to update state values to converge to the optimal values.

Policy extraction

- If needed, extract the optimal policy by selecting the action that maximizes the expression inside the argmax of the Bellman equation.

Key Takeaways

- Deep reinforcement learning merges deep learning and reinforcement principles for efficient algorithm creation.
- Policy gradient methods enhance policy parameters by navigating the gradients towards higher rewards.
- Markov chain is memoryless stochastic process with state-dependent transition probabilities.
- MDP is a decision-making model with states, actions, and transition probabilities.

Knowledge Check

Knowledge Check

1

What is the primary purpose of combining deep learning with reinforcement learning in deep reinforcement learning?

A. To improve memory management.

B. To enhance classification tasks.

C. To create efficient algorithms for decision-making.

D. To speed up training times.

Knowledge Check

1

What is the primary purpose of combining deep learning with reinforcement learning in deep reinforcement learning?

A. To improve memory management.

B. To enhance classification tasks.

C. To create efficient algorithms for decision-making.

D. To speed up training times.

The correct answer is **C**

The primary purpose of combining deep learning with reinforcement learning in deep reinforcement learning is to efficiently model complex decision-making by leveraging deep learning's capacity for intricate pattern recognition.

Knowledge Check

2

What is the Markov property in the context of a Markov Chain?

A. Future states depend only on the current state.

B. Future states depend only on the initial state.

C. Future states depend on the entire history of states.

D. Future states depend on external factors, not the current state.

Knowledge Check

2

What is the Markov property in the context of a Markov chain?

A. Future states depend only on the current state.

B. Future states depend only on the initial state.

C. Future states depend on the entire history of states.

D. Future states depend on external factors, not the current state.

The correct answer is **A**

The Markov property states that the future states in a Markov chain depend only on the current state, not on the sequence of events that preceded it. This property is essential for modeling systems with a memoryless transition.

Knowledge Check

3

What does the transition probability in a Markov chain represent?

A. The probability of transitioning from one state to another.

B. The probability of remaining in the same state.

C. The probability of reaching the goal state.

D. The probability of receiving a reward in a state

Knowledge Check

3

What does the transition probability in a Markov chain represent?

A. The probability of transitioning from one state to another.

B. The probability of remaining in the same state.

C. The probability of reaching the goal state.

D. The probability of receiving a reward in a state

The correct answer is **A**

The transition probability in a Markov chain represents the likelihood of moving from one state to another in the next time step. It defines the dynamics of the system.

Thank You