CS 581

Advanced Artificial Intelligence

February 26, 2024

Announcements / Reminders

- **Please follow the Week 07 To Do List instructions (if you haven't already)**

- **Programming Assignment #01: due on Sunday 03/03 at 11:59 PM CST**

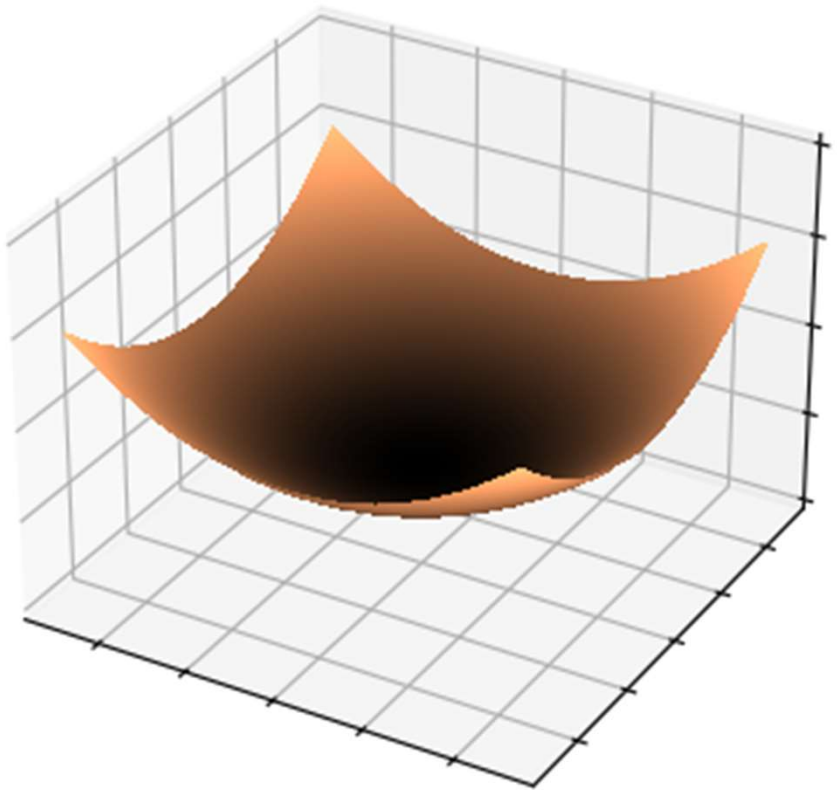
Plan for Today

- Gradient Descent
- Searching in Nondeterministic environments
- Reinforcement Learning

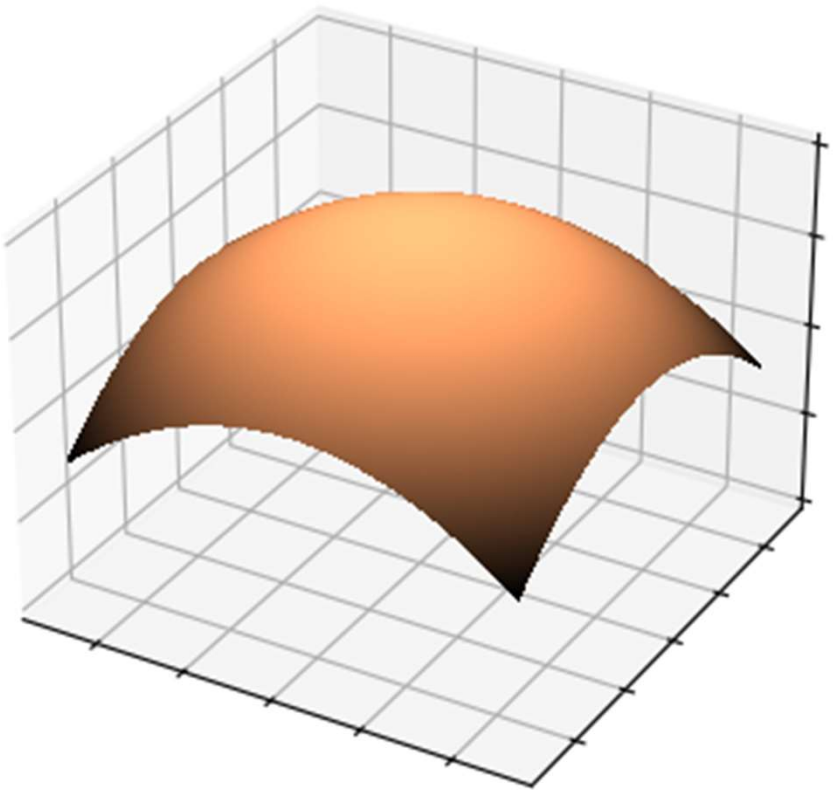
Gradient Descent

Convex vs. Concave Functions

Convex Function

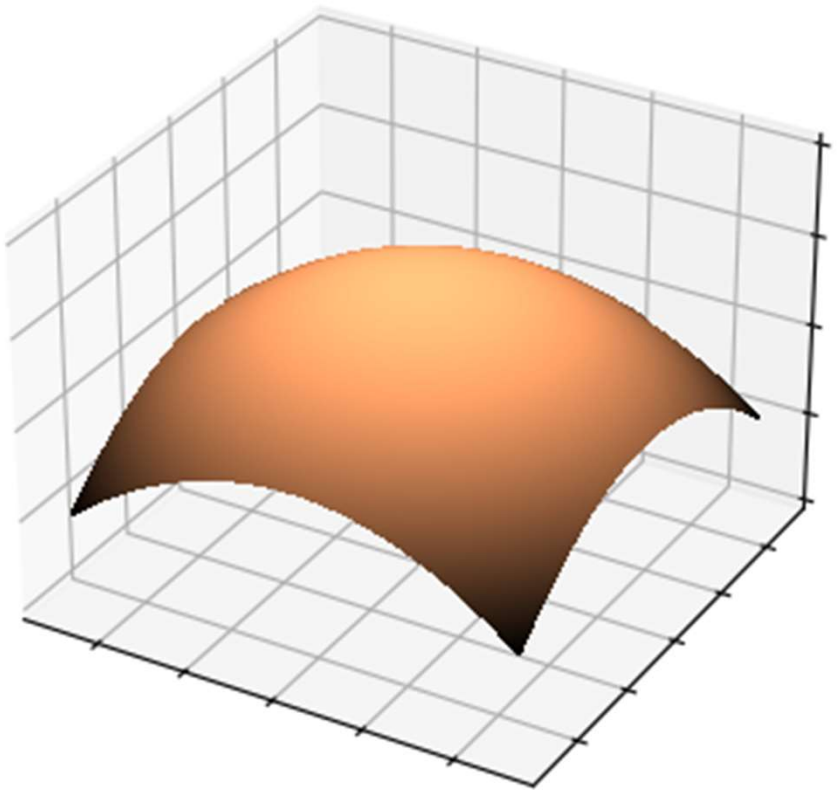


Concave Function

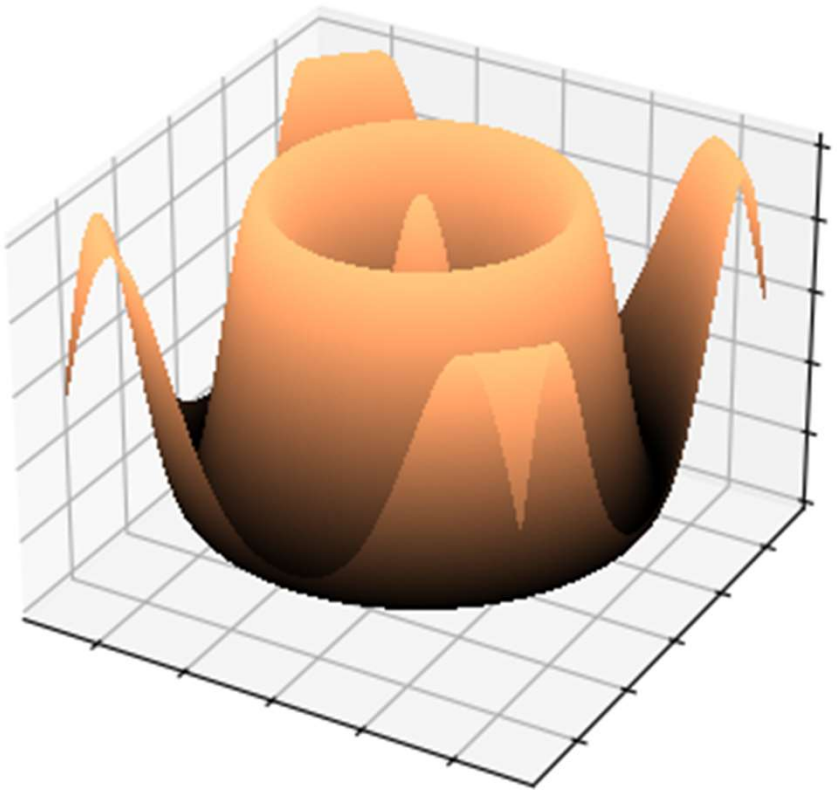


Concave vs. Non-Convex Functions

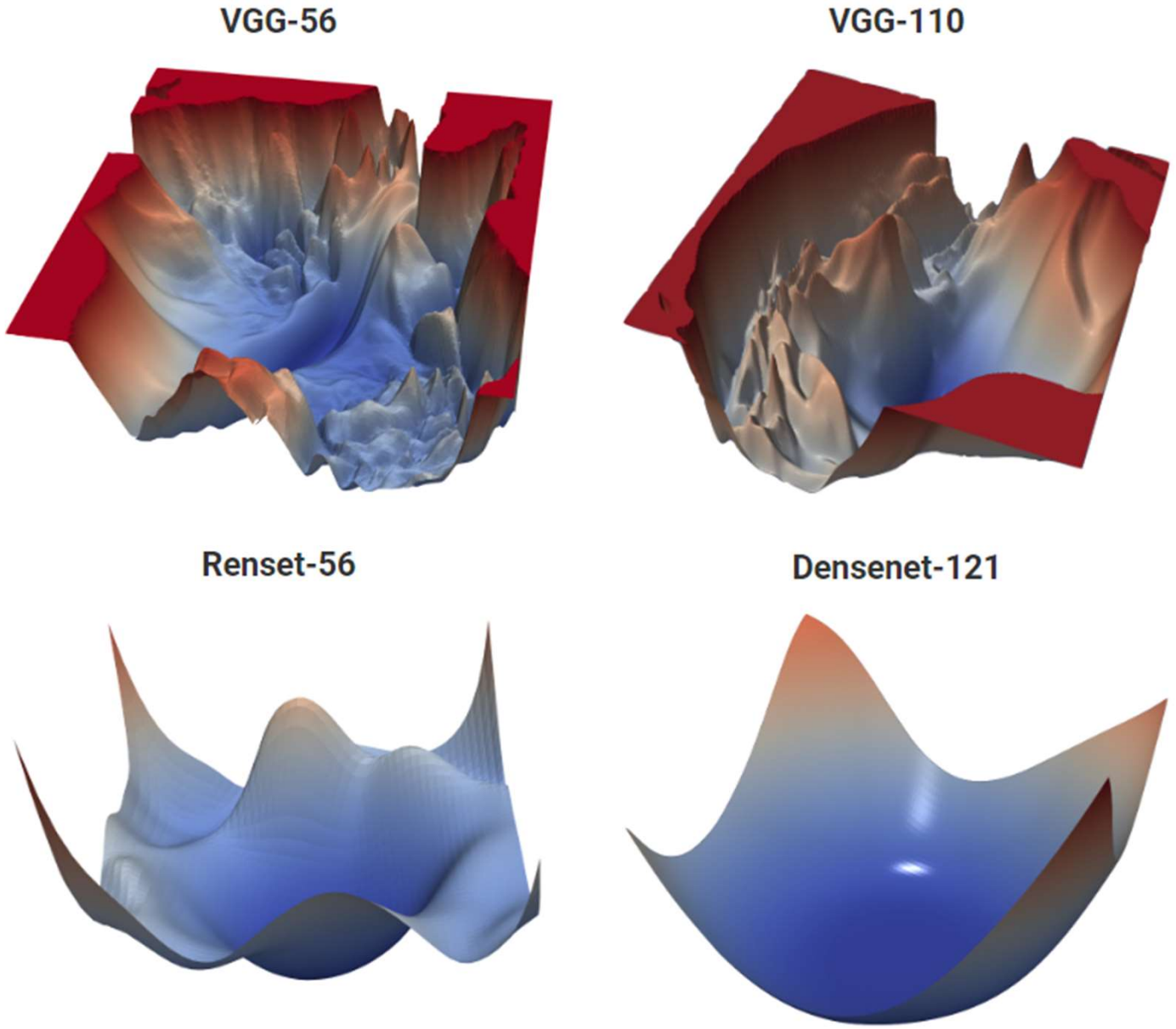
Concave Function



Non-Convex Function



Neural Network Loss Landscape



Source: https://www.cs.umd.edu/~tomg/projects/landscapes/

Evaluation / Objective Function

Say we have the following objective function:

$$f(\mathbf{x}) = f(x_1, y_1, x_2, y_2, x_3, y_3)$$

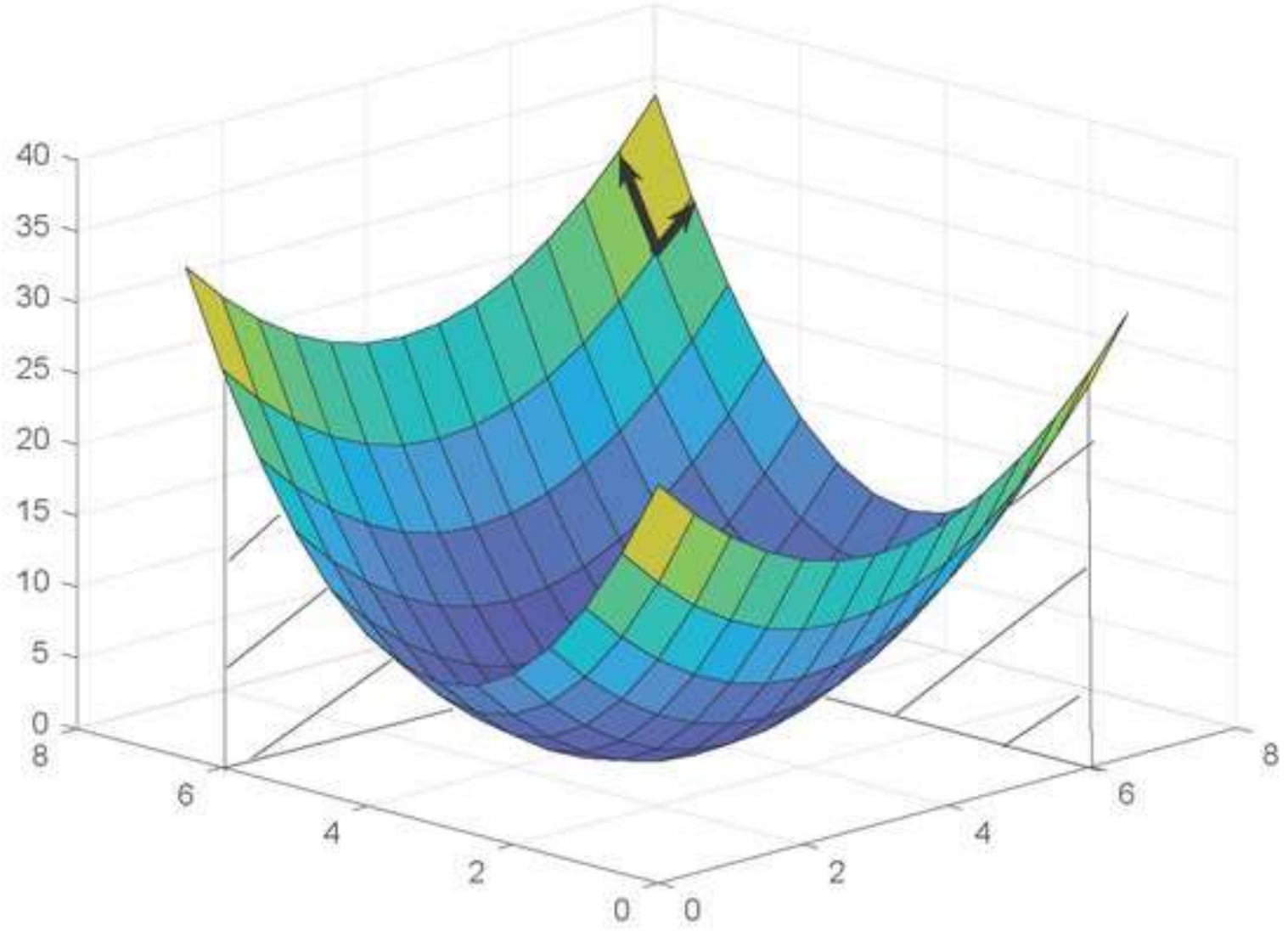
Gradients and Gradient Descent

The **gradient** of a function of many variables is a **vector pointing in the direction of the greatest increase** in a function.

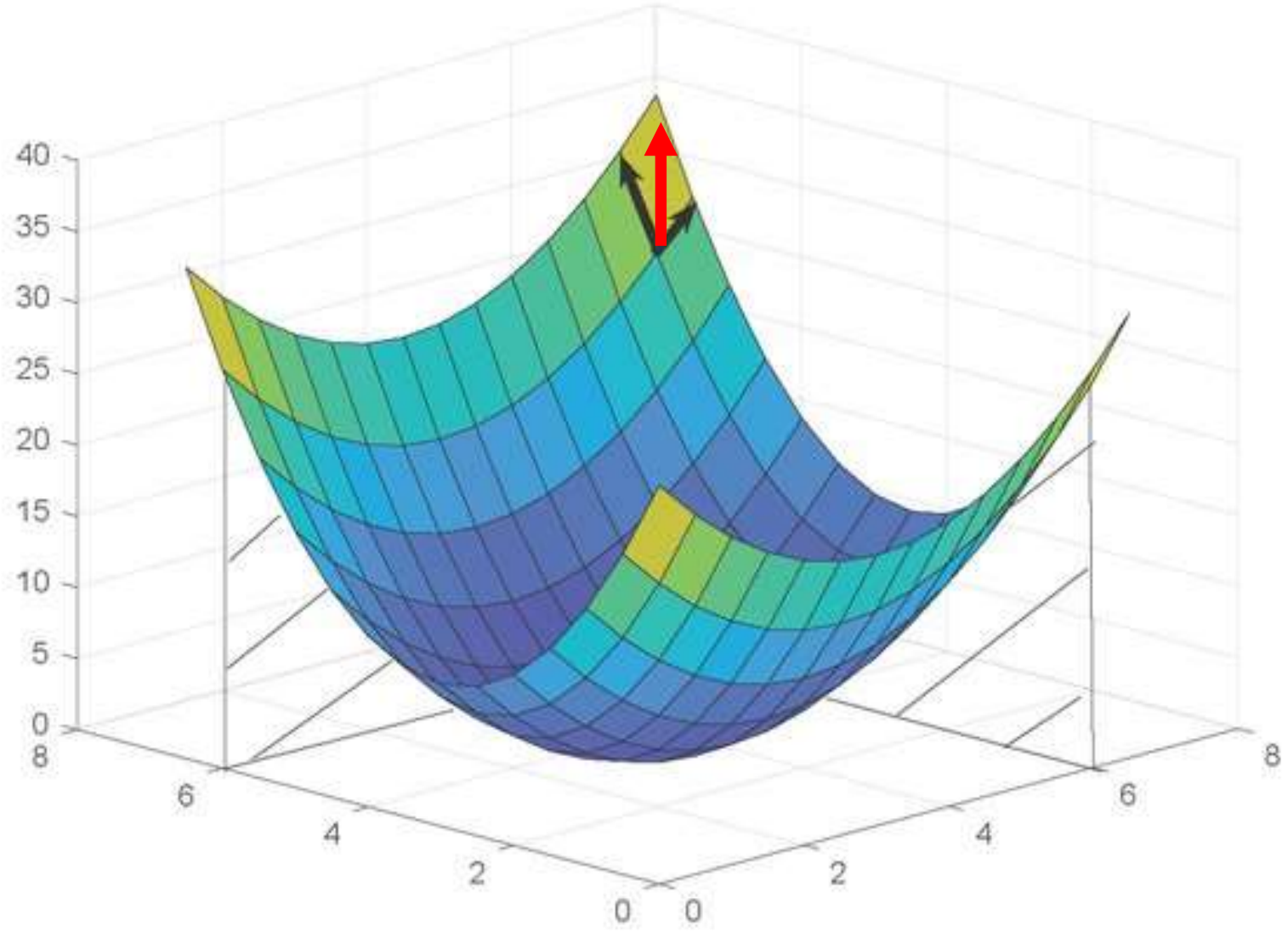
$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right)$$

Gradient Descent: Find the gradient of the function at the current point and move in the **opposite** direction.

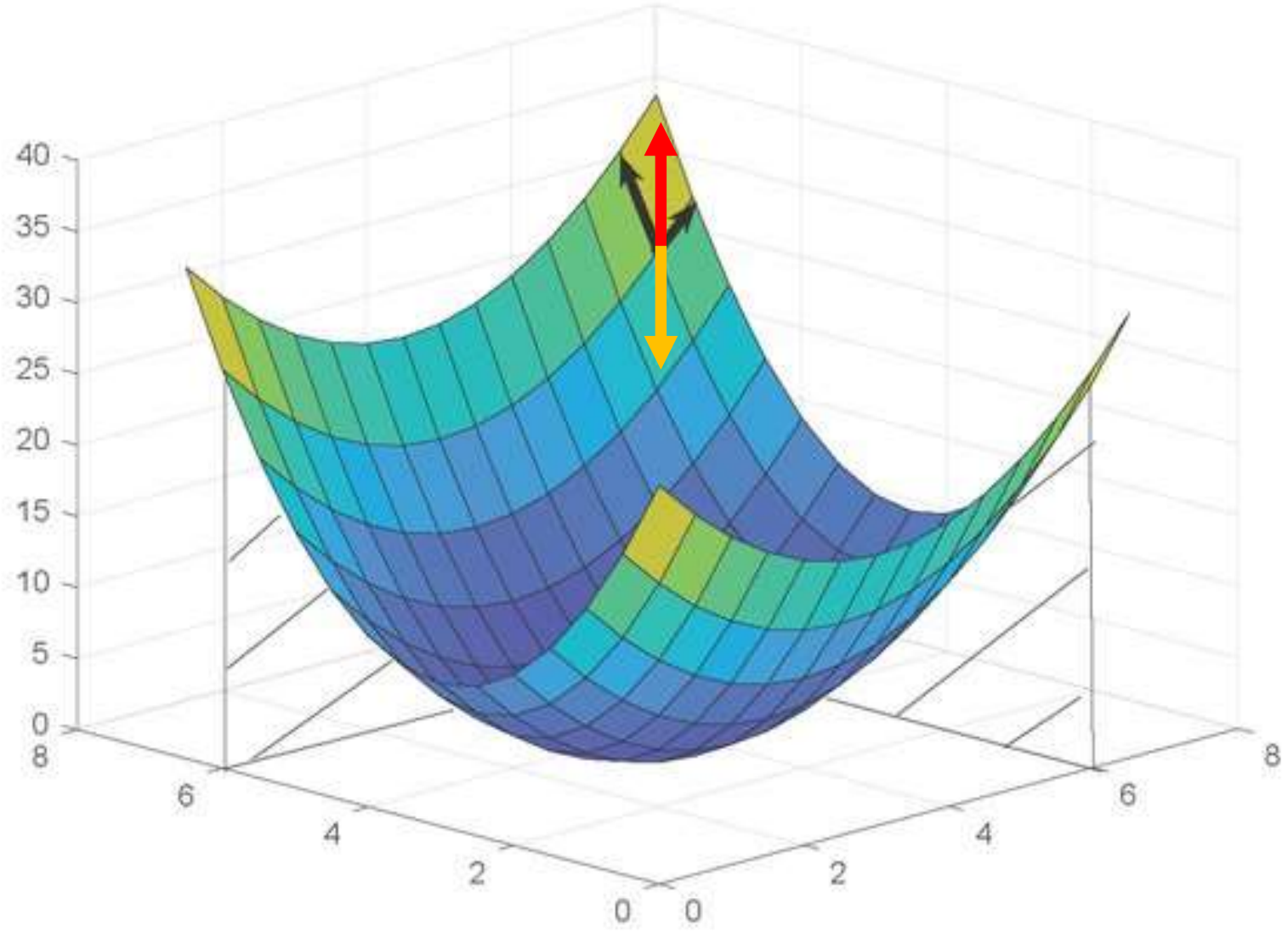
Gradient=Steepest Ascent Direction



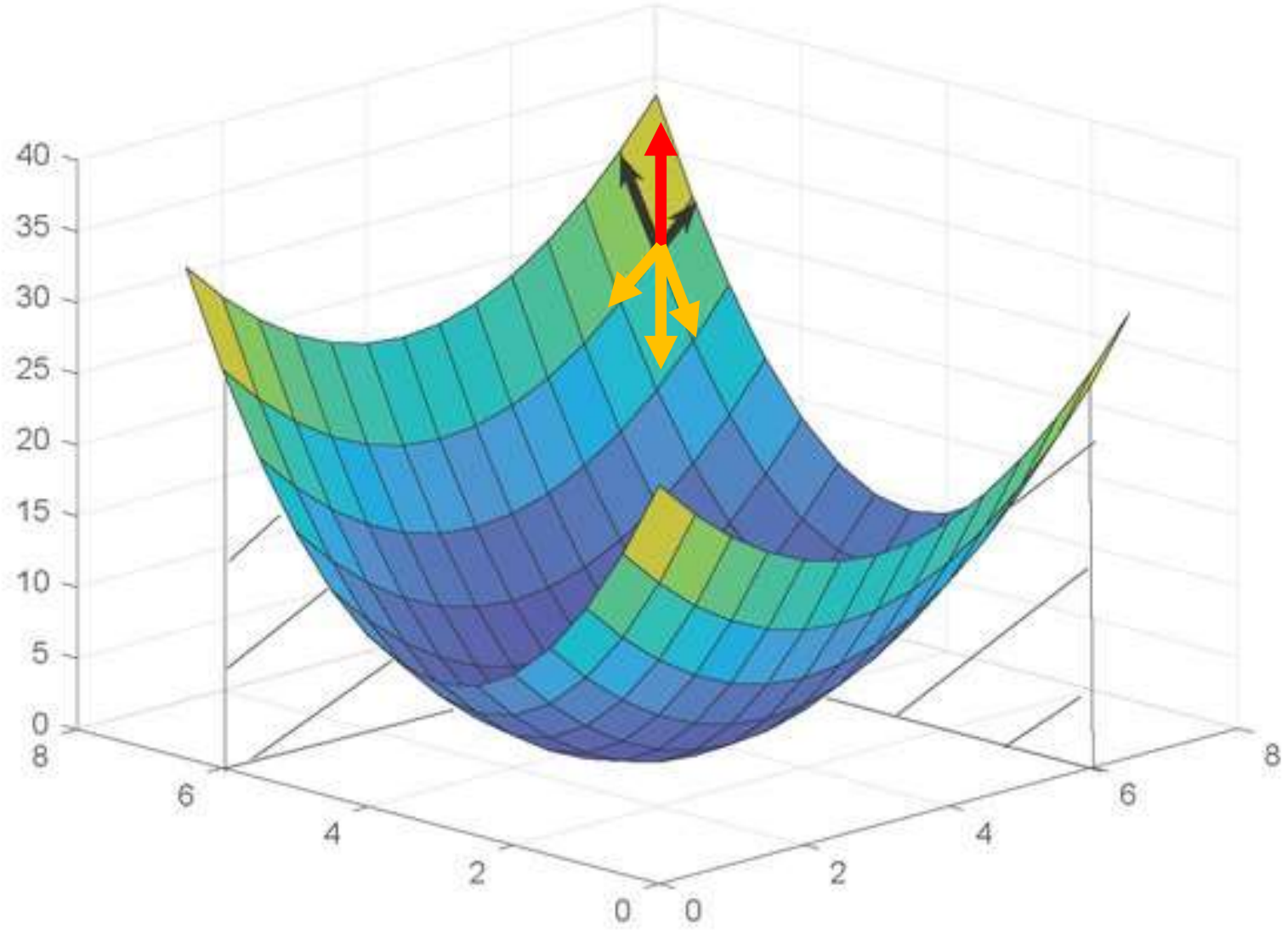
Gradient=Steepest Ascent Direction



-Gradient=Steepest Descent Direction



-Gradient=Steepest Descent Direction



Gradients and Gradient Descent

Gradient Descent Algorithm:

- **Pick an initial point x_0**
- **Iterate until convergence**

$$x_{t+1} = x_t - \gamma_t \nabla f(x_t)$$

where γ_t is the t^{th} step size (sometimes called learning rate)

Empirical Gradient

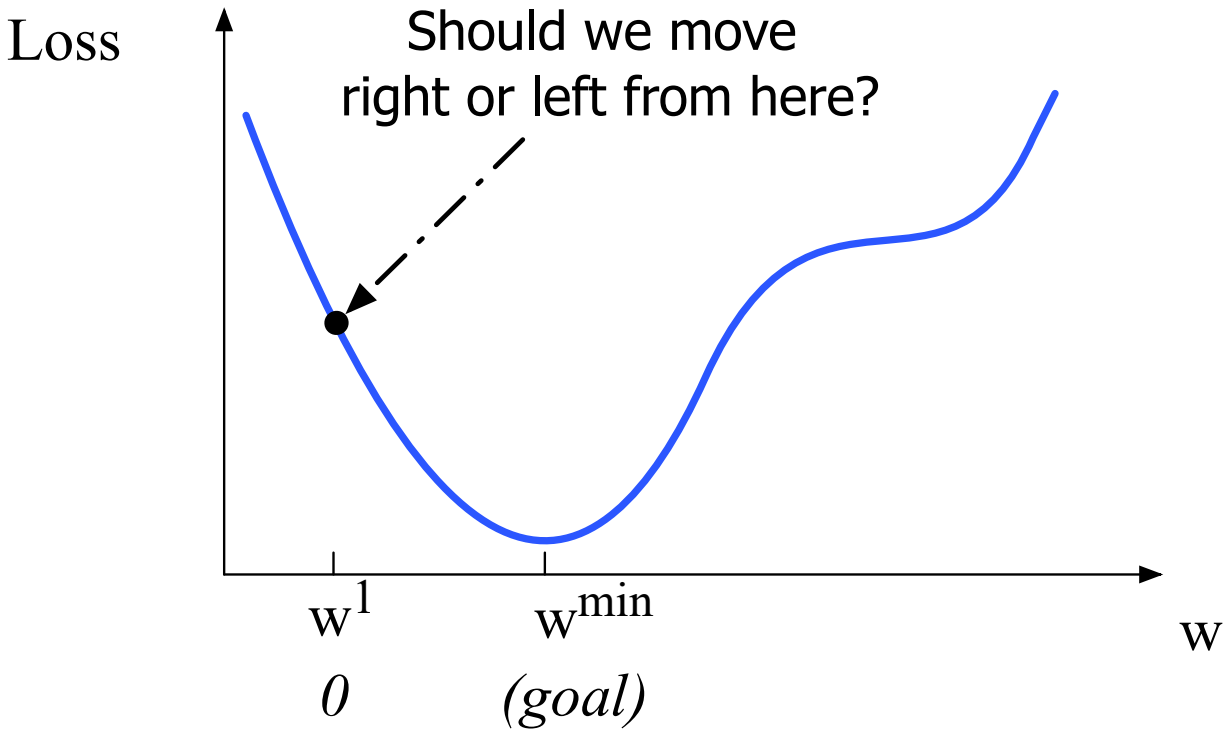
In other cases, the objective / evaluation function might not be available in a differentiable form at all.

In those cases, a so-called empirical gradient can be determined by evaluating the response to small increments and decrements in each coordinate.

Minimizing Functions

Q: Given current **w**, should we make it bigger or smaller?

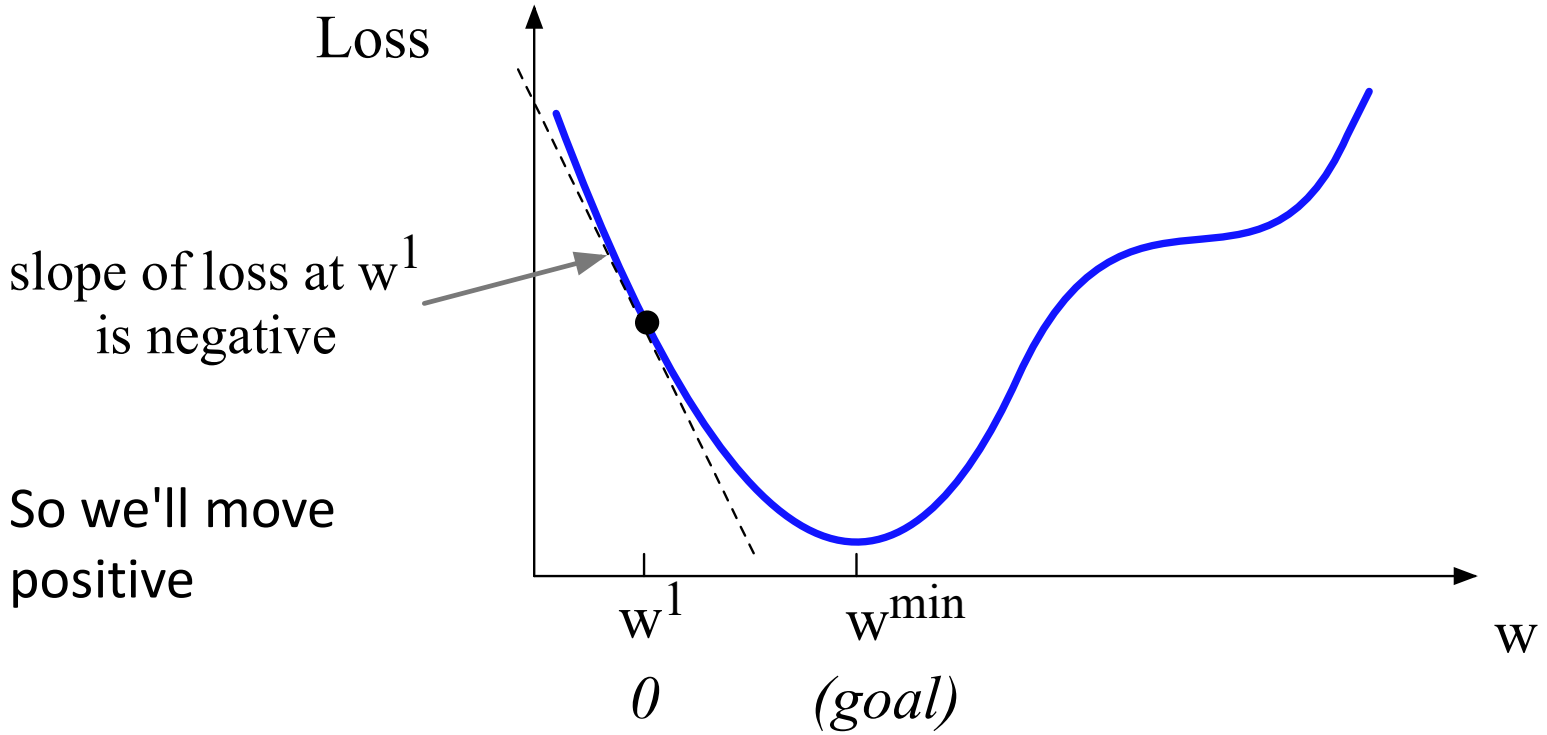
A: Move **w** in the reverse direction from the slope of the function



Minimizing Functions

Q: Given current **w**, should we make it bigger or smaller?

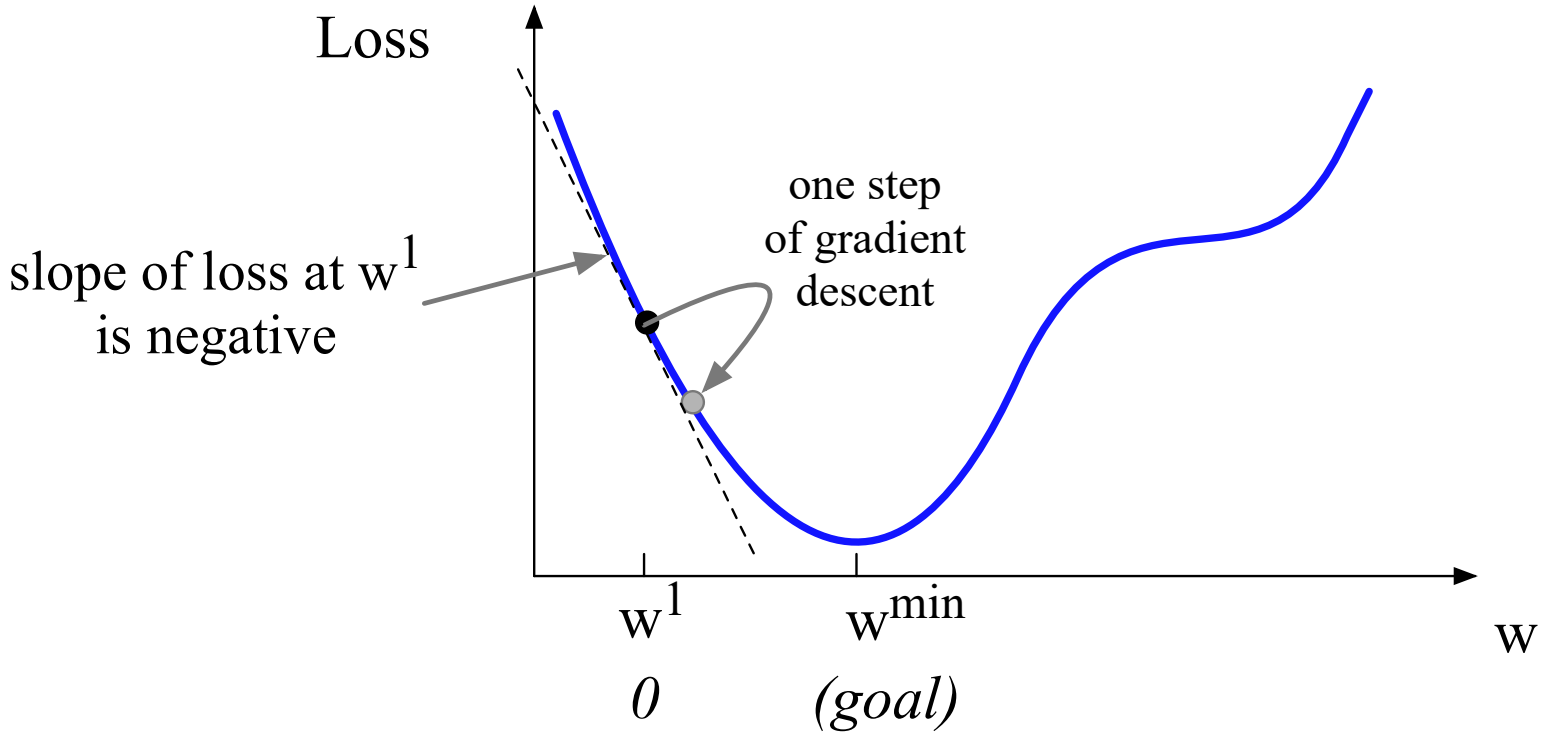
A: Move **w** in the reverse direction from the slope of the function



Minimizing Functions

Q: Given current **w**, should we make it bigger or smaller?

A: Move **w** in the reverse direction from the slope of the function

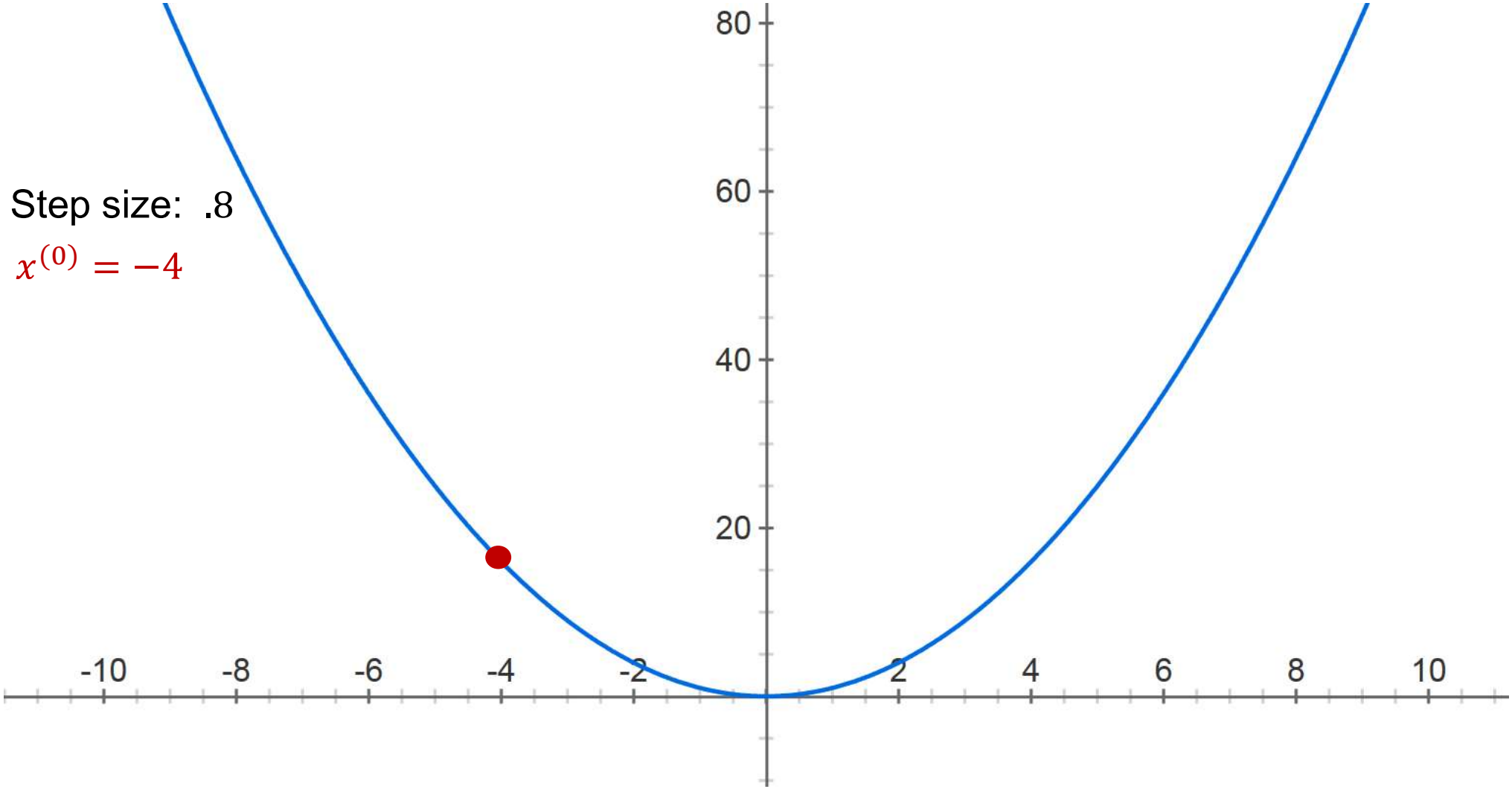


Minimizing Functions

$f(x) = x^2$

Step size: .8

$x^{(0)} = -4$



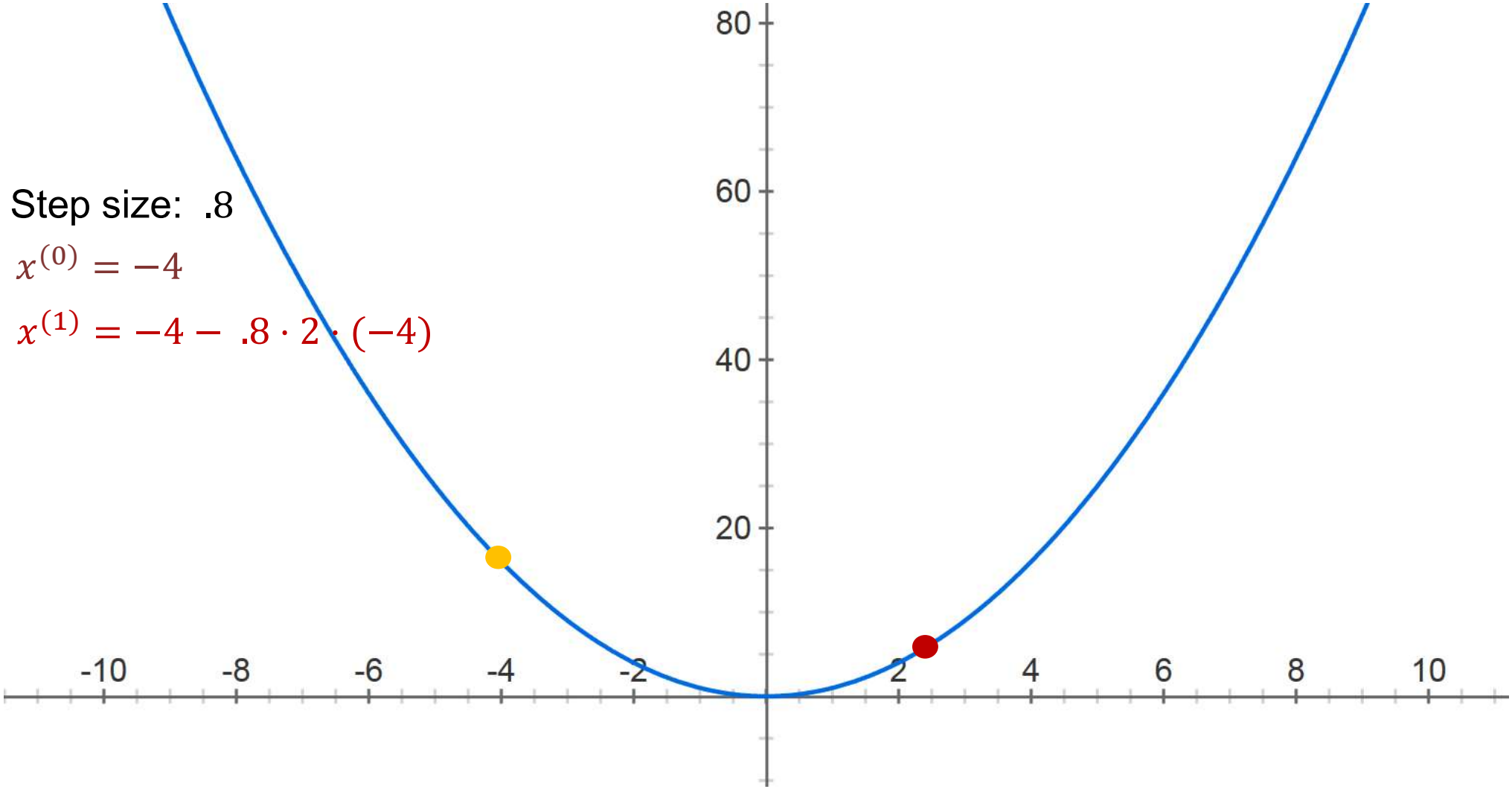
Minimizing Functions

$f(x) = x^2$

Step size: .8

$x^{(0)} = -4$

$x^{(1)} = -4 - .8 \cdot 2 \cdot (-4)$



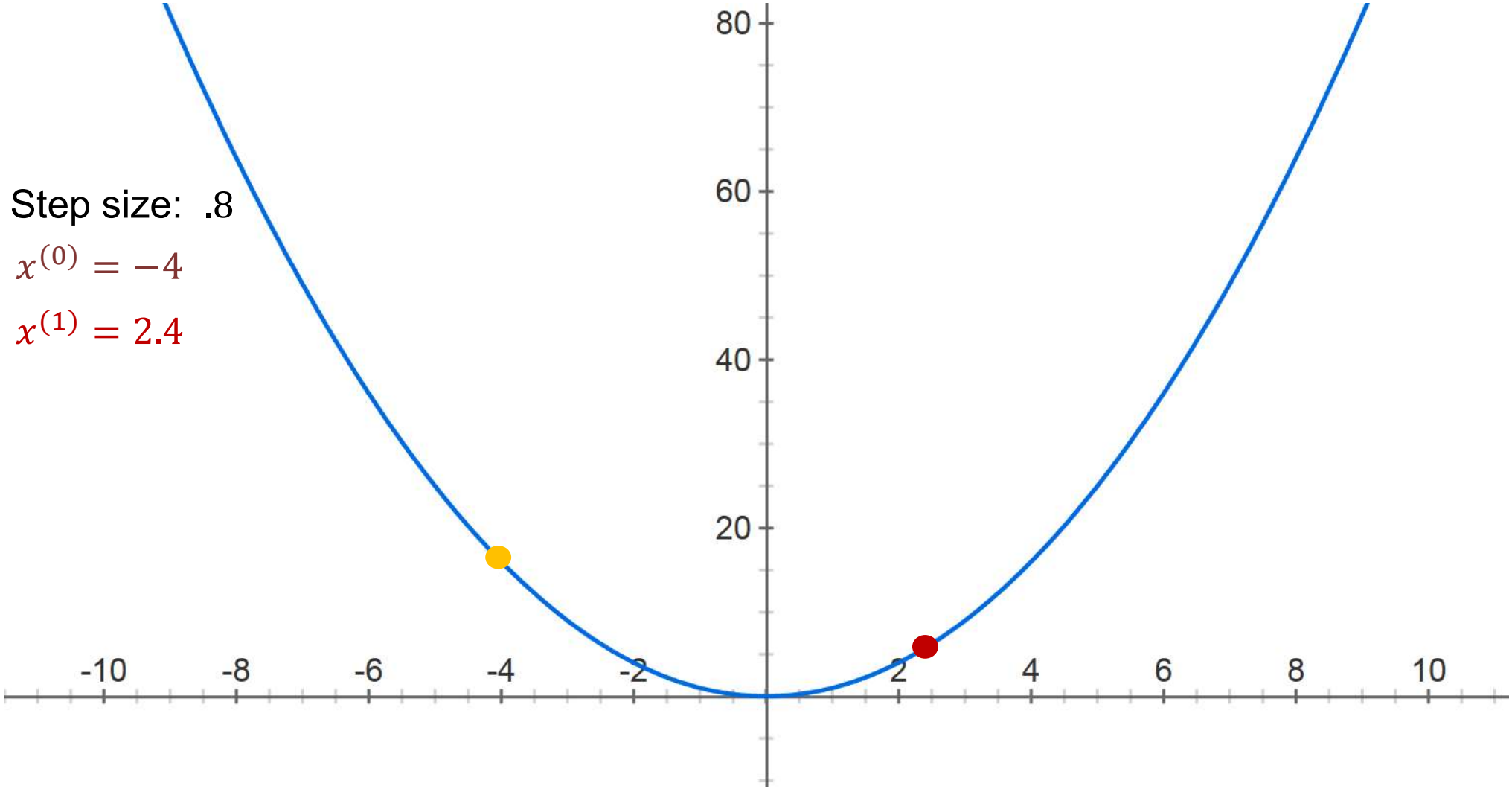
Minimizing Functions

$f(x) = x^2$

Step size: .8

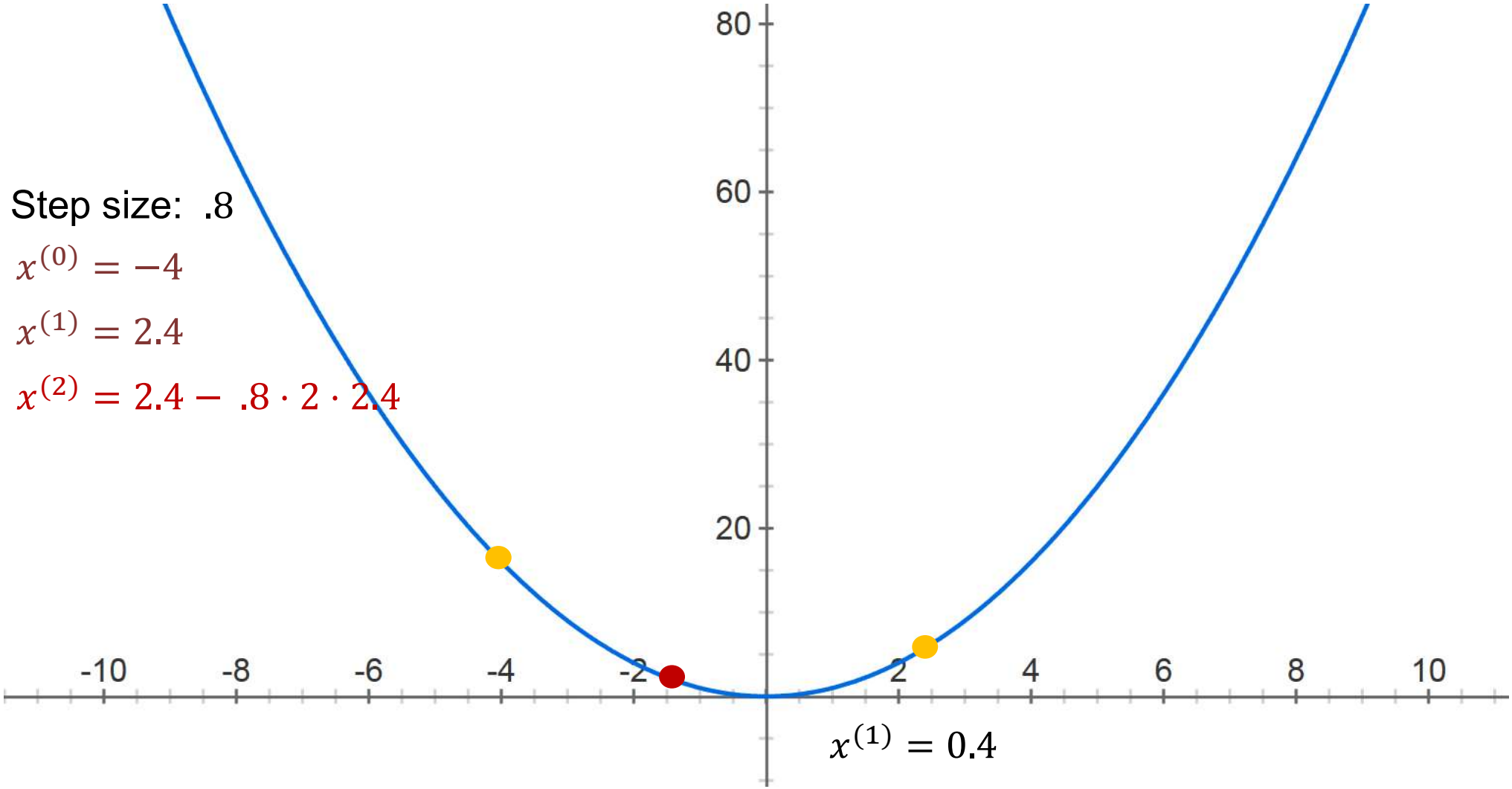
$x^{(0)} = -4$

$x^{(1)} = 2.4$



Minimizing Functions

$f(x) = x^2$



Minimizing Functions

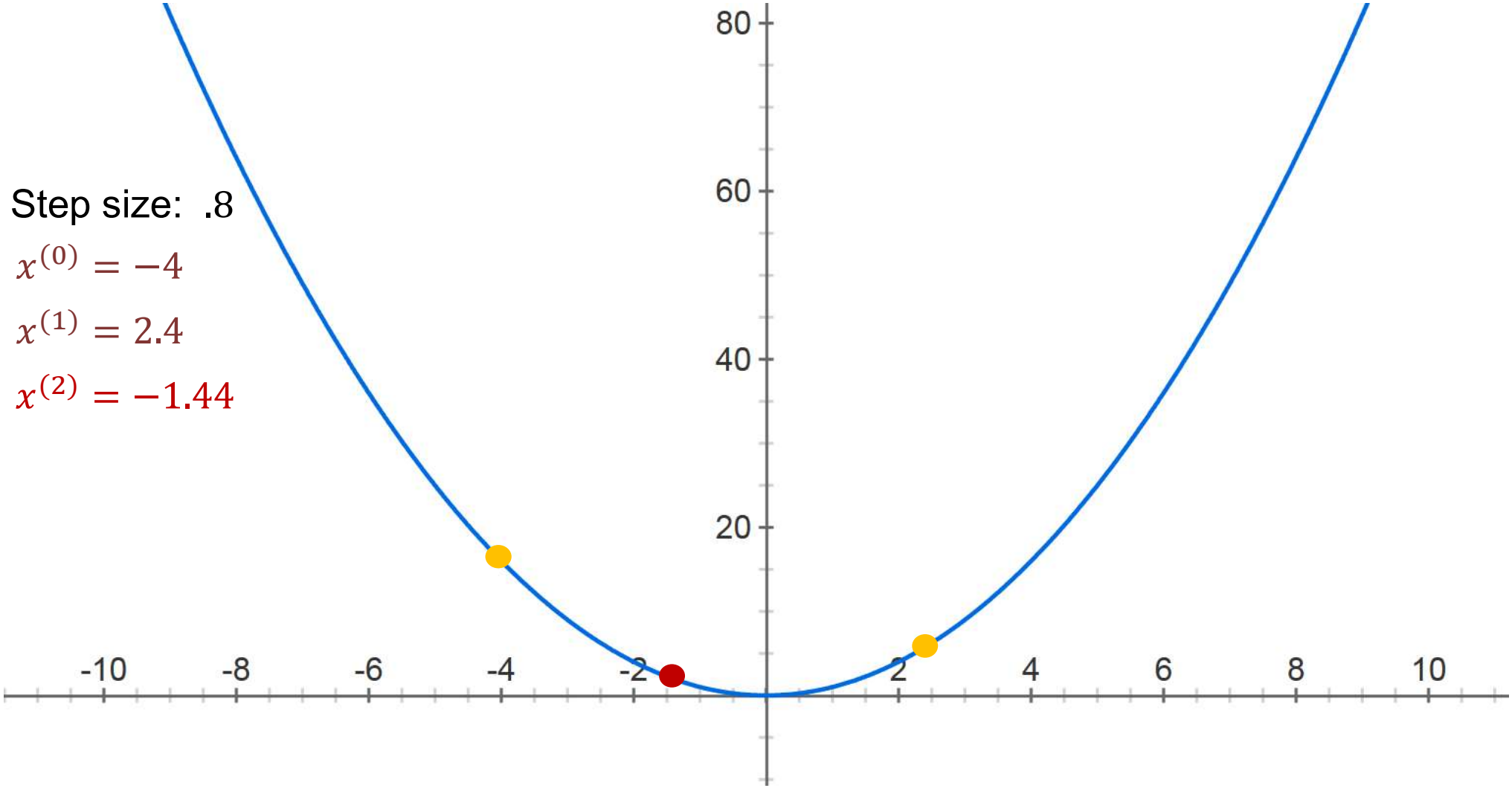
$f(x) = x^2$

Step size: .8

$x^{(0)} = -4$

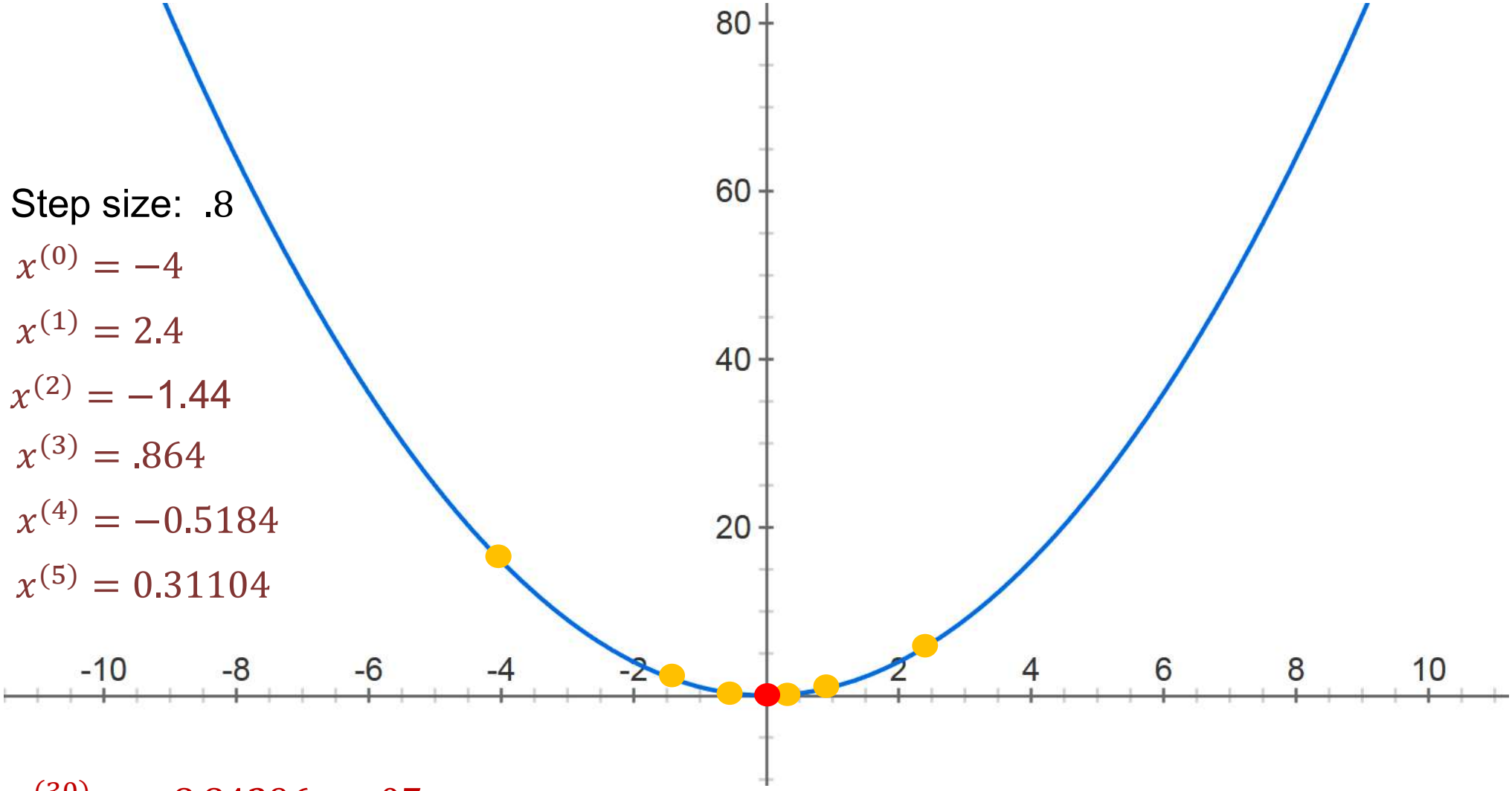
$x^{(1)} = 2.4$

$x^{(2)} = -1.44$



Minimizing Functions

$f(x) = x^2$



Step size: .8

$x^{(0)} = -4$

$x^{(1)} = 2.4$

$x^{(2)} = -1.44$

$x^{(3)} = .864$

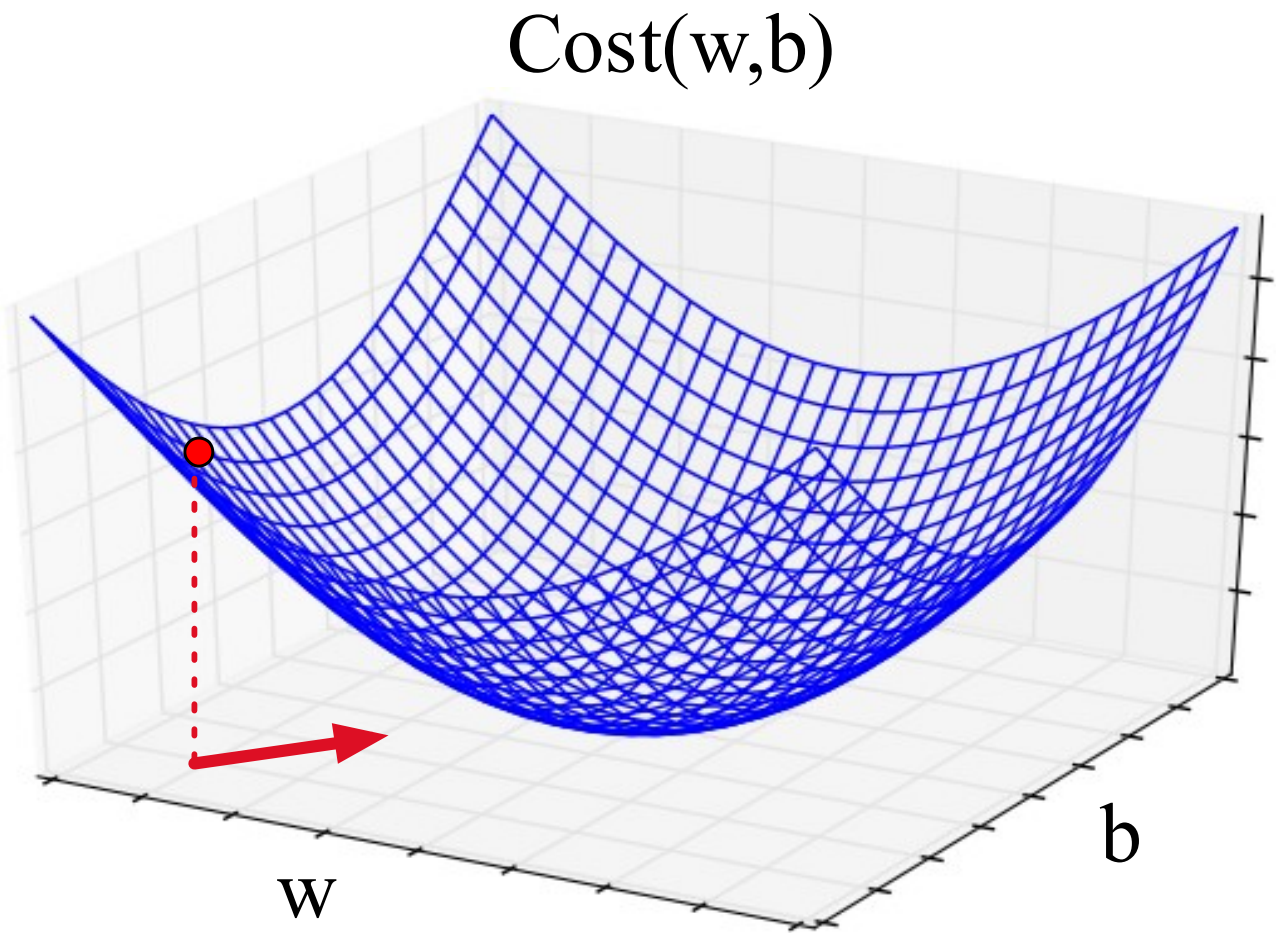
$x^{(4)} = -0.5184$

$x^{(5)} = 0.31104$

$x^{(30)} = -8.84296e - 07$

Gradients: Visualized

Visualizing the gradient vector at the **red** point



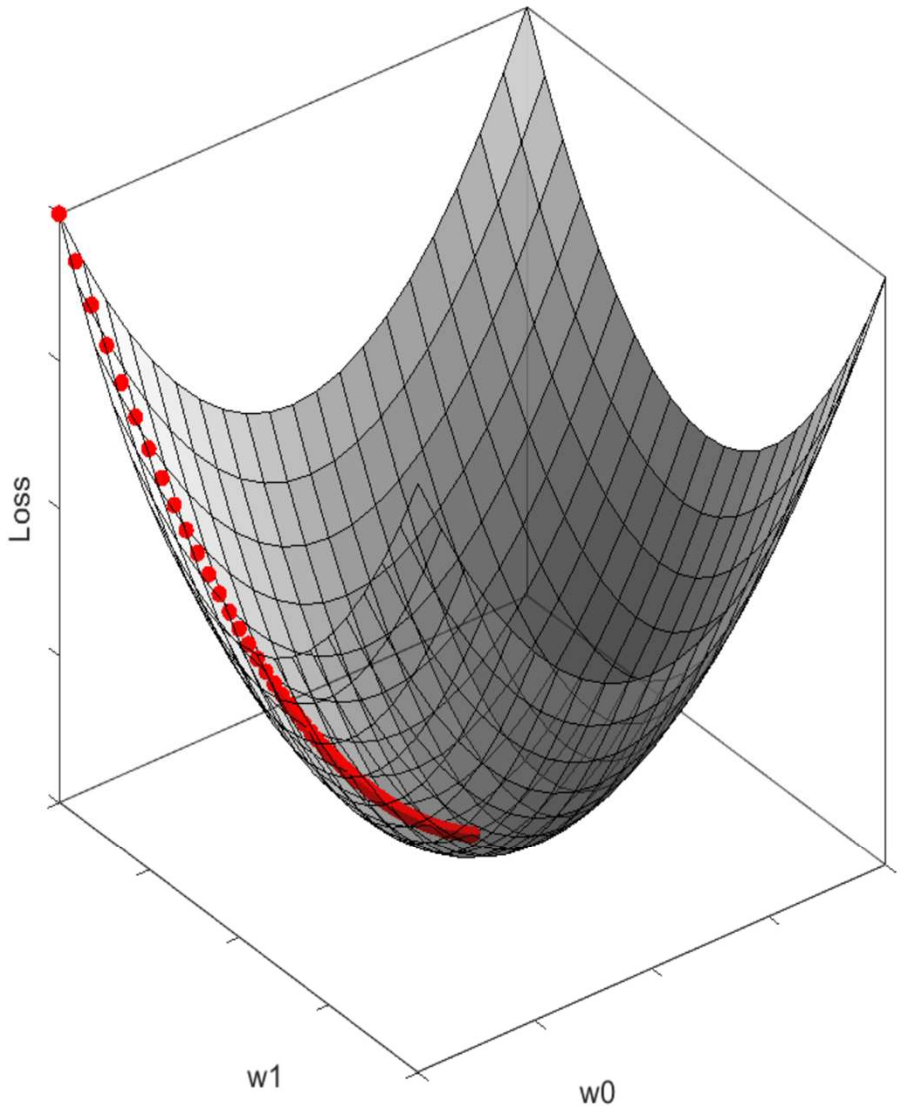
Gradients and Learning Rate / Step

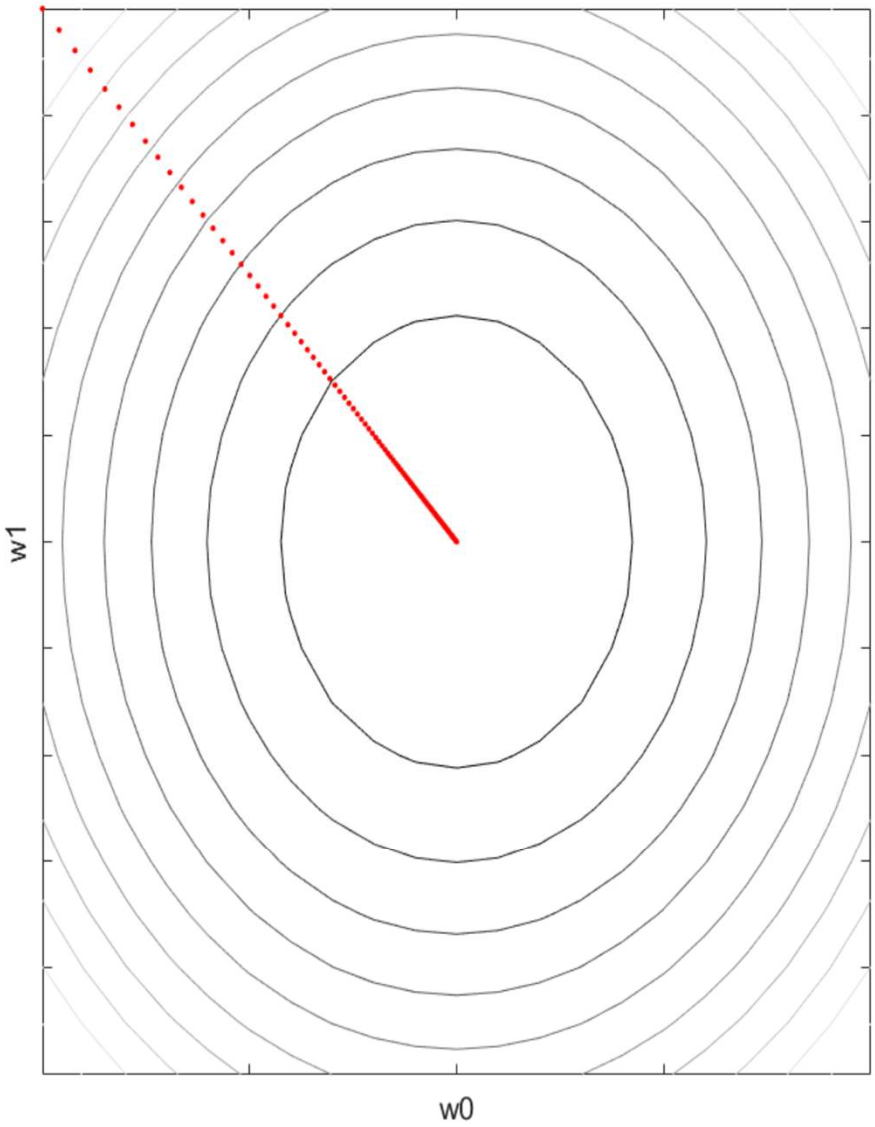
- The value of the gradient (slope in our example) $\frac{d}{dw} L(f(x;w), y)$ weighted by a **learning rate / step** η

- Higher learning rate means move **w** faster

$$w^{t+1} = w^t - \eta \frac{d}{dw} L(f(x;w), y)$$

Gradient Descent





Gradients and Gradient Descent

Gradient Descent Algorithm:

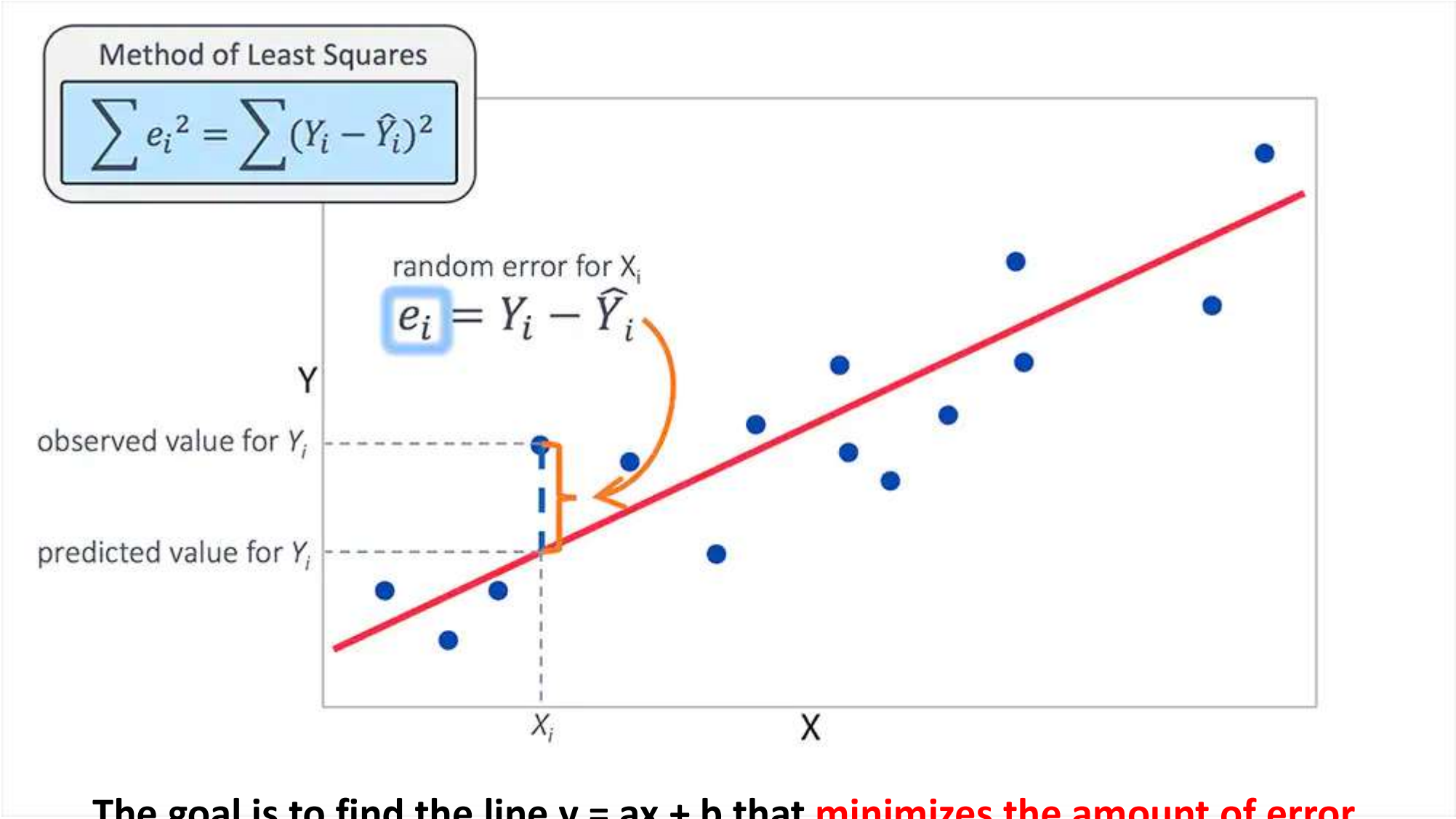
- **Pick an initial point x_0**
- **Iterate until convergence**

$$x_{t+1} = x_t - \gamma_t \nabla f(x_t)$$

where γ_t is the t^{th} step size (sometimes called learning rate)

When to stop?

Linear Regression Using Least-Squares



The goal is to find the line y = ax + b that minimizes the amount of error.

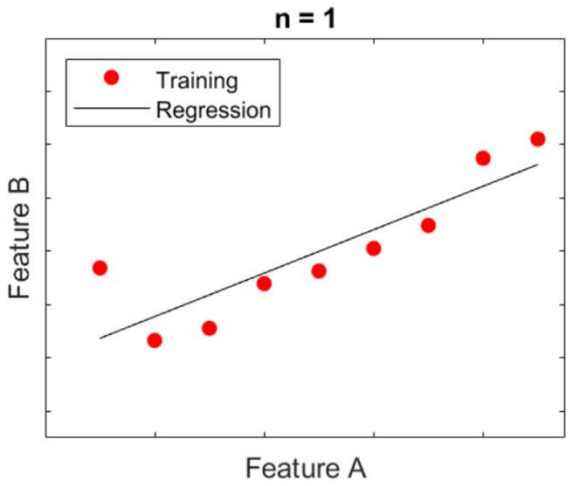
Source: https://www.jmp.com/en_us/statistics-knowledge-portal/what-is-multiple-regression/fitting-multiple-regression-model.html

Origins of ‘Regression’ Term

Source: https://en.wikipedia.org/wiki/Francis_Galton

Sir Francis Galton, an English polymath studied, among other things, heredity in humans. In one experiment he compared children height to their parent heights. He observed that children heights **regressed** towards the average height of an adult.

Univariate Linear Regression



Real function: $y = w_1 * x + w_0$

Hypothesis / model: $h_{\boldsymbol{w}}(x) = w_1 * x + w_0$

where $\boldsymbol{w} = \langle w_0, w_1 \rangle$ **are coefficients / weights.**

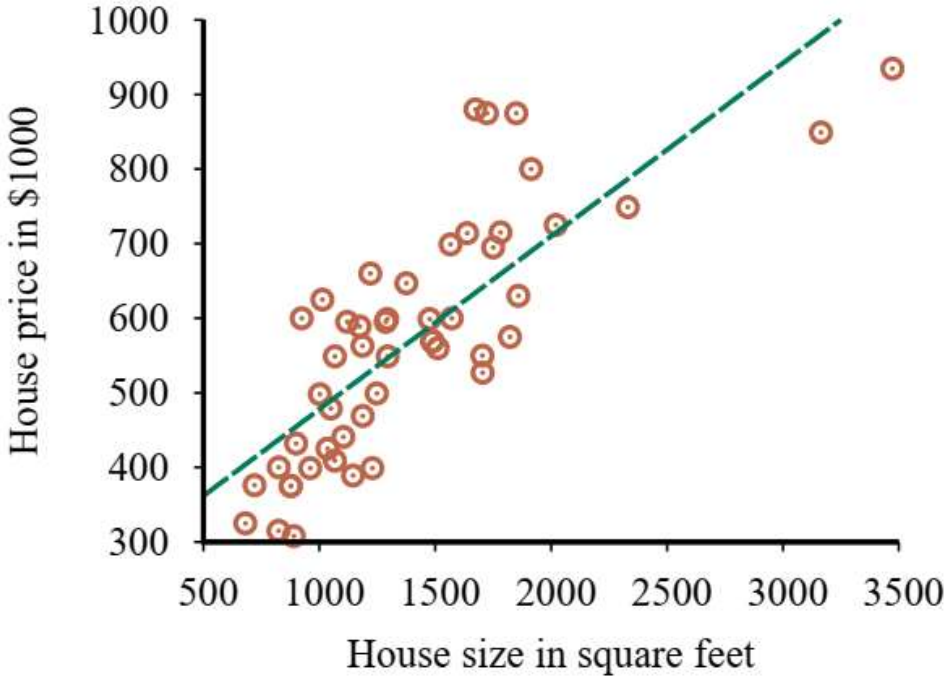
Squared-error loss function:

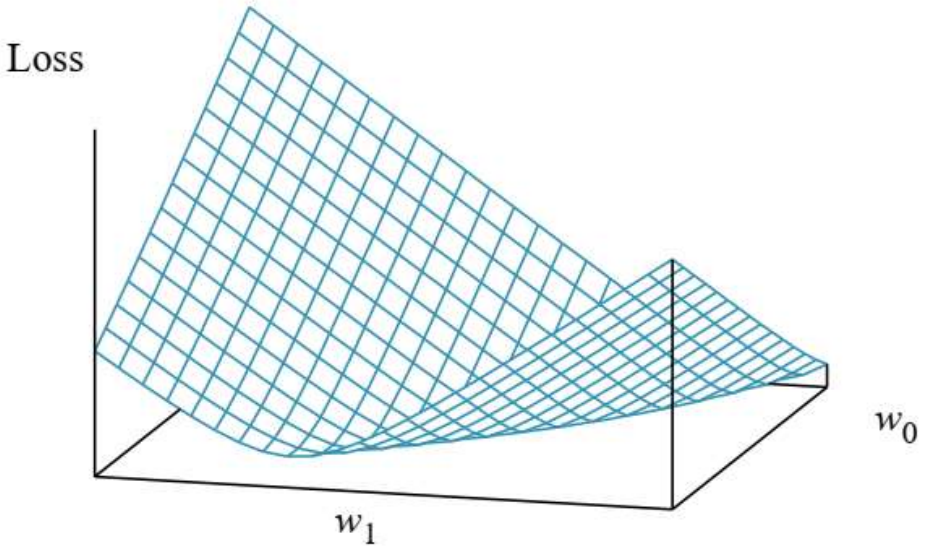
$$Loss(h_{\boldsymbol{w}}) = \sum_{j=1}^{N} \left(y_j - h_{\boldsymbol{w}}(x_j)\right)^2 = \sum_{j=1}^{N} \left(y_j - (w_1 * x_j + w_0)\right)^2$$

We want to find $\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin}\, Loss(h_{\boldsymbol{w}})$:

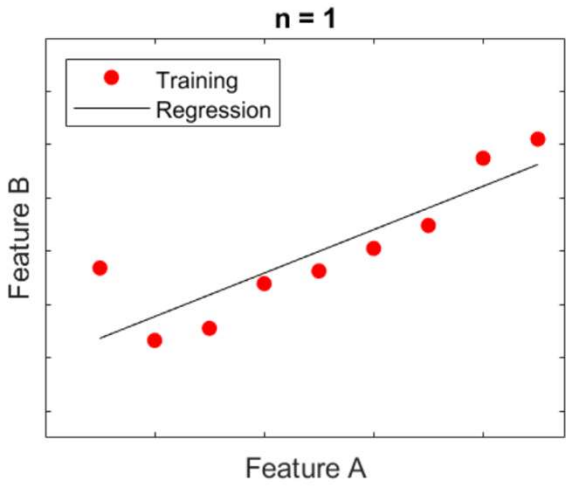
solve $\frac{\partial Loss(h_{\boldsymbol{w}})}{\partial w_0} = 0$, $\frac{\partial Loss(h_{\boldsymbol{w}})}{\partial w_1} = 0$

Weight / Parameter Space





Linear Regression: Gradient Descent



Given a squared-error loss function:

$$Loss(h_{\boldsymbol{w}}) = \sum_{j=1}^{N} \left(y_j - h_{\boldsymbol{w}}(x_j)\right)^2 = \sum_{j=1}^{N} \left(y_j - (w_1 * x_j + w_0)\right)^2$$

We want to find $\boldsymbol{w}^* = \underset{\boldsymbol{w}}{argmin}\, Loss(h_{\boldsymbol{w}})$:

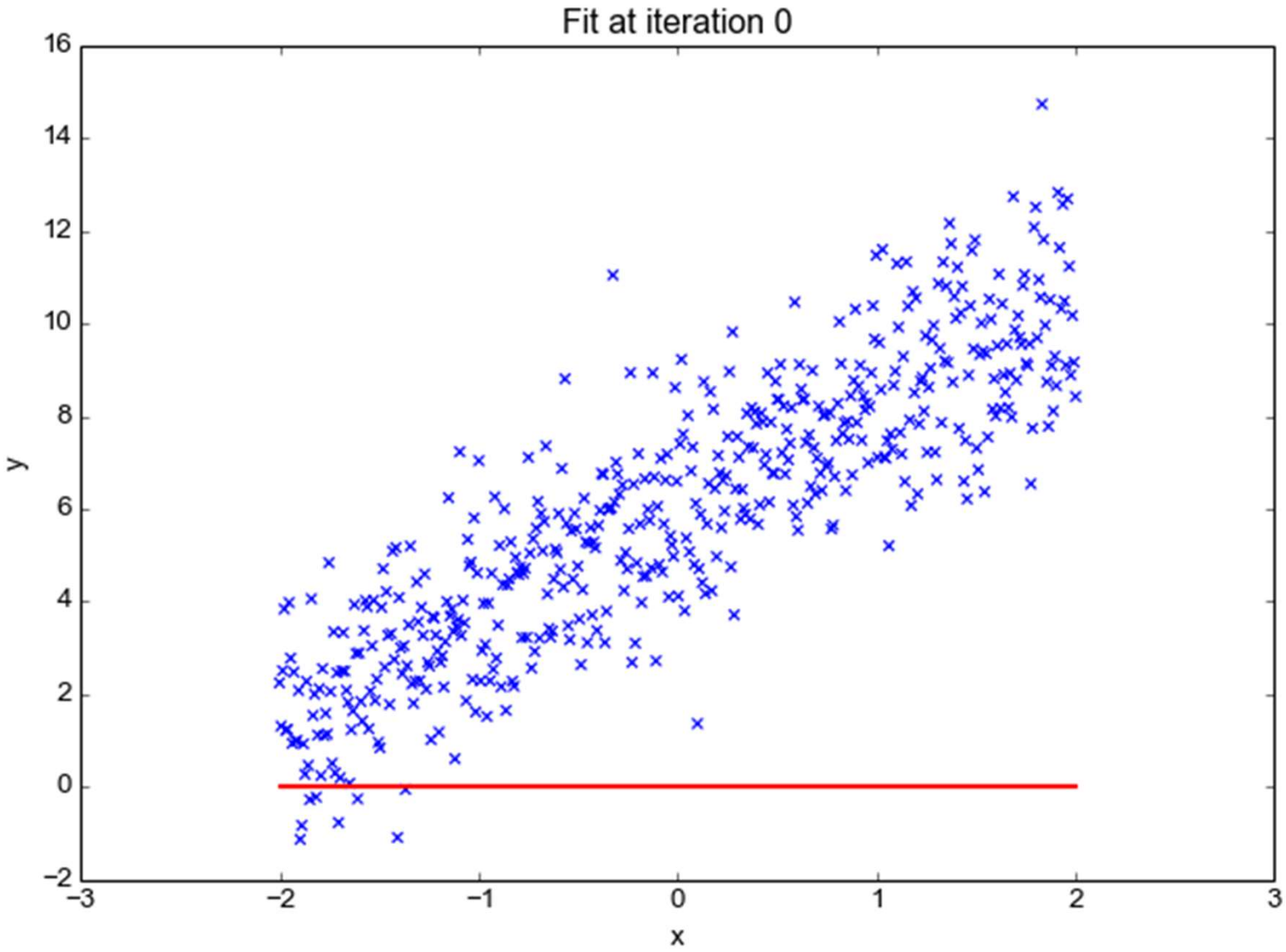
solve $\frac{\partial Loss(h_{\boldsymbol{w}})}{\partial w_0} = 0\,, \frac{\partial Loss(h_{\boldsymbol{w}})}{\partial w_1} = 0$

Sometimes these will be hard (or impossible) to solve. Gradient descent technique can be instead:

w ← **any point in the parameter space**
while not converged **do**
for each w_i **in w do** $w_i \leftarrow w_i - \alpha * \frac{\partial Loss(\boldsymbol{w})}{\partial\, w_i}$

α **- step size / learning rate**

Linear Regression: Gradient Descent



Source: Andrew Ng

Stochastic Gradient Descent

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

where: L is the loss function
f is a function parameterized by θ
x is the set of training inputs $x^{(1)}$, $x^{(2)}$, ..., $x^{(m)}$
y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$, ..., $y^{(m)}$

$\theta \leftarrow 0$

repeat til done

For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)

1. Optional (for reporting):
   Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$
   Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$
2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$
3. $\theta \leftarrow \theta - \eta\, g$

How are we doing on this tuple?
What is our estimated output $\hat{y}$?
How far off is $\hat{y}^{(i)}$) from the true output $y^{(i)}$?
How should we move θ to maximize loss?
Go the other way instead

return θ

Learning Rate

The learning rate η is a **hyperparameter**

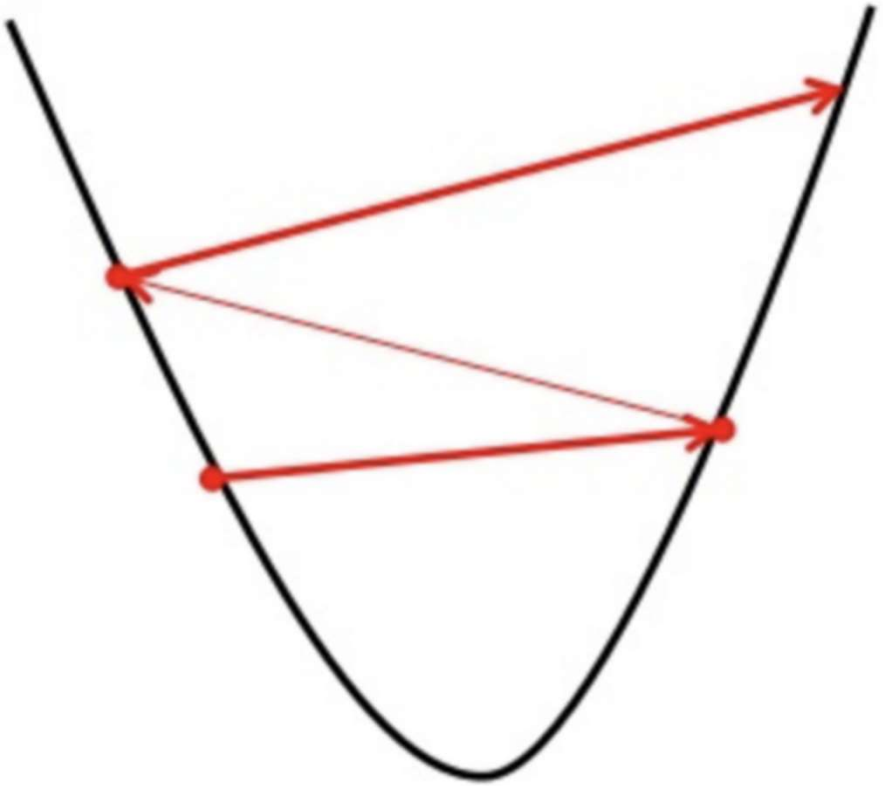
- too high: the learner will take big steps and overshoot
- too low: the learner will take too long

Hyperparameters:

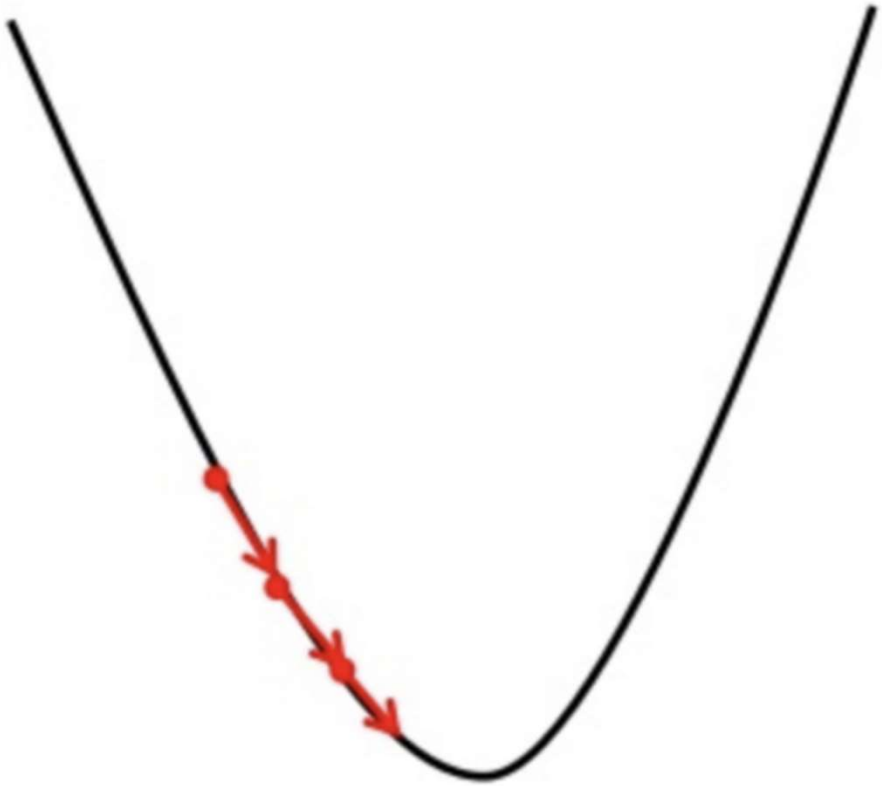
- Briefly, a special kind of parameter for an ML model
- Instead of being learned by algorithm from supervision (like regular parameters), they are chosen by algorithm designer.

Learning Rate / Step

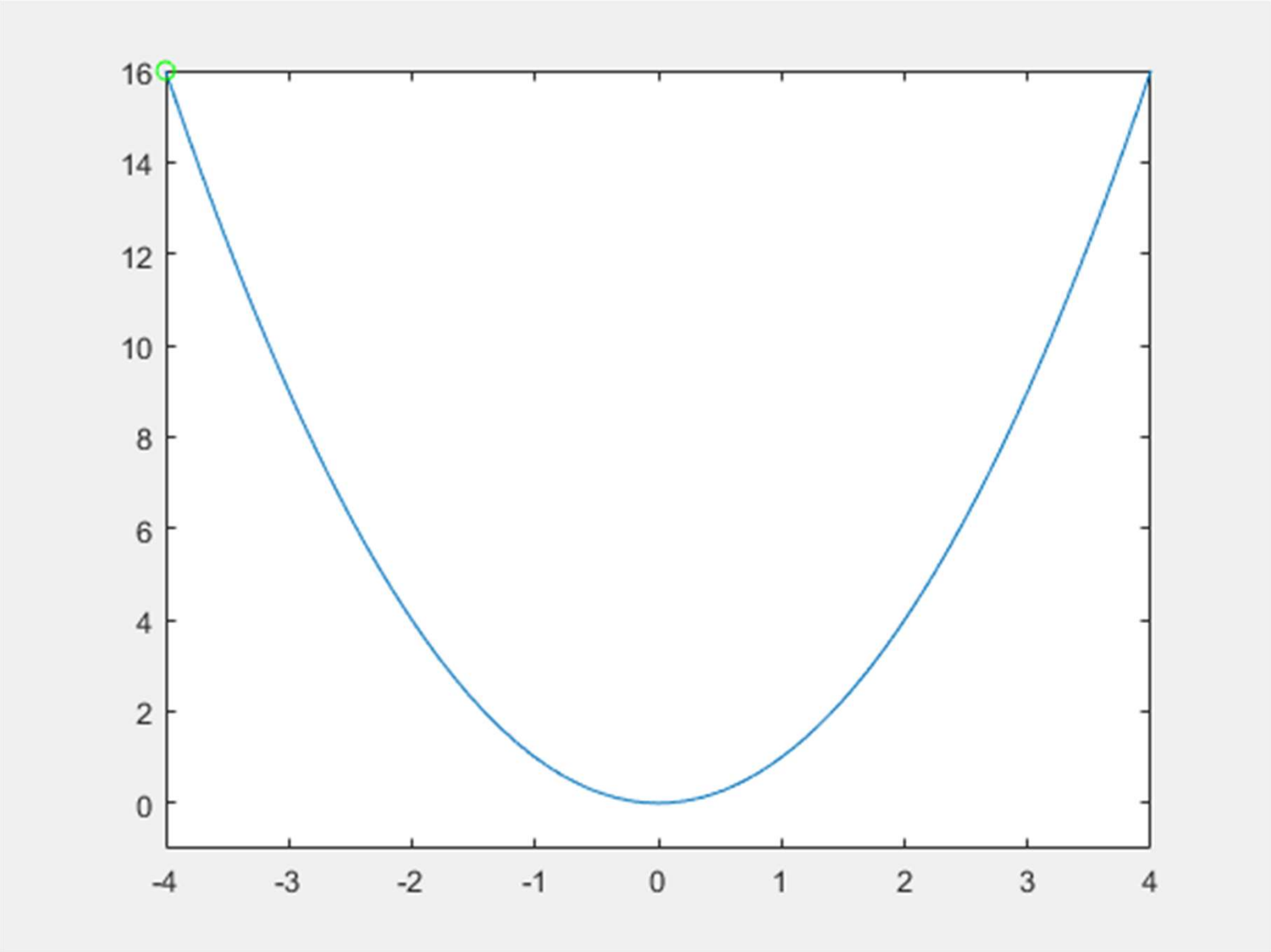
High Learning Rate / Step



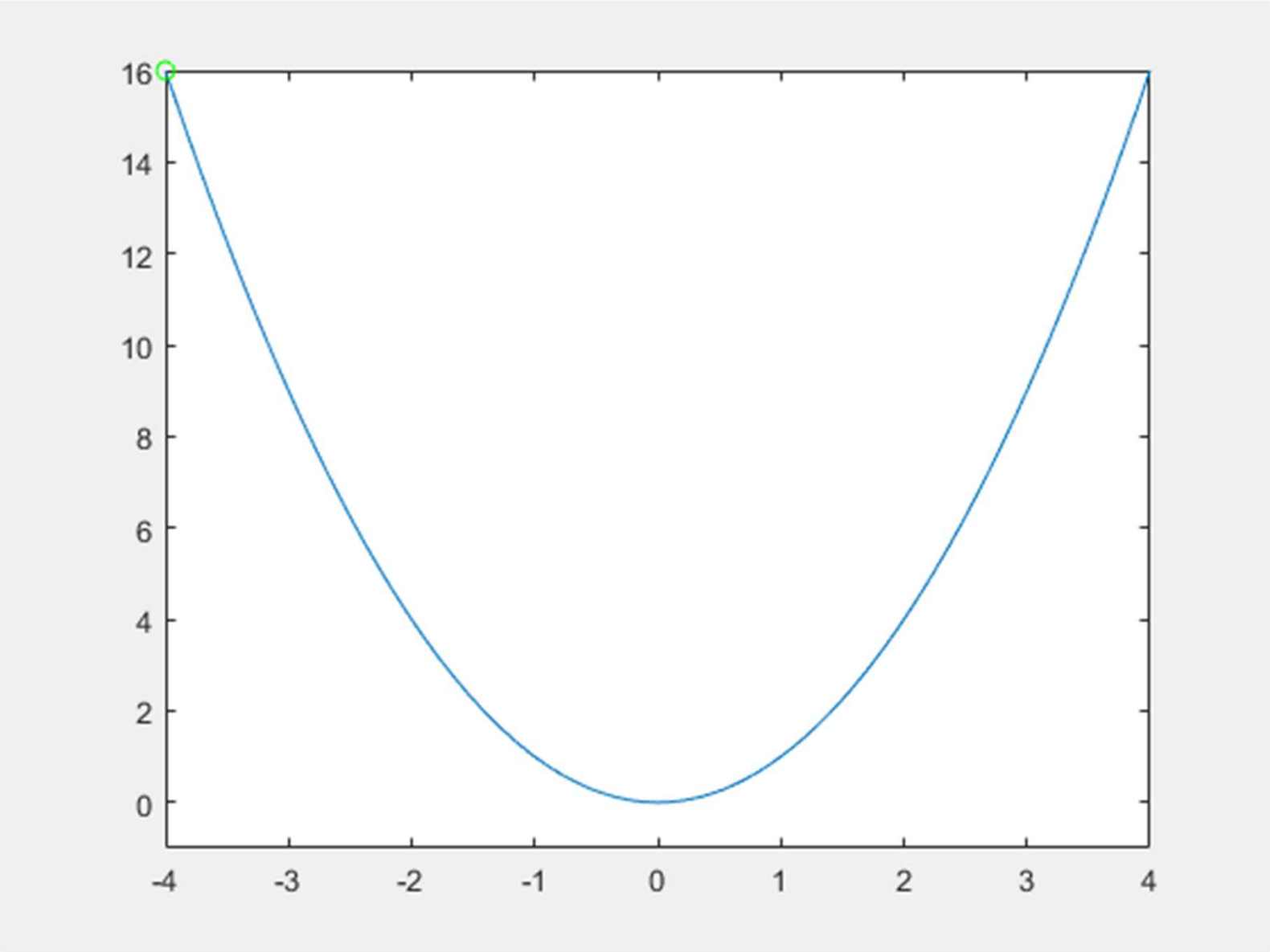
Low Learning Rate / Step



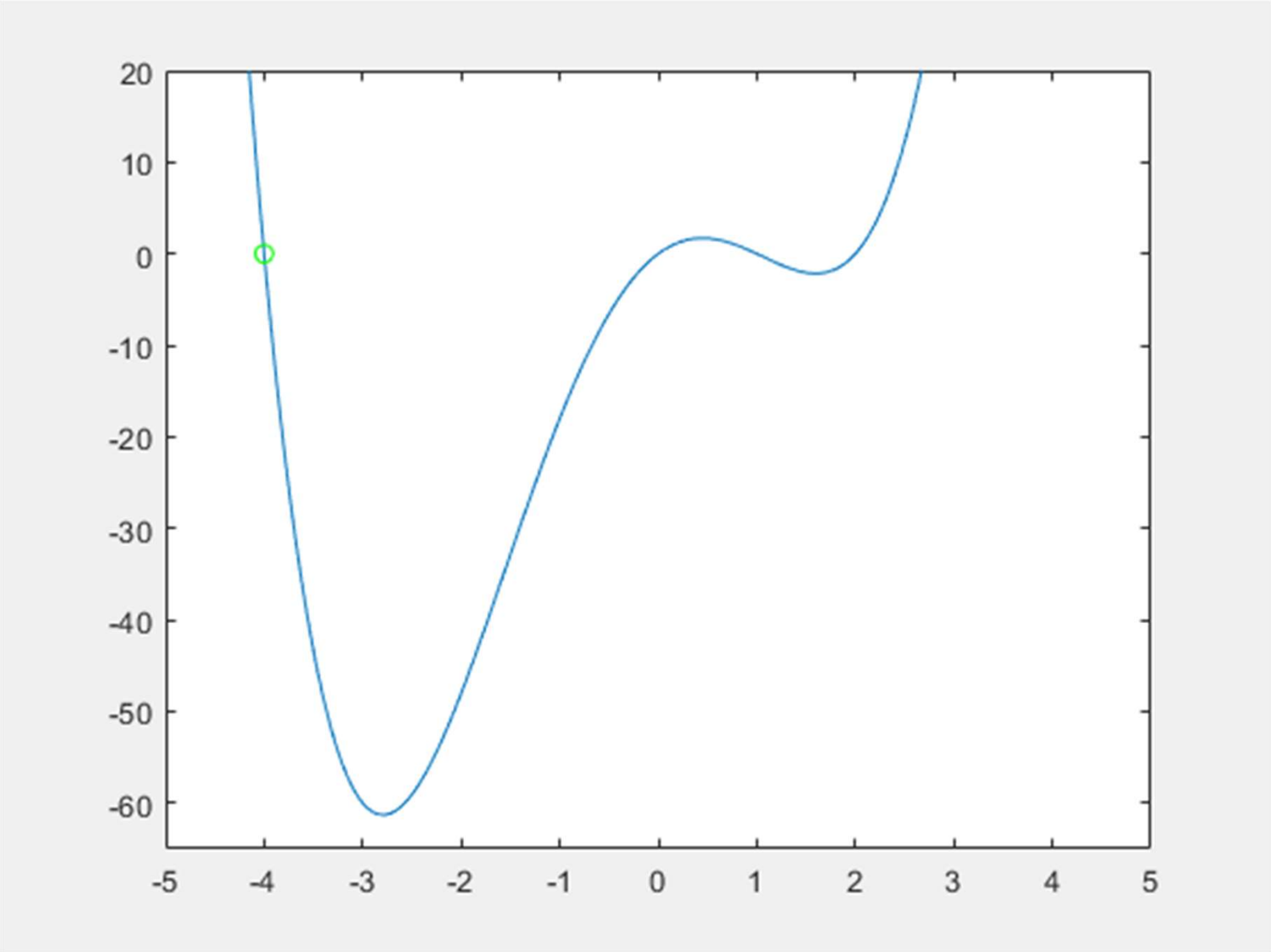
Rate / Step Size Matters



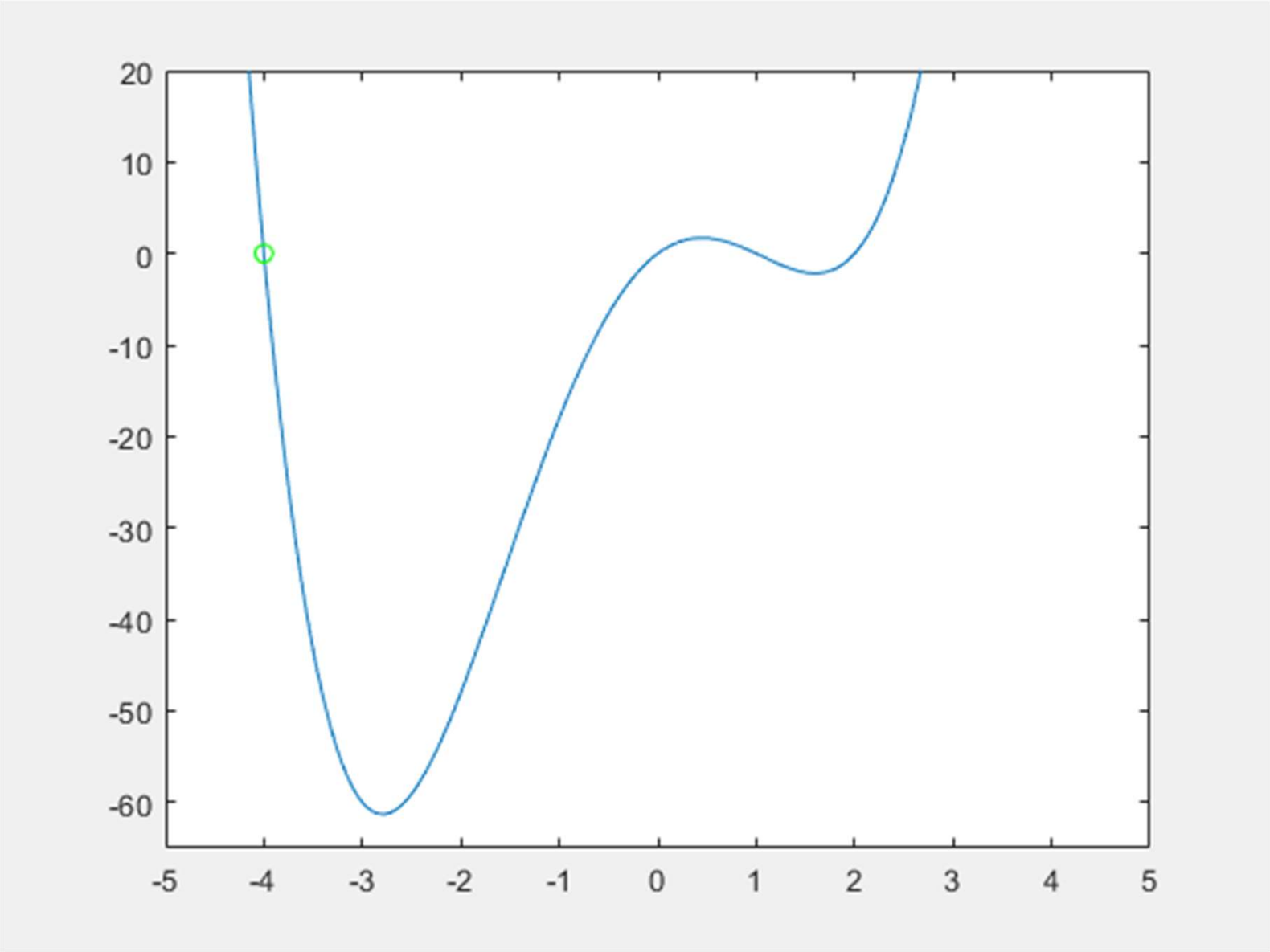
Rate / Step Size Matters



Rate / Step Size Matters

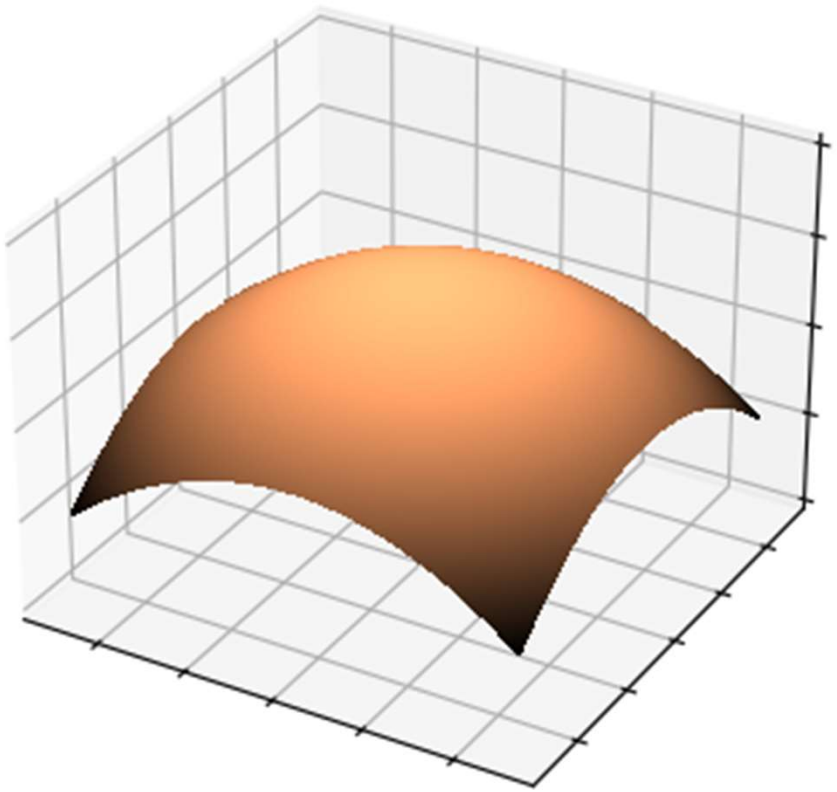


Rate / Step Size Matters

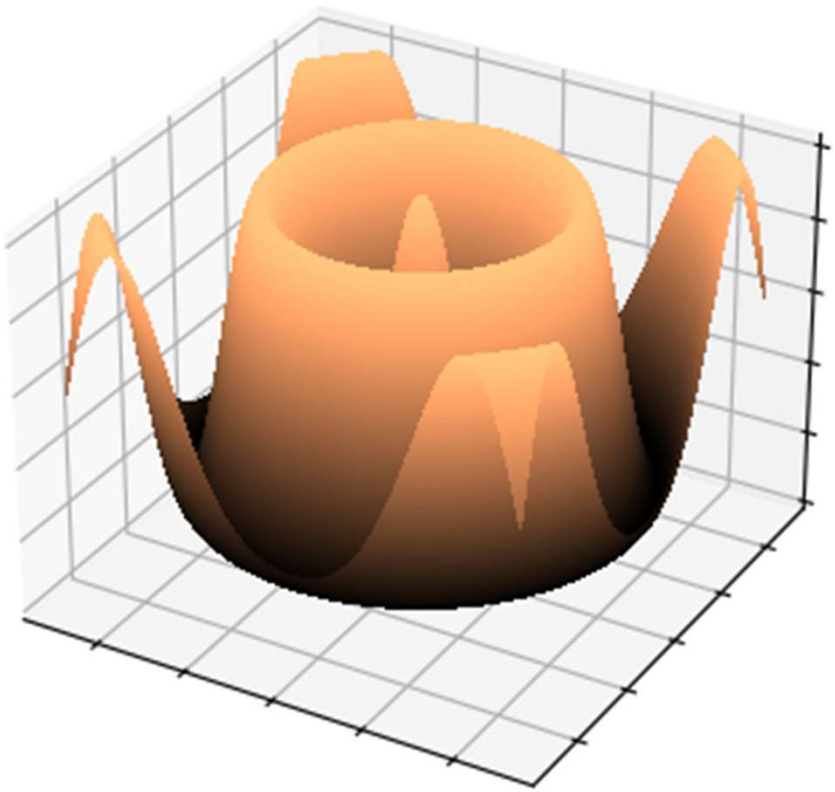


Concave vs. Non-Convex Functions

Concave Function



Non-Convex Function



Searching in Partially Observable and Nondeterministic Environments

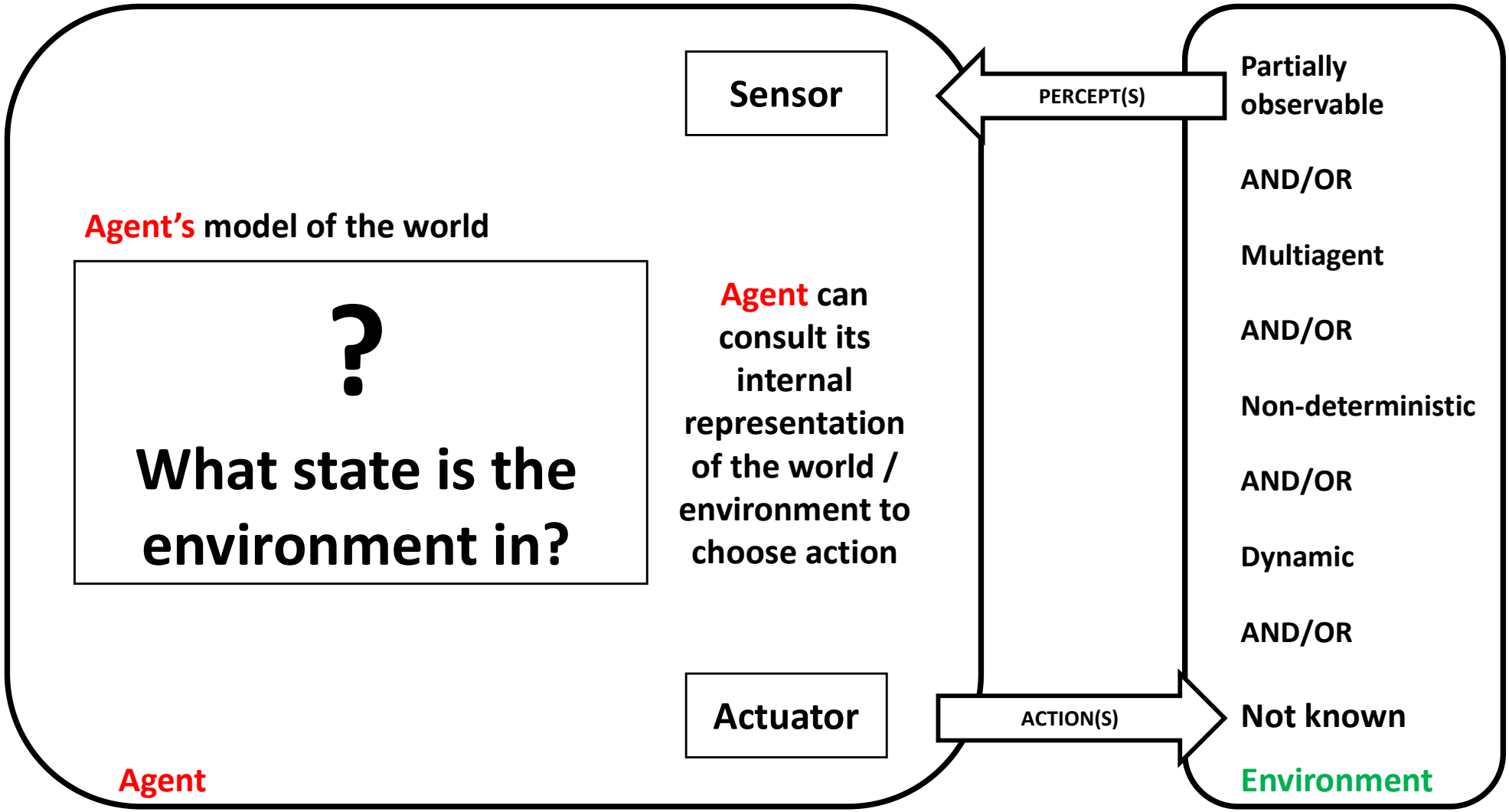
Defining Search Problem

- Define a set of possible states: State Space
- Specify Initial State
- Specify Goal State(s) (there can be multiple)
- Define a FINITE set of possible Actions for EACH state in the State Space
- Come up with a Transition Model which describes what each action does
- Specify the Action Cost Function: a function that gives the cost of applying action *a* in state *s*

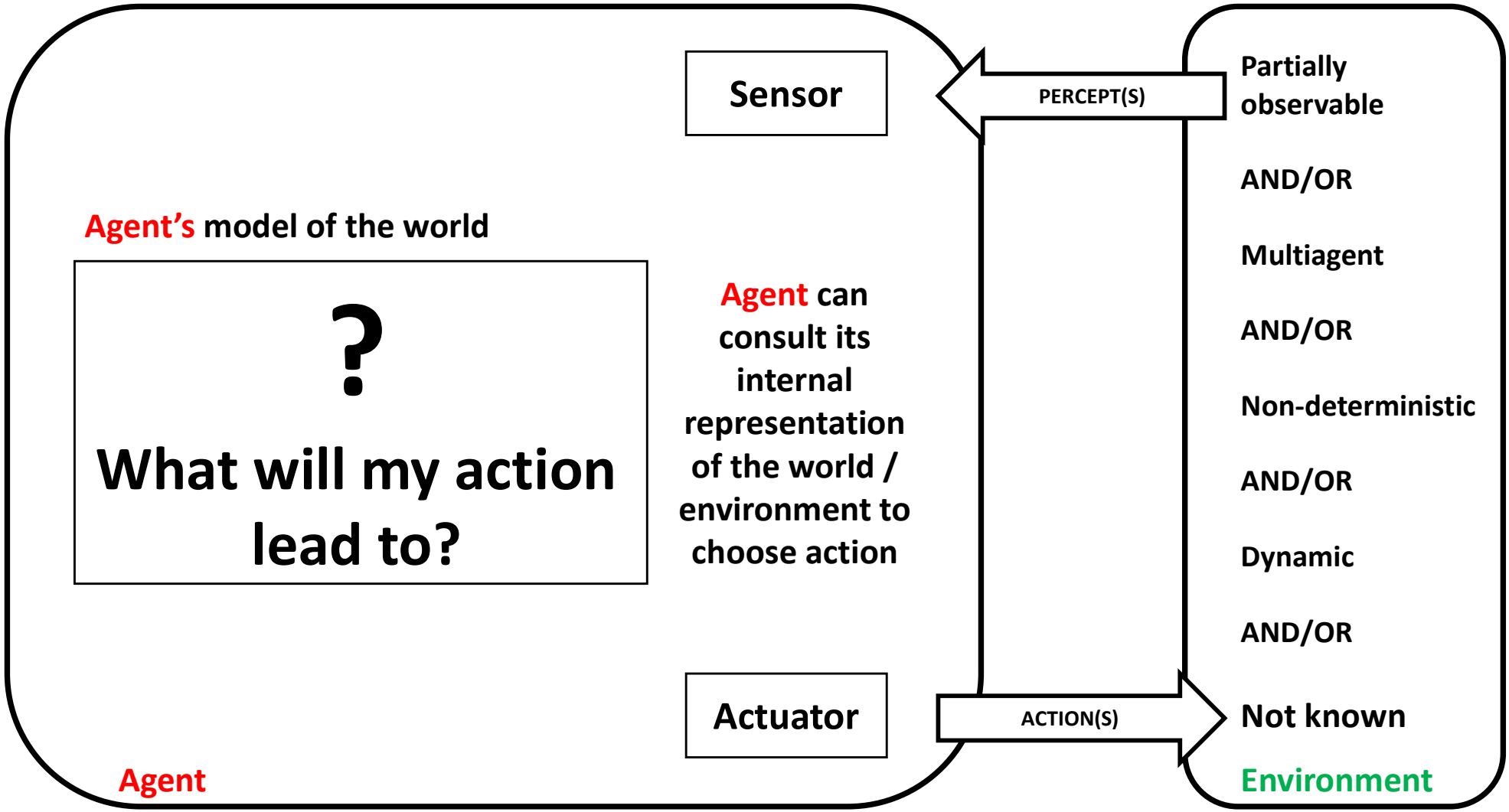
Hardest Case / Problem

- Partially observable (incomplete information, uncertainty)
- Multiagent (complex interactions)
- Nondeterministic (uncertainty)
- Sequential (planning usually necessary)
- Dynamic (changing environment, uncertainty)
- Continuous (infinite number of states)
- Unknown (agent needs to learn / explore, uncertainty)

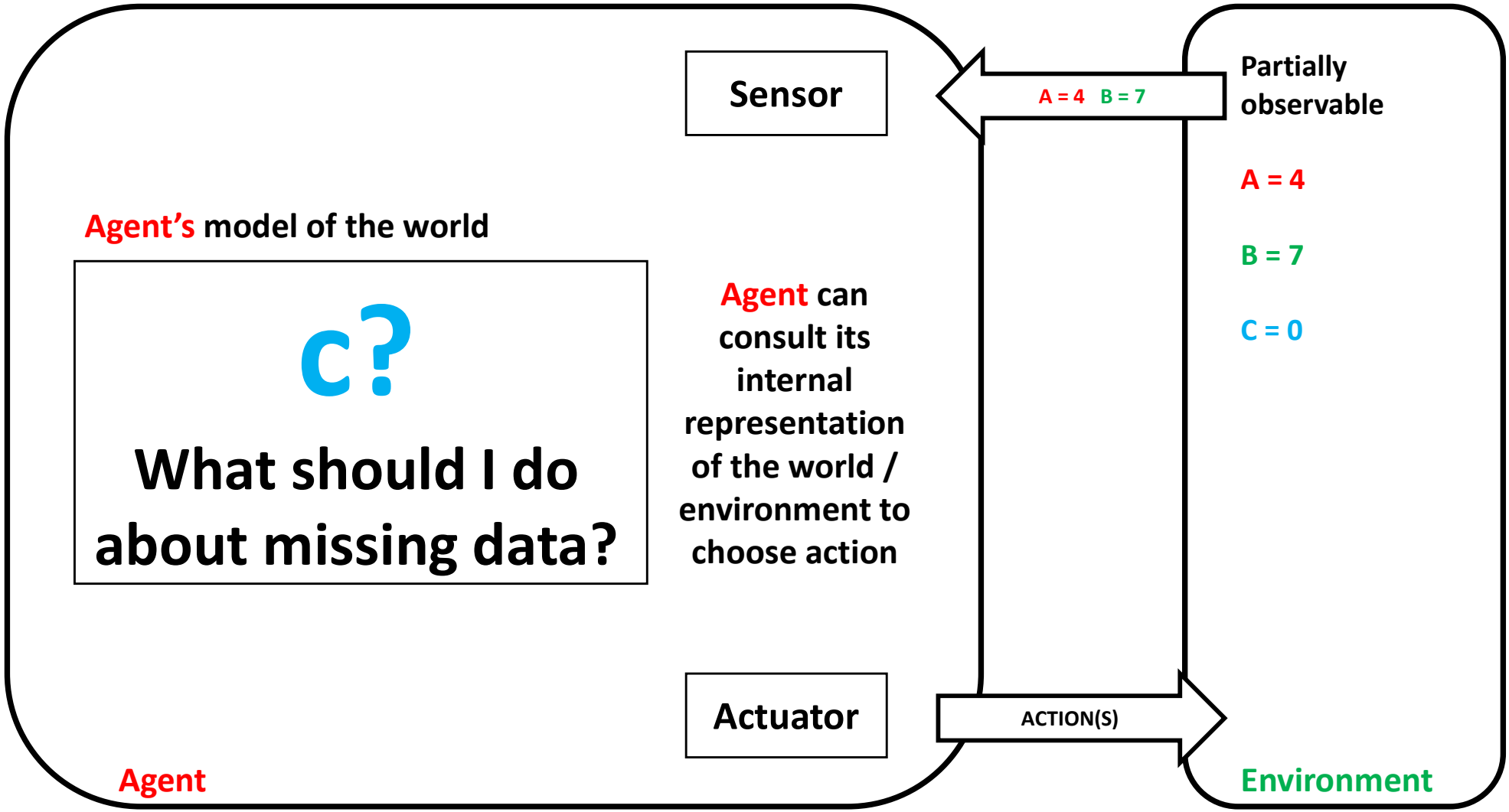
Agents and Uncertainty



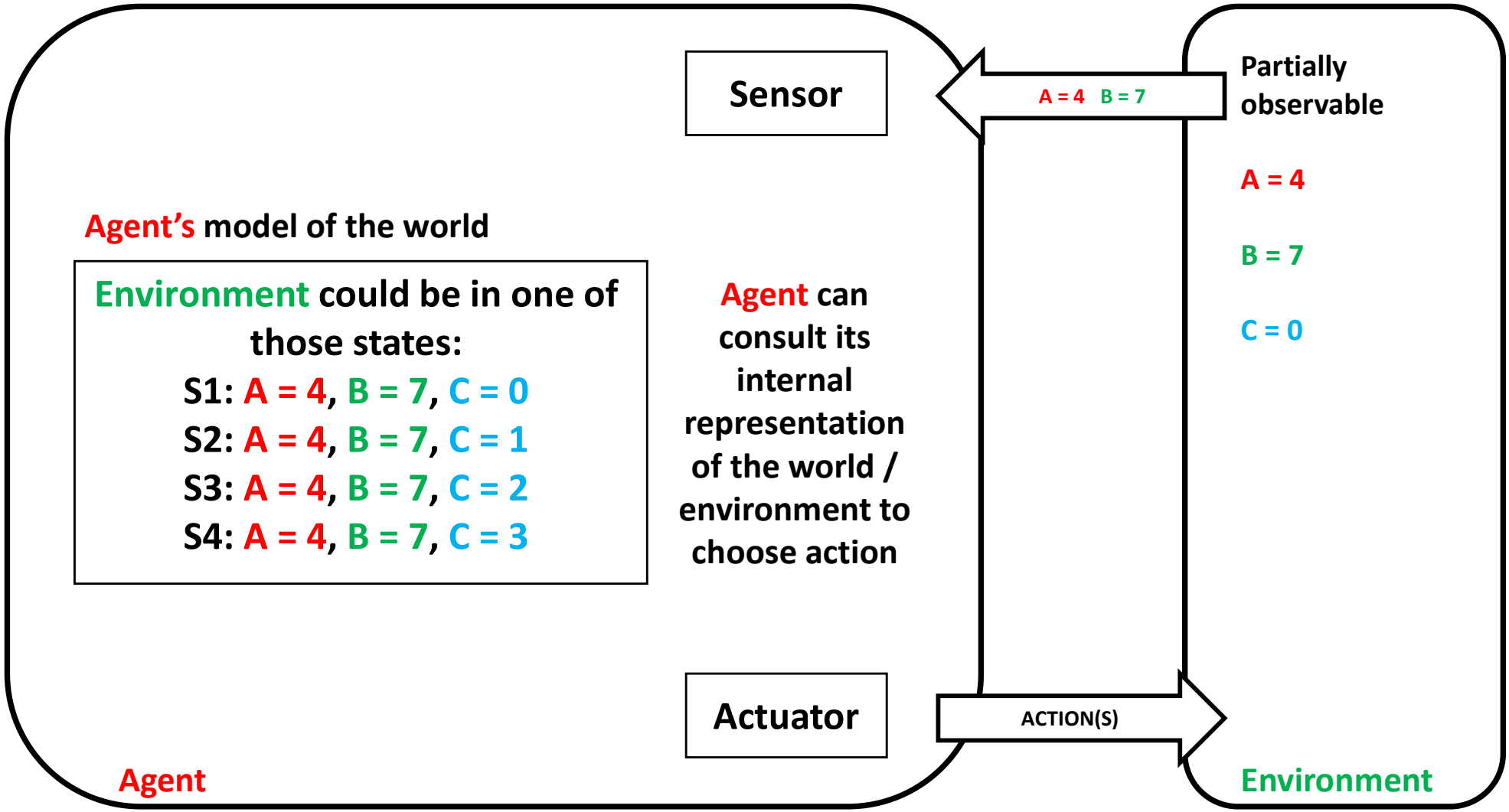
Agents and Uncertainty



Agents and Uncertainty



Agents and Belief State



Assume: $D_C = \{0,1,2,3\}$

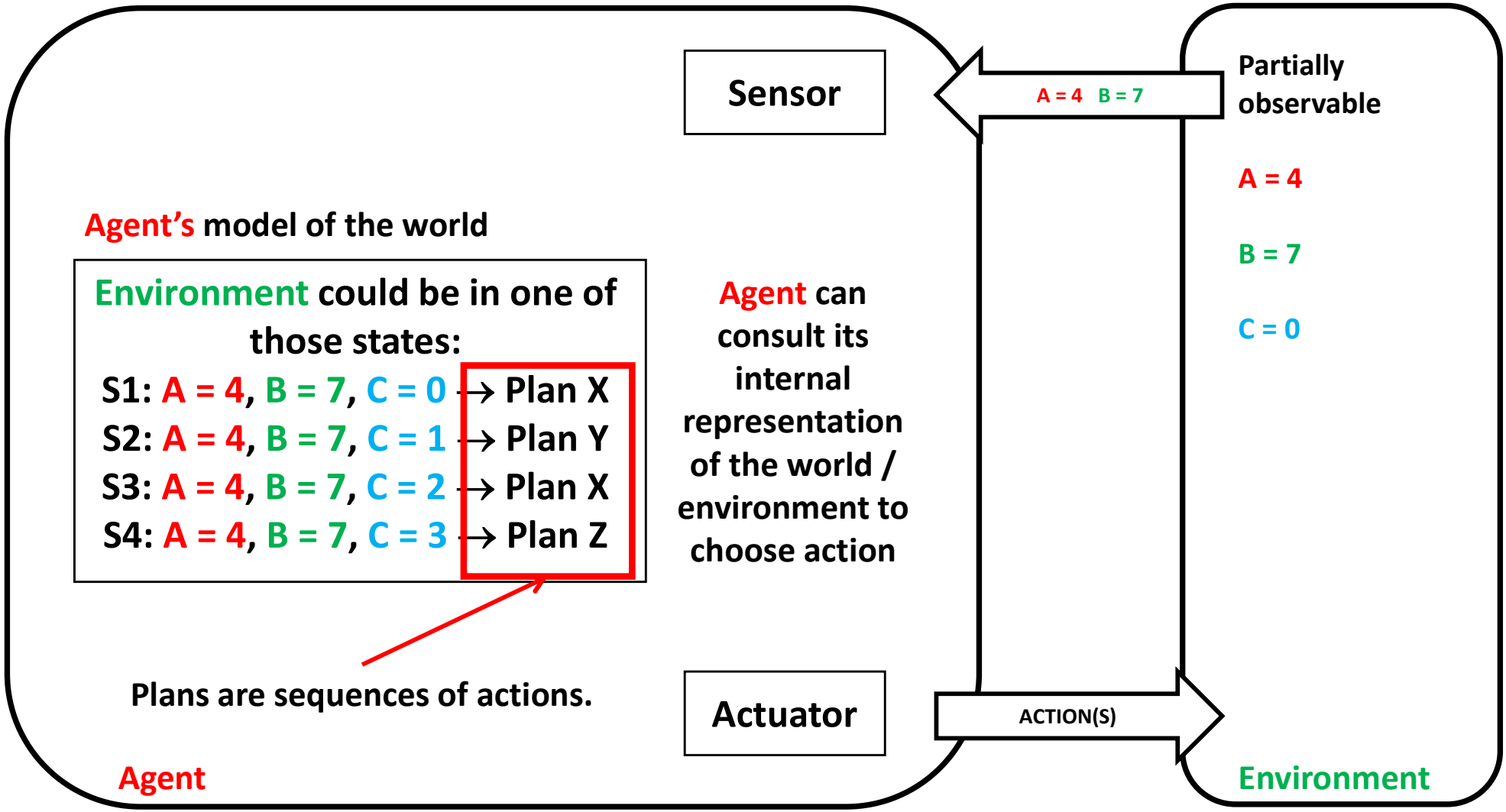
Agent Belief State

Belief state: a set of all possible environment states that the agent can be in and needs to keep track of to handle uncertainty.

Problems:

- **agent needs to consider every possible state some are going to be unlikely**
- **agent needs plans for every eventuality**
- **there may be no known plan, agent needs to act**

Agents and Belief State



Assume: $D_C = \{0,1,2,3\}$

Conditional / Contingency Plan

In partially observable and nondeterministic environments, the solution to a problem is no longer a sequence of actions, but rather a conditional plan (contingency plan also called a strategy).

It specifies what to do depending on agent percepts received while executing a plan.

Defining Search Problem

- Define a set of possible states: State Space
- Specify Initial State
- Specify Goal State(s) (there can be multiple)
- Define a FINITE set of possible Actions for EACH state in the State Space
- Come up with a Transition Model which describes what each action does ← WE NEED TO REDEFINE THAT!
- Specify the Action Cost Function: a function that gives the cost of applying action *a* in state *s*

UPDATED Transition Model

RESULT(CURRENT_STATE, ACTION) = NEXT_STATE

Becomes

RESULTS(CURRENT_STATE, ACTION) =
{NEXT_STATE1, NEXT_STATE2, …, NEXT_STATEN}

UPDATED Solution

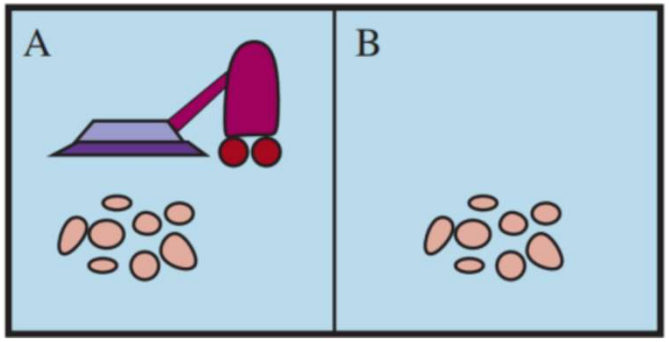
ACTION1, ACTION2, …., ACTIONN

Becomes

[ACTION1, IF STATE=STATEi THEN [ACTION2, ACTION3] ELSE [], … IF…..]

and so on

Vacuum Cleaner Agent Example



Percept sequence	Action
$[A, Clean]$	$Right$
$[A, Dirty]$	$Suck$
$[B, Clean]$	$Left$
$[B, Dirty]$	$Suck$
$[A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Dirty]$	$Suck$
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	$Right$
$[A, Clean], [A, Clean], [A, Dirty]$	$Suck$
⋮	⋮

Vacuum Cleaner Agent Example

function TABLE-DRIVEN-AGENT(*percept*) **returns** an action
  persistent: *percepts*, a sequence, initially empty
    table, a table of actions, indexed by percept sequences, initially fully specified

  append *percept* to the end of *percepts*
  action ← LOOKUP(*percepts*, *table*)
  return *action*

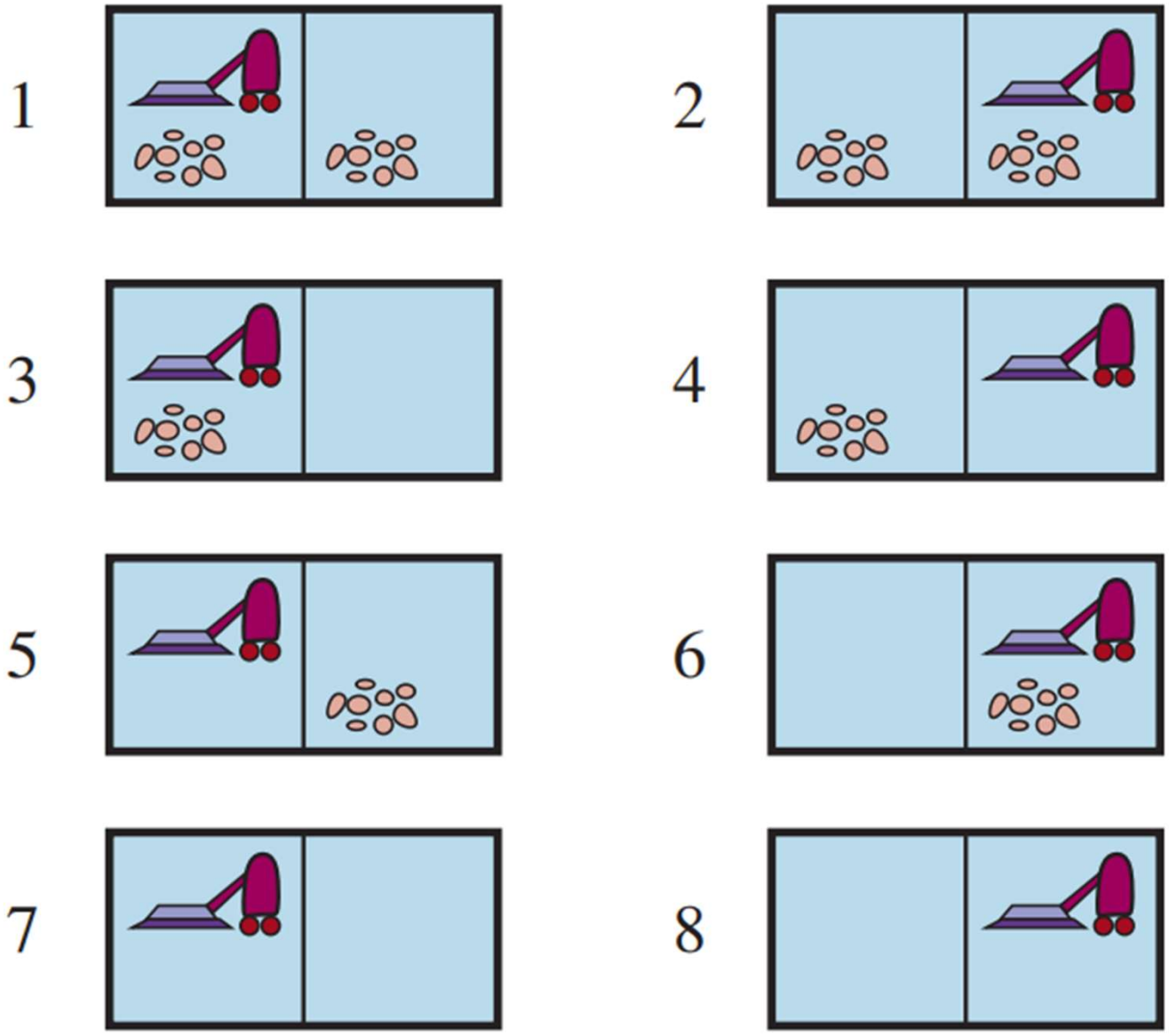
function REFLEX-VACUUM-AGENT([*location*, *status*]) **returns** an action

  if *status* = *Dirty* **then return** *Suck*
  else if *location* = *A* **then return** *Right*
  else if *location* = *B* **then return** *Left*

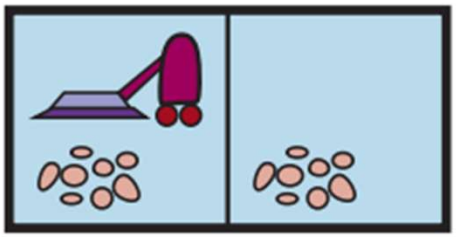
Vacuum Cleaner State Space

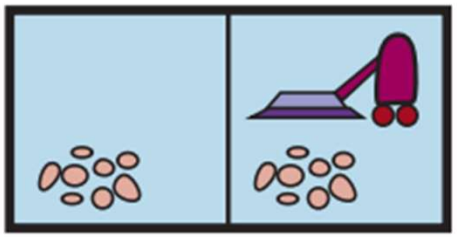
Possible actions:

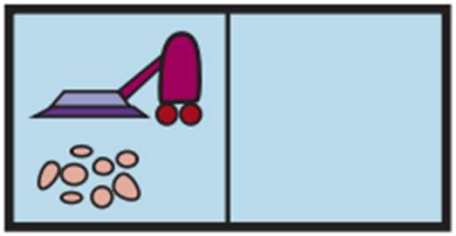
- **SUCK**
- **LEFT**
- **RIGHT**

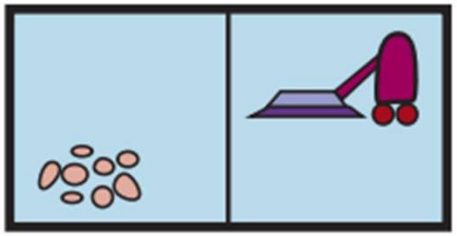


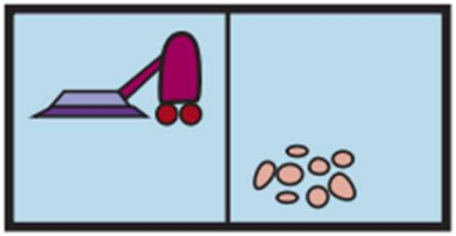
Vacuum Cleaner State Space

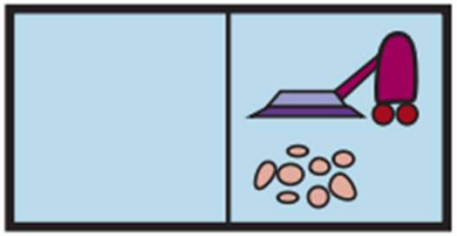
1 

2 

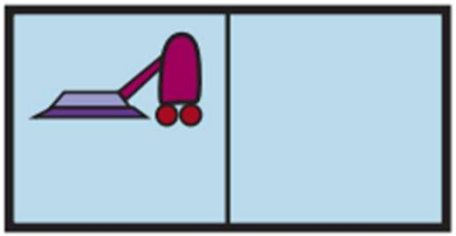
3 

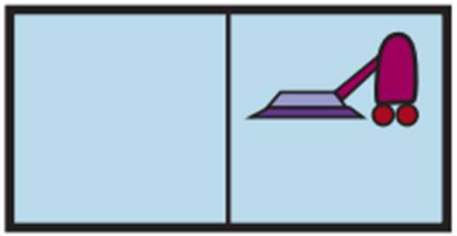
4 

5 

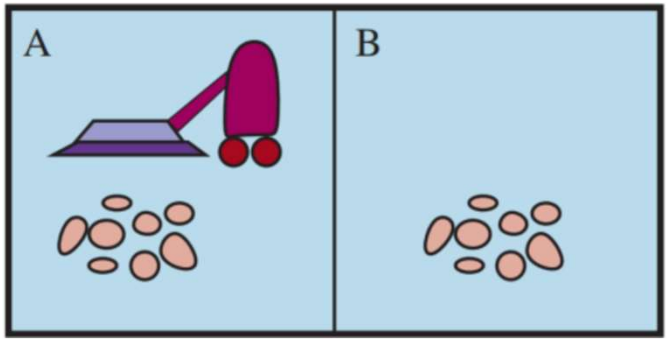
6 

GOAL STATES

7 

8 

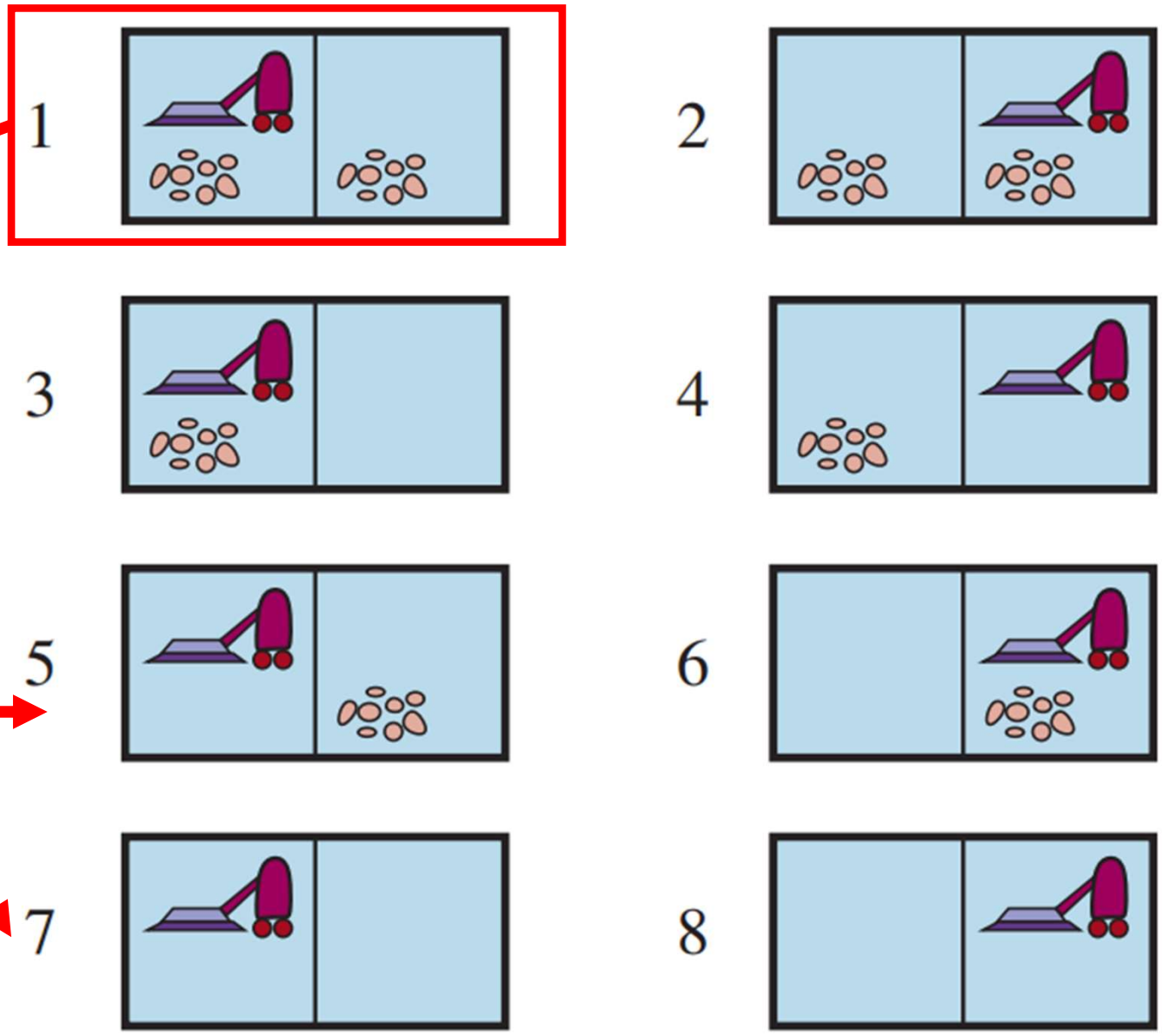
Erratic Vacuum Cleaner World



Consider a nondeterministic ("erratic") variant of that vacuum cleaner world, where action SUCK:

- **applied on a dirty square cleans that square and sometimes cleans dirt on the adjacent square**
- **applied to a clean square can sometimes end up depositing dirt on it**

Erratic Vacuum Cleaner State Space



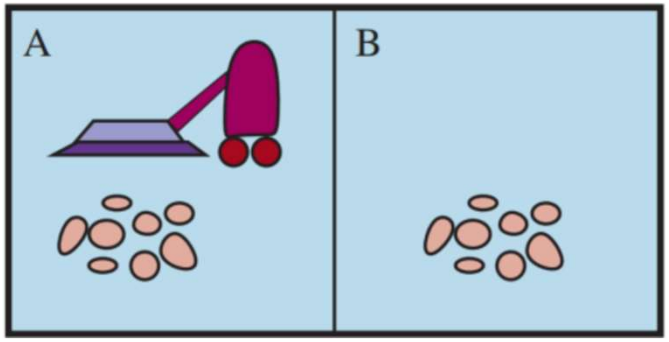
Possible action:

- **SUCK**

Possible: outcomes:

- **State 5**
- **State 7**

Erratic Vacuum Cleaner World



RESULT(1, SUCK) = 5

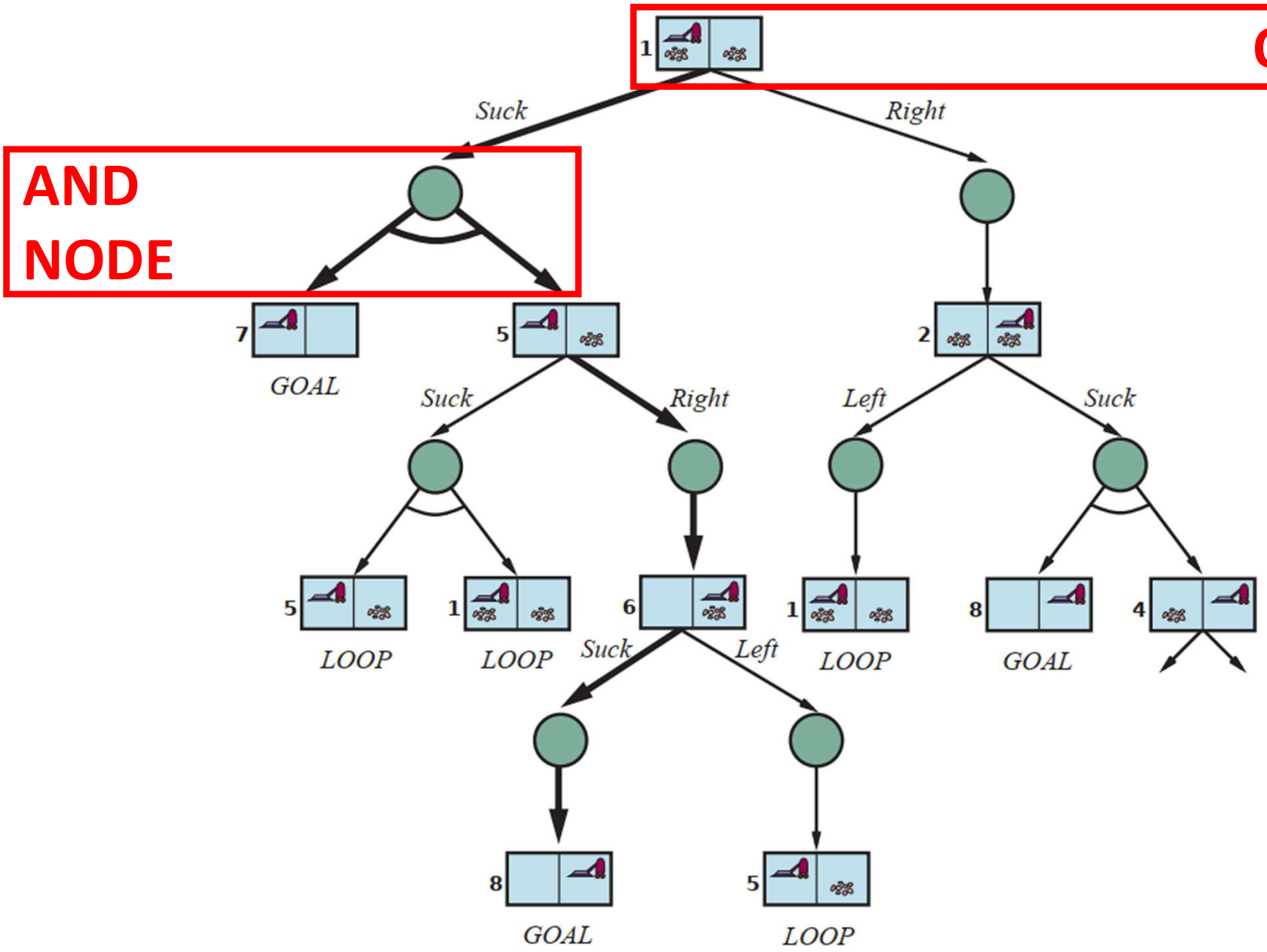
Becomes

RESULTS(1, SUCK) = {5,7}

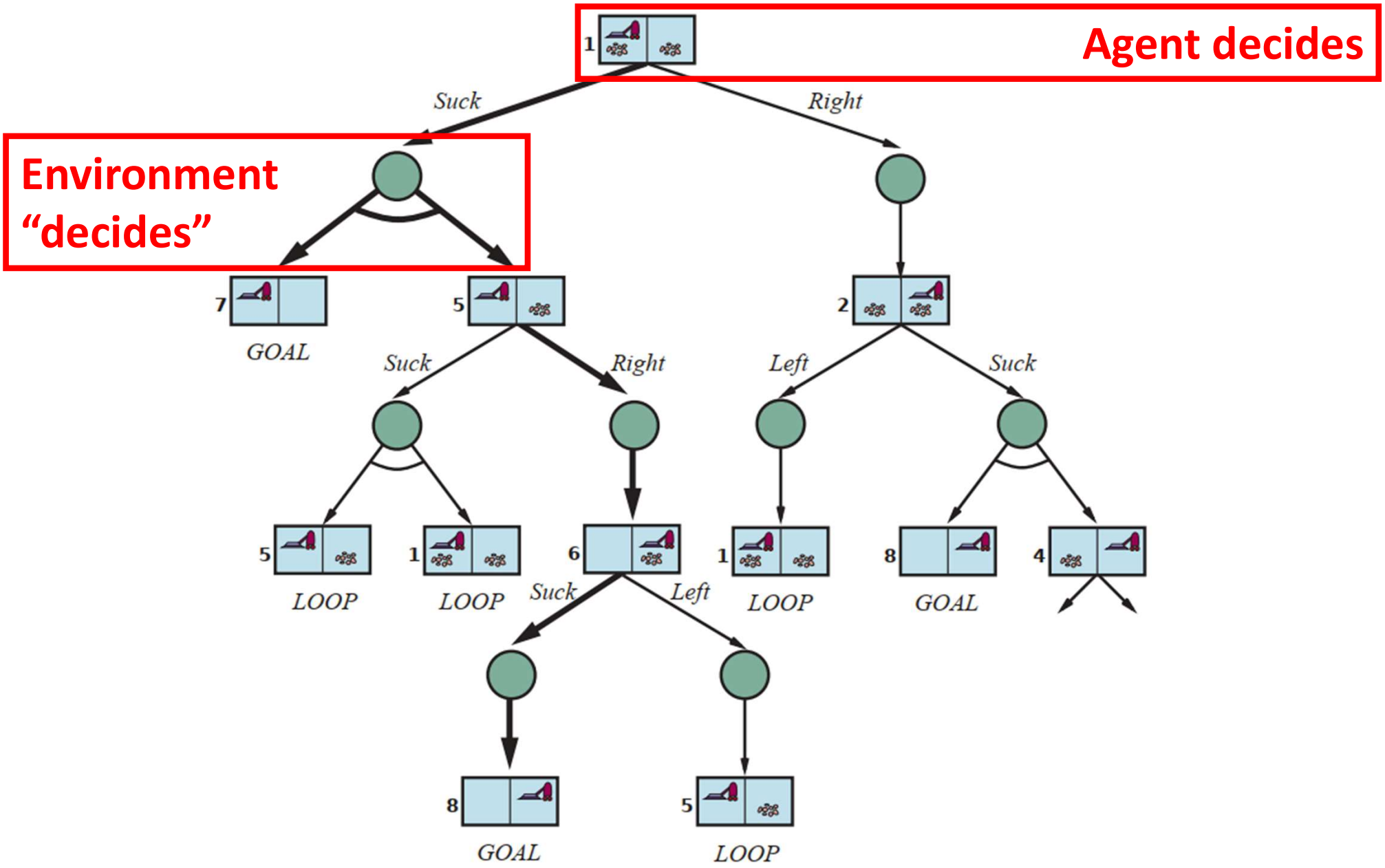
AND-OR Search: Pseudocode

```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
    return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
    if problem.IS-GOAL(state) then return the empty plan
    if IS-CYCLE(path) then return failure
    for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
        if plan ≠ failure then return [action] + plan]
    return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
    for each s_i in states do
        plan_i ← OR-SEARCH(problem, s_i, path)
        if plan_i = failure then return failure
    return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

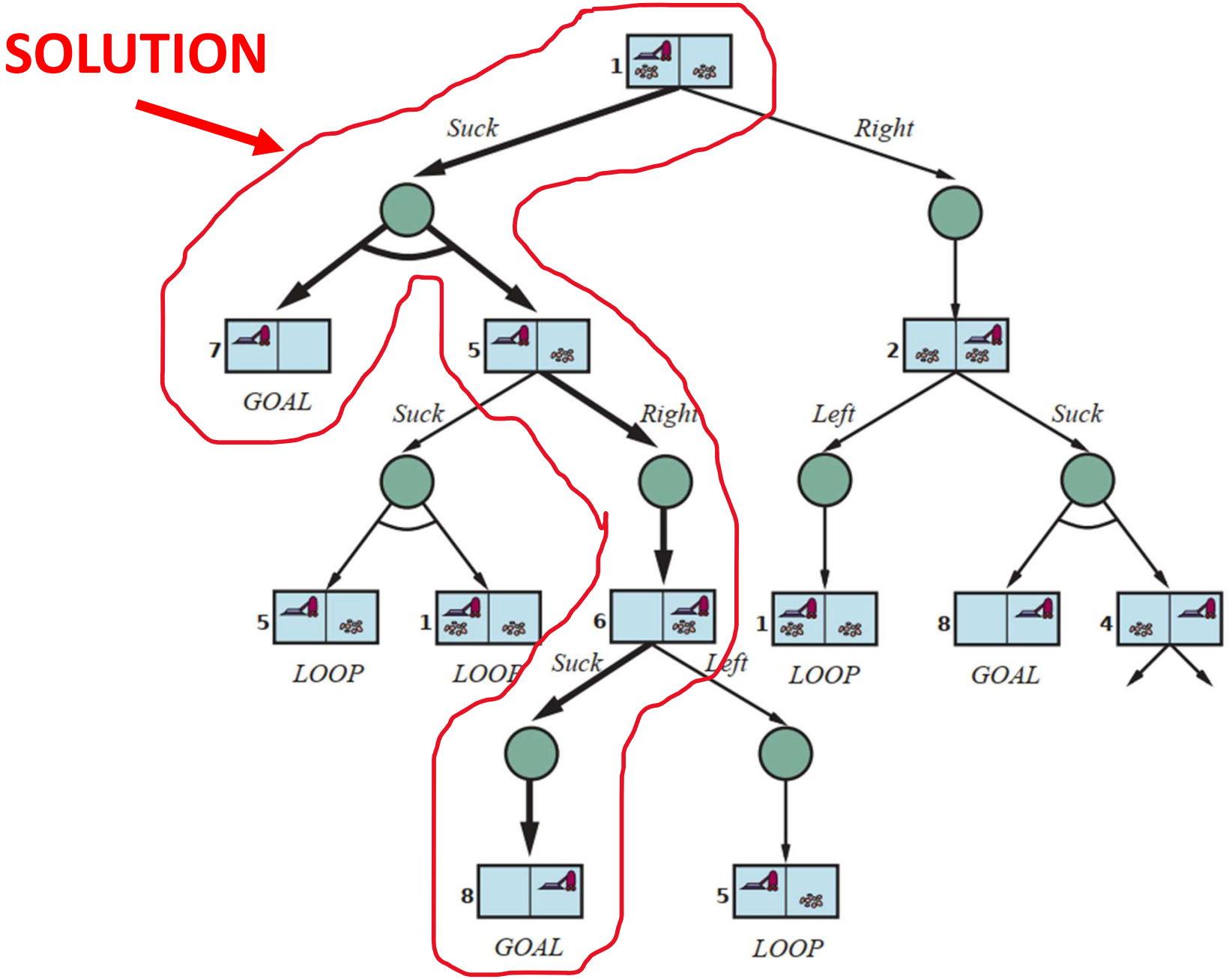
Erratic Vacuum World AND-OR Tree



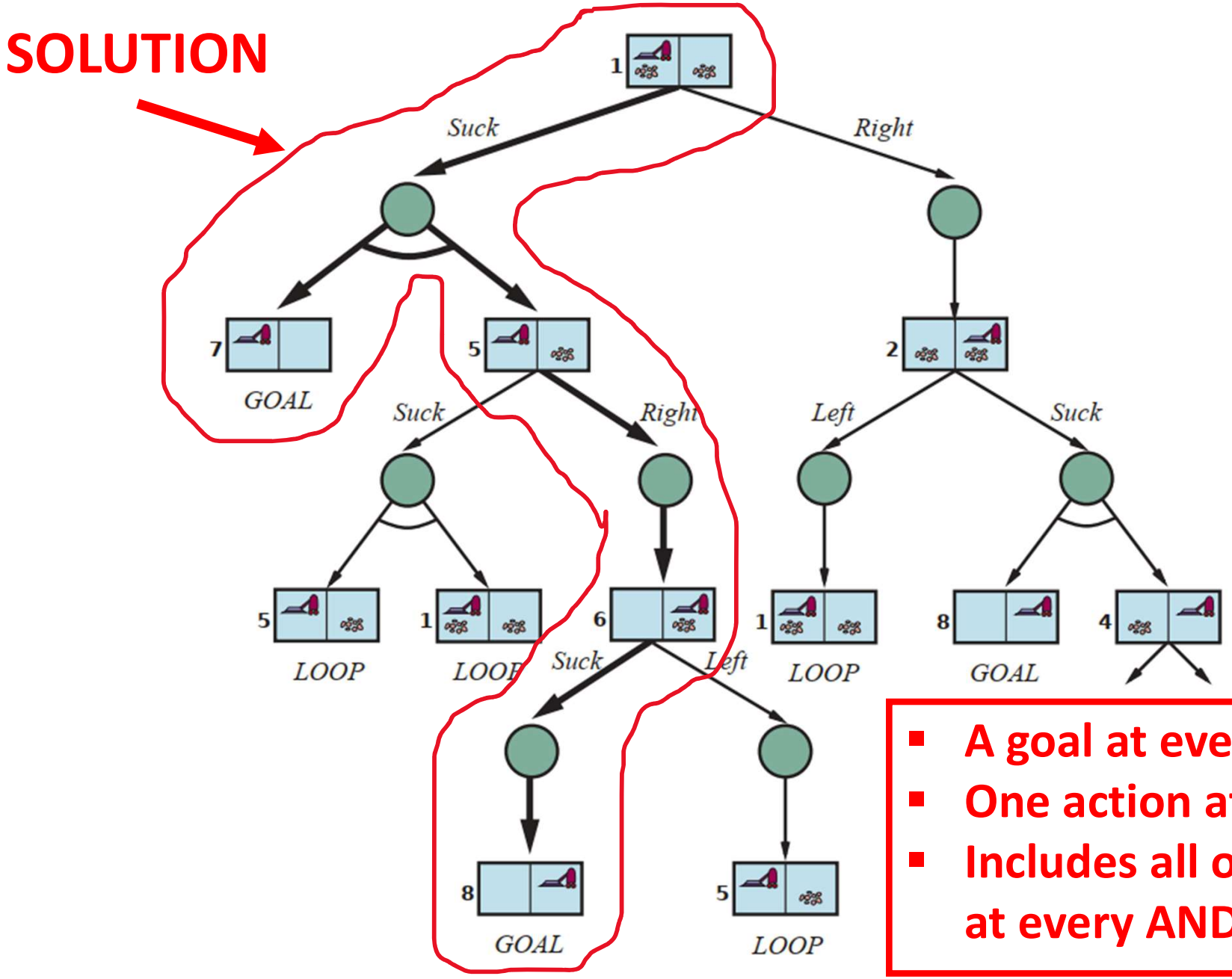
Erratic Vacuum World AND-OR Tree



Erratic Vacuum World AND-OR Tree



Erratic Vacuum World AND-OR Tree



- A goal at every leaf
- One action at every OR
- Includes all outcomes at every AND

AND-OR Search: Pseudocode

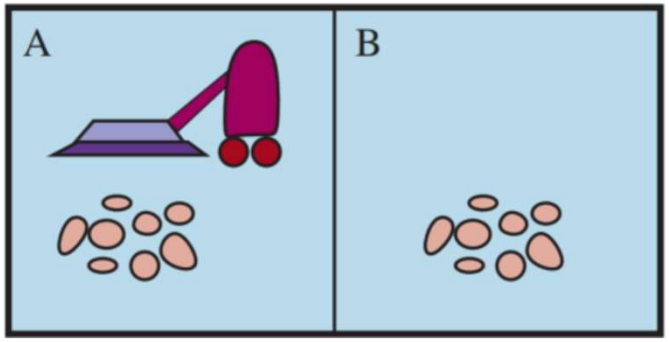
```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
    return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
    if problem.IS-GOAL(state) then return the empty plan
    if IS-CYCLE(path) then return failure
    for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
        if plan ≠ failure then return [action] + plan]
    return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
    for each s_i in states do
        plan_i ← OR-SEARCH(problem, s_i, path)
        if plan_i = failure then return failure
    return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Depth first search / recursive algorithm

AND-OR Search: Pseudocode

```
function AND-OR-SEARCH(problem) returns a conditional plan, or failure
    return OR-SEARCH(problem, problem.INITIAL, [])

function OR-SEARCH(problem, state, path) returns a conditional plan, or failure
    if problem.IS-GOAL(state) then return the empty plan
    if IS-CYCLE(path) then return failure   ← CYCLE FOUND!
    for each action in problem.ACTIONS(state) do
        plan ← AND-SEARCH(problem, RESULTS(state, action), [state] + path])
        if plan ≠ failure then return [action] + plan]
    return failure

function AND-SEARCH(problem, states, path) returns a conditional plan, or failure
    for each s_i in states do
        plan_i ← OR-SEARCH(problem, s_i, path)
        if plan_i = failure then return failure
    return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

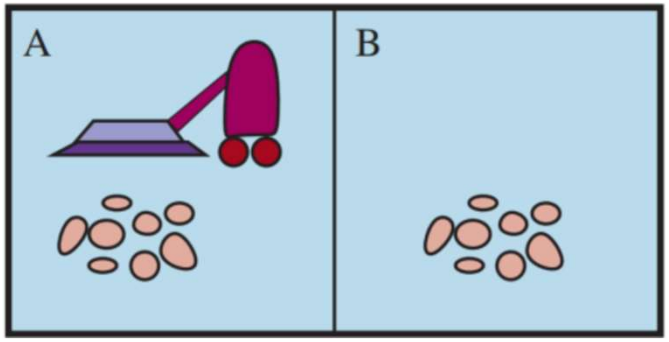
Slippery Erratic Vacuum Cleaner World



Consider a nondeterministic ("erratic") variant of that vacuum cleaner world,

- **where action SUCK:**
  - **applied on a dirty square cleans that square and sometimes cleans dirt on the adjacent square**
  - **applied to a clean square can sometimes end up depositing dirt on it**
- **where action LEFT or RIGHT:**
  - **Sometimes leaves the vacuum cleaner in the same location**

Slippery Erratic Vacuum Cleaner World

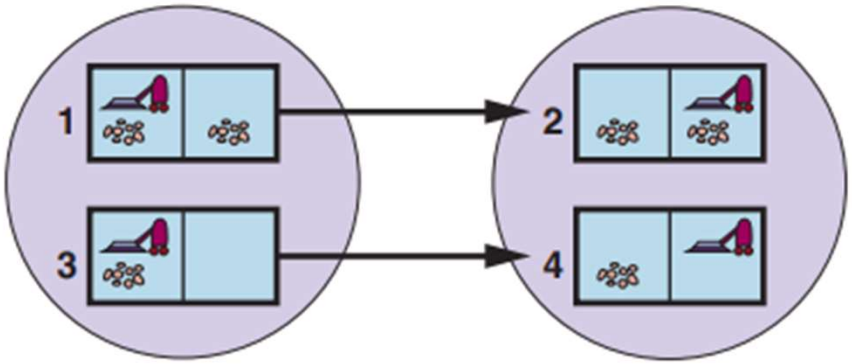


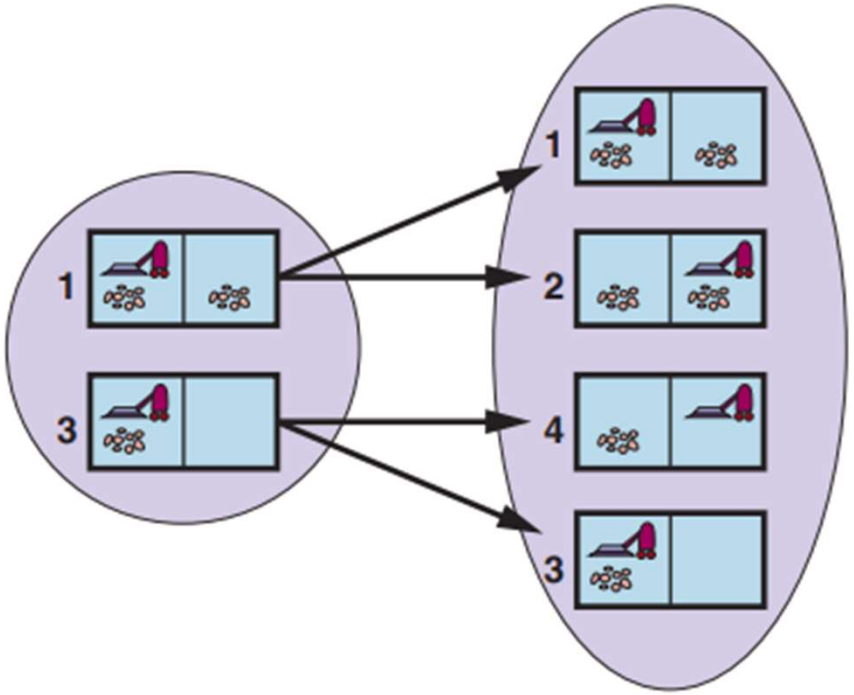
Sometimes the problem naturally calls for cycles ("try again"): a cyclic solution exists, but not acyclic

Things to consider:

- **is it a temporary glitch → it will help**
- **is something permanently broken → it will not help**

Slippery Erratic World: BELIEF State

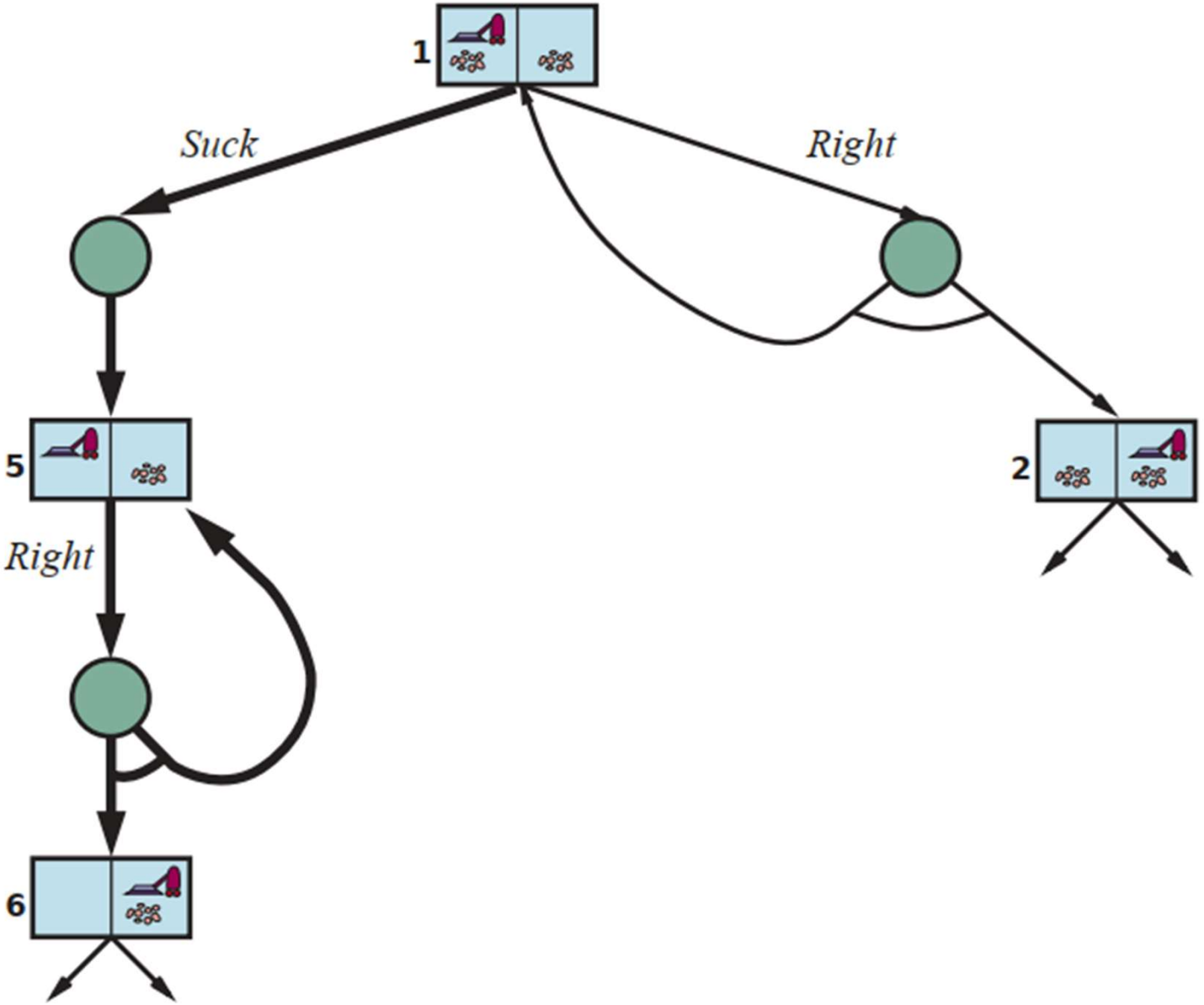




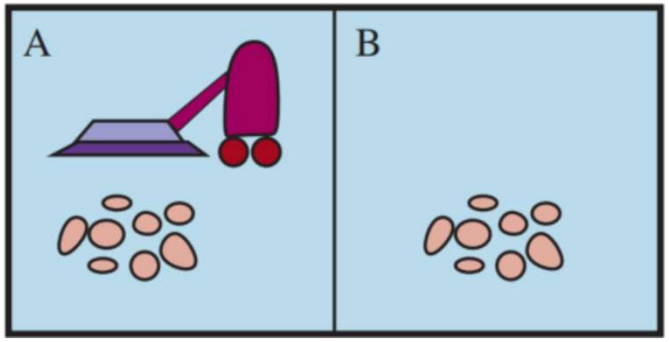
Deterministic World

“Slippery” World

Slippery Erratic World: Cyclic Plans



Slippery Erratic Vacuum Cleaner World



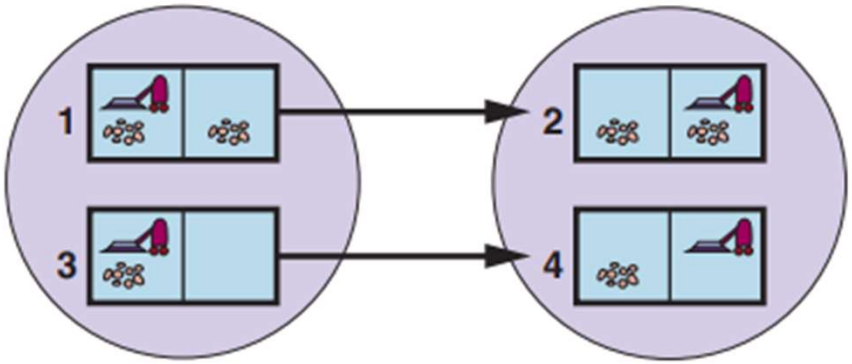
Solution: introduce a while construct

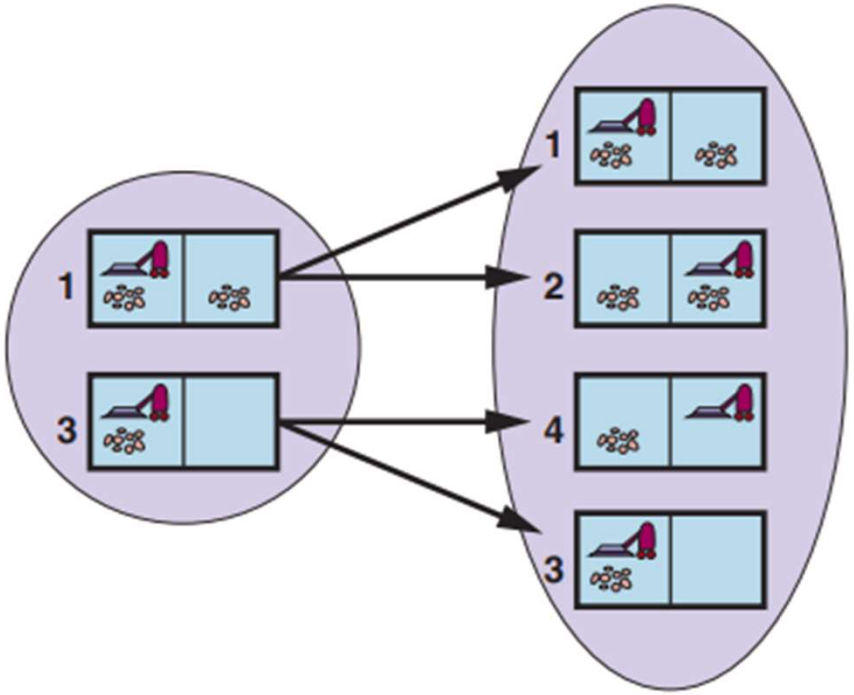
[SUCK, while State=5 do RIGHT, SUCK]

Or add a label to a portion of the plan

[SUCK, L1: RIGHT if State=5 then L1 else SUCK]

Predicting Next BELIEF State

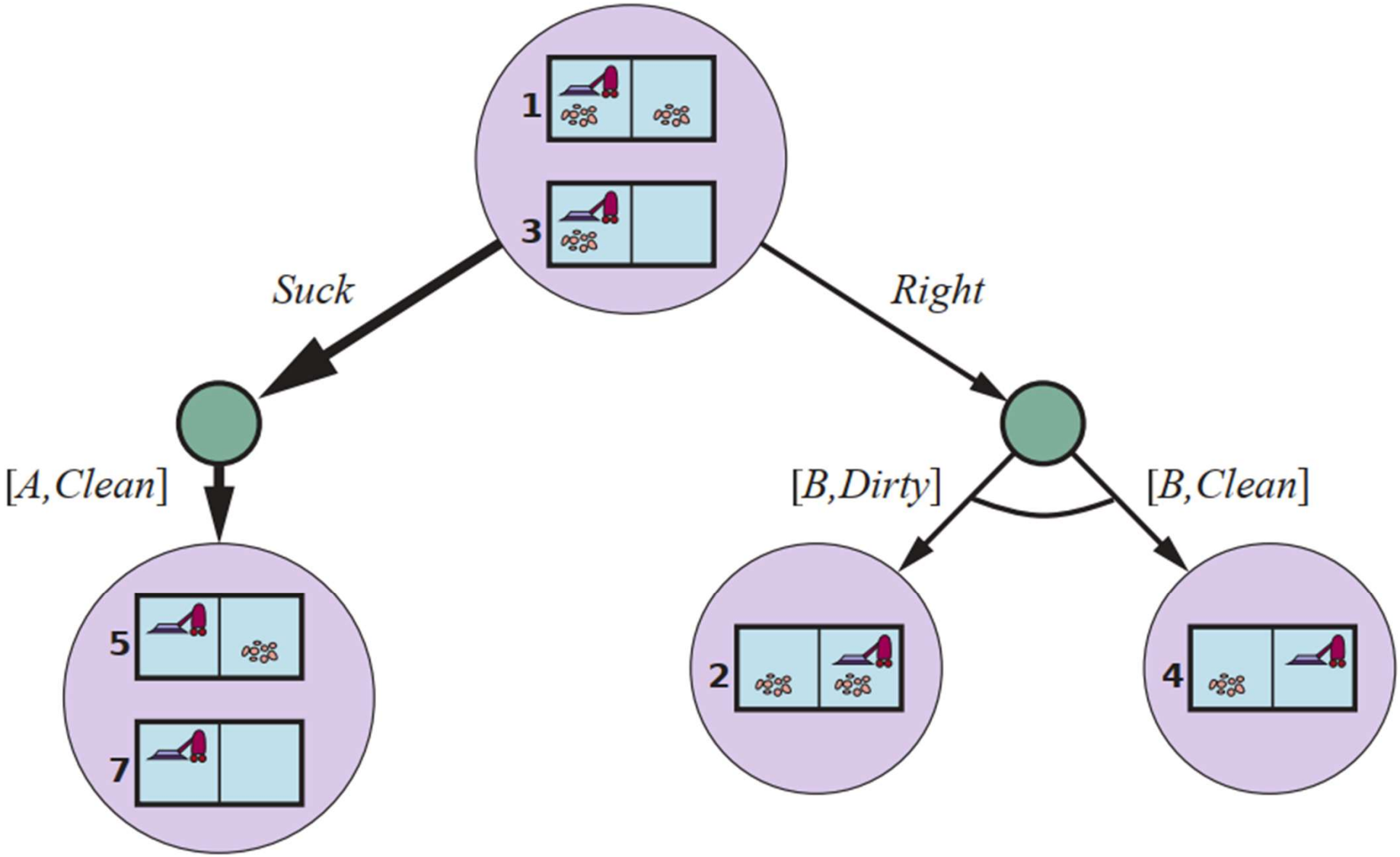




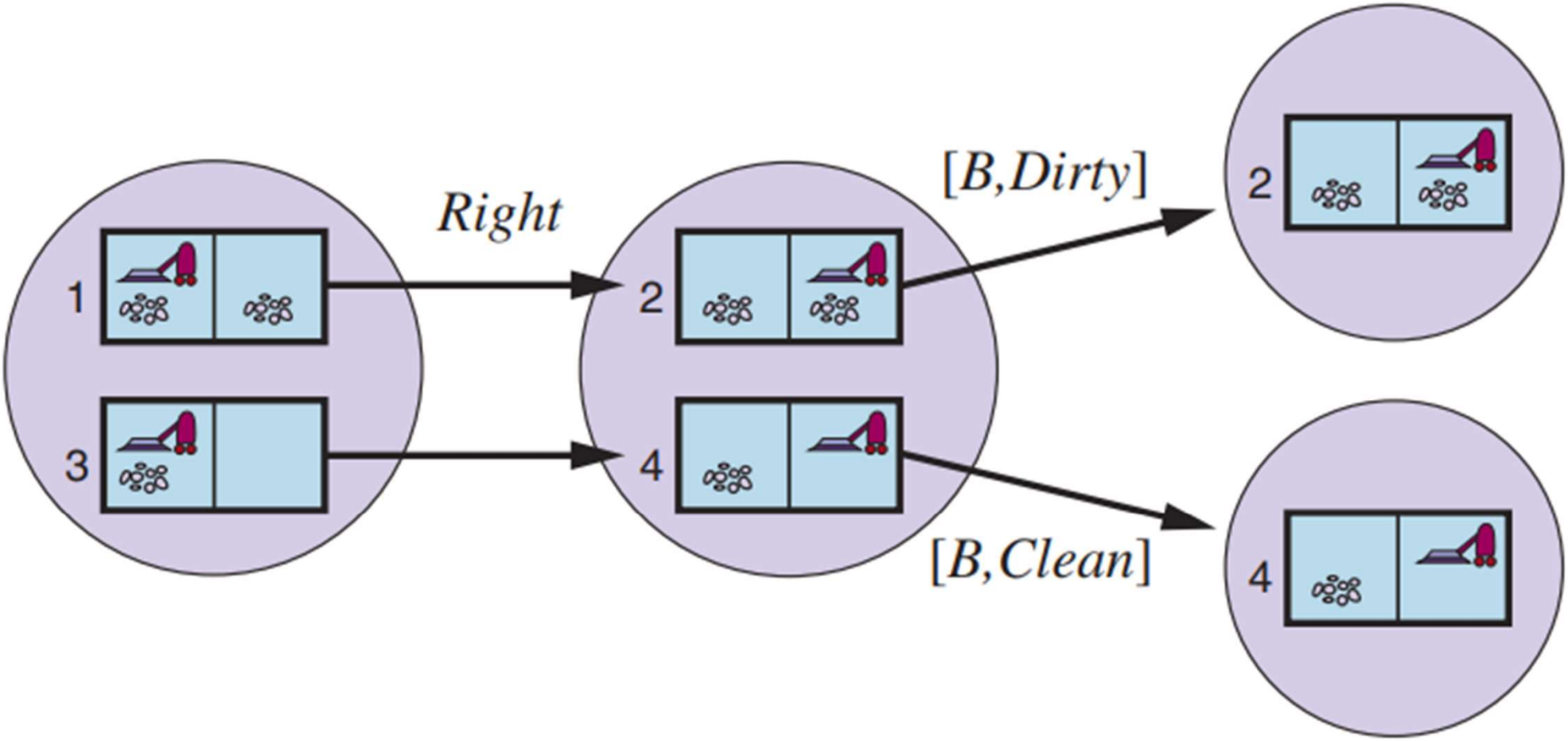
Deterministic World

“Slippery” World

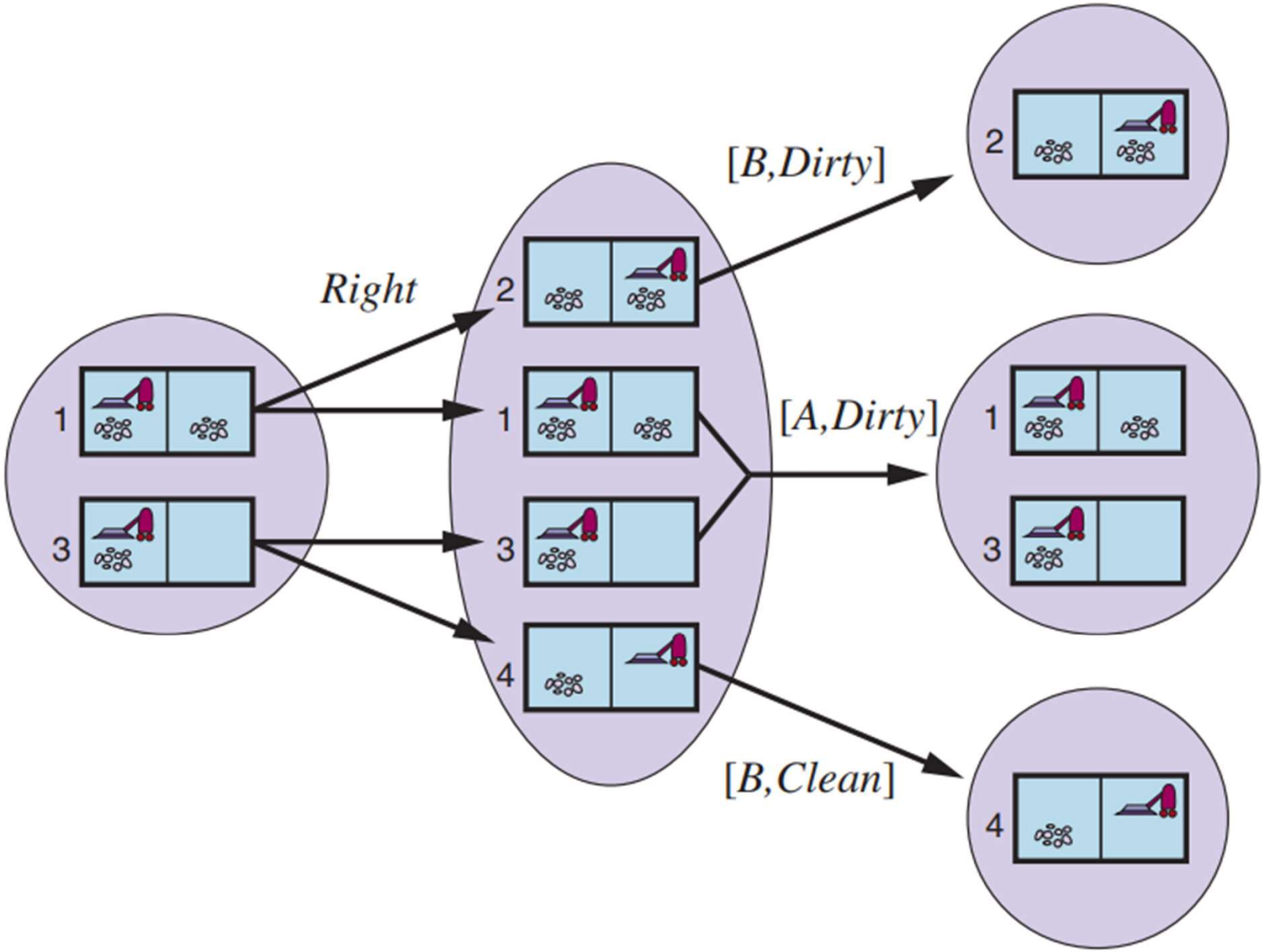
Local Sensing / AND-OR Tree



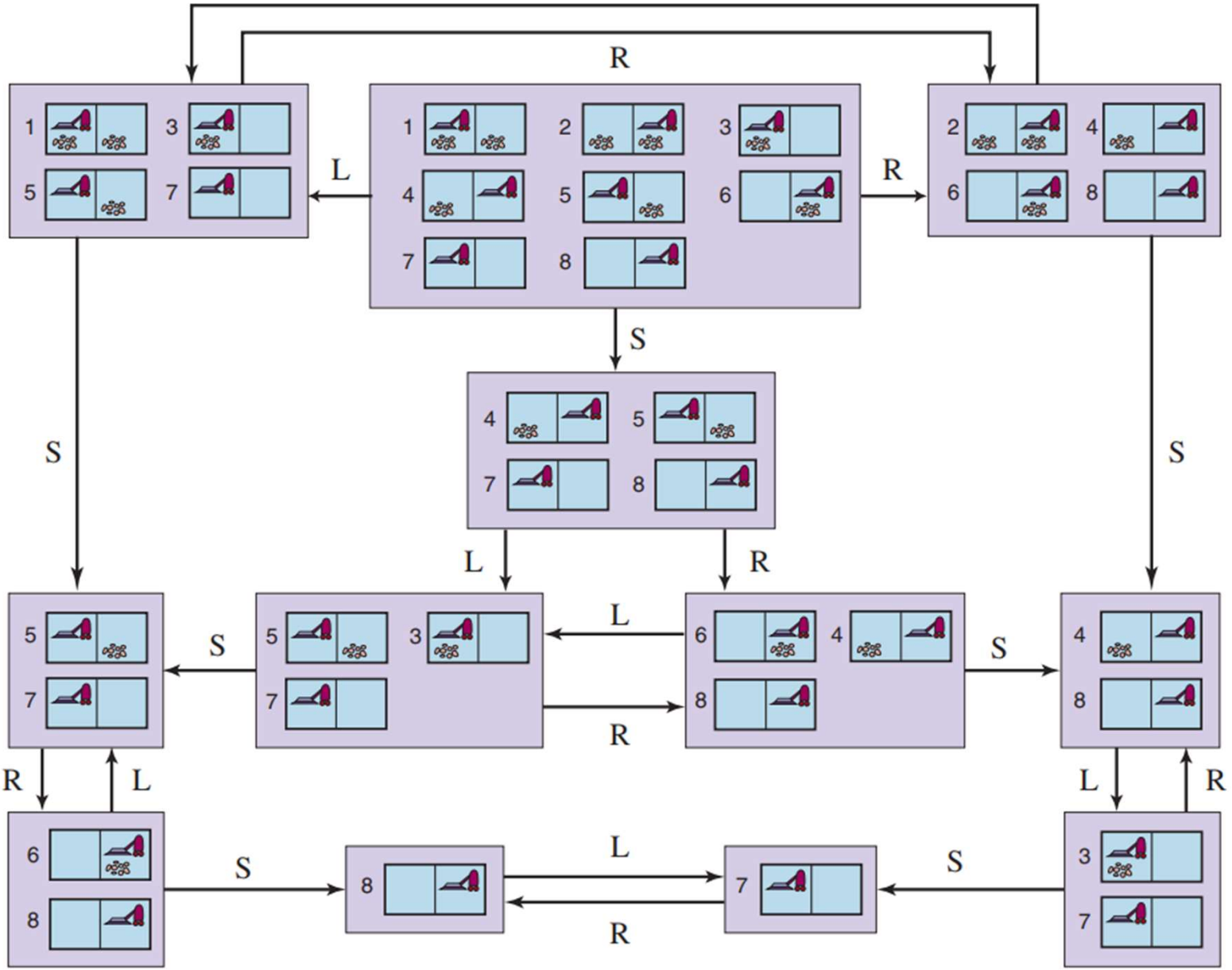
Deterministic World: Transitions



“Slippery” World: Transitions



Reachable Belief-State Space



Reinforcement Learning

Main Machine Learning Categories

Supervised learning	Unsupervised learning	Reinforcement learning
Supervised learning is one of the most common techniques in machine learning. It is based on **known relationship(s) and patterns within data** (for example: relationship between inputs and outputs). Frequently used types: **regression**, and **classification**.	**Unsupervised learning** involves finding underlying patterns within data. Typically used in **clustering** data points (similar customers, etc.)	Reinforcement learning is inspired by behavioral psychology. It is **based on a rewarding / punishing an algorithm**. Rewards and punishments are based on algorithm's action within its environment.

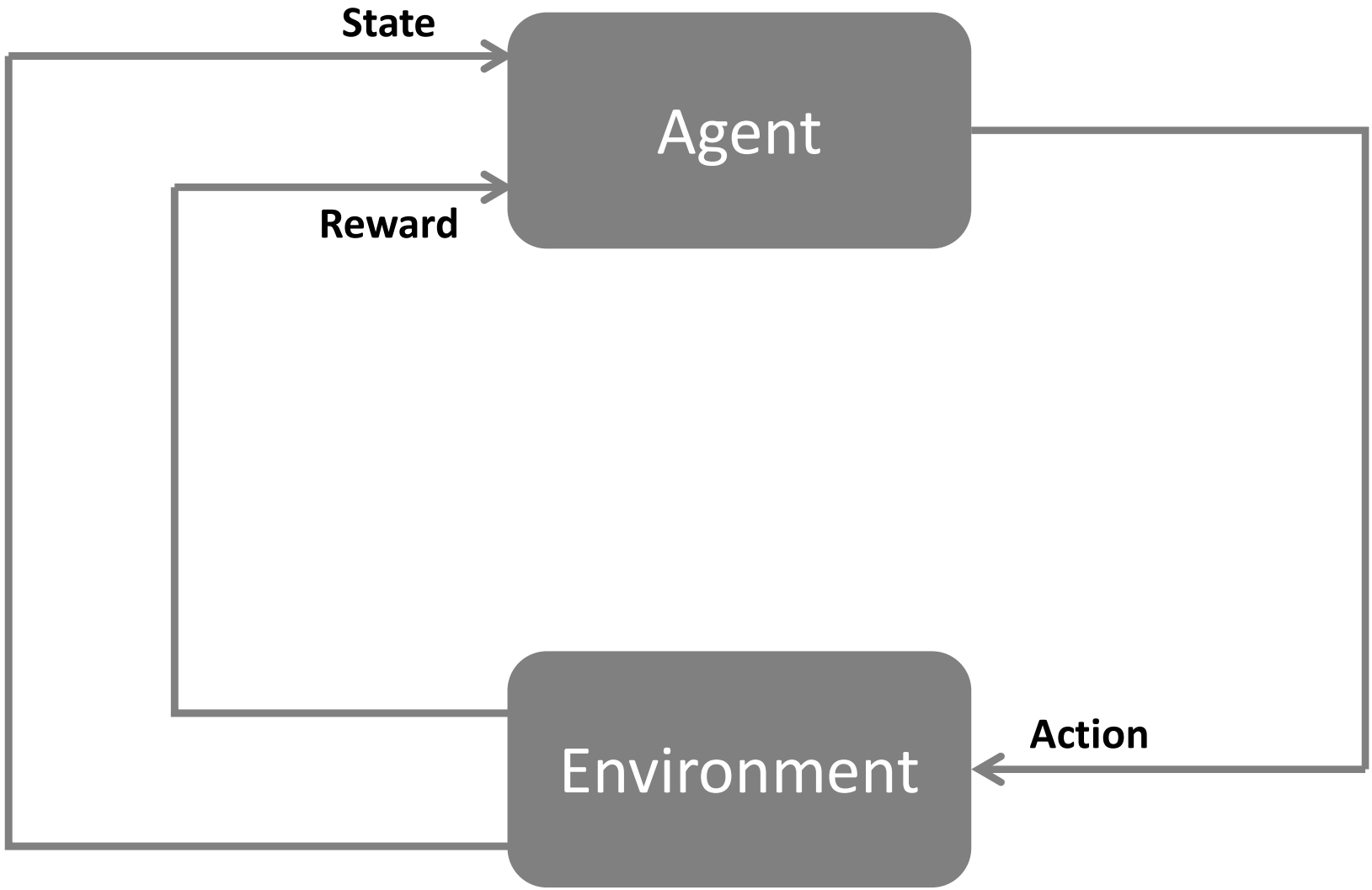
What is Reinforcement Learning?

Idea:

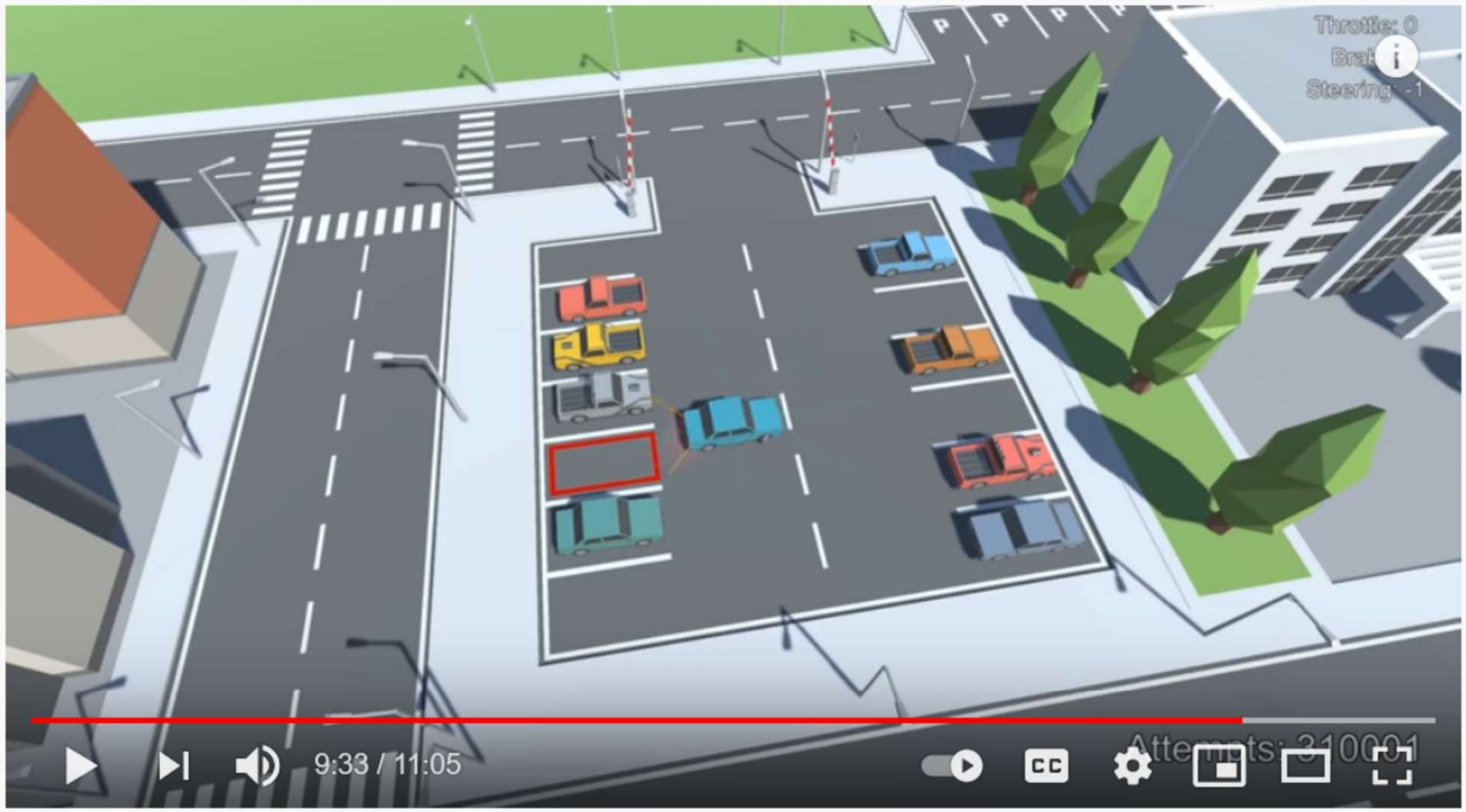
Reinforcement learning is inspired by behavioral psychology. It is **based on a rewarding / punishing an algorithm**.

Rewards and punishments are based on algorithm's action within its environment.

RL: Agents and Environments



Reinforcement Learning in Action

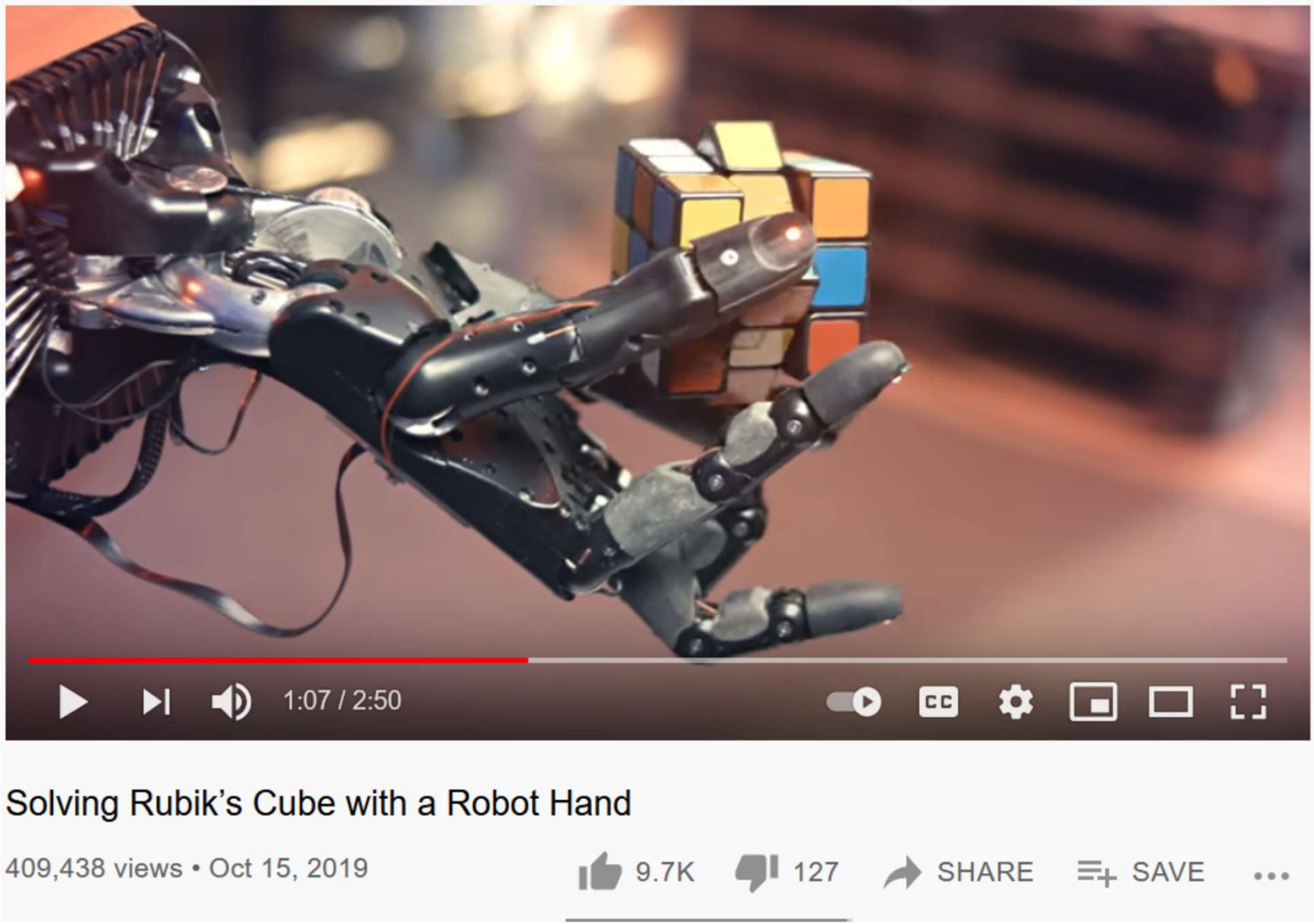


#ArtificialIntelligence #MachineLearning #ReinforcementLearning

AI Learns to Park - Deep Reinforcement Learning

1,744,342 views • Aug 23, 2019 28K 1.1K SHARE SAVE

Reinforcement Learning in Action

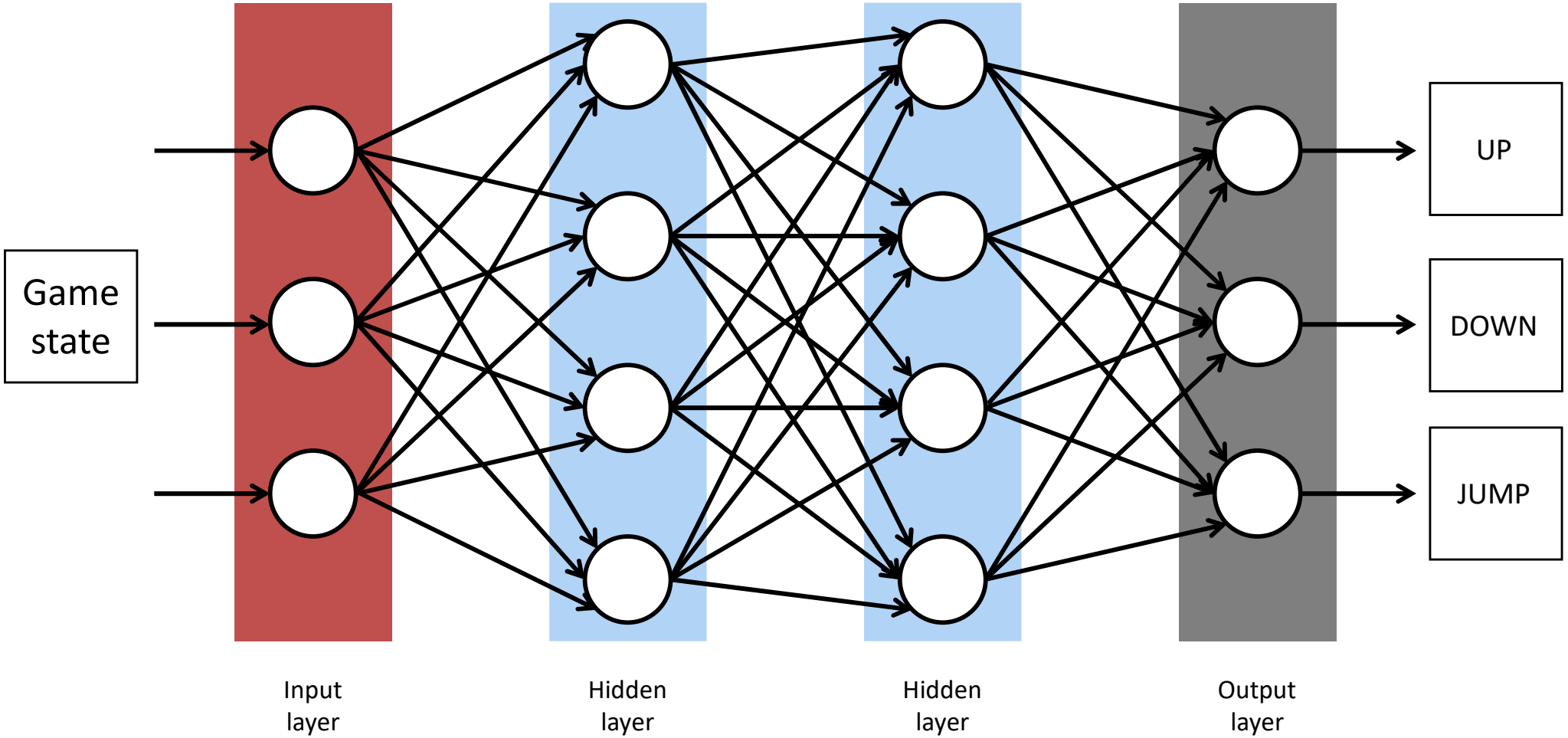


Source: https://www.youtube.com/watch?v=x4O8pojMF0w

Reinforcement Learning in Action

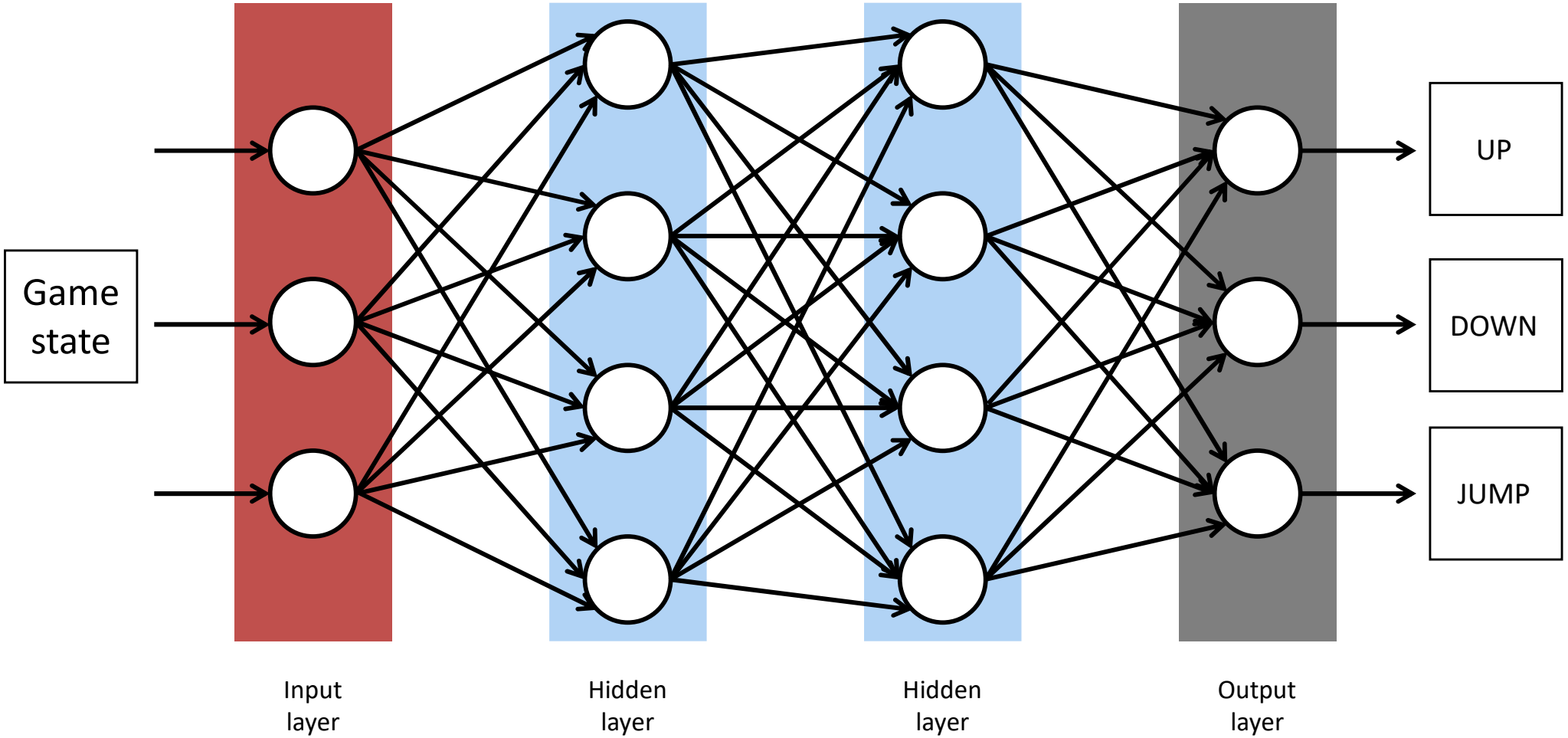
Source: https://www.youtube.com/watch?v=kopoLzvh5jY

ANN for Simple Game Playing



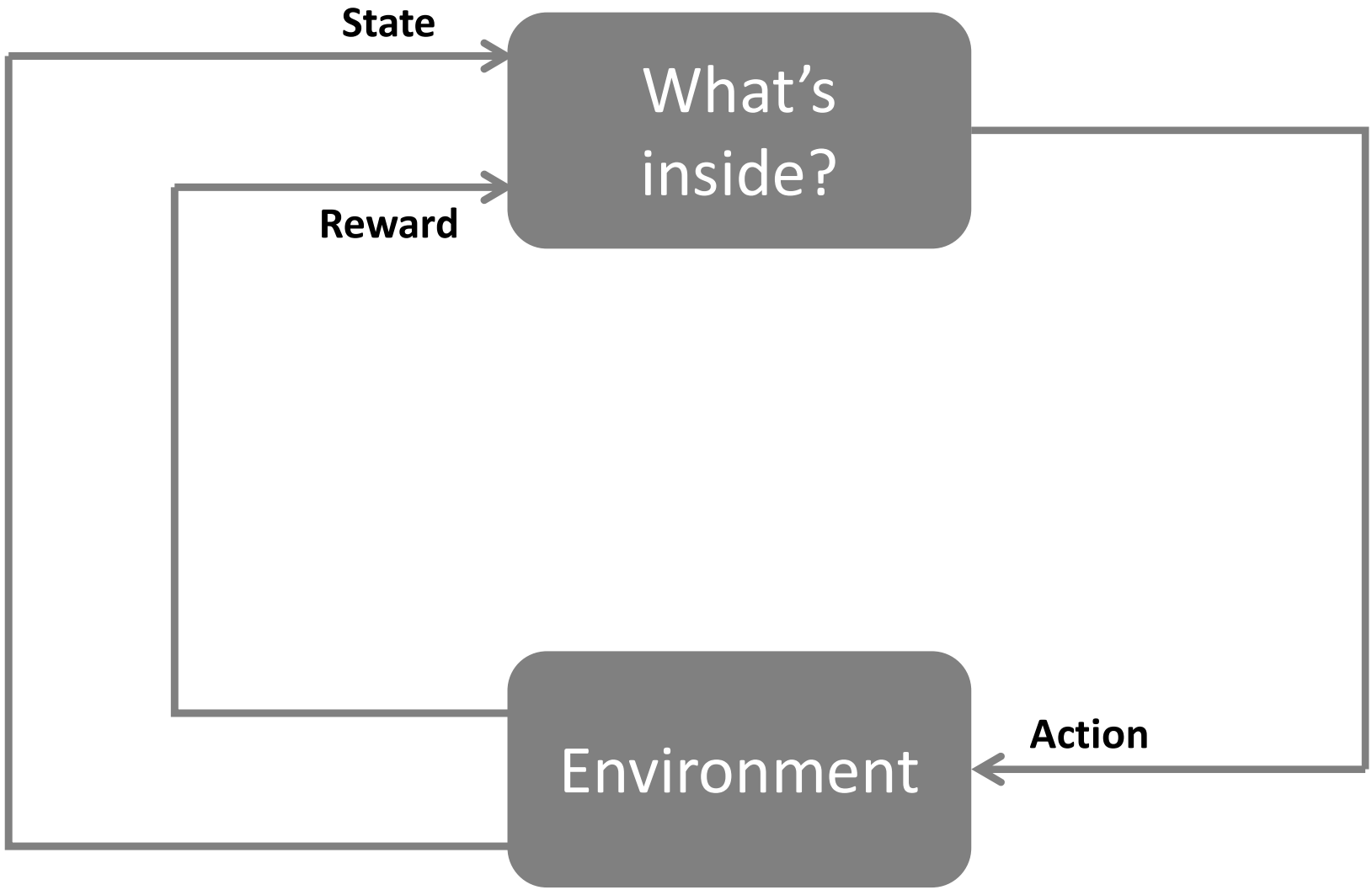
ANN for Simple Game Playing

Current game is an input. Decisions (UP/DOWN/JUMP) are rewarded/punished.

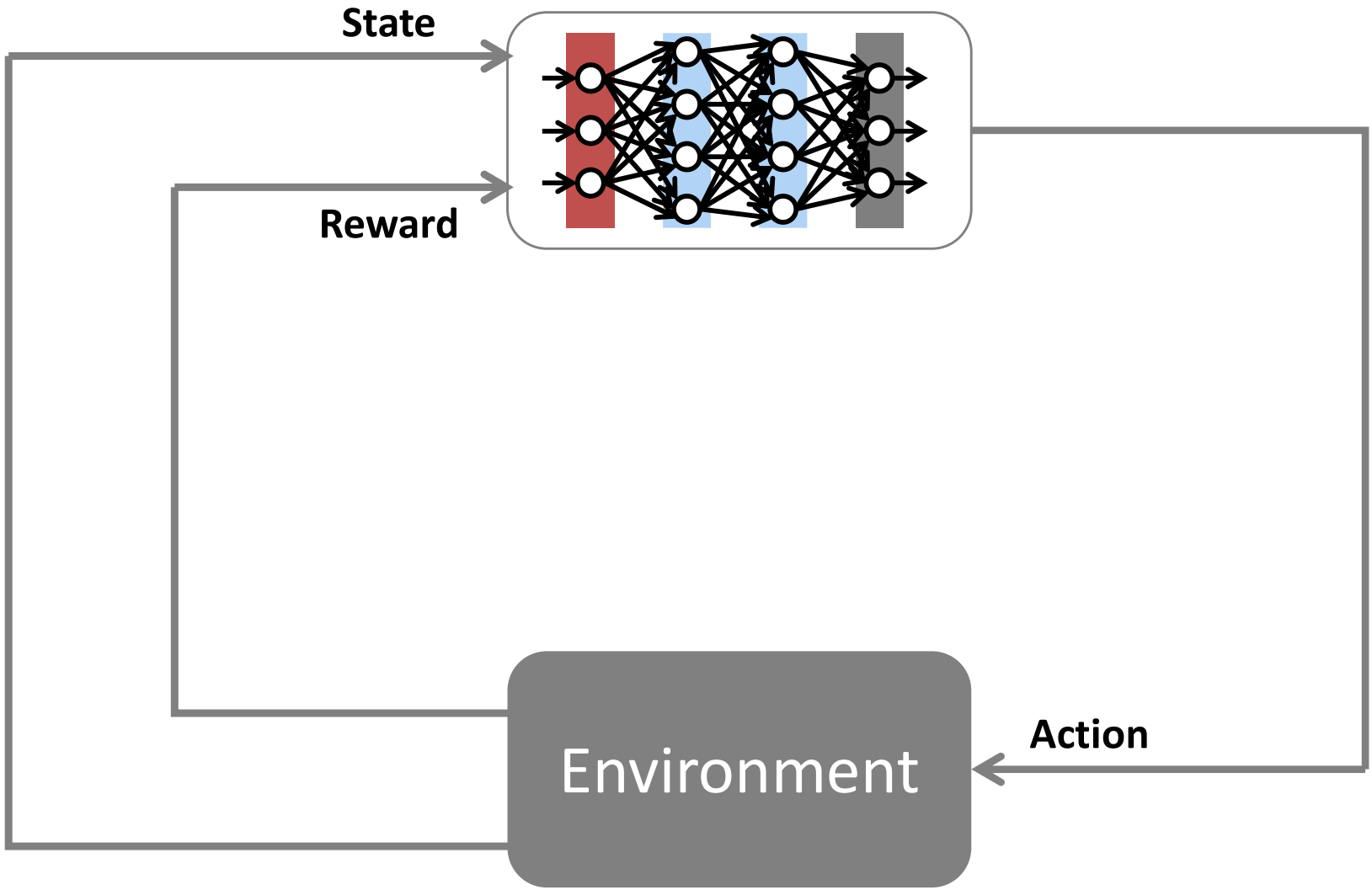


Correct all the weights using Reinforcement Learning.

RL: Agents and Environments



RL: Agents and Environments



K-Armed Bandit Problem

K-Armed Bandit Problem

The K-armed bandit problem is a problem in which a fixed limited set of resources must be allocated between competing (alternative) choices in a way that maximizes their expected gain.

Each choice's properties are only partially known at the time of allocation, and may become better understood as time passes or by allocating resources to the choice.

K-Armed Bandit Problem

In the problem, each machine provides a random reward from a probability distribution specific to that machine, that is not known a-priori.

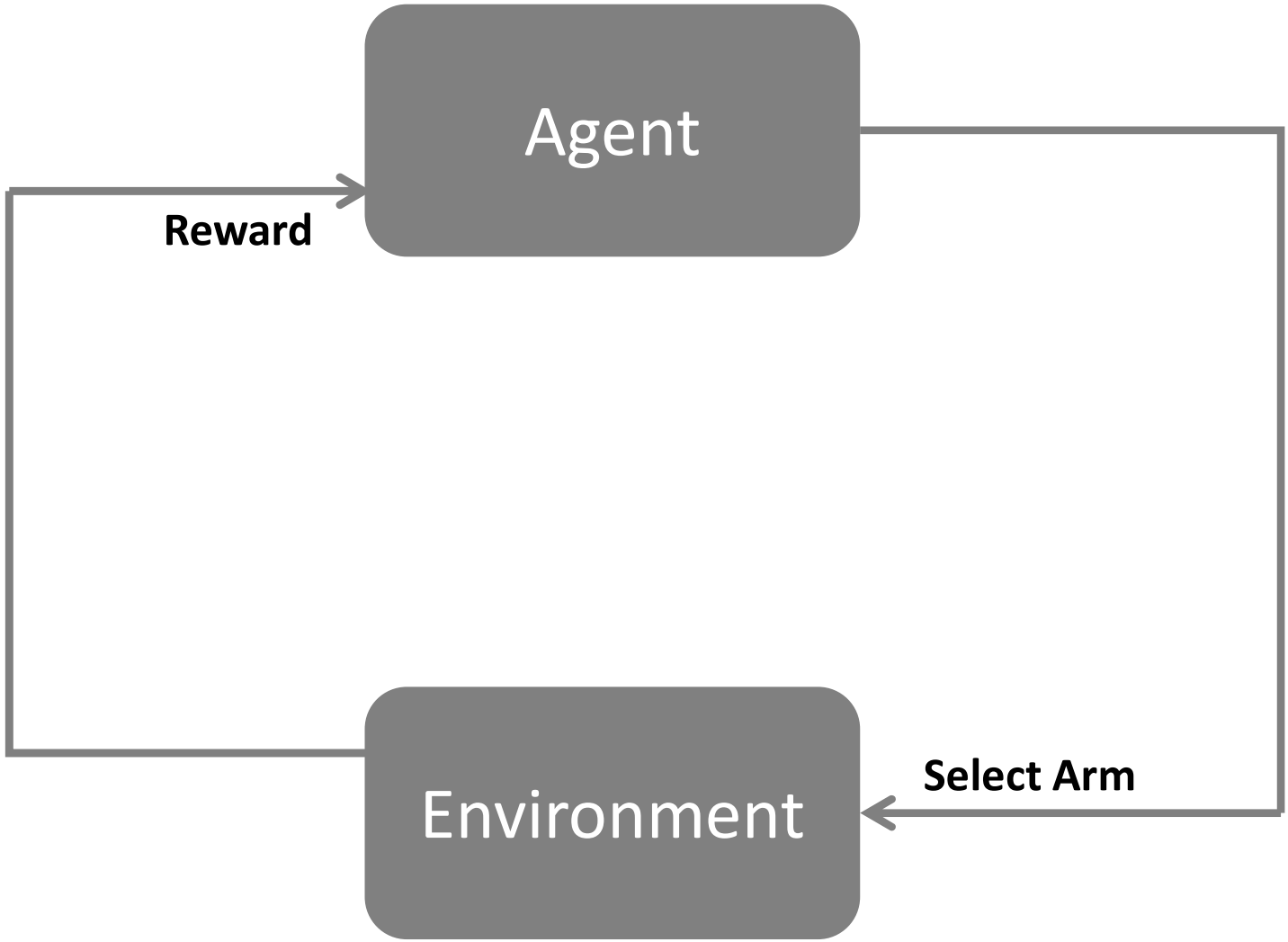
The objective of the gambler is to maximize the sum of rewards earned through a sequence of lever pulls.

K-Armed Bandit Problem

Bandit/Arm 1	Bandit/Arm 2	Bandit/Arm 3
33 %	52 %	78 %
current success (win) rate	**current** success (win) rate	**current** success (win) rate

Which bandit shall we play next?

K-Armed Bandit



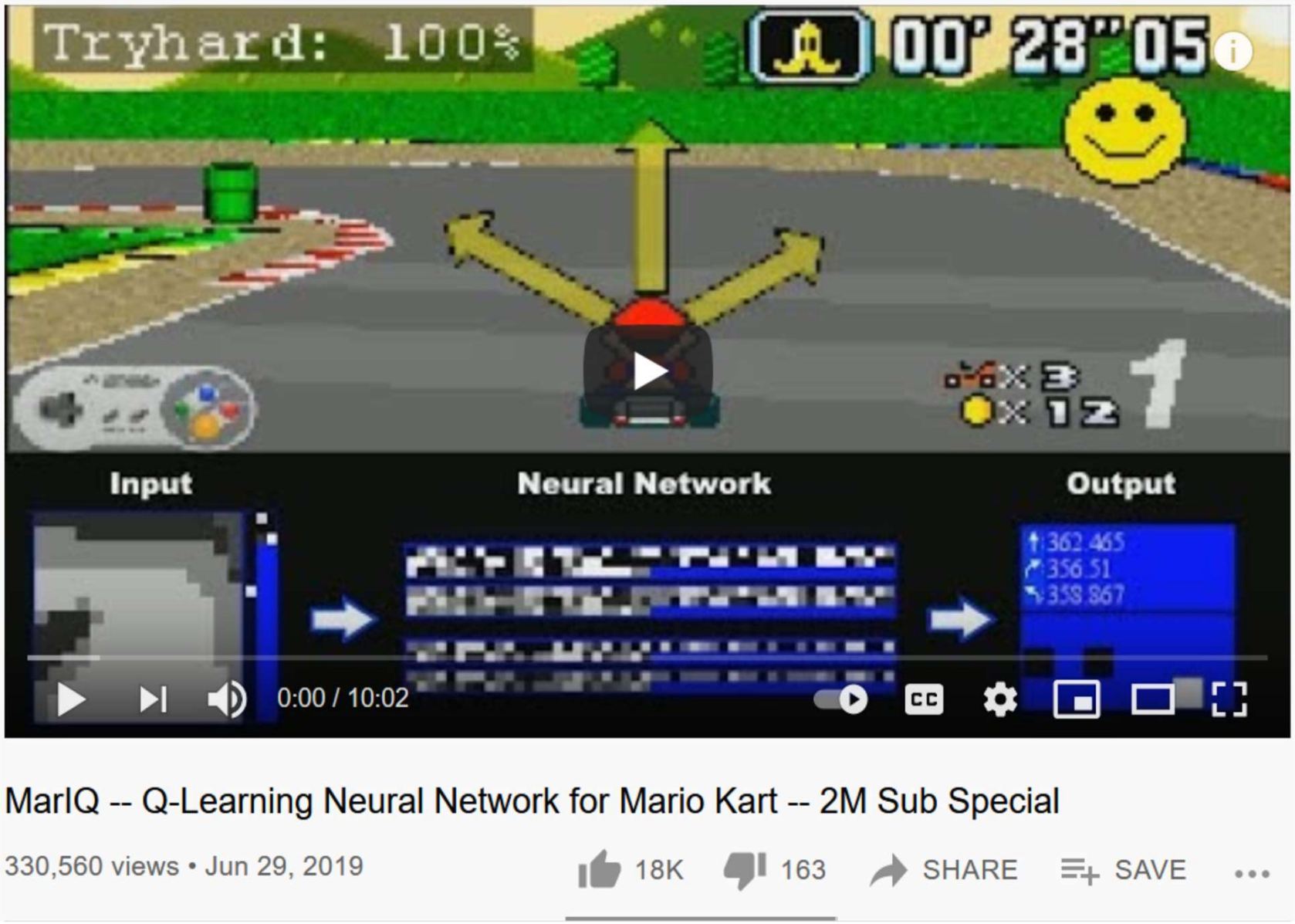
Exploration vs. Exploitation

The crucial tradeoff the gambler faces at each trial is between "exploitation" of the machine that has the highest expected payoff and "exploration" to get more information about the expected payoffs of the other machines.

ε-greedy Algorithm

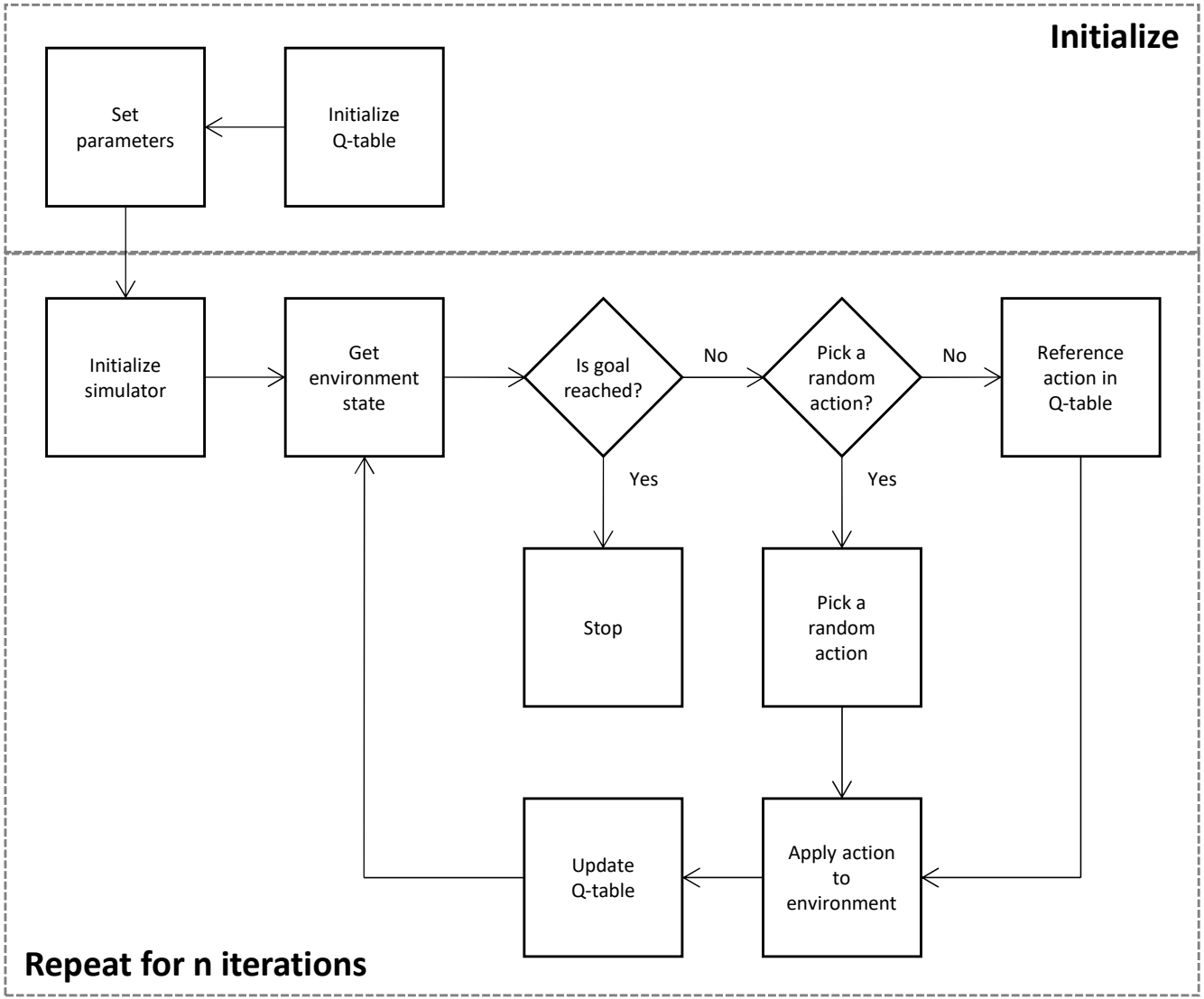
```
generate random number p ∈ [0,1]

if (p < ε)          // explore

    select random arm

else                // exploit

    select current best arm

end
```

Reinforcement Learning in Action

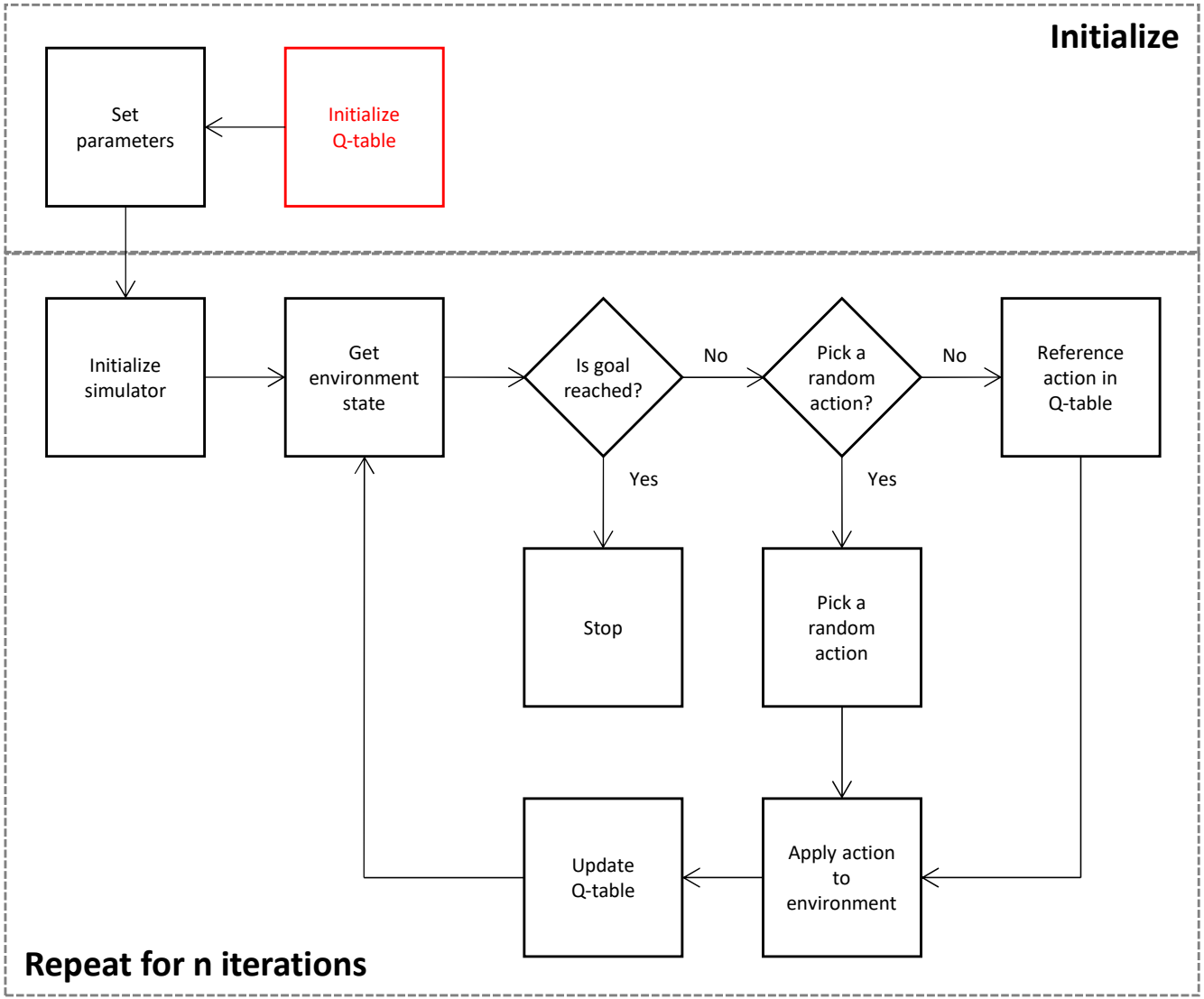


Source: https://www.youtube.com/watch?v=Tnu4O_xEmVk

Q-Learning Algorithm



Q-Learning Algorithm

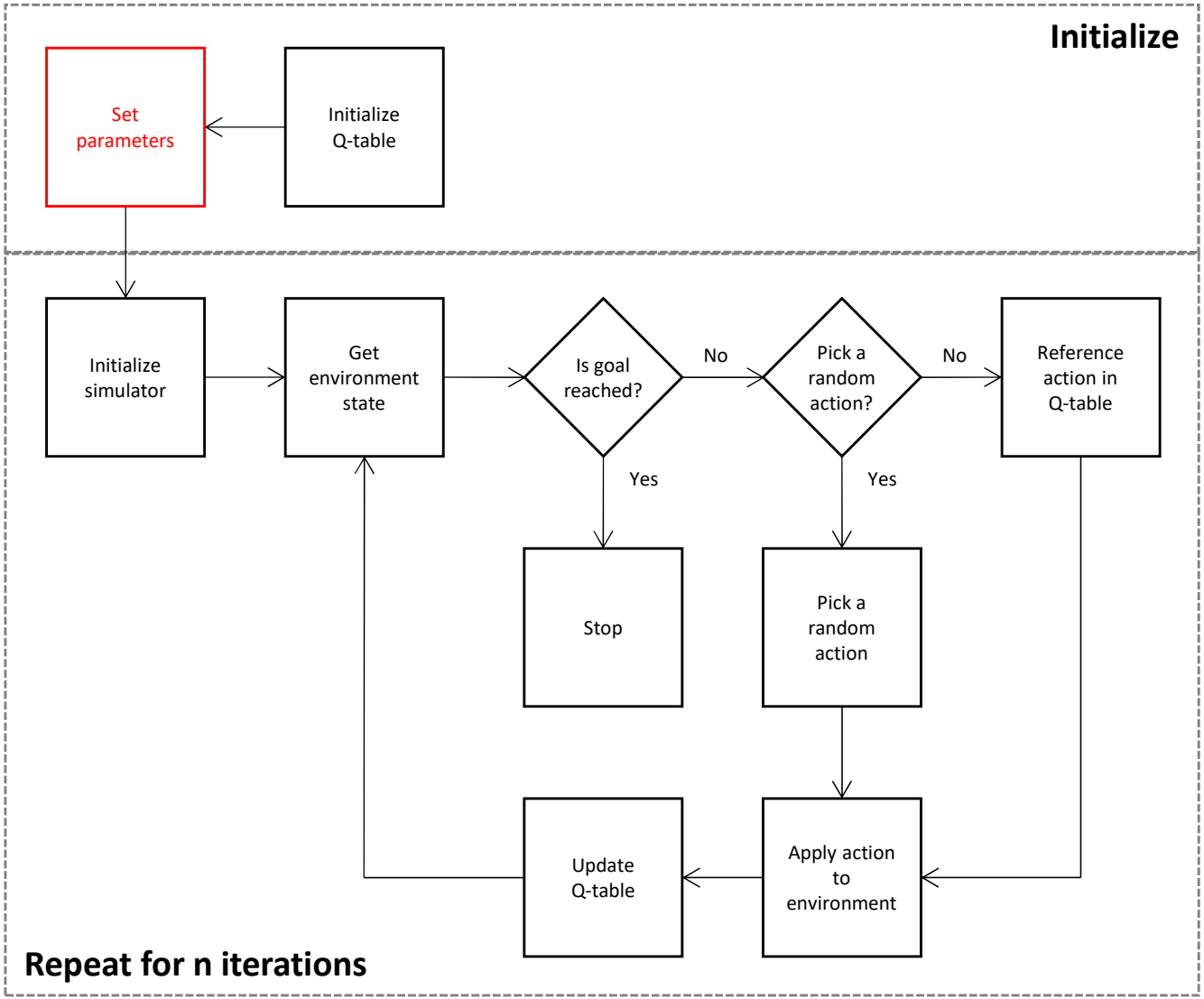


Initialize Q-table:
Set up and initialize (**all values set to 0**) a table where:

- rows represent **possible states**
- columns represent **actions**

Note that additional states can be added to the table when encountered.

Q-Learning Algorithm

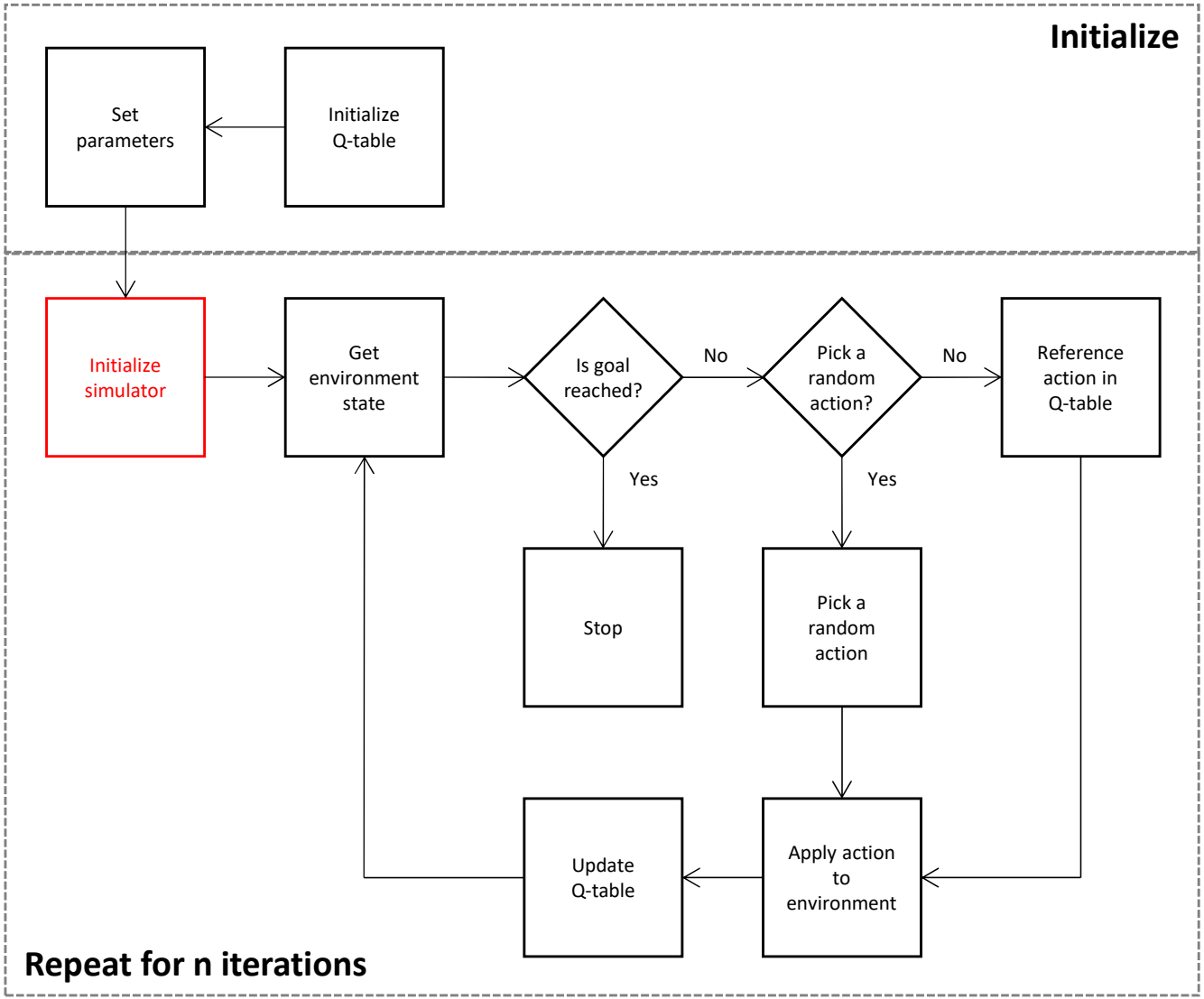


Set parameters:
Set and initialize **hyperparameters** for the Q-learning process.

Hyperparemeters include:

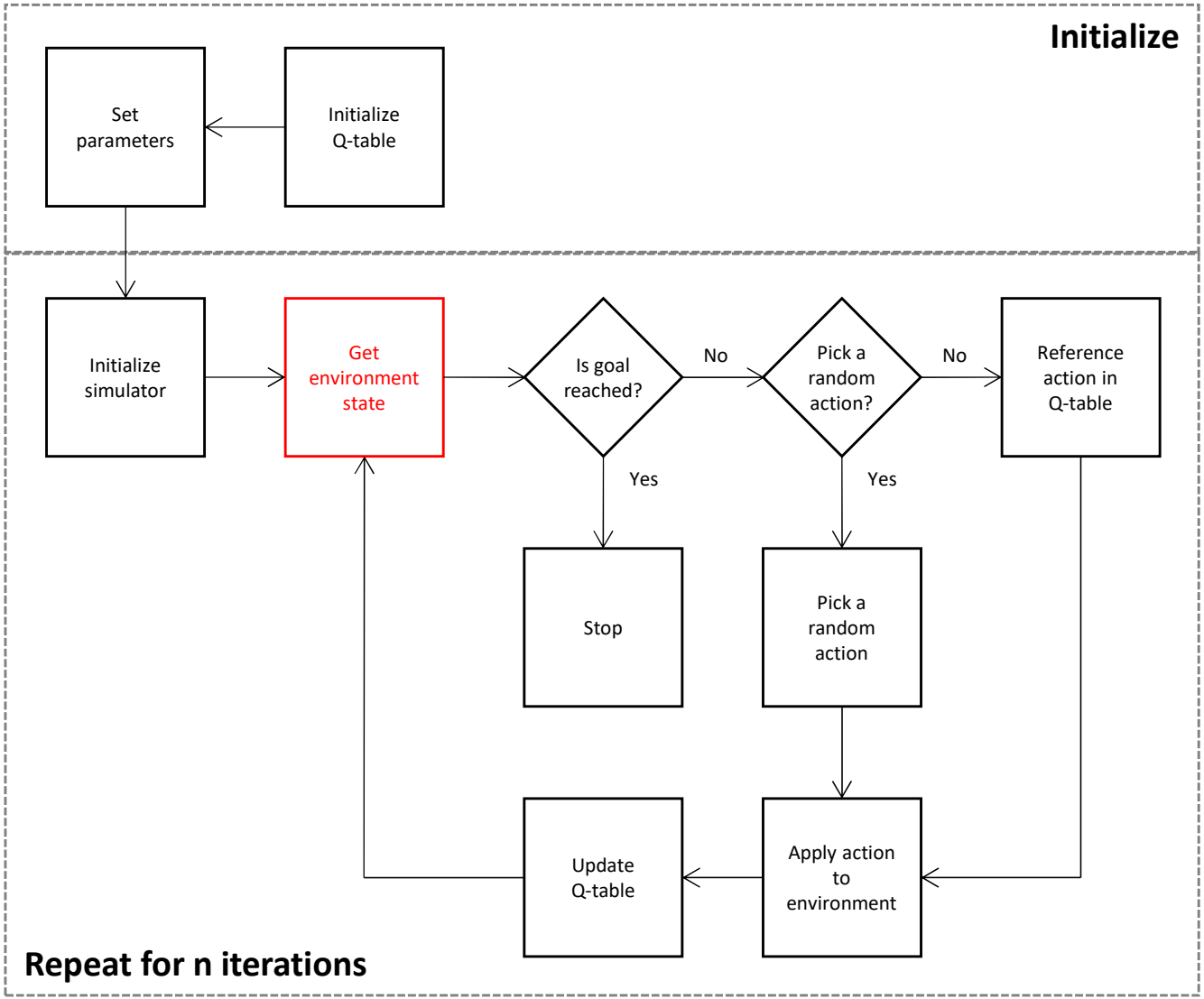
- **chance of choosing a random action**: a threshold for choosing a random action over an action from the Q-table
- **learning rate**: a parameter that describes how quickly the algorithm should learn from rewards in different states
  - high: faster learning with erratic Q-table changes
  - low: gradual learning with possibly more iterations
- **discount factor**: a parameter that describes how valuable are future rewards. It tells the algorithm whether it should seek "immediate gratification" (small) or "long-term reward" (large)

Q-Learning Algorithm



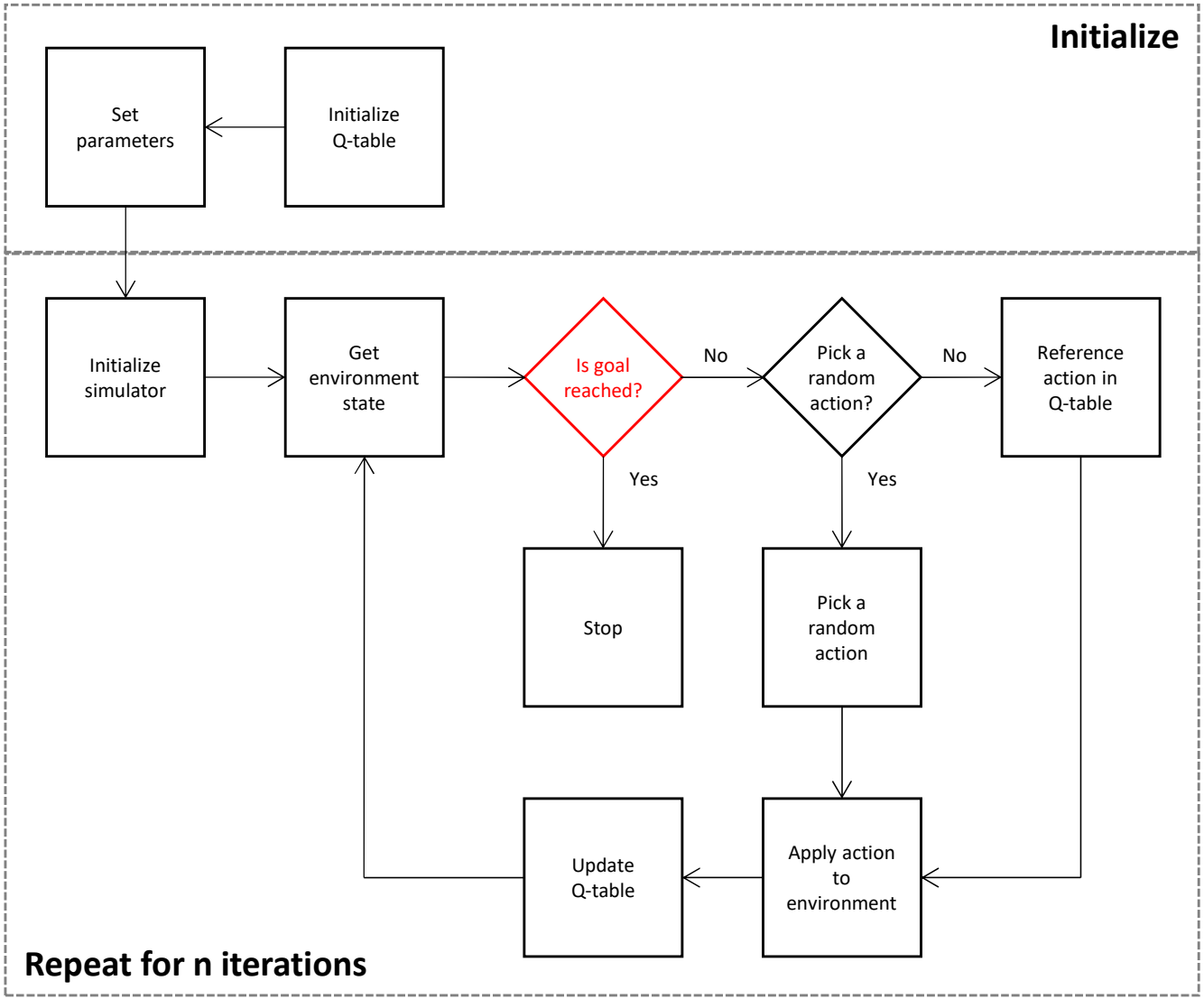
Initialize simulator:
Reset the simulated environment to its initial state and place the agent in a neutral state.

Q-Learning Algorithm



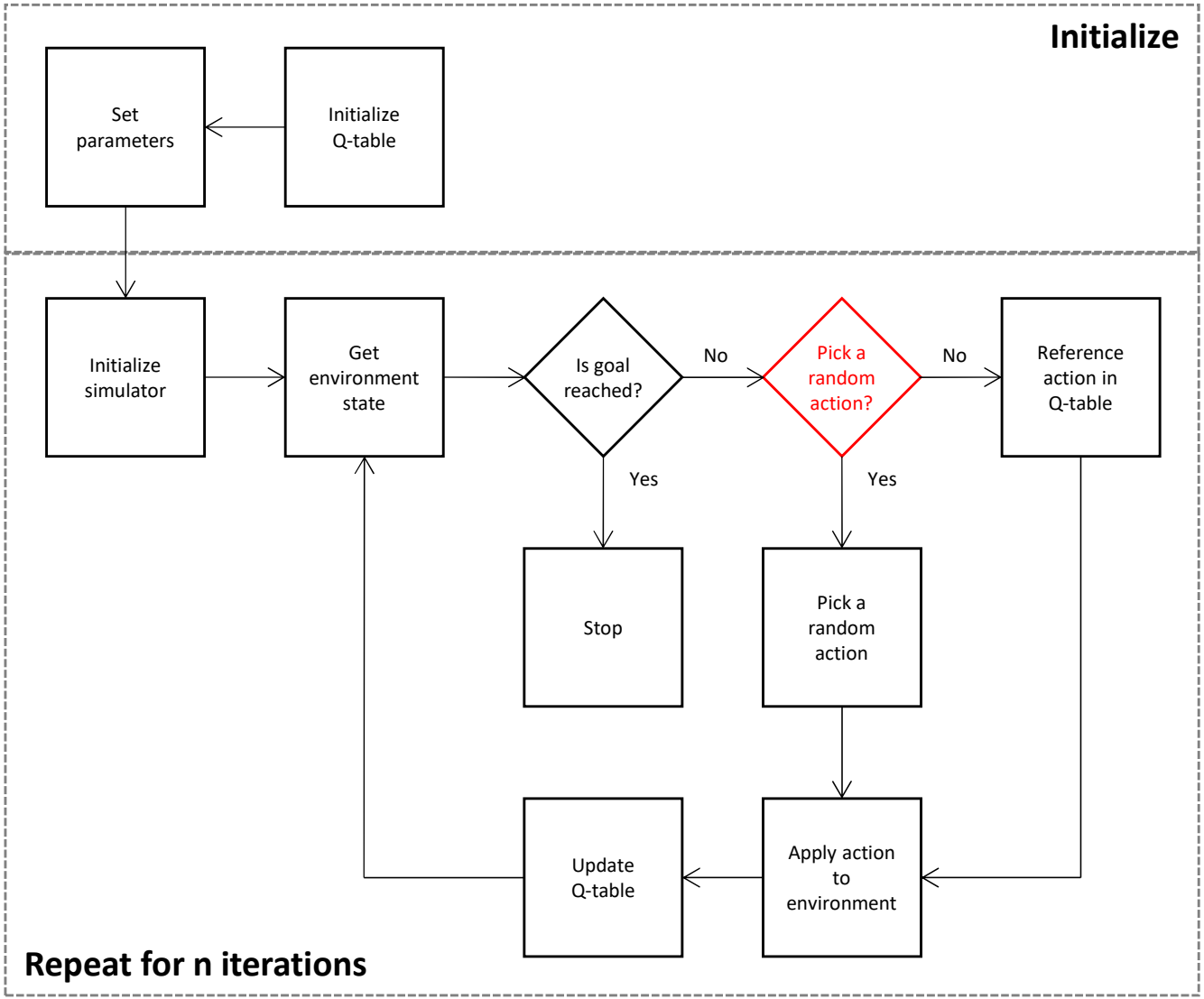
Get environment state:
Report the current state of the environment. Typically a vector of values representing all relevant variables.

Q-Learning Algorithm



Is goal reached?:
Verify if the goal of the simulation has been achieved. It could be decided with the agent arriving in expected final state or by some simulation parameter.

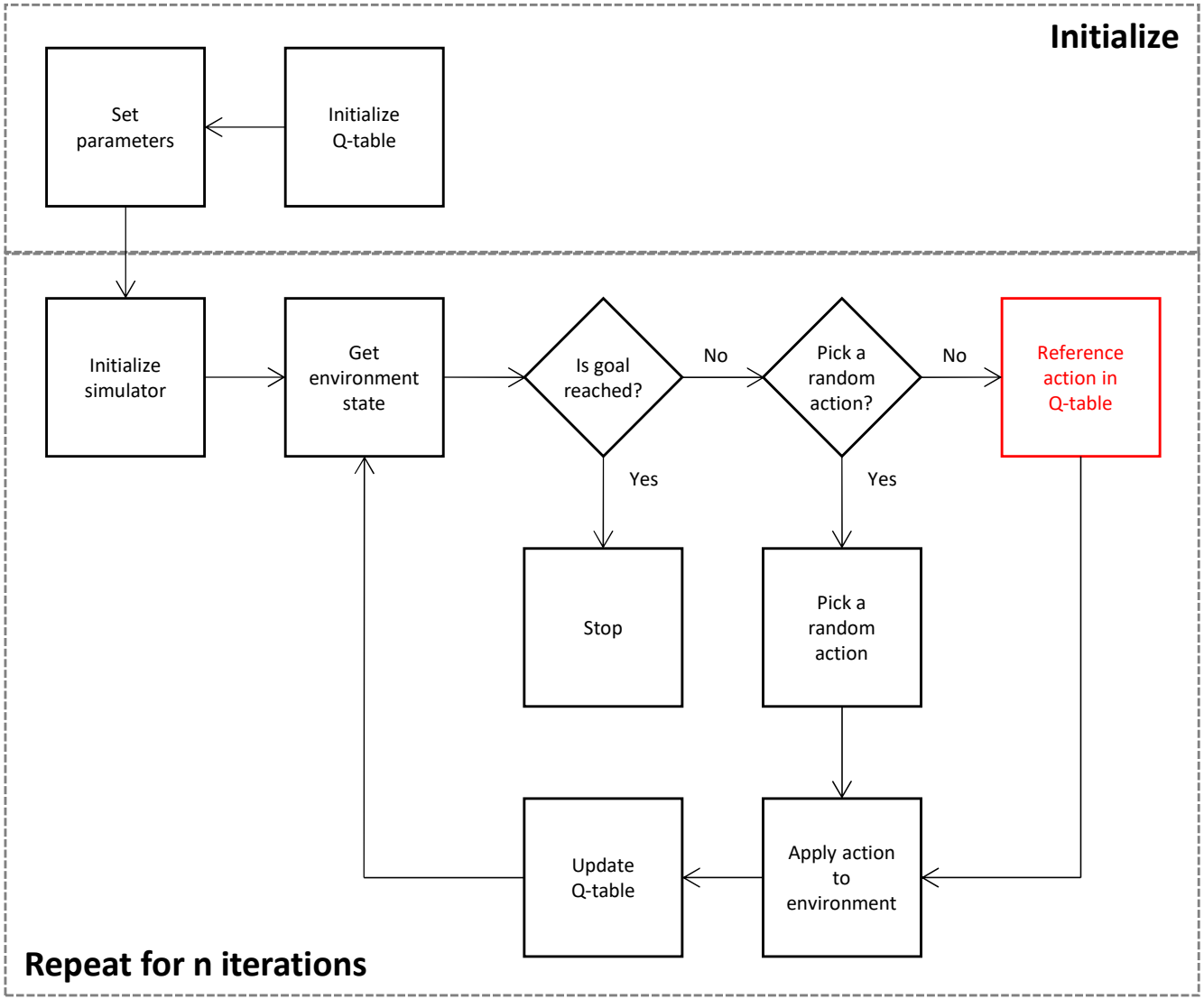
Q-Learning Algorithm



Pick a random action?:
Decide whether next action should be picked at random or not (it will be selected based on Q-table data then).

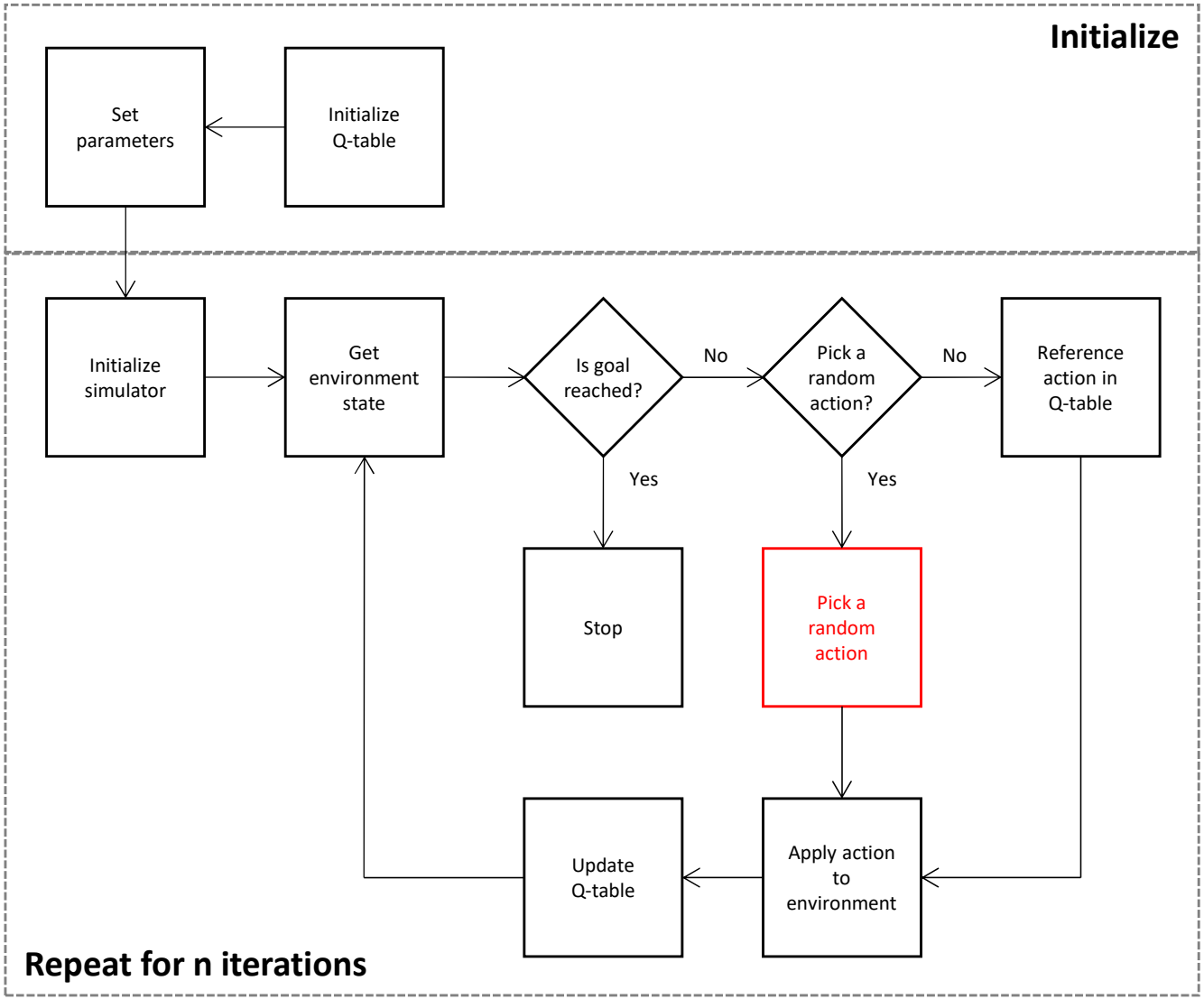
Use the **chance of choosing a random action hyperparameter** to decide.

Q-Learning Algorithm



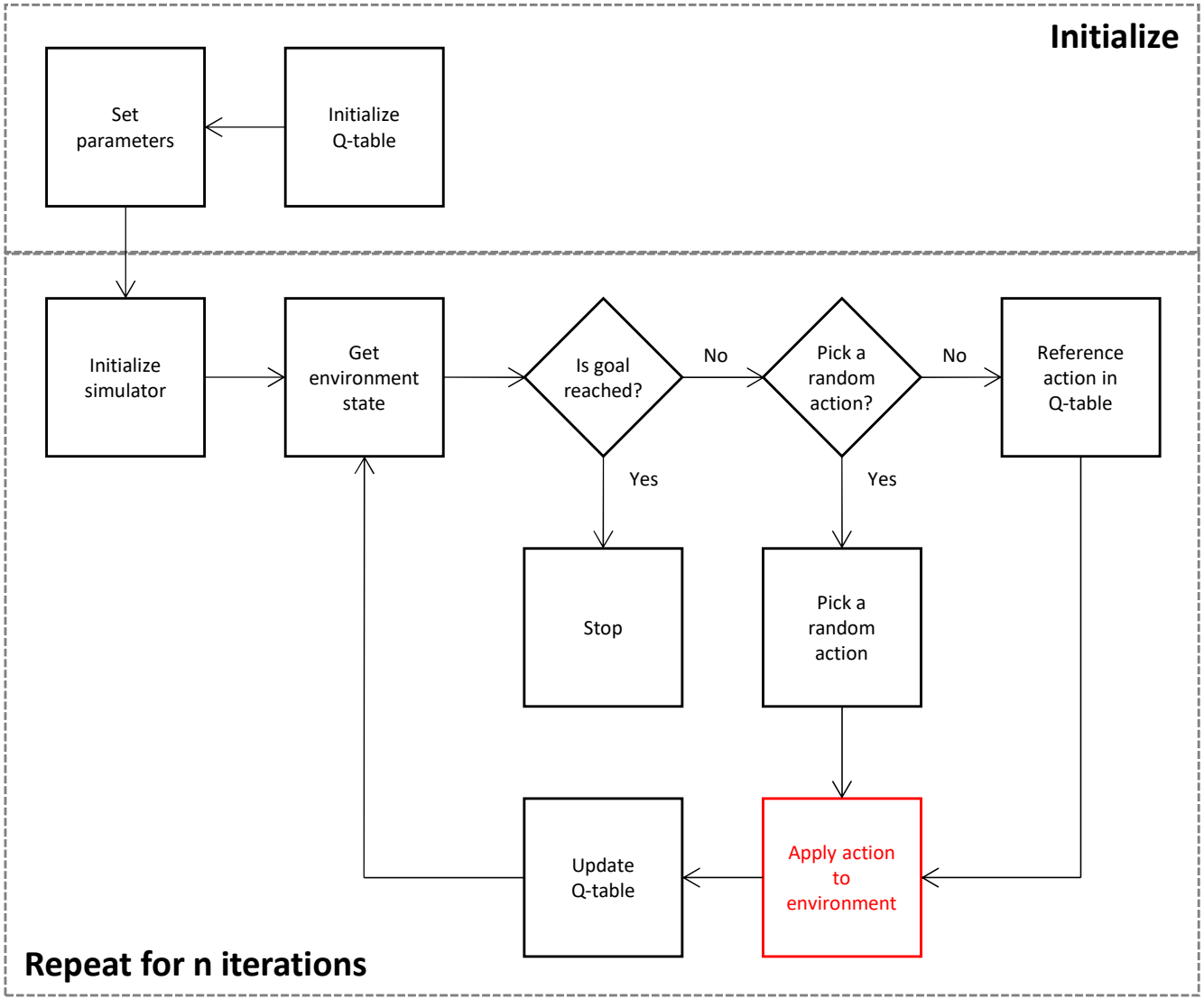
Reference action in Q-table:
Next action decision will be based on data from the Q-table given the current state of the environment.

Q-Learning Algorithm



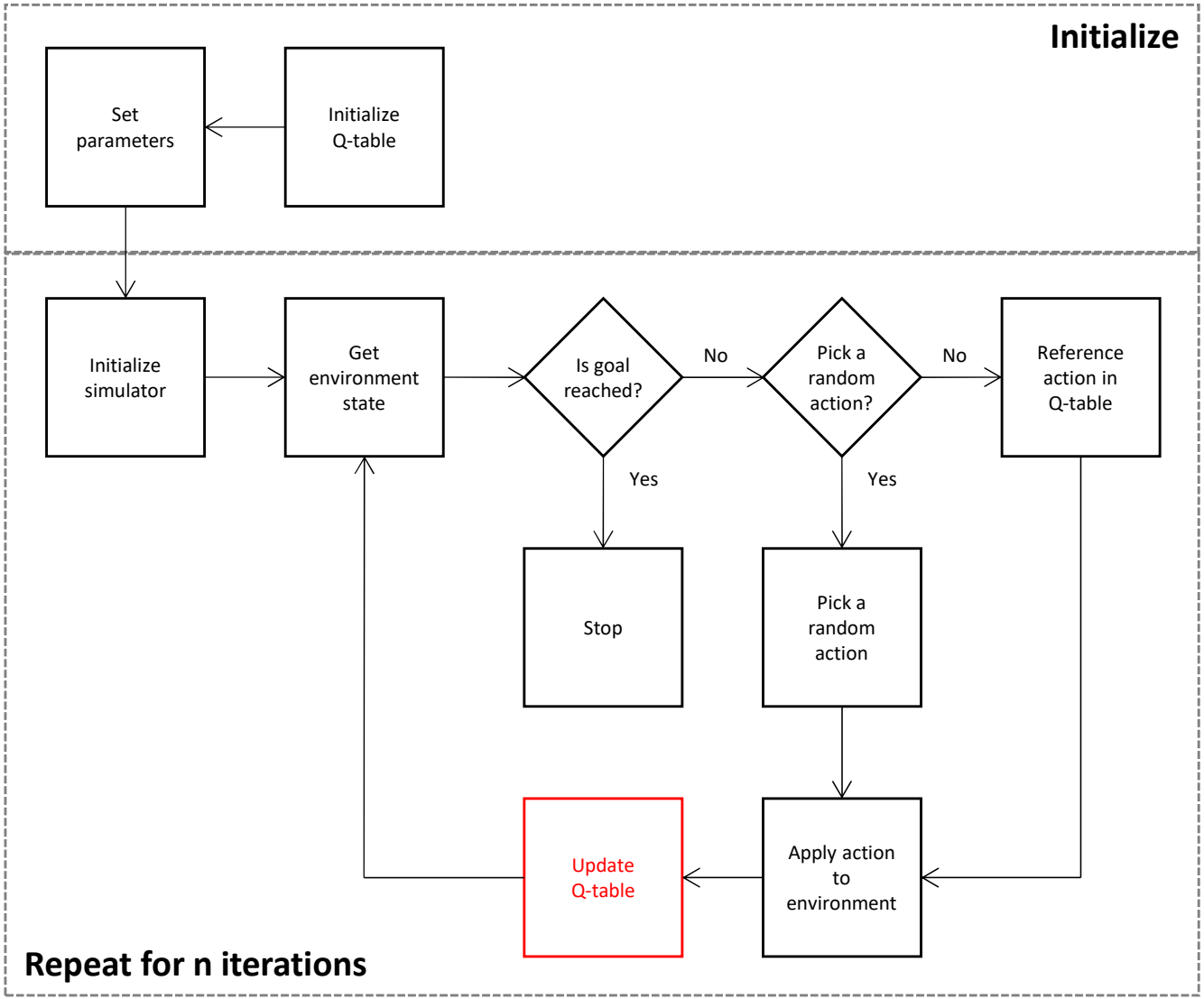
Pick a random action:
Pick any of the available actions at random. Helpful with exploration of the environment.

Q-Learning Algorithm



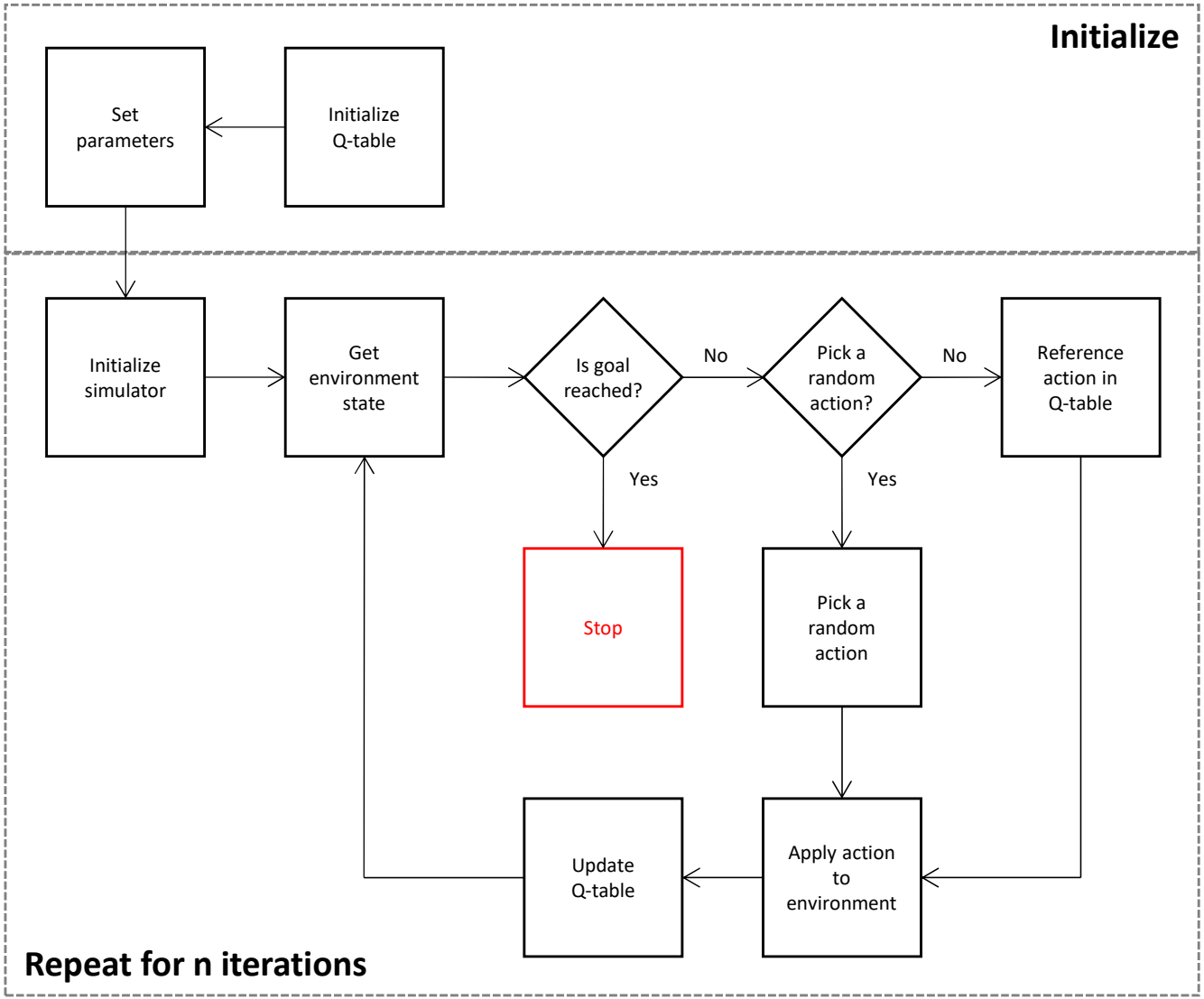
Apply action to environment:
Apply the action to the environment to change it. Each action will have its own reward.

Q-Learning Algorithm



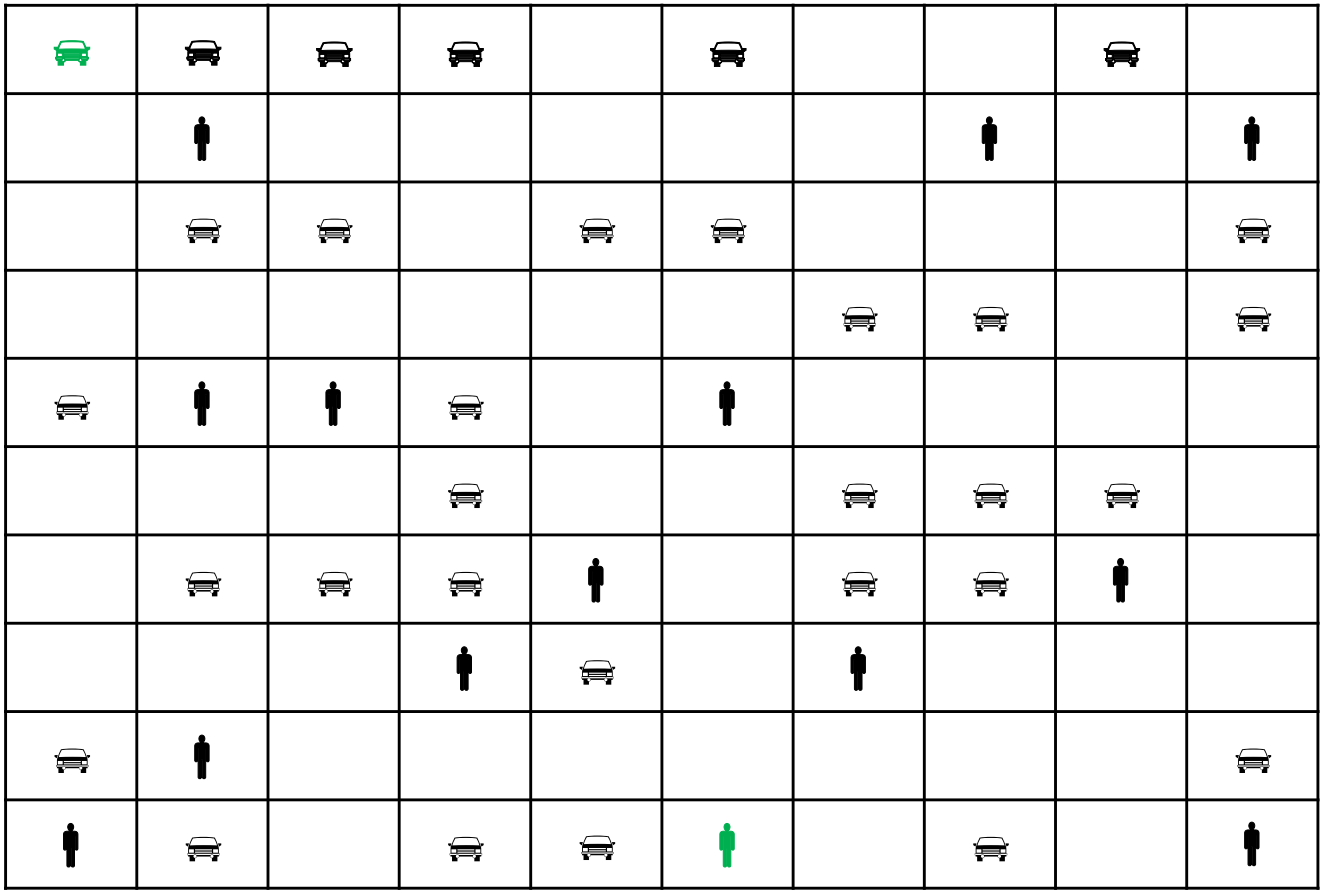
Update Q-table:
Update the Q-table given the reward resulting from recently applied action (feedback from the environment).

Q-Learning Algorithm



Stop:
Stop the learning process

Q-Learning Algorithm



Q-table		Actions			
		↑	↓	→	←
States	1	0	0	0	0
	2	0	0	0	0
	...	...	...	...	...
	n	0	0	0	0

Rewards:

Move into car: -100

Move into pedestrian: -1000

Move into empty space: 100

Move into goal: 500

Action: **Reward:**

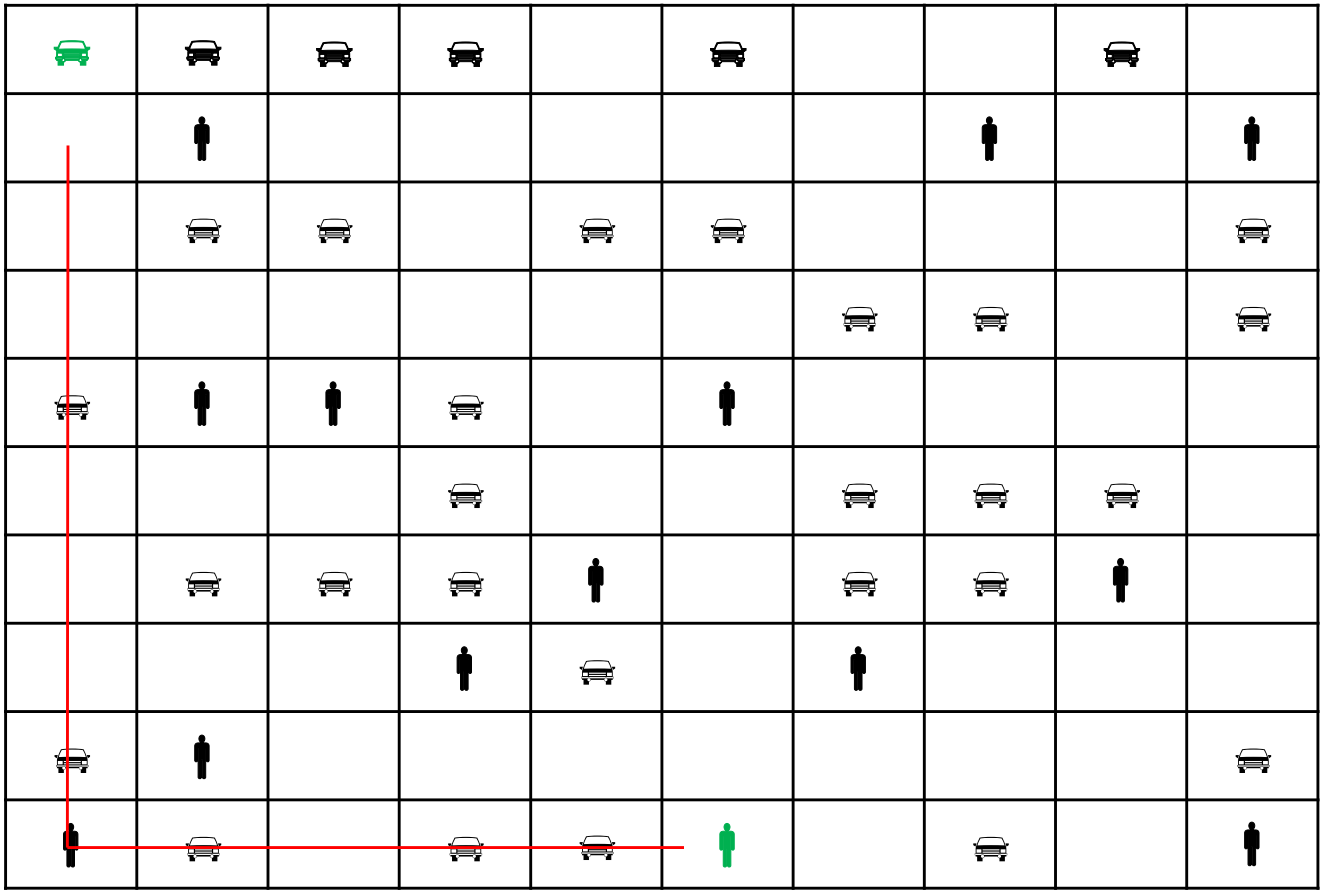
Q-table value:

Learning rate Discount

$$Q(state, action) = (1 - alpha) * Q(state, action) + alpha * (reward + gamma * Q(next\ state, all\ actions))$$

Current value Maximum value of all actions on next state

Q-Learning Algorithm



Q-table		Actions			
		↑	↓	→	←
States	1	0	0	0	0
	2	0	0	0	0
	...	...	...	...	...
	n	0	0	0	0

Rewards:

Move into car: -100

Move into pedestrian: -1000

Move into empty space: 100

Move into goal: 500

Action: **Reward:**

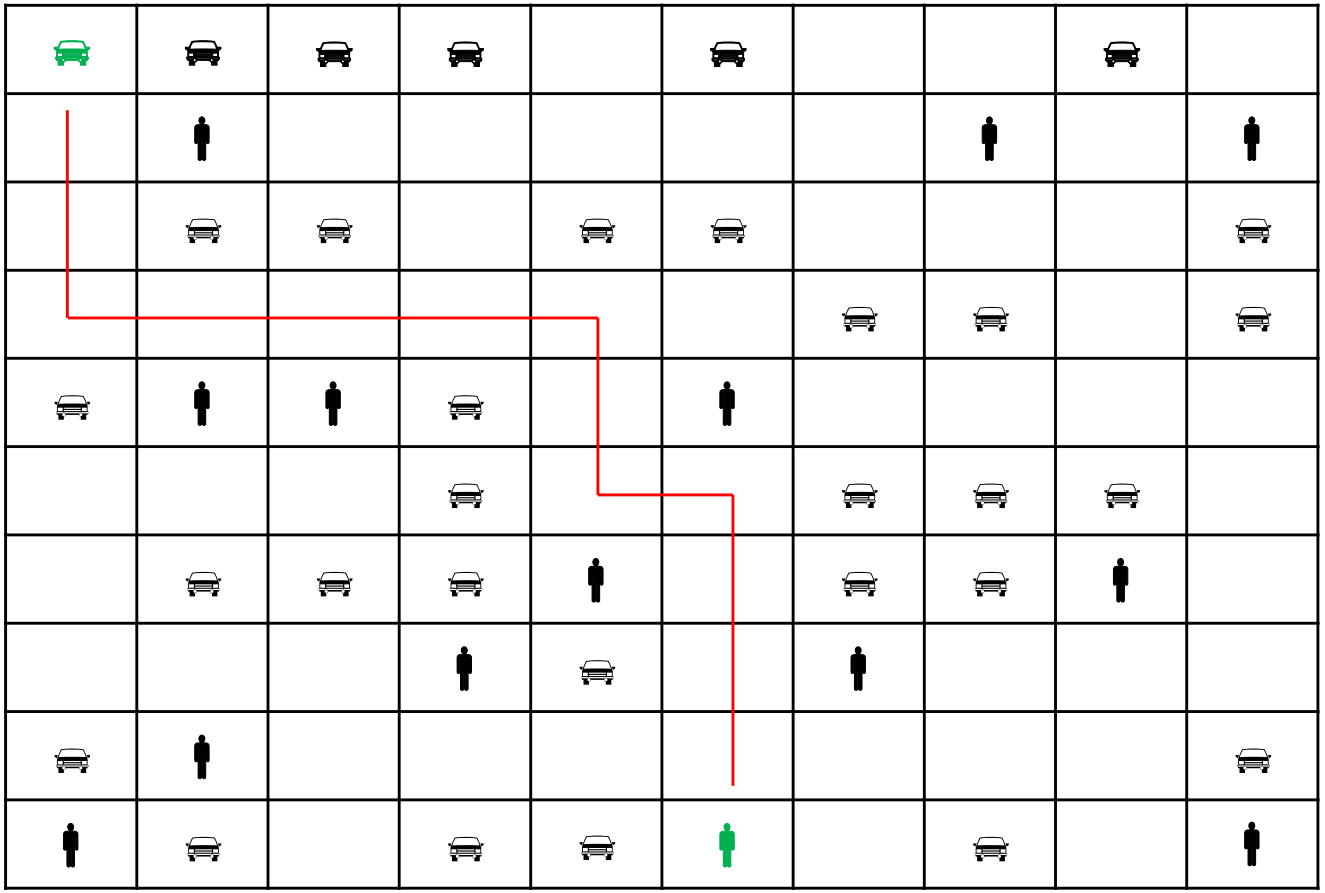
Q-table value:

Learning rate Discount

$$Q(\text{state}, \text{action}) = (1 - \text{alpha}) * Q(\text{state}, \text{action}) + \text{alpha} * (\text{reward} + \text{gamma} * Q(\text{next state}, \text{all actions}))$$

Current value Maximum value of all actions on next state

Q-Learning Algorithm



Q-table		Actions			
		↑	↓	→	←
States	1	0	0	0	0
	2	0	0	0	0
	...	...	...	...	...
	n	0	0	0	0

Rewards:

Move into car: -100

Move into pedestrian: -1000

Move into empty space: 100

Move into goal: 500

Action: **Reward:**

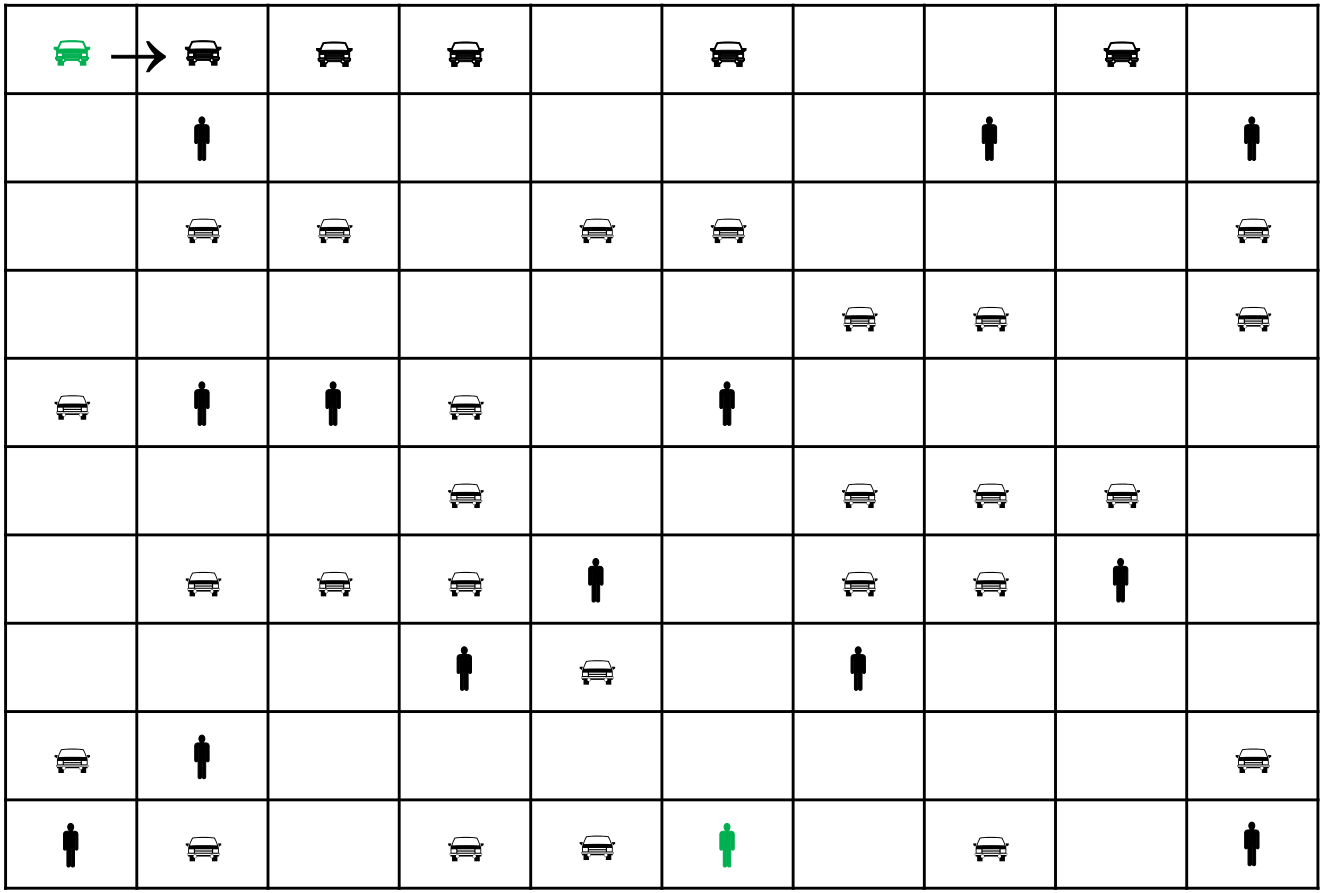
Q-table value:

Learning rate Discount

$$Q(\text{state}, \text{action}) = (1 - \text{alpha}) * Q(\text{state}, \text{action}) + \text{alpha} * (\text{reward} + \text{gamma} * Q(\text{next state}, \text{all actions}))$$

Current value Maximum value of all actions on next state

Q-Learning Algorithm



Q-table		Actions			
		↑	↓	→	←
States	1	0	0	0	0
	2	0	0	0	0
	...	...	...	...	...
	n	0	0	0	0

Rewards:
Move into car: -100
Move into pedestrian: -1000
Move into empty space: 100
Move into goal: 500

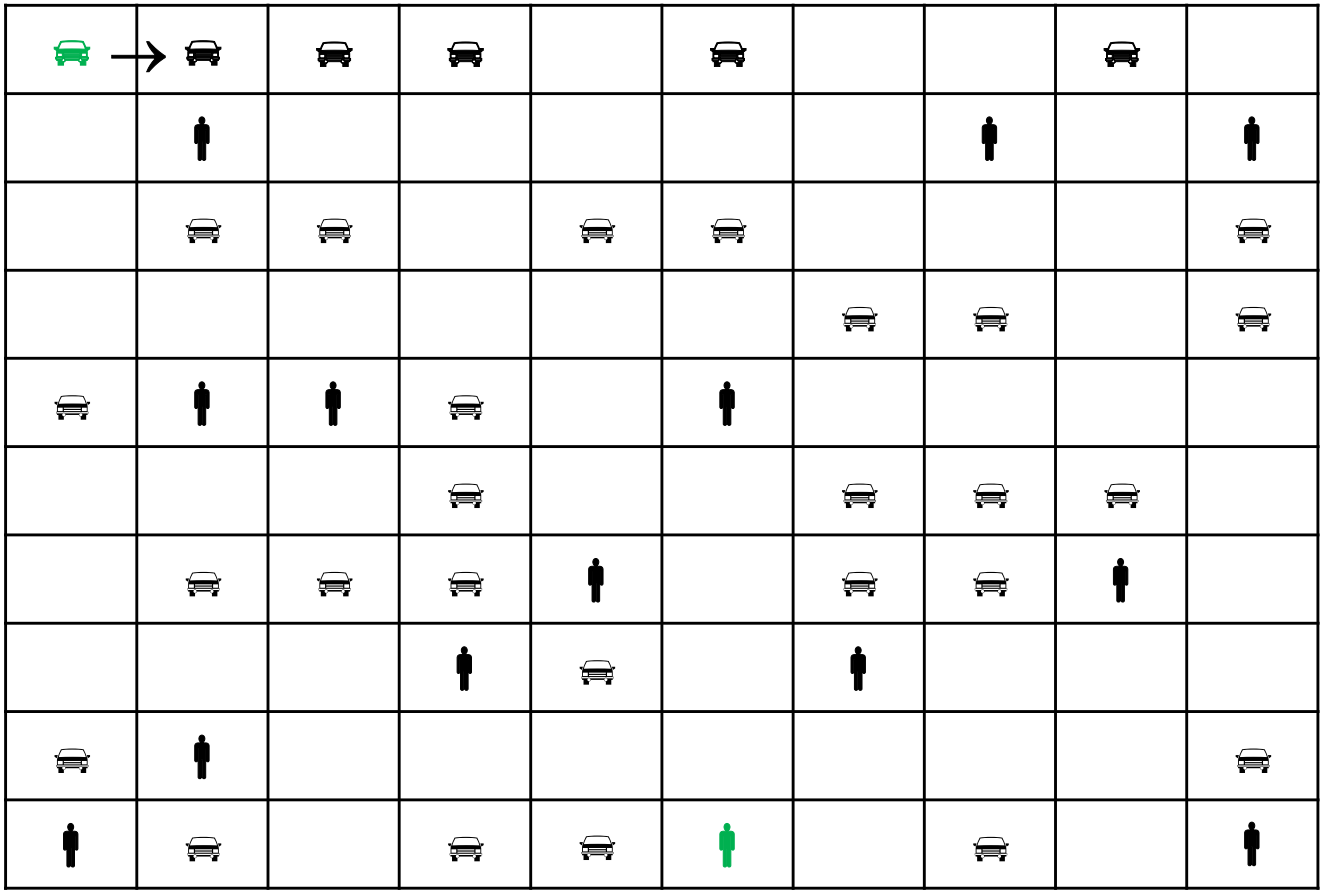
Action: →

Reward: -100

Q-table value:

$$Q(1, \text{east}) = (1 - 0.1) * 0 + 0.1 * (-100 + 0.6 * \max \text{ of } Q(2, \text{all actions}))$$

Q-Learning Algorithm



Q-table		Actions			
		↑	↓	→	←
States	1	0	0	-10	0
	2	0	0	0	0
	...	...	...	...	...
	n	0	0	0	0

Rewards:
Move into car: -100
Move into pedestrian: -1000
Move into empty space: 100
Move into goal: 500

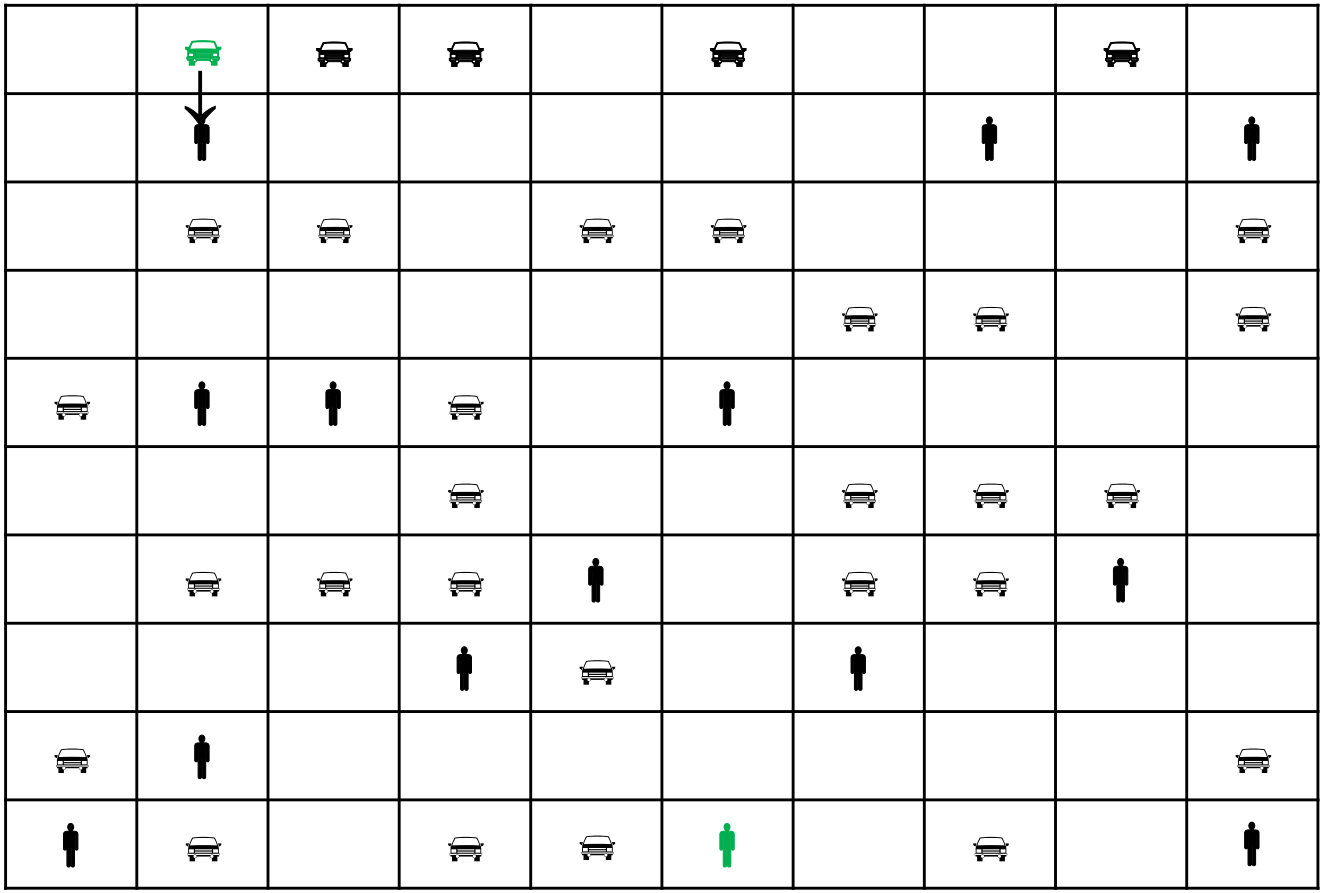
Action: →

Reward: -100

Q-table value:

$$Q(1, \text{east}) = (1 - 0.1) * 0 + 0.1 * (-100 + 0.6 * 0) = -10$$

Q-Learning Algorithm



Q-table		Actions			
		↑	↓	→	←
States	1	0	0	-10	0
	2	0	0	0	0
	...	...	...	...	...
	n	0	0	0	0

Rewards:
Move into car: -100
Move into pedestrian: -1000
Move into empty space: 100
Move into goal: 500

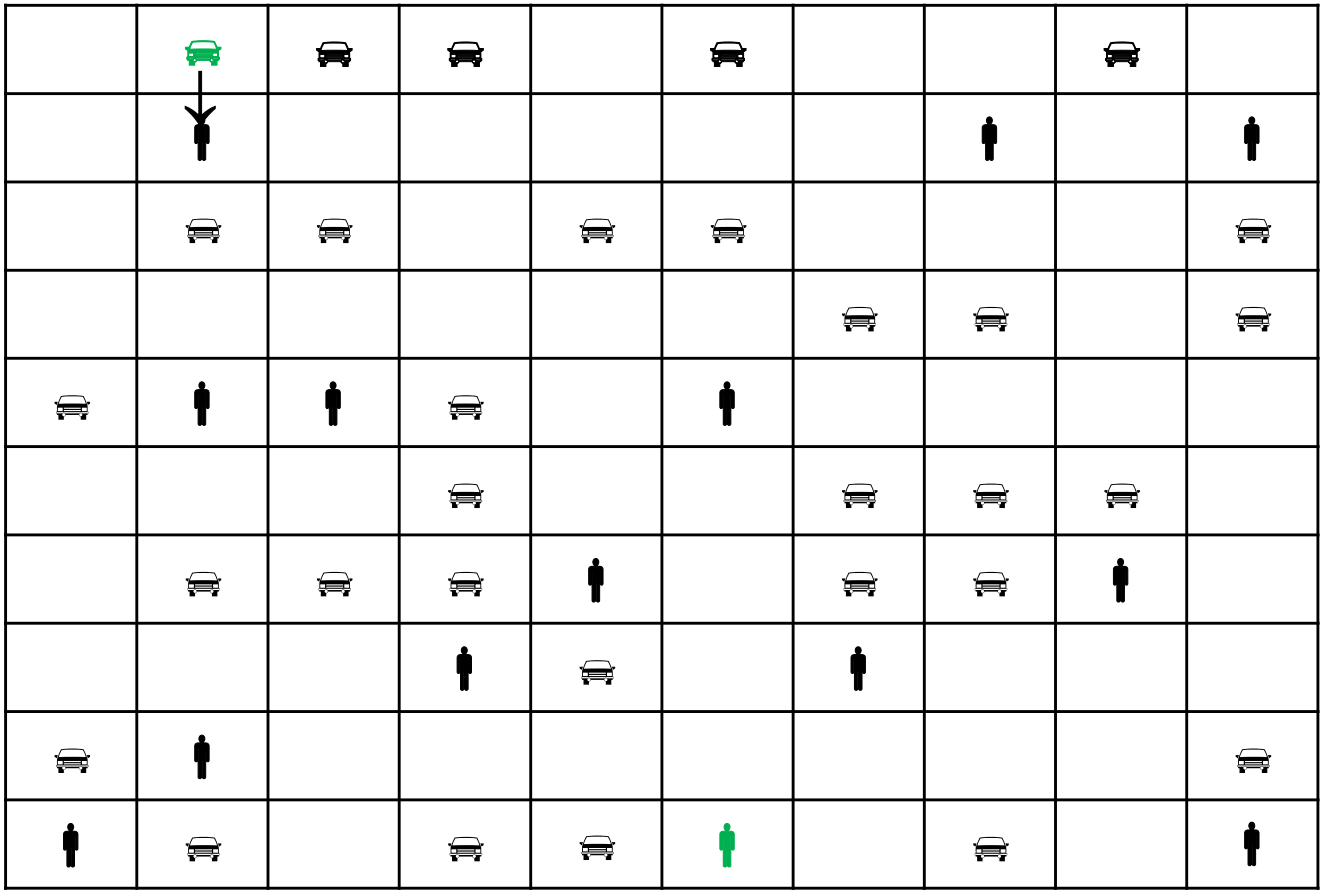
Action: →

Reward: -1000

Q-table value:

$$Q(2, \text{south}) = (1 - 0.1) * 0 + 0.1 * (-1000 + 0.6 * \max \text{ of } Q(3, \text{all actions}))$$

Q-Learning Algorithm



Q-table		Actions			
		↑	↓	→	←
States	1	0	0	-10	0
	2	0	-100	0	0
	...	...	...	...	...
	n	0	0	0	0

Rewards:
Move into car: -100
Move into pedestrian: -1000
Move into empty space: 100
Move into goal: 500

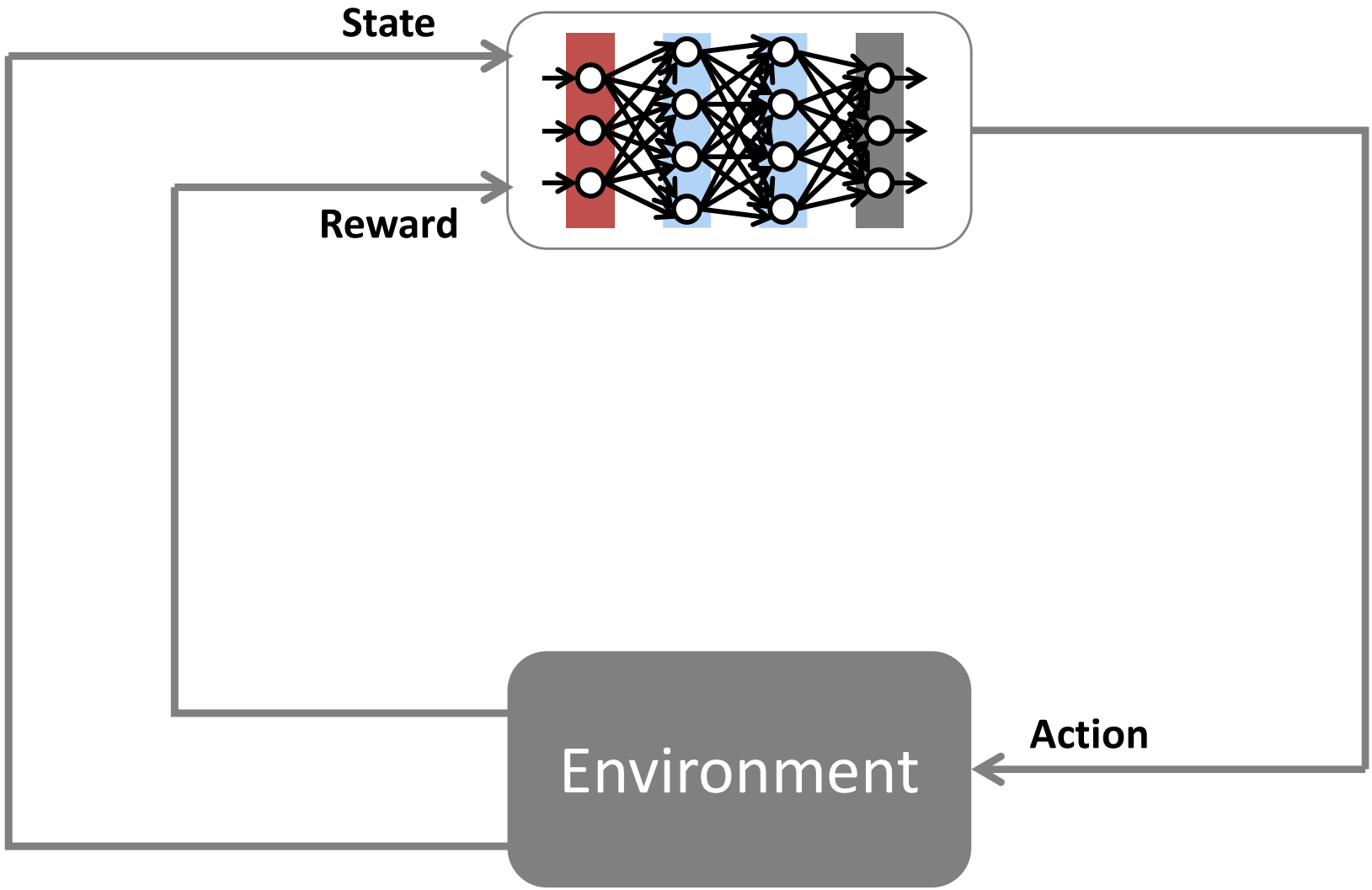
Action: →

Reward:   -1000

Q-table value:

$$Q(2, \text{south}) = (1 - 0.1) * 0 + 0.1 * (-1000 + 0.6 * 0) = -100$$

Deep Reinforcement Learning



RL: Agents and Environments