CIS 419/519

Reinforcement Learning: ML For Decision Making Over Time

Lecture 19

Mar 27, 2023

Instructor: Dinesh Jayaraman

Based on slides from Sergey Levine, Dan Klein, Eric Eaton

Machine Learning Systems Make Decisions

- ML systems make decisions, broadly speaking. For example:
  - A spam classifier might decide whether to place an email in your inbox or spam.
  - ML-based credit scoring in a financial institution might decide whether to approve a loan application.

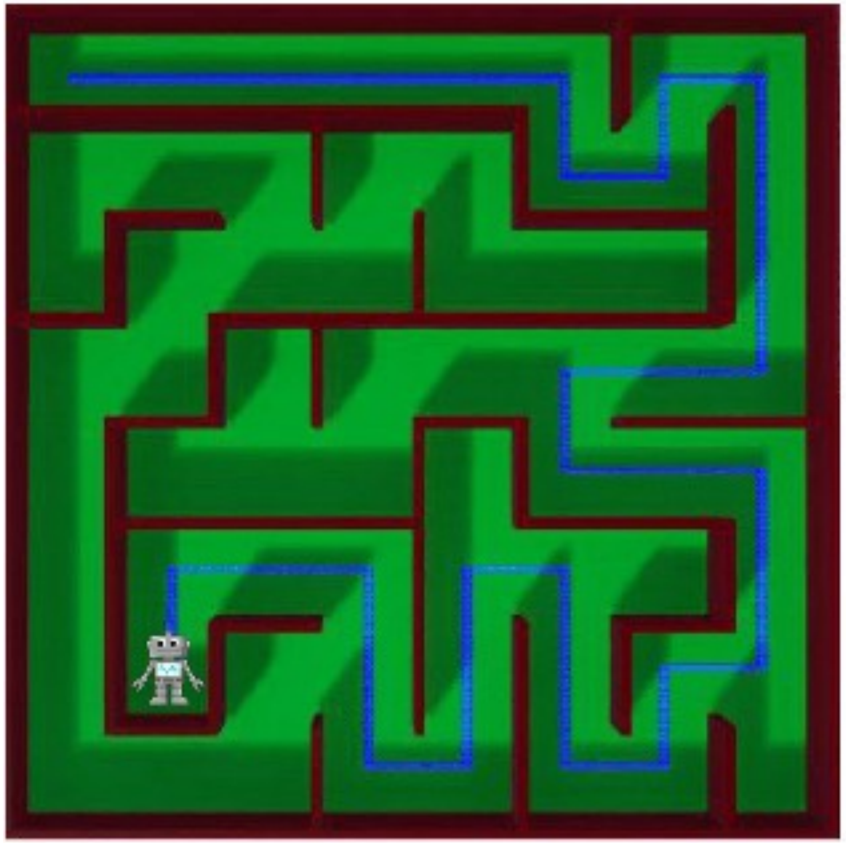
- In these and all the settings we have considered so far, the ML system makes a *one-time* decision.
  - For each loan application or each email, the system would make an independent decision. There is no reason to be influenced by the previous decision.

What if we need to make a series of interconnected decisions over time?

New Problem Setting: Sequential Decision Making

- The decision-making "agent" must make a series of interconnected decisions that affect each other. The outcome of one decision affects the future decision-making process.

- Performance score is typically a function of the full sequence of states and decisions.



Examples of Sequential Decision Making

Must make a sequence of decisions to
maximize some success measure/”reward”,
which is a cumulative effect of the full sequence.

Actions a_t:	muscle contractions	motor current or torque	what to purchase
Observations s_t:	sight, smell	camera images	inventory levels
Reward r_t:	food	average speed	profit

Could we solve sequential decision making with supervised learning?

Towards answering that, let’s try …

Imitation Learning Through Behavior Cloning

Solving sequential decision making problems with supervised learning!

Imitation of Televised Models by Infants

Andrew N. Meltzoff
University of Washington

“Policies” for Sequential Decision Making

For any input state of the system, the ML model maps it to a decision.

- This motivates the following input-output structure of the model:
  - Input: state observation, like sight and smell for the dog.
  - Output: actions, like muscle contractions.

This mapping from input states to a probability distribution over output actions (or sometimes just a single deterministic action) is called a decision-making “policy”, often denoted π.

Supervised learning of Action Policies?

- Given the current "state" x, make a decision $\hat{y} = \max_y \pi_\theta(y|x)$.
  - Supervision => labels for "good" decisions that maximize future rewards.
  - So, we'd like to have some dataset of (state x, good decision y^*) pairs. Then we could try running supervised learning just as always.

- For the sequential decision making problem, we will use the notation:
  - state input s instead of x,
  - action output a instead of y.
  - We will often subscript these items with time indices as s_t, a_t etc.

Behavior Cloning (BC)

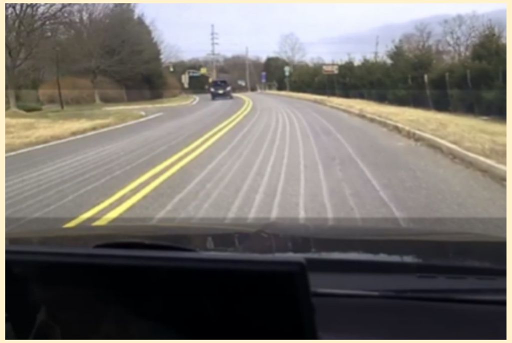
expert

observed states
s_1, s_2, ..., s_H
a_1, a_2, ..., a_H
actions

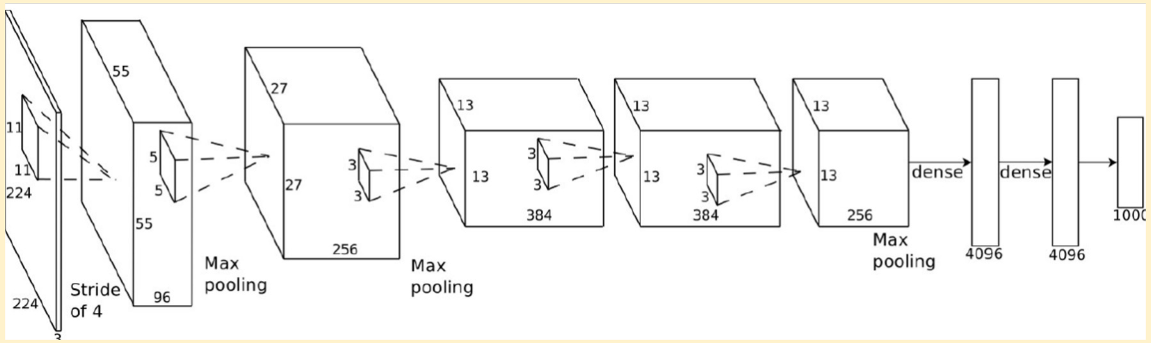
training data

supervised learning

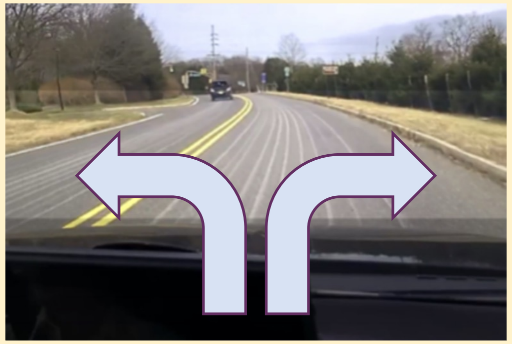
policy
$\pi_\theta(a_t|s_t)$



observed state s_t



convolutional network



action a_t

Behavior Cloning Objective Function

Supervised maximum-likelihood objective to train a function that maps from expert sensory inputs to expert actions.

$$Loss = -\frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\log \pi_\theta(a_{i,t}|s_{i,t})\right)$$

Demonstration data

Expert actions

Could minimize by following the gradient:

$$\frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)$$

trajectories

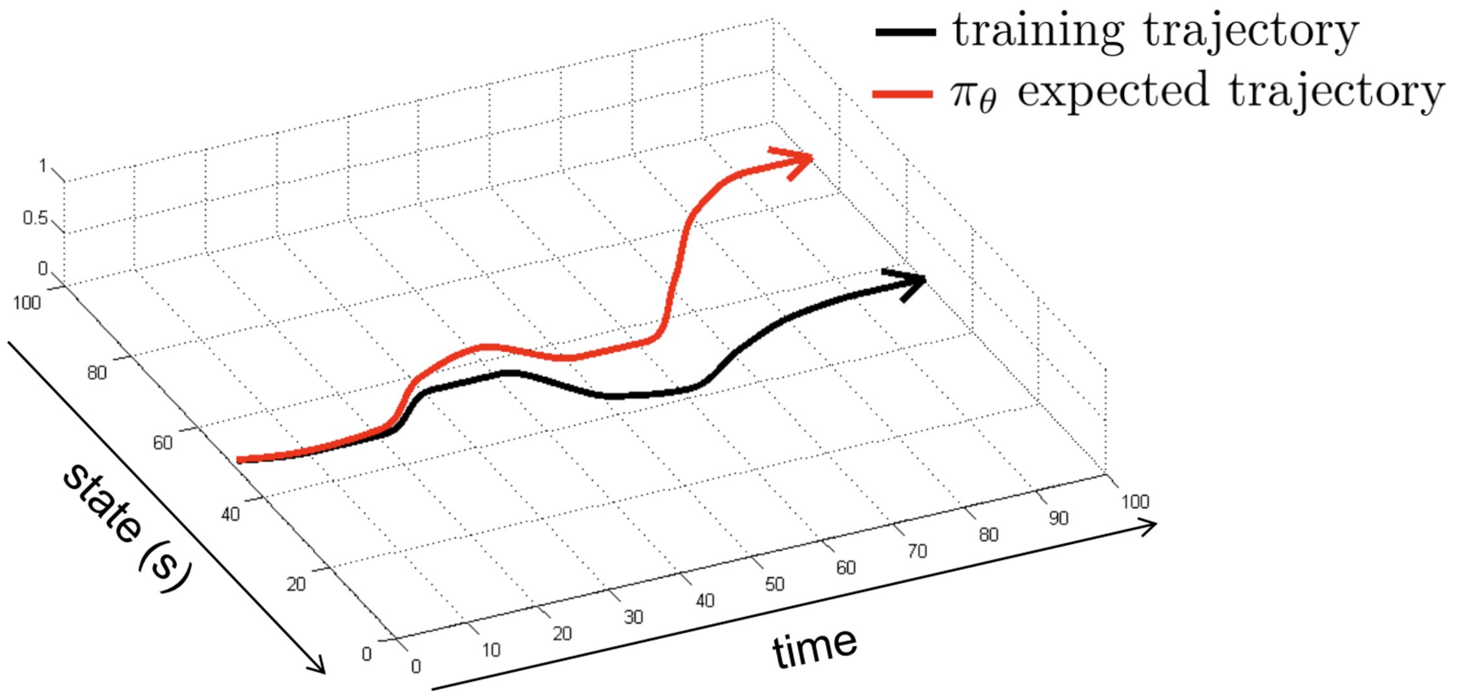
time

Likelihood gradient:
"Change the policy to make these actions more likely".

Does this work?

Key Issue with BC: Distributional Shift

The policy is trained on *demonstration data* that is different from the data it encounters in the world.



The cloned policy is imperfect; this leads to "compounding" errors, and the agent soon encounters unfamiliar states, leading to failure.

Note how these errors arise from ignoring the the *sequential, interconnected* nature of the task. Past decisions influence future states!

Active Behavior Cloning: DAGGER

A general trick for handling distributional shift: requery expert on new states encountered by the initial cloned policy upon execution, then retrain.

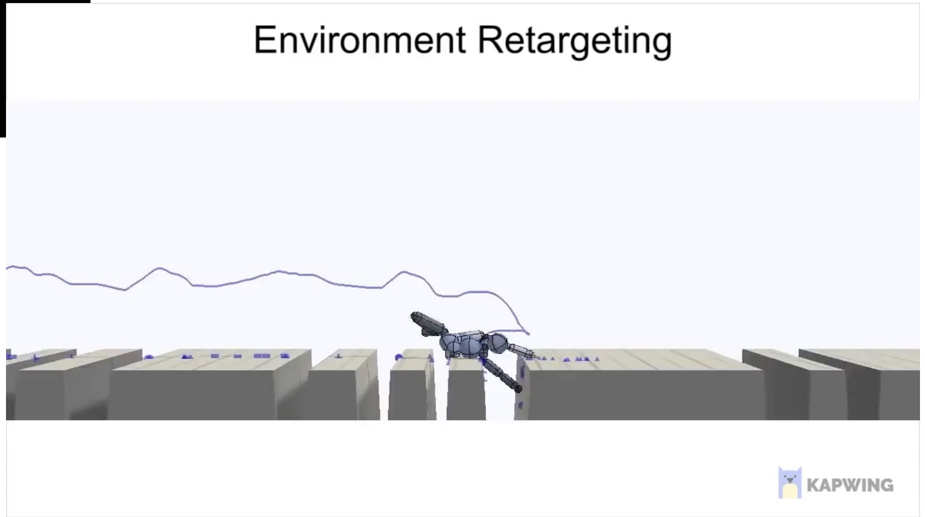
1. Train $\pi_\theta(a_t|s_t)$ from expert data $\mathcal{D} = \{s_1, a_1, \ldots, s_N, a_N\}$
2. Run $\pi_\theta(a_t|s_t)$ to get dataset $\mathcal{D}_\pi = \{s_1^{new}, \ldots, s_M^{new}\}$
3. Ask expert to label each state in $\mathcal{D}_\pi$ with actions a_t^{new}
4. Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$

Assumes it is okay to keep asking the expert all through the training process.
“Queryable experts”. Might not always be practical.

Ross et al, DAGGER, 2011

https://www.youtube.com/watch?v=-96BEoXJMs0

https://www.youtube.com/watch?v=M-QUkgk3HyE



https://xbpeng.github.io/projects/SFV/index.html

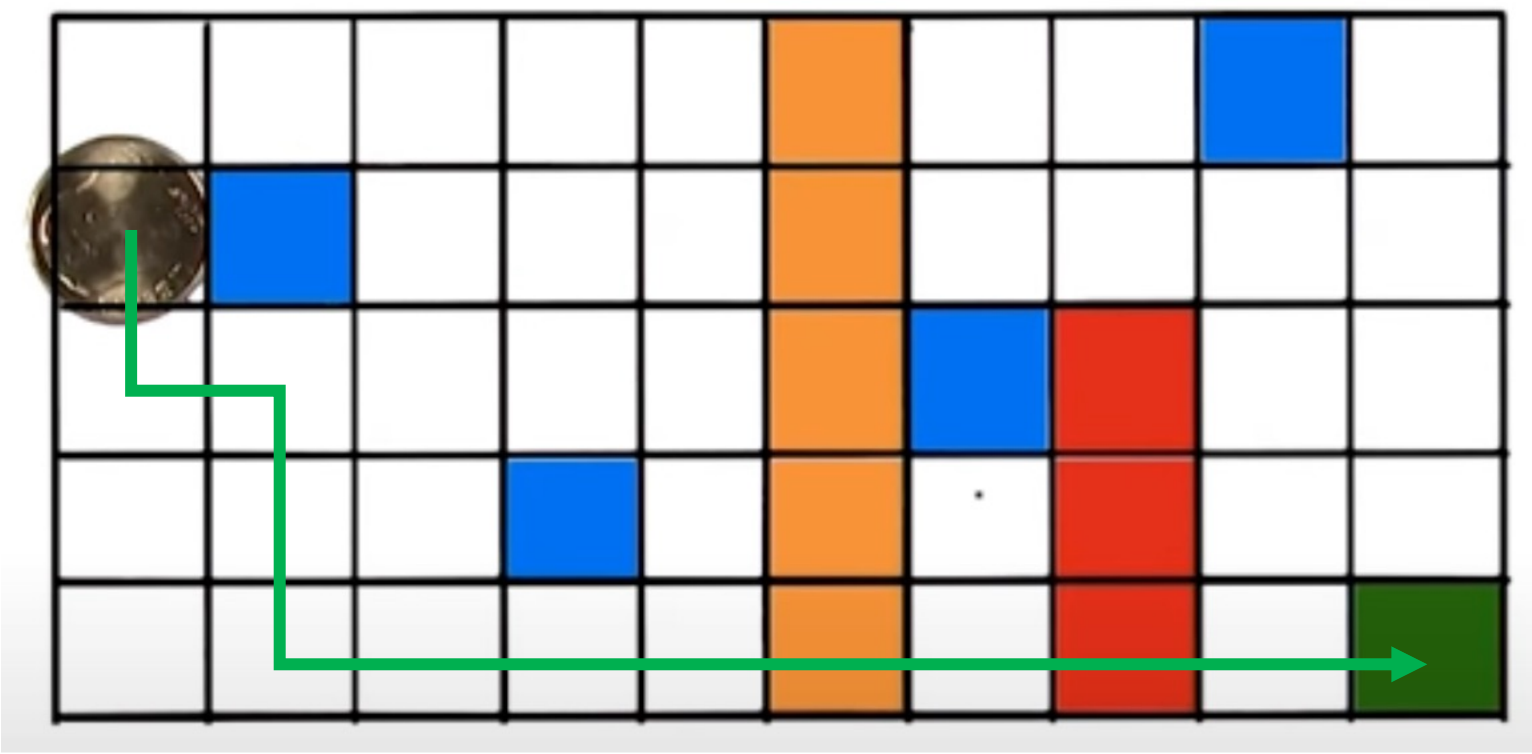
Aside: Distribution Shift More Broadly

- When supervised ML systems are deployed, it is common for the distribution to shift.
    - E.g. when a new spam classifier is deployed on gmail, spammers might notice that their old spamming techniques are not working, and innovate to break the new spam classifier.
- One strategy to fix this is continuous data aggregation, like in DAGGER.
    - E.g., Allow users to mark new emails that slip through the filter as spam. Add these to the training data, and retrain the spam classifier from time to time.

Lesson: ML systems *are* often deployed in sequential decision making settings without realizing it: later inputs may be influenced in some complex way by older decisions of the ML system. Warrants caution!

Other Ways to Do Imitation

- BC might not generalize beyond demonstrations. Instead learn explicitly about the "reward" function that the demonstrator is trying to maximize?
  - This is called "inverse reinforcement learning"



Would you conclude that this agent likes / dislikes:

- Blue squares?
- White squares?
- Orange squares?
- Red squares?
- Green square?

Knowing the *reward* could inform more generalizable imitation, e.g. starting from a different location than expert

BC Operates Per-Timestep, Ignores Future Impacts

- Suppose you try to imitate driving. The imitator is not perfect, and you either:
  - Are slower by 5 mph than the expert behavior on a highway, or
  - Are off by 5 mph as you start your car in your garage (e.g. moving forward at 4 mph, instead of backing out at 1 mph).
  - BC objective might value both errors similarly, but one is much worse!
- Another example:
  - You make a 5 degree heading error when turning into a lane, but keep the steering exactly straight once you're on the lane.
  - You make uncorrelated small 0.1 degree errors at every instant during driving.
  - BC objective could like both equally, but one is much worse than the other.

Going Beyond Imitation

- Imitation is often *very* useful. In most cases where you have access to expert demonstrations, you should aim to use it through some kind of imitation. But there are limitations.
- BC usually takes the short-term myopic view:
  - The BC loss is only per-timestep deviations from the expert actions.
  - It does not account for the impacts of current actions on the future.
- More broadly, imitation is limited to mimicking experts and cannot discover new solutions. What about solving new problems, like controlling a new robot, or trading on the stock market, or beating the world's best Go player?
- Reinforcement Learning (next) addresses all this more carefully. There are also ways to naturally combine imitation and RL (out of class scope).

Introducing Reinforcement Learning

Learning Through Trial and Error

The aim of RL is to learn to make sequential decisions in an environment:

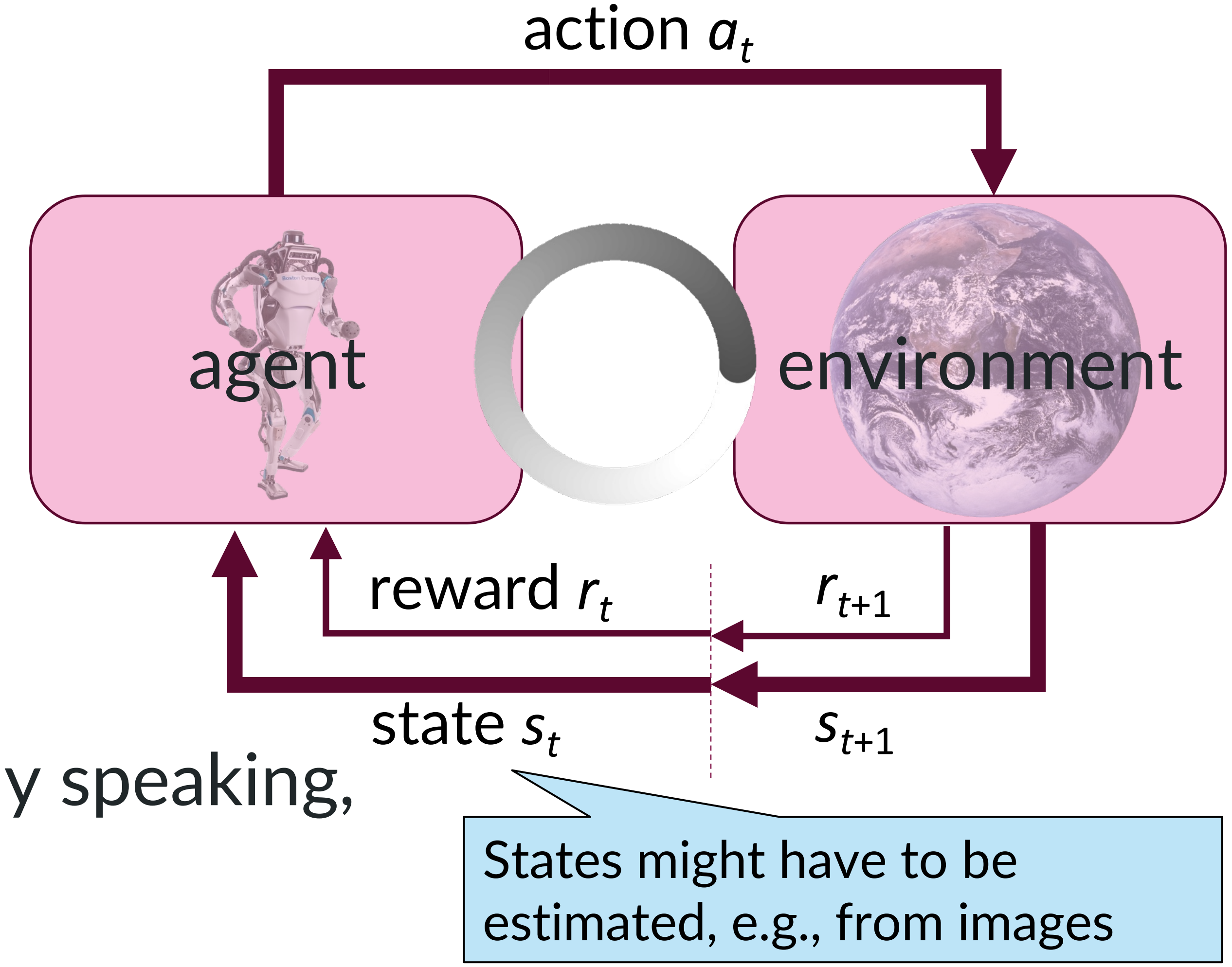
- Driving a car
- Cooking
- Playing a videogame
- Controlling a power plant
- Riding a bicycle
- Making movie recommendations
- Navigating a webpage
- Treating a trauma patient

How does an RL agent learn to do these things?

- Assume only occasional feedback, such as a tasty meal, or a car crash, or video game points.
- Assume very little is known about the "environment" in advance.
- Learn through trial and error.

The Standard Reinforcement Learning Interface

- Agent receives observations (state $s_t \in$ S) and feedback (reward r_t) from the world
- Agent takes action $a_t \in$ A
- Agent receives updated state s_{t+1} and reward r_{t+1}



- Agent's goal is to maximize, loosely speaking, "expected rewards in the future".

Image: https://robots.ieee.org/robots/pr2/

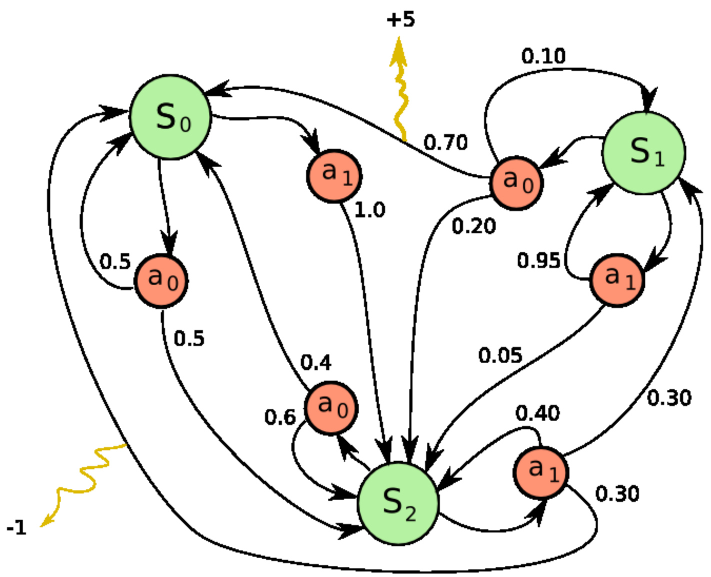
The Environment as a Markov Decision Process

An MDP (S, A, P, R, γ) is defined by:

- Set of states $s \in S$
- Set of actions $a \in A$
- Transition function $P(s' \mid s, a)$
  - Probability $P(s' \mid s, a)$ that a from s leads to s'
  - Also "dynamics model" / just "model"
- Reward function $r_t = R(s, a, s')$
- Discount factor $\gamma < 1$, expressing how much we care about the future (vs. immediate rewards)
- "utility" = *discounted* future reward sum $\sum_t \gamma^t r_{t+1}$
- Goal: maximize *expected* utility

Unknown to agent

Example



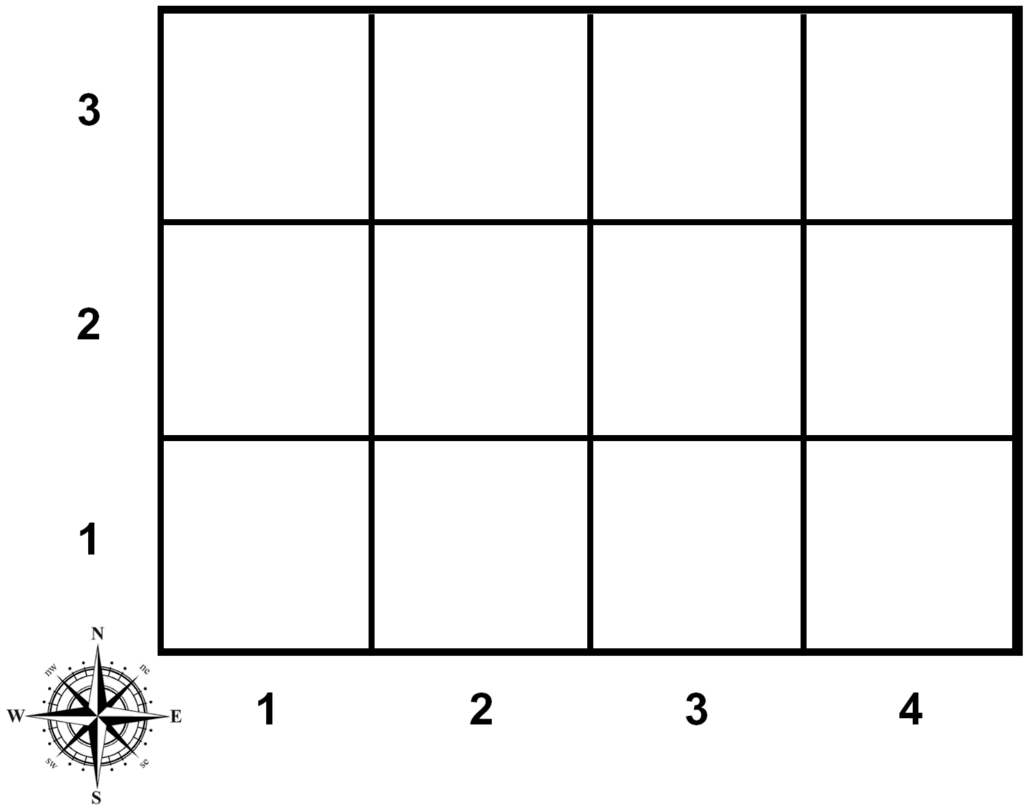
In RL , we assume no knowledge of the true functions $P(\cdot)$ or $R(\cdot)$

Sample RL environment: Grid World

- The agent's state is one cell $s = (x, y)$ within the grid $x \in \{1,2,3,4\}$ and $y \in \{1,2,3\}$.
- The agent can execute 4 actions: $a =$"N", "E", "S", "W"

For the moment, this is all that that the RL agent knows about the environment. In particular, it does not know:

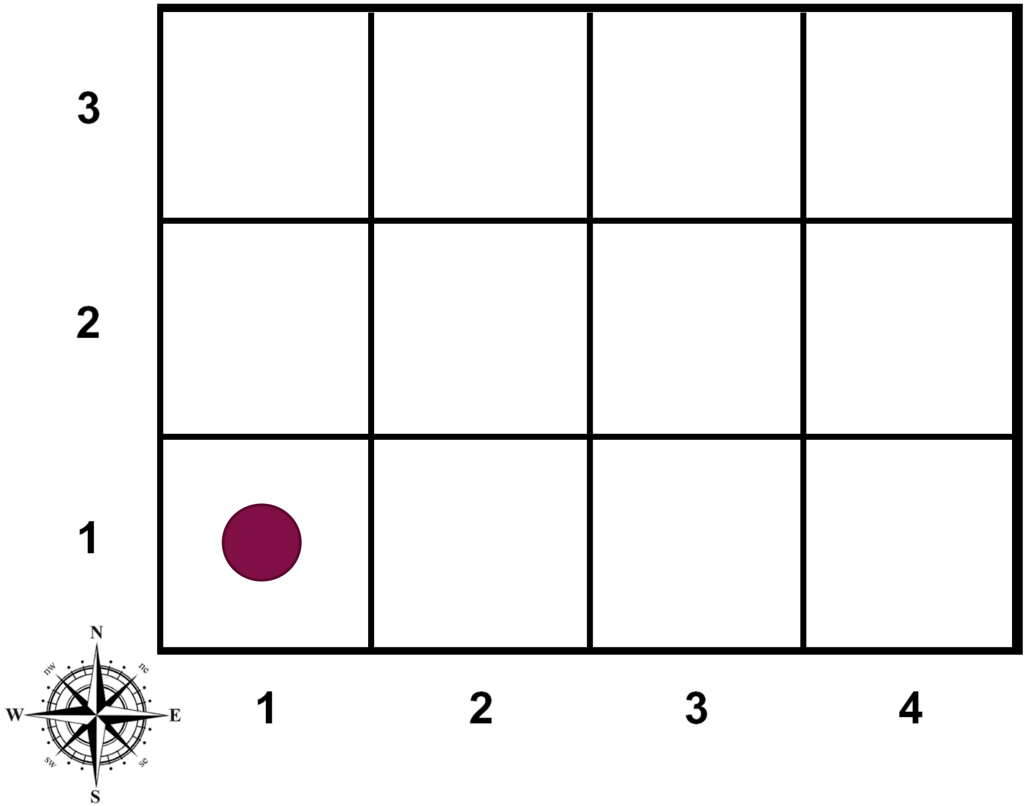
- $P(s'|s, a)$
  - Which cell would it move to, if it executes an action from a cell? (e.g. $a =$"N" from $s = (1, 2)$)
  - The result might even be non-deterministic.
- $R(s, a, s')$
  - What is the instantaneous reward it would get if it moved from $s = (1,2)$ to $s' = (1,3)$ by executing action $a =$"N"?



A random trajectory of an RL agent

Time t=1

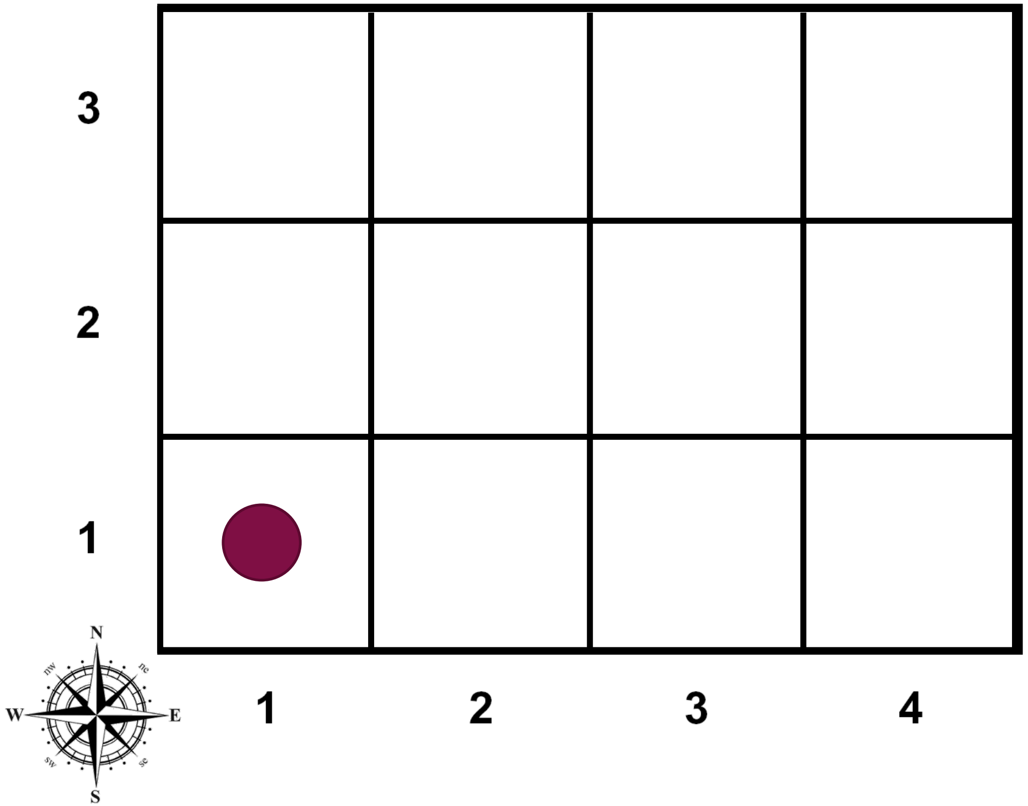
s=(1,1)



A random trajectory of an RL agent

Time t=1

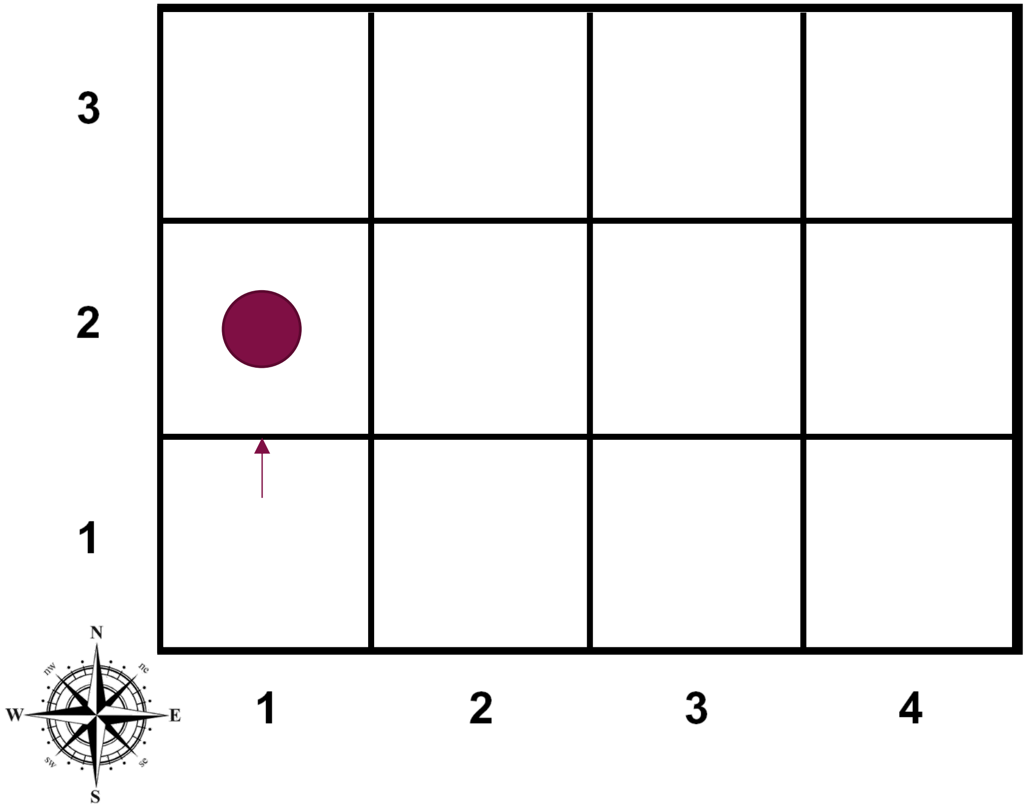
s=(1,1)
Action= “N”



A random trajectory of an RL agent

Time t=1

s=(1,1)
Action= “N”
s’=(1,2)
Reward = -0.03



Time step t=1 over

A random trajectory of an RL agent

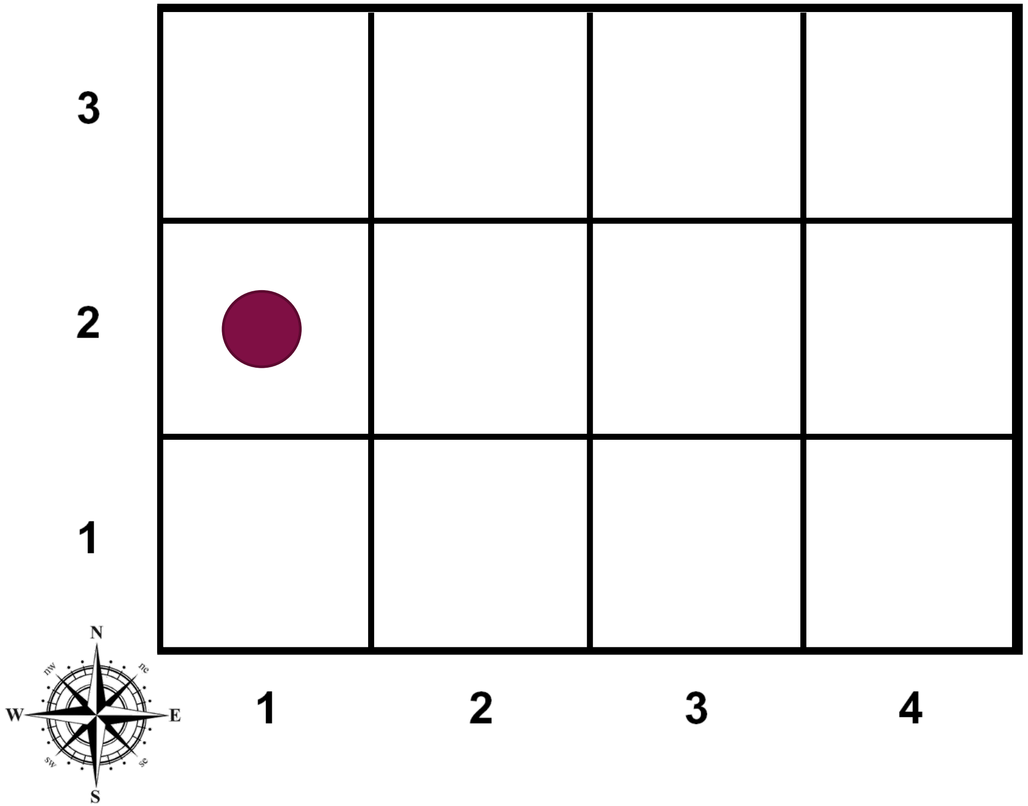
Time t=2

s=(1,2)

Action= “N”

s’=?

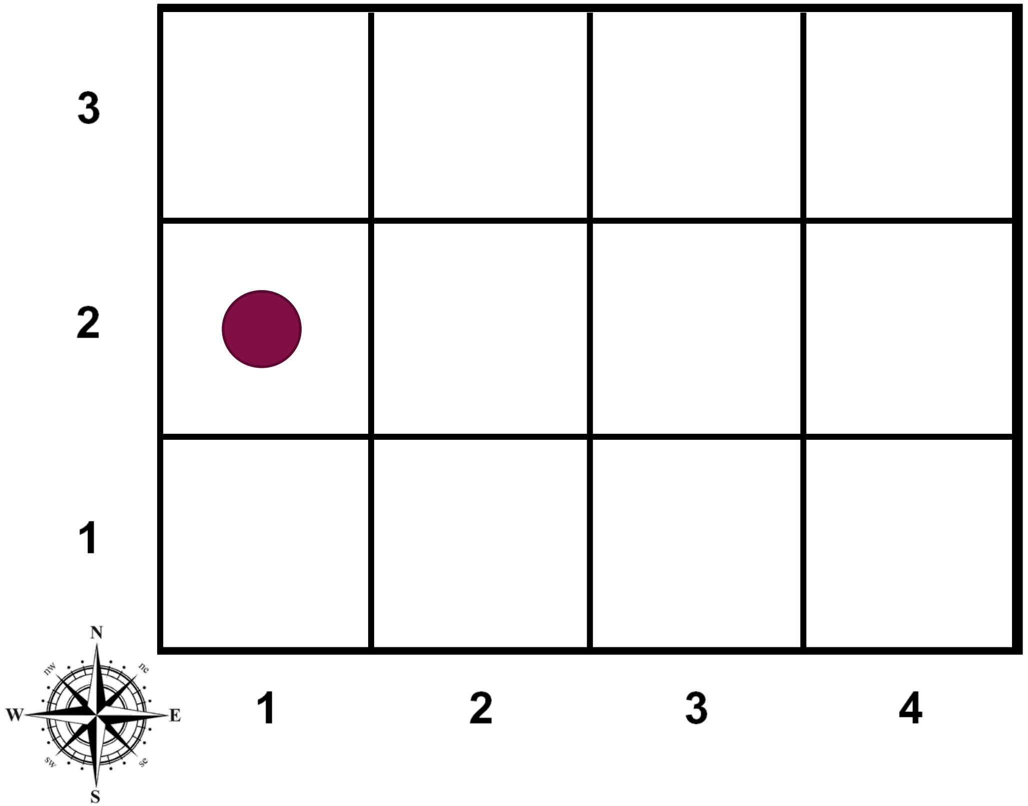
Reward = ?



A random trajectory of an RL agent

Time t=2

s=(1,2)
Action= “N”
s’=(1,2)
Reward = -0.03



Time step t=2 over

A random trajectory of an RL agent

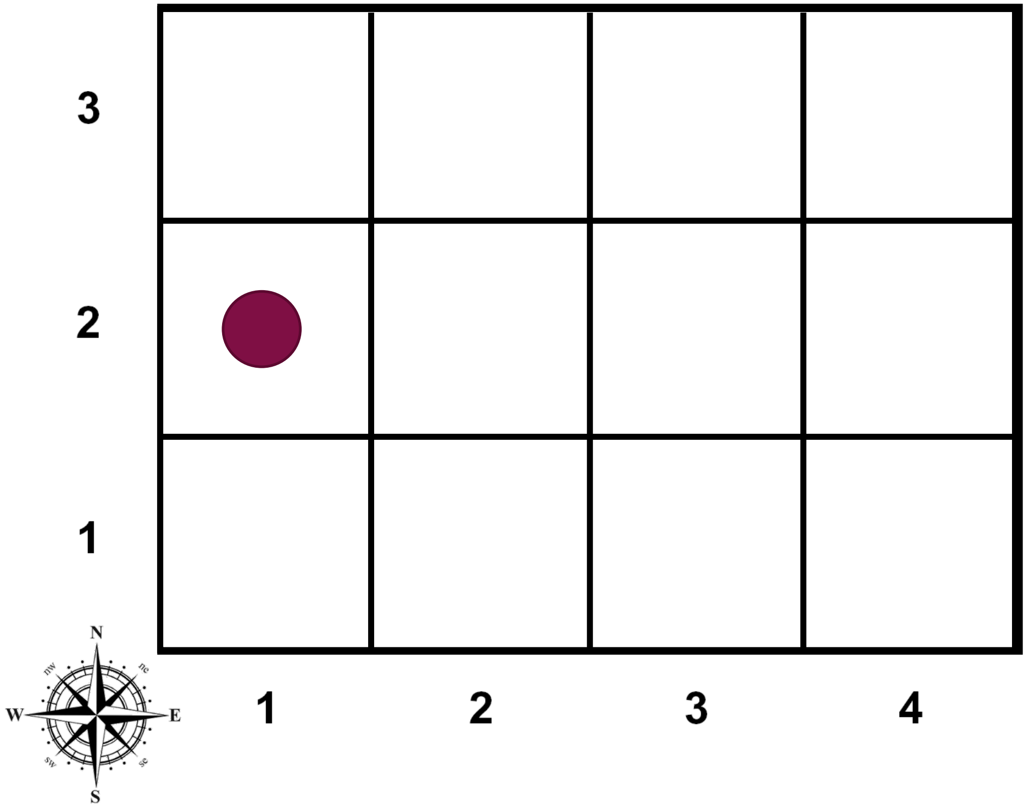
Time t=3

s=(1,2)

Action= “N”

s’=?

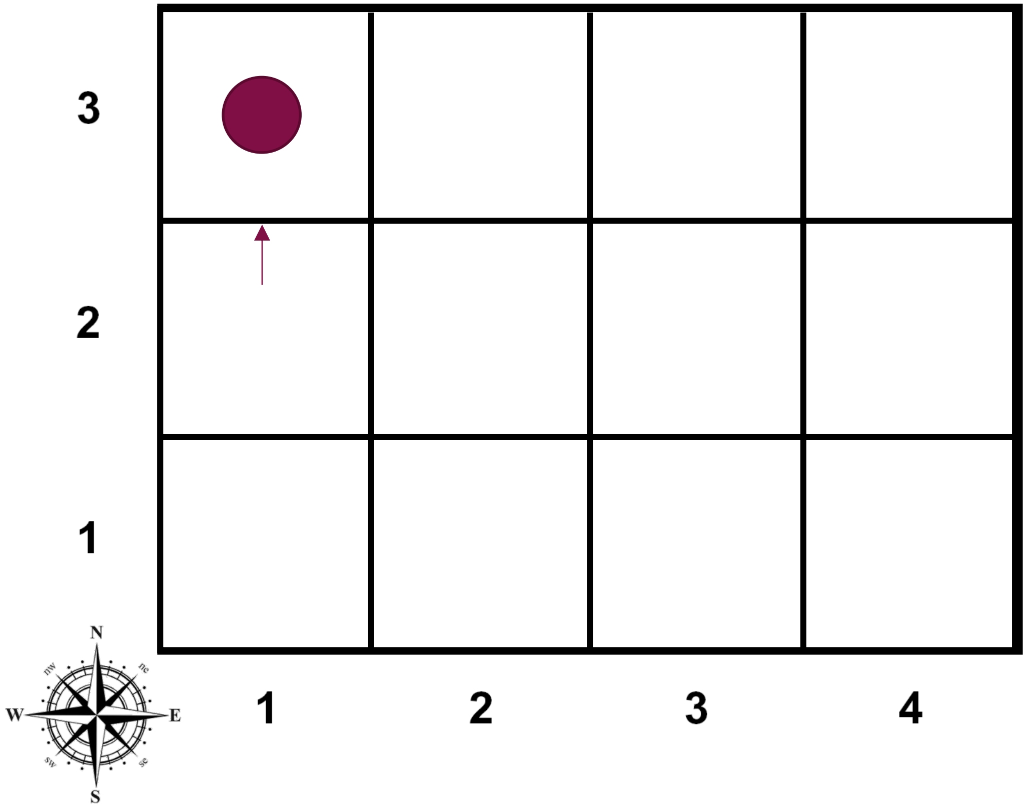
Reward = ?



A random trajectory of an RL agent

Time t=3

s=(1,2)
Action= “N”
s’=(1,3)
Reward = -0.03

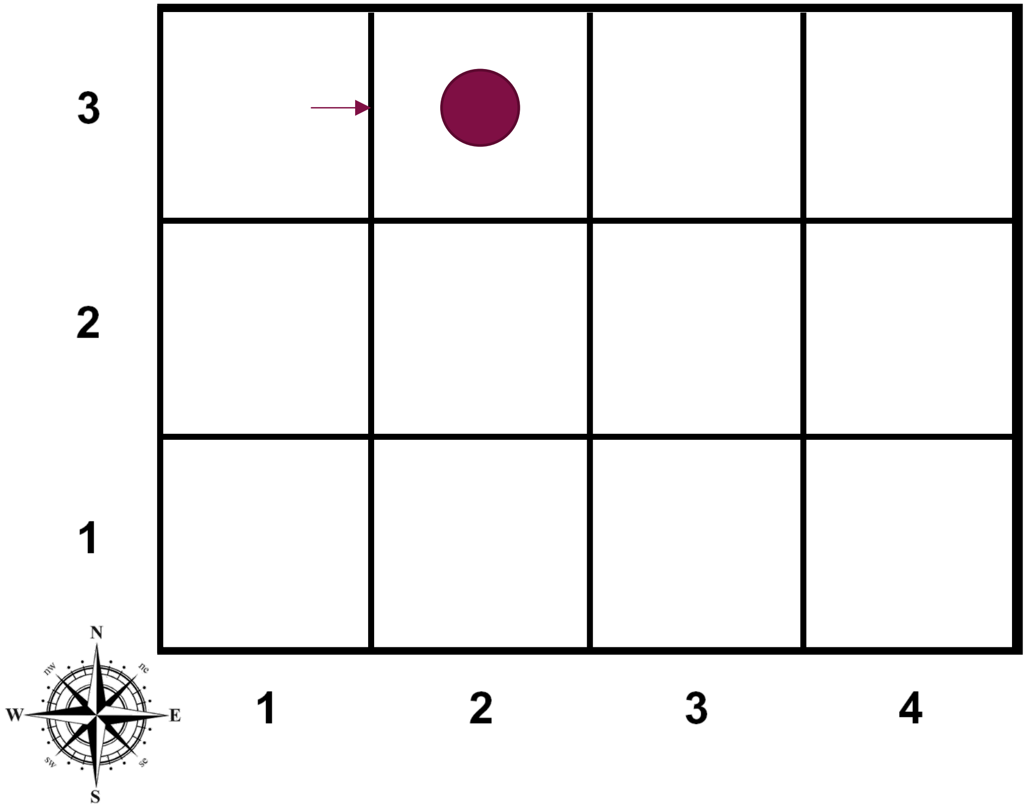


Time step t=3 over

A random trajectory of an RL agent

Time t=4

s=(1,3)
Action= “N”
s’=(2,3)
Reward = -0.03

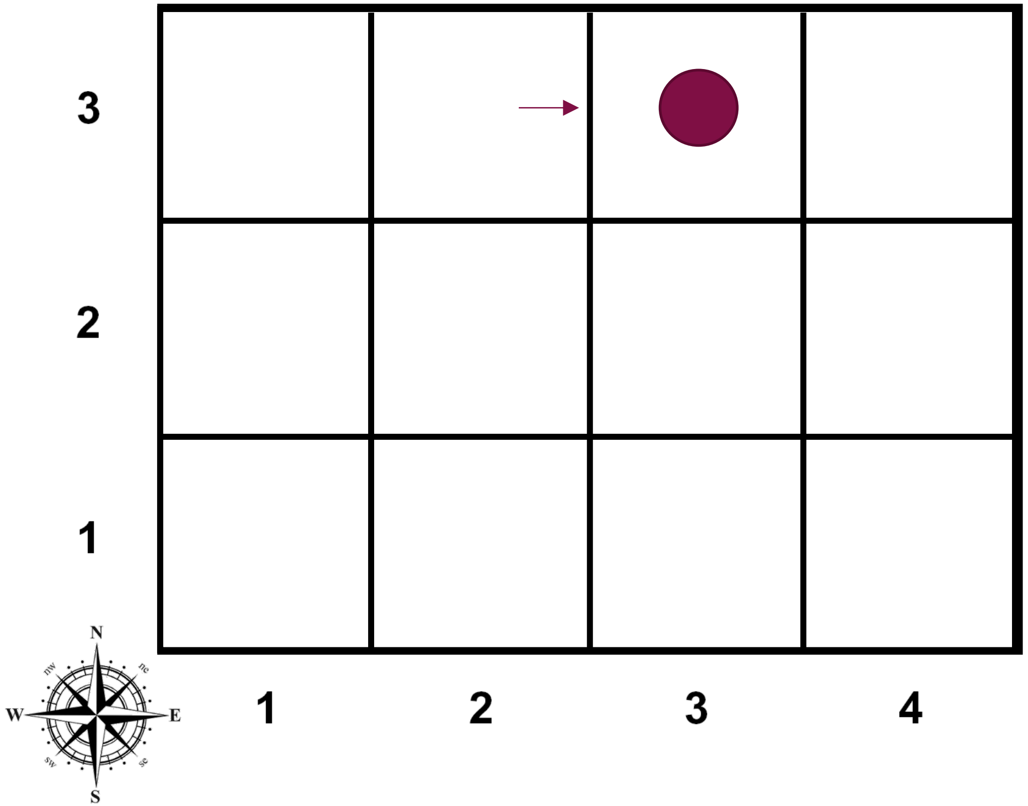


Time step t=4 over

A random trajectory of an RL agent

Time t=5

s=(2,3)
Action= “E”
s’=(3,3)
Reward = -0.03

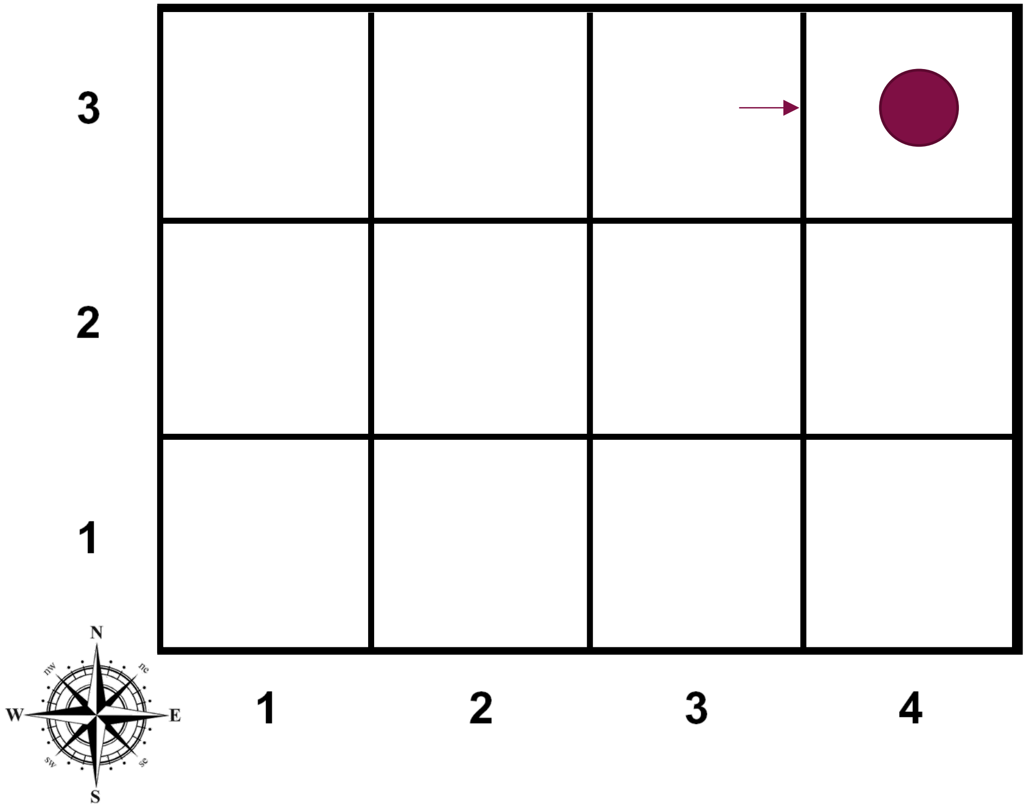


Time step t=5 over

A random trajectory of an RL agent

Time t=6

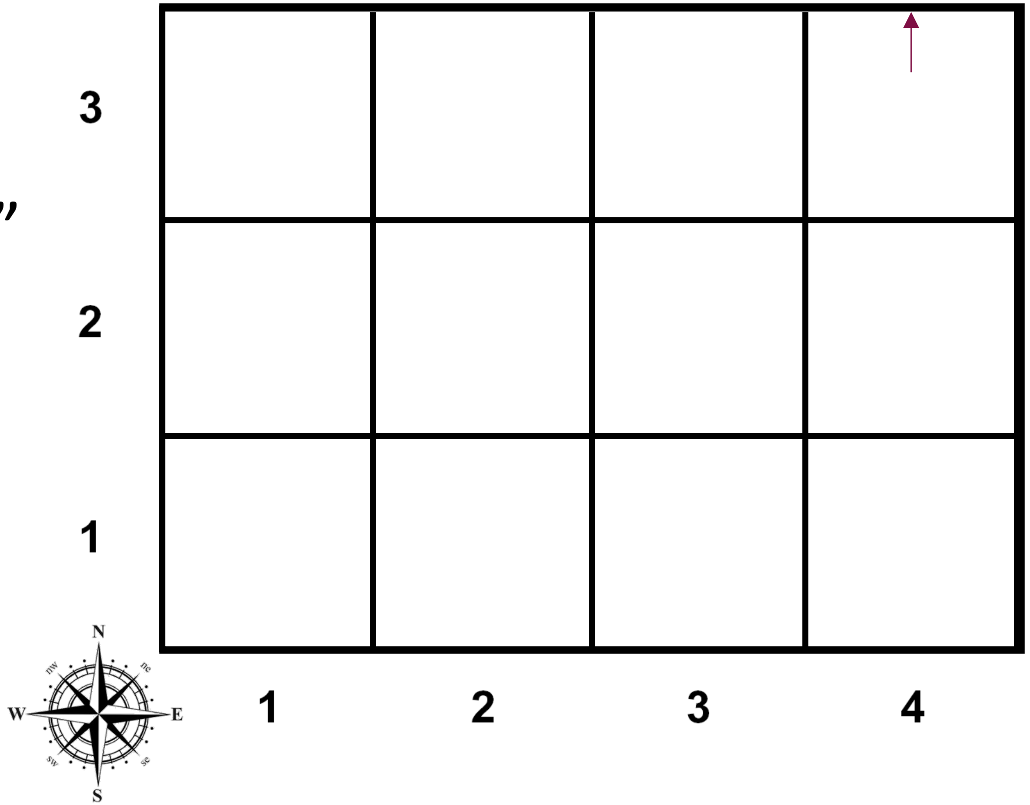
s=(3,3)
Action= "E"
s'=(4,3)
Reward = -0.03



Time step t=6 over

A random trajectory of an RL agent

s=(4,3)
Action= "N"
s'= special state "END"
Reward = +1



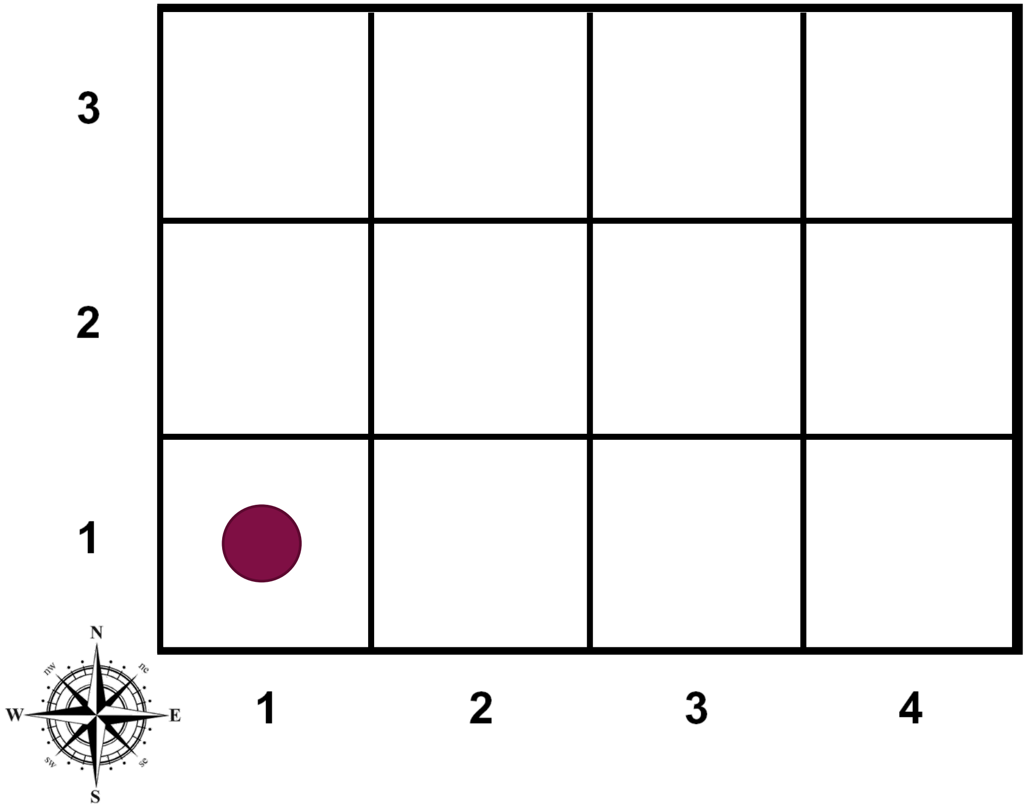
One "episode" is over.

Next, the agent respawns in the environment. "Reset"

Reset

Another episode begins!

s=(1,1)
Action=?
s'= ?
Reward = ?



So, can we maximize rewards in this environment?

- What have we learned about this environment after having acquired this experience?
  - Do we know something about P, R?
  - Do we know how to act optimally now?

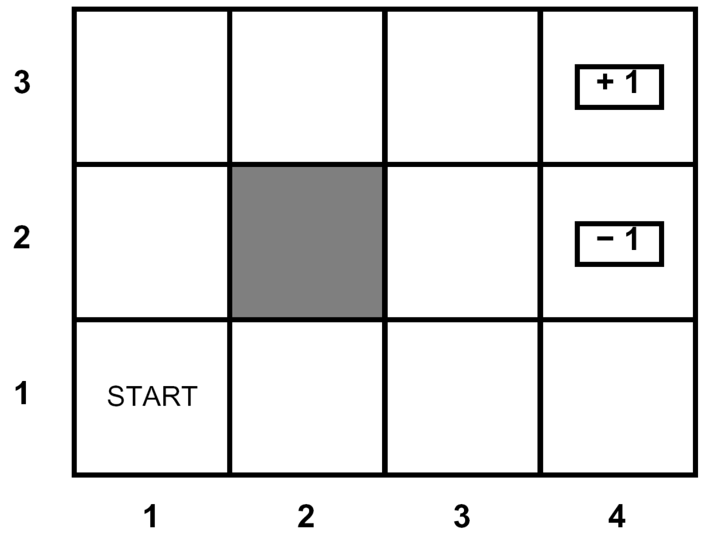
We have learned some things, but there is still far too much ambiguity.

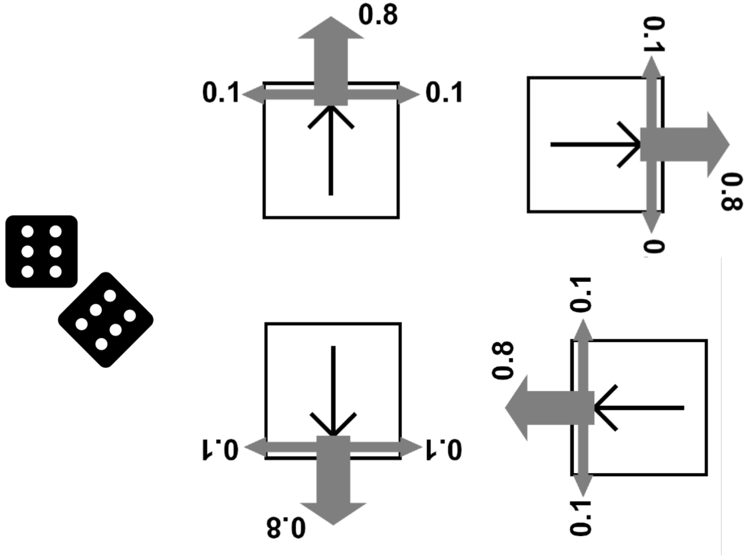
Perhaps with more experience …

Provided sufficient experience, RL algorithms can learn optimal policies!

Behind The Scenes: The Full Environment

- A grid map with solid / open cells. Agent moves between open cells.
- From terminal states $(4{,}3)$ and $(4{,}2)$, any action ends the episode, and results in a +1/-1 reward respectively.
- For each timestep outside terminal states , the agent pays a small "living" cost (negative reward): -0.03
- The agent actions N, E, S, W correspond to North, East, South, West
  - **But the outcomes of actions are not deterministic!**
    - The chosen motion direction is attempted 80% of the time
    - 10% of the time, the agent instead executes a different direction 90° off. Another 10% of the time, -90° off.
    - E.g. an agent surrounded by open cells and executing action N will end up in the northern cell 80% of the time, in the eastern cell 10% of the time, and in the western cell 10% of the time.
  - **The agent stays put if it attempts to move into a solid cell** or outside the world. (Imagine the map is surrounded by solid cells)
- Goal: As always, maximize the sum of discounted future rewards within an episode

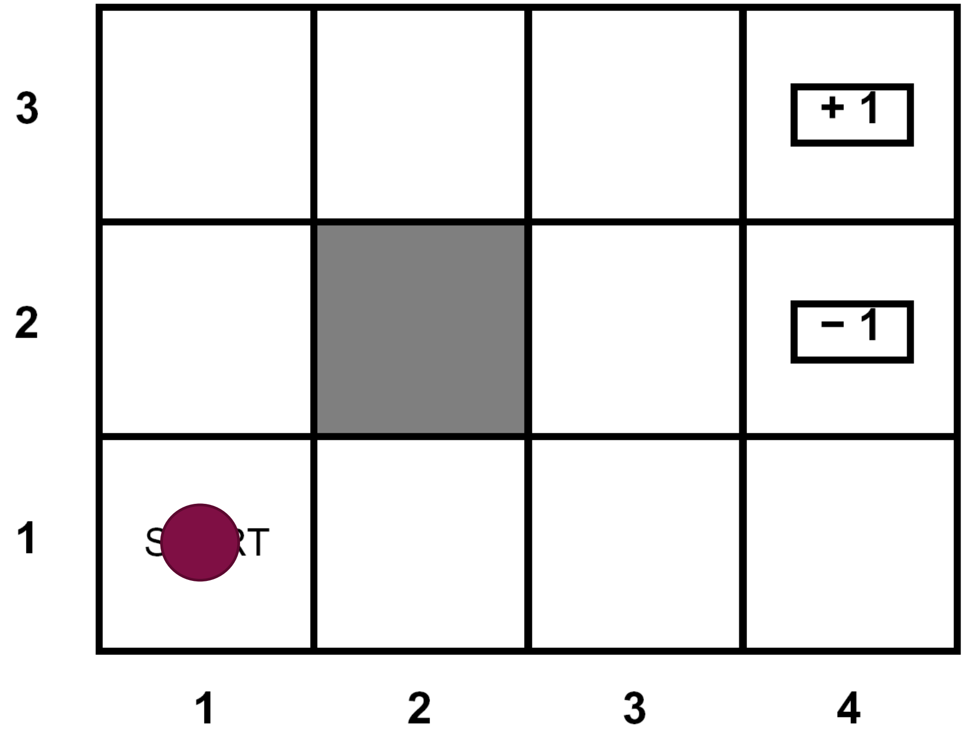


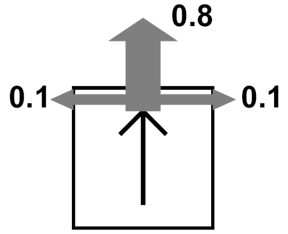


What actually happened in that episode?

- Now that we have seen the full environment, let’s view a replay with all this extra information to see what actually happened during that one episode of experience we saw before.

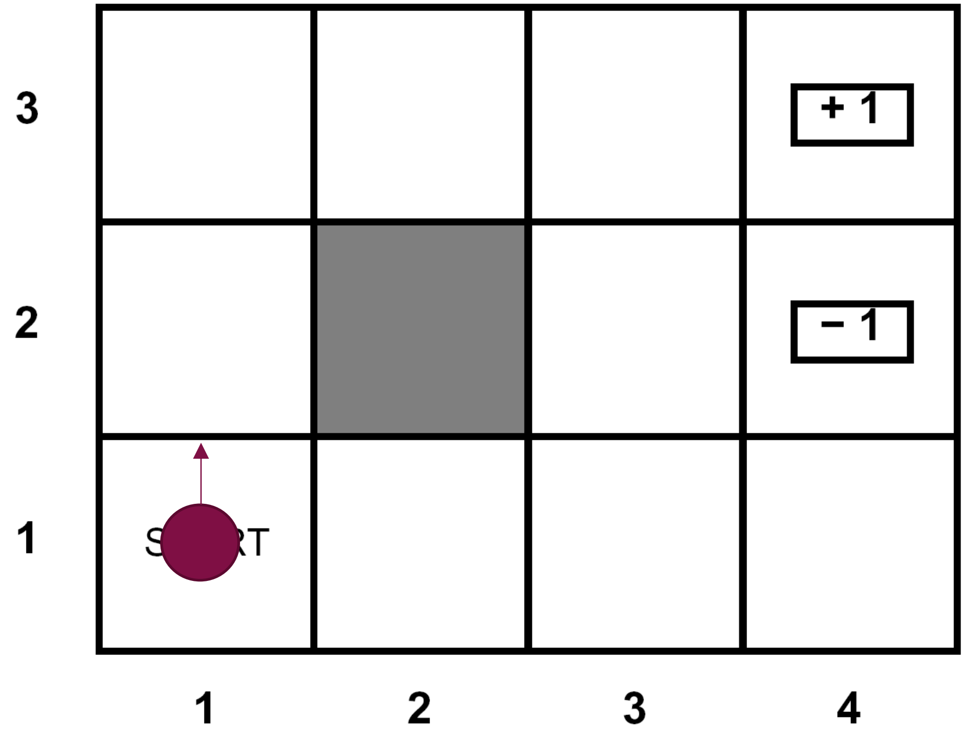
What actually happened in that episode?

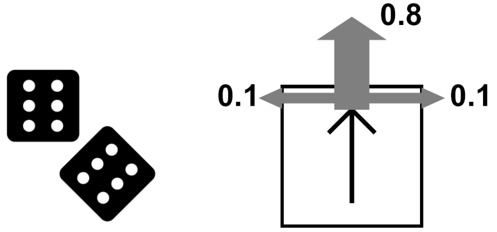




What actually happened in that episode?

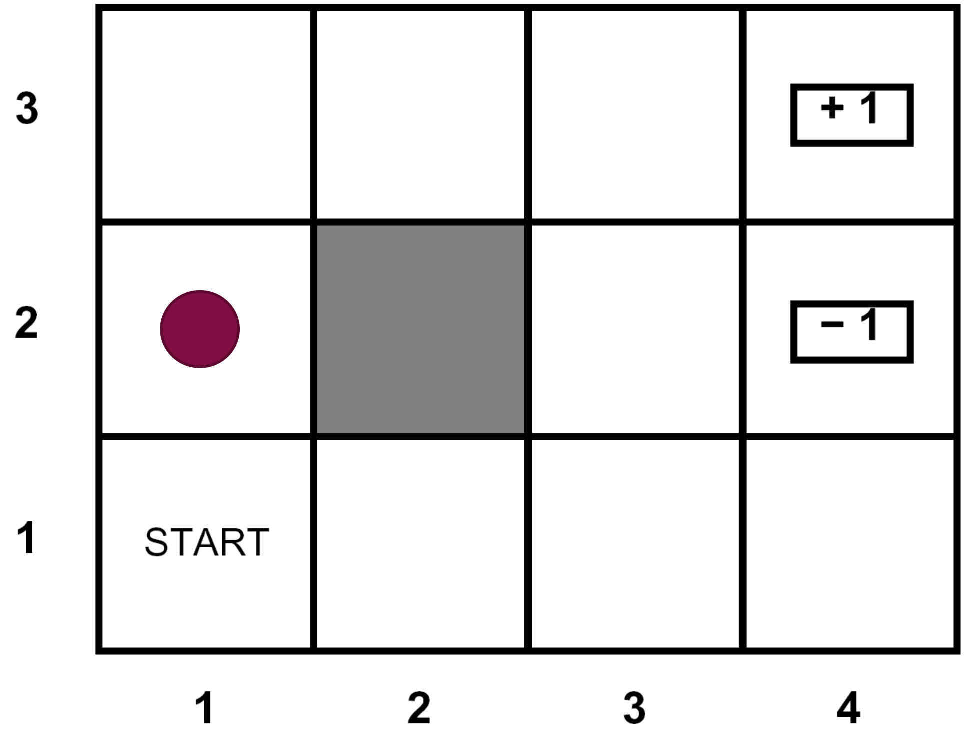
Action= “N”

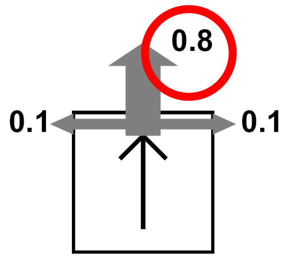




What actually happened in that episode?

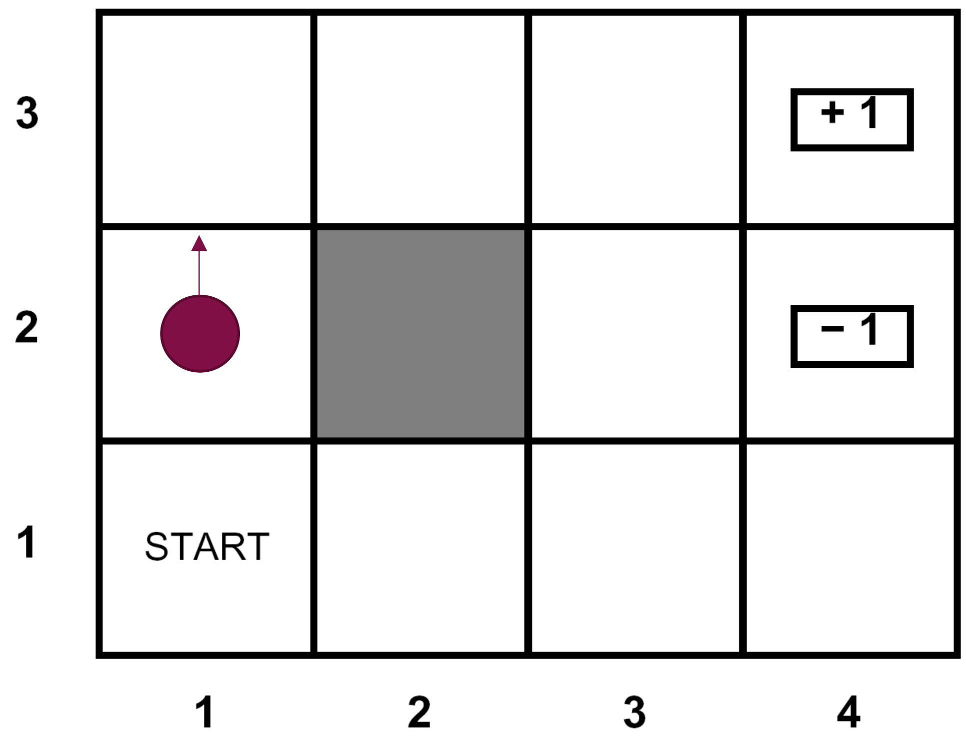
Action= “N”
Attempted motion = “N”
Reward = -0.03

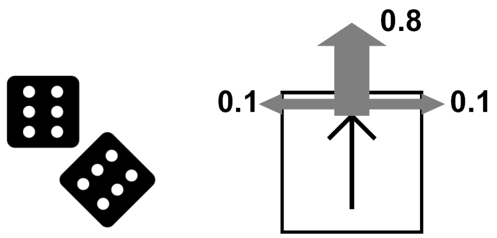




What actually happened in that episode?

Action= “N”

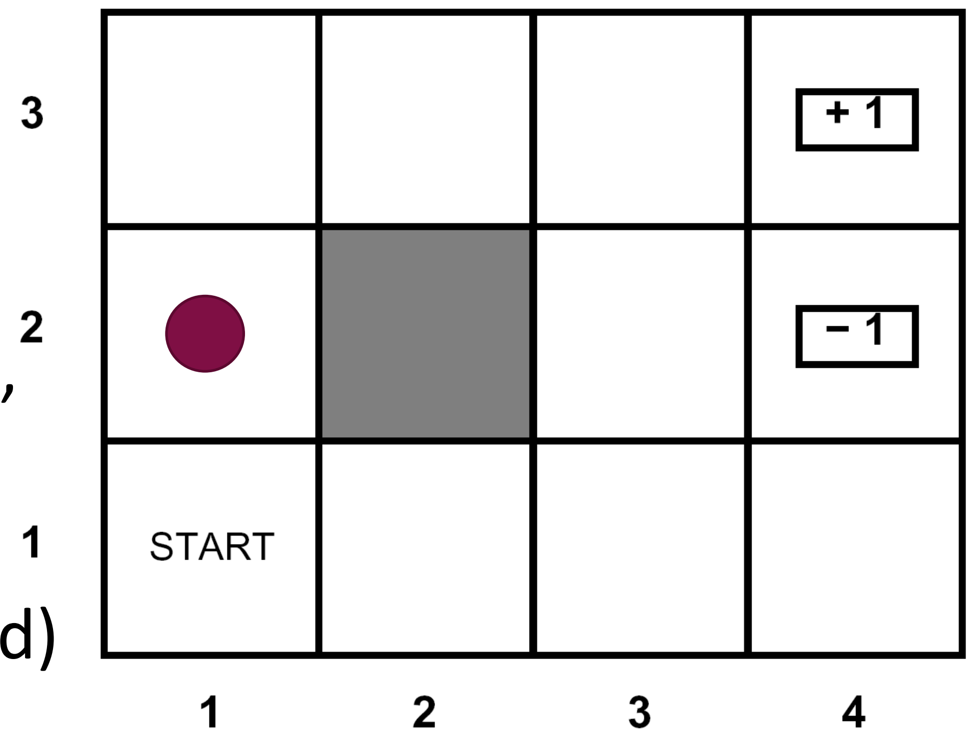


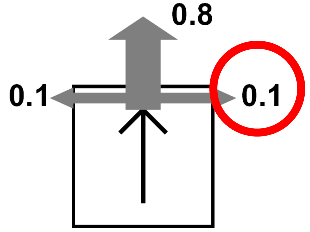


What actually happened in that episode?

Action= “N”
Attempted Motion=“E”
Reward = -0.03

(stays still because blocked)



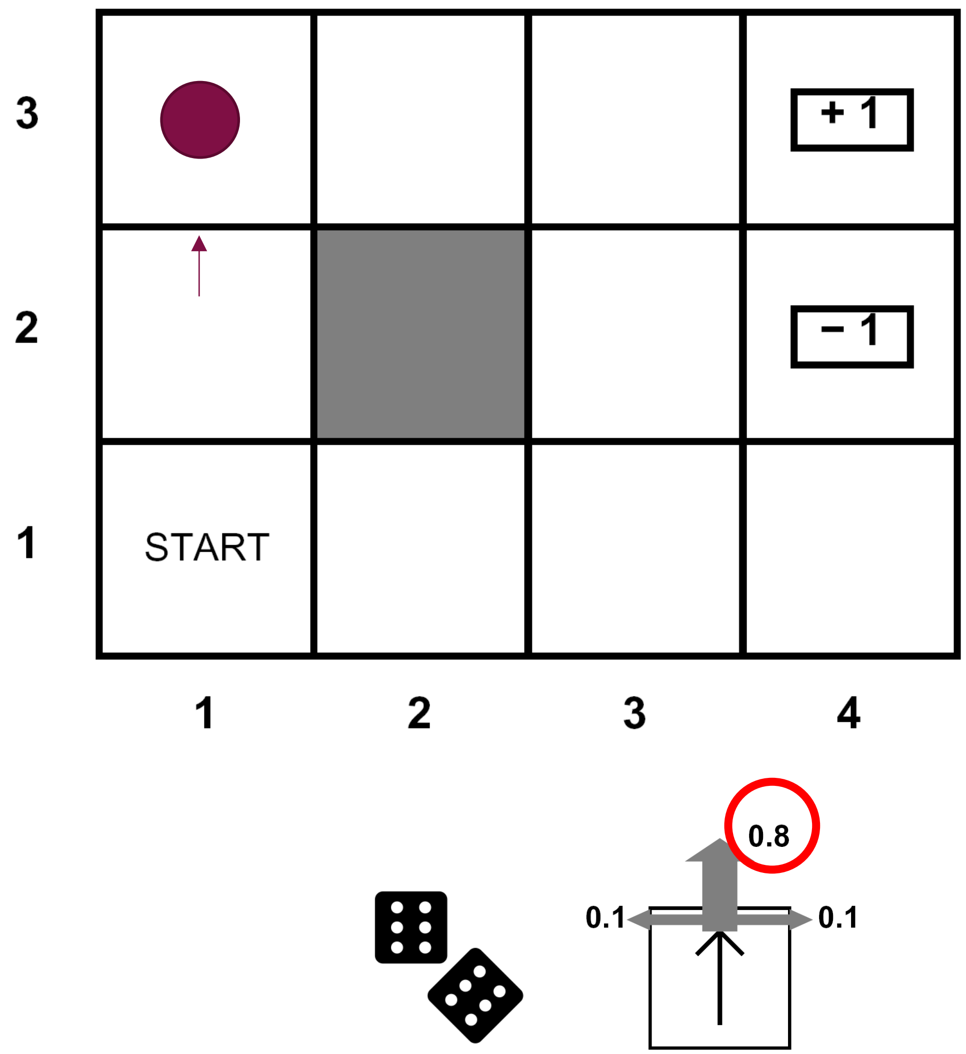


What actually happened in that episode?

Action= “N”

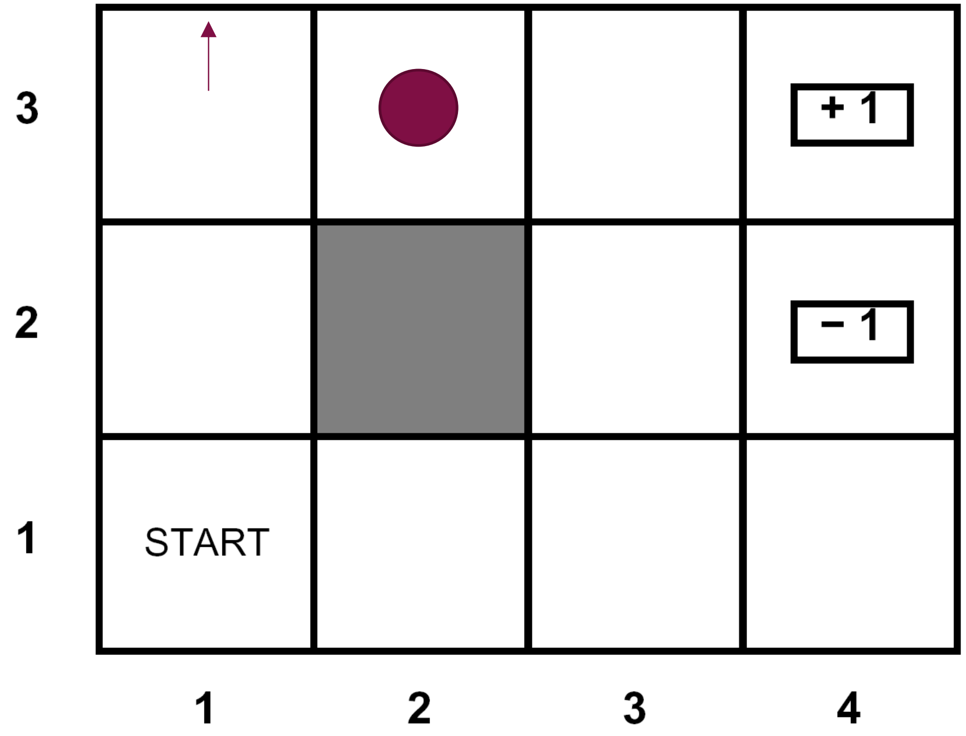
Attempted Motion=“N”

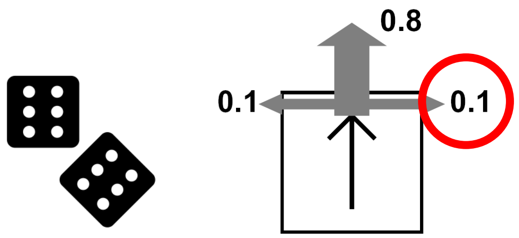
Reward = -0.03



What actually happened in that episode?

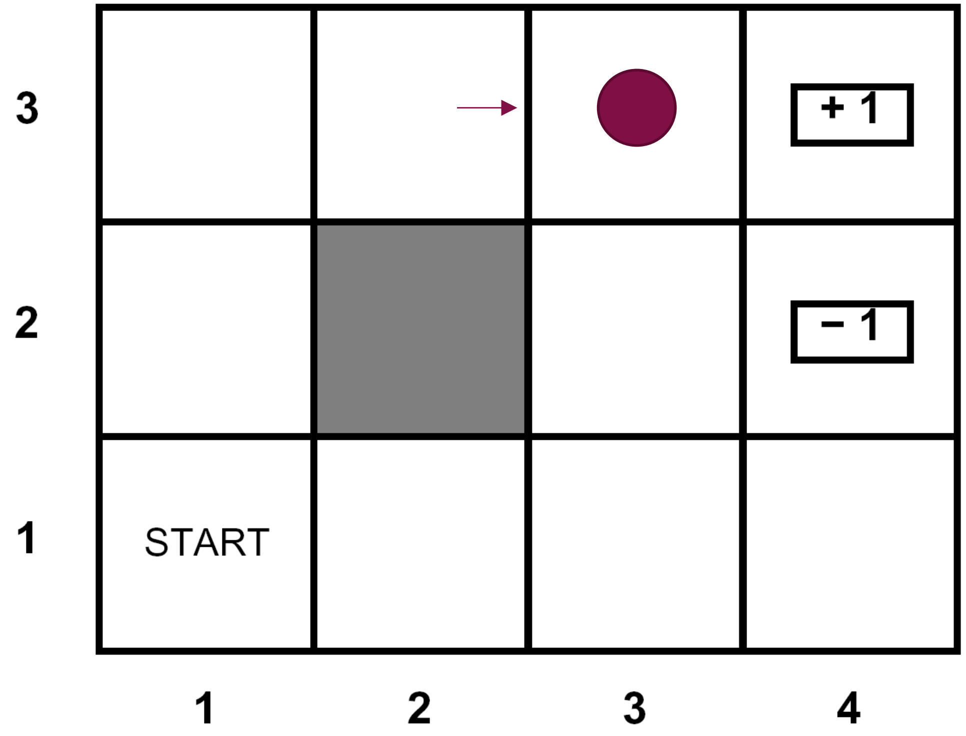
Action= “N”
Attempted Motion=“E”
Reward = -0.03

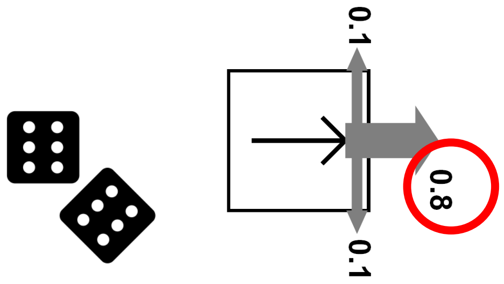




What actually happened in that episode?

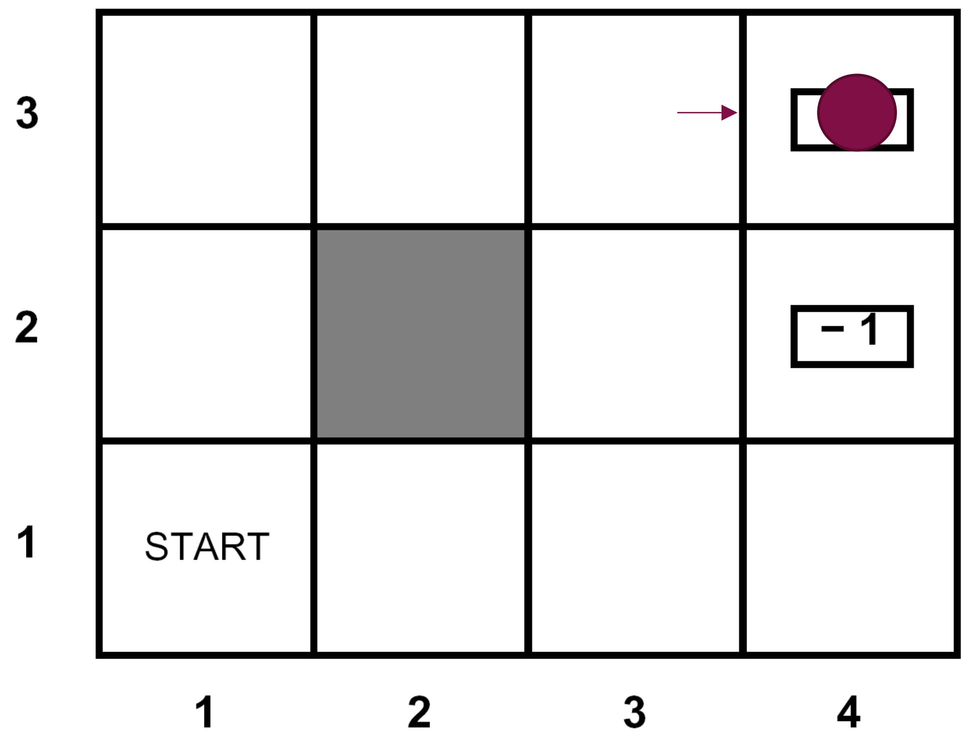
Action= “E”
Attempted Motion=“E”
Reward = -0.03

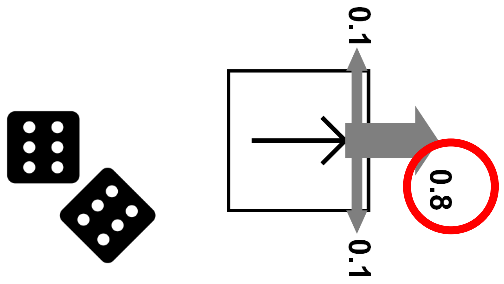




What actually happened in that episode?

Action= “E”
Attempted Motion=“E”
Reward = -0.03





What actually happened in that episode?

Action= "N"

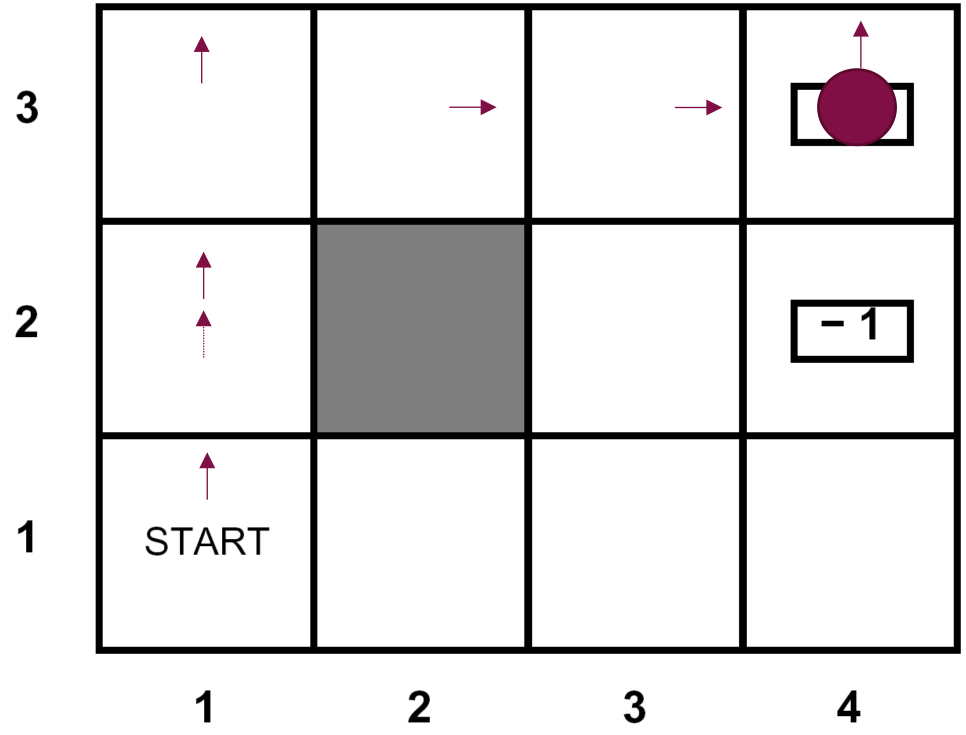
Attempted Motion: ?

Result="**the end**"

Reward = +1

It so happened that our random trajectory did end up at the right place!

Was this action sequence "optimal?" No



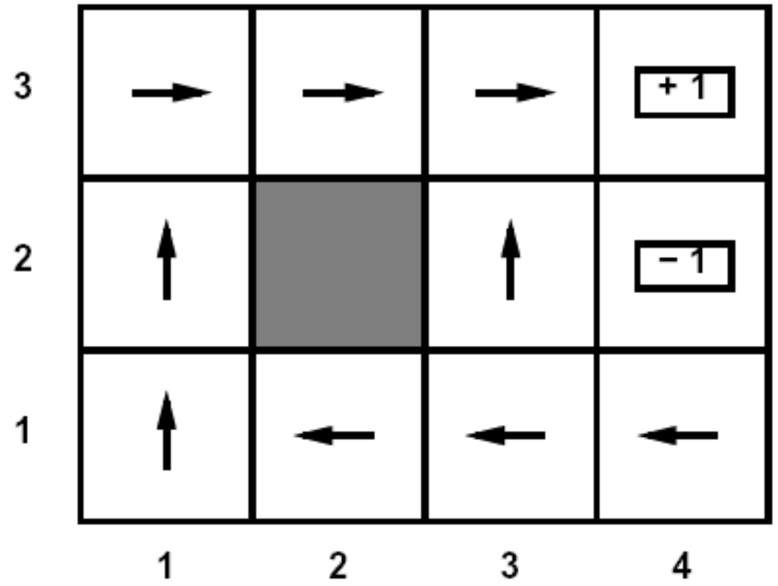
Note: this corresponds to saying: "when $s = (4{,}3)$, for any a, the reward is $R(s, a, s') = R(s) = +1$".
This is meaningfully different from: "when $s' = (4{,}3)$, the reward is $R(s, a, s') = R(s') = +1$ for any s, a."

Desired Outcome of RL: Optimal Policies

Goal: given some environment, find the optimal policy $\pi^*(s): S \to A$

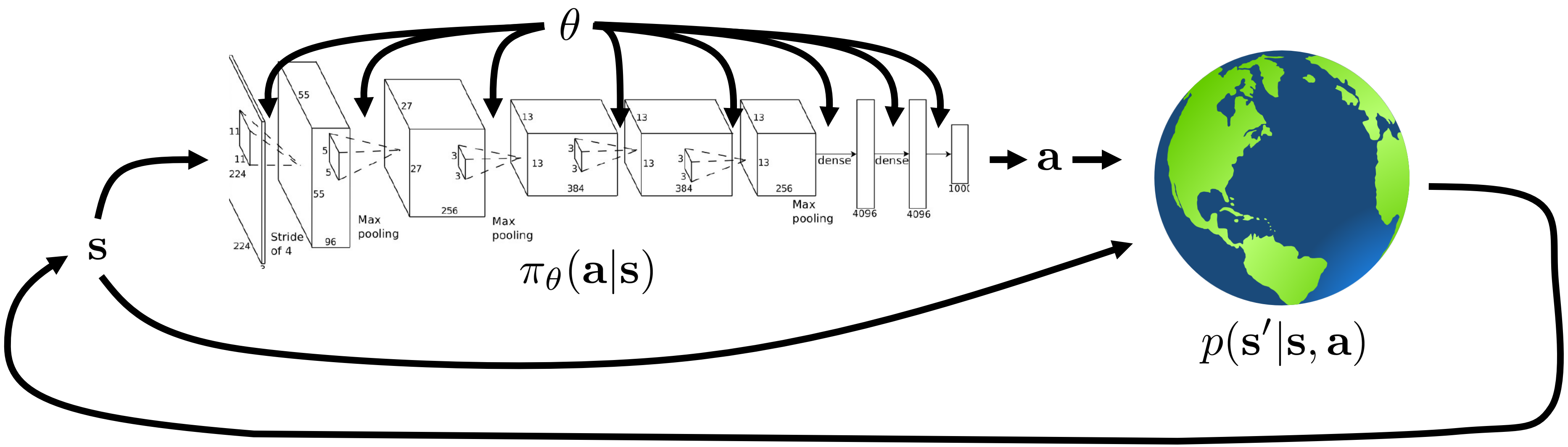
- "Optimal" $\Longrightarrow$ Following π^* maximizes expected utility

Example optimal policy π^*



Optimal policy when living cost is
$$R(s, a, s') = R(s) = -0.03$$
for all non-terminal states s

Goal of RL: Learn Optimal Policies



Trajectory distribution

$$\underbrace{p_\theta(\mathbf{s}_1,\mathbf{a}_1,\ldots,\mathbf{s}_T,\mathbf{a}_T)}_{p_\theta(\tau)\text{ or }\pi_\theta(\tau)} = p(\mathbf{s}_1)\prod_{t=1}^{T}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)$$

Optimal policy parameters

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t \gamma^t r(s_t,a_t)\right]$$

e.g. state s_t = robot pose, action a_t= motor torques, r_t= running speed

Why discounts?

Idea: future rewards are worth exponentially less than current rewards.

- They are less certain

Future rewards are discounted by $0 < \gamma < 1$:

$$\sum_{t=0}^{\infty} \gamma^t r_{t+1}$$

discounted cumulative future reward / "utility"

Future rewards matter less to the decision than more recent rewards

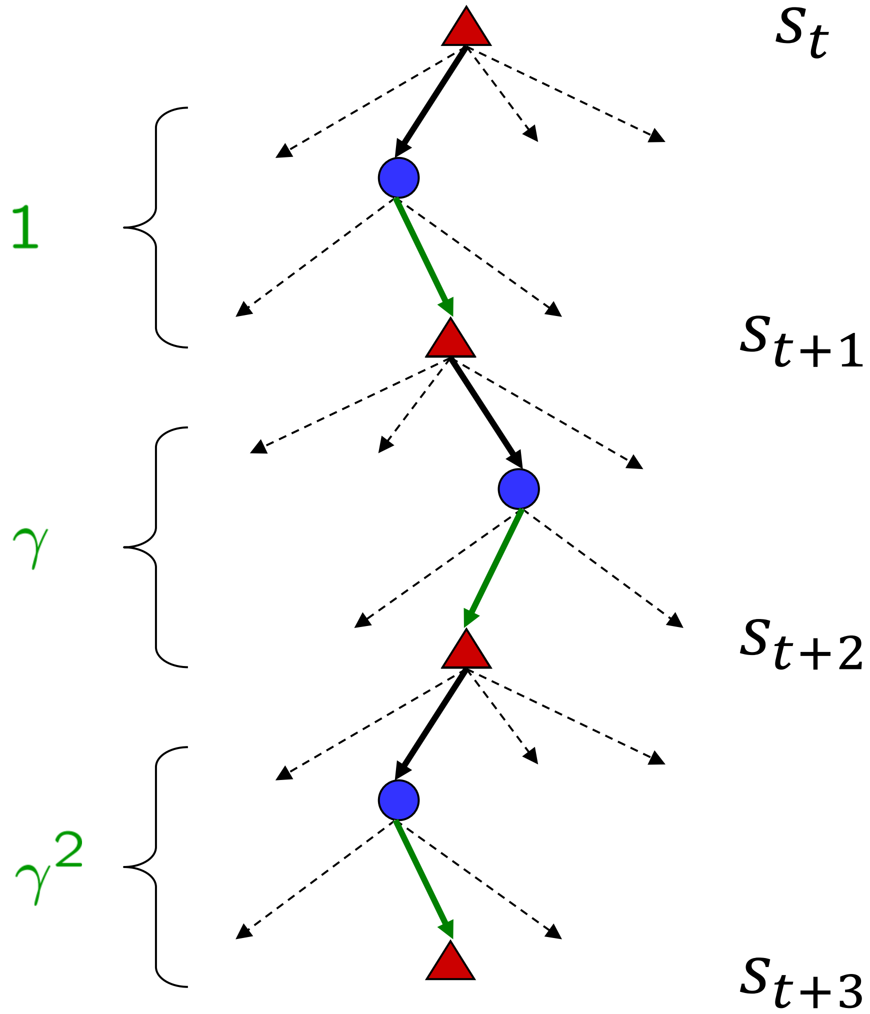


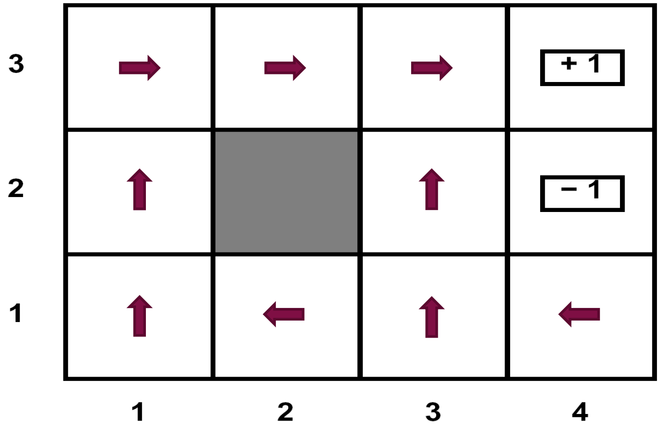
Image by Dan Klein

Also very useful for theoretical analysis

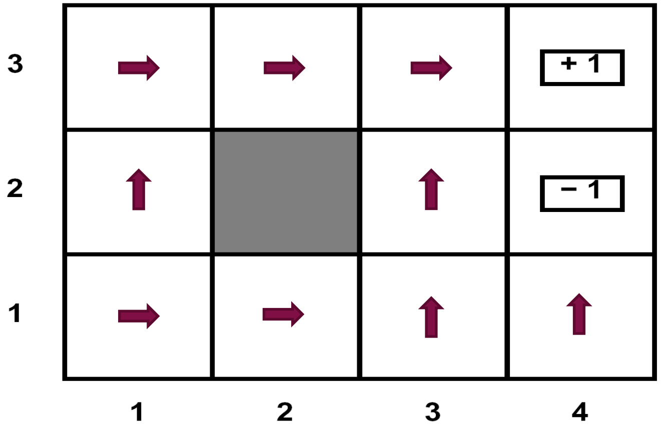
Sensitivity of Optimal Policy To R And γ

The task specification through R (and γ) is critical!

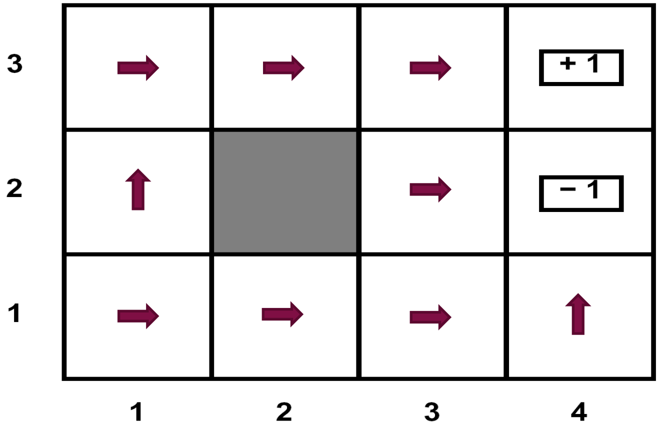
$R(s,a,s') = 0$



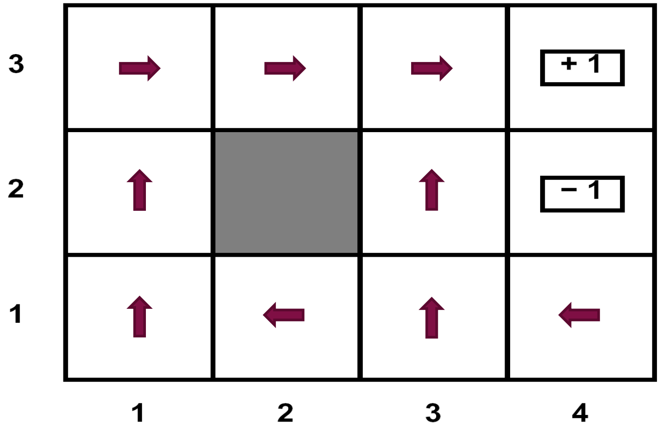
$R(s,a,s') = -1$



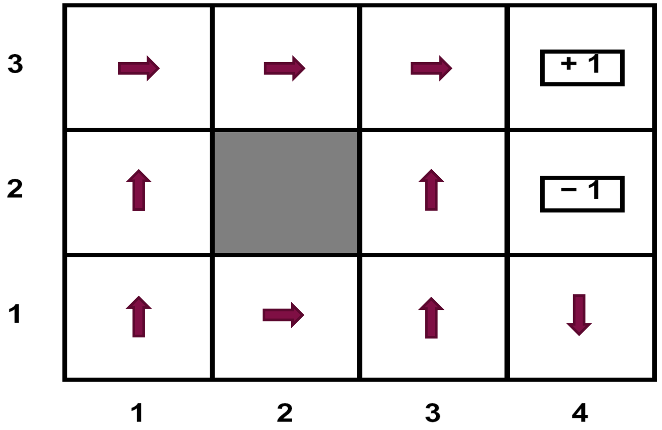
$R(s,a,s') = -2$



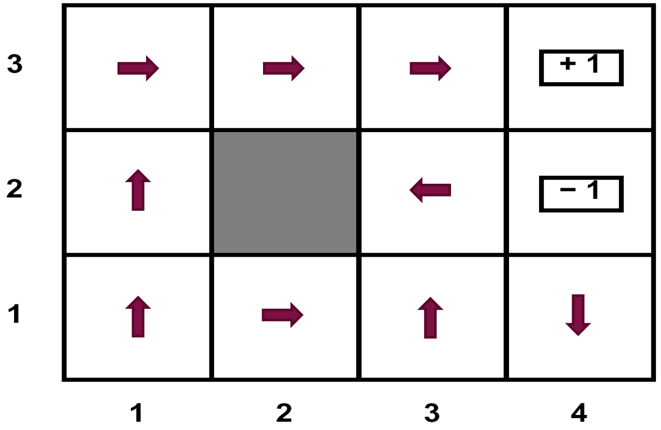
$\gamma = 0.9$



$\gamma = 0.5$



$\gamma = 0.1$



Warning: “Reward Hacking”

- Reward functions as task specifications can be surprisingly hard to get right!

```
def reward_function(params):
    '''
    A complex reward function for a robot arm reaching a specific target position and
orientation.
    '''
    # Set up the target position and orientation
    target_pos = [0.5, 0.5, 0.5]
    target_orient = [0.0, 0.0, 0.0, 1.0]

    # Get the current position and orientation of the robot arm
    robot_pos = params['position']
    robot_orient = params['orientation']

    # Calculate the distance to the target position and orientation
    pos_diff = math.sqrt((robot_pos[0] - target_pos[0])**2 + (robot_pos[1] -
target_pos[1])**2 + (robot_pos[2] - target_pos[2])**2)
    orient_diff = np.linalg.norm(np.subtract(robot_orient, target_orient))

    # Penalize the robot for being too far away from the target position or orientation
    if pos_diff > 0.1 or orient_diff > 0.1:
        reward = -1.0
    else:
        # Calculate a reward based on the proximity to the target position and orientation
        pos_reward = (1.0 - pos_diff) ** 2
        orient_reward = (1.0 - orient_diff) ** 2

        # Penalize the robot for moving too much
        movement_penalty = params['speed'] * 0.01

        # Combine the rewards and penalties to get the final reward
        reward = (pos_reward + orient_reward) - movement_penalty

    return reward
```

How is RL Different from Supervised Learning (SL)?

SL: Find $h(x): X \rightarrow Y$, that minimizes a loss L over training (x, y) pairs

RL: Find $\pi(s): S \rightarrow A$ that maximizes expected utility

Supervised Learning

- Target labels for h are directly available in the training data
- Train to map (regress/classify) from x to y in the training data

Reinforcement Learning

- Optimal action labels a for states s are not given to us. No predefined solutions!
- Train by trying various action sequences in an environment, and observing which ones produce good rewards over time.

Key Problems Specific to RL:

- **Credit assignment:** Which actions in a sequence were the good/bad ones?
- **Exploration vs Exploitation:** Yes, trial-and-error, but smartly pick what to try?