Business Analytics & Machine Learning

Reinforcement Learning

Prof. Dr. Martin Bichler

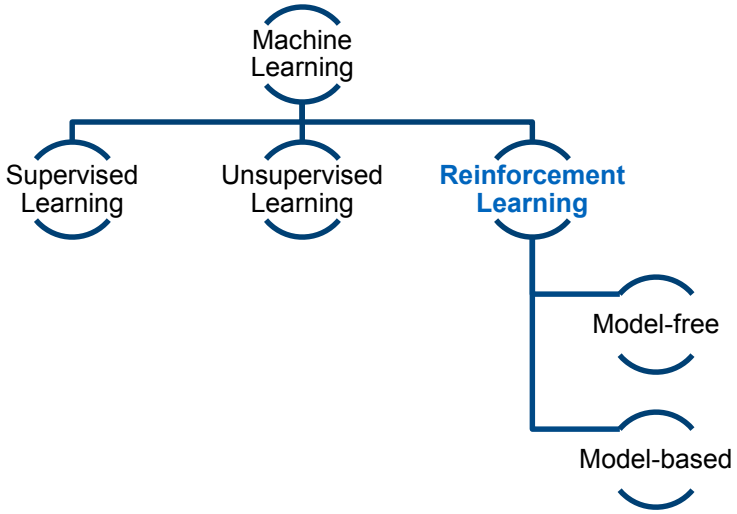
Department of Computer Science

School of Computation, Information, and Technology

Technical University of Munich

Course Content

- Introduction
- Regression Analysis
- Regression Diagnostics
- Logistic and Poisson Regression
- Naive Bayes and Bayesian Networks
- Decision Tree Classifiers
- Data Preparation and Causal Inference
- Model Selection and Learning Theory
- Ensemble Methods and Clustering
- Dimensionality Reduction
- Convex Optimization
- Neural Networks
- **Reinforcement Learning**



Reinforcement Learning

Reinforcement Learning incorporates time into learning.

Learning to make sequential decisions in an environment so as to maximize some notion of overall ***rewards*** acquired along the way.

- A scalar value obtained from the environment
- It can be positive or negative, large or small
- The purpose of reward is to tell our agent how well they have behaved.

reinforcement = reward or **reinforced** behavior

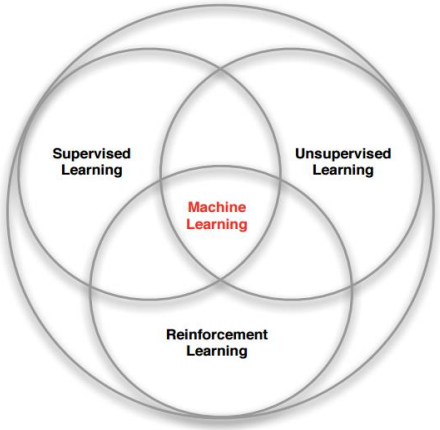
RL within the ML Spectrum

What makes RL different from other ML paradigms ?

- No supervision, just a reward signal from the environment
- Feedback is sometimes delayed (example: time taken for drugs to take effect)
- Time matters - sequential data
- Feedback - agent's action affects the subsequent data it receives (not i.i.d.)

Challenges:

- Credit assignment: which decisions were responsible for the reward?
- Exploration vs. exploitation: how to pick what to try in trial and error?



Markov Process

- **System**: Weather in Munich
- **States**: We can observe the current day as sunny or rainy
- **History**: A sequence of observations over time forms a chain of **states**, such as [sunny, sunny, rainy, sunny, …],
- For a given system we observe **states**
- The system changes between states according to some dynamics
- We do not influence the system just observe
- There are only a finite number of states (could be very large)
- Observe a sequence of states or a chain => **Markov process (or chain)**

Markov Process

A system is a **Markov Process**, if it fulfils the **Markov property**.

$p(x_i \mid x_1 \dots x_{i-1}) \equiv p(x_i \mid x_{i-1})$

The future system dynamics from any state have to depend only on this state.

Only the current state is required to model the future dynamics of the system, not the whole history or the last n states.

"The future is independent of the past, given the present."

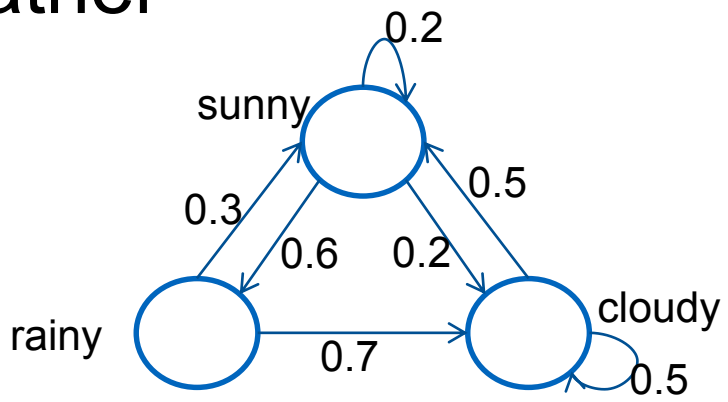
Weather example: Assume that the probability of a sunny day followed by rainy day is independent of the amount of sunny days we've seen in the past.

Simple Markov Model of Weather

Transition matrix of a Markov Chain:

$A_{ij} = P(S_t = j \mid S_{t-1} = i)$

$$A = \begin{matrix} \\ \text{s} \\ \text{r} \\ \text{c} \end{matrix} \begin{matrix} \begin{matrix} \text{s} & \text{r} & \text{c} \end{matrix} \\ \begin{pmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0 & 0.7 \\ 0.5 & 0 & 0.5 \end{pmatrix} \end{matrix}$$



Initial state: $\pi_0 = (0 \quad 1 \quad 0)$

state 1: $(0 \quad 1 \quad 0)\begin{pmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0 & 0.7 \\ 0.5 & 0 & 0.5 \end{pmatrix} = (0.3 \quad 0 \quad 0.7)$

state 2: $(0.3 \quad 0 \quad 0.7)\begin{pmatrix} 0.2 & 0.6 & 0.2 \\ 0.3 & 0 & 0.7 \\ 0.5 & 0 & 0.5 \end{pmatrix} = (0.41 \quad 0.18 \quad 0.41)$

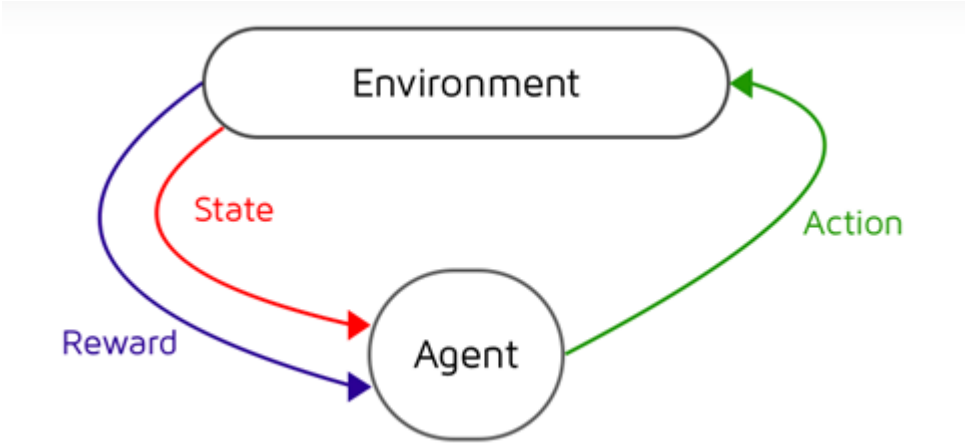
...

- Stochastic matrix:
  Rows sum up to 1
- Double stochastic matrix:
  Rows and columns sum up to 1

- After a certain number of time steps the output vector should correspond to the input vector: $\pi A = \pi$.
- This is an Eigenvector problem $\pi A = \lambda\pi$ with $\lambda = 1$ and π is a left eigenvector of matrix A with eigenvalue of 1.
- The steady state of (s,r,c) is: $\pi = (0.35 \quad 0.21 \quad 0.44)$ and describes a limit distribution.

Markov Processes vs. Markov Decision Processes

- Markov decision processes (MDPs) are an extension of Markov chains; the difference is the addition of **actions** (allowing choice) and **rewards** (giving motivation).
- Conversely, if only one action exists for each state (e.g. "wait") and all rewards are the same (e.g. "zero"), an MDP reduces to a Markov chain.
- While we deal with **discrete-time MDPs**, they can also be analyzed in continuous time.



Markov Decision Process

Definition: An MDP is a 5-tuple (S, A, P, R, γ)

- Set of states $s \in S$
- Set of actions $a \in A$
- Transition function / probabilities:

$$P(s, a, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$$

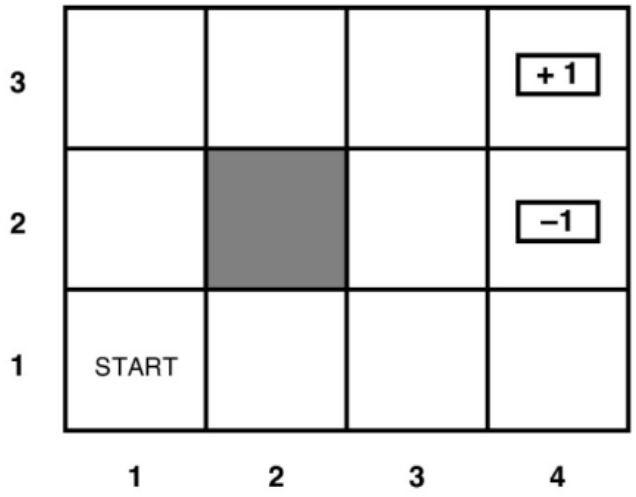
This definition satisfies the Markov property, because the next state only depends on the current

- Reward function (or stochastic distribution):

$$R(s, a, s') = \Pr(r_{t+1} | s_{t+1} = s', s_t = s, a_t = a)$$

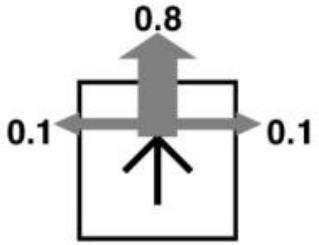
- Discount parameter γ

Example: Grid World



- An agent operates in a grid with solid and open cells.
- Each timestep, the agent receives a small negative „living“ reward.
- There are two bigger magnitude rewards at terminal states.
- The agent can try move North, East, South, West.
- The agent remains where it is if it tries to move into a solid cell or outside the world.
- „Noisy Control“: The chosen action only succeeds 80% of the time (for an open cell).
- 10% of the time, the agent ends up 90 degree off.
  - For example an agent surrounded by open cells and trying moving north 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.

Transition model:



Source of Example and Images: Berkeley CS188 (Pieter Abbeel and Dan Klein)

Actions and Action Spaces

- Different environments allow different kinds of actions. The set of all valid actions in a given environment is often called the action space.
- Some environments, like Atari games and Go, have **discrete action spaces**, where only a finite number of moves are available to the agent.
- Other environments, like where the agent controls a robot in a physical world, have **continuous action spaces**. In continuous spaces, actions are real-valued vectors.

Policies

- A policy is a rule used by an agent to decide what actions to take. It can be **deterministic**, in which case it is usually denoted by μ:

$$a_t = \mu(s_t)$$

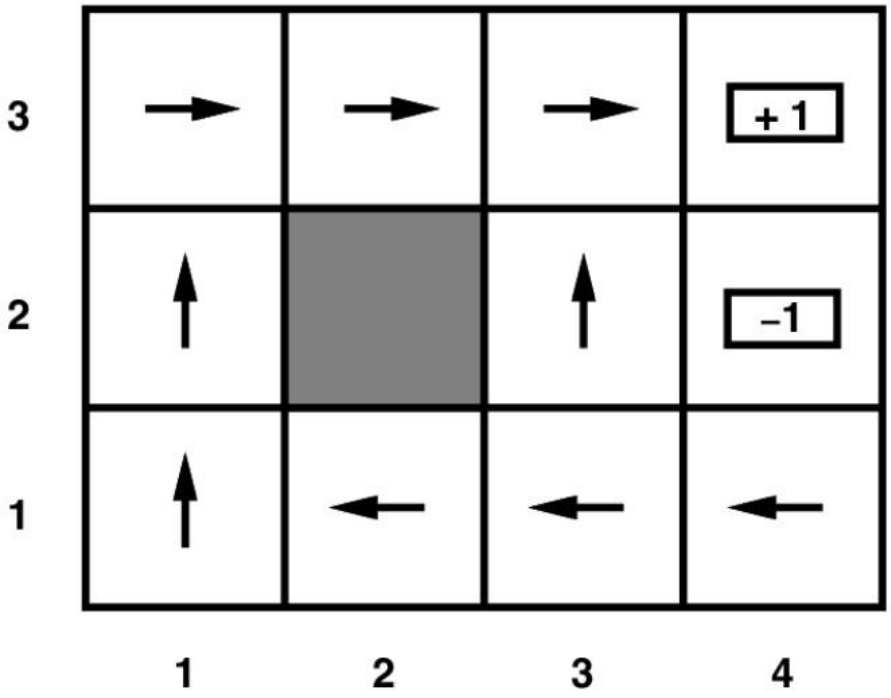
or it may be **stochastic**, usually denoted by π:

$\pi(a \mid s) :=$ probability of choosing a when observing state s

Then, sample a_t :

$$a_t \sim \pi(\cdot \mid s_t)$$

A Deterministic Policy



How do we evaluate different policies?

Trajectories

- A trajectory τ is a sequence of states and actions in the world:

$$\tau = (s_o, a_o, s_1, a_1, \dots)$$

- **State transitions** (what happens to the world between the state at time *t* (s_t) and the state at *t+1* (s_{t+1}) are governed by the natural laws of the environment and depend on only the most recent action a_t. They can be either deterministic,

$$s_{t+1} = f(s_t, a_t)$$

or stochastic $s_{t+1} \sim P(\cdot \mid s_t, a_t)$

- Actions come from an agent according to its policy.
- Trajectories are also frequently called **rollouts**.
  A trajectory that ends in a terminal state is also called an **episode**.

Reward and Return

- The **reward function** R is important in RL.
- When the environment transitions from state s_t to s_{t+1} after the agent has taken action a_t, the agent gets a (possibly stochastic) reward r_t:

$$r_t = R(s_t, a_t, s_{t+1})$$

Sometimes simplified notations are used, e.g.:

$$r_t = R(s_t, a_t) = \mathbf{E}_{s_{t+1}|s_t,a_t}[R(s_t, a_t, s_{t+1})]$$
$$r_t = R(s_t) = \mathbf{E}_{a_t \sim \pi(s_t)}[R(s_t, a_t)]$$

- The reward r_t just refers to a *single step*, is usually represented by a real number, and is often 0.
- Then we can define **return**: the **cumulative reward** for a whole trajectory τ – the total reward the agent receives from (or at the end of) the trajectory:

$$G(\tau) = \sum_{t=0}^{T-1} r_t = \sum_{t=0}^{T-1} R(s_t, a_t, s_{t+1})$$

Return

- We often apply **discount factor γ** (gamma; between 0 and 1) to rewards obtained later in the trajectory (especially if the trajectories are infinite).

$$G(\tau) = \sum_{t=0}^{\infty} \gamma^t \cdot r_t = \gamma^0 \cdot r_0 + \gamma^1 \cdot r_1 + \gamma^2 \cdot r_2 + \cdots$$

- The ultimate goal of the agent is to find an optimal policy that maximizes the **expected return** (over all trajectories). Assuming stochastic policy and environment transitions, we first define the probability distributions.

- The probability of a T-step trajectory $\tau = (s_0, a_0, s_1, \ldots s_T)$ is

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t) \cdot \pi(a_t|s_t)$$

Initial state probability

State transition probability of environment

Action probability of agent

Expected Return

- Then the **expected return** of π, denoted $J(\pi)$, is defined as:

$$J(\pi) = \int_{\tau} P(\tau|\pi) \cdot G(\tau) = \mathbf{E}_{\tau\sim\pi}[\, G(\tau)\,]$$

Return of the trajectory

Probability of trajectory w.r.t the given policy

- The central optimization problem in RL can then be expressed by

$$\pi^* = \underset{\pi}{\operatorname{argmax}} J(\pi)$$

where π^* is the **optimal policy**.

But how to find the optimal out of exponentially many policies?

Value Functions

- Value functions are used in most RL algorithms.
- A value function returns the **expected return** when starting from a given state s ("state value" $V(s)$) or for a given state-action pair ("state-action value" or "Q value" $Q(s,a)$).

There are four main functions to note:

1. **On-policy Value Function** – returns the expected return if you start from the given state s and always acting according to the given/current policy π:

$$V^{\pi}(s) = \mathbf{E}_{\tau\sim\pi}[\, G(\tau) \mid s_0 = s \,]$$

2. **On-policy Action-Value Function** – returns the expected return if you start from the given state s, take a fixed action a, then forever after act according to policy π:

$$Q^{\pi}(s,a) = \mathbf{E}_{\tau\sim\pi}[\, G(\tau) \mid s_0 = s, a_0 = a]$$

Optimal Value Function

off-policy

3. **Optimal Value Function** – returns the expected return if you start from the given state *s* and always act according to the **optimal policy** in the environment π^*:

$$V^*(s) = \max_{\pi} \mathbf{E}_{\tau\sim\pi}[\, G(\tau) \mid s_0 = s \,]$$

4. **Optimal Action-Value Function** – returns the expected return if you start from the given state s, take an arbitrary action a, then forever after act according to the **optimal policy π***:

$$Q^*(s,a) = \max_{\pi} \mathbf{E}_{\tau\sim\pi}[\, G(\tau) \mid s_0 = s, a_0 = a \,]$$

* Note on the connection between V and Q functions.

- $V^{\pi}(s) = \mathbf{E}_{a\sim\pi}[Q^{\pi}(s,a)]$
- $V^*(s) = \max_{a} Q^*(s,a)$

Bellman Equations

- All four value functions obey the Bellman equations.
- Its basic idea is that the value of the starting state is **the reward you expect to get from being there, plus the value of wherever you land next**.

Bellman equations for the on-policy value functions are:

$$V^{\pi}(s) = \mathbf{E}_{a\sim\pi, s'\sim P}[\, r(s,a) + \gamma \cdot V^{\pi}(s') \,]$$

$$Q^{\pi}(s,a) = \mathbf{E}_{s'\sim P}\Big[r(s,a) + \gamma \cdot \mathbf{E}_{a'\sim\pi}[Q^{\pi}(s',a')]\Big]$$

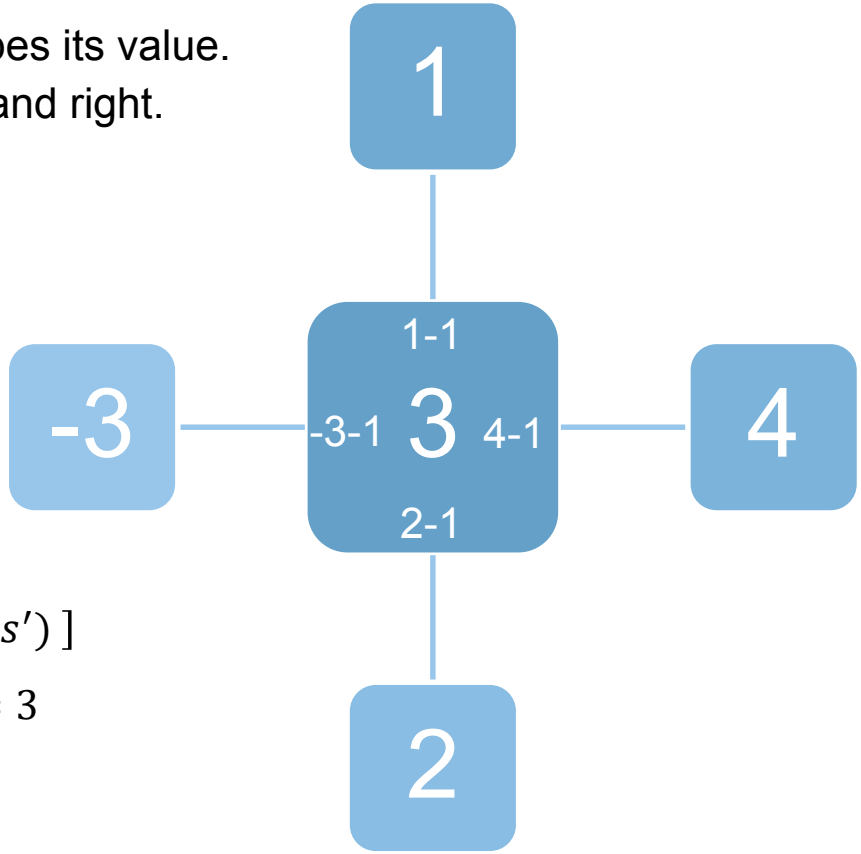
Bellman equations for the **optimal** value functions are:

$$V^{*}(s) = \max_{a} \mathbf{E}_{s'\sim P}\,[\, r(s,a) + \gamma \cdot V^{*}(s') \,]$$

$$Q^{*}(s,a) = \mathbf{E}_{s'\sim P}\Big[r(s,a) + \gamma \cdot \max_{a'} Q^{*}(s',a')\Big]$$

Example: Values and Rewards

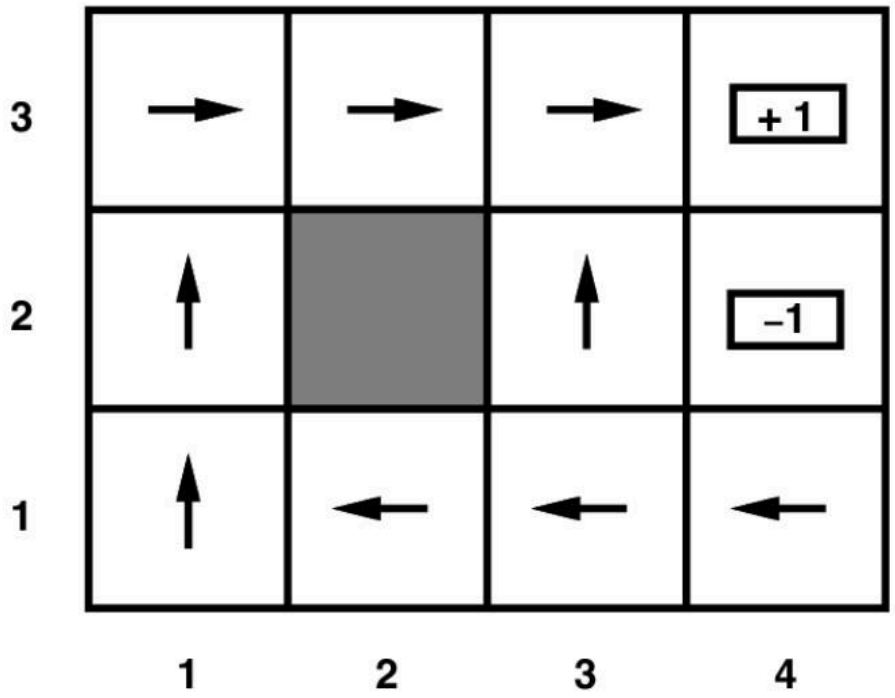
- The number in each square describes its value.
- We have 4 actions: up, down, left, and right.
- The reward of moving from one square to another is $r(s,a) = -1$.



$$V^*(s) = \max_a \mathbf{E}_{s' \sim P} [\, r(s,a) + \gamma \cdot V^*(s') \,]$$

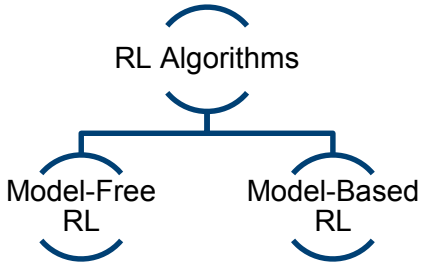
$$V^*(s) = \max_a \mathbf{E}_{s' \sim P} [\, -1 + 1 \cdot 4 \,] = 3$$

How to Find an Optimal Policy?



Optimal policy when
R(s) = -0.04 for every
non-terminal state

Model-free vs. Model-based



Model-based RL algorithms are applied when the agent either

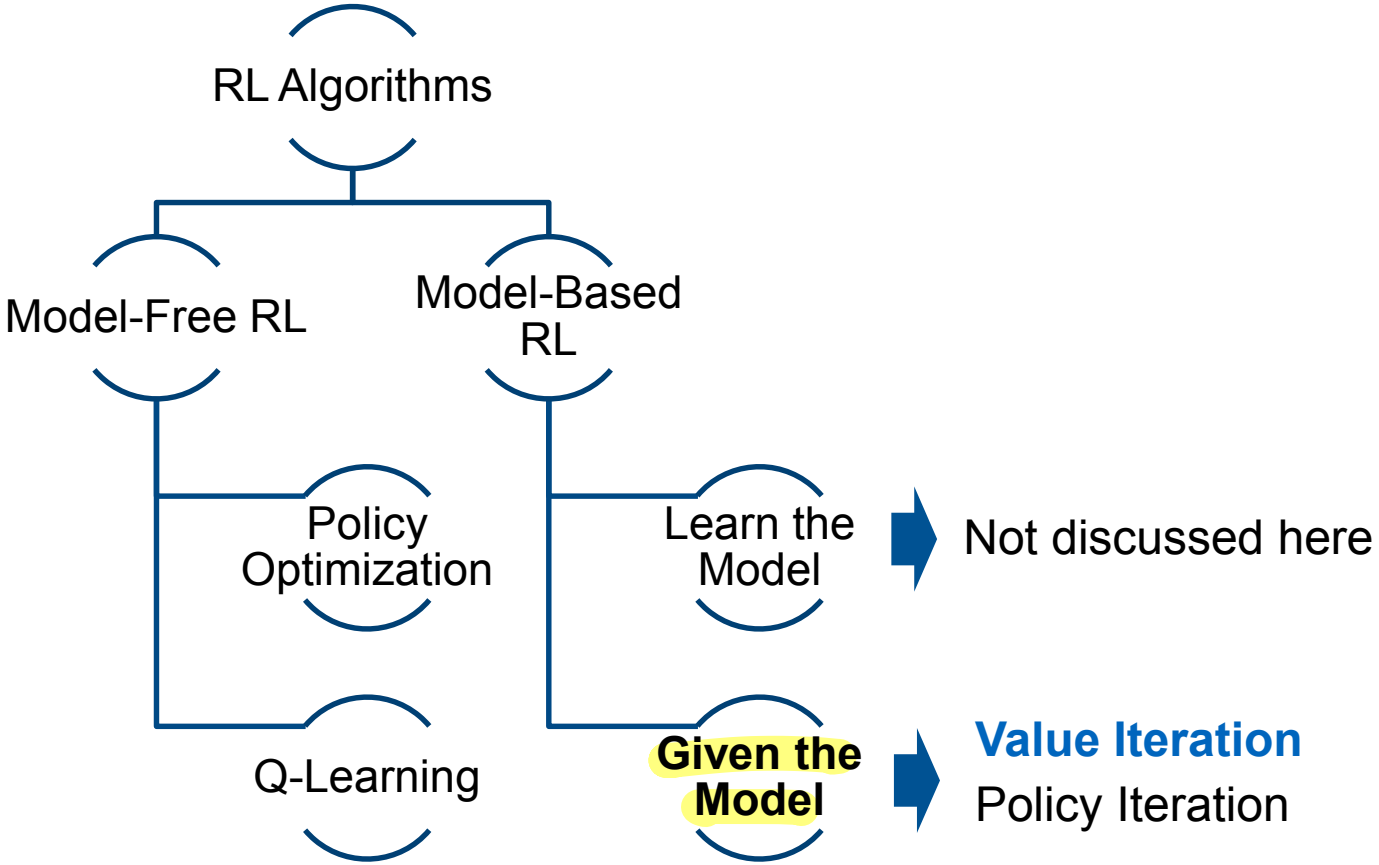
- has access to the true environment, i.e., the state transition P and reward functions R. This is usually not the case!
- uses a model of the environment: $\hat{P}$, $\hat{R}$

Acting in the model environment is itself a **Markov Decision Process**.

- Now, the agent can **plan** by thinking ahead to obtain (or learn) an optimal policy before actually acting out the plan in the "real" environment.
- Upon interacting with the real environment, the model can be updated from observed data and will become more accurate over time.

In **model-free RL**, agents do not maintain an explicit model of the environment.

Classification of RL-Algorithms



Value Iteration

Remember the value function $V^{\pi}(s) = \mathbf{E}_{a\sim\pi, s'\sim P}[\, r(s,a) + \gamma \cdot V^{\pi}(s')\,]$

Value iteration iteratively builds a refined estimate of the value function through trial and error.

1. Start with some arbitrary value estimates $V_0(S) = 0$ for all states s other than terminal states.
2. Iterate the Bellman update until $|V_{i+1}(s) - V_i(s)| < \epsilon$

$$V_{i+1}(s) = \max_{a\in A} \sum_{s'\in S} P(s'|s,a)(R(s,a,s') + \gamma V_i(s'))$$

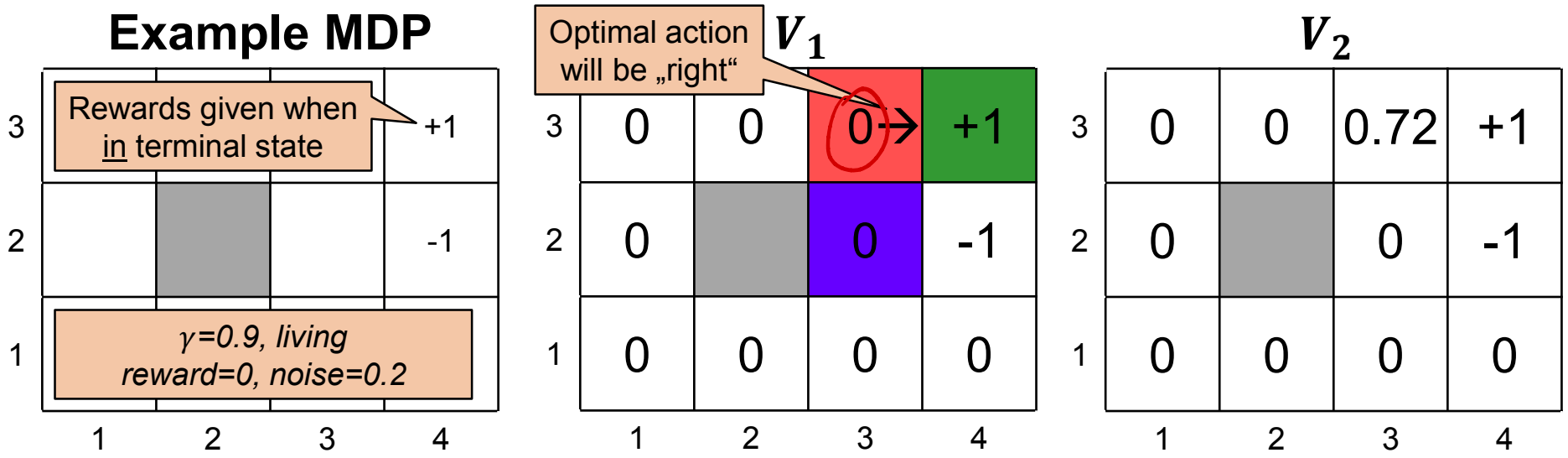
This iteratively improves your value estimates using Q, V relations.

There are convergence guarantees.

The method was described by Bellman (1957), but goes back to Shapley in game theory (1953).

Example: Value Iteration

Bellman Update Rule $V_{i+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a)(R(s',s,a) + \gamma V_i(s'))$



$$V_2(\langle 3,3 \rangle) = \sum_{s' \in S} P(s'|\langle 3,3 \rangle, right)(R(\langle 3,3 \rangle, right, s') + \gamma V_i(s'))$$

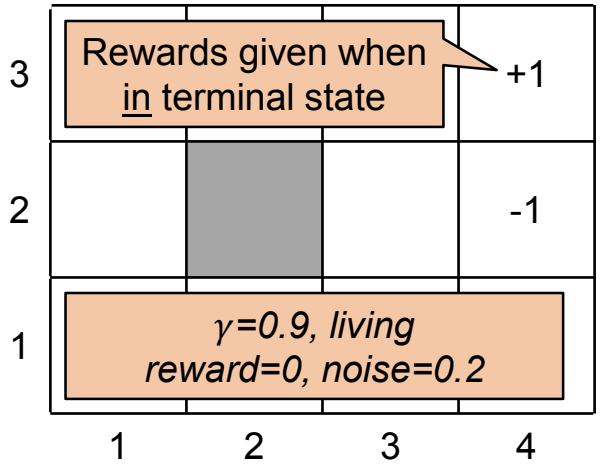
$$= 0.8[0 + 0.9 * 1] + 0.1[0 + 0.9 * 0] + 0.1[0 + 0.9 * 0] = 0.72$$

We don‘t fully compute the maximum, but assume we know the optimal action (right) in the example.

Example: Value Iteration

$$V_{i+1}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s,a)(R(s',s,a) + \gamma V_i(s'))$$

Example MDP



V_2

3	0	0	0.72	+1
2	0		0	-1
1	0	0	0	0
	1	2	3	4

V_3

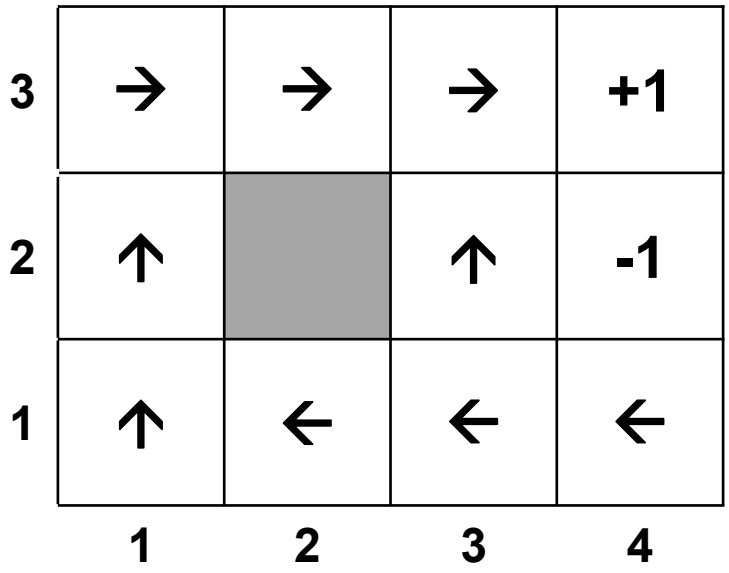
3	0	0.52	0.78	+1
2	0		0.43	-1
1	0	0	0	0
	1	2	3	4

$$V_3(\langle 2,3 \rangle) = 0.8[0 + 0.9 * 0.72] + 0.1[0 + 0.9 * 0] + 0.1[0 + 0.9 * 0] = 0.5184$$
$$V_3(\langle 3,2 \rangle) = 0.8[0 + 0.9 * 0.72] + 0.1[0 + 0.9 * -1] + 0.1[0 + 0{,}9 * 0] = 0.4284$$
$$V_3(\langle 3,3 \rangle) = 0.8[0 + 0.9 * 1] + 0.1[0 + 0.9 * 0.72] + 0.1[0 + 0.9 * 0] = 0.7848$$

go right go up (but cannot) go down

Information propagates outward from the terminal states – a form of dynamic programming.
Value iteration converges to the optimal value function.

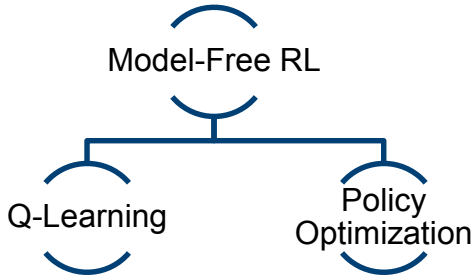
Policy Extraction



This is the optimal policy when R(s) = -0.04 for every non-terminal state.

- After value iteration is completed, find out the optimal policy (policy extraction).
- Note that the values might still change while the policy does not anymore.

Why Model-Free Methods ?



A **Markov Decision Process** is a formalism (a process) that describes a fully observable environment in RL.

Usually, the transition probabilities $P(s'|s,a)$ or the rewards $R(s',s,a)$ are not given a priori (autonomous driving, stock trading, etc.)

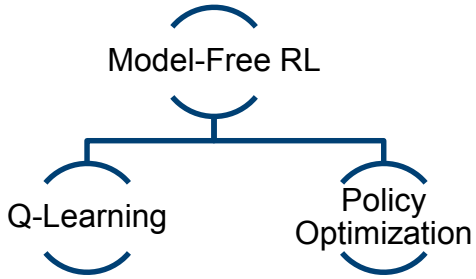
$$Q^{\pi}(s,a) = \mathbf{E}_{s' \sim P}\left[r(s,a) + \gamma \cdot \mathbf{E}_{a' \sim \pi}[Q^{\pi}(s',a')]\right]$$

$$= \sum_{s'} P(s'|s,\pi(s))(R(s',s,\pi(s)) + \gamma V(s'))$$

What can you do?

- Obtain a set of simulations/trajectories/samples with each transition in the episodes of the form (s,a,r,s')
  - Using sensors to understand robot's new position when it does an action
  - Recording new patient vitals when given a drug from a state etc.

Q-Learning



- Methods in this family learn an **approximator $Q_\theta(s,a)$** for the optimal action-value function, $Q^*(s,a)$.
- Typically, they use an objective function based on the **Bellman optimality equation**.
- Given the learned Q-values, episodes follow an ϵ-greedy policy:

balance exploration vs exploitation

$$\pi(s) = \begin{cases} \underset{a}{\text{argmax}}\, Q_\theta(s,a), & \text{with probability } 1-\epsilon \text{ (exploitation)} \\ \text{random action}, & \text{with probability } \epsilon \text{ (exploration)} \end{cases}$$

- How do we learn Q_θ?

Q-Learning Background 1: Monte Carlo Learning

- Play many **episodes τ** via trial and error and distribute the total reward on all actions played in an episode.
- Total reward of an episode: $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$
- Distribute the reward equally on all states in an episode:
  - either the value fct. $V^{new}(s_t) = V^{old}(s_t) + \frac{1}{T}\left(R(\tau) - V^{old}(s_t)\right) \; \forall t \in [1, \dots, T]$
  - or for Q-fct. $Q^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \frac{1}{T}\left(R(\tau) - Q^{old}(s_t, a_t)\right) \; \forall t \in [1, \dots, T]$
- **Problems:** if one bad action leads to a loss in an otherwise good episode, all the good moves are penalized. If only complex policies led to a reward, the agent never learns. Episodes may be infinitely long, so the total reward cannot be sampled. MC learning is very **sample-inefficient**.

➢ **Temporal Credit Assignment Problem**: Given sequence of actions and rewards, how to assign credit/blame for each action?

Q-Learning Background 2:
Temporal Difference (TD) Learning

Remember Bellman's optimality condition: $V(s_t) = \mathbb{E}[r_t + \gamma V(s_{t+1})]$

Given a **single** transition sample (s, a, r, s'), not the entire episode, a **TD(0)** update adjusts the value function estimate in line with the Bellman equation.

expected value of being in s_t based on sample

$$V^{new}(s_t) = V^{old}(s_t) + \alpha\left(r_t + \gamma V^{old}(s_{t+1}) - V^{old}(s_t)\right) \forall t \in [1, \dots, T]$$

add a weighted error term

- If you do the step above for each sample, you compute a running average over the samples. Each sample is considered a representative of the transition probability distribution.
- Perform many such updates over many transitions and we should see convergence ($V^{new} = V^{old}$).
- In the formula above, we sample **one** reward, then bootstrap using the old value function. This is called TD(0). However, multi-step lookaheads (taking multiple actions) are also possible w. Monte Carlo learning as other extreme.

(Tabular) Q-Learning: TD-learning on the Q-fct.

Q-learning is off-policy TD-learning on the Q-function.

Start with a random Q-table ($S \times A$). For all transitions collected according to any behavior policy, perform this TD Update

sample

$$Q^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q^{old}(s_{t+1}, a') - Q^{old}(s_t, a_t) \right)$$

$$\forall t \in [1, \dots, T]$$

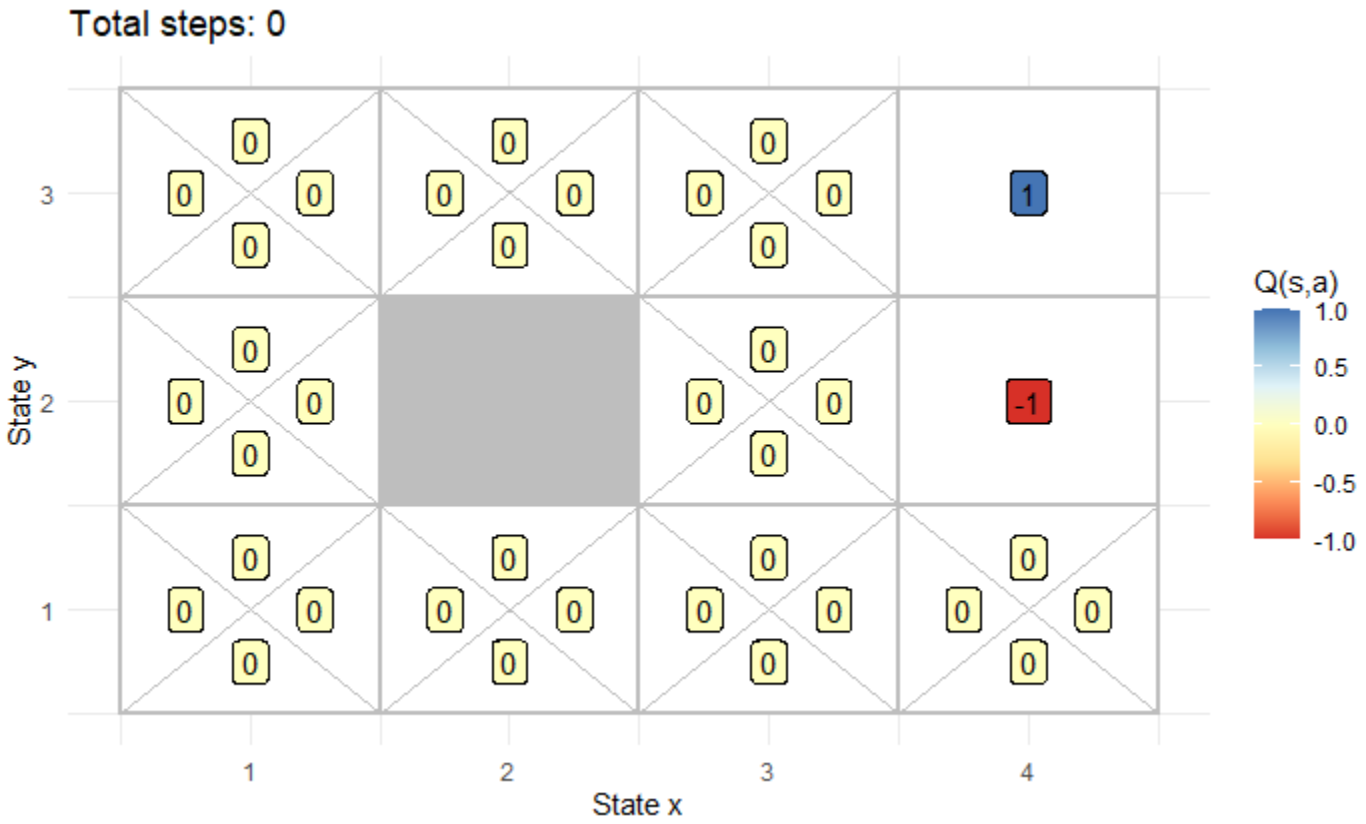
- In words: If you experience more reward than expected, increase your Q^{new}
- Off-policy: Q-Learning learns an approximation to the *optimal* action value function, $\max_{a'} Q^{old}(s_{t+1}, a')$, independently of the policy being followed to collect samples.

Q-Learning

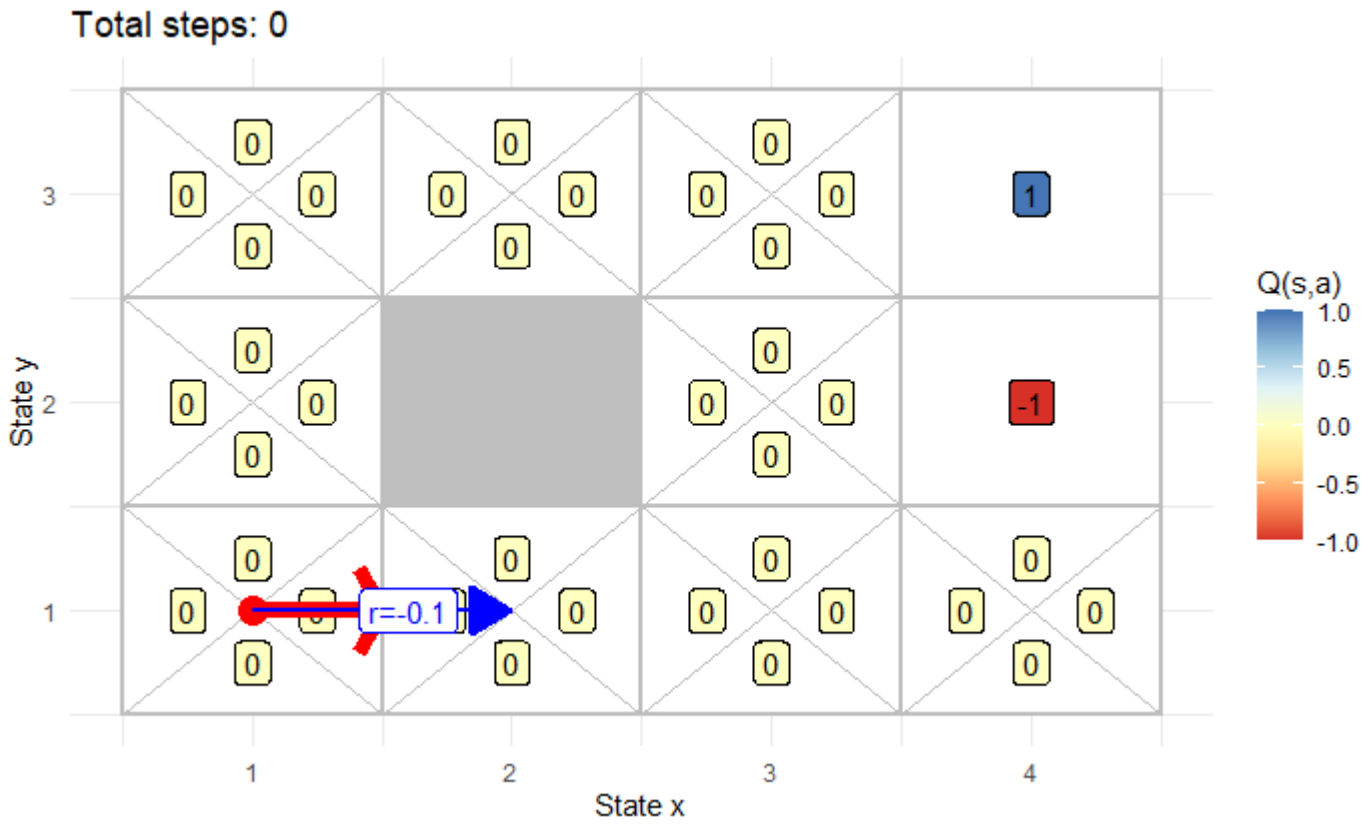
1. Start with a random Q-table ($S \times A$).
2. Choose one among the two actions
    a. (ε-greedy) With probability ε, choose a random action (EXPLORATION)
    b. With probability 1-ε, an action that maximizes Q-value from a state. (EXPLOITATION)
3. Perform an action and collect transition (s, a, r, s')
4. Update Q-table using the corresponding TD updates.
5. Repeat steps 2-5 till convergence of Q-values across all states.

Q-Learning: Intitial State 0

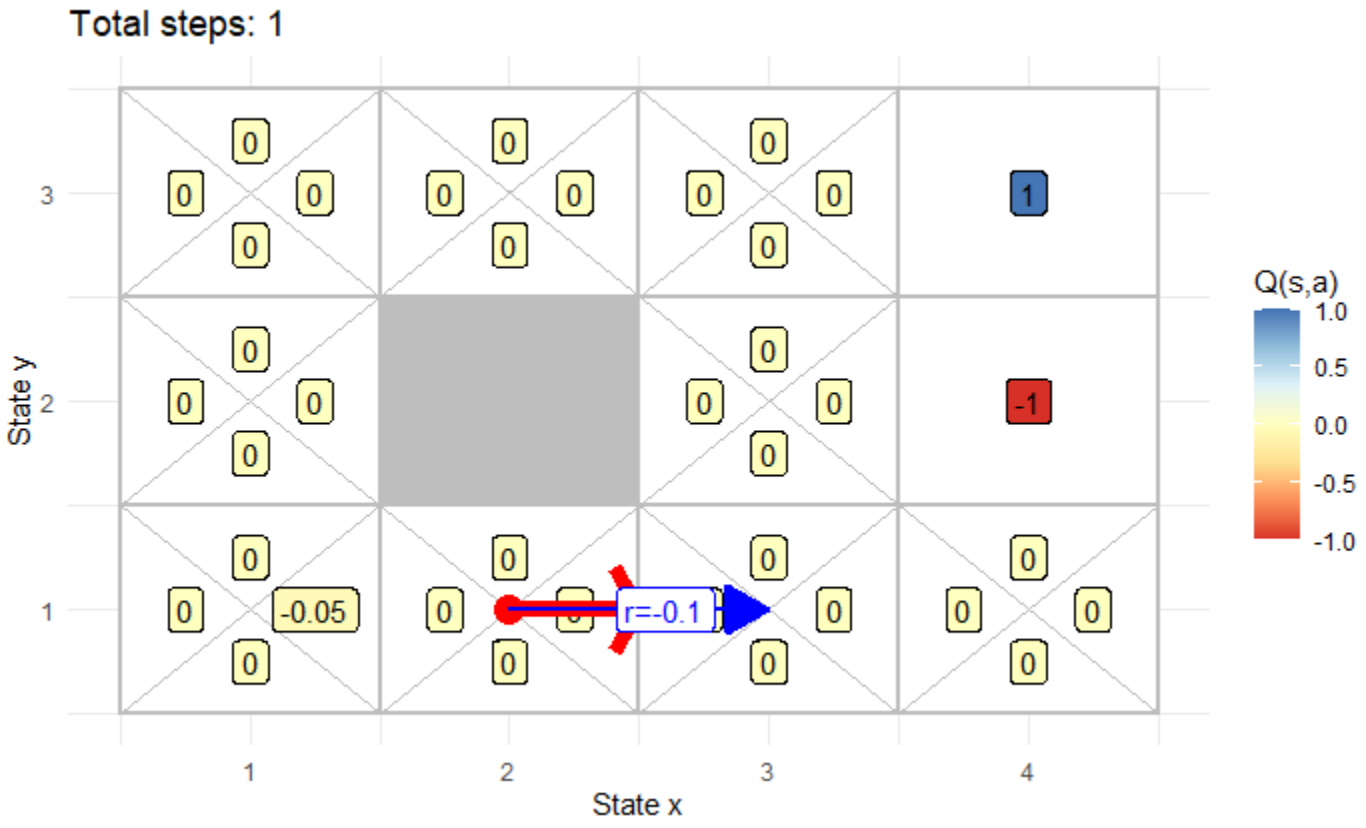
ε = 0.2
step size α = 0.5
discount rate γ = 1
default reward = -0.1



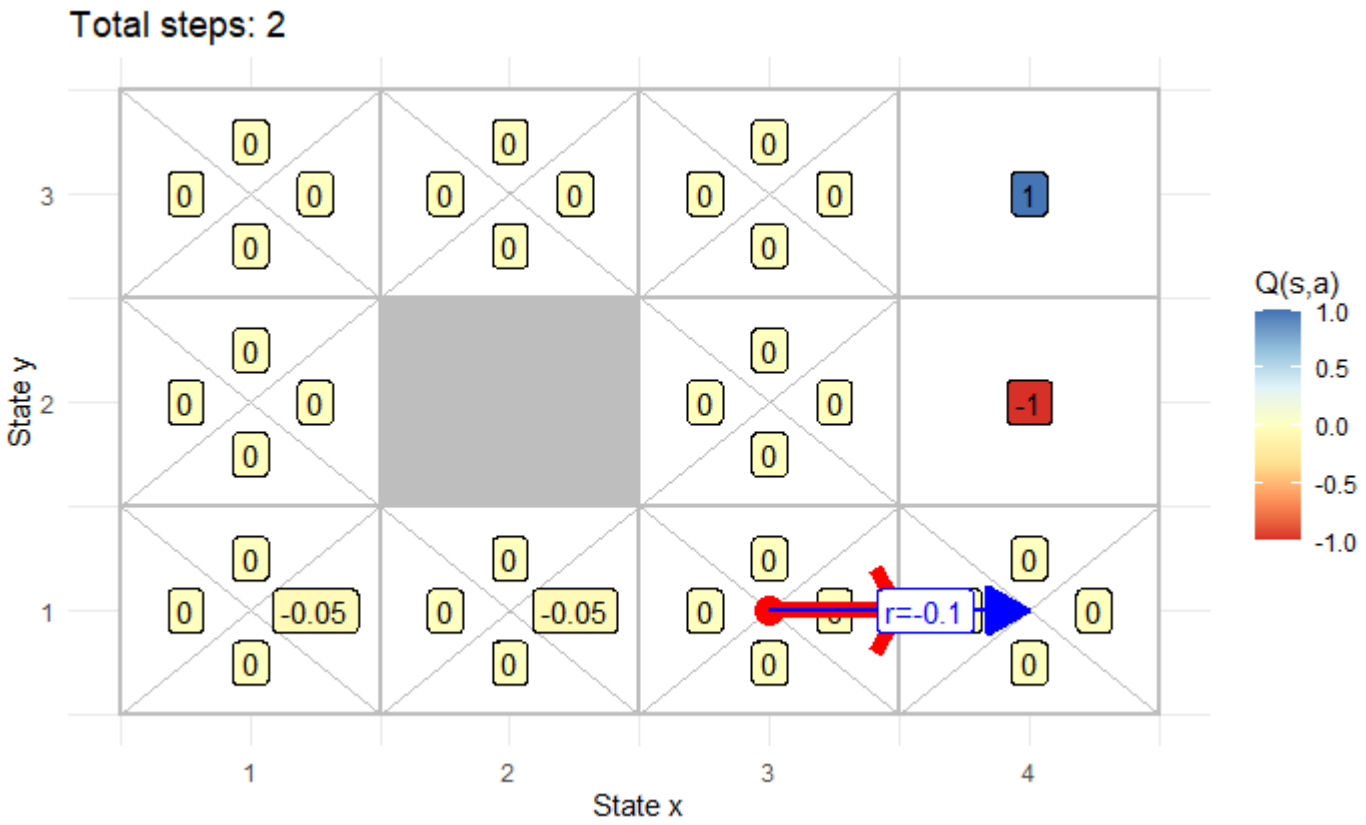
Step 1



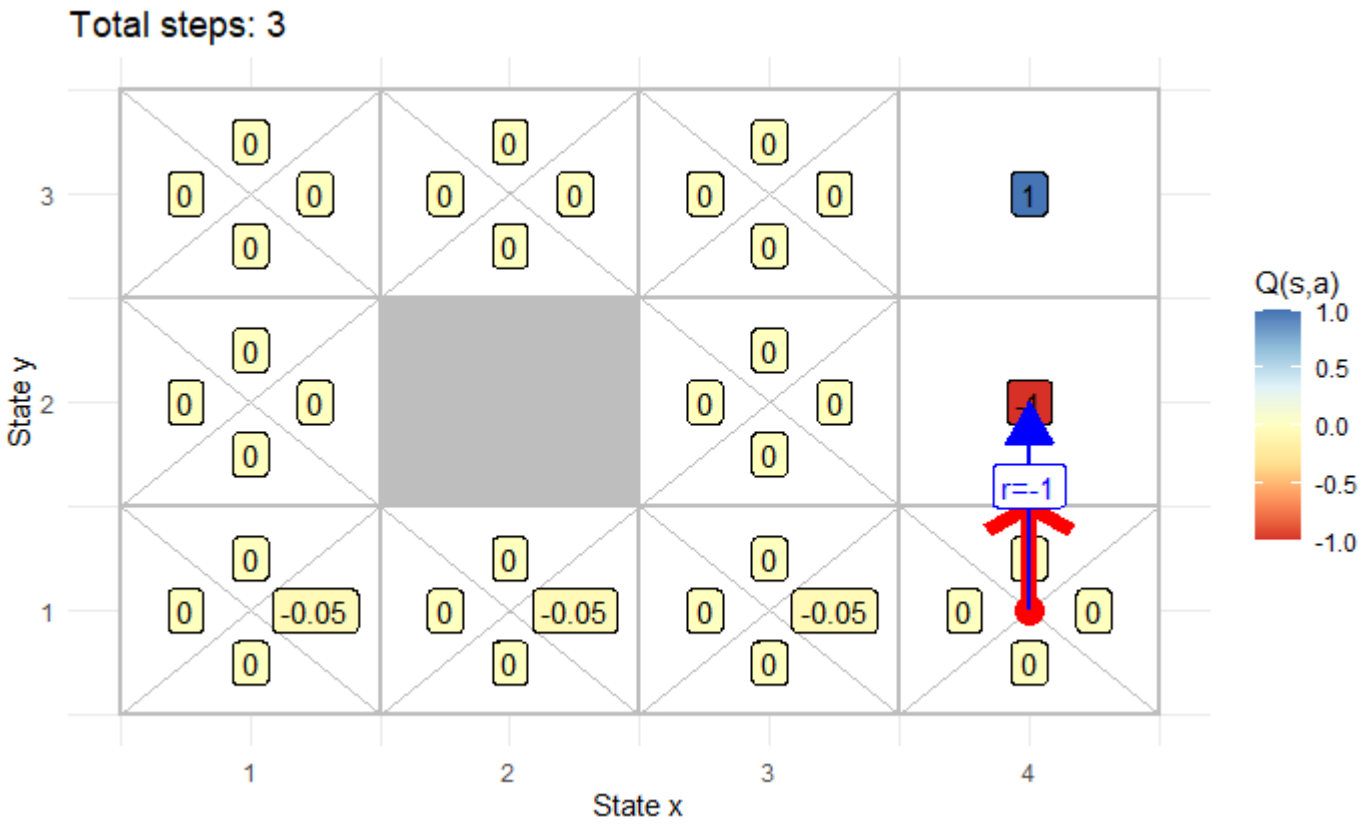
Step 2



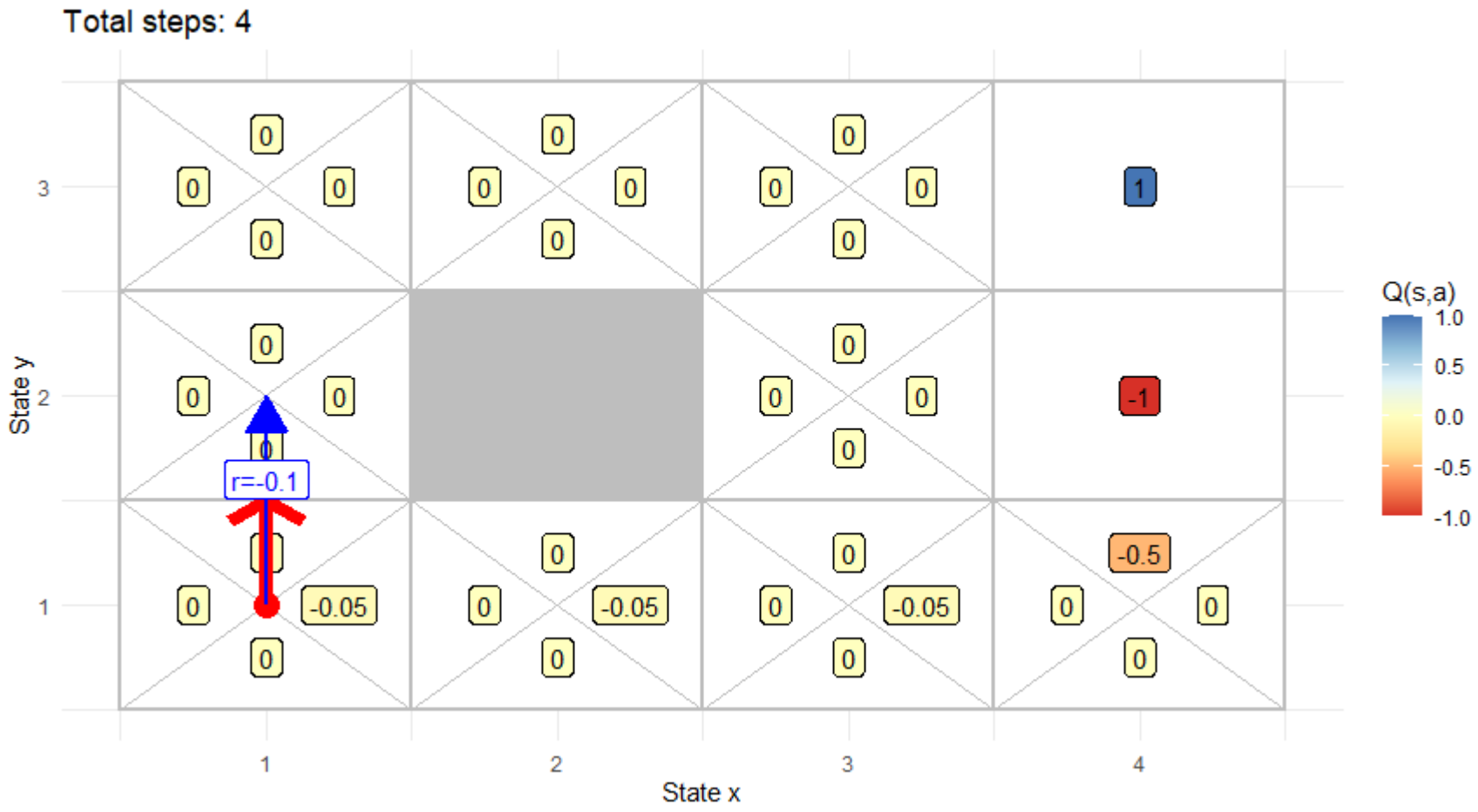
Step 2



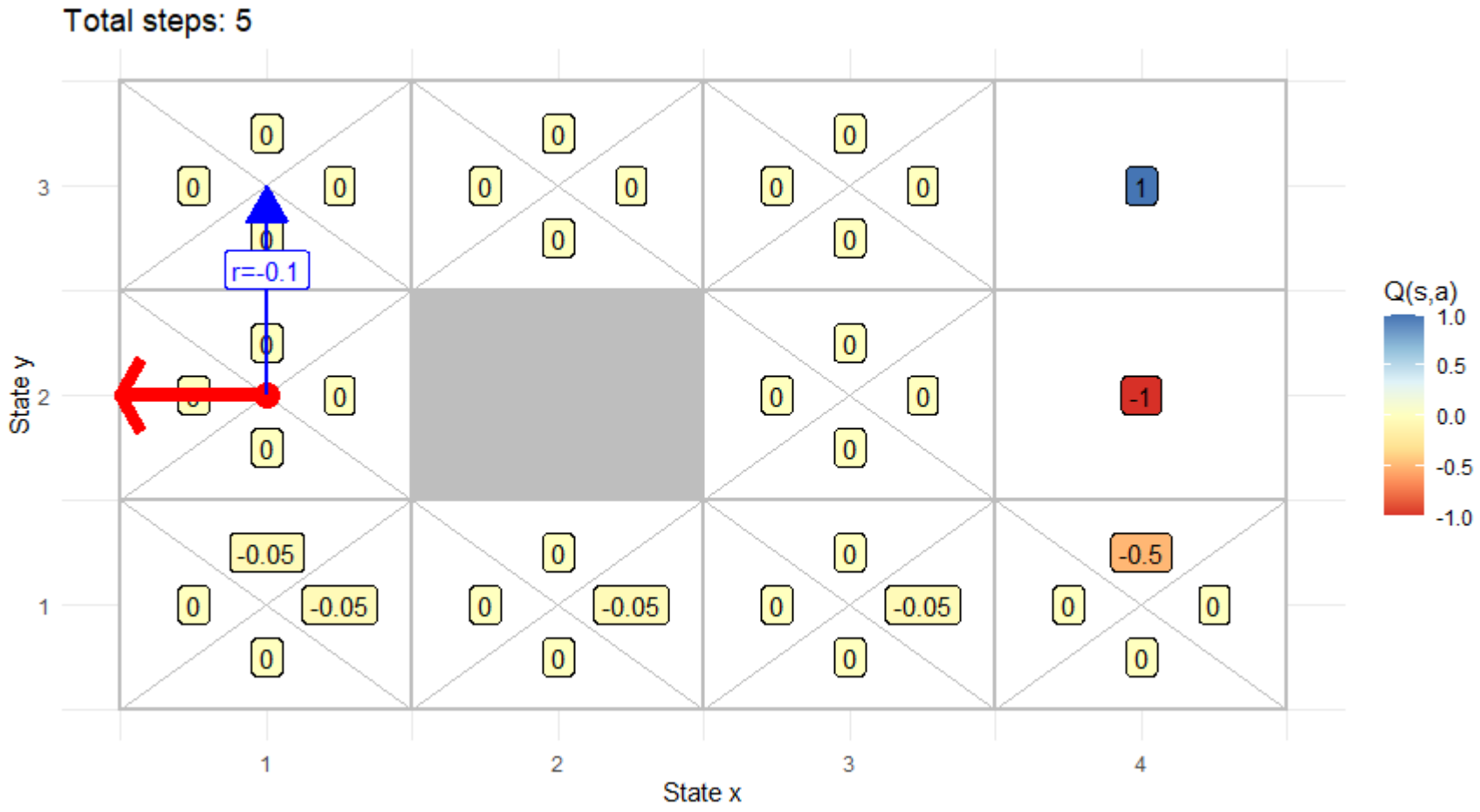
Step 3



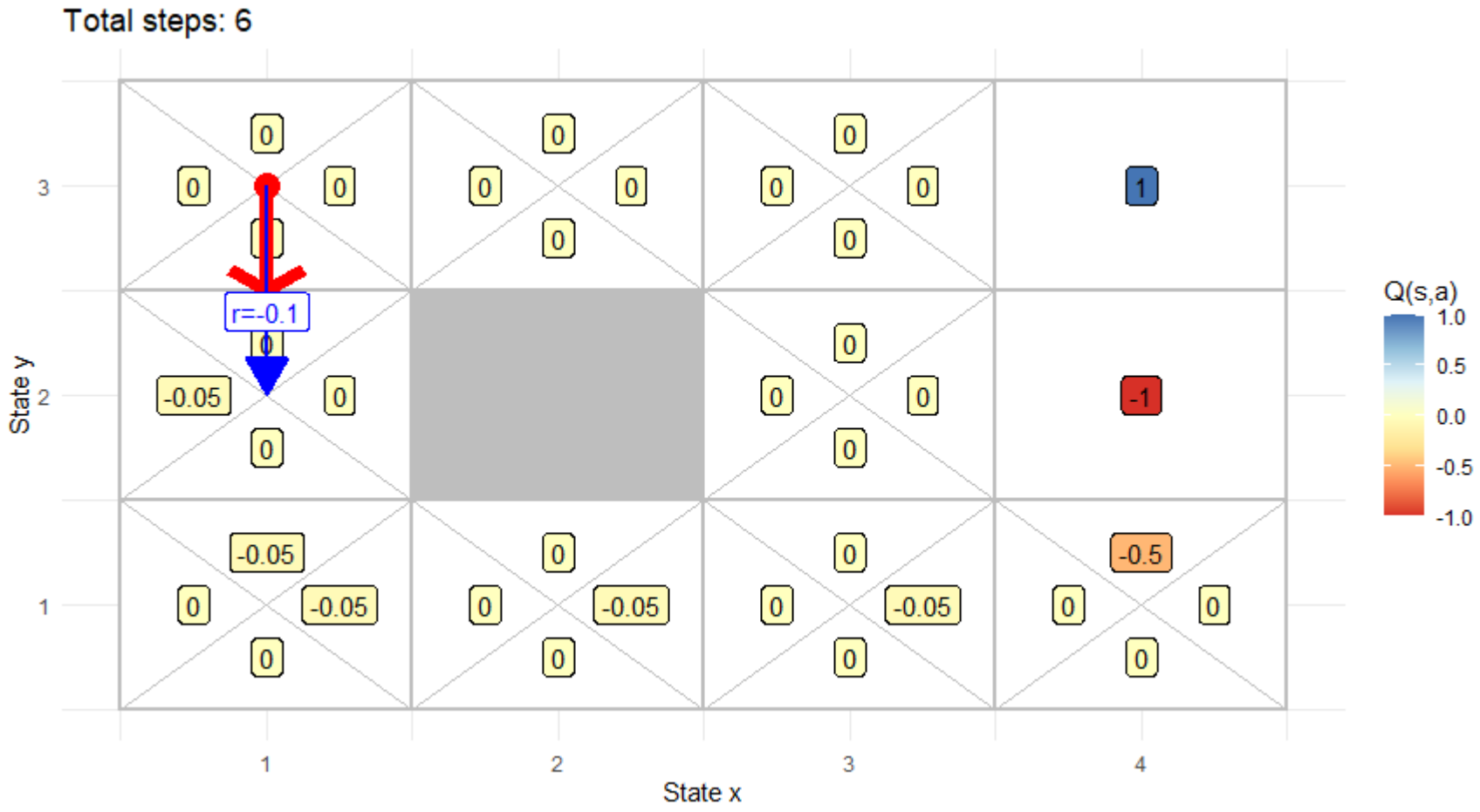
Step 4: New Episode



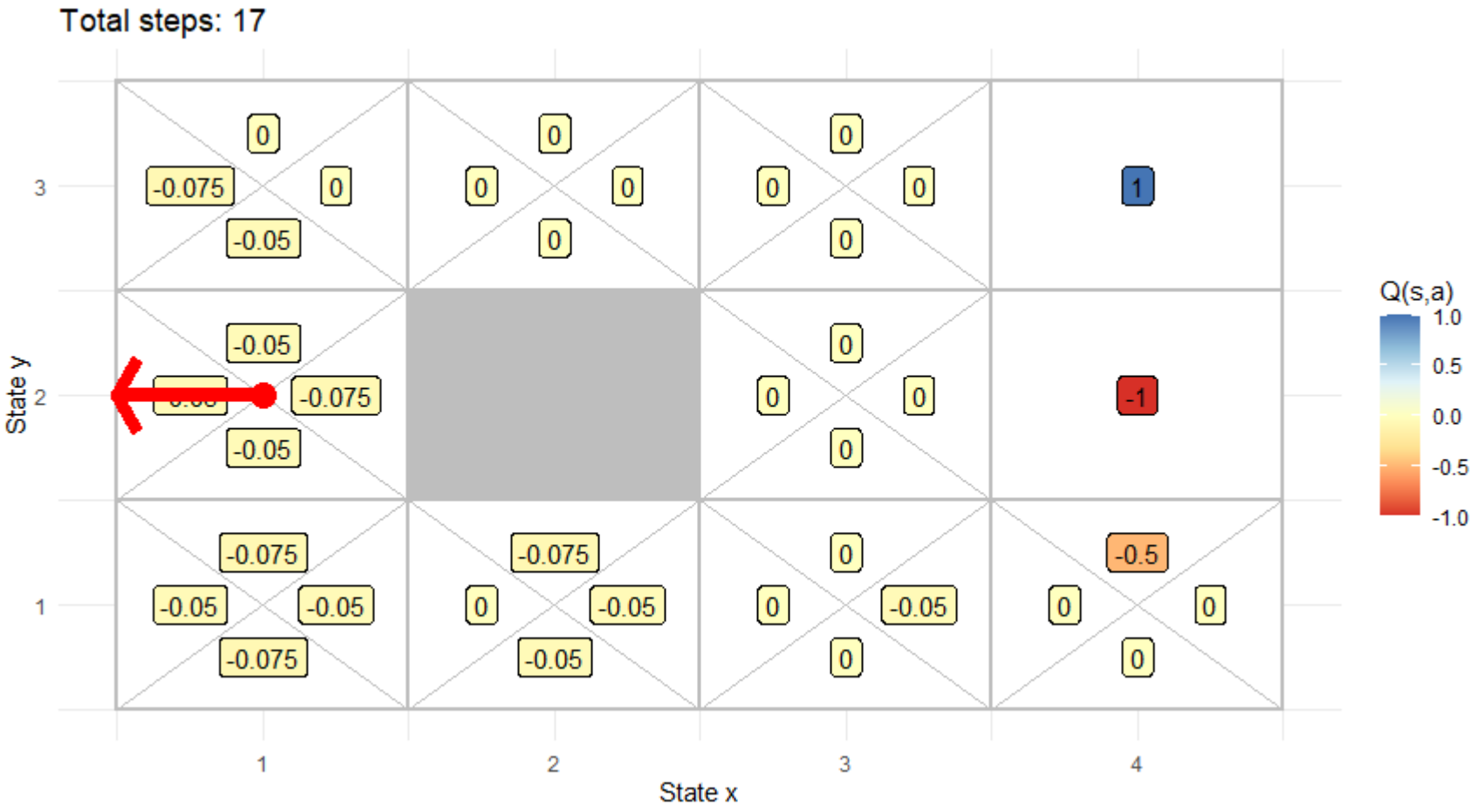
Step 5



Step 6

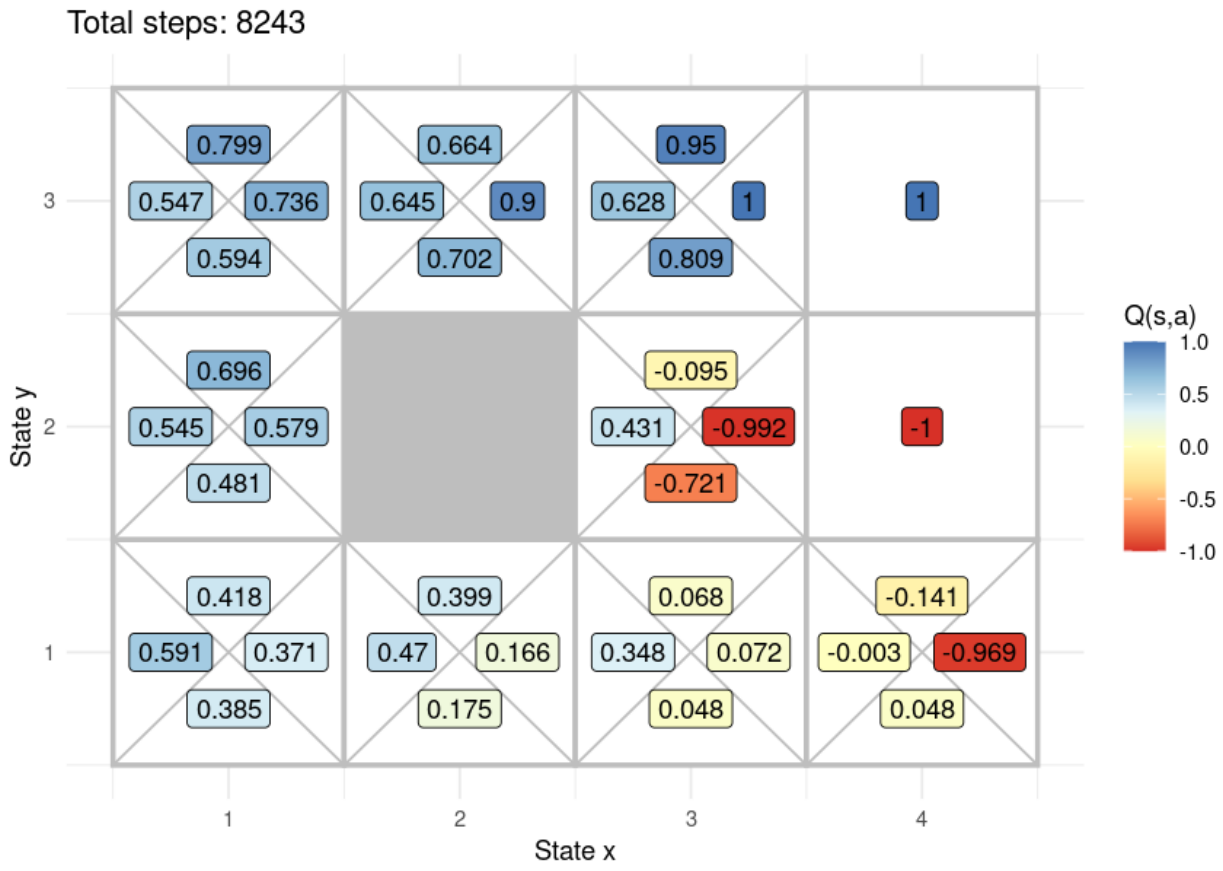

Total steps: 6
r=-0.1
-0.05
-0.05
-0.05
-0.05
-0.5
1
-1
State y
State x
1
2
3
4
Q(s,a)
1.0
0.5
0.0
-0.5
-1.0

Step 17



Example of Q-Learning after 8243 Steps

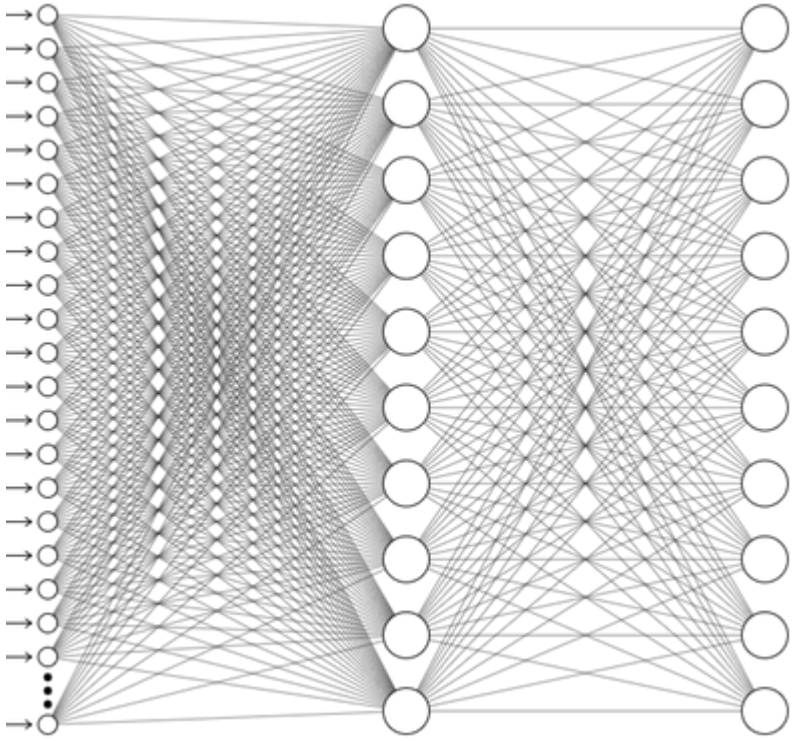
We have a Q-value for each possible action:



Deep Q-Learning Networks (DQNs) with Discrete Actions

Optimal Q-function is approximated using Neural Network. We learn the Q-value of a set of |A| actions and pick the maximum. The NN generalizes and can better handle large state spaces as compared to tabular methods.

state s



$Q(s, a = 1)$

$Q(s, a = 2)$

...

$Q(s, a = |A|)$

Policy: $\pi(s) = a^* = argmax_a Q(s, a)$

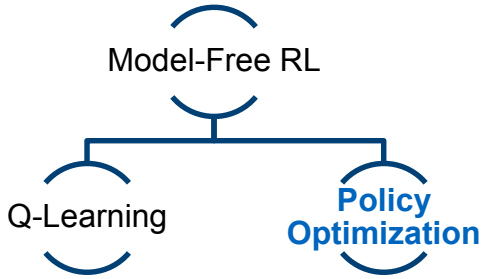
Approximate Q-Learning Methods

- DQN (Deep Q-Networks, 2014): Breakthrough results in Deep RL, achieving superhuman performance in Atari Games, learned from pixels

- Several extensions
  - Double DQN, dueling DQN *(deal with positive bias present in standard Q-learning)*
  - Distributional Q-Learning (e.g. C51) *(learns a Q-distributions rather than point-estimates)*
  - RAINBOW *(combines several proposed improvements into a single algorithm)*
  - …

Further reading on DQN-methods: Rainbow Paper (Hessel et al, 2017): https://arxiv.org/pdf/1710.02298.pdf

Policy Optimization



Methods in this family represent a policy explicitly as $\pi_\theta(a|s)$. They optimize the parameters θ directly by **gradient ascent** on the performance objective $J(\pi_\theta)$.

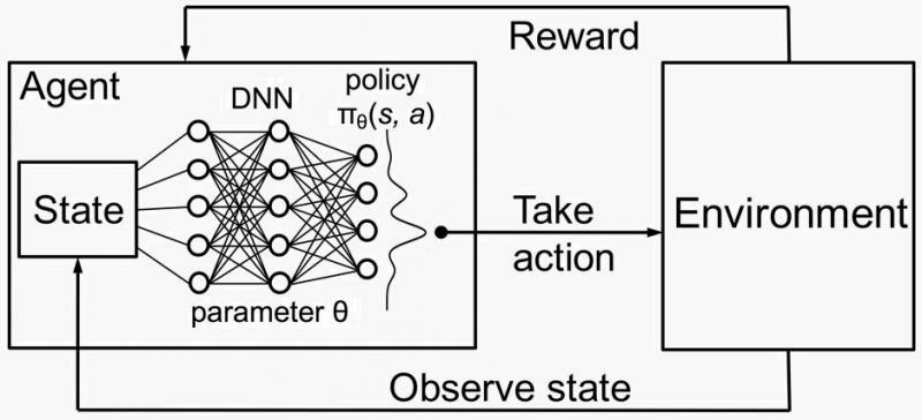
In a tabular setting with discrete actions, π_θ is an explicit vector of probabilities. In deep RL, the policy is a deep neural network, either outputting probabilities or a specific action. This allows us to handle **large state spaces** with an NN.

Policy optimization is almost always performed **on-policy**, which means that each update only uses data collected while acting according to the most recent version of the policy. Each sample is used once only.

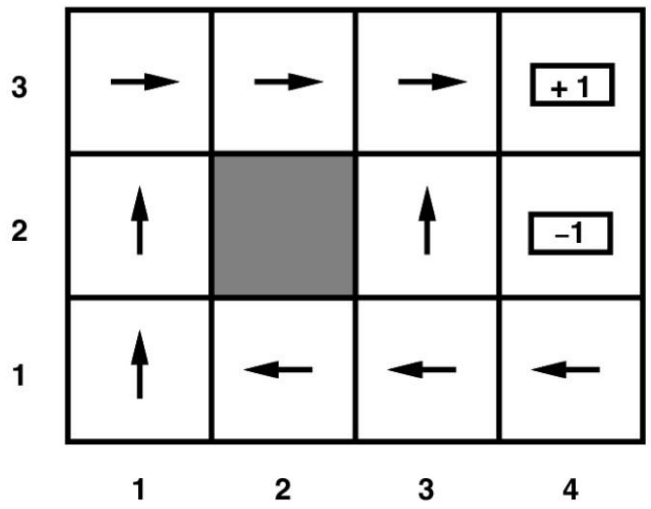
Policy Optimization Methods

Characteristics

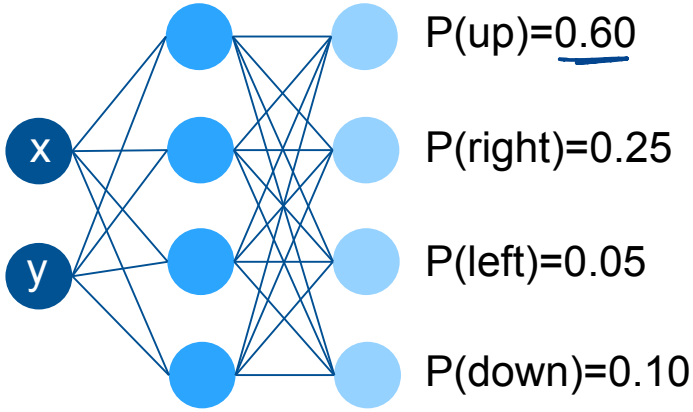
- We don‘t need the distribution of states or the environment dynamics. It is model-free.
- We learn a parametrized policy and select stochastic actions **without consulting a value function**. The stochastic policy class smooths out the problem.
- Parameters in DeepRL are the weights of a function-approximating NN, learned with SGD
- Policy gradient methods work well with **continuous action spaces**.



Example Policy NN for GridWorld



Policy NN: $\pi_\theta(a|s) = \Pr(a|s;\theta)$



Use a sigmoid activation for discrete actions or a Gaussian for continuous (but stochastic) actions.

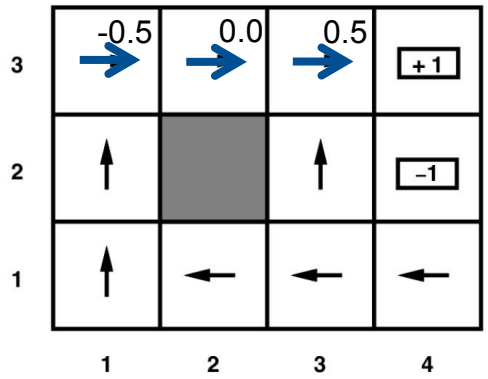
Stochastic policies allow us to explore new actions, not just exploit known ones.

on-policy

REINFORCE: Monte-Carlo Policy-Gradient Control

- Generate an episode using your policy network and keep track of the states, actions and the reward in memory
- At the end of the episode, go back through states, actions, and rewards, and compute discounted rewards at each time step (see example below).
- Use the returns and the actions the agent took to perform backpropagation, and update the weights of the NN. This increases the good actions and decreases the bad actions for all paths.

Example episode moving from 1,3 to 3,3:



X	Y	Discounted Rewards	Action	Change Weights based on Rewards
3	3	0.5	Right	Increase
2	3	0	Right	stay
1	3	-0.5	Right	decrease
...				

Adapt the weights of the NN: If the probability of „Right“ in the NN is 0.4, but we want it to go right (return 1), then increase the weights and consider rewards (next slides).

Reinforcement Learning