An Introduction to Reinforcement Learning

Dr Ing. Gianluca Valentino

Department of Communications and Computer Engineering

University of Malta

Outline

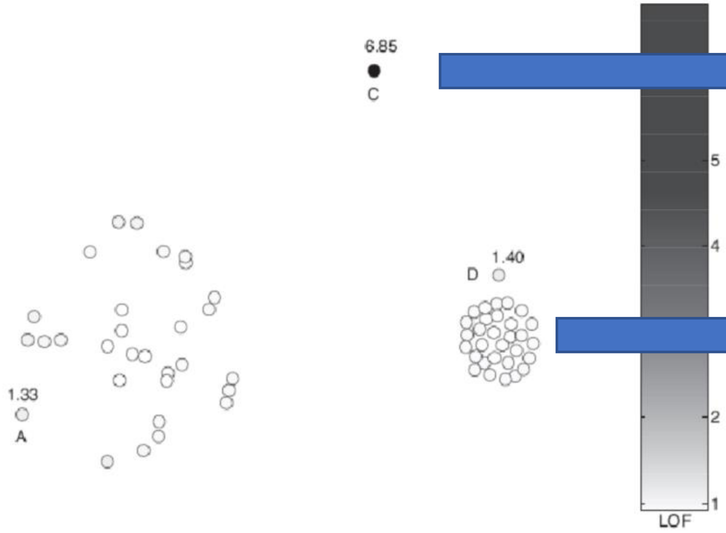
- What is Reinforcement Learning?
- RL terminology: states, actions, reward, policy
- Value function and Q-value function
- Q-learning and neural networks
- Python notebooks: Grid World and Cart Pole

What is Reinforcement Learning?

- So far: **Supervised Learning**
  - **Data:** (X, y)
  - **Goal:** Learn a function to map X -> y
  - **Examples:** classification, regression, object detection etc

Dog

- So far: **Unsupervised Learning**
  - **Data**: X (no y)
  - **Goal:** Learn some underlying hidden structure in the data
  - **Examples:** clustering, dimensionality reduction, anomaly detection

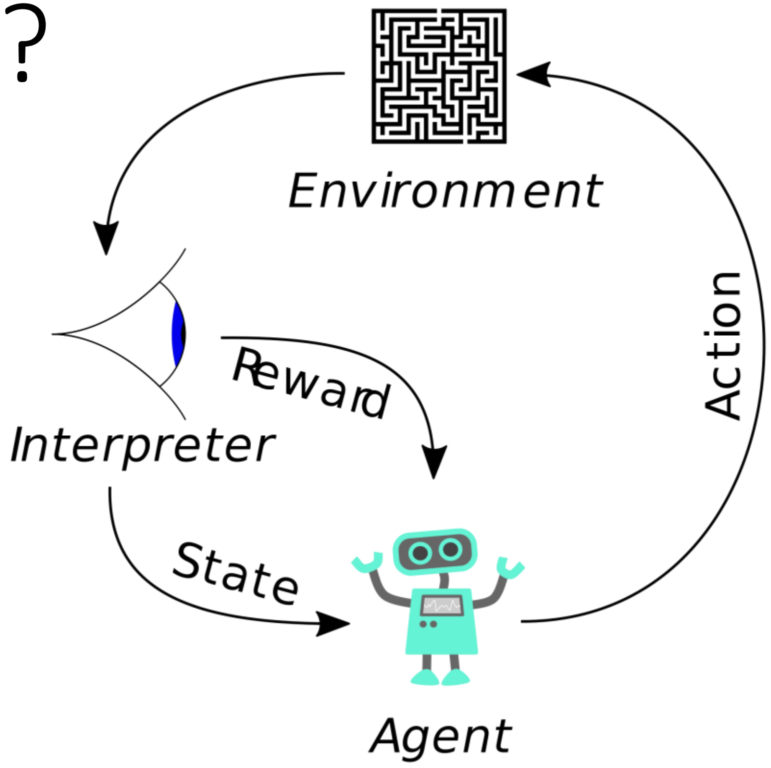


relative density (LOF) outlier scores

Abnormal

Normal

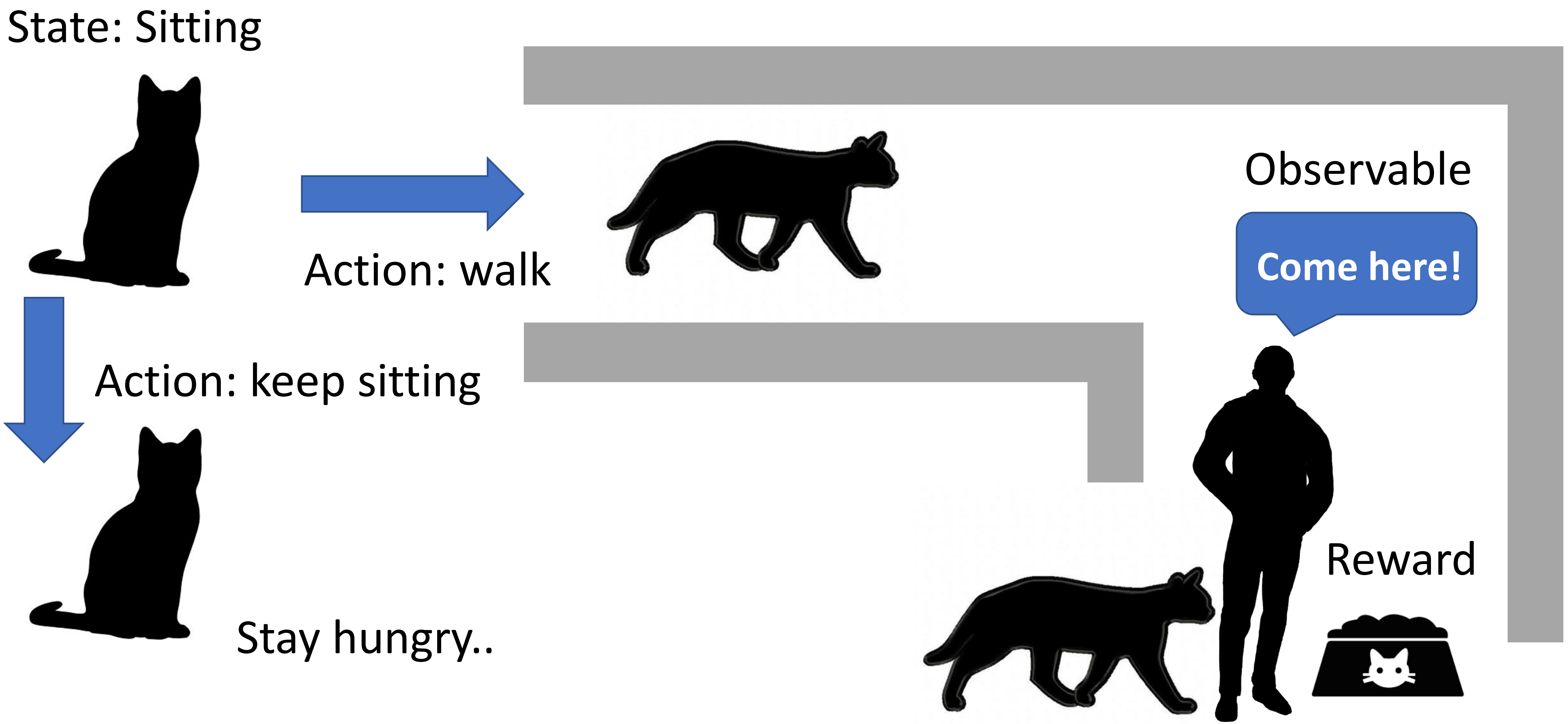
What is Reinforcement Learning?



- In Reinforcement Learning, an **agent** interacts with an **environment** to learn how to perform a particular task **well**.

- How is it different to the other learning paradigms?
  - There is no supervisor, only a **reward.**
  - The agent's actions **affect the subsequent data it receives**
  - **Feedback is delayed**, and may be received after several actions

Cat Agent


State: Sitting
Action: walk
Observable
Come here!
Action: keep sitting
Reward
Stay hungry..

Examples of Reinforcement Learning

Fly a helicopter

Make a robot walk

Manage an investment portfolio

Play Atari games better than humans

Rewards

- The agent receives feedback from the environment through reward
- A reward R_t is a scalar feedback signal
- It is an indication of how well the agent is doing at step t
- The agent's job is to **maximise cumulative reward**
- Examples:
  - Winning a game
  - Achieving design luminosity in a collider
  - Maintaining an inverted pendulum at the top

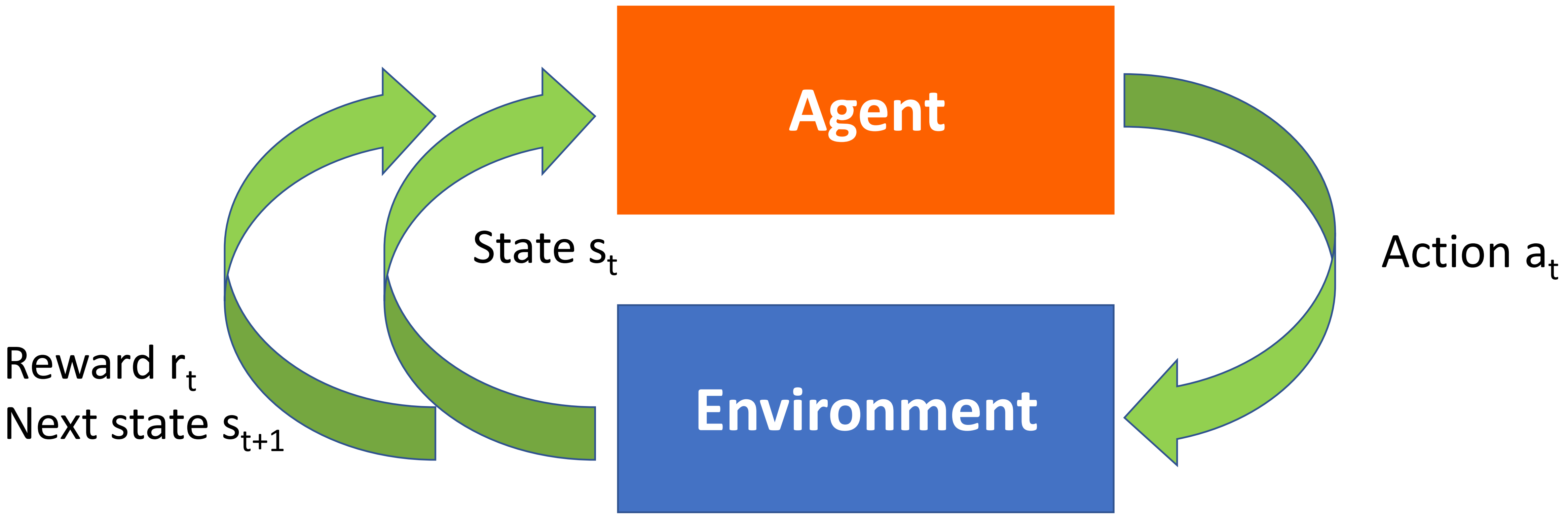
Sequential decision making

- **Goal:** select actions to maximise total future reward
- Actions may have long term consequences
- Reward may be delayed
- It may be better to sacrifice immediate reward to gain more long-term reward
- Examples:
  - A financial investment (may take months to mature)
  - Blocking opponent moves (might help winning probability many moves from now)

States

- **State:** what the agent is observing about the environment
- Examples:
  - Pixels in an image (of a game, of a driverless car, etc)
  - Data from beam instrumentation in an accelerator
  - The position of all pieces in a game of chess

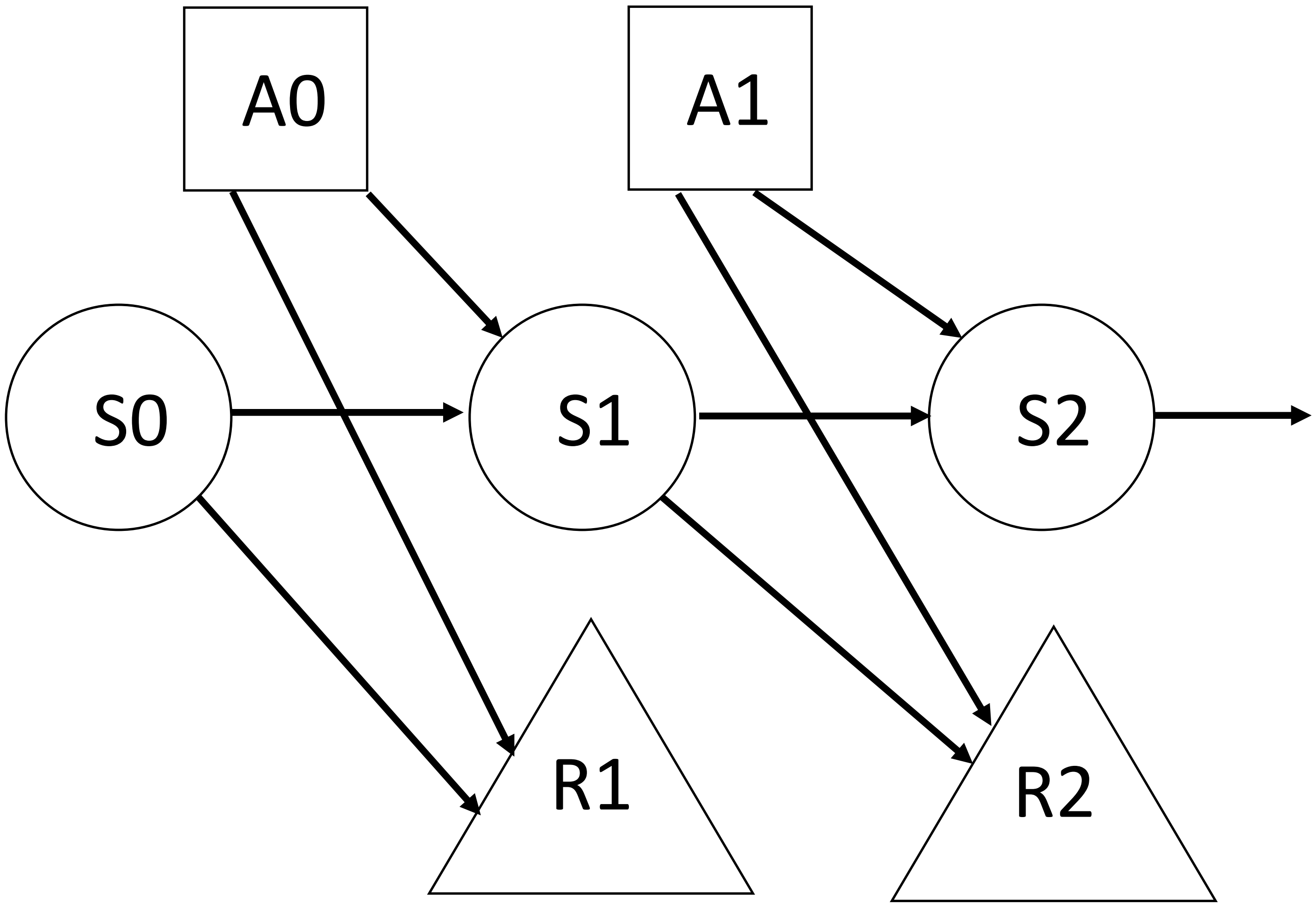
The agent and its environment



How can we formalize this mathematically?

Markov Decision Process (MDP)

- **Markov property:** current state completely characterizes state of the world.
- Defined by: (S, A, R, P, γ)
  - **S:** set of possible states
  - **A:** set of possible actions
  - **R:** reward for a given (state, action) pair
  - **$P(s_t|s_{t-1}, a_t)$:** transition probability
  - **γ:** Discount factor (usually close to 1)



Markov Decision Process (MDP)

- At time step t = 0, environment samples initial state $s_0 \sim P(s_0)$
- Then, for t = 0 until done:
  - Agent selects action a_t
  - Environment samples reward $r_t \sim R(. \mid s_t, a_t)$
  - Environment samples next state $s_{t+1} \sim P(. \mid s_t, a_t)$
  - Agent receives reward r_t and next state s_{t+1}.

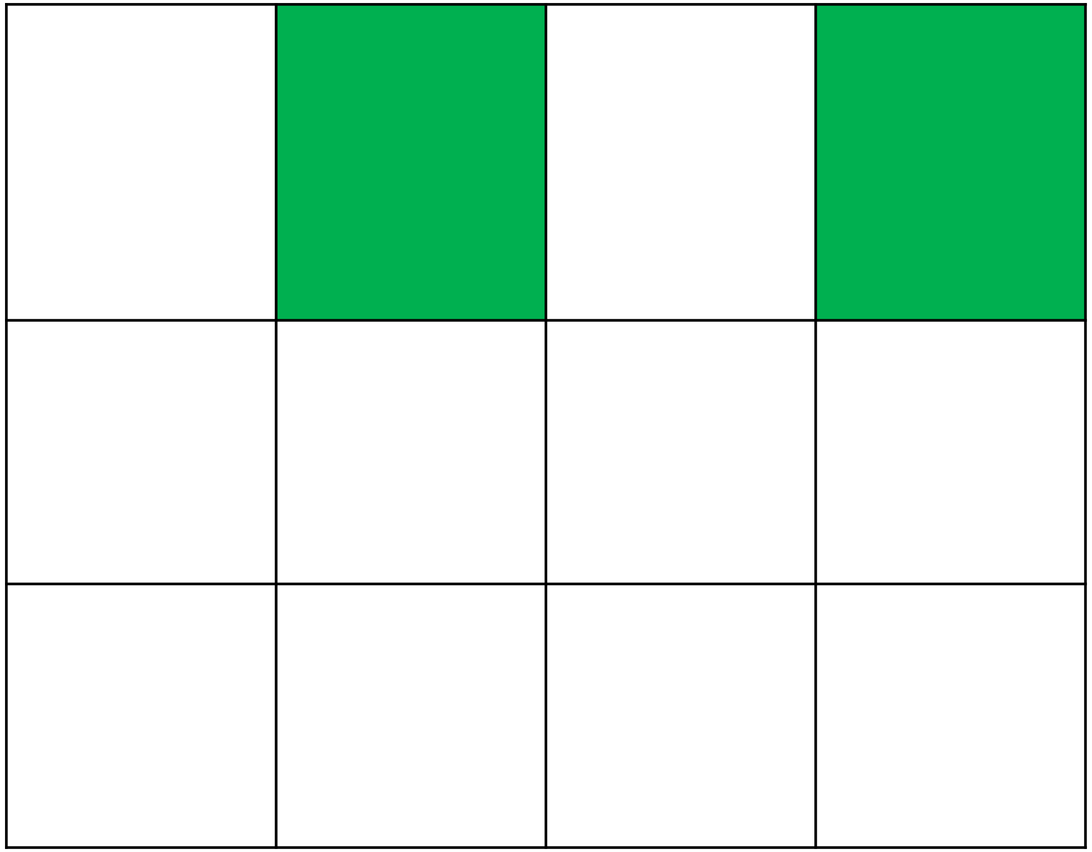
- A policy π is a function which specifies what action to take by the agent in each state.
- **Objective:** find a policy **π*** that maximizes cumulative discounted reward $\sum_{t \geq 0} \gamma^t r_t$

A simple MDP: Grid World

actions = {

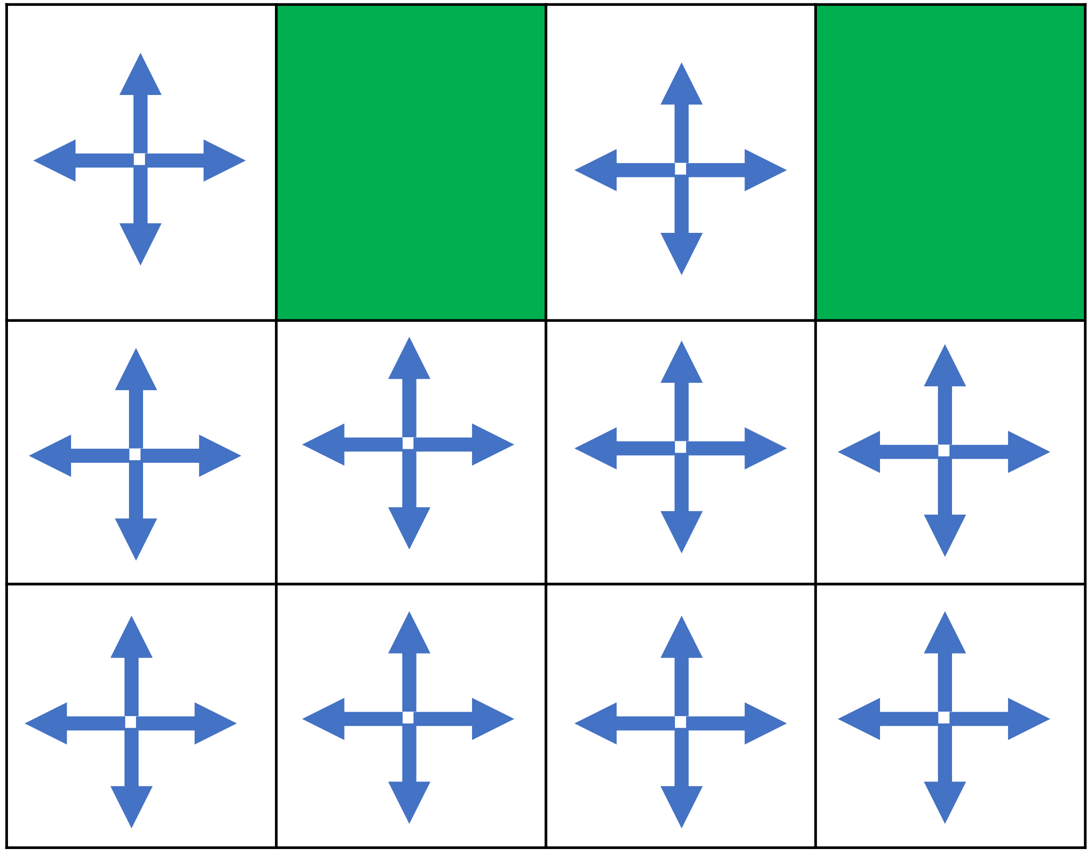
1. right
2. left
3. up
4. down

}

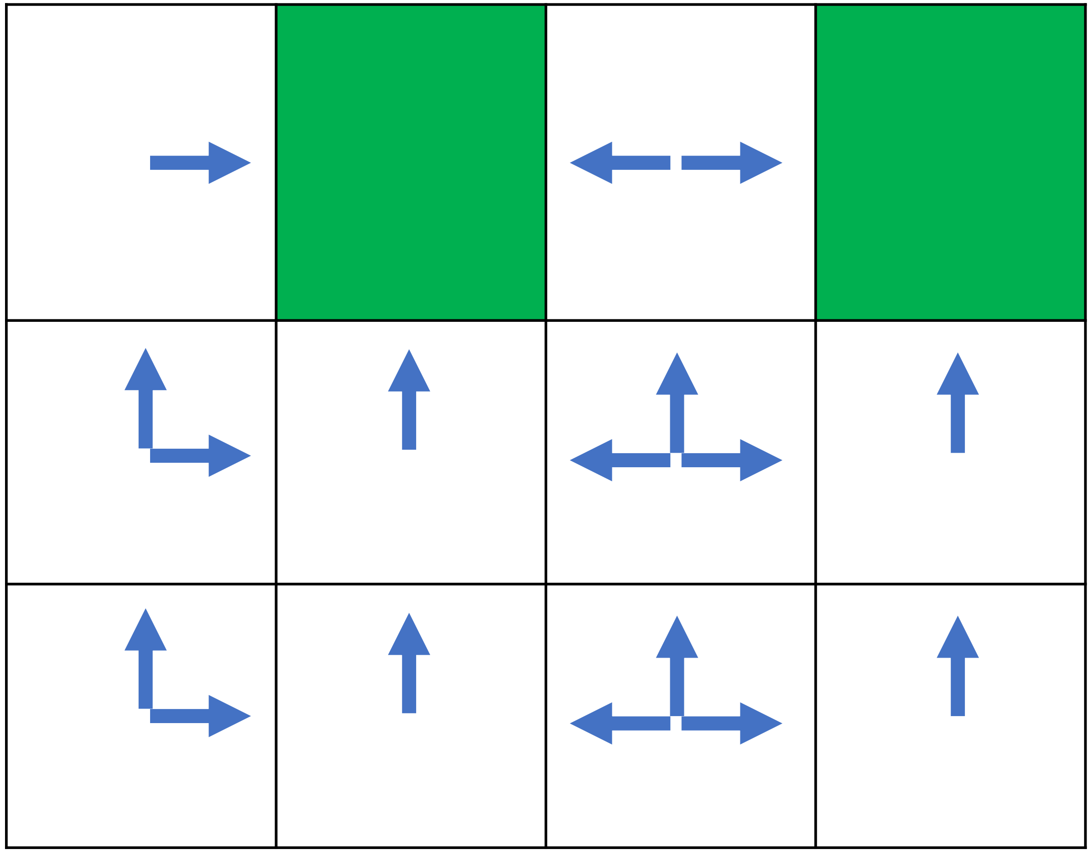


Objective: reach one of the terminal states (green) with the least number of actions

A simple MDP: Grid World



Random Policy



Optimal Policy

The optimal policy π*

- Need to find the optimal policy π* that maximizes the sum of rewards.
- To handle randomness (initial state, transition probability etc):
  - Maximize the **expected sum of rewards**

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | \pi\right] \text{ with } s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$$

Definitions: Value function and Q-value function

- Following a policy produces sample trajectories (or paths) s_0, a_0, r_0, s_1, a_1, r_1, …
- **How good is a state?**
  - The **value function** at state s is the expected cumulative reward from following the policy from state s:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

- **How good is a state-action pair?**
  - The **Q-value function** at state s **and** action a, is the expected cumulative reward from taking action a in state s and then following the policy:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Bellman equation

- The optimal Q-value function Q* is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

- Q* satisfies the **Bellman equation:**

$$Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a')|s,a\right]$$

- Intuition: if the optimal state-action values for the next time-step Q*(s’,a’) are known, then the optimal strategy is to take the action that maximizes the expected value of

$$r + \gamma Q^*(s',a')$$

- Optimal policy π* -> taking the best action in any state as specified by Q*.

Solving for the optimal policy

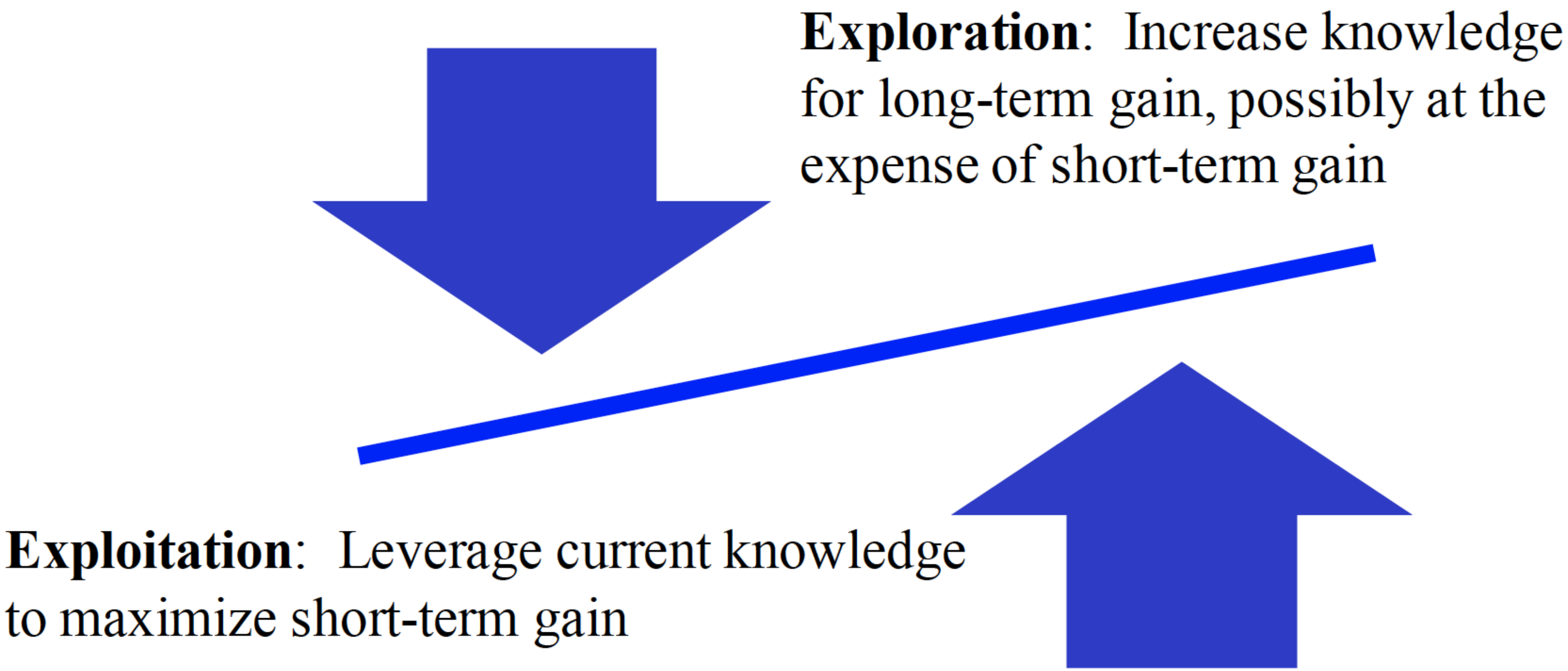
- **Value iteration algorithm:** use the Bellman equation as an iterative update:

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

- Q_i will converge to Q* as i -> infinity.

$$Q^{new}(s_t,a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Exploration vs Exploitation

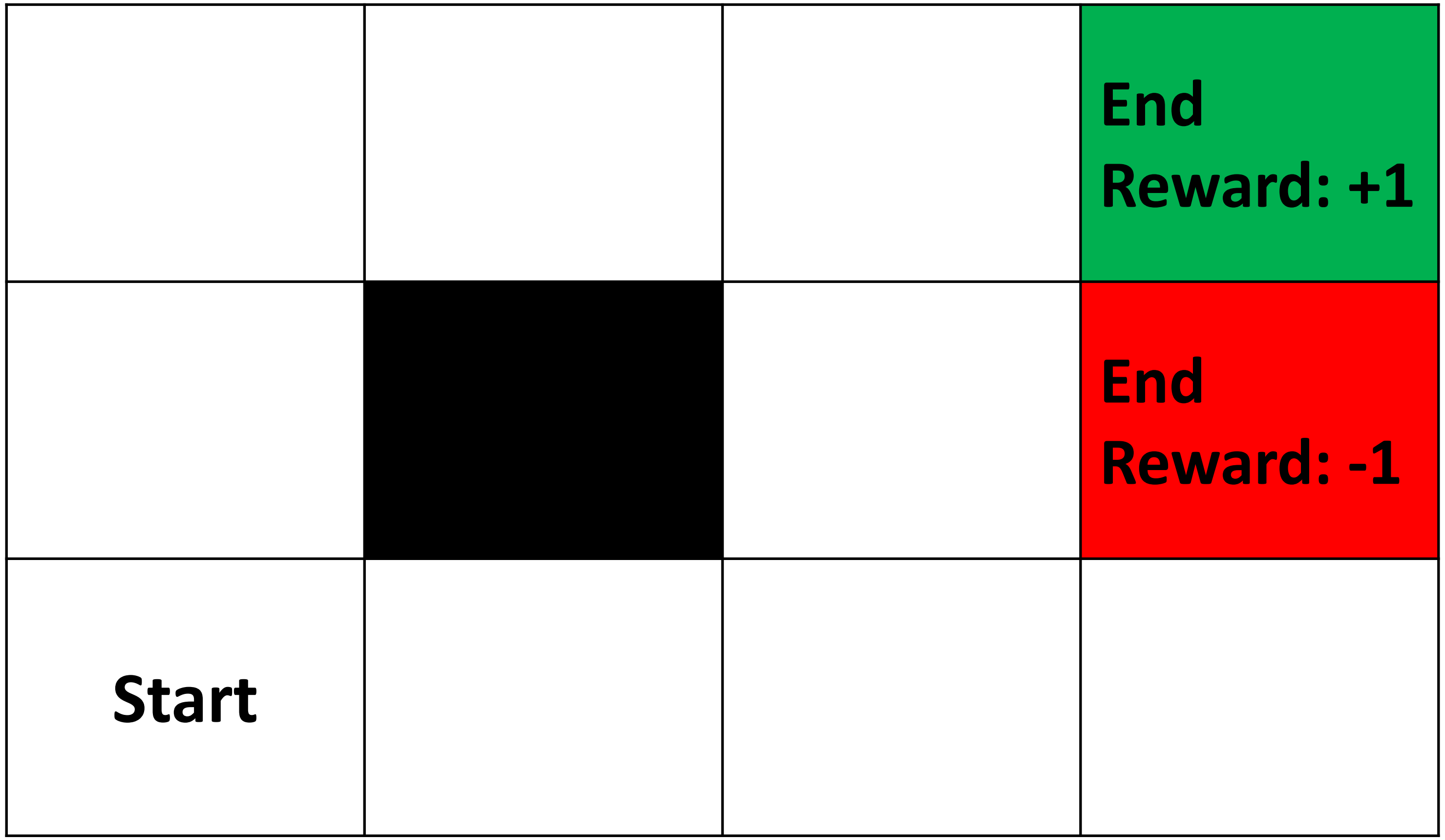


During training, we could e.g.:

30% of the time we choose a random action

70% of the time we choose an action with the most expected value

Grid world example



- Agent starts at bottom left.
- At each step, agent has 4 possible actions (up, down, left, right).
- Black square: agent cannot move through it.
- Assume each action is deterministic.

Grid world example

- First, define the grid world parameters:

```
import numpy as np

BOARD_ROWS = 3
BOARD_COLS = 4
WIN_STATE = (0, 3)
LOSE_STATE = (1, 3)
START = (2, 0)
#DETERMINISTIC = False
DETERMINISTIC = True
```

Grid world example

- Define the reward:

```
def giveReward(self):
    if self.state == WIN_STATE:
        return 1
    elif self.state == LOSE_STATE:
        return -1
    else:
        return 0
```

Grid world example

- Probabilistic result of taking an action:

```
def _chooseActionProb(self, action):
    if action == "up":
        return np.random.choice(["up", "left", "right"], p=[0.8, 0.1, 0.1])
    if action == "down":
        return np.random.choice(["down", "left", "right"], p=[0.8, 0.1, 0.1])
    if action == "left":
        return np.random.choice(["left", "up", "down"], p=[0.8, 0.1, 0.1])
    if action == "right":
        return np.random.choice(["right", "up", "down"], p=[0.8, 0.1, 0.1])
```

Grid world example

- Define how the state is updated when the action is taken by the agent.
- Need to check that the next state is not the black box or else outside the grid.

```python
def nxtPosition(self, action):
    """
    action: up, down, left, right
    -------------
    0 | 1 | 2| 3|
    1 |
    2 |
    return next position on board
    """
    if self.determine:
        if action == "up":
            nxtState = (self.state[0] - 1, self.state[1])
        elif action == "down":
            nxtState = (self.state[0] + 1, self.state[1])
        elif action == "left":
            nxtState = (self.state[0], self.state[1] - 1)
        else:
            nxtState = (self.state[0], self.state[1] + 1)
        self.determine = False
    else:
        # non-deterministic
        action = self._chooseActionProb(action)
        self.determine = True
        nxtState = self.nxtPosition(action)

    #self.showBoard()

    # if next state is legal
    if (nxtState[0] >= 0) and (nxtState[0] <= 2):
        if (nxtState[1] >= 0) and (nxtState[1] <= 3):
            if nxtState != (1, 1):
                return nxtState
    return self.state
```

Grid world example

- Tradeoff between exploration (new info) and exploitation (greedy actions):

```
def chooseAction(self):
    # choose action with most expected value
    mx_nxt_reward = 0
    action = ""

    if np.random.uniform(0, 1) <= self.exp_rate:
        action = np.random.choice(self.actions)
    else:
        # greedy action
        for a in self.actions:
            current_position = self.State.state
            nxt_reward = self.Q_values[current_position][a]
            if nxt_reward >= mx_nxt_reward:
                action = a
                mx_nxt_reward = nxt_reward
        # print("current pos: {}, greedy aciton: {}".format(self.State.state, action))

    if action == "":
        action = np.random.choice(self.actions)

    return action
```

Grid world example

- Define stopping condition:

```
def isEndFunc(self):
    if (self.state == WIN_STATE) or (self.state == LOSE_STATE):
        self.isEnd = True
```

Grid world example

- Bring everything together:

```
def play(self, rounds=10):
    i = 0
    while i < rounds:
        # to the end of game back propagate reward
        if self.State.isEnd:
            # back propagate
            reward = self.State.giveReward()
            for a in self.actions:
                self.Q_values[self.State.state][a] = reward
            print("Game End Reward", reward)
            for s in reversed(self.states):
                current_q_value = self.Q_values[s[0]][s[1]]
                reward = current_q_value + self.lr * (self.decay_gamma * reward - current_q_value)
                self.Q_values[s[0]][s[1]] = round(reward, 3)
            self.reset()
            i += 1
        else:
            action = self.chooseAction()

            # append trace
            self.states.append([(self.State.state), action])
            print("current position {} action {}".format(self.State.state, action))
            # by taking the action, it reaches the next state
            self.State = self.takeAction(action)
            # mark is end
            self.State.isEndFunc()
            print("nxt state", self.State.state)
            print("---------------------")
            self.isEnd = self.State.isEnd
```

Grid world example

- Let’s have a look at test_gridworld_qlearning.ipynb
- http://bit.ly/338nV5e
- After running the notebook, change “DETERMINISTIC” from True to False. What do you notice?

Solving for the optimal policy: Q-learning

- **Value iteration algorithm:** use the Bellman equation as an iterative update:

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

- Q_i will converge to Q* as i -> infinity.

- What is the problem with this?
  - Not scalable: must compute Q(s, a) for every state-action pair. If state is e.g. current game state pixels, computationally infeasible to compute for entire state space!

- Solution: use a function approximator to estimate Q(s,a).
  - A **neural network!**

Solving for the optimal policy: Q-learning

- Q-learning: use a function approximator to estimate the action-value function:

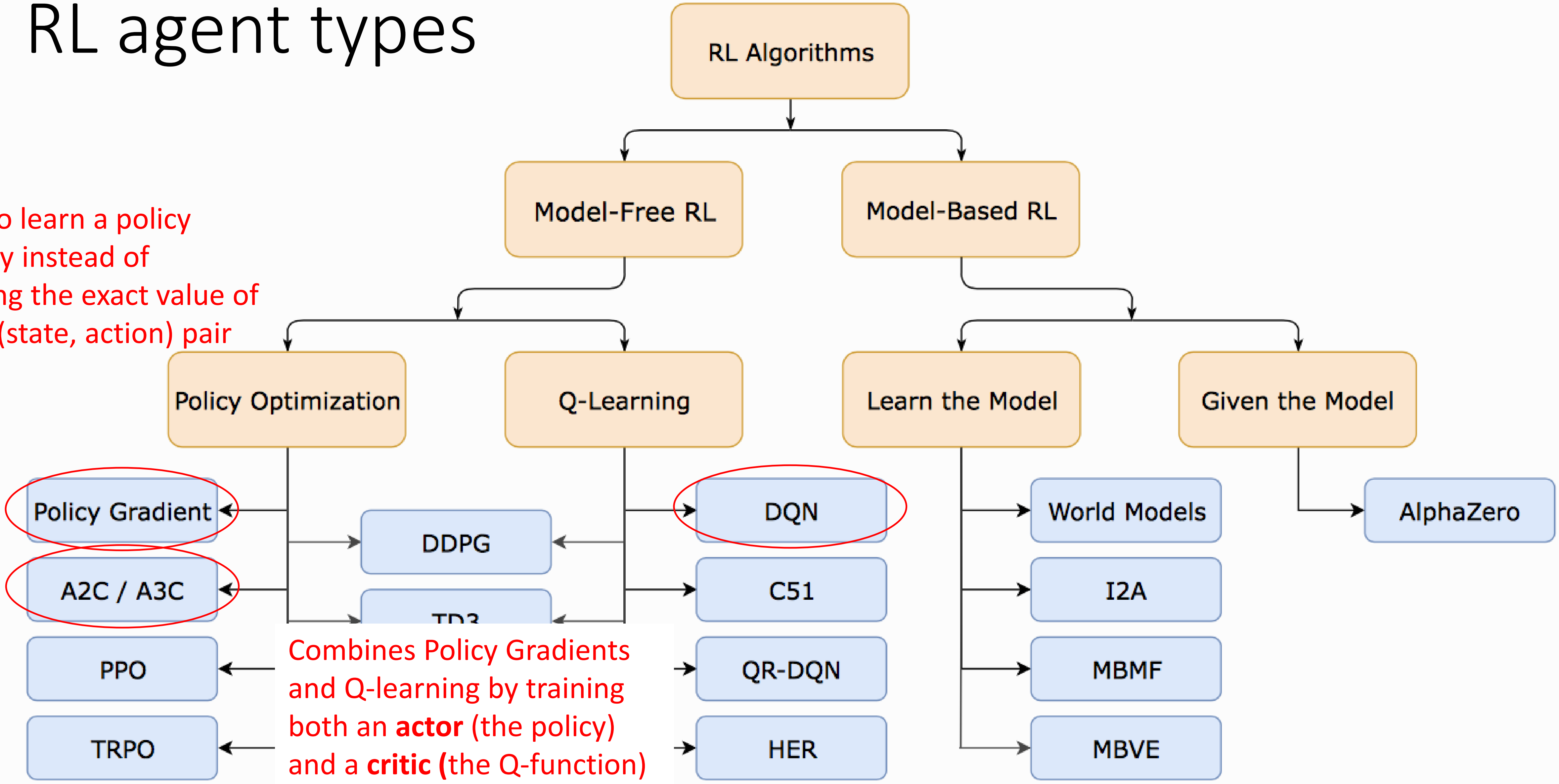
$$Q(s, a; \Theta) \approx Q^*(s, a)$$

Where Θ are the neural network weights which need to be learned.

- If the function approximator is a deep neural network -> **deep q-learning (DQN)!**

RL agent types

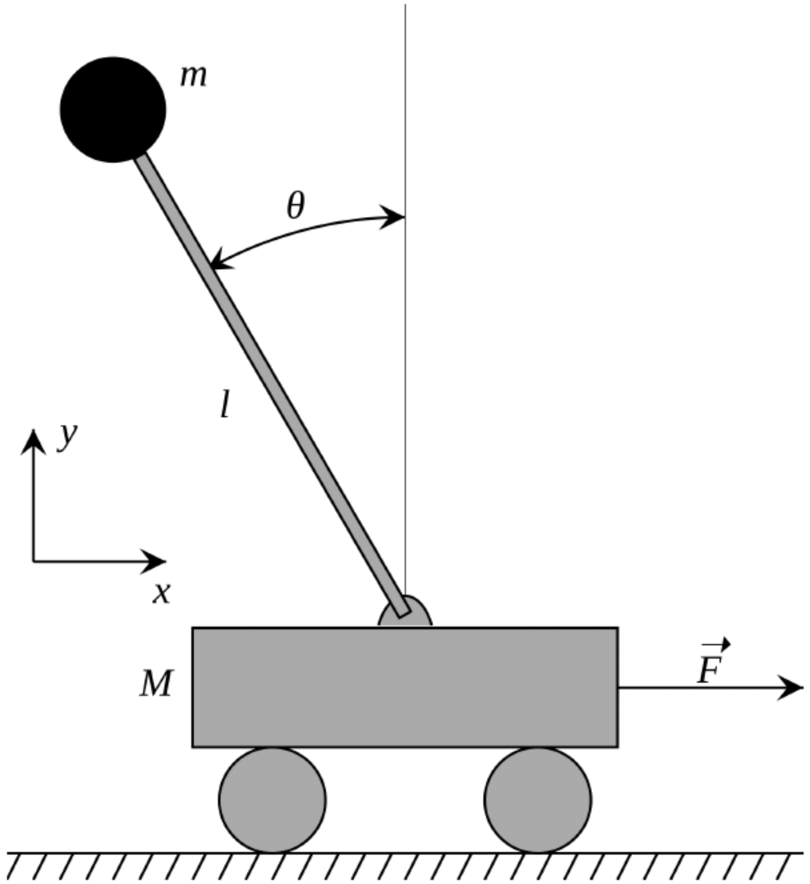
Tries to learn a policy directly instead of learning the exact value of every (state, action) pair



Combines Policy Gradients and Q-learning by training both an **actor** (the policy) and a **critic (**the Q-function) -> 2 neural nets

Source: spinningup.openai.com

Cartpole Problem



- **Objective:** Balance a pole on top of a movable cart

- **State:** angle, angular speed, position, horizontal velocity
- **Action:** horizontal force applied on the cart (or not)
- **Reward:** +1 at each time step if the pole is upright (within some limits)

OpenAI Gym

- In order to train an agent to perform a task, we need a suitable physical environment.
- OpenAI gym provides a number of ready environments for common problems, e.g. Cart Pole, Atari Games, Mountain Car
- However, you can also define your own environment following the OpenAI Gym framework (e.g. physical model of accelerator operation)

OpenAI Gym – Cart Pole Environment

- Let’s have a look at the Cart Pole environment in cartpole.ipynb
- Main component: **step function**
  - Updates state
  - Calculates reward
- Also has rendering functionality

Implementation of a DQN agent

- There are several ready implementations of RL agents
  - E.g. Keras RL

- We first define the Q network architecture (in Keras fashion):

```
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(16))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
print(model.summary())
```

Implementation of a DQN agent

- We can use a ready-made policy (BoltzmannQPolicy)
  - Builds a probability law on q-values and returns an action selected randomly according to this law.

- We also define the number of actions, the learning rate and the number of steps that we want to train the agent for, trying to optimize some metric.

- Memory: stores the agent’s experiences
- Number of warmup steps: avoids early overfitting
- Target Model update: how often are weights of target network updated

```
memory = SequentialMemory(limit=50000, window_length=1)
policy = BoltzmannQPolicy()
dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory, nb_steps_warmup=10,
               target_model_update=1e-2, policy=policy)
dqn.compile(Adam(lr=1e-3), metrics=['mae'])

history = dqn.fit(env, nb_steps=100, visualize=True, verbose=2)
```

Rendering the training of the agent

- Google Colaboratory does not support OpenGL through the browser

- I had to:
  - modify rendering.ipynb to avoid using pyglet.gl
  - modify cartpole.ipynb to render via matplotlib

- If you were to download the notebook on your laptop, you can do without the above two notebooks and directly pass the OpenAI gym environment name as a string.

Let’s try to train DQNAgent for Cart Pole!

Summary

- **Reinforcement Learning:** an agent learns to perform a task from its interactions with its environment
- Formulation as a Markov Decision Process
- Definitions of state, reward, policy, action
- Concepts of value function and Q-value function
- Q-learning and DQN