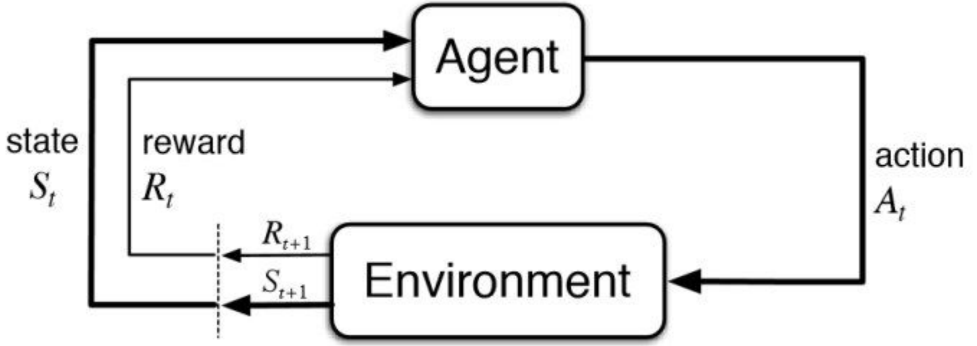
Reinforcement Learning

Olivier Teboul
Google Brain, Paris

Some links and bibliography

- Reinforcement Learning: an introduction (Sutton and Barto)
- Open AI Spinning up in Deep RL

The problem: Markov Decision Process



At each **time step**, the agent is in (perceives) a **state** s(t), and choose an **action** a(t) to interact with its **environment**.

The environment moves the agent to a state s(t+1) and **rewards** it with r(t+1).

The goal of the agent is to choose the sequence of actions that **maximizes the long term return**: the sum of all the rewards.

MDP’s nomenclature

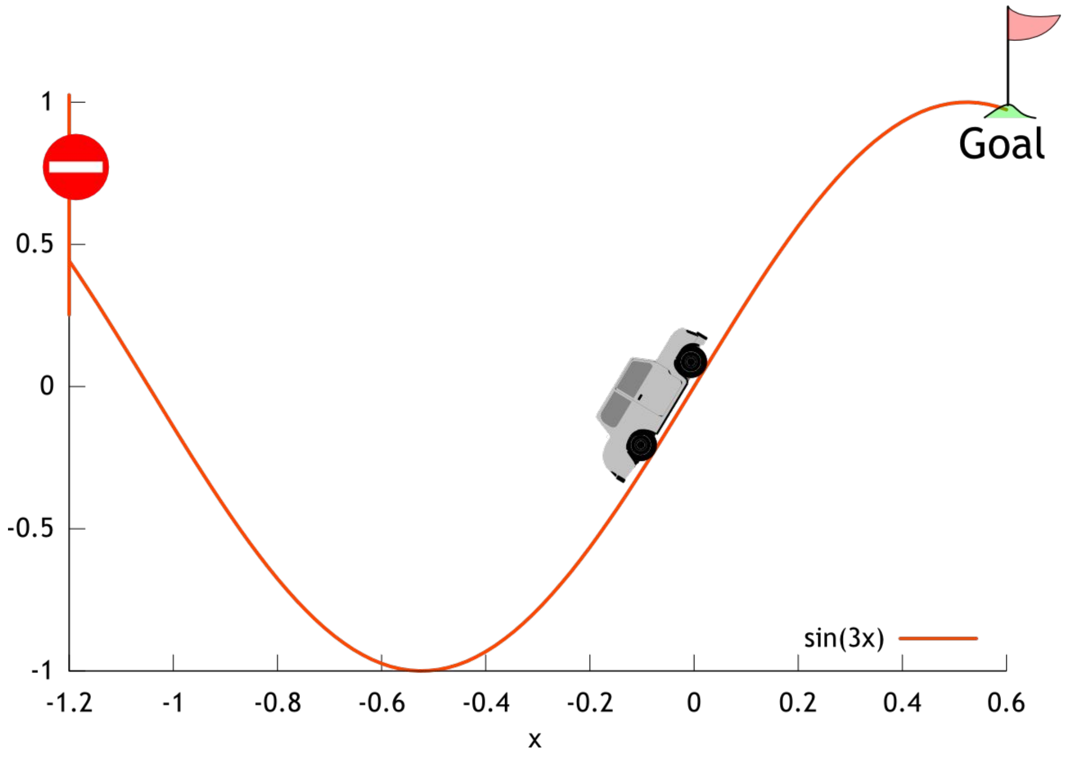
- The state space can be discrete or continuous
- The action space can be discrete or continuous
- The transitions between states can be stochastic (and unknown): **p(s’ | a, s)**
- The rewards can be stochastic
- The **horizon** of the process can be finite (i.e. game play) or infinite (number of steps till the end)
- The return can be discounted or undiscounted.
- We call an **episode** a sequence of actions from the first state to the horizon
- There exists variations of MDPs such as semi MDPs or partially observable MDPs.

Examples of MDPs

- Games are (single, multi players) are typically MDPs.
- Other tasks can be seen as a game: in robotics
- Planning: how much to produce based on demand (agriculture, industry)
- Finance: buying or selling on the stock market
- Self-driving cars

Classical problems in Reinforcement Learning

- Grid Worlds
- Mountain Car
- Black jack
- Backgammon
- Car racing
- Etc.



The problem: Markov Decision Process

The return at time step t, is the sum of all rewards that will be gained from t on:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

γ is the discount factor - Interpretation : a future gain r(t+k) value for γ r(t+k) now (how much we value the future)

- If γ=0, short term vision
- If γ=1, long term vision

We want to maximize the expected return E[Gt]

Policy

A policy π is a mapping from states to probabilities of selecting each possible action. An agent follows a policy.

π(s, a) is the probability that the agent chooses action a while in state s.

Under the policy π, the expected return of being in state s and following a is:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

Q-value of a policy : quality of the pair (s,a) for a fixed policy π.

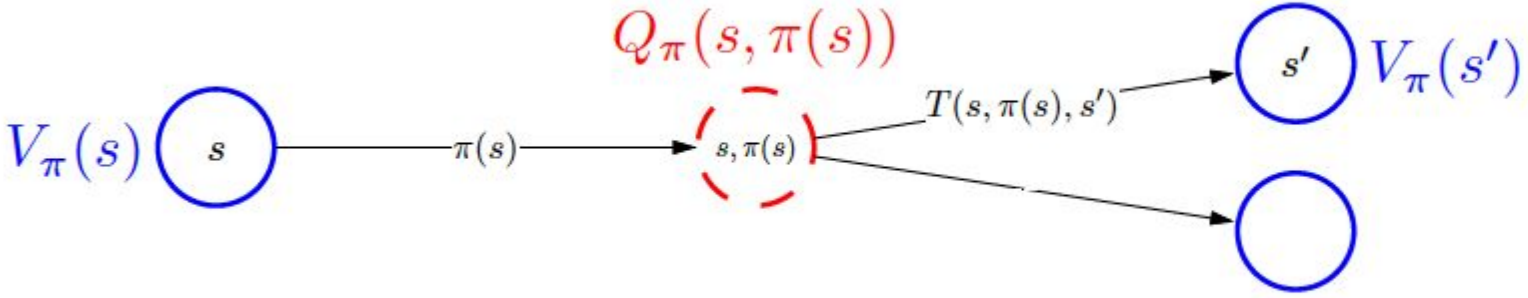
Policy evaluation

Value of a policy $V_\pi(s)$

Expected utility received by the policy π from state s (espérance du critère lorsque qu'on part de l'état s et on suit la politique π)

Q-value of a policy $Q_\pi(s,a)$

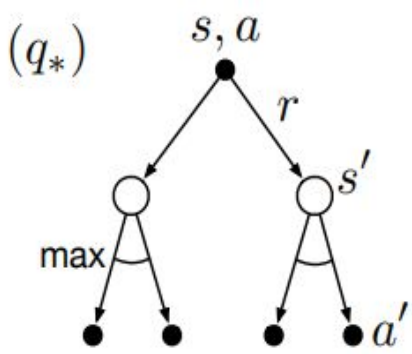
Expected utility of taking action a from state s and following the policy π



Bellman (optimality) equation

The optimal policy maximizes the long term expected return and for this policy the q* function satisfies:

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \,\Big|\, S_t = s, A_t = a\Big] \\
&= \sum_{s',r} p(s', r \,|\, s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big].
\end{aligned}
$$



From value to policy

If I can solve the Bellman equation, I can derive from Q^* the optimal policy π^*

Example: let assume the agent is in state **s**, and can choose among 3 actions, and we know that :

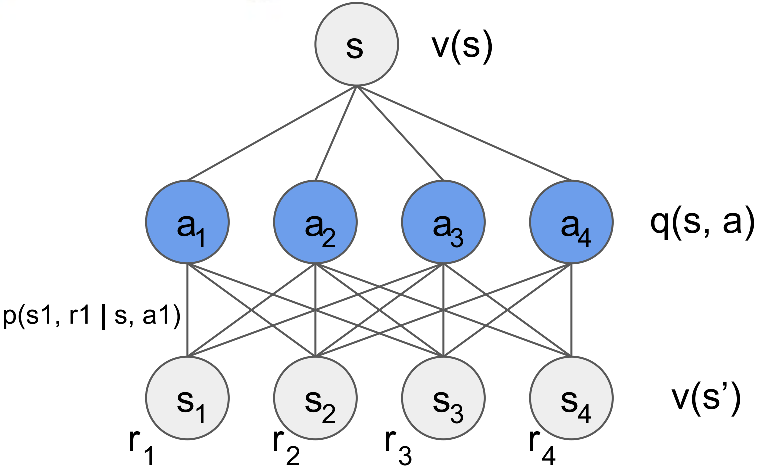
$$\left.\begin{aligned} Q^*(s, a1) &= 10 \\ Q^*(s, a2) &= 100 \\ Q^*(s, a3) &= -1 \end{aligned}\right\} \Rightarrow \pi^*(s) = a2 \quad \text{(with probability 1.0)}$$

Therefore, our goal is to estimate Q^*

Bellman optimality equation for states

$$
\begin{aligned}
v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_*(s')\Big]
\end{aligned}
$$

Same exact idea as the Dijkstra algorithm. The shortest path is made of shortest paths.



how to solve it?

Dynamic Programming

- Start from V(s) = 0 for all states
- **Use Bellman equation as an update sweeping over the states**
  ⇒ Use all the estimate of V(s') to update the estimates of V(s)

- Iterate until convergence
  ⇒ derive the optimal policy from the v

- Can be synchronous or asynchronous, in a specific order to minimize the number of sweeps or total in a random order. It converges.
- Can also be done on q, even better.

Algorithme d'itération de la valeur

Initialiser V_0 aléatoirement
$i \leftarrow 0$
Répéter
  Faire $\forall s \in S$
    $V_{i+1}(s) \leftarrow \max_{a \in A(s)} \sum_{s' \in S} P(s,a,s')[R(s,a,s') + \gamma V_i(s')]$
    $i \leftarrow i + 1$
Jusque $\|V_i - V_{i-1}\| \leq \epsilon$
Faire $\forall s \in S$
  $\pi(s) \leftarrow argmax_{a \in A(s)} \sum_{s' \in S} P(s,a,s')[R(s,a,s') + \gamma V(s')]$

Limitations of DP

- Dynamic Programming is great
- Dynamic Programming converges

But…

- It suffers the curse of dimensionality, when the state space is too big because the update rule has to take every possible action at every possible state.
- The transition probabilities (s, a) → (s’, r) must be known.
- It spends a lot of time estimating an exact value of a function everywhere… just to get its maximum

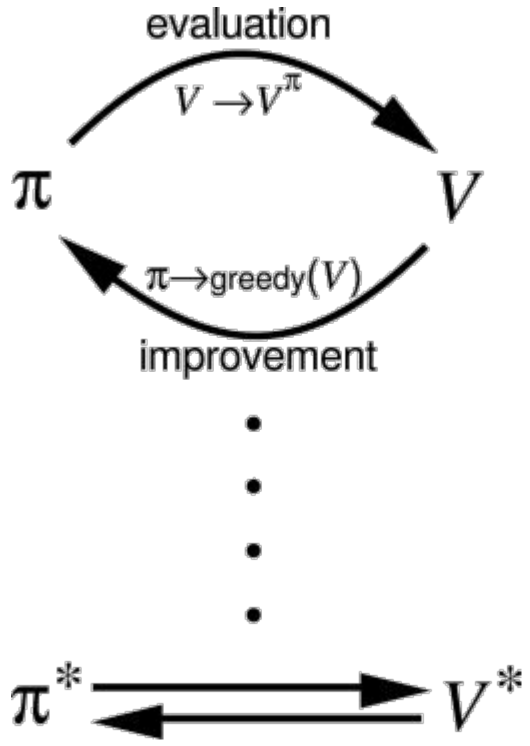
Monte-Carlo : sampling

- Monte Carlo samples one action at a time, **using a given policy π**
- The idea: follow one branch of the tree till the end. Once you reach the end of the episode, you get a sample of the a return.
- The expected return is obtain by Monte-Carlo: repeat many time the sampling.

⇒ **the expected return is estimated by averaging the <u>empirical</u> returns**

Policy Iteration

Starting from a policy, estimate the value function using it. Then improve the policy using the new estimation of V, and keep looping till convergence.



Temporal Differences: the key idea of RL

The key ideas in RL are borrowed from DP and MC:

- MC: sample one action at a time.
- DP: update the estimate of q based on other estimates of q.

The temporal differences methods that are at the core of RL do both sampling, and update the estimates after one step only, based on other estimates.

Q-Learning update equation

Lies somewhere in between Dynamic Programming and Monte-Carlo

- As Monte-Carlo: sample one action at a time.
- As Dynamic Programming: update the estimate of q based on other estimates of q.

While in state St, following action At, the agent ends up in St+1 and receives Rt+1.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

Q-learning keeps improving its long term prediction based on short term moves.

Link with Stochastic Gradient Descent

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

new estimate of Q(s, a) - old estimate of Q(s, a)

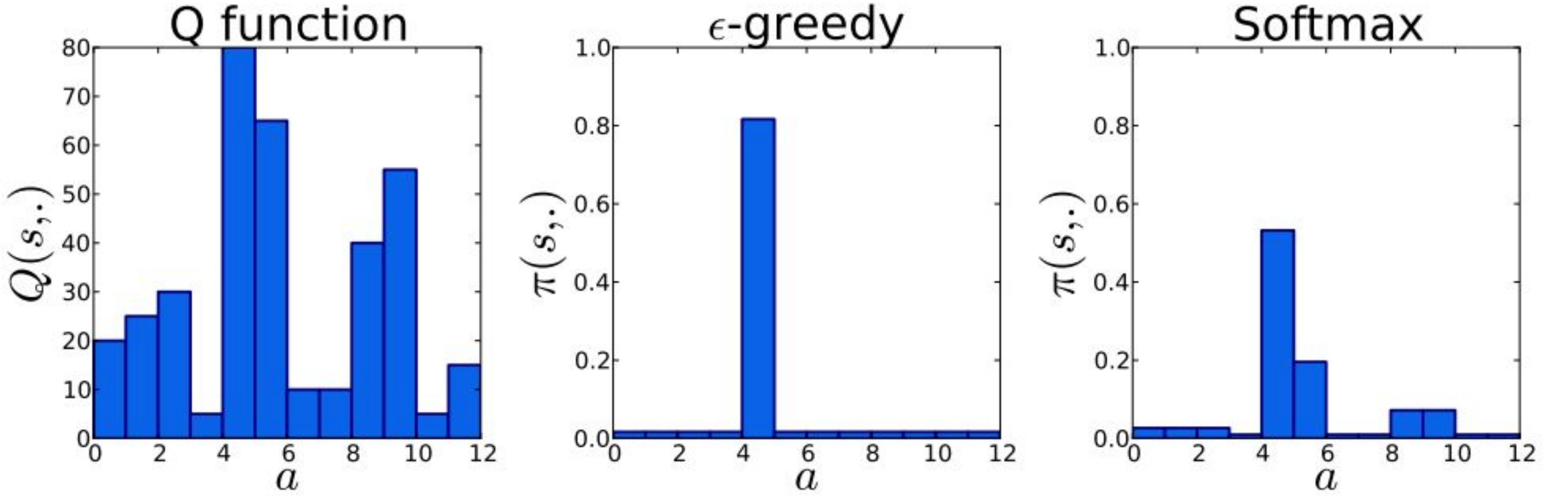
learning rate

- estimation error

How to sample the next action ?

Based on the current estimate of Q(s, a)

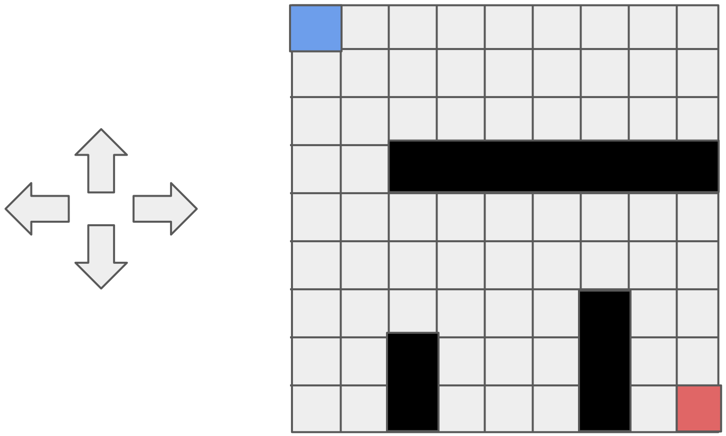
⇒ **exploration-exploitation trade-off**



Example

Grid World

On a grid 50x50 with some obstacles, how to train an agent that can only move Left, Right, Up, Down to go from the top left corner to the bottom right ?



Modeling

state: (i, j) ⇒ position on the grid of the agent.

actions: [left, right, up, down]   ⇒ impossible actions lead to the same state.

episode: ends when the agent reaches the bottom-left corner.

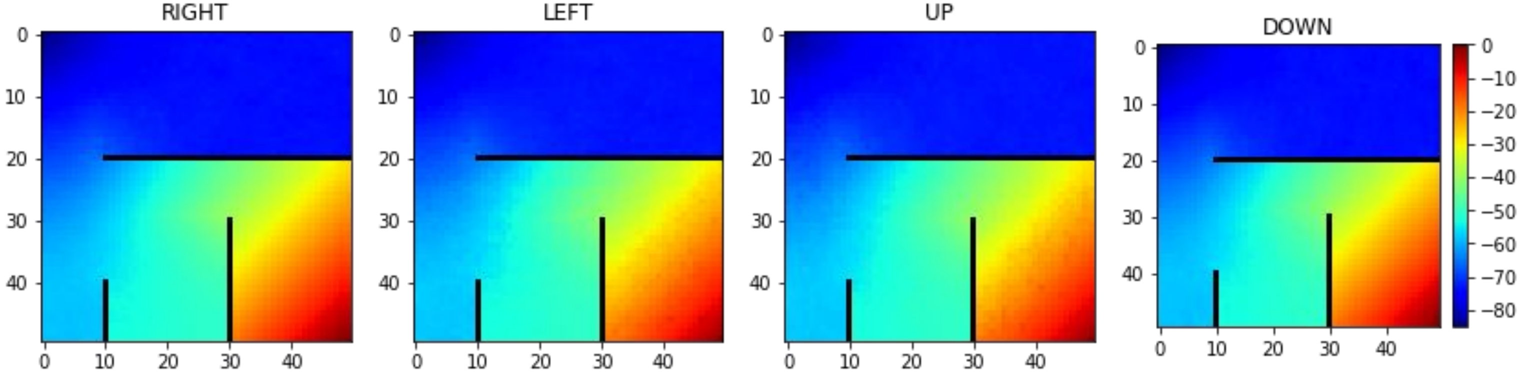
parameters: gamma=1, epsilon=0.20, alpha=0.5

reward: -1 for each actions ⇒ train the agent to find the exit as soon as possible.

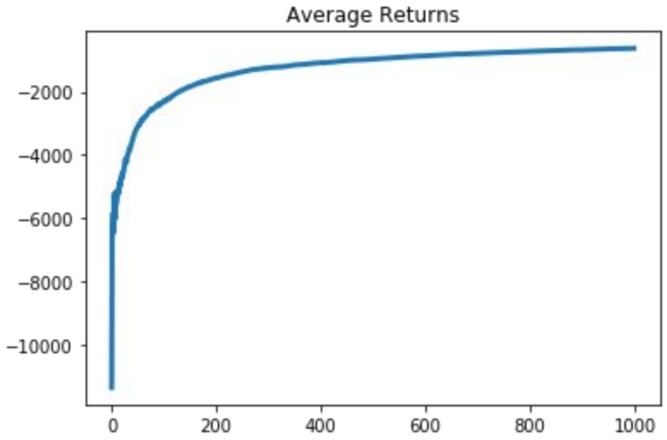
Train over 1000 episodes with Q-Learning

Results

Q(s, a)



RIGHT
LEFT
UP
DOWN
0
10
20
30
40
0
−10
−20
−30
−40
−50
−60
−70
−80

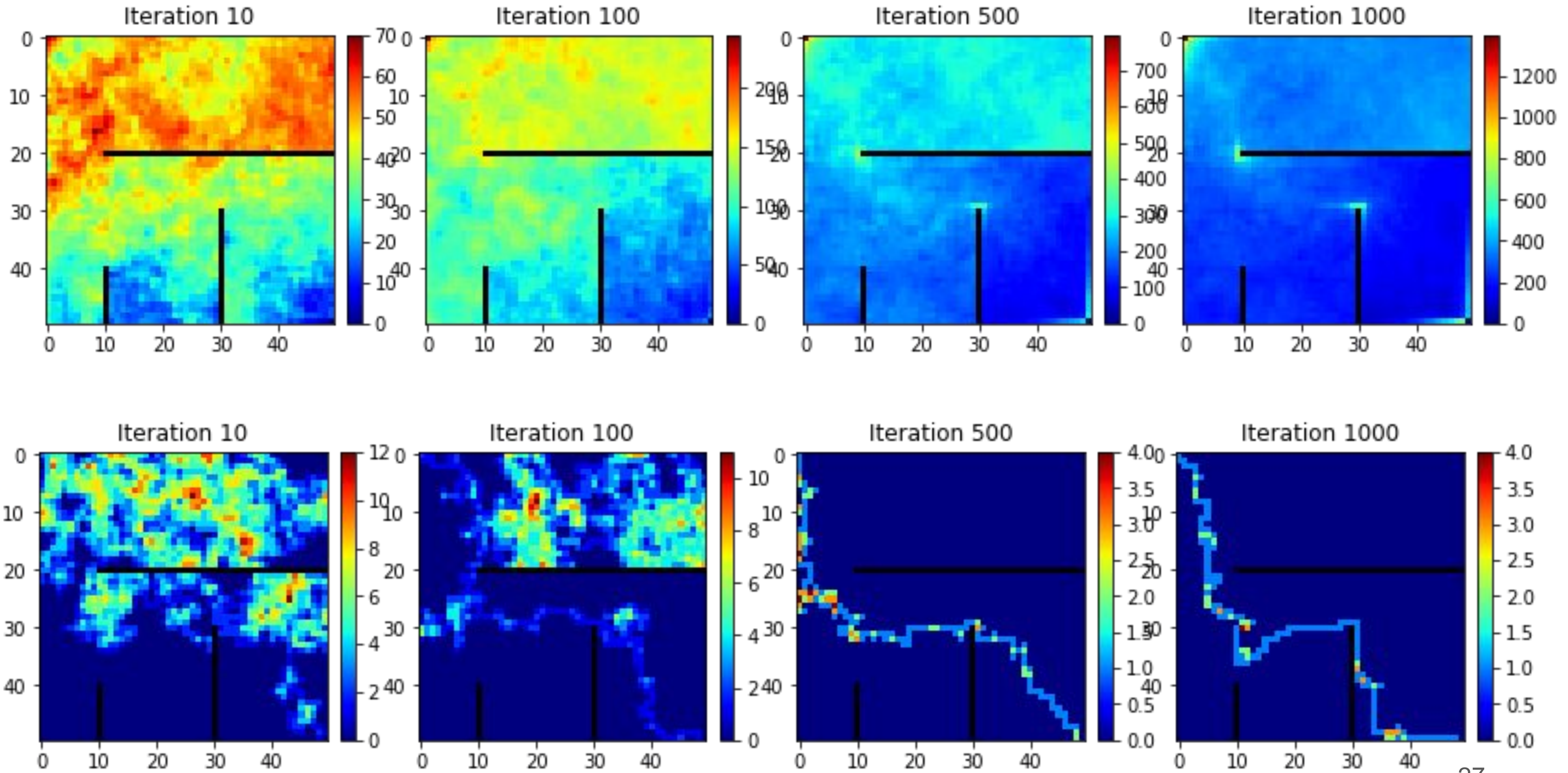


Average Returns
−2000
−4000
−6000
−8000
−10000
0
200
400
600
800
1000

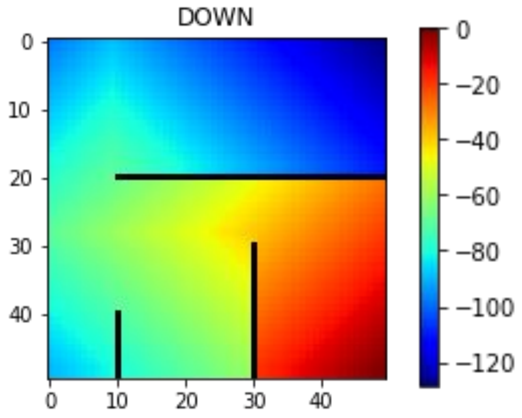
Looking at Visits

Total number of visits in each cell

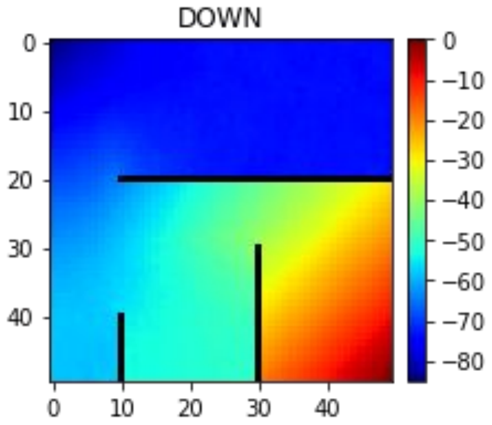
Visits for a given episode



Comparing Q-Learning and Dynamic Programming



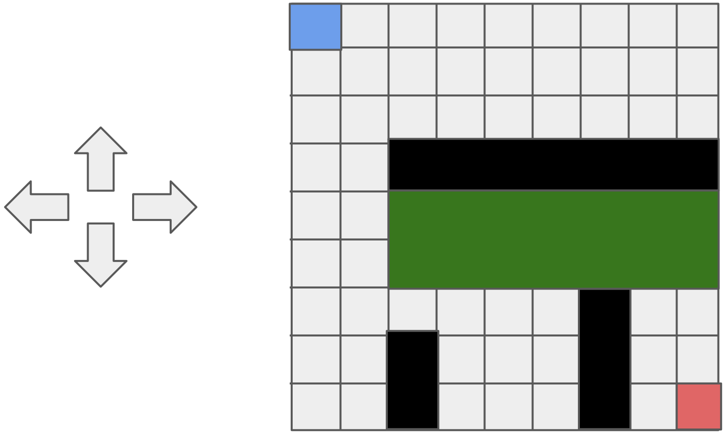
Dynamic Programming



Q-Learning

Stochastic Grid World Example

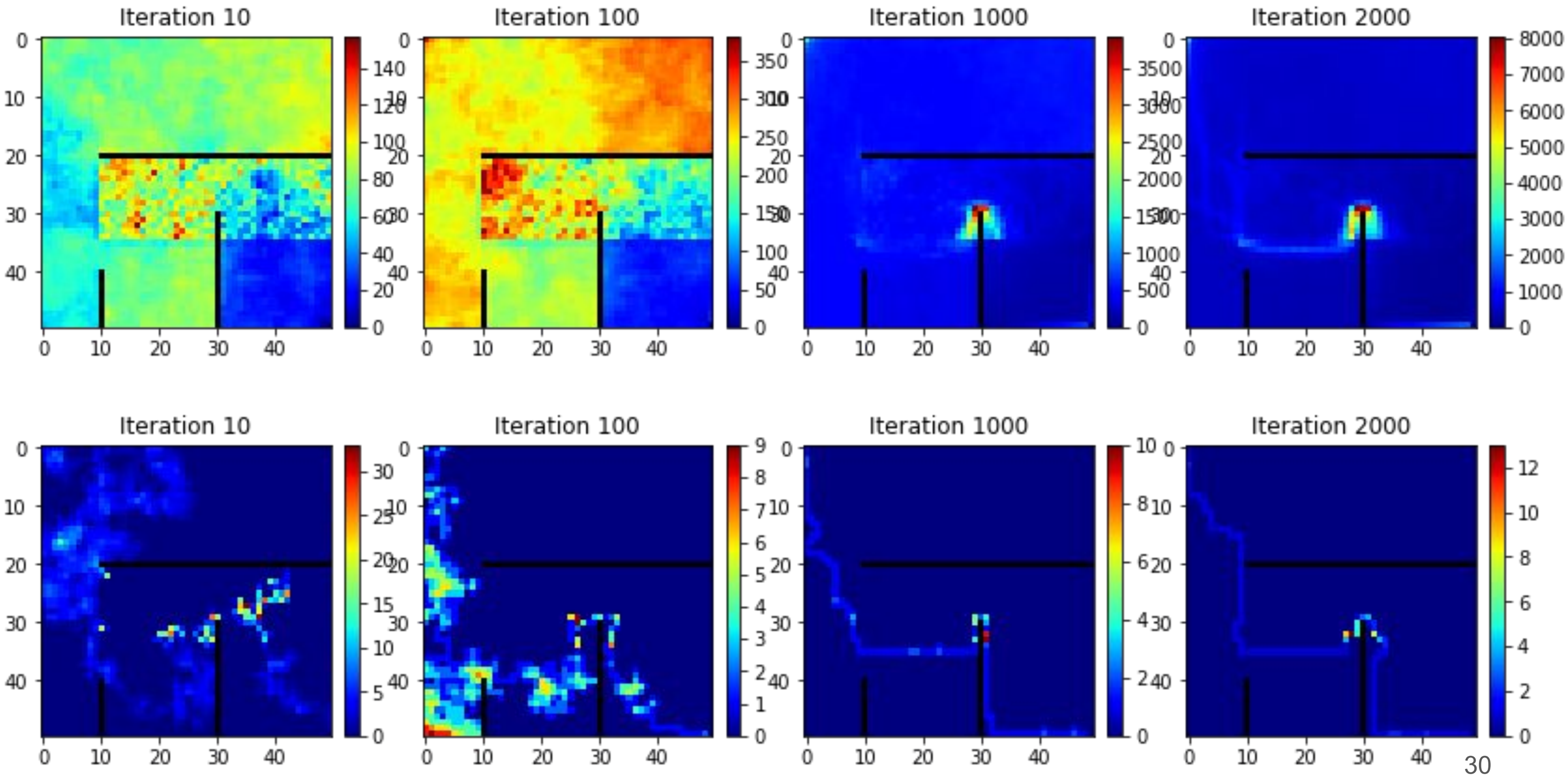
In a specific region of the grid, the probability to get stuck in the same cell is 70%



Looking at visits in this stochastic world

Total number of visits in each cell

Visits for a given episode



Limitations of classic Q-Learning

The number of states can become quite big or even continuous and the states and actions among them may have some correlations, that plain Q learning does not capture.

It would be better to find a more compact and learned representation of Q(s, a)

→ neural networks are good at finding representations !

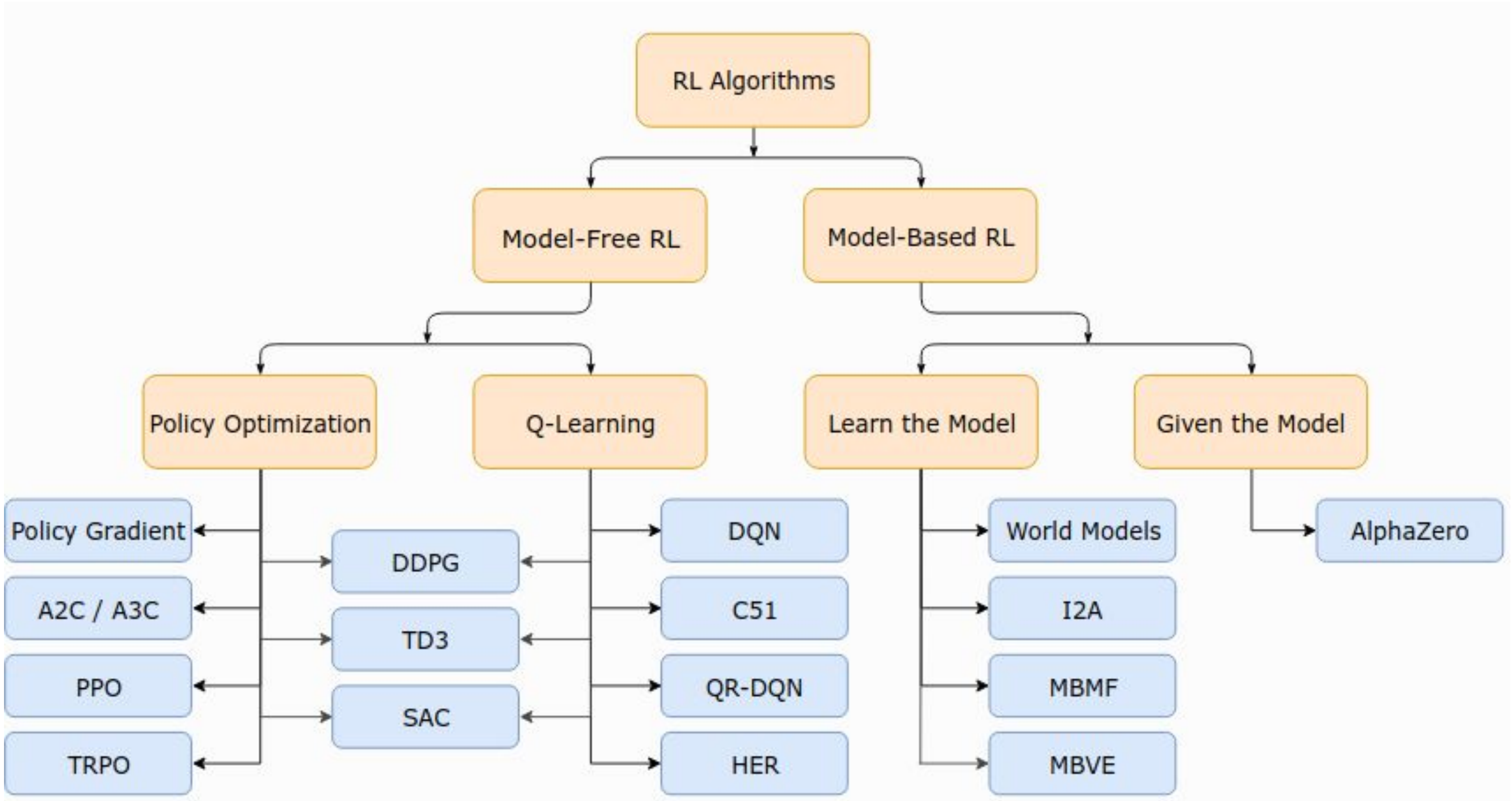
Enable to have many states as input that are now well understood in DL: images, audio, etc.

Deep RL

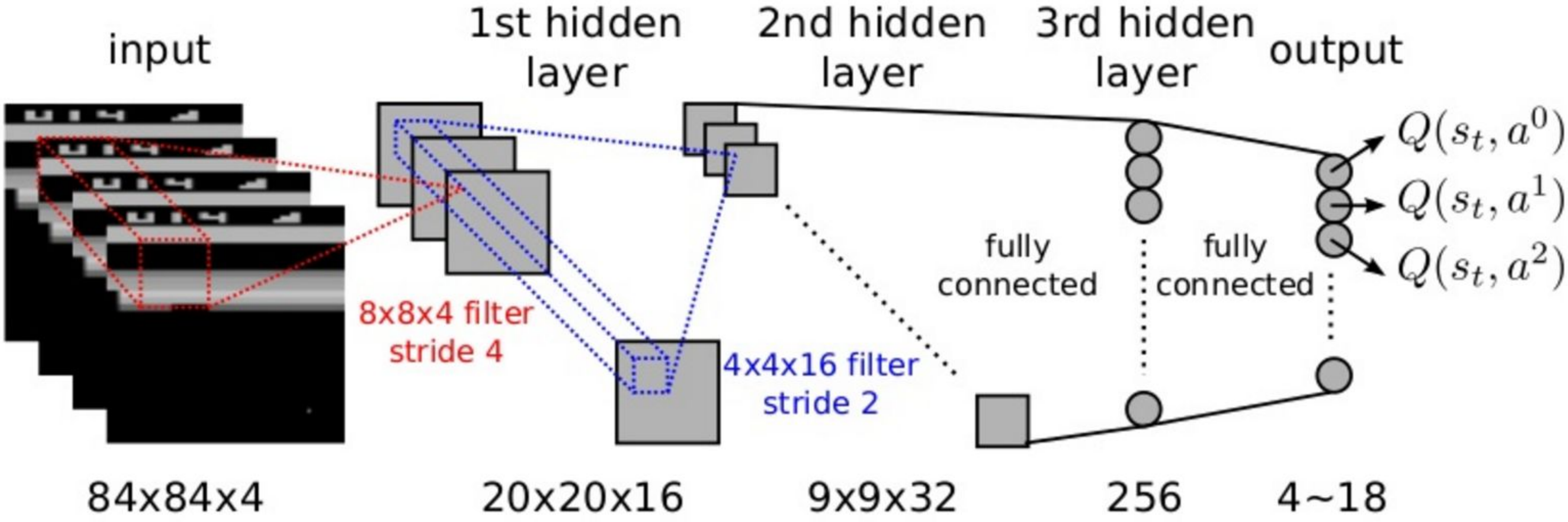
DQN: Deep Q Learning with Replay

- The network has as many output as possible actions.
- Keep playing use the current network as a **prediction** for *a* at each *s* (coupled with **e-greedy policy**)
- **Remember** each (s, a, r, s) tuple when playing.
- **from time to time**, do a **replay**: train the network
  - Random-sample from the memory some input-output pairs
    ⇒ break correlations between subsequents samples
  - target = reward + γ maxQ(s, a) (using the prediction from the network)
  - train the model with the input / output as state - target

The RL algorithms family



Atari games: learning from frames



Atari games: learning from frames

Robotics

Other methods such as the Actor-Critic enables to deal with continuous states and continuous actions.

Challenges in RL

- Exploration (especially in sparse environments)
- Catastrophic forgetting
- Off-policy learning
- Convergence: sample efficiency, variance, cost, ...
- Network architectures for RL tasks
- Safe RL
- Real-world RL
- ...

Lab: self-driving car

A “car” is equipped with sensors that tells the distance from the front in several direction, to the limit of the circuit.

Can you find a way for the car to learn how to stay in the circuit and close the lap as fast as possible ?