*** Applied Machine Learning Fundamentals ***
Reinforcement Learning

M. Sc. Daniel Wehner

SAP SE

Winter term 2019/2020

Find all slides on GitHub

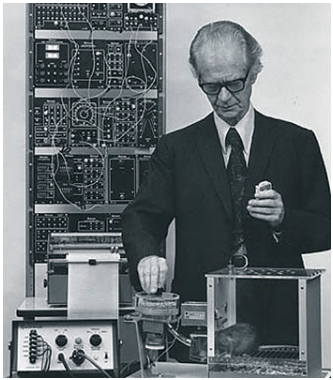
Agenda for this Unit

Section:

Introduction

Psychological Motivation

- B. F. Skinner (1904 – 1990)
- **Operant conditioning** (behaviorism)
  - Associative learning
  - Strength of a behavior is modified by reinforcement or punishment
  - Operant behavior is **voluntary**
- The image on the right depicts the 'Skinner box'



Psychological Motivation (Ctd.)

Operant conditioning	Punishment	Reinforcement
Positive	add noxious stimulus to decrease behavior	add pleasant stimulus to increase behavior
Negative	remove pleasant stimulus to decrease behavior	remove noxious stimulus to increase behavior

What is Reinforcement Learning?

- Agent $\Leftrightarrow$ (Unknown) Environment
- **Goal**
  - Learn optimal decisions based on feedback provided by the environment
  - Environment rewards agent for actions but **does not (!)** reveal the correct solution (reward can be positive or negative)
- **Applications**
  - Games, e. g.:
    - Tic-Tac-Toe: MENACE [*Michie*, 1963]
    - Backgammon: TD-Gammon [*Tesauro*, 1995]
  - Other: E. g. `robot control` (btw.: $robot_{en}$ derived from $robota_{cz}$ – labor service)

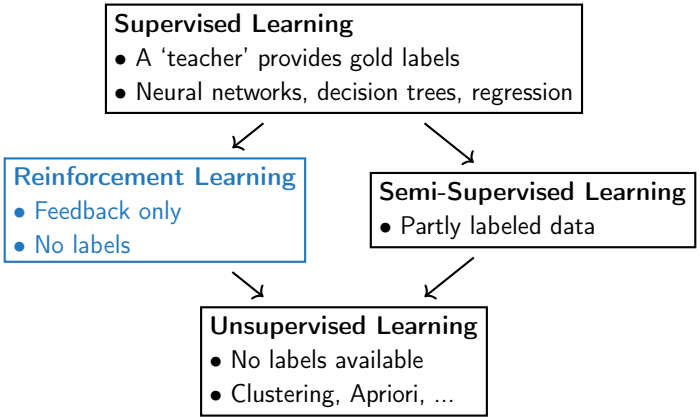
Reward Hypothesis

- All goals of an agent can be explained by a single scalar called the reward
- It is the RL practitioners' task to find the right set of rewards
- This is referred to as **reward shaping**

Reward hypothesis:

What we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

Dimensions of Learning: Type of Training Information



Difference to other Approaches

1. **Delayed rewards**
   - Sequence of actions $\Rightarrow$ Problem of **temporal credit assignment**
   - Which actions were good? Which were bad?
2. **Exploitation vs. exploration**
   - Should the agent perform actions which are known to be good...
   - ...or should it try out new (possibly even better) actions?
3. **Partially observable states** (not everything my be observable)
4. **Life-long learning**
   - Environment may not be static, it could change!
   - Agents need to adapt to these changes...

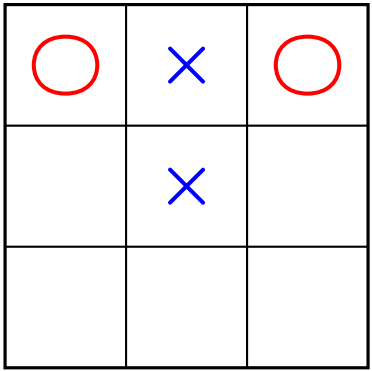
Credit Assignment Problem

- **Fundamental problem in reinforcement learning**
  - Especially in games: Reward at the end of the match (have I won or lost?)
  - Central Question: **Which moves contributed to winning / losing?**
  - This problem is referred to as the **credit assignment problem**
- **Solution:**
  - All moves of the sequence are rewarded / punished equally
  - After many games the algorithm converges (bad moves will be reinforced less positively, good moves will be positively reinforced more often)

MENACE: Learning to play 'Tic-Tac-Toe'

- Introduced by [*Mitchie*, 1963]
- Learns to play 'Tic-Tac-Toe'
- MENACE stands for:

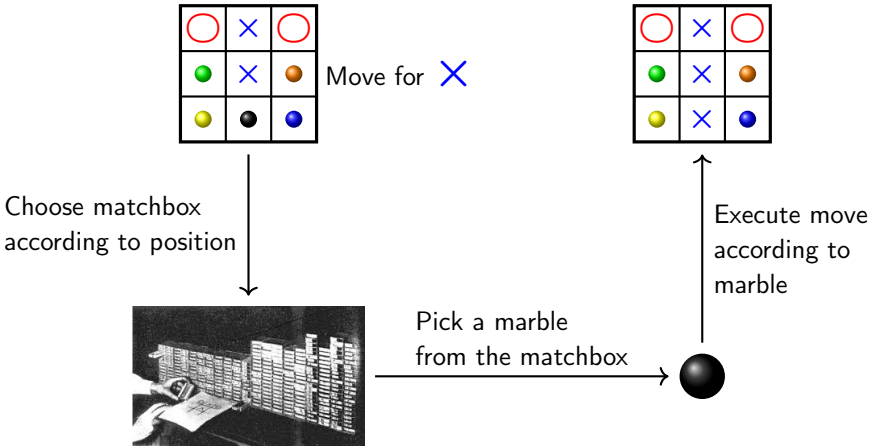
  Matchbox
  Educable
  Noughts
  And
  Crosses
  Engine



MENACE: Setup

- **'Hardware':**
  - 287 matchboxes (1 per playing position)
  - Marbles in nine different colors (one color per field)
  - Initially, each matchbox contains the same amount of marbles of each color
- **Algorithm:**
  - Choose matchbox corresponding to current playing position
  - Pick a marble randomly from the matchbox
  - Put marker ($\times$ or $\bigcirc$) on the field that corresponds to the color
- **How can good moves be reinforced positively?**

MENACE: Visualized



Reinforcement Learning in MENACE

- **Initialization:** All moves are equally likely $\Rightarrow$ Each matchbox contains the same amount of marbles of each color
- **Learning algorithm**
  - Game **lost**: Marble is removed (negative reinforcement)
  - Game **won**: Marble of corresponding color is added (positive reinforcement)
  - **Remis**: Marbles are put back into the matchbox (no change)

$\Uparrow$ Probability of successful moves is **increased**
$\Downarrow$ Probability of bad moves is **decreased**

Reinforcement Learning: Formalization

- Definitions
  - $s \in \mathcal{S}$: state space (discrete or continuous)
  - $a \in \mathcal{A}$: action space (discrete or continuous)
  - $s_0 \in \mathcal{S}_0 \subseteq \mathcal{S}$: initial states
  - $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$: state transition function (deterministic or stochastic)
  - $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$: reward function
- **Markov property / Markov Decision Process (MDP)**
  - Rewards and state transitions depend on previous state only...
  - ...and not how you got into this state (earlier states and actions don't matter)

Reinforcement Learning: Formalization (Ctd.)

- The agent repeatedly chooses an action according to a **policy** π

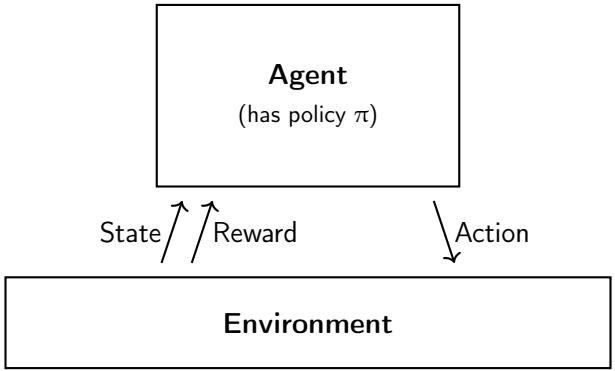
$$\pi(s) = a \tag{1}$$

- This leads the agent into a new state s'

$$s' = \delta(s, \pi(s)) \tag{2}$$

- The agent receives feedback (reinforcement) in some states
  - Not necessarily in all states
  - E. g. in a game there is usually no reward until the end (game is won or lost)

Reinforcement Learning: Formalization (Ctd.)



$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} \cdots$$

MENACE Formalization

States	Matchboxes (discrete)
Actions	Possible moves (discrete)
Policy	Prefer actions with higher number of marbles (stochastic)
Reward	Game won / game lost
Transition function	Choose next matchbox according to rules (deterministic)

Task: Find a policy π that maximizes the sum of future rewards ($\widehat{=}\ \pi^*$).

Learning Task

- **Goal:** Maximize the **cumulative reward** for the **trajectory** τ which a policy π generates
  - **Deterministic:** τ is always the same
  - **Stochastic:** τ can differ (random elements included)
- Cumulative reward:

$$R(\pi) = R(\tau^{\pi}) = \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \tag{3}$$

- **Discount factor** γ: Immediate rewards are weighted higher, rewards further in the future are discounted (exponentially)

Learning Task (Ctd.)

- Rewards t time steps in the future are discounted exponentially by a factor of γ^t
- Value of γ:
  - $0 \leqslant \gamma \leqslant 1$
  - $\gamma = 0$: Only immediate rewards are taken into account
  - $\gamma = 1$: Future rewards are given same emphasis as immediate rewards
- Usually, γ is set to a value close to 1 (0.99, 0.98, 0.95, ...)
- Immediate rewards are usually preferred

How to compute $R(\tau^\pi)$?

$$
\begin{aligned}
R(\tau^\pi) &= \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \\
&= r(s_0, a_0) + \gamma r(s_1, a_1) + \gamma^2 r(s_2, a_2) + \dots \\
&= r(s_0, \pi(s_0)) + \sum_{t=1}^{\infty} \gamma^t r(\delta(s_{t-1}, \pi(s_{t-1})), \pi(s_t)) \\
&= V^\pi(s_0)
\end{aligned}
$$

- V is called the 'value' of the first state
- **Value function**: Reward when starting in state s_0 and following policy π

Optimal Policies and Value Functions

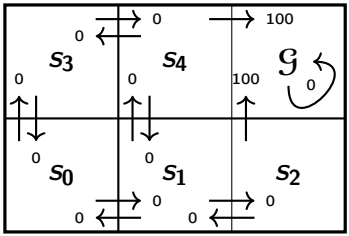
- The optimal policy is denoted by π^*
- The optimal policy has the highest expected value for all states:

$$\begin{aligned}\pi^*(s) &= \arg\max_{\pi} V^{\pi}(s) \quad \forall s \\ &= \arg\max_{a \in \mathcal{A}} r(s,a) + \gamma V^{\pi^*}(\delta(s,a)) \quad \forall s\end{aligned}$$

- Always select the action that maximizes the value function for the next step, when following π^* afterwards
- **Problem: Recursive definition of** π^* requires perfect knowledge

An illustrative Example (given perfect Knowledge)

- Each square represents a state
- Each arrow represents a possible action
- Small numbers refer to the reward $r(s, a)$ obtained by taking the action
- In this case: Reward is always set to 0, except when entering the goal state $\mathcal{G}$
- $\mathcal{G}$ is an absorbing state (impossible to get out again)
- $\gamma = 0.9$



An illustrative Example [given perfect Knowledge] (Ctd.)

In this case we can derive the optimal policy π^* directly, e. g. $s_0 \rightarrow s_3 \rightarrow s_4 \rightarrow \mathcal{G}$

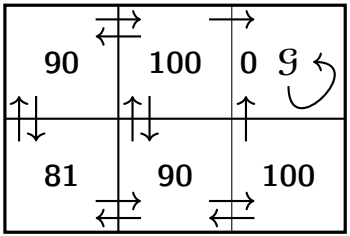
$$V^{\pi^*}(s_0) = 0 + \gamma 0 + \gamma^2 100 + \gamma^3 0 + \ldots = \mathbf{81}$$

$$V^{\pi^*}(s_1) = 0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \ldots = \mathbf{90}$$

$$V^{\pi^*}(s_2) = 100 + \gamma 0 + \gamma^2 0 + \gamma^3 0 + \ldots = \mathbf{100}$$

$$V^{\pi^*}(s_3) = 0 + \gamma 100 + \gamma^2 0 + \gamma^3 0 + \ldots = \mathbf{90}$$

$$V^{\pi^*}(s_4) = 100 + \gamma 0 + \gamma^2 0 + \gamma^3 0 + \ldots = \mathbf{100}$$



Section:

Basic Algorithms

How to learn the optimal Policy?

- W/o perfect knowledge it's difficult to learn the optimal policy directly, since training data of the form $\langle s, a \rangle$ is not available ($\neq$ **supervised learning**)
- The only information at the agent's disposal is the sequence of observed immediate rewards:

$$r_0 \longrightarrow r_1 \longrightarrow r_2 \longrightarrow r_3 \longrightarrow r_4 \longrightarrow \ldots$$

- What evaluation function should be learned?
  - One obvious choice is V^{π^*}
  - The agent should prefer state s_1 over s_2, if $V^{\pi^*}(s_1) > V^{\pi^*}(s_2)$

How to learn the optimal Policy? (Ctd.)

- The optimal action in state s is action a that maximizes the sum of the immediate reward $r(s, a)$, plus the value of V^{π^*} of the immediate successor, discounted by γ:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} r(s, a) + \gamma V^{\pi^*}(\delta(s, a)) \tag{4}$$

- **Problem:**
  - In order to calculate V^{π^*} the agent needs to know $\delta(s, a)$ and $r(s, a)$
  - Learning $V^{\pi *}$ is of no use, since the equation cannot be evaluated
- $\overset{Remedy}{\Longrightarrow}$ Value iteration, policy iteration, Q-Learning, SARSA, ...

Value Iteration Algorithm

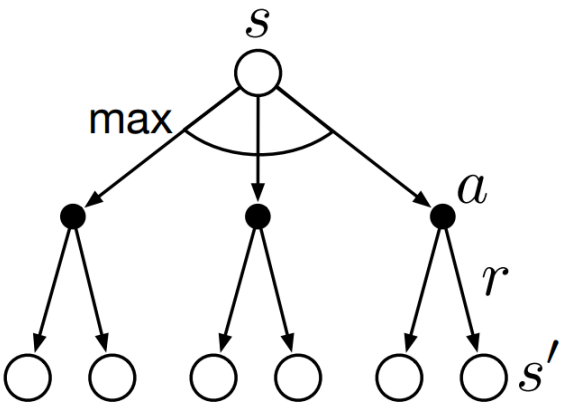
Algorithm 1: Value Iteration

Input: environment $\mathcal{E} = (\mathcal{S}, \mathcal{A})$

Output: optimal policy π

Initialize $V(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$
while $\Delta > \varepsilon$ **do**
    $\Delta \longleftarrow 0$
    foreach $s \in \mathcal{S}$ **do**
        $v \longleftarrow V(s)$
        $V(s) \longleftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        $\Delta \longleftarrow \max(\Delta, |v - V(s)|)$
return π such that $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

Backup Diagram for Value Iteration



Policy Iteration

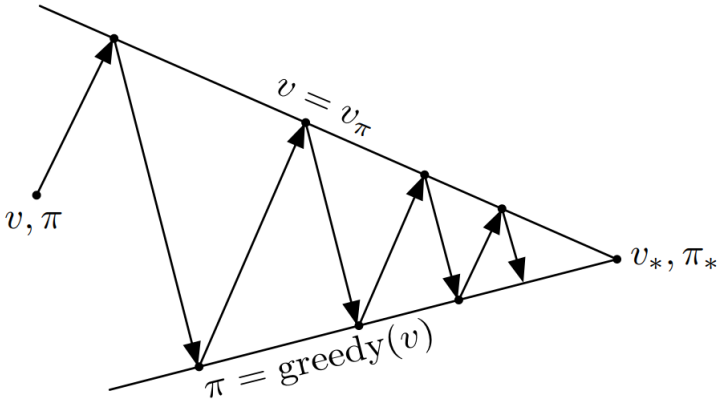
- Initialize a policy randomly, find its value function and iterate
  - **Policy evaluation**: Find the value function of the current policy
  - **Policy improvement**: Find a better policy based on the current one
- Select the action that maximizes the value function of the current policy:

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} r(s, a) + \gamma V^{\pi}(\delta(s, a)) \tag{5}$$

- This leads to a sequence of improving policies:

$$\pi^0(s) \longrightarrow V^{\pi^0}(s) \longrightarrow \pi^1(s) \longrightarrow V^{\pi^1}(s) \longrightarrow \cdots \longrightarrow \pi^*(s)$$

Policy Iteration (Ctd.)



Policy Improvement Theorem

Policy improvement theorem:

If it is true that selecting the first action in each state according to a policy π' and continuing with policy π is better than always following π, then π' is a better policy than π.

Policy Iteration Algorithm

Algorithm 2: Policy Iteration (deterministic case)

Input: environment $\mathcal{E} = (\mathcal{S}, \mathcal{A})$

Output: optimal policy π

Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}$ arbitrarily for all $s \in \mathcal{S}$
policy_stable $\longleftarrow$ *false*
while *notpolicy_stable* **do**
    $V_{new} \longleftarrow PolicyEvaluation(\pi, V, \varepsilon = 0.001)$
    π, *policy_stable* $\longleftarrow PolicyImprovement(V_{new})$
    $V \longleftarrow V_{new}$
return π

Policy Evaluation and Policy Improvement

Algorithm 3: Policy Evaluation

Input: current policy π, current value function V, threshold ε

Output: updated value function V

while $\Delta > \varepsilon$ **do**
    $\Delta \longleftarrow 0$
    foreach $s \in \mathcal{S}$ **do**
        $v \longleftarrow V(s)$
        $V(s) \longleftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V^{\pi}(s')]$
        $\Delta \longleftarrow \max(\Delta, |v - V(s)|)$
return V

Policy Evaluation and Policy Improvement

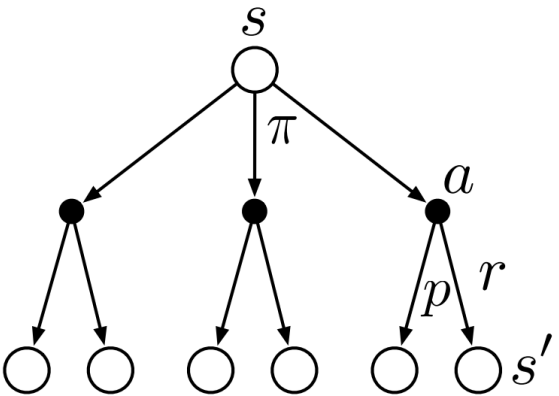
Algorithm 4: Policy Improvement

Input: updated value function V

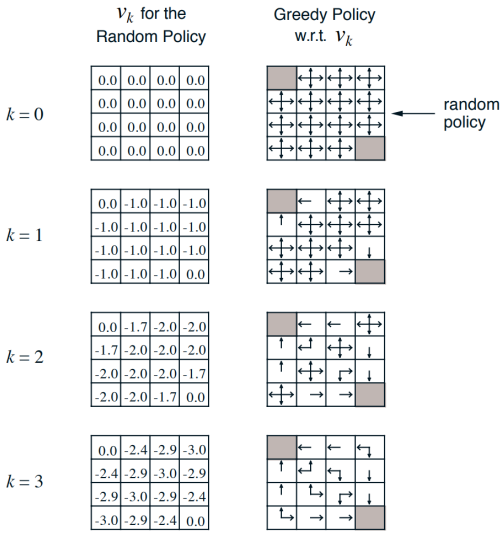
Output: updated policy π, boolean flag *policy_stable*

```
1 policy_stable ⟵ true
2 foreach s ∈ 𝒮 do
3     old_action ⟵ π(s)
4     π(s) ⟵ arg max_a Σ_{s',r} p(s', r|s, a)[r + γ V^π(s')]
5     if old_action ≠ π(s) then
6         policy_stable ⟵ false
7 return π, policy_stable
```

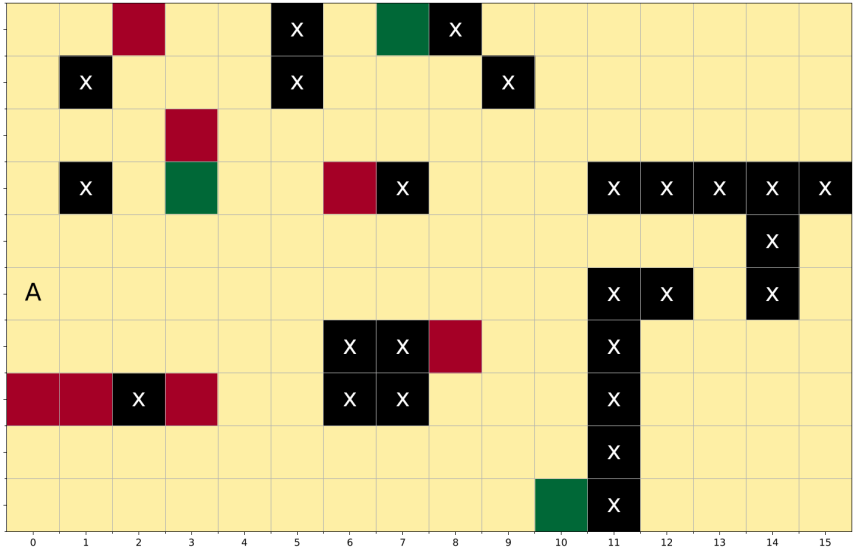
Backup Diagram for Policy Iteration



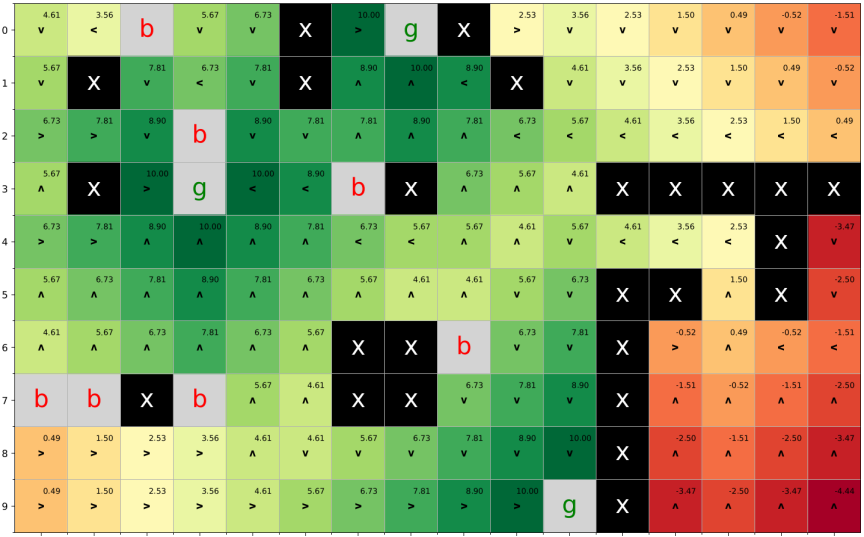
Policy Iteration Example: Grid World I



Policy Iteration Example: Grid World II



Policy Iteration Example: Grid World II (solved)



Q-Learning

- $Q(s, a) \equiv$ maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action:

$$Q(s, a) = r(s, a) + \gamma V^{\pi^*}(\delta(s, a)) \qquad \textit{looks familiar?} \tag{6}$$

- Formula for π^* can be rewritten in terms of $Q(s, a)$:

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q(s, a) \tag{7}$$

- **This rewrite is important:** Learning the Q-function the agent can select optimal actions without any knowledge of r and δ

Q-Learning (Ctd.)

- By always choosing the action with the maximum Q-value...

$$\arg\max_{a \in \mathcal{A}} Q(s, a)$$

- ...we maximize the expected cumulative reward
- What may seem surprising: Local optimal choices lead to a global optimal solution

Learning the Q-function corresponds to learning the optimal policy π^*

Algorithm for Q-Learning

How can Q be learned?

- We learn the Q-function by iterative approximation
- Note the close relationship of Q and V^{π^*}:

$$V^{\pi^*}(s) = \max_{a' \in \mathcal{A}} Q(s, a') \tag{8}$$

- Therefore, $Q(s, a)$ can be expressed in terms of itself: **Bellman equation**

$$Q(s, a) = r(s, a) + \gamma \max_{a' \in \mathcal{A}} Q(\delta(s, a), a') \tag{9}$$

Algorithm for Q-Learning (Ctd.)

- $\widehat{Q}$ denotes the agent's current estimate of the true Q-value (which is unknown)
- The Q-function is represented by a table (one entry per state-action pair)
- Initially, the table is filled with random numbers or zeroes
- **The table may become large, if there are many states and actions**

State-action pair	$\widehat{Q}$-value
$\langle s_0, a_0 \rangle$	$\widehat{Q}(s_0, a_0)$
$\langle s_0, a_1 \rangle$	$\widehat{Q}(s_0, a_1)$
$\langle s_1, a_0 \rangle$	$\widehat{Q}(s_1, a_0)$
$\langle s_1, a_1 \rangle$	$\widehat{Q}(s_1, a_1)$
...	...

Algorithm for Q-Learning (Ctd.)

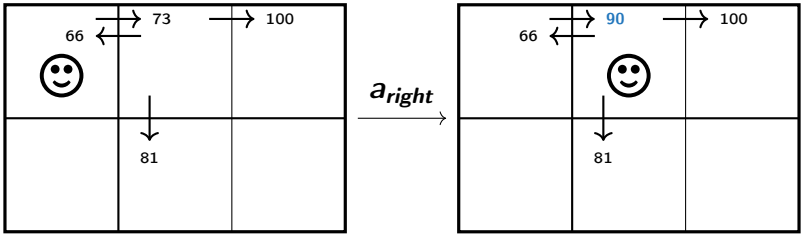
- The agent repeatedly observes the current state s, chooses some action a, executes it and observes r as well as the new state s'
- Update the table entry for $\widehat{Q}(s, a)$ according to the Bellman equation
- The agent doesn't have to know $\delta(s, a)$ and $r(s, a)$
- Instead, it executes some actions and observes what happens
- In the limit, $\widehat{Q}$ will converge towards the actual Q-function **iff**...
  - ...the system can be modeled as a deterministic MDP...
  - ...**and** the reward function is bounded...
  - ...**and** each state-action pair is visited infinitely often

Algorithm for Q-Learning (Ctd.)

Algorithm 5: Learning the Q-function

foreach $\langle s, a \rangle$ **do** // initialize table entries with zeroes
    $\widehat{Q}(s, a) \longleftarrow 0$ // Alternative: Initialize with random numbers
while *true* **do** // endless loop
    Observe current state s
    Select an action a and execute it
    Receive the immediate reward r
    Observe the new state s'
    Update the table entry for $\widehat{Q}(s, a)$: $\widehat{Q}(s, a) \longleftarrow r + \gamma \max_{a' \in \mathcal{A}} \widehat{Q}(s', a')$
    $s \longleftarrow s'$

Q-Learning: An illustrative Example



$$\widehat{Q}(s_3, a_{right}) \longleftarrow r + \gamma \max_{a' \in \mathcal{A}} \widehat{Q}(s_4, a')$$

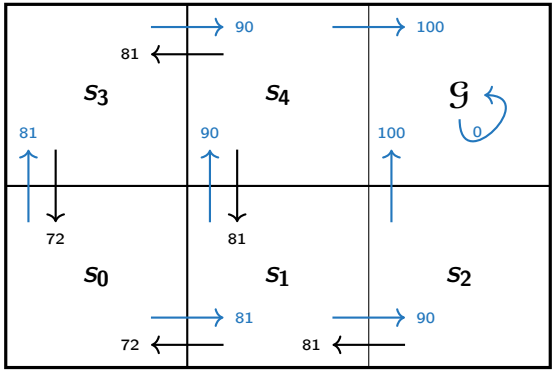
$$\longleftarrow 0 + 0.9 \max_{a' \in \mathcal{A}} \{66, 81, 100\}$$

$$\longleftarrow 90$$

Q-Learning: An illustrative Example (Ctd.)

- Each time the agent makes a move, the Q-value estimates are **propagated backwards** from the new state s' to the old state s
- The immediate reward is used to **augment** the propagated values of $\widehat{Q}$
- Since there is an absorbing goal state $\mathcal{G}$, there will be a series of episodes
- **One episode:**
  1. Initialize the agent at a random state
  2. The episode ends as soon as the goal state $\mathcal{G}$ is reached
  3. Repeat until convergence (`goto 1`)

Q-Learning: An illustrative Example (Ctd.)

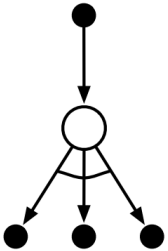


SARSA: State-Action-Reward-State-Action

- SARSA is very similar to Q-learning
- The abbreviation stands for 'State-Action-Reward-State-Action'
- It refrains from calculating the 'max' in the formula
- Instead it uses the current policy (**on-policy update**)

$$\widehat{Q}(s,a) \longleftarrow r(s,a) + \gamma \widehat{Q}(\underbrace{\delta(s,a)}_{=s'}, \pi(s')) \tag{10}$$

Backup Diagrams for Q-Learning and SARSA



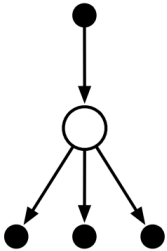


Figure: Backup diagrams of Q-learning (left) and SARSA (right)

Section:

Miscellaneous

Exploitation vs. Exploration

- By always picking the best action the agent runs the risk of overcommitting to actions that were found during early training
- **It fails to explore other actions that might be even better**
- **Exploitation:**
  - Use the action the agent assumes to be the best one
  - Approximate the optimal policy
- **Exploration:**
  - 'Optimal action' may be wrong due to approximation errors
  - Try a sub-optimal one

There is an exploitation $\Leftrightarrow$ exploration trade-off!

Exploitation vs. Exploration (Ctd.)

- ε-**greedy** (small $\varepsilon \Rightarrow$ exploitation)

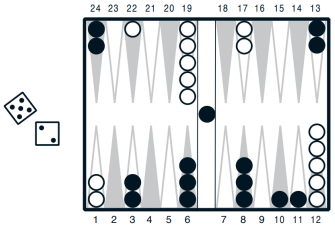
$$p(a_i|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}|} & \textbf{if } a_i = \arg\max_{a \in \mathcal{A}} \widehat{Q}(s,a) \\ \frac{\varepsilon}{|\mathcal{A}|} & \textbf{otherwise} \end{cases} \tag{11}$$

- **Softmax** (actions with high $\widehat{Q}$-values get a higher probability)

$$p(a_i|s) = \frac{k^{\widehat{Q}(s,a_i)}}{\sum_j k^{\widehat{Q}(s,a_j)}} \qquad k > 0 \quad (\text{large } k \Rightarrow \text{exploitation}) \tag{12}$$

Non-Deterministic Rewards and Actions

- So far we have only considered the deterministic case
- **What if the environment is non-deterministic?**
  - E. g. in Backgammon, each move involves a roll of the dice $\Rightarrow \delta(s, a)$ varies
  - Also, the rewards may change $\Rightarrow r(s, a)$ varies



Non-Deterministic Rewards and Actions (Ctd.)

- $\delta(s, a)$ and $r(s, a)$ can be viewed as first producing a probability distribution over the outcomes based on s and a, and then drawing an outcome at random according to this distribution
- If these probability distributions depend solely on s and a (not on earlier states or actions), then the system is called **non-deterministic Markov decision process**

The Q-learning algorithm has to be modified in order to be able to handle the non-deterministic case.

Non-Deterministic Rewards and Actions (Ctd.)

- Restate the agent's objective $\Rightarrow$ Redefine the value V^π of a policy π to be the expected value (solve e. g. by using **Monte Carlo sampling**, choose k):

$$V^\pi(s_0) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, \pi(s_t))\right] = r(s_0, \pi(s_0)) + \underbrace{\frac{1}{k}\sum_{i=0}^{k}}_{k \text{ samples}} \sum_{t=1}^{\infty} \gamma^t r(s_t^i, \pi(s_t^i))$$

- The definition of Q has to be updated as well:

$$\begin{aligned} Q(s,a) &= \mathbb{E}[r(s,a) + \gamma V^{\pi^*}(\delta(s,a))] = \mathbb{E}[r(s,a)] + \gamma \mathbb{E}[V^{\pi^*}(\delta(s,a))] \\ &= \mathbb{E}[r(s,a)] + \gamma \sum_{s'} p(s'|s,a) V^{\pi^*}(s') \end{aligned}$$

Non-Deterministic Rewards and Actions (Ctd.)

- We can again re-express Q recursively:

$$Q(s,a) = \mathbb{E}[r(s,a)] + \gamma \sum_{s'} p(s'|s,a) \max_{a'} Q(s',a') \tag{13}$$

- We need a new training rule for the algorithm (the old one fails to converge due to changing rewards...)

$$\widehat{Q}_n(s,a) \longleftarrow (1-\alpha_n)\widehat{Q}_{n-1}(s,a) + \alpha_n[r + \gamma \max_{a'} \widehat{Q}_{n-1}(s',a')] \tag{14}$$

- Where $\alpha_n = \frac{1}{1+visits_n(s,a)}$ ($visits_n(s,a)$ is the number of times the state-action pair has been visited)

Non-Deterministic Rewards and Actions (Ctd.)

- Updates of $\widehat{Q}$ are made more gradually than in the deterministic case (average)
- For $\alpha_n = 1$ we get the old training rule
- α_n decreases over time $\Rightarrow$ Updates become smaller in later iterations
- By using α the algorithm ultimately converges
- Q-learning often requires many thousands of training iterations to converge

E. g. TD-Gammon trained for **1.5 million (!)** Backgammon games.

Temporal Difference Learning

- Q-learning works by iteratively reducing the discrepancy between Q-value estimates at different time steps (based on a one-step lookahead)
- Q-learning is a special case of **temporal difference learning**

$$\widehat{Q}_n(s,a) \longleftarrow (1-\alpha_n)\widehat{Q}_{n-1}(s,a) + \alpha_n[r(s,a) + \gamma \max_{a'} \widehat{Q}_{n-1}(s',a')]$$

$\Updownarrow$ **rewrite the equation**

$$\widehat{Q}_n(s,a) \longleftarrow \widehat{Q}_{n-1}(s,a) + \alpha_n[\overbrace{(r(s,a) + \gamma \max_{a'} \widehat{Q}_{n-1}(s',a'))}^{\text{after}} - \overbrace{\widehat{Q}_{n-1}(s,a)}^{\text{before}}]$$

$$\text{Temporal difference}$$

Temporal Difference Learning (Ctd.)

- Q-learning performs a one-step lookahead (can be generalized to n steps)
- $Q^{(1)}(s_t, a_t)$ denotes the training value calculated by this one-step lookahead:

$$Q^{(1)}(s_t, a_t) = r_t + \gamma \max_a \widehat{Q}(s_{t+1}, a) \tag{15}$$

- For two steps:

$$Q^{(2)}(s_t, a_t) = r_t + \gamma r_{t+1} + \gamma^2 \max_a \widehat{Q}(s_{t+2}, a) \tag{16}$$

- For n steps:

$$Q^{(n)}(s_t, a_t) = r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \widehat{Q}(s_{t+n}, a) \tag{17}$$

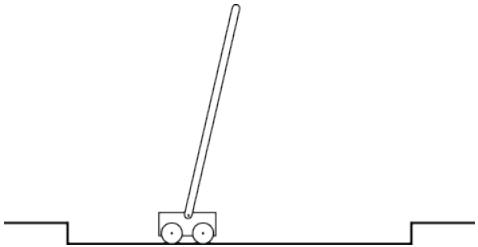
Temporal Difference Learning: TD(λ) [*Sutton*, 1988]

- TD(λ) is a method for blending these alternative training estimates
- Introduce a constant λ ($0 \leqslant \lambda \leqslant 1$) to combine the estimates from various lookaheads (recursive definition):

$$Q^{\lambda}(s_t, a_t) = r_t + \gamma \left[(1 - \lambda) \max_a \widehat{Q}(s_t, a_t) + \lambda Q^{\lambda}(s_{t+1}, a_{t+1}) \right]$$

- If $\lambda = 0$ we get the original training estimate $Q^{(1)}$ (only one-step lookahead)
- If $\lambda = 1$ only the observed rewards r_{t+i} are taken into account
- Increasing values of λ put more emphasis on discrepancies based on more distant lookaheads

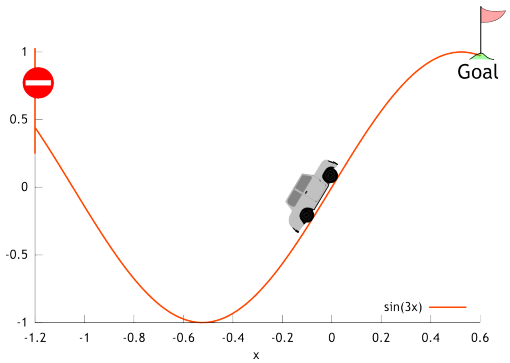
Demo: Cartpole Task



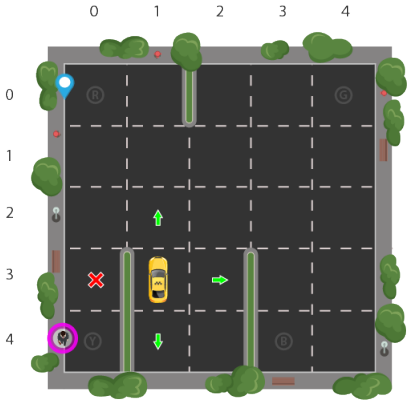
- Move the cart **without** the pole falling over
- Action space $\mathcal{A}$:
  - Move *LEFT* (0)
  - Move *RIGHT* (1)
- 4D state space $\mathcal{S}$:
  - Cart position, velocity
  - Angle of pole to cart...
  - ...and its derivative

Demo: Mountain Car Task

- The engine of the car is not strong enough
- You have to go back and forth to acquire momentum
- The car must not hit the wall (left side)
- The car has to reach the top of the hill



Demo: Taxi Task



- Pick up passengers and get them to the drop-off location
- Filled square represents the taxi
- '|' represents a wall; you should not go there ;-)
- Blue letter: **Pick-up location**
- Purple letter: **Drop-off location**
- Taxi turns **green** if a passenger is aboard

Section:

Advanced Algorithms

Policy Gradient Methods

- Parameterize policy π with parameters $\boldsymbol{\theta}$: $\pi_{\boldsymbol{\theta}}$
- Total reward for a trajectory τ: $r(\tau)$
- **Goal:** Maximize the expected reward following a parameterized policy:

$$\mathcal{J}(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[r(\tau)] \tag{18}$$

- We can use gradient ascent to find the optimal parameters $\boldsymbol{\theta}^*$
- **Problem:** How do we find the gradient of the objective which contains the expectation?

Policy Gradient Theorem

$$
\begin{aligned}
\nabla \mathbb{E}_{\pi_{\boldsymbol{\theta}}} &= \nabla \int \pi(\tau) r(\tau) \, \mathrm{d}\tau \\
&= \int \nabla \pi(\tau) r(\tau) \, \mathrm{d}\tau && \text{pull } \nabla \text{ inside} \\
&= \int \pi(\tau) \nabla \log \pi(\tau) r(\tau) \, \mathrm{d}\tau && \text{use log trick: } \nabla_x \log(f(x)) = \frac{1}{f(x)} \nabla_x f(x) \\
&= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[r(\tau) \nabla \log \pi(\tau)]
\end{aligned}
$$

$$
\pi_{\boldsymbol{\theta}}(\tau) = p(s_0) \cdot \prod_{t=1}^{T} \pi_{\boldsymbol{\theta}}(a_t|s_t) \cdot p(s_{t+1}, r_{t+1}|s_t, a_t)
$$

Policy Gradient Theorem

Policy gradient theorem:

The derivative of the expected reward is the expectation of the product of the reward and gradient of the log of the policy π_θ.

Policy Gradient Theorem (Ctd.)

Taking the logarithm of $\pi_{\boldsymbol{\theta}}$, we get:

$$\log \pi_{\boldsymbol{\theta}}(\tau) = \log p(s_0) + \sum_{t=1}^{T} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) + \sum_{t=1}^{T} \log p(s_{t+1}, r_{t+1}|s_t, a_t)$$

$$\nabla \log \pi_{\boldsymbol{\theta}}(\tau) = \sum_{t=1}^{T} \nabla \log \pi_{\boldsymbol{\theta}}(a_t|s_t)$$

Putting it all together:

$$\implies \nabla \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[r(\tau)] = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[r(\tau) \cdot \left(\sum_{t=1}^{T} \nabla \log \pi_{\boldsymbol{\theta}}(a_t|s_t) \right) \right]$$

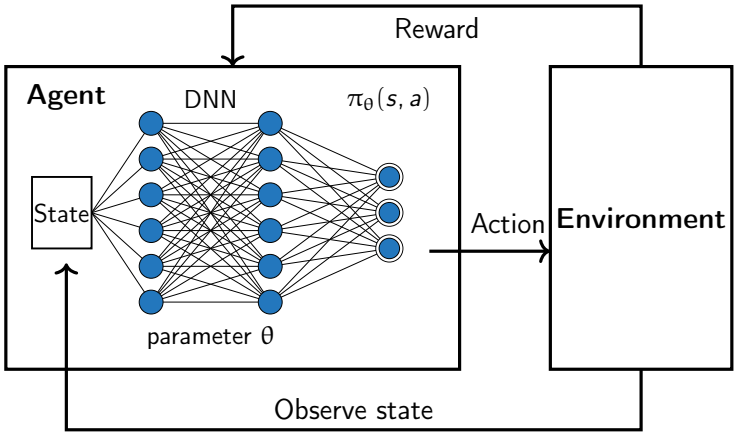
Policy Gradient Theorem (Ctd.)

- This result is appealing, since no knowledge about the initial state distribution nor the environment dynamics is necessary
- Therefore, this is an instance of a **model-free** algorithm
- There is still an expectation term in the equation, but we can sample a large number of trajectories and average
  $\Rightarrow$ **Markov Chain Monte Carlo (MCMC)**
- **Problem: Reward term $r(\tau)$ adds a lot of variance to the sampling process** $\Rightarrow$ We can introduce a baseline
- Algorithm **REINFORCE**

Deep Reinforcement Learning

- As already mentioned: The Q-learning table can become really large if the state space and the action space are really large
- **It is not viable to maintain a table of all entries**
- Maybe it does not even fit into memory...
- An alternative is needed!
- **Idea**:
  - Use deep learning methods to replace the table
  - E. g. a deep MLP could predict the action to take given the observed state, the reward, ...

Deep Reinforcement Learning (Ctd.)



Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

Section:

Wrap-Up

Final Example

An agent can move in a simple deterministic world which looks as following:

a	**b**	**c**
d	**e**	**f**

The agent can move up (u), down (d), left (l) or right (r), given the target field exists. The state 'f' is an absorbing goal state. As soon as the agent reaches 'f', it receives a reward of one point, otherwise 0 points. The discount factor γ be set to 0.8.

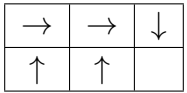
Final Example – Task a)

Specify the reward function $r(s, a)$ for all state-action pairs.

$r(a, r) = 0$	$r(b, r) = 0$	$r(c, d) = 1$	$r(d, u) = 0$	r(e, u) $= 0$
$r(a, d) = 0$	$r(b, d) = 0$	$r(c, l) = 0$	$r(d, r) = 0$	r(e, r) $= 1$
	$r(b, l) = 0$			r(e, l) $= 0$

Final Example – Task b)

Calculate the value function $V^\pi(s)$ for all states s given policy π:



$$V^\pi(d) = \gamma^0 r(d,u) + \gamma^1 r(a,r) + \gamma^2 r(b,r) + \gamma^3 r(c,d)$$
$$= 1 \cdot 0 + 0.8 \cdot 0 + 0.8^2 \cdot 0 + 0.8^3 \cdot 1 = \mathbf{0.512}$$

$V^\pi(a) = 0.640$	$V^\pi(b) = 0.800$	$V^\pi(c) = 1.000$
$V^\pi(d) = 0.512$	$V^\pi(e) = 0.640$	

Final Example – Task c)

How would **policy improvement** change the policy π from task b) for state **e**?

Policy improvement: $\pi(s) = \arg\max_{a \in \mathcal{A}} r(s,a) + \gamma V^{\pi}(\delta(s,a))$

up: 0.64 (cf. above)
left: $r(e,l) + \gamma V^{\pi}(d) = 0 + 0.8 \cdot 0.512 = 0.4096$
right: $r(e,r) = 1 \Longleftarrow$ **We would take this one!**

$\Rightarrow \pi(e) = \textit{right}$

Final Example – Task d)

Think of an optimal way to reach the goal state for each state s. Calculate $V^{\pi^*}(s)$.

Optimal routes:

$a \longrightarrow b \longrightarrow c \longrightarrow f$

$a \longrightarrow d \longrightarrow e \longrightarrow f$

$V^{\pi^*}(a) = 0.640$	$V^{\pi^*}(b) = 0.800$	$V^{\pi^*}(c) = 1.000$
$V^{\pi^*}(d) = 0.800$	$V^{\pi^*}(e) = 1.000$	

Final Example – Task e)

Calculate the optimal Q-values for all possible state-action pairs.

Hint: Use the results from task d): $Q(s, a) = r(s, a) + \gamma V^{\pi^*}(s')$

For state **a** we get:
$Q(a, r) = r(a, r) + \gamma V^{\pi^*}(b) = 0 + 0.8 \cdot 0.8 = 0.64 = Q(a, u)$

$Q(a, r) = 0.640$	$Q(b, r) = 0.800$	$Q(c, u) = 1.000$	$Q(d, o) = 0.512$	Q(e, o) = 0.640
$Q(a, u) = 0.640$	$Q(b, u) = 0.800$	$Q(c, l) = 0.640$	$Q(d, r) = 0.800$	Q(e, r) = 1.000
	$Q(b, l) = 0.512$			Q(e, l) = 0.640

Summary

- What is reinforcement learning?
  - Learn optimal decisions based on feedback
  - The agent only knows how good the action was, but not the optimal solution
  - **Credit assignment problem**:
    Which move(s) is / are responsible for success / failure?
  - Example: MENACE
- Important terms:
  - State s, action a
  - Policy π *(optimal policy* π^**)*
  - Reward r *(immediate reward, discounted reward)*

Summary (Ctd.)

- Algorithms
  - Policy iteration
  - Q-learning
  - SARSA
- Exploitation vs. Exploration
  - **Exploitation**: Exploit what is already known to be good
  - **Exploration**: But don't miss to explore new states and actions
- Non-deterministic case ($\delta(s, a)$ and $r(s, a)$ vary)
- Temporal difference learning
- Deep reinforcement learning

Recommended Literature and further Reading I

[1] Machine Learning

T. Mitchell. McGraw-Hill Science. 1997.

→ Click here, cf. chapter 13

[2] Solving an MDP with Q-Learning from scratch

V. Valkov. 2017.

→ Click here

[3] Hands-On Machine Learning with Scikit-Learn & TensorFlow

A. Géron. O'Reilly. 2017.

→ Click here, cf. chapter 16

Recommended Literature and further Reading II

[4] Machine Learning; An algorithmic perspective

S. Marsland. CRC Press. Chapman & Hall. 2015.

→ Click here, cf. chapter 17

[5] Learning to predict by the methods of temporal differences

R. Sutton. Kluwer Academic Publishers. 1988.

→ Click here

[6] Machine Learning Lecture

A. Ng. Stanford University. 2008.

→ Click here (Lecture 16), Click here (Lecture 17),
Click here (Lecture 18), Click here (Lecture 19), Click here (Lecture 20)

Recommended Literature and further Reading III

[7] OpenAI Gym

OpenAI. 2018.

→ Click here

[8] Getting started with reinforcement Q-learning

P. Jaiswal. Towards Data Science. 2017.

→ Click here

[9] Reinforcement Learning: An Introduction

R. Sutton, A. Barto. MIT Press. 2014.

→ Click here

Meme of the Day

Thank you very much for the attention!

Topic: *** Applied Machine Learning Fundamentals *** Reinforcement Learning
Term: Winter term 2019/2020

Contact:
M. Sc. Daniel Wehner
SAP SE
daniel.wehner@sap.com

Do you have any questions?