

Reinforcement learning

Applied Machine Learning (EDAN95)

Lectures 13 and 14

2018-12-17 and 2018-12-19

Elin A. Topp

Material based on “Hands-on Machine Learning with SciKit-learn and TensorFlow” (course book, chapter 16),
and on lecture “Belöningsbaserad inläring / Reinforcement learning”
by Örjan Ekeberg, CSC/Nada, KTH, autumn term 2006 (in Swedish)

Outline

- Reinforcement learning
 - Problem definition
 - Learning situation
 - Role of the reward
 - Simplified assumptions
 - Central concepts and terms
 - Known environment
 - Bellman's equation
 - Approaches to solutions
 - Unknown environment
 - Temporal-Difference learning
 - Q-Learning
 - Sarsa-Learning
 - Improvements
 - The usefulness of making mistakes
 - Eligibility Trace

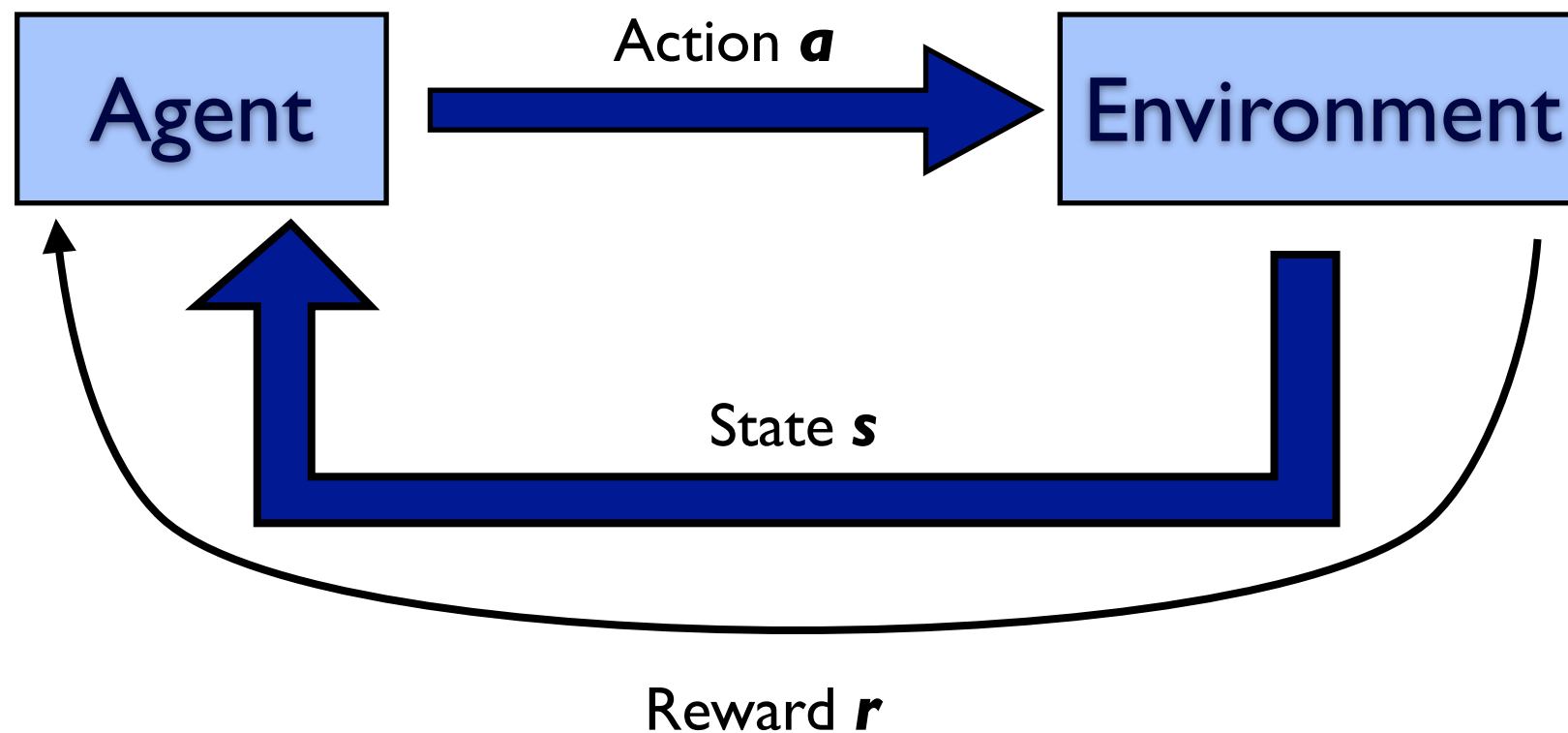
Learning situation: A model

An *agent* interacts with its *environment*

The agent performs *actions*

Actions have *influence* on the environment's *state*

The agent *observes* the environment's *state* and receives a *reward* from the environment



Outline

- Reinforcement learning
 - Problem definition
 - Learning situation
 - Roll of the reward
 - Simplified assumptions
 - Central concepts and terms
 - Known (observable) environment
 - Bellman's equation
 - Approaches to solutions
 - Outlook: unknown environments, Monte Carlo method and policy gradients
 - Unknown environment
 - Temporal-Difference learning
 - Q-Learning
 - Sarsa-Learning
 - Improvements
 - The usefulness of making mistakes
 - Eligibility Trace

Solving the equation

There are two ways of solving (this “optimal” version of) *Bellman’s equation*

$$U^\pi(s) = r(s, \pi(s)) + \gamma \cdot U^\pi(\delta(s, \pi(s)))$$

- Directly: $U^\pi(s) = r(s, \pi(s)) + \gamma \cdot \sum_{s'} P(s' | s, \pi(s)) U^\pi(s')$
- Iteratively (*Value / utility iteration*), stop when equilibrium is reached, i.e., “nothing happens”

$$U_{k+1}^\pi(s) \leftarrow r(s, \pi(s)) + \gamma \cdot U_k^\pi(\delta(s, \pi(s)))$$

Finding optimal policy and value function

How can we find an *optimal policy* π^* ?

That would be easy if we had the *optimal value / utility function* U^* :

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} (r(s, a) + \gamma \cdot U^*(\delta(s, a)))$$

Apply to the “optimal version” of Bellman’s equation

$$U^*(s) = \underset{a}{\operatorname{max}} (r(s, a) + \gamma \cdot U^*(\delta(s, a)))$$

Tricky to solve ... but possible:

Combine policy and value iteration by switching in each iteration step

Policy iteration

Policy iteration provides exactly this switch.

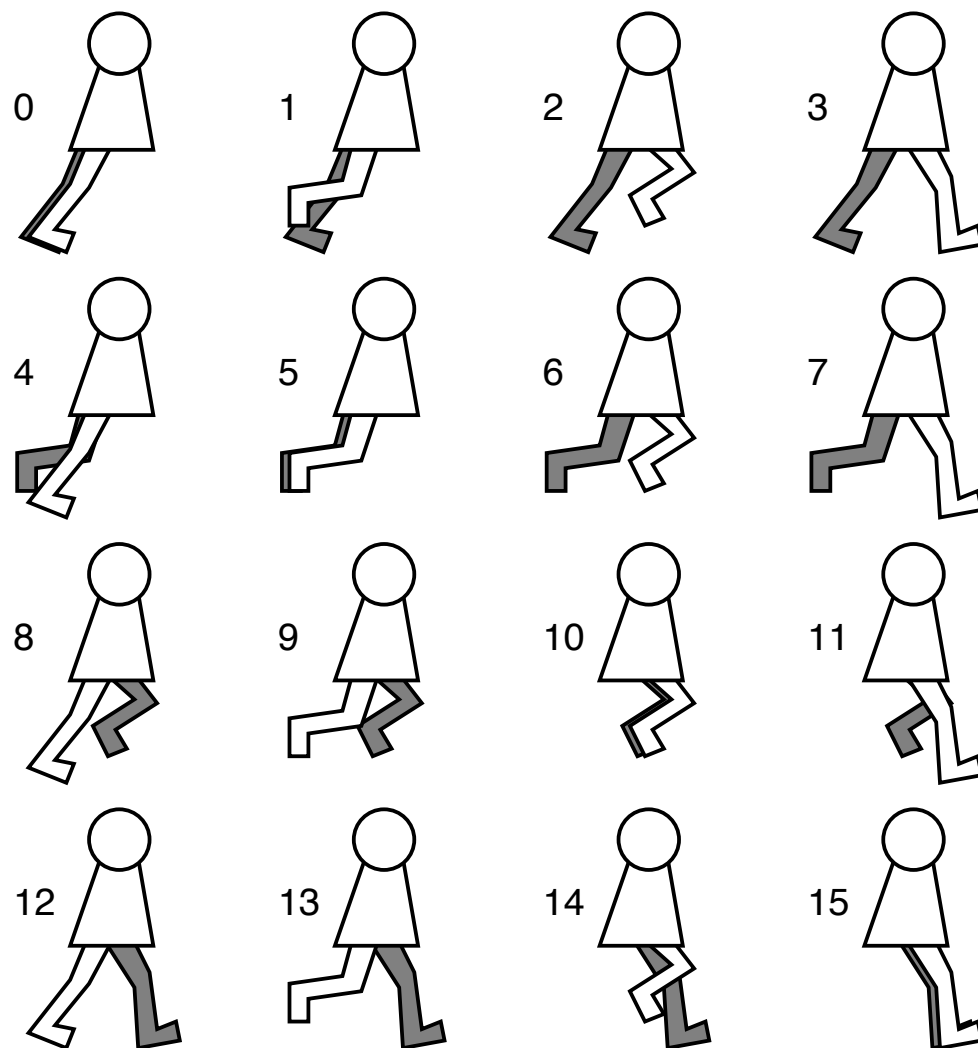
For each iteration step k :

$$\pi_k(s) = \underset{a}{\operatorname{argmax}} (r(s, a) + \gamma \cdot U_k(\delta(s, a)))$$

$$U_{k+1}(s) = r(s, \pi_k(s)) + \gamma \cdot U_k(\delta(s, \pi_k(s)))$$

Policy Iteration for Cartoon Walker

We cheat a bit, and use entirely known reward and transition functions...



Action	Effect
0	Move right (white) leg up / down
1	Move right (white) leg backward / forward
2	Move left (grey) leg up / down
3	Move left (grey) leg backward / forward

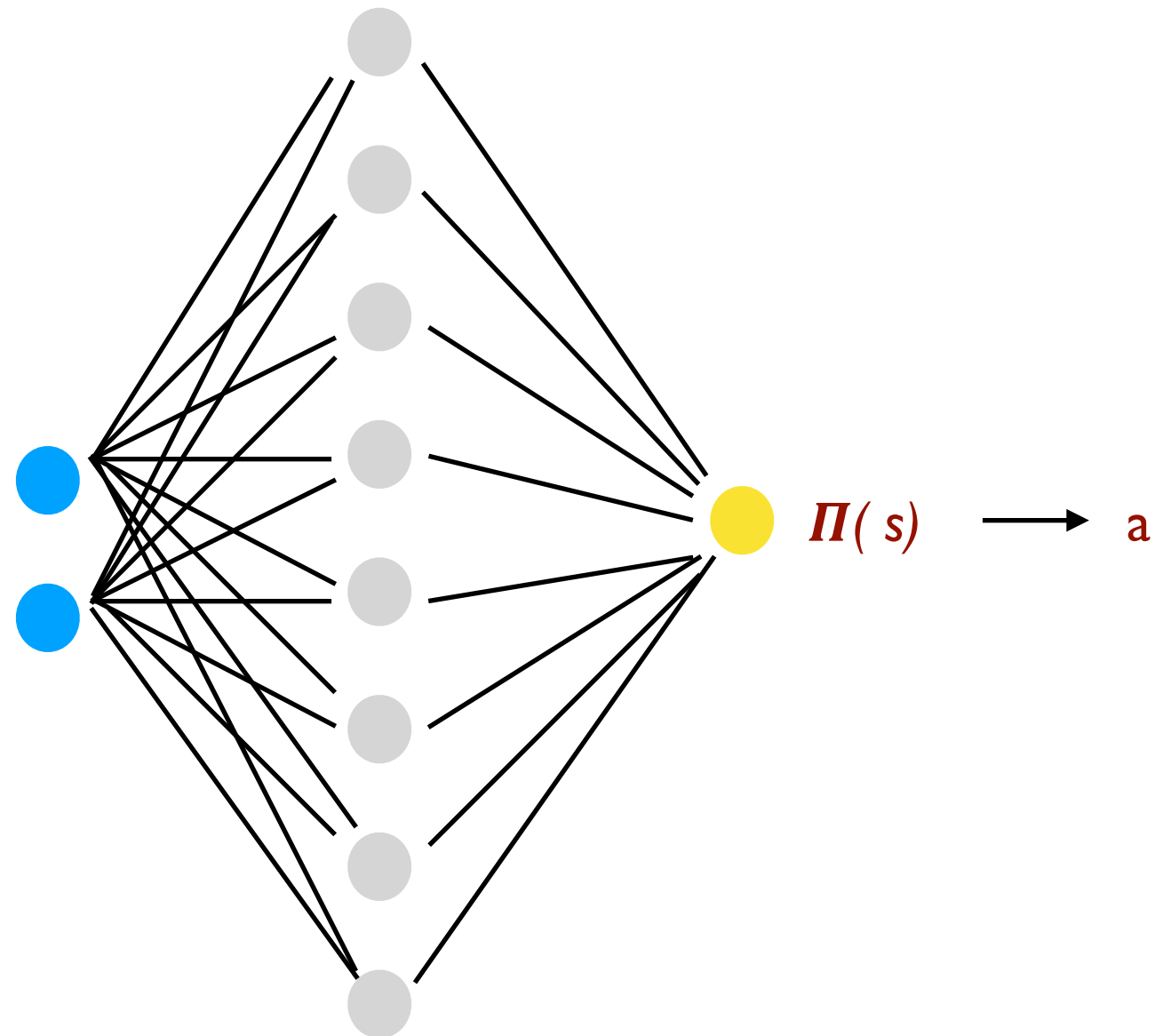
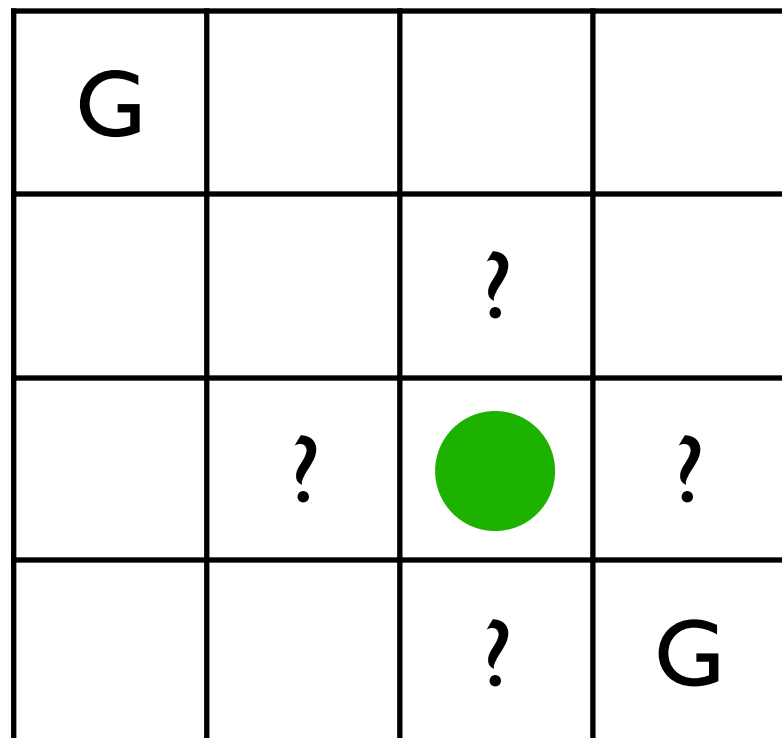
```
for s in range(len(policy)):
    policy[s] = argmax(
        lambda a: rew[s][a] + gamma * value[trans[s][a]],
        range(len(trans[s])))
```

```
for s in range(len(value)):
    a = policy[s]
    value[s] = rew[s][a] + gamma * value[trans[s][a]]
```


Policy gradients

What if...

... we take help of an ANN to learn a good policy?



Training the network

If we had a “label” saying after a forward run that DOWN is the optimal thing to do for this state...

... we would compute the loss as:

$$- \log P(y=\text{DOWN} \mid x)$$

... but we do not have this label, so we use the reward R we get from using our

policy (the sampled action) to compute the loss:

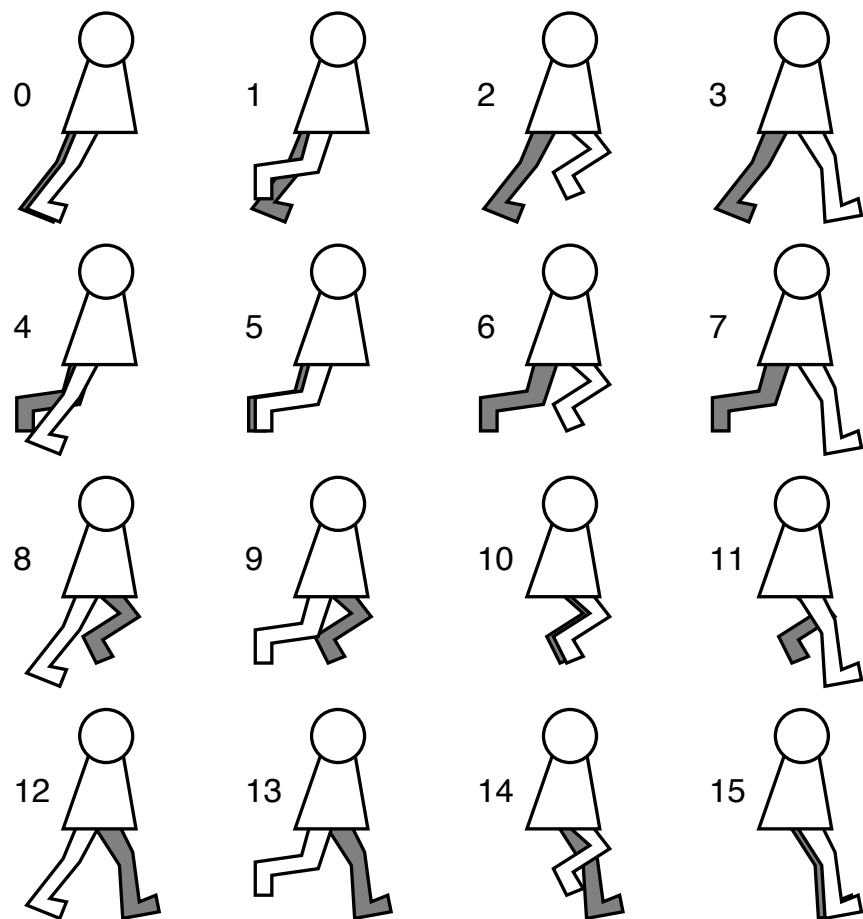
$$\text{Loss} = - R \log P(a) \text{ with } R \text{ being } r(s, a)$$

but that means that we have to save the gradients along our path through the state-action space (if we do not train immediately after each episode), and all the $\langle s, a, r \rangle$ tuples (or actually $\langle s, a, r, s' \rangle$)

Policy Gradients for Cartoon Walker

Represent the walker's policy in a network with

- a single valued array (one input value) for the state
- one of four possible output “classes” (sampled from probability distribution)
- softmax activation
- and not too many hidden neurons ;-)



Action	Effect
0	Move right (white) leg up / down
1	Move right (white) leg backward / forward
2	Move left (grey) leg up / down
3	Move left (grey) leg backward / forward

Will need a lot more time and tweaking than the policy iteration!

Outline

- Reinforcement learning
 - Problem definition
 - Learning situation
 - Roll of the reward
 - Simplified assumptions
 - Central concepts and terms
 - Known environment
 - Bellman's equation
 - Approaches to solutions
 - Outlook: unknown environments, Monte Carlo method and policy gradients
 - Unknown environment
 - Temporal-Difference learning
 - Q-Learning
 - Sarsa-Learning
 - Improvements
 - The usefulness of making mistakes
 - Eligibility Trace

Monte Carlo approach

Usually the reward $r(s, a)$ and the state transition function $\delta(s, a)$ are unknown to the learning agent.

(What does that mean for learning to ride a bike?



)

Still, we can estimate U^* from experience, as a *Monte Carlo approach* will do:

- Start with a randomly chosen s
- Follow a policy π , store rewards and s_t for the step at time t
- When the goal is reached, update the $U^\pi(s)$ estimate for all visited states s_t with the future reward that was given when reaching the goal
- Start over with a randomly chosen s ...

Converges slowly...

Temporal Difference learning

Temporal Difference learning ...

... uses the fact that there are two estimates for the value of a state:

before and after visiting the state

Or: What the agent believes *before* acting

$$U^{\pi}(s_t)$$

and *after* acting

$$r_{t+1} + \gamma \cdot U^{\pi}(s_{t+1})$$

Applying the estimates

The second estimate in the *Temporal Difference* learning approach is obviously “better”, ...

... hence, we update the overall approximation of a state's value towards the more accurate estimate

$$U^{\pi}(s_t) \leftarrow U^{\pi}(s_t) + \alpha [r_{t+1} + \gamma \cdot U^{\pi}(s_{t+1}) - U^{\pi}(s_t)]$$

Which gives us a measure of the “surprise” or “disappointment” for the outcome of an action.

Converges significantly faster than the pure Monte Carlo approach.

Q-learning

Problem:

even if U is appropriately estimated, it is not possible to compute π , as the agent has no knowledge about δ and r , i.e., it needs to learn also that.

Solution (trick): Estimate $Q(s, a)$ instead of $U(s)$:

$Q(s, a)$: Expected total reward when choosing a in s

$$\pi(s) = \underset{a}{\operatorname{argmax}} Q(s, a)$$

$$U^*(s) = \max_a Q^*(s, a)$$

Learning Q

How can we learn Q ?

Also the Q -function can be learned using the Temporal Difference approach:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

With s' being the next state that is reached when choosing action a'

Again, a problem: the \max operator requires obviously a search through all possible actions that can be taken in the next step...

SARSA-learning

SARSA-learning works similar to *Q-learning*, but it is the *currently active policy* that controls the actually taken action a' :

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$$

Got its name from the “experience tuples” having the form

State-Action-Reward-State-Action

$$\langle s, a, r, s', a' \rangle$$

Outline

- Reinforcement learning
 - Problem definition
 - Learning situation
 - Roll of the reward
 - Simplified assumptions
 - Central concepts and terms
 - Known environment
 - Bellman's equation
 - Approaches to solutions
 - Outlook: unknown environments, Monte Carlo method and policy gradients
 - Unknown environment
 - Temporal-Difference learning
 - Q-Learning
 - Sarsa-Learning
- Improvements
 - The usefulness of making mistakes
 - Eligibility Trace

Improvements and adaptations

What can we do, when ...

- ... the environment is not fully observable?
- ... there are too many states?
- ... the states are not discrete?
- ... the agent is acting in continuous time?

Allowing to be wrong sometimes

Exploration - Exploitation dilemma: When following one policy based on the current estimate of Q , it is not guaranteed that Q actually converges to Q^* (the optimal Q).

A simple solution: Use a policy that has a certain probability of “being wrong” once in a while, to explore better.

- **ϵ -greedy**: Will sometimes (with probability ϵ) pick a random action instead of the one that looks best (*greedy*)
- **Softmax**: Weighs the probability for choosing different actions according to how “good” they appear to be.

ϵ -greedy Q-learning

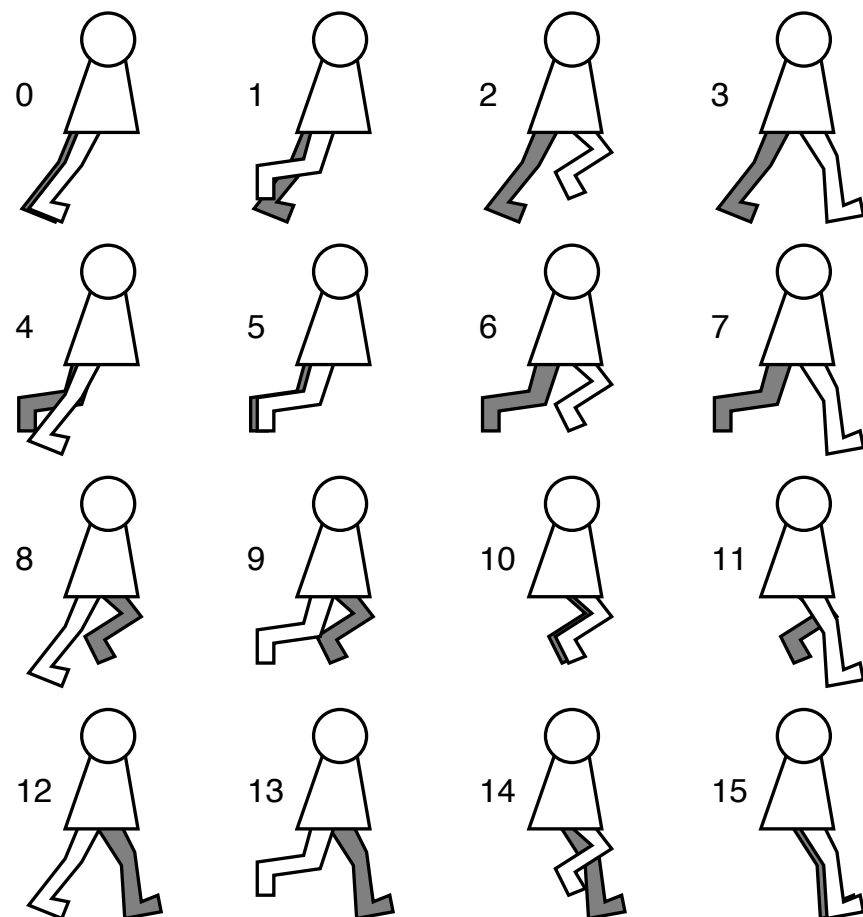
A suggested algorithm (ϵ -greedy implementation, given some “black box”, that produces r and s' , given s and a)

- Initialise $Q(s, a)$ arbitrarily $\forall s, a$, choose learning rate α and discount factor γ
- Initialise s
- Repeat for each step:
 - Choose a from s using ϵ -greedy policy based on $Q(s, a)$
 - Take action a , observe reward r , and next state s'
 - Update $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
 - replace s with s'

until T steps.

Q-Learning for Cartoon Walker

Use the reward-function estimated in the Policy Iteration experiment. Apply Q-Learning with ϵ -greedy policy to compute (s', a') from (s, a)



Action	Effect
0	Move right (white) leg up / down
1	Move right (white) leg backward / forward
2	Move left (grey) leg up / down
3	Move left (grey) leg backward / forward

```

for i in range(steps):
    a = eps_greedyPolicy(Q[s], eps)
    s_n, rew, _ = env.go(a) #obtain new state and reward from s, a
    sequence.append(s_n)
    Q[s][a] = Q[s][a] + eta*(rew + gamma * max( Q[s_n]) - Q[s][a])
    s = s_n
    
```

Speeding up the process

Idea: the Time Difference (TD) updates can be used to improve the estimation also of states where the agent has already been earlier.

$$\forall s, a : Q(s, a) \leftarrow Q(s, a) + \alpha [r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \cdot e$$

With e the eligibility trace, telling how long ago the agent visited s and chose action a

Often called $TD(\lambda)$, with λ being the time constant that describes the “annealing rate” of the trace.

Application examples

- End-to-end learning systems
 - learning to interact with humans
(Ali Ghadirzadeh et al, IROS 2016,
<https://ieeexplore.ieee.org/document/7759417>)
accessible from inside LU's network (or when running over VPN)

Application examples

- End-to-end learning systems
 - learning to throw a ball to hit the Pokémon
(Ali Ghadirzadeh et al, IROS 2017,
<https://ieeexplore.ieee.org/document/8206046>)
accessible from inside LU's network (or when running over VPN)

Application examples

- End-to-end learning systems
 - learning to play Go [<https://www.nature.com/articles/nature24270>]

Lab assignment 7

- The lab assignment is given as a package with instructions, code skeleton and some useful links also to hands-on material at <https://github.com/ErikGartner/edan95-rlagent-handout>
- Some hands-on experimenting material can be found at <https://github.com/ageron/handson-ml>