IN3050/IN4050, Lecture 12
Reinforcement Learning

1: Introduction

Ole Christian Lingjærde

Supervised learning

training set (x_i, y_i), $i=1,\ldots,n$

Unsupervised learning

training set x_i, $i=1,\ldots,n$

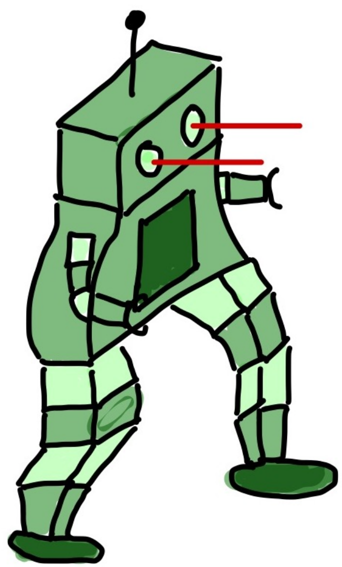
Reinforcement learning

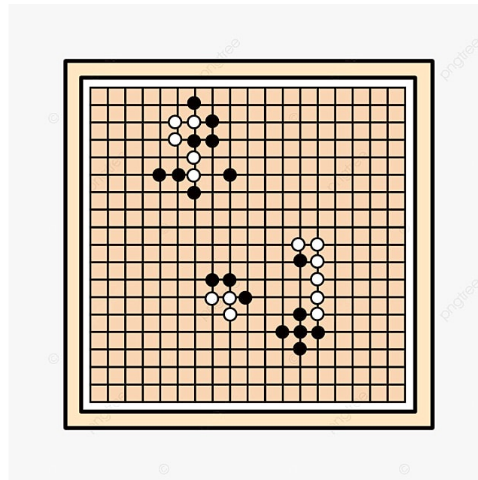
training set generated dynamically

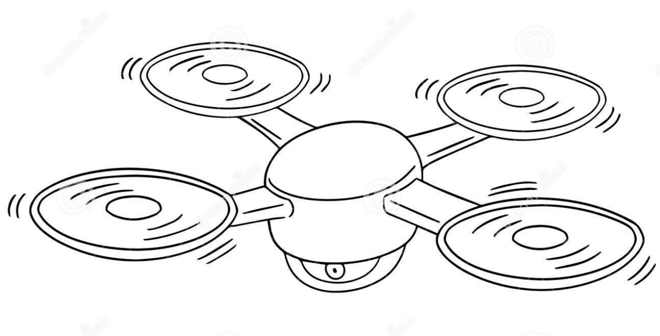
told if actions are good or bad

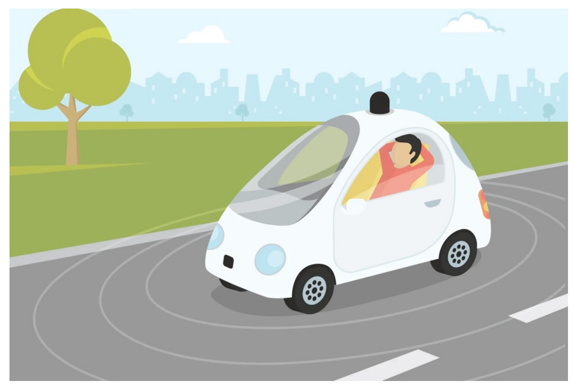
exploration to find right actions

Reinforcement learning (RL) applications

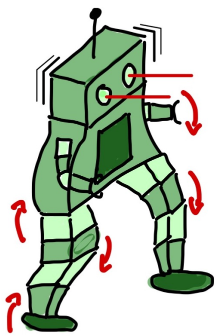








Robot with sensors and motors:



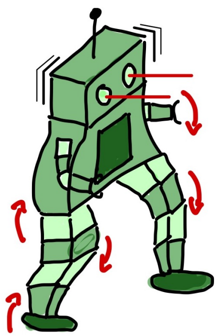
Observations:

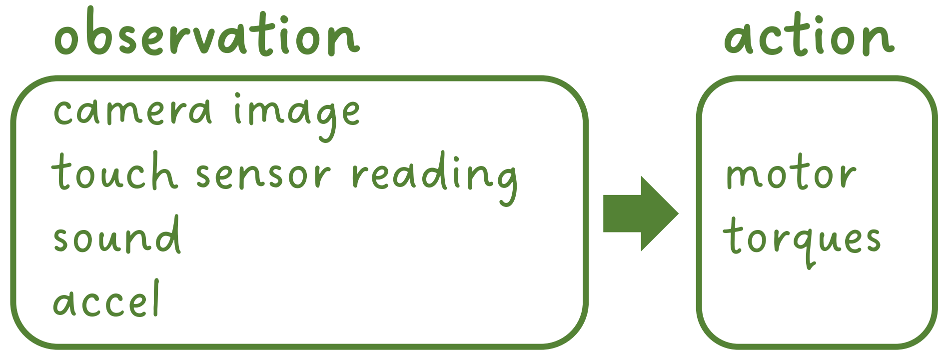
- camera sensors
- touch sensors
- accelerometers
- microphones

Actions:

- motor torques

Robot with sensors and motors:





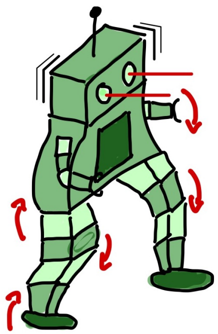
Observations:

- camera sensors
- touch sensors
- accelerometers
- microphones

Actions:

- motor torques

Robot with sensors and motors:



observation

camera image
touch sensor reading
sound
accel

→

action

motor
torques

Observations:

- camera sensors
- touch sensors
- accelerometers
- microphones

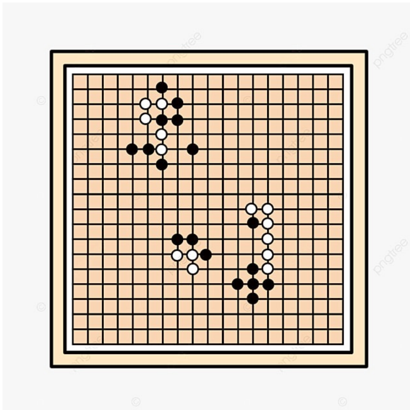
Goal:

- keep balance
- walk past obstacles
- reach a destination

Actions:

- motor torques

Computer playing a game



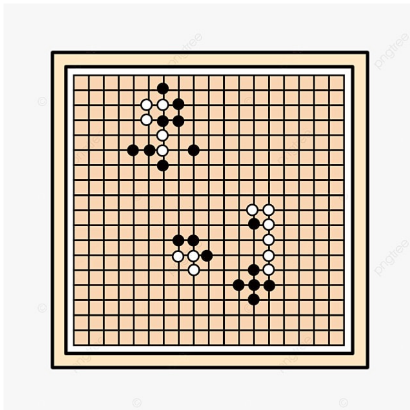
Observation:

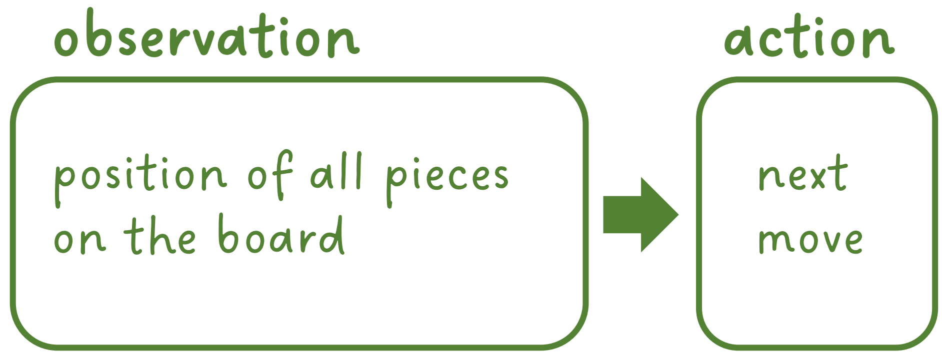
- current state

Action:

- next move

Computer playing a game





Observation:
- current state

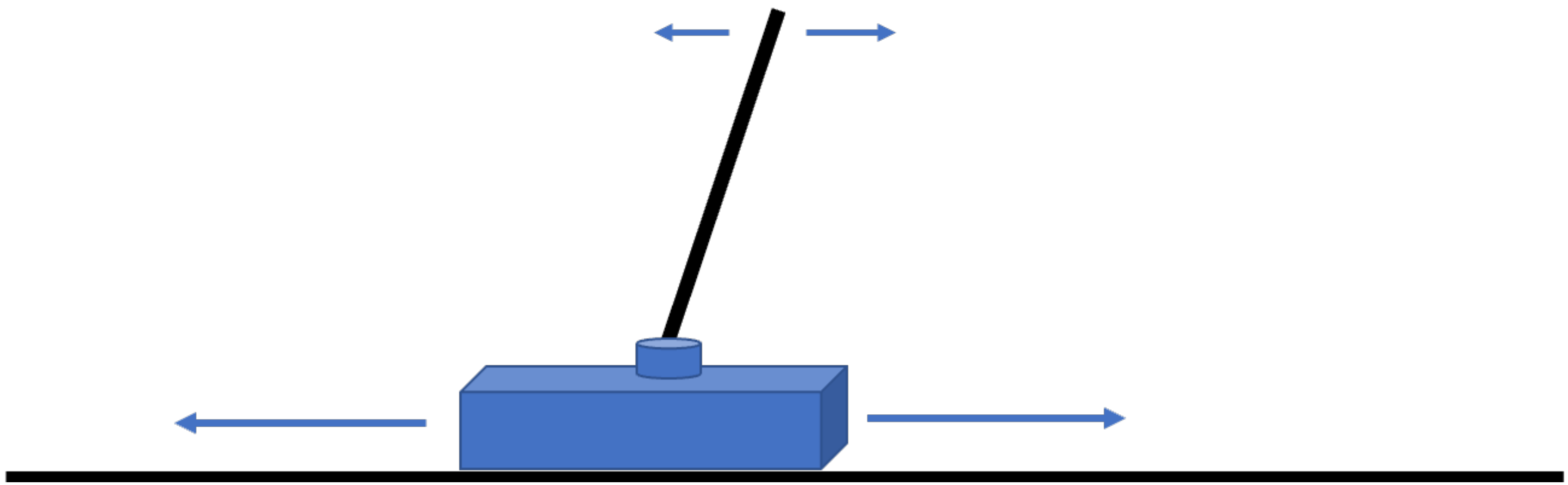
Action:
- next move

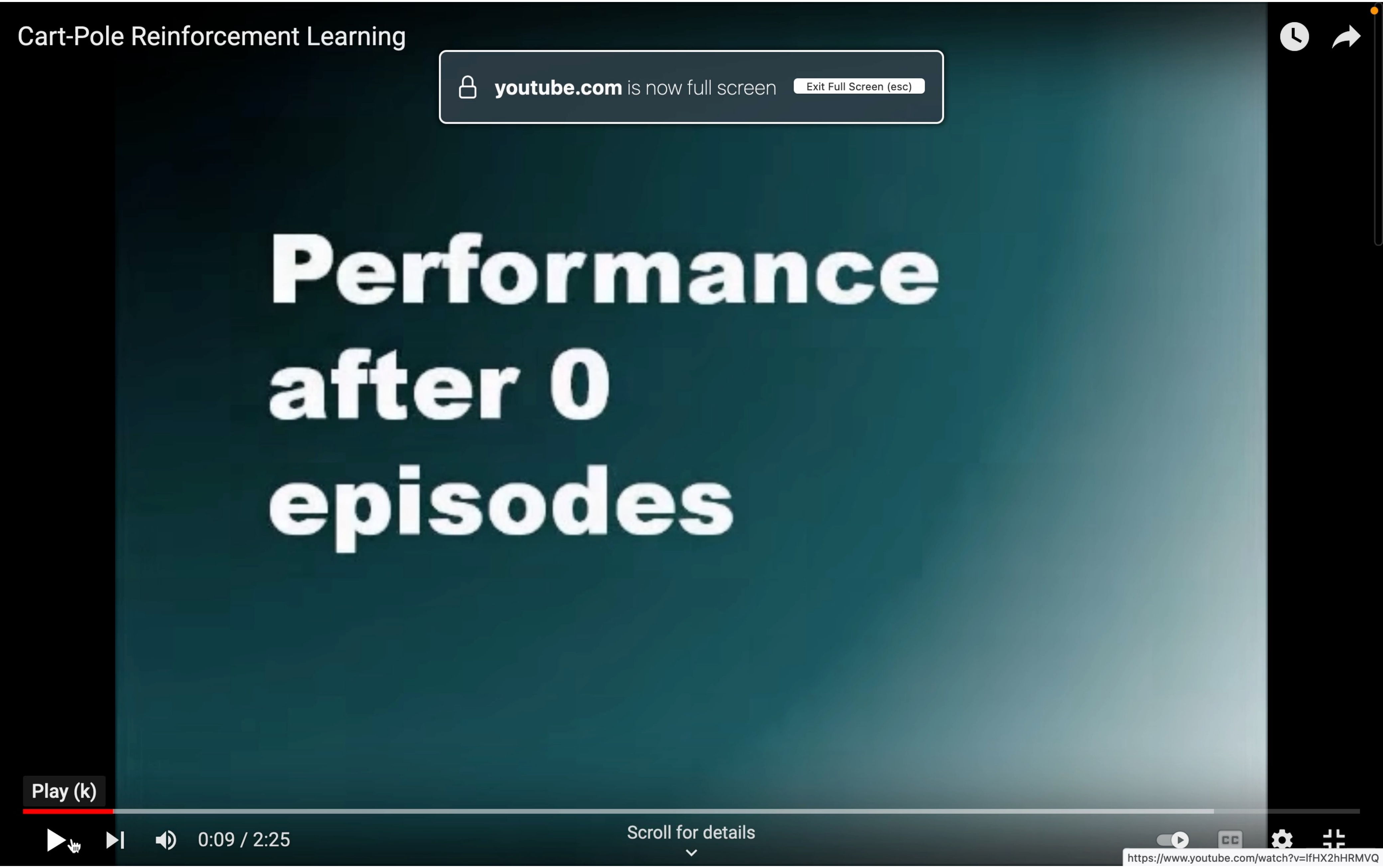
Goal: to win the game

Balancing a pole

Observations: angle between pole and cart

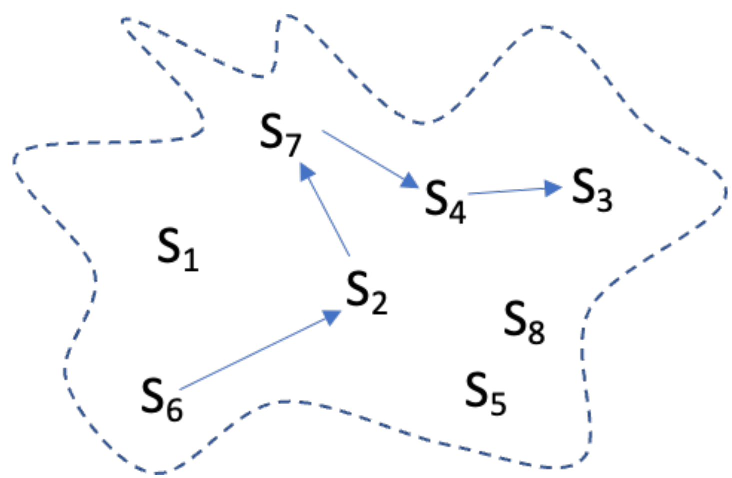
Actions: moving left or right




Cart-Pole Reinforcement Learning
youtube.com is now full screen
Exit Full Screen (esc)
Performance
after 0
episodes
Play (k)
0:09 / 2:25
Scroll for details
https://www.youtube.com/watch?v=lfHX2hHRMVQ

IN3050/IN4050, Lecture 12
Reinforcement Learning

2: Policy and states
Ole Christian Lingjærde



Policy

The way observations are mapped to actions defines a policy:

OBSERVATION → $\boxed{\text{POLICY}}$ → ACTION

A policy can be deterministic or stochastic.

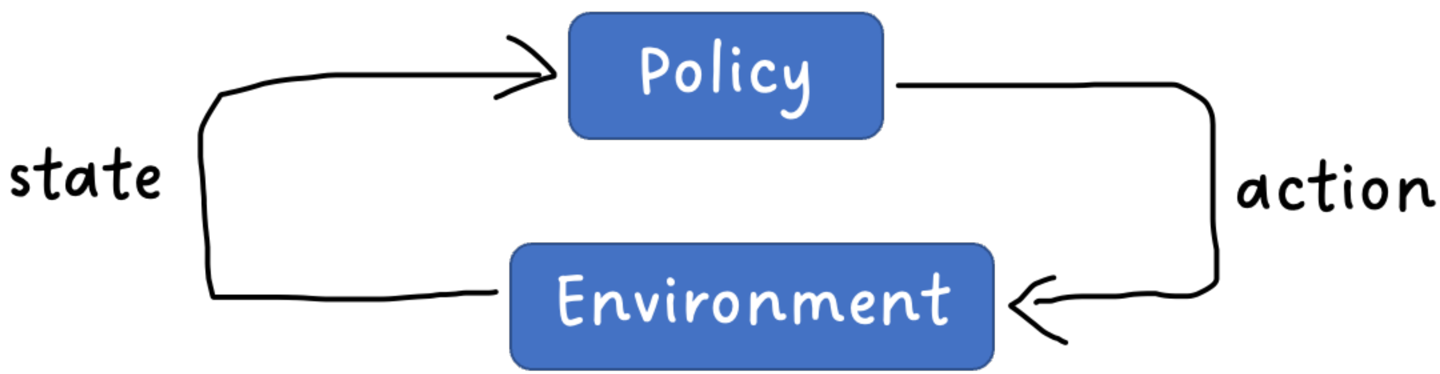
Exploitation vs. exploration

States

All RL systems work within an environment.

At any time the environment has a particular state and this is what we observe.

Actions can change the state:



Algorithm:

```
S[0] = <initial state>
for i in range(N):
    a = action(S[i])
    S[i+1] = newstate(S[i], a)
```

Algorithm:

```
S[0] = <initial state>
for i in range(N):
    a = action(S[i])                implements the policy
    S[i+1] = newstate(S[i], a)
```

Algorithm:

```
S[0] = <initial state>
for i in range(N):
    a = action(S[i])
    S[i+1] = newstate(S[i], a)
```

implements the policy

implements the environment - internal logic not known

Algorithm:

```
S[0] = <initial state>
for i in range(N):
    a = action(S[i])
    S[i+1] = newstate(S[i], a)
```

implements the policy

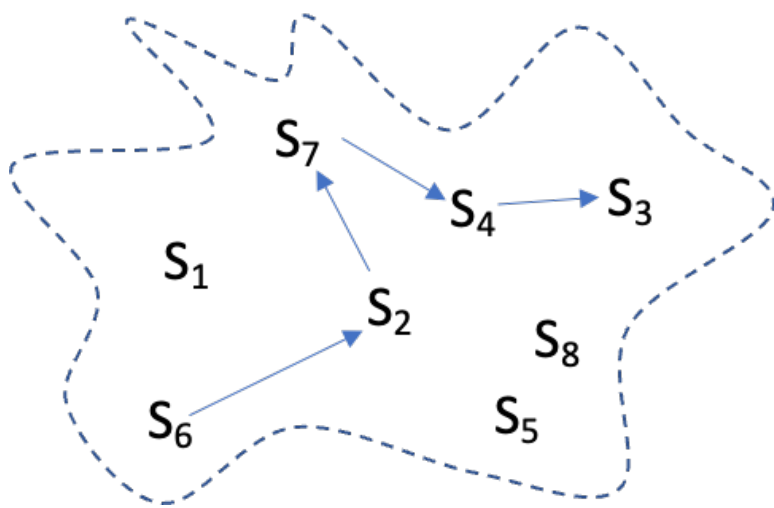
implements the environment - internal logic not known

Result: S[0] S[1] S[2] ...... S[N-1]

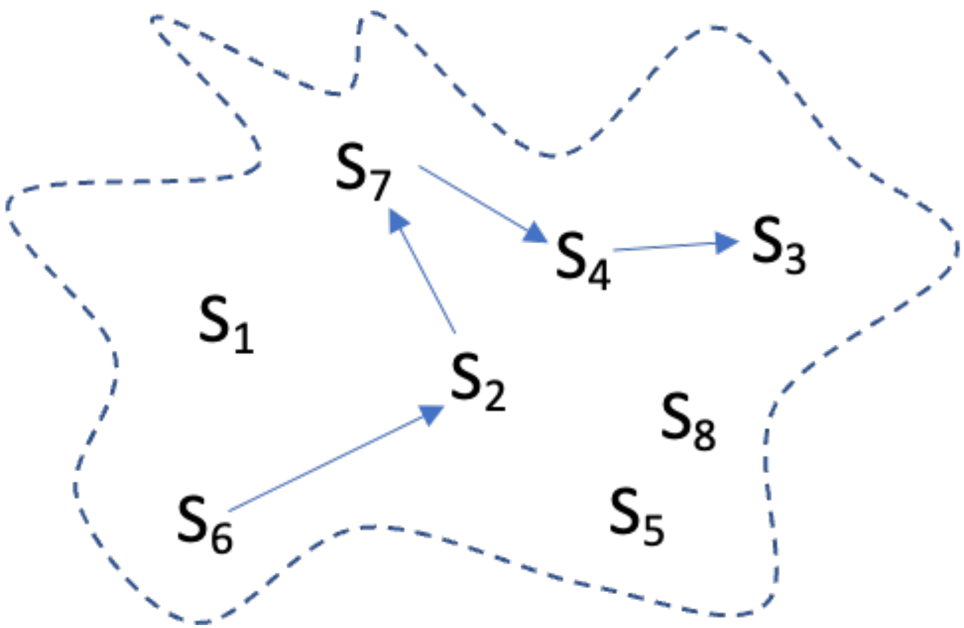
An RL session involves a series of state changes each being the result of an action:

$$s^1 \xrightarrow{a^1} s^2 \xrightarrow{a^2} s^3 \xrightarrow{a^3} s^4 \xrightarrow{a^4} s^5 \xrightarrow{a^5} \ldots$$

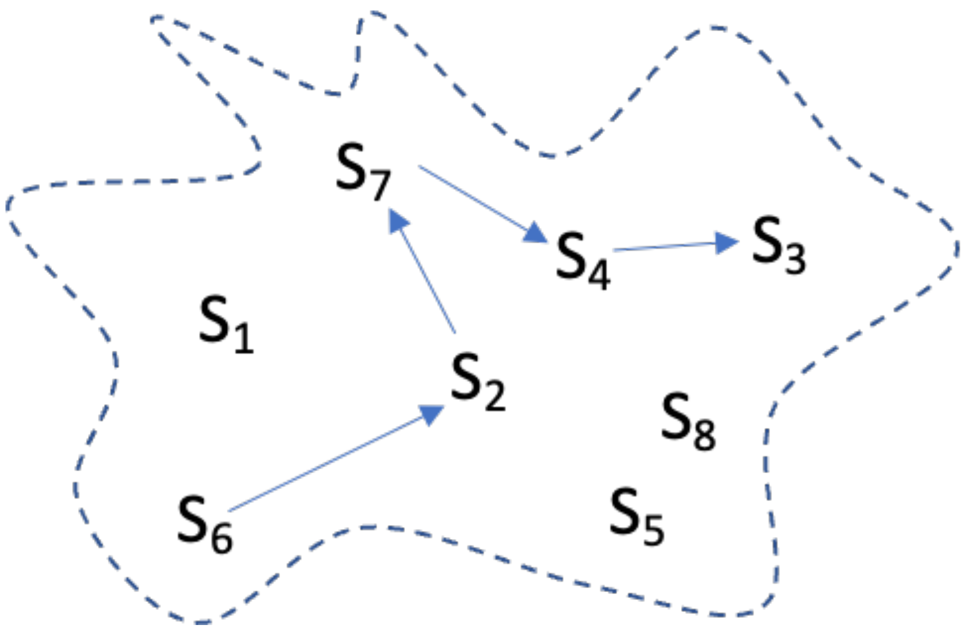
This trails a path in state space:



Typical problem in RL:

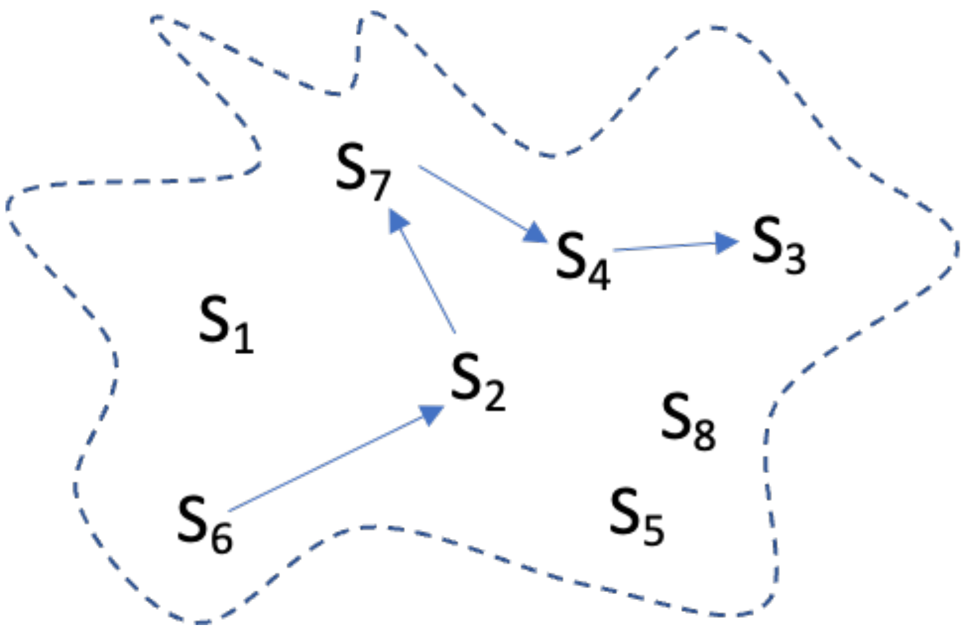


Typical problem in RL:



Goal:
to get from a start state to a destination state in the least number of moves

Typical problem in RL:



Goal:

to get from a start state
to a destination state in
the least number of moves

Constraints:

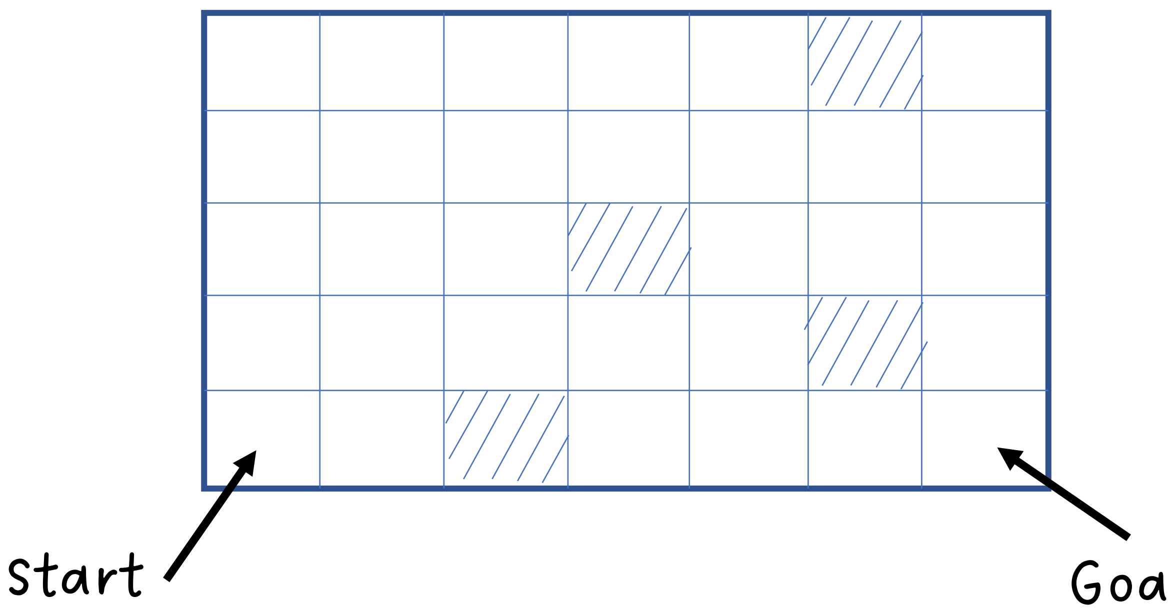
only some state
transitions are allowed

some moves are bad and
some are good

The iced lake

Environment: iced lake with holes

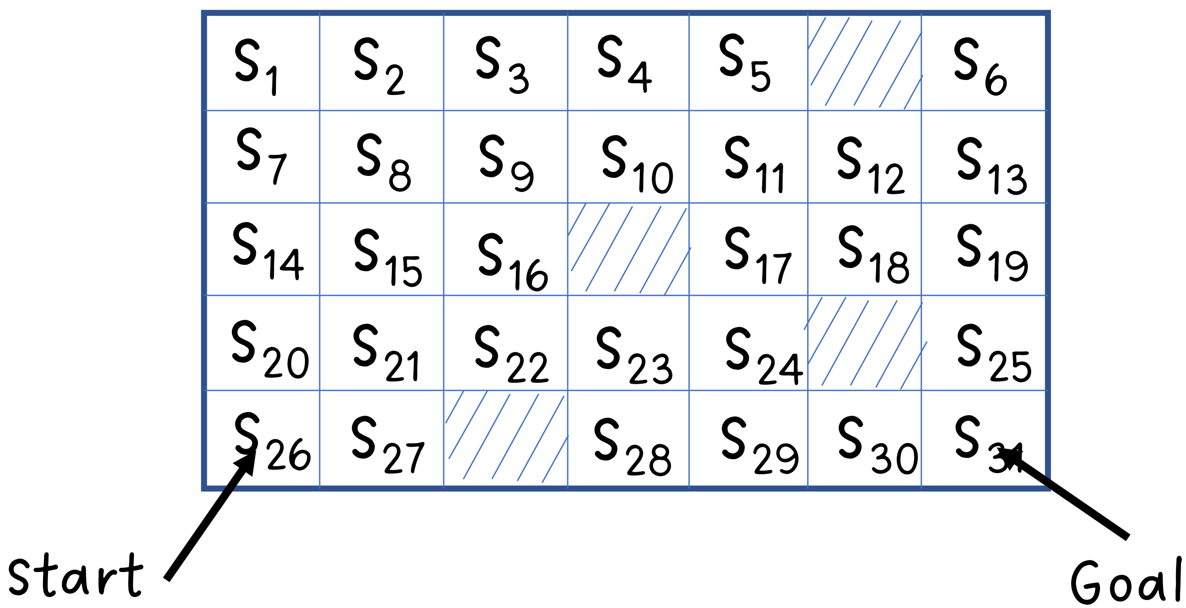
Aim: learn the shortest route from start to goal



The iced lake

The environment is an iced lake with holes

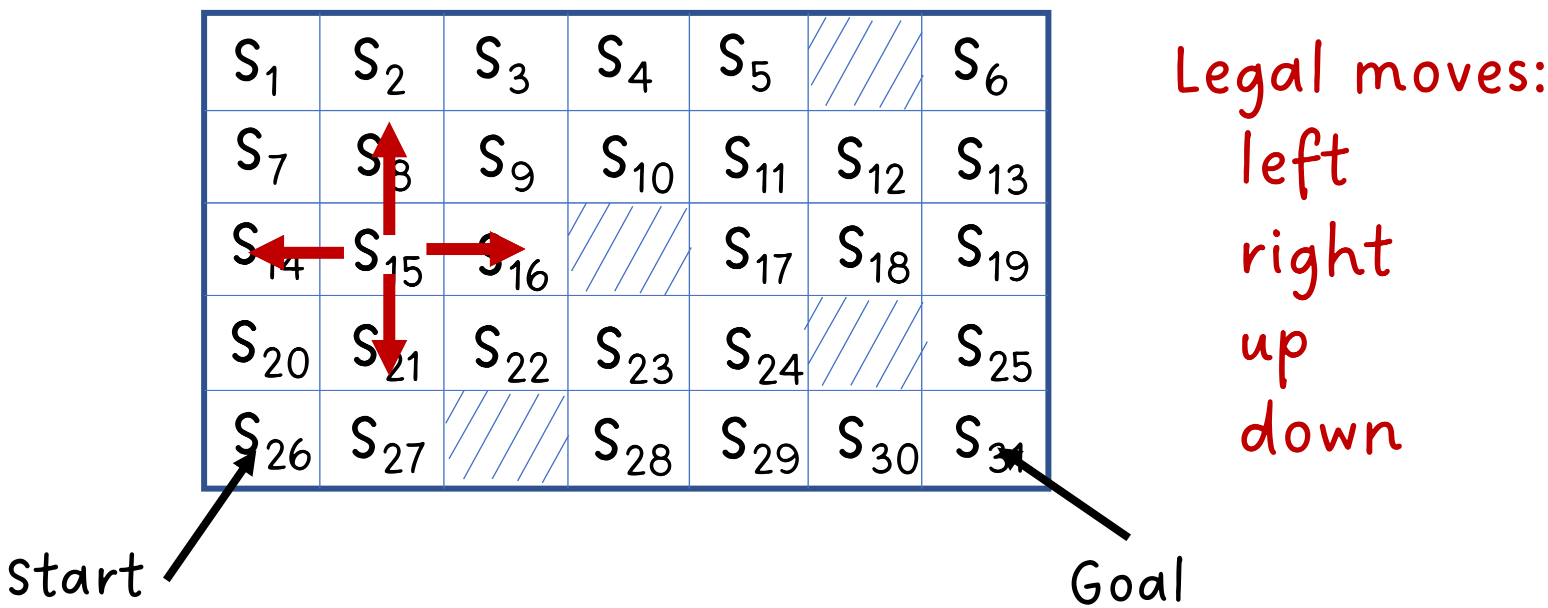
Aim is to learn the shortest route from start to goal



The iced lake

The environment is an iced lake with holes

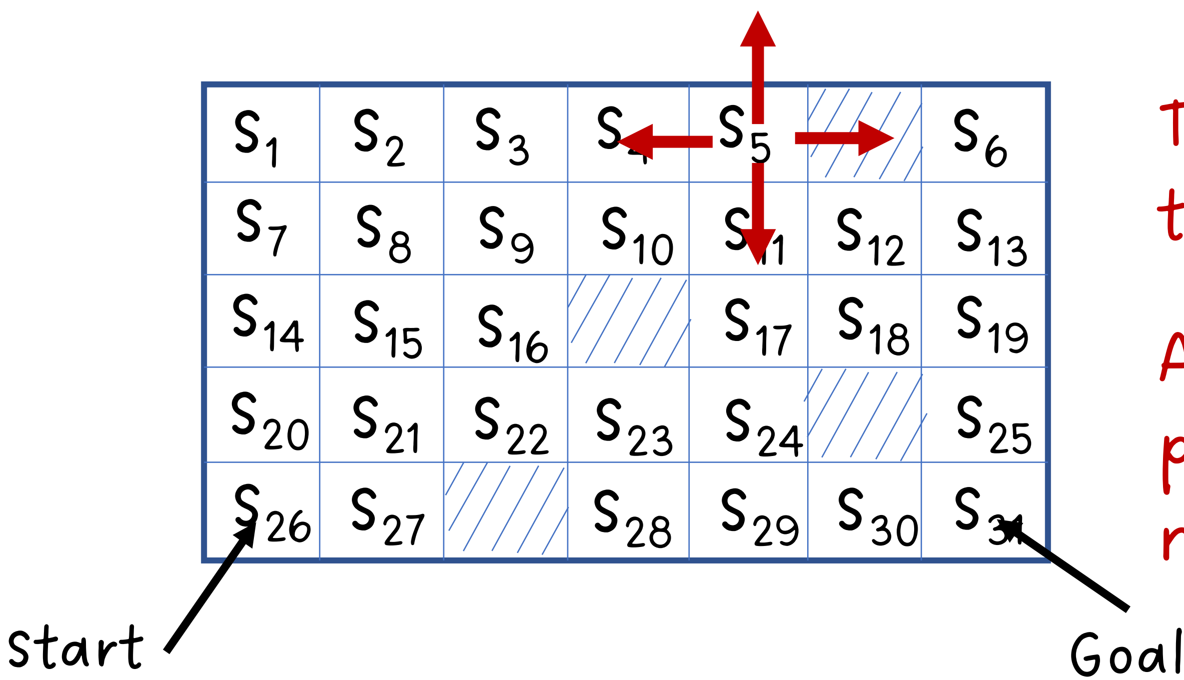
Aim is to learn the shortest route from start to goal



The iced lake

The environment is an iced lake with holes

Aim is to learn the shortest route from start to goal



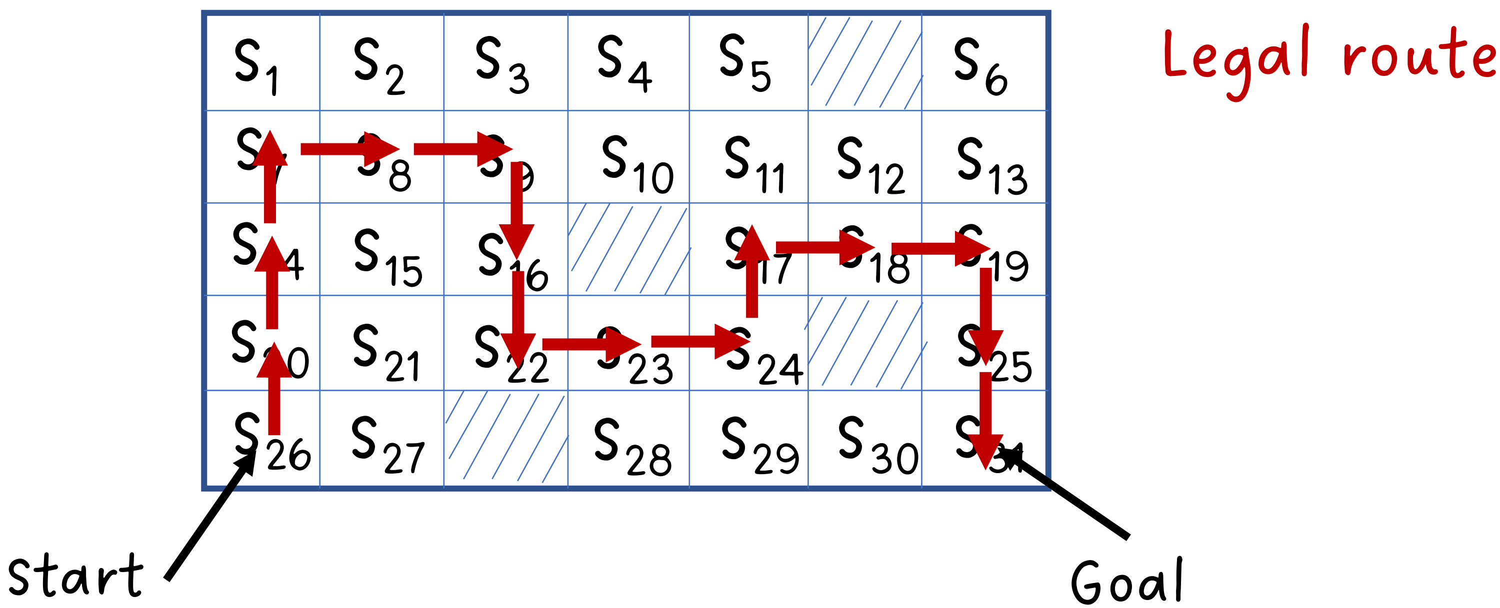
Two actions lead to illegal states

Attempts to perform these will not change state

The iced lake

The environment is an iced lake with holes

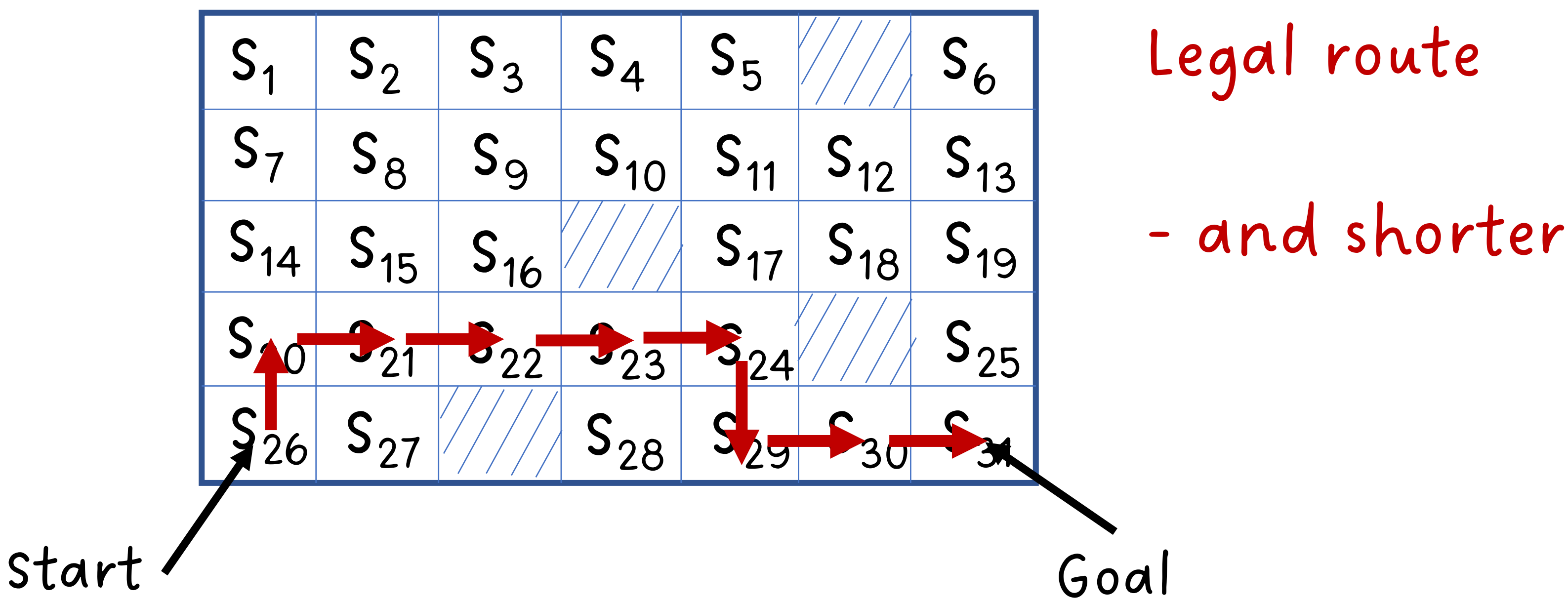
Aim is to learn the shortest route from start to goal



The iced lake

The environment is an iced lake with holes

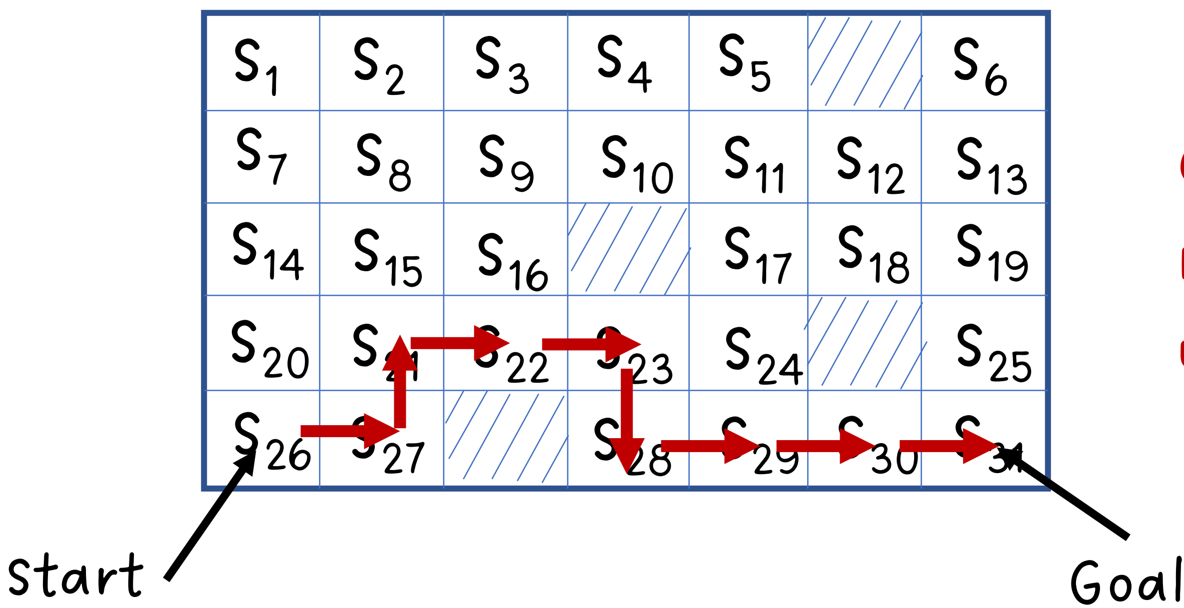
Aim is to learn the shortest route from start to goal



The iced lake

The environment is an iced lake with holes

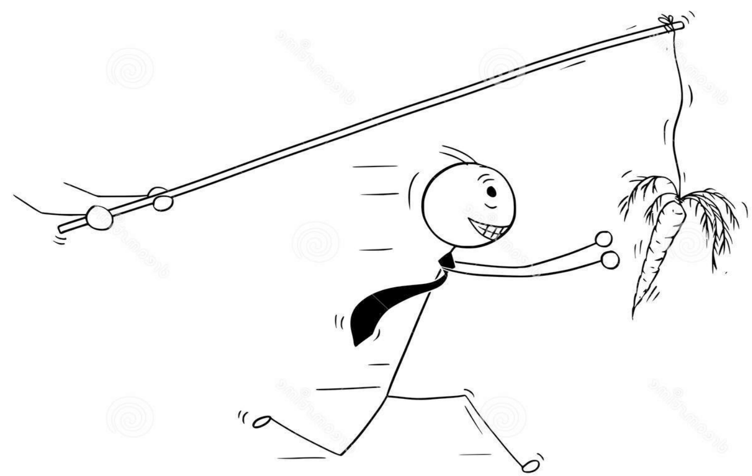
Aim is to learn the shortest route from start to goal



Optimal route may not be unique!

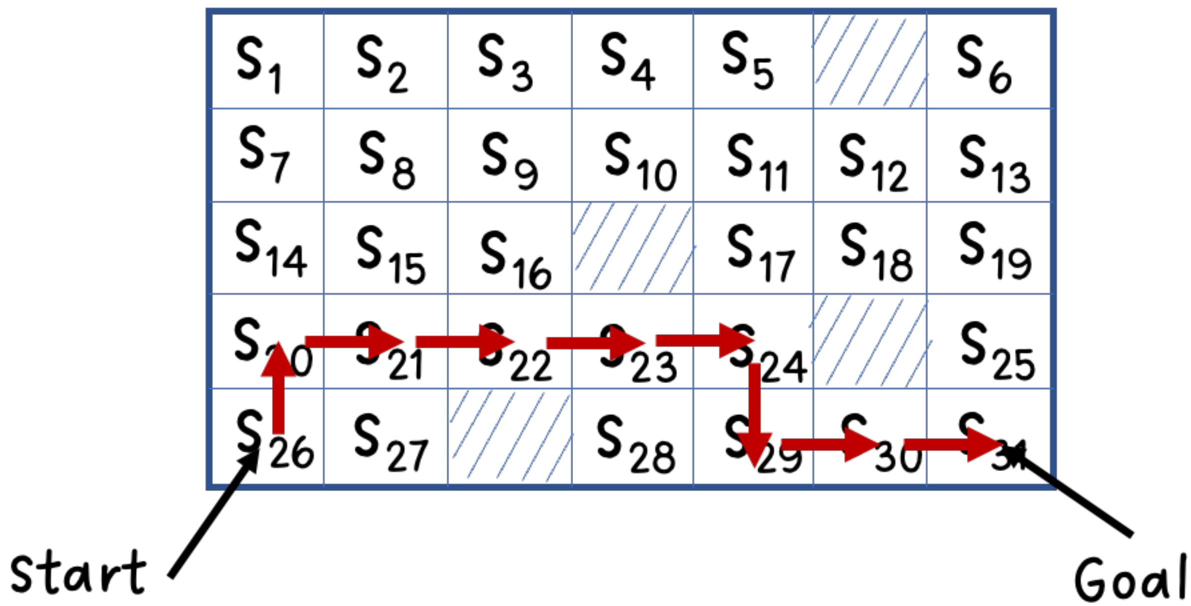
IN3050/IN4050, Lecture 12
Reinforcement Learning

3: Learning from rewards
Ole Christian Lingjærde



Finding the best action

It may seem easy for a human to find the best action in the iced lake example:



But can we train a computer to do it, too?

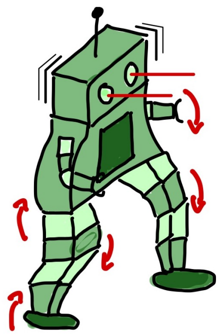
Finding the best action

If we could make a computer tackle the iced lake example, we may use same principle to solve much harder problems:

robotics
self-driving cars
health care
finance
...

This is where reinforcement learning (RL) comes in.

Supervised learning is not practical



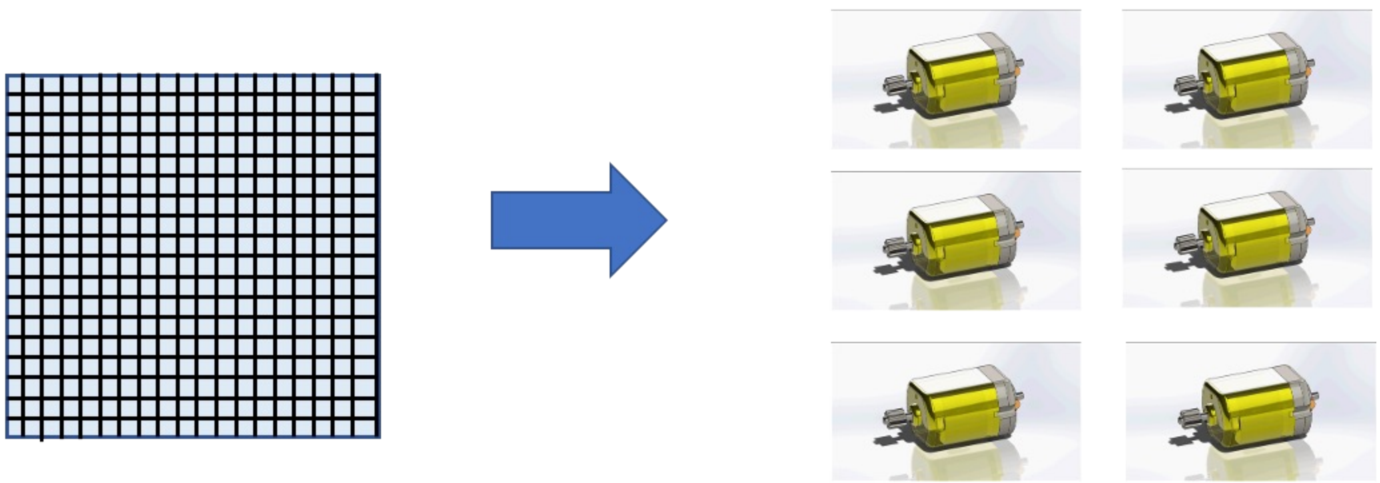
Training set: (x_i, y_i), $i=1,\ldots,n$

x_i : current sensor readings

y_i : correct/best action

Pixel values, sensor readings, ...

Correct torque for each motor



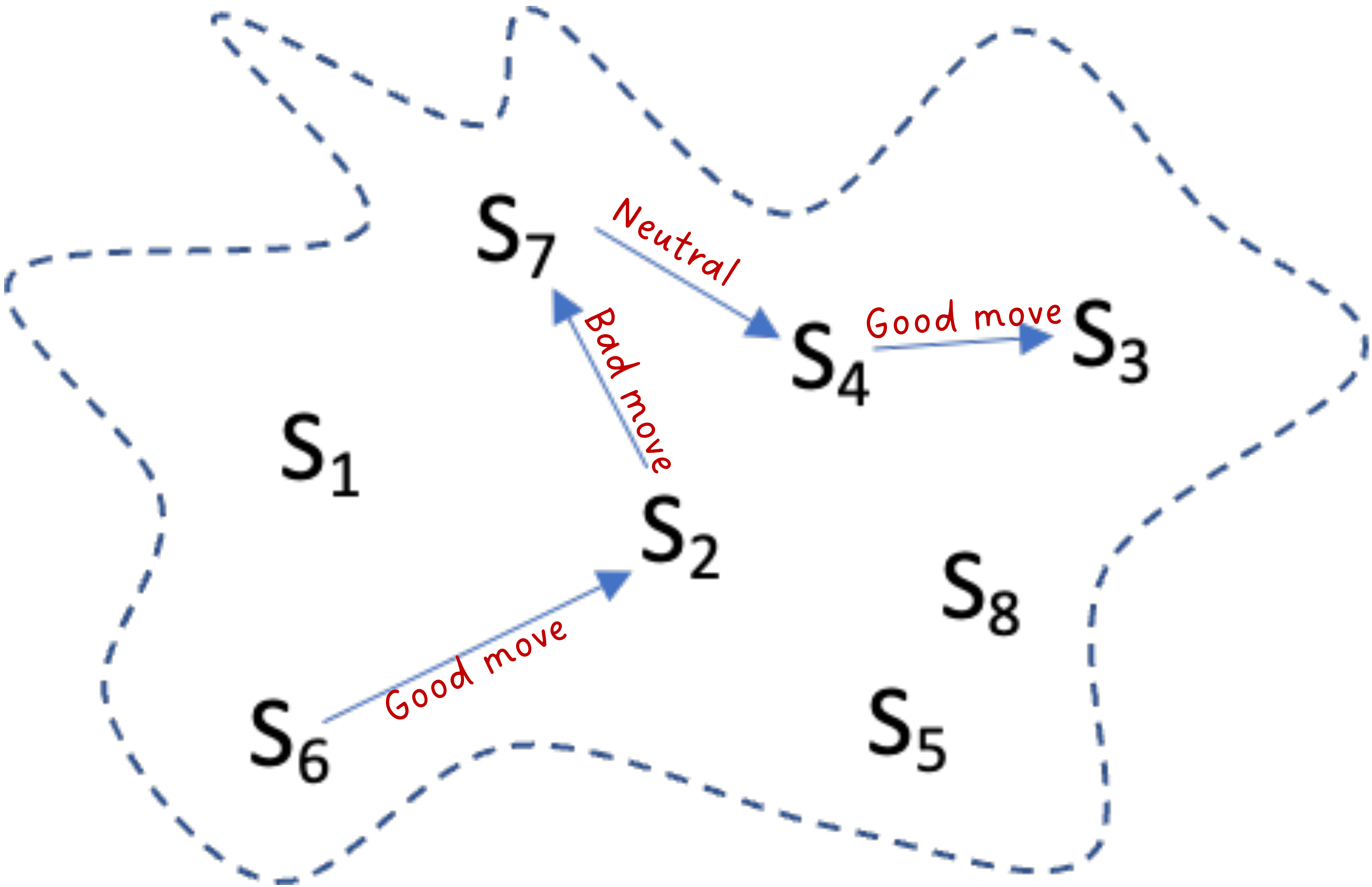
Difficult and extremely time consuming to create such data

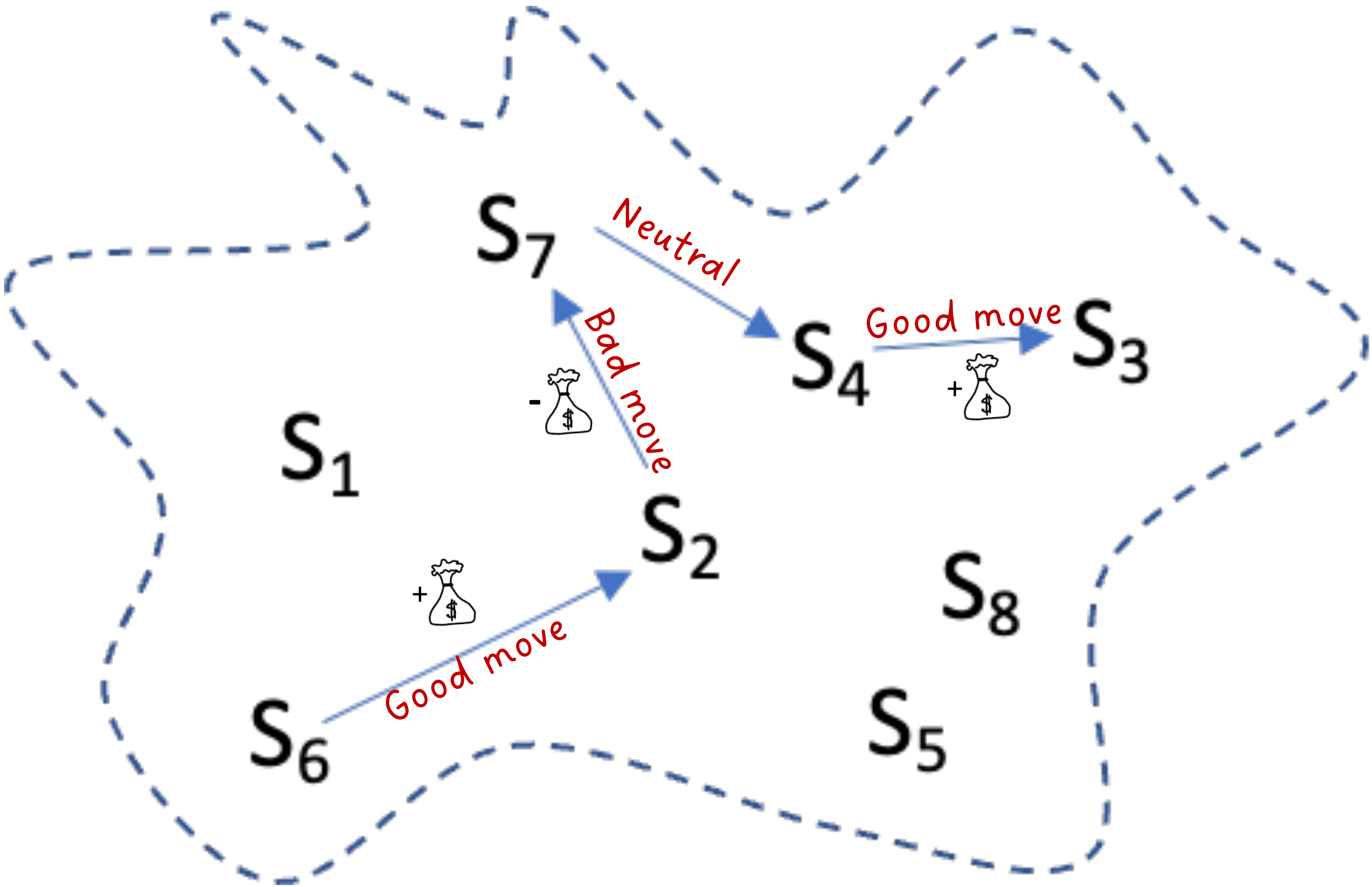
Rewards

Instead of supplying the learner with the correct outputs (which we do not possess) we will tell if the result of an action was good or bad.

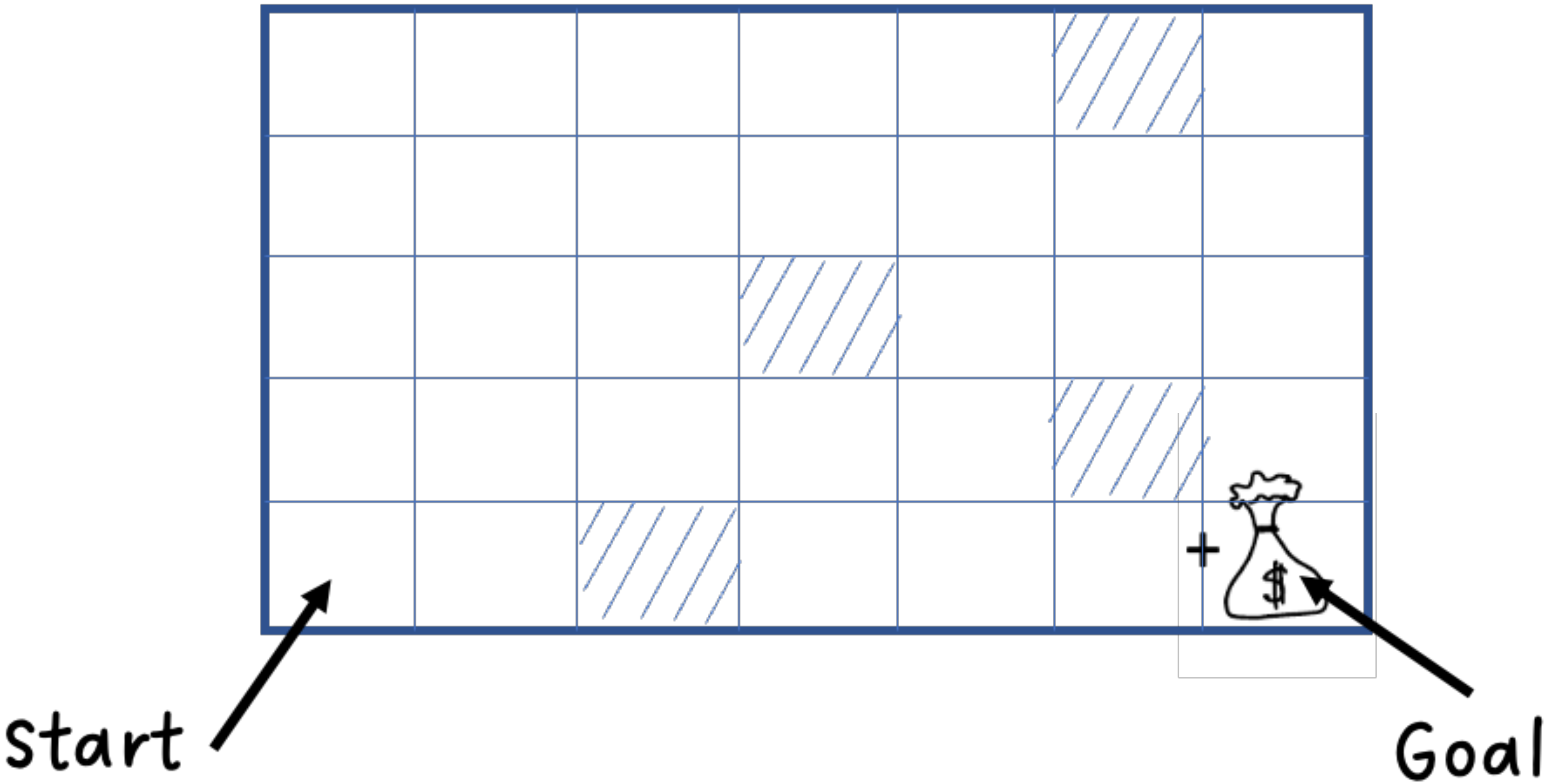
+  = reward

-  = negative reward (penalty)

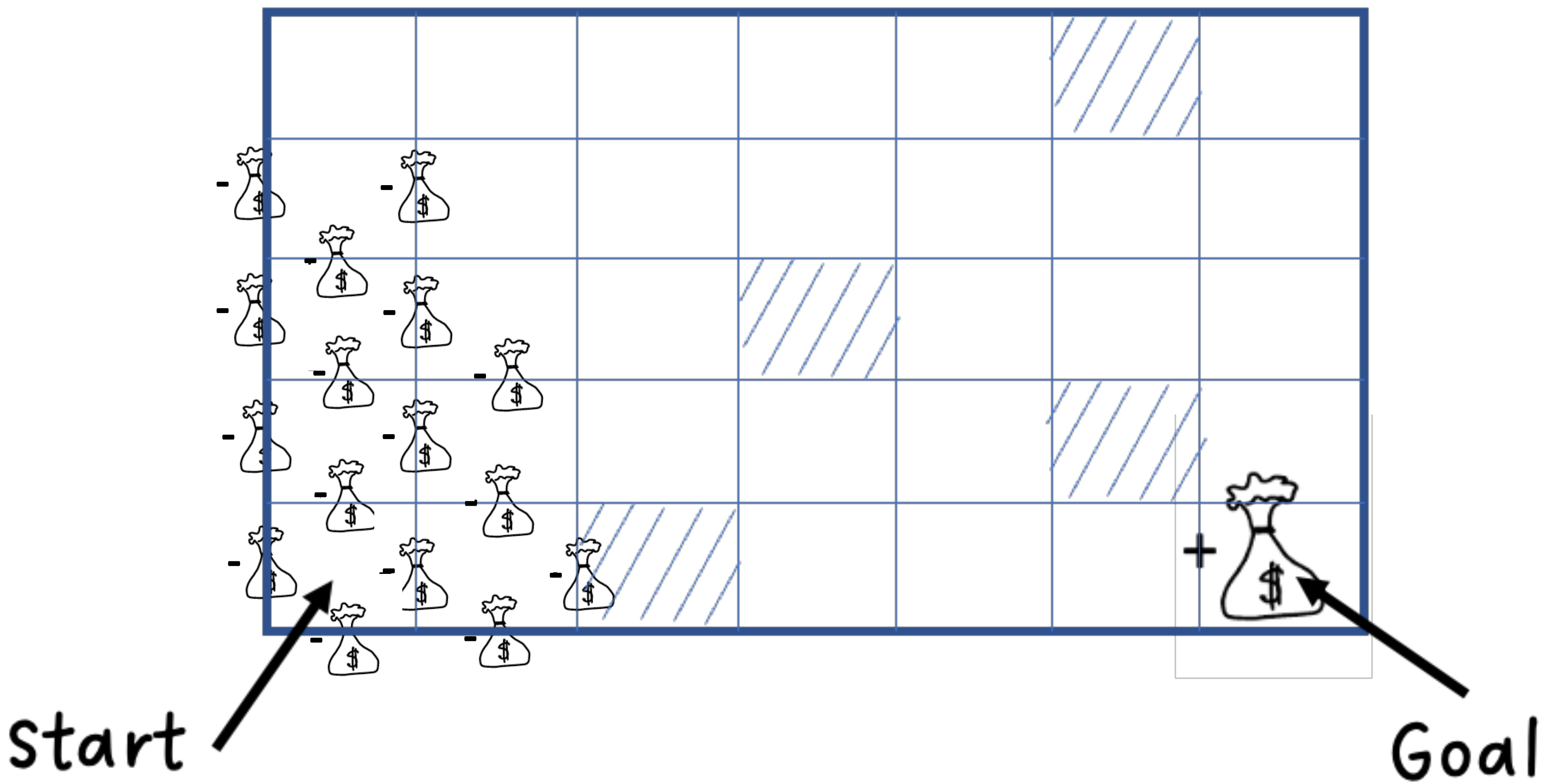

S7
Neutral
S4
Good move
S3
Bad move
S1
S2
S8
Good move
S6
S5


S7
Neutral
S4
Good move
S3
+
Bad move
-
S1
S2
S8
+
Good move
S6
S5

The iced lake

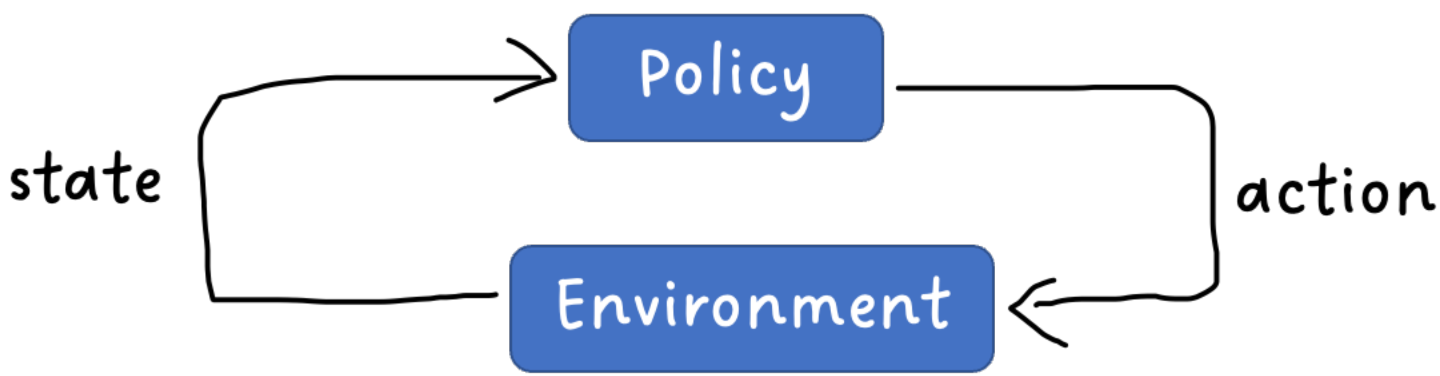


The iced lake



A static policy means no improvement over time

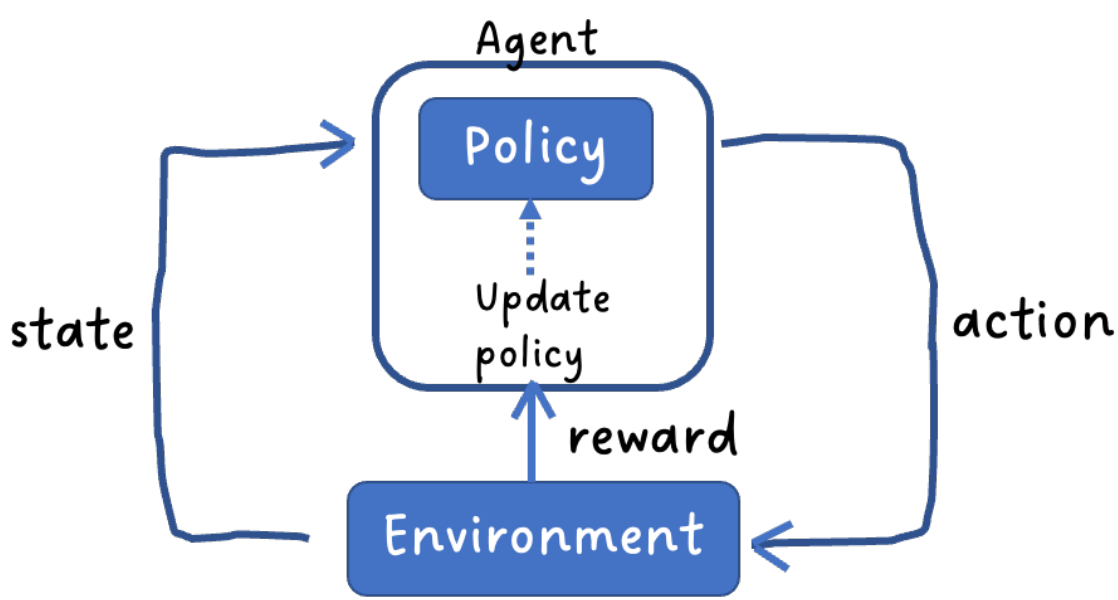
In the static system described earlier in this lecture, the policy was fixed and the system could not improve over time:



Learning means changing the policy

By allowing the policy to change over time, the system can adapt to the environment.

An agent repeatedly adjusts the policy based on the previous state, the current state and the reward received:



Reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
    a = action(S[i], w)
    S[i+1] = newstate(S[i], a)
    r = reward(S[i], S[i+1], a)
    <update w>
```

Reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
    a = action(S[i], w)              # POLICY
    S[i+1] = newstate(S[i], a)
    r = reward(S[i], S[i+1], a)
    <update w>
```

Reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
    a = action(S[i], w)            # POLICY
    S[i+1] = newstate(S[i], a)     # ENVIRONMENT
    r = reward(S[i], S[i+1], a)
    <update w>
```

Reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
    a = action(S[i], w)            # POLICY
    S[i+1] = newstate(S[i], a)     # ENVIRONMENT
    r = reward(S[i], S[i+1], a)    # ENVIRONMENT
    <update w>
```

Reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
    a = action(S[i], w)              # POLICY
    S[i+1] = newstate(S[i], a)       # ENVIRONMENT
    r = reward(S[i], S[i+1], a)      # ENVIRONMENT
    <update w>                       # POLICY LEARNING
```

Reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
    a = action(S[i], w)                  # MUST DEFINE
    S[i+1] = newstate(S[i], a)
    r = reward(S[i], S[i+1], a)
    <update w>                           # MUST DEFINE
```

Reinforcement learning is stochastic

Just as in evolutionary learning, there are several ways to introduce randomness into RL:

```
S[0] = <initial state>              # Can be stochastic
w = <initial weights>               # Can be stochastic
 for i in range(N):
     a = action(S[i], w)            # Can be stochastic
     S[i+1] = newstate(S[i], a)     # Can be stochastic
     r = reward(S[i], S[i+1], a)    # Can be stochastic
     <update w>                     # Can be stochastic
```

Reinforcement learning involves exploration

Reinforcement learning receives less guidance from trainer than supervised learning.

Exploration is essential:

- explore different actions
- see where they lead
- accumulate knowledge

As training progresses, exploration can be replaced by exploitation.

IN3050/IN4050, Lecture 12
Reinforcement Learning

4: Defining the policy
Ole Christian Lingjærde

Recall the reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
        a = action(S[i], w)
        S[i+1] = newstate(S[i], a)
        r = reward(S[i], S[i+1], a)
        <update w>
```

Recall the reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
        a = action(S[i], w)
        S[i+1] = newstate(S[i], a)
        r = reward(S[i], S[i+1], a)
        <update w>
```

Defines the current policy
Adjustable through the parameter vector w

How do we design this function?

The policy

Decides what action(S,w) to take in a given state S, using prior knowledge collected in a parameter vector w.

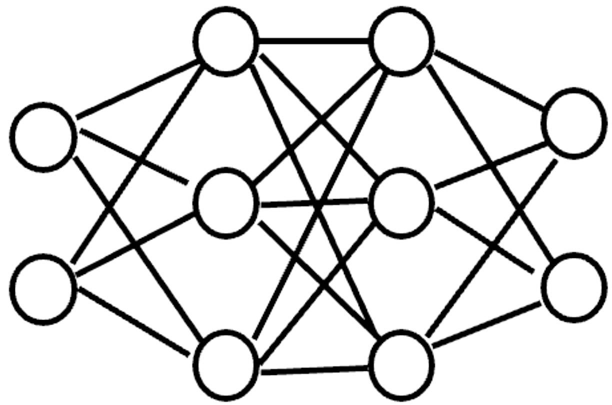
Two common approaches:

- Represent policy as a neural network
- Represent policy as a table

Representing the policy as a neural network

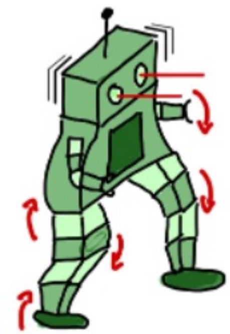
input

digital image
sound
touch sensors
accel
sound



output

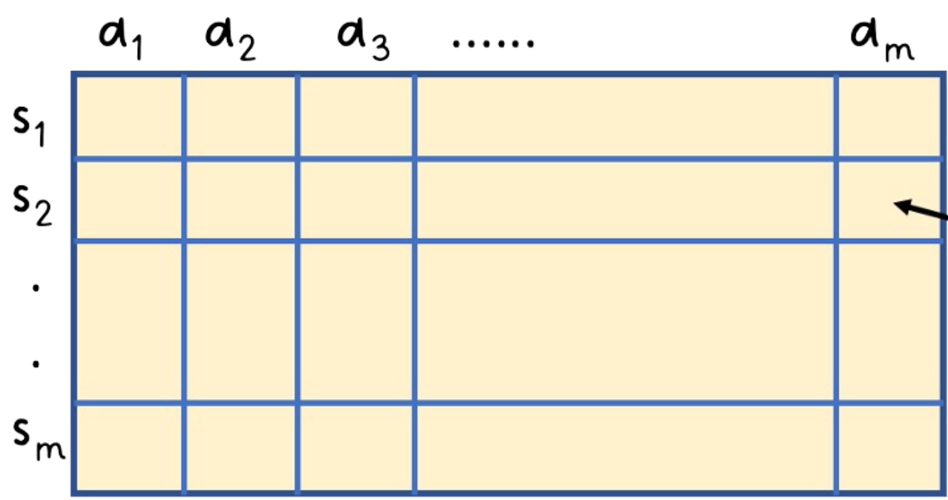
motor torques



The weights in the network are given by the parameter vector w in action(S,w)

Representing the policy as a table

An alternative to neural networks and the idea is to store all previous learning experience in a table with one entry for each (state, action) combination:



average reward obtained after taking this action when in this state

Q-table

Illustration:

	a_1	a_2	a_3
s_1	5	-1	3
s_2	0	2	11
s_3	7	9	-5
s_m	8	2	0

In state s_3 the highest average reward is obtained when taking action a_2

```
# S = 0,1,2,... is a state and Q is a numpy table.
def action(S, Q):
    return np.where(Q[S,:] == max(Q[S,:]))[0][0]
```

Three different ways of using the Q-table

greedy: $\text{action} = \underset{a}{\operatorname{argmax}}\, Q[s,a]$

ε-greedy: $$\text{action} = \begin{cases} \underset{a}{\operatorname{argmax}}\, Q[s,a], & \text{with prob } 1-\varepsilon \\ \text{random action}, & \text{with prob } \varepsilon \end{cases}$$

softmax: $$\text{action} = \begin{cases} a_1 & \text{with prob } \dfrac{e^{Q[s,a_1]/\tau}}{\sum e^{Q[s,a_k]/\tau}} \\ \dots & \dots\dots \\ a_m & \text{with prob } \dfrac{e^{Q[s,a_m]/\tau}}{\sum e^{Q[s,a_k]/\tau}} \end{cases}$$

An example

Suppose we are in state s_2.
How do we choose the next action?

	a_1	a_2	a_3
s_1	5	-1	3
s_2	0	2	11
s_3	7	9	-5
s_m	8	2	0

greedy: choose a_3

ε-greedy: Draw a random number $X \sim U[0,1]$
if $X \geq \varepsilon$: choose a_3
else: select action at random

softmax: Draw $(X_1, X_2, X_3) \sim$ multinomial(1; p1,p2,p3)
if $X_1 = 1$: choose a_1
else if $X_2 = 1$: choose a_2
else if $X_3 = 1$: choose a_3

$$p_1 = \frac{\exp(0)}{\exp(0)+\exp(2)+\exp(11)}$$

$$p_2 = \frac{\exp(2)}{\exp(0)+\exp(2)+\exp(11)}$$

$$p_3 = \frac{\exp(11)}{\exp(0)+\exp(2)+\exp(11)}$$

IN3050/IN4050, Lecture 12
Reinforcement Learning

5: Learning the policy
Ole Christian Lingjærde

Main reinforcement learning cycle:

```
S[0] = <initial state>
w = <initial weights>
for i in range(N):
        a = action(S[i], w)
        S[i+1] = newstate(S[i], a)
        r = reward(S[i], S[i+1], a)
        <update w>
```

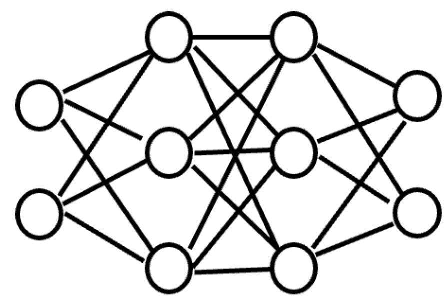
Main reinforcement learning cycle:

```
S[0] = <initial state>
w = <weights from a previous learning cycle (episode)>
for i in range(N):
        a = action(S[i], w)
        S[i+1] = newstate(S[i], a)
        r = reward(S[i], S[i+1], a)
        <update w>
```

Learning (= updating w)

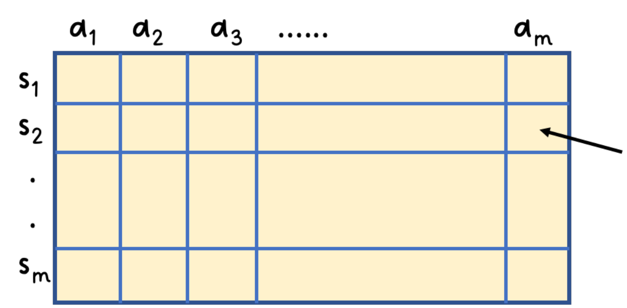
Neural network implementation of policy:

w = network weights
policy-gradient methods
actor-critic methods



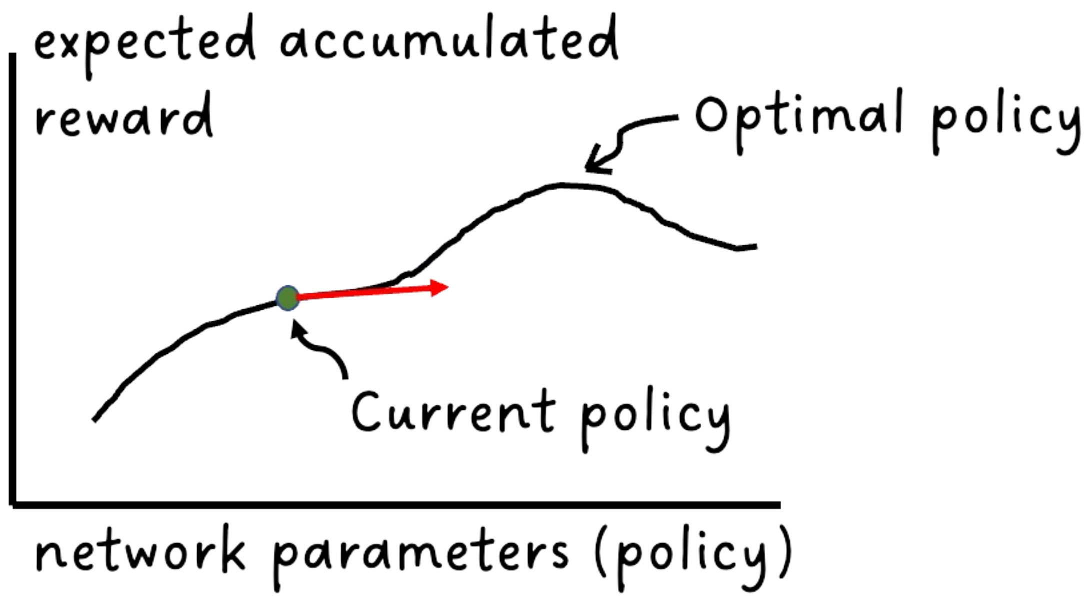
Q-table implementation of policy:

w = the table Q
SARSA
Q-learning



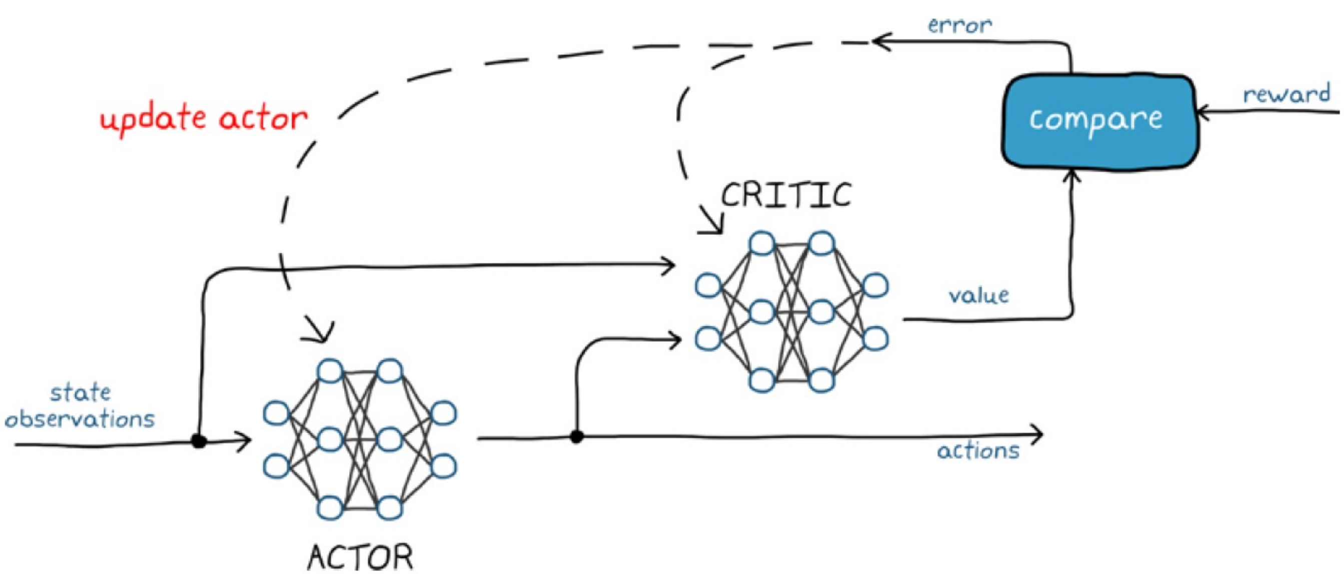
Policy-gradient methods

- Execute current policy
- Collect rewards
- After several iterations adjust the weights in the direction of increased expected reward



Actor-critic methods

- Two neural networks interact and are trained together
- The critic network learns the values of different (state, action) pairs from the received rewards
- The actor network learns correct actions using feedback from the critic network



Learning algorithms for Q-tables

Store all previous learning experience as a table.

average reward obtained after taking this action when in this state

Suppose we have experienced (s_i, a_j) a total of k times and have received rewards

$$r_1, r_2, \ldots, r_k$$

Seems reasonable to let

$$Q[s_i, a_j] = (r_1 + r_2 + \ldots + r_k)/k$$

NOTE: If the environment is completely deterministic then newstate(...) and reward(...) are deterministic, too.

Then $r_1 = r_2 = \ldots = r_k$.

Problem 1

The environment may change over time and in that case we would trust recent rewards more than older ones

Solution:

Suppose we are in state s, current policy says take action a, and the reward is r. Then we let

$$Q[s,a] = (1-\mu)Q[s,a] + \mu r$$

Updates (first to last):

$$Q[s,a] = \mu r_1$$

$$Q[s,a] = (1-\mu)\mu r_1 + \mu r_2$$

$$Q[s,a] = (1-\mu)^2\mu r_1 + (1-\mu)\mu r_2 + \mu r_3$$

Problem 2

The update rule only considers the reward for the <u>next move</u>. We want Q[s,a] to reflect the total accumulated reward from s and <u>all the way</u> to the final destination.

Solution:

Suppose we are in state s, policy says take action a, the reward is r, the new state is s', and policy says we should then take action a'.

$$Q[s,a] = (1-\mu)Q[s,a] + \mu\,(r + \gamma Q[s',a'])$$

Discount rate – a number $\gamma \in (0,1)$ close to 1

This learning rule is called SARSA.

SARSA actually takes into account all future steps!

Suppose the (state,action) sequence is (s,a), (s',a'), (s'',a''), ...

Updates in (s,a):

$$Q[s,a] = (1-\mu)Q[s,a] + \mu\ [r + \gamma Q[s',a']])$$

Updates in (s',a'):

$$Q[s',a'] = (1-\mu)Q[s',a'] + \mu\ [r' + \gamma Q[s'',a'']])$$

Thus (assuming μ=1 for simplicity):

$$Q[s,a] = r + \gamma r' + \gamma^2 r'' + \gamma^3 r''' + \ldots$$

(called the discounted future reward when γ<1)

Discounted future reward

The expression

$$Q[s,a] = r + \gamma r' + \gamma^2 r'' + \gamma^3 r''' + \ldots$$

tells us that rewards received later on the path to the destination are reduced (discounted). Why?

$$(s,a) \xrightarrow{r} (s',a') \xrightarrow{r'} (s'',a'') \xrightarrow{r''} (s''',a''') \xrightarrow{r'''} \quad \ldots\ldots \quad \text{GOAL}$$

Reason: there is some uncertainty in each action we take and that uncertainty accumulates along the path.

Q-learning

This is an alternative to SARSA.

In Q-learning we replace the term Q[s',a'] in SARSA with the maximal value of Q[s', :]

$$Q[s,a] = (1-\mu)Q[s,a] + \mu\,(r + \gamma \max Q[s', :])$$

If the policy used is the greedy algorithm then Q-learning and SARSA are identical.

The Q-Learning Algorithm

- **Initialisation**
  - set $Q(s,a)$ to small random values for all s and a
- Repeat:
  - initialise s
  - repeat:
    * select action a using ϵ-greedy or another policy
    * take action a and receive reward r
    * sample new state s'
    * update $Q(s,a) \leftarrow Q(s,a) + \mu(r + \gamma \max_{a'} Q(s',a') - Q(s,a))$
    * set $s \leftarrow s'$
  - For each step of the current episode
- Until there are no more episodes

The Q-Learning Algorithm

- **Initialisation**
  - set $Q(s, a)$ to small random values for all s and a
- Repeat:

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

  - initialise s
  - repeat:
    * select action a using ϵ-greedy or another policy
    * take action a and receive reward r
    * sample new state s'
    * update $\boxed{Q(s, a) \leftarrow Q(s, a) + \mu(r + \gamma \max_{a'} Q(s', a') - Q(s, a))}$
    * set $s \leftarrow s'$
  - For each step of the current episode
- Until there are no more episodes