REINFORCEMENT LEARNING

Partly from link

So far ….

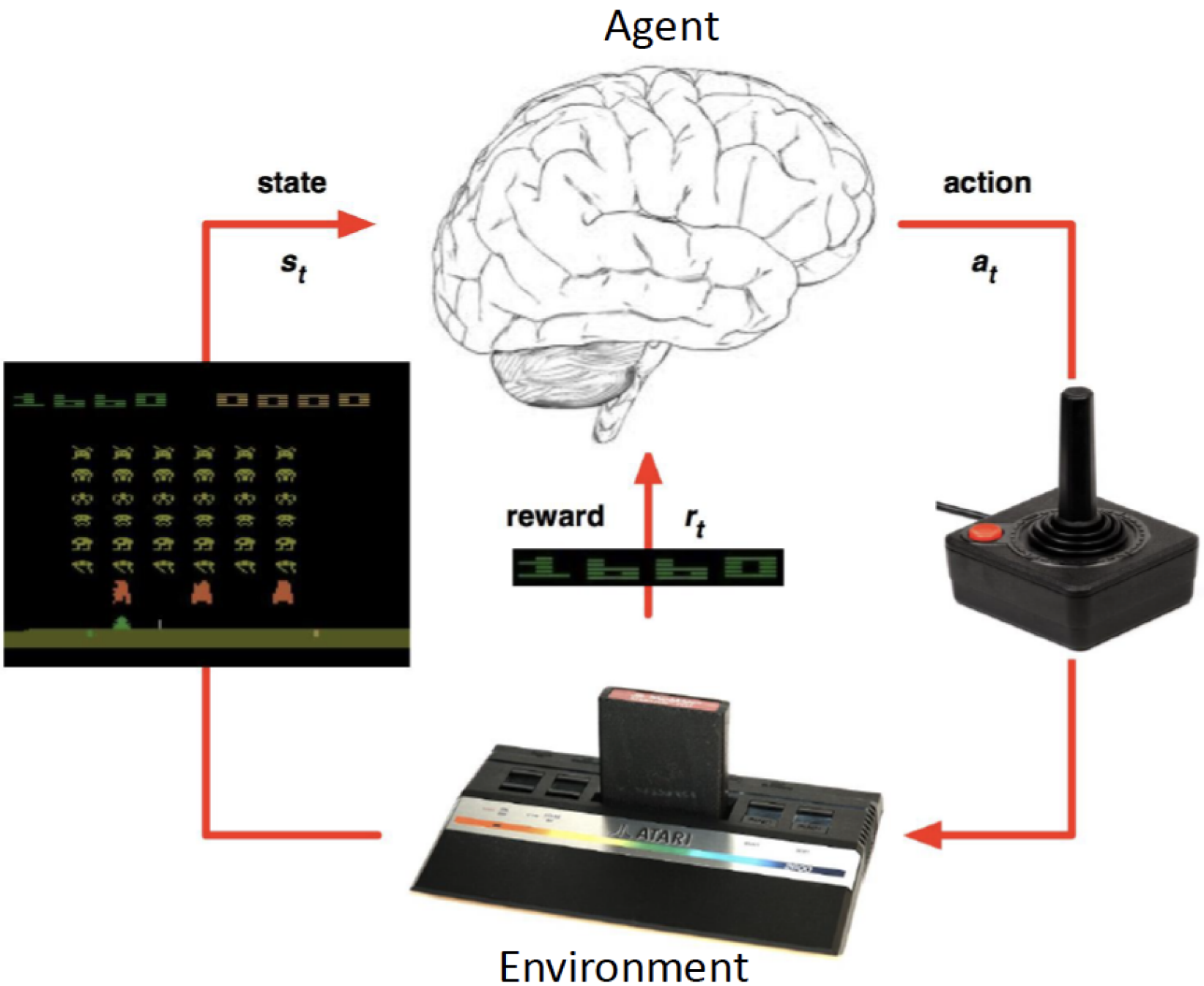
- **Supervised machine learning**: given a set of annotated instances and a set of categories, find a model to automatically categorize unseen instances (DT, SVM, NN, NaiveBayes..)
- **Unsupervised learning**: categorize or cluster instances when no examples are available. (Clustering, Denoising Autoencoders)
- Today: **Reinforcement learning:** software agents that must learn to act in an environment so as to maximize some notion of cumulative reward.
- **«Learn to act»**: Learn strategies, plans, in a virtual or physical environment (rather than learning classifications as for previous algorithms)

Overview

- In reinforcement learning: decisions we make will be about actions to take, such as a robot deciding which way to move next, which will influence what we see next
- The decisions influence the next status of the robot
- The goal will be not just to predict (say, whether there is a door in front of us or not) but to decide what to do

Reinforcement Learning

- The loop:
  - Agent perceives the state of the environment
  - Agent acts
  - Agent receives reward/punishment, the state of the environment changes
- The task:
  - Learn a sequence of action and states (e.g. a policy, mapping from states to actions) to maximize rewards

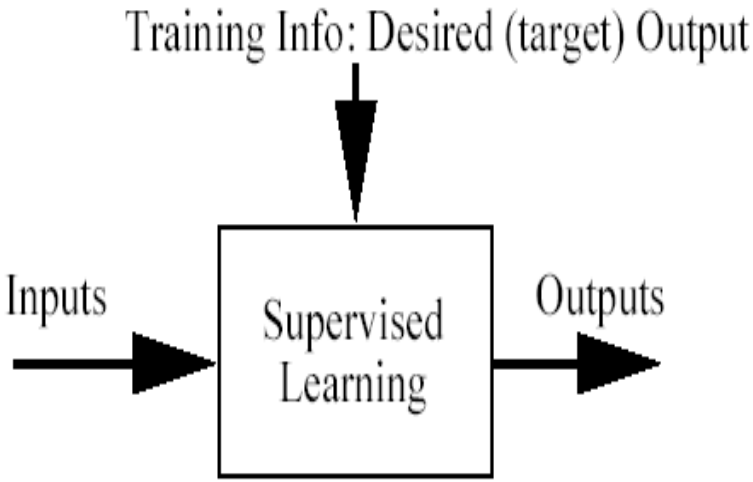

Agent
state
s_t
action
a_t
reward
r_t
ATARI
Environment

Learning by reinforcement

Characteristics:

- No direct training examples – (possibly delayed) **rewards** instead, or **penalties**
- Objective is **exploration** and **exploitation** of environment
- The environment might be deterministic, stochastic (governed by probabilities) and/or unknown
- The actions of the learner **affect future rewards**

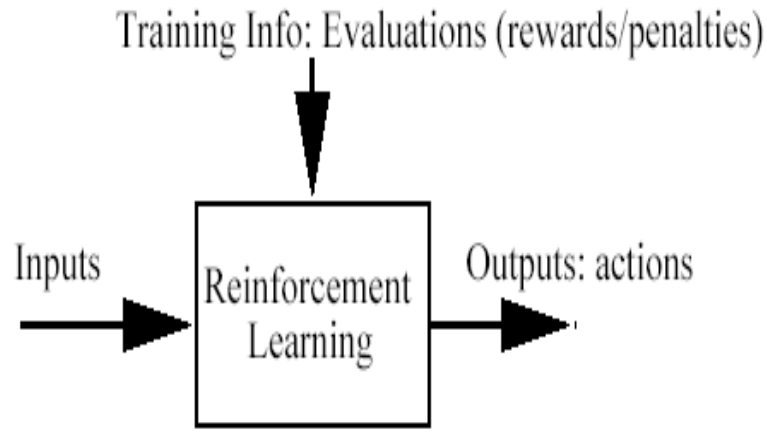
Supervised Learning



Error = (target output - actual output)

Input is an instance, output is a classification of the instance

Reinforcement Learning



Objective: Get as much reward as possible

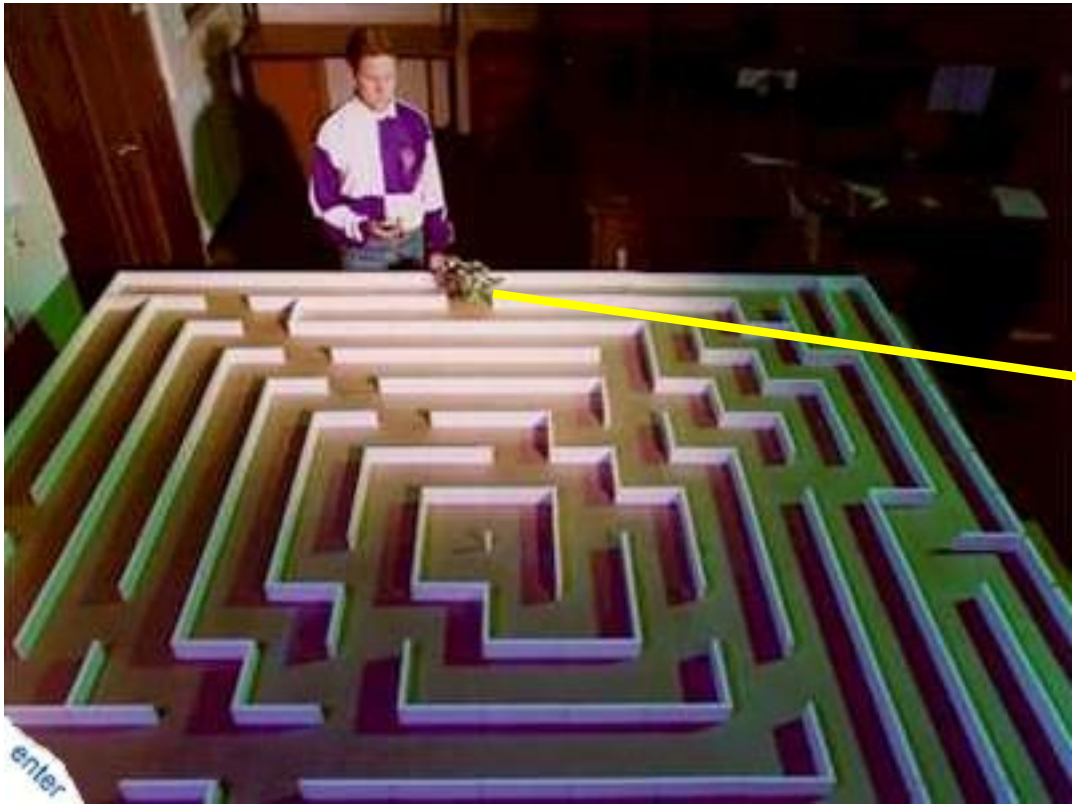
Input is some "goal", output is a sequence of actions to meet the goal

Examples of applications

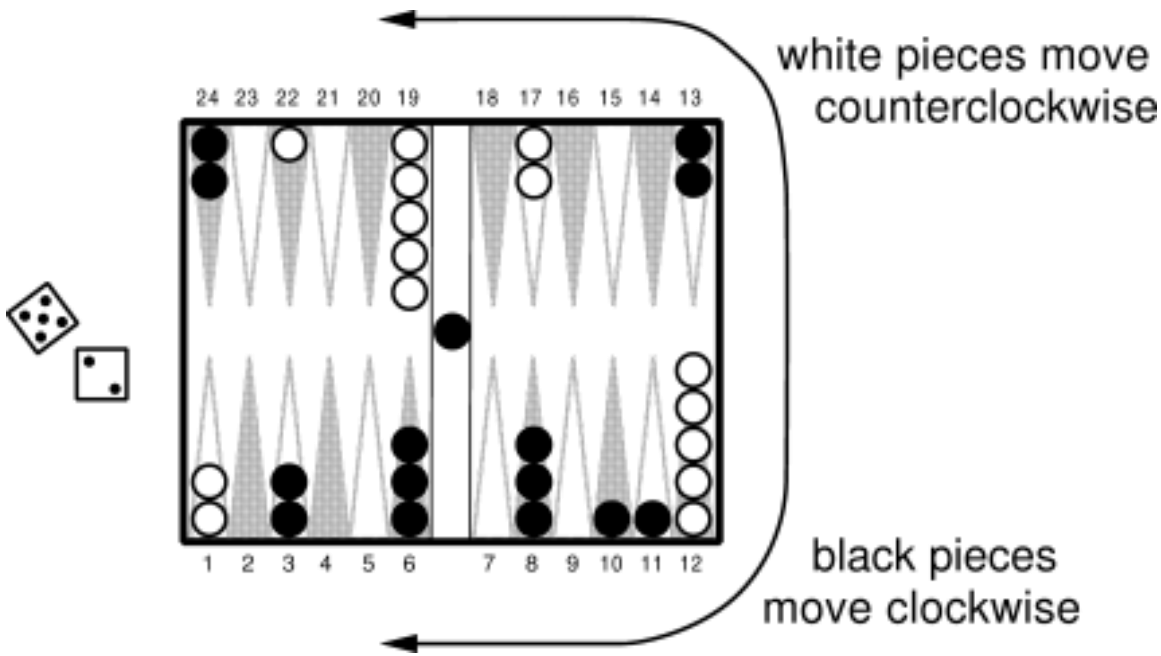
- **Control physical systems (robotics)**: walk, drive, swim, …
- **Interact with users (virtual agents)**: personal assistent to engage customers, optimise user experience, …
- **Solve logistical problems**: scheduling, bandwidth allocation, elevator control, power optimisation, ..
- **Play games (game theory)**: chess, checkers, Go, Atari games, …

Example:

Robot moving in an environment (a maze)



Example:
Playing a game (e.g. backgammon, chess)



Elements of RL

In RL, the "model" to be learned is a **policy** to meet "at best" some given goal (e.g., win a game)

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

- **Transition model:** $\boldsymbol{\delta}(s_i, \boldsymbol{a_i}) \rightarrow s_j$ how actions A influence states S
- **Reward** r$(s_i, \boldsymbol{a_i}) \rightarrow r_i$ immediate value of state-action transition (negative if penalty)
- **Policy** π: S $\rightarrow$ A, maps states to actions

Elements of RL

Agent lives in some environment E, in some state s:

- Robot: where robot is, what direction it is pointing, etc.
- Backgammon: state of the board (where all pieces are)

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

- Goal: Maximize the long term Discounted Reward (DR)
  - i.e.: want a lot of rewards, prefer getting it earlier than getting it later
  - i.e.: Learn to choose actions that maximize:

$$DR = \sum_{i=0}^{\infty} \gamma^i r_{t+i} = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots \quad where\ 0 \le \gamma \le 1$$

Markov Decision Process (MDP)

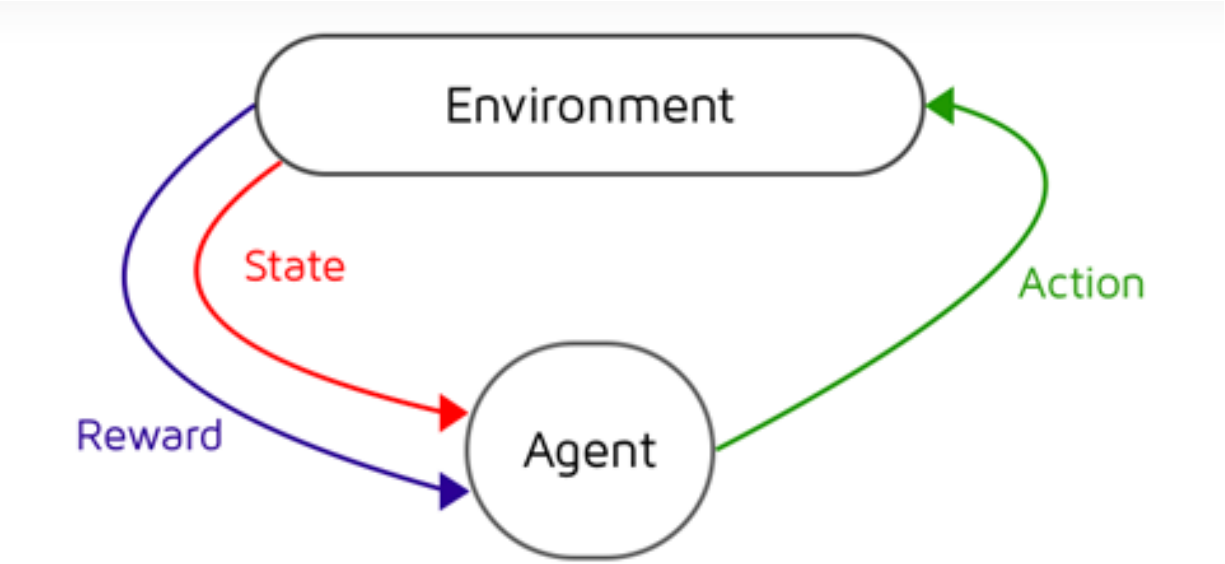
- MDP is a formal model of the RL problem. An MRP is a tuple (S, A, δ, R, γ) where:
  - S is a finite **state** space
  - A is a finite **action** space
  - δ : $(S, A) \rightarrow S$ is the **state transition** function (a matrix, called the model). $\delta(s, a) \rightarrow s'$ is the function that outputs the landing state s' of an agent that performs action **a** in state s.
  - ***r:*** $(S,A) \rightarrow R$ is a **reward** function. ***r:*** $(s,a) \rightarrow r'$ is the function that output the reward r' for an agent performing action **a** in state **s** .
  - γ is a discount factor, where $\gamma \in [0, 1]$. It informs the agent of how much it should care about rewards now, wrt rewards in the future.

The γ parameter

- Parameter $\boldsymbol{\gamma}$ has range value of 0 to 1($\mathbf{0} \leq \gamma \leq \mathbf{1}$).
- If $\boldsymbol{\gamma}$ is closer to zero, the agent will tend to consider only immediate reward.
- f $\boldsymbol{\gamma}$ is closer to one, the agent will consider future reward with greater weight, willing to delay the reward.

Markov Decision Process (MDP)

- At each discrete time point t
  - **Agent** observes state $s_t \in \mathbf{S}$ and chooses an **action** $a_t \in \boldsymbol{A}$ *(according to some probability distribution or strategy)*
  - Receives **reward** $r_t \in \boldsymbol{R}$ from the **environment** and the **state changes** to s_{t+1}



Markov Decision Process (MDP) Deterministic VS Non-Deterministic (2)

In general, the functions *r and* δ can be deterministic or stochastic.

In the **deterministic case**, the state transition function <s,a, s'> is determined (i.e. each time the agent from state **s** performs the action **a**, it transits **on the same** state **s'** every time; the same for the reward):

- δ: *(S, A)* → s' is the state transition function (a SxA matrix, called th $\boldsymbol{s'} = \boldsymbol{\delta}(a, s)$
- ***r:*** *(S,A)* → $r' \epsilon R$ is a reward function.

Markov Decision Process (MDP)

Deterministic VS Non-Deterministic

In the **non-deterministic case**, the reward function $r(s, a)$ and action transition function δ(s, a) are probabilities (P(R=r' | s,a) and P(S=s' | s,a))

In such cases, the functions $\delta(s, a)$ and $r(s, a)$ can be viewed as <u>first</u>, producing a probability distribution over possible outcomes based on s and a, and <u>then</u> drawing an outcome at random according to this distribution.

Fundamental Markov Assumption

"The future is independent of the past given the present"

Implies:

r_t and s_{t+1} depend **only on the *current* state** and action

- *Non-Deterministic MDP:*
  - $P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(s_{t+1}|s_t, a_t)$
  - $P(r_t|s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(r_t|s_t, a_t)$

- *Deterministic MDP:*

$$r_t = r(s_t, a_t)$$

$$s_{t+1} = \delta(s_t, a_t)$$

Agent's Learning Task

Execute actions in environment, observe results and learn (= learning from EXPERIENCE rather than from examples)

- The task for MDP is:

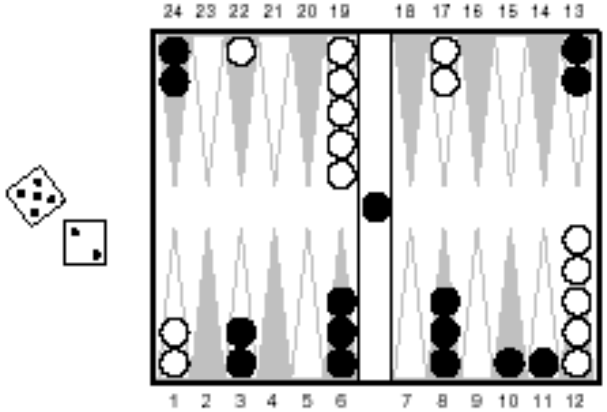
  Learn a policy π choosing actions that maximizes the **expected cumulative reward** from any starting state s_0 in S:

$$E[CR] = E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right] = E\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots\right] = \sum_{t=0}^{\infty} \gamma^i E[r_t]$$

- Note:
  - Target function is $\pi: S \rightarrow A$
  - There are **no training examples** of the form <s,a> but only of the form <<s,a>,r>, i.e. for – **some! -** state-action pair we know the reward/penalty

Example:
TD-Gammon

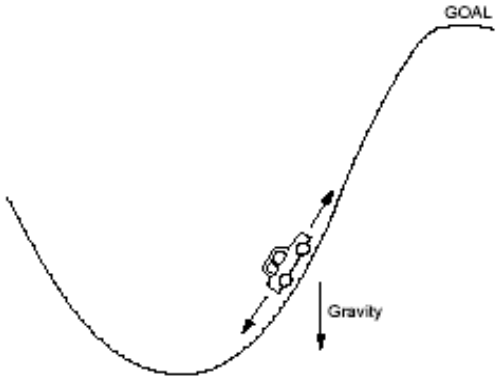
- Immediate reward:
  - +100 if win
  - -100 if lose
  - 0 for all other states



- Trained by playing 1.5 million games
- Now approximately equal to the best human player

Example:
Mountain-Car

- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- Rewards.
  - Reward = -1 for every step, until the car reaches the top
  - Reward = 1 at the top, 0 otherwise
- The possible reward will be maximized by minimizing the number of steps to the top of the hill



RL by Value function
for each policy

Given a policy π, let define $V^{\pi}(s)$, the **value** of being in state s **according to policy $\boldsymbol{\pi}$**:

$$V^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^{t} r_t]$$

(*non-deterministic case: the expected future cumulative rewards*)

$$V^{\pi}(s) = \sum_{t=0}^{\infty} \gamma^{t} r_t$$

$$(deterministic\ case: the\ future\ cumulative\ \ rewards)$$

- Assuming sequence of states and actions chosen according to π, starting at state s at time t=0
- The difference is that in the deterministic case there is no «expectation» about values, but just values!

RL by Value function

Recursive Value Function

(AKA. Bellman Equation)

- **Non-Deterministic case**:

$$V^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^t r_t]$$

π is «some» policy

Given a sequence of states and actions $[s_1, a_1, s_2, a_2, s_3, \dots]$ for any state according to the policy π :

$$V^{\pi}(s_1) = E[\gamma^0 r(s_1, a_1)] + E[\gamma^1 r(s_2, a_2)] + E[\gamma^2 r(s_3, a_3)] + \cdots =$$

$$= 1 \cdot E[r(s_1, a_1)] + \gamma E[r(s_2, a_2)] + \gamma^2 E[r(s_3, a_3)] + \cdots =$$

$$E[r(s_1, a_1)] + \gamma(E[r(s_2, a_2)] + \gamma E[r(s_3, a_3)] + \cdots) =$$

$$= E[r(s_1, a_1)] + \gamma \mathrm{E}[V^{\pi}(s_2)]$$

Recursive definition of value

$$V^{\pi}(s) = E[r(s, \pi(s))] + \gamma \mathrm{E}[V^{\pi}(s')] = \boxed{E[r(s, \pi(s))] + \sum P(s'|s, a)\, \gamma\ V^{\pi}(s')}$$

Since we have probabilities for future state s'

- **Deterministic case**: $V^{\pi}(s) = r(s, \pi(s)) + \gamma V^{\pi}(s')$

$$\boldsymbol{s' = \delta(a, s)}$$

RL by Value function
for each policy

Using the value function approach, the goal is:

$$Find\ the\ optimal\ policy\ \pi^*\ where:$$
$$\forall s, \qquad \pi^* = argmax_{\pi}(\mathrm{V}^{\pi}(\mathrm{s}))$$

Find a policy whose value function is the maximum out of all policies simultaneously for all states

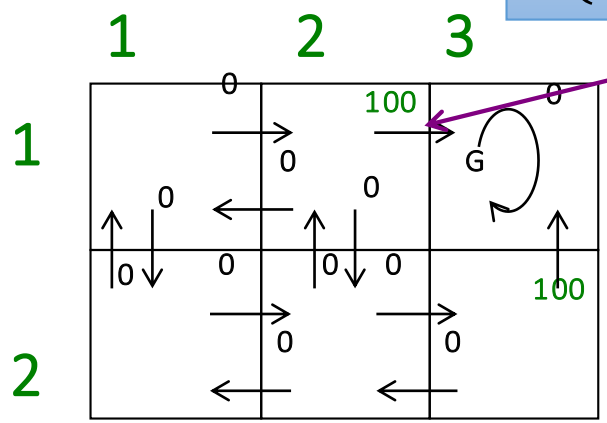
- For any MDP, such a policy exists!
- According to π^* we can compute the **values of the states** for this policy – denoted V^*:

$$\mathrm{V}^*(s) = max_{\pi}(\mathrm{V}^{\pi}(\mathrm{s}))$$

Deterministic Example:

Robot in a grid environment

$$r(\,(1,2), RIGHT) = 100$$



$r(state, action)$

Grid world environment
Six possible states
Arrows represent possible actions
G: **goal state**
Actions: UP, DOWN, LEFT, RIGHT, IDLE
Deterministic: ->immediate and fixed reward values and actions

Known environment: grid with cells
Deterministic: possible actions in any cell are known, rewards are known, immediate **(we don't have the expectation E[])**

Deterministic Example:

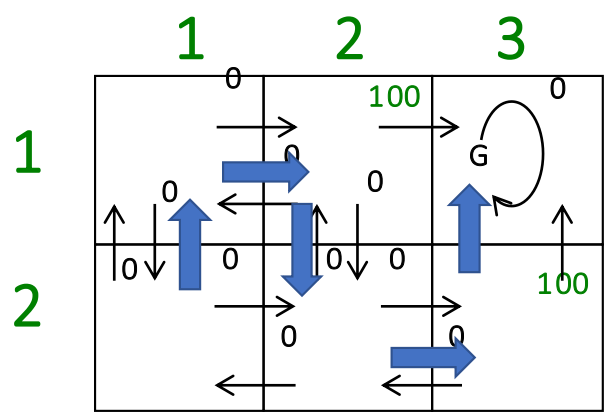
Given a policy π, what are the values $V^{\pi}(\boldsymbol{s})$?

- Suppose π is shown by blue arrows from each state
- Suppose $\boldsymbol{\gamma = 0.9}$
- Remember deterministic rewards and actions.
- Actions in each state according to the policy π:

$$[<(1,3), IDLE>, <(2,3), UP>, <(2,2), RIGHT>, <(1,2), DOWN>, <(1,1), RIGHT>, <(2,1), UP>]$$

s

a



Blue arrows here represent a policy

Deterministic Example:

Given π, what are the values $V^{\pi}(\boldsymbol{s})$?

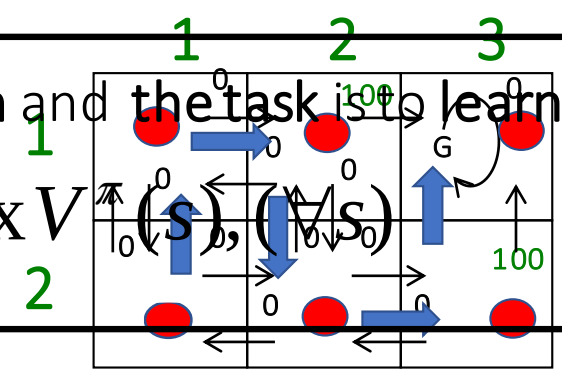
Deterministic case: $V^{\pi}(s) = r(s, \pi(s)) + \gamma V^{\pi}(s')$:

Value of being in **s** according to $\boldsymbol{\pi}$ is equal to the reward obtained when performing in **s** the action suggested by $\boldsymbol{\pi}$, plus the discounted value of the landing state s'

- $V^{\pi}((1,3)) = r((1,3),\ IDLE) = 0$
- $V^{\pi}((2,3)) = r((2,3),\ UP) + \gamma V^{\pi}((1,3)) = 100 + 0.9 \cdot 0 = 100$
- $V^{\pi}((2,2)) = r((2,2),\ RIGHT) + \gamma V^{\pi}((2,3)) = 0 + 0.9 \cdot 100 = 90$
- $V^{\pi}((1,2)) = r((1,2),\ DOWN) + \gamma V^{\pi}((2,2)) = 0 + 0.9 \cdot 90 = 81$
- $V^{\pi}((1,1)) = r((1,1), RIGHT) + \gamma V^{\pi}((1,2)) = 0 + 0.9 \cdot 81 \approx 73$
- $V^{\pi}((2,1)) = r((2,1), UP) + \gamma V^{\pi}((1,1)) = 0 + 0.9 \cdot 73 \approx 66$

- However, π^* is usually **unknown** and **the task** is to **learn the optimal policy:**

$$\pi^* = \arg\max_{\pi} V^{\pi}(s), (\forall s)$$



How do we learn an optimal policy π^* ?

$s' = \delta(a, s)$

- What evaluation function should the agent attempt to learn? One obvious choice is optimal values V^*. The agent should prefer action **a** that brings to state s_1 rather than another state s_2 whenever $V^*(s_1) > V^*(s_2)$, because the **cumulative future reward** will be greater from s_1:

$$(\textbf{Non-}\text{Deterministic})\ \pi^*(s) = argmax_a(E[r(s,a)] + \gamma E[\,V^*(s')]) = argmax_a(E[r(s,a)] + \gamma \textstyle\sum_{s'} P(s'|s,a)\ \ V^*(s'))$$

$$(\text{Deterministic})\ \pi^*(s) = argmax_a(r(s,a) + \gamma\, V^*(s'))$$

- However…:
  - learning V^* is a useful way to learn the optimal policy only when the agent has perfect knowledge of δ and r, **but when it doesn't, it can't choose actions in this way. (Note that perfect knowledge doesn't mean deterministic).**
  - *The reward and (next) state might not be known for* every possible *state-action transitions!*

RL by Value Functions
Summary

- **Value function-learning** is a form of reinforcement learning that **require** that the learner has **prior knowledge** of how its **actions** affect the environment and their **rewards**.
- An optimal policy can be still be found applying **iterative** methods, e.g. Dynamic Programming, Monte-Carlo evaluation, and Temporal-Difference learning
- **Temporal Difference** is an approach to learning how to predict a quantity that depends on future values of a given signal.
- We next introduce Q-learning, a model-free TD learning method.

What if we don't know δ and r ? Q-Learning

- We want a function that directly learns **good state-action pairs**, i.e. **what action should we take in each state**. We call this $\boldsymbol{Q(s,a)}$.

- *Note: rather than assigning a value to a state, we assign a value* **to states and actions**

- If $Q(s,a)$ is given, it is trivial to execute the optimal policy, *without knowing* $r(s,a)$ *and* $\delta(s,a)$. We have:

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

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

In other terms, the value of s is estimated as the value of the «best move» we could do from s.

Q-Function

- We define $Q(s,a)$:

$$\boldsymbol{s' = \delta(a,s)}$$

(Non-Deterministic) $Q(s,a) \equiv E[r(s,a)] + \gamma E[V^*(s')] =$

$$E[r(s,a)] + \gamma \cdot \sum_{s'} P(s'|s,a)\, \boldsymbol{V^*(s')}$$

$$= E[r(s,a)] + \gamma \cdot \sum_{s'} P(s'|s,a)\, \boldsymbol{max_{a'}(Q(s',a'))}$$

The value of being in s' according to <u>optimal</u> policy is equal to the (Q) value of performing the best possible action in s'

(Deterministic) $Q(s,a) \equiv r(s,a) + \gamma V^*(s') = r(s,a) + \gamma \cdot max_{a'}(Q(s',a'))$

The only difference is absence of expectations

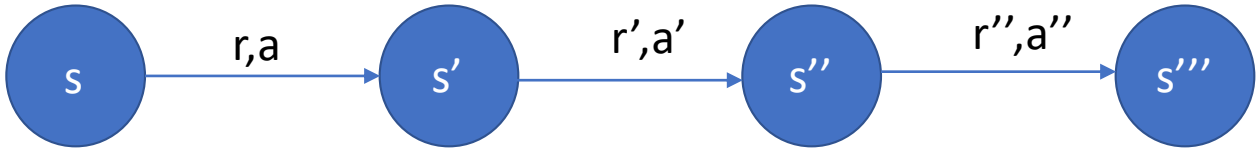
- But this still depends on $r(s,a)$ *and* δ*(s, a) (that might be unknown)*!! What's new?

Q-Function

- Imagine a robot exploring its environment, trying new actions and observing rewards "as it goes".

- At every step (being in state **s**) it observes the environment, performs an action **a**, receives some reward $\boldsymbol{r}$, and lands into a new state s'.

Can we "use" the observed sequences of states and actions to learn a model?



This can be described as «learning by doing» or learning by experience (rather by «being told» or by examples).

Deterministic Q-Learning
Q-Update Rule

- **Let consider the deterministic case for simplicity**, we can estimate our $Q - function$ **recursively, starting from some initial guess**!
- Let $\hat{Q}$ denote the learner's current approximation to Q, at each step the agent is in s, chooses an action a , moves to s' and updates its current estimate of Q(s,a):

$$\hat{Q}(s,a) \leftarrow r(s,a) + \gamma \cdot max_{a'}(\hat{Q}(s',a'))$$

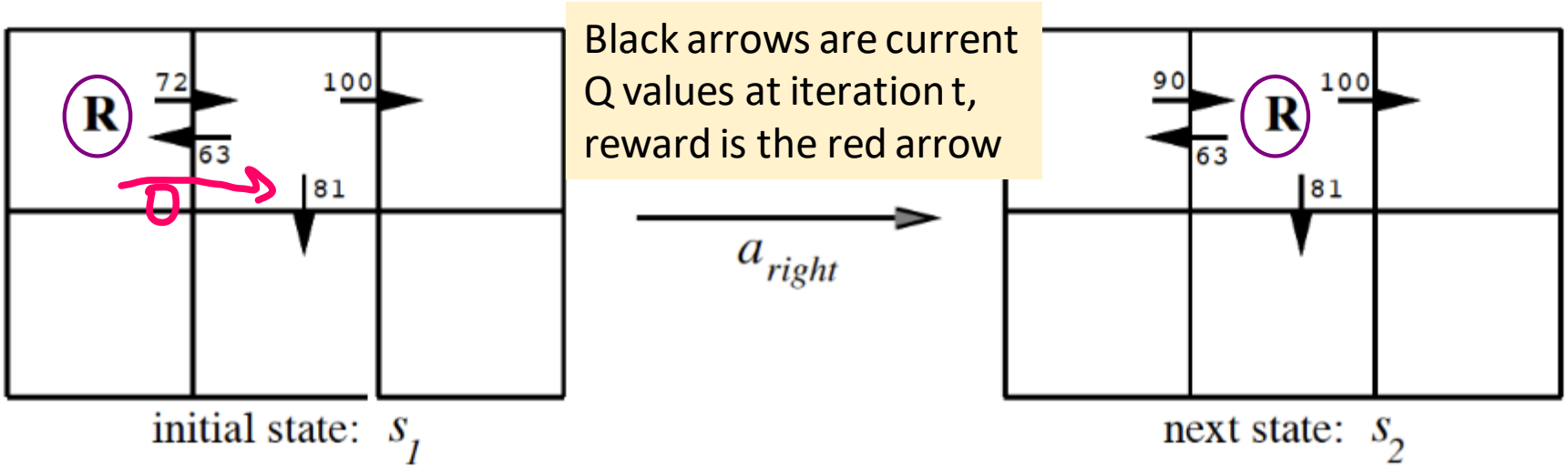
$$\boldsymbol{s'} = \boldsymbol{\delta}(a,s)$$

- This equation "continually" estimates Q at state s "consistent" with an estimate of Q at state s', i.e., one step in the future. This is called **temporal difference (TD) learning**.

- Note that, in principle, $\boldsymbol{s}$' **is "closer to goal"** w.r.t. s, and hence more "reliable", but still an estimate itself.
- Updating estimates based on other estimates is called **bootstrapping**.
- We do an update **after each state-action transition**. Under suitable conditions, these updates can be proved to **converge** (see link for the non deterministic case).

Deterministic Example:

Q-Learning updating step for robot R



Black arrows are current Q values at iteration t, reward is the red arrow

$$Q(s_1, right) \leftarrow r(s_1, right) + \gamma \cdot max_{a'}\big(Q(s_2, a')\big) =$$

$$= 0 + 0.9 \cdot \max\{63, 81, 100\} = 0 + 0.9 \cdot 100 = 0 + 90$$

Q-learning propagates Q-estimates **1-step backwards**

Q-learning algorithm (deterministic)

- **Input**: **Q matrix with all zeros**, initial state
  1. **Set** current state = initial state
  2. **Select** randomly a possible action a
  3. **Compute** next state via the delta function
  4. **From** next state, **find** action that produces maximum Q value (based on current Q table)
  5. **Update** Q
  6. **Set** current state = next state
  7. **Go to** 2 **until** (current state == goal state)

This algorithm above will **return** a **sequence** from initial state until goal state.

The algorithm is repeated for many *episodes* (new initial states) until convergence.

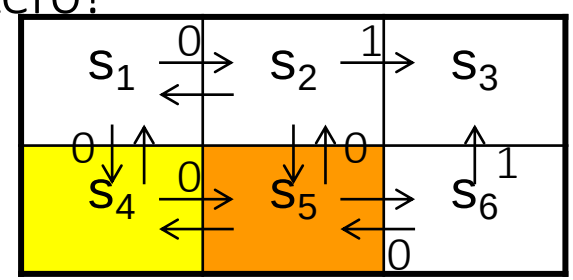
Deterministic Q-Learning Example:

γ = 0.9 - Episode 1

Consider the algorithm, in s_5 : $\forall a, \hat{Q}(s_5, a) = 0$, therefore:

$$\hat{Q}(s_4, East) \leftarrow r(s_4, East) + \gamma \cdot max_{a'} \hat{Q}(s_5, a')$$
$$= 0 + 0.9 \cdot 0 = 0$$

The "max Q" function search for the "most promising action" from s_5
However in this case nothing happens, all Q are zero!



s=s_4 possible moves: North, East
Select (at random) to move to s_5

actions

	N	S	W	E
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	0
s_5	0	0	0	0
s_6	0	0	0	0

Q-values table

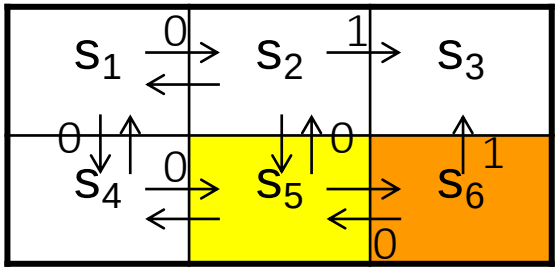
Deterministic Q-Learning Example:

γ = 0.9 - Episode 1

From s_5, moving to s_6 again does not allow rewards (all $\hat{Q}(s_6, a)$ =0)

Update: $\hat{Q}(s_5, East) \leftarrow 0 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_6, a')\right) =$

$$= 0 + 0.9 \cdot \hat{Q}(s_6, North)) = 0 + 0.9 \cdot 0 = 0$$



$S_4 \rightarrow S_5$ randomly select $\mathbf{s_6}$ as next move

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

	N	S	O	E
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	**0**
s_5	0	0	0	**0**
s_6	0	0	0	0

Deterministic Q-Learning Example:

$\gamma = 0.9$ - Episode 1

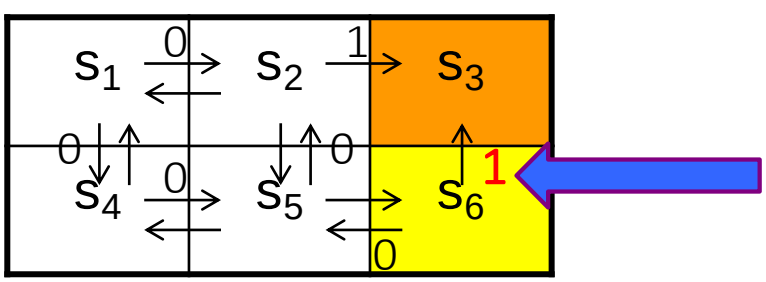
$s_4 \rightarrow s_5 \rightarrow s6$ randomly select s_3 as next move

In s_3 $r = 1$ so $\hat{Q}$ is updated to 1

$$\hat{Q}(s_6, North) \leftarrow 1 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_3, a')\right)$$
$$= 1 + 0.9 \cdot 0 = 1$$

- GOAL STATE REACHED:

  END OF FIRST EPISODE



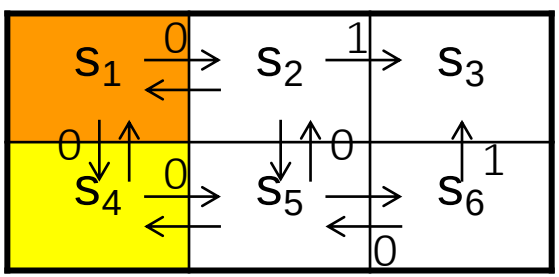
	N	S	O	E
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	0	0	0	**0**
s_5	0	0	0	**0**
s_6	1+0.9*0 =1	0	0	0

Deterministic Q-Learning Example:

γ = 0.9 - Episode 2

$s = s_4,\ s' = s_1$

$$\hat{Q}(s_4, North) \leftarrow$$
$$= 0 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_1, a')\right)$$
$$= 0$$



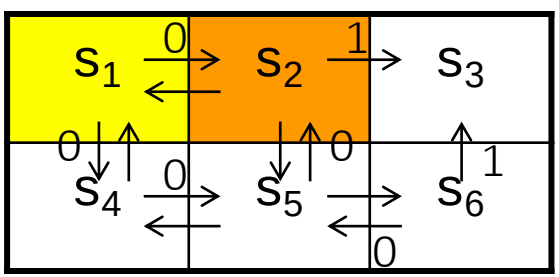
	N	S	O	E
s_1	0	0	0	0
s_2	0	0	0	0
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	0	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

γ = 0.9 - Episode 2

$$s_4 \rightarrow s_1, choose\ s' = s_2$$

$$\hat{Q}(s_1, East) \leftarrow$$
$$= 0 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_2, a')\right)$$
$$= 0$$



	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	0
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	0	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

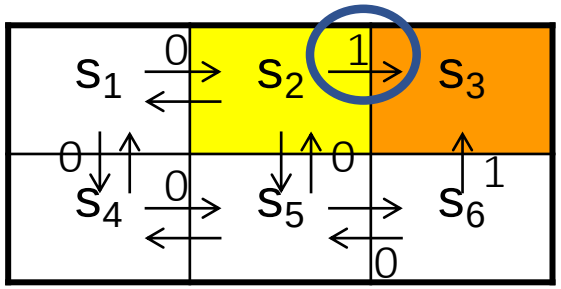
γ = 0.9 - Episode 2

$$s_4 \rightarrow s_1 \rightarrow s_2, choose\ s' = s_3$$

$$\hat{Q}(s_2, East) \leftarrow$$
$$= 1 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_3, a')\right)$$
$$= 1$$

- GOAL STATE REACHED:

  END OF 2nd EPISODE



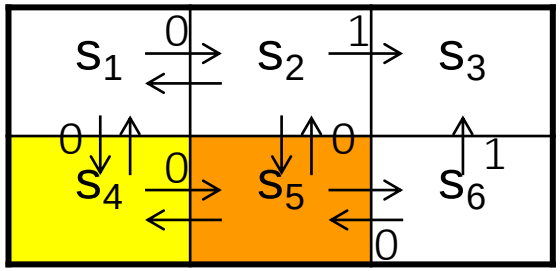
	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	1
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	0	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

$\gamma = 0.9$ - Episode 3

$s = s_4,\ s' = s_5$

$$\hat{Q}(s_4, East) \leftarrow 0 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_5, a')\right) = 0$$



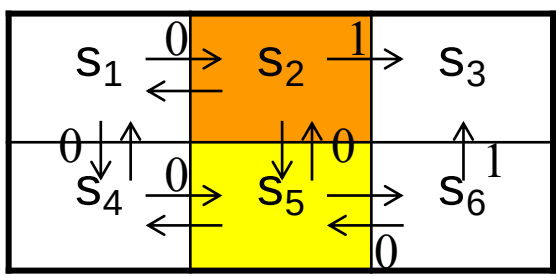
	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0+0**
s_5	0	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

γ = 0.9 - Episode 3

$s = s_5,\ s' = s_2$

$$\hat{Q}(s_5, North) \leftarrow 0 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_2, a')\right) = 0 + 0.9 \cdot 1 = 0.9$$



	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	0,9	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

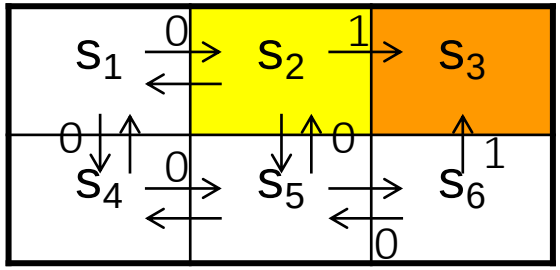
γ = 0.9 - Episode 3

$s=s_2,\ s'=s_3$

$$\hat{Q}(s_2, East) \leftarrow 1 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_3, a')\right) = 1$$

- GOAL STATE REACHED:

  END OF 3rd EPISODE



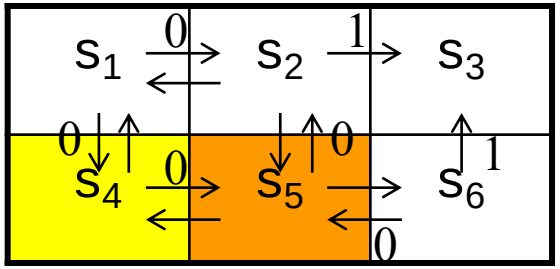
	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1+0**
s_3	0	0	0	0
s_4	**0**	0	0	**0**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

γ = 0.9 - Episode 4

$s = s_4,\ s' = s_5$

$$\hat{Q}(s_4, East) \leftarrow 0 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_5, a')\right) = 0.9 \cdot 0.9 = 0.81$$



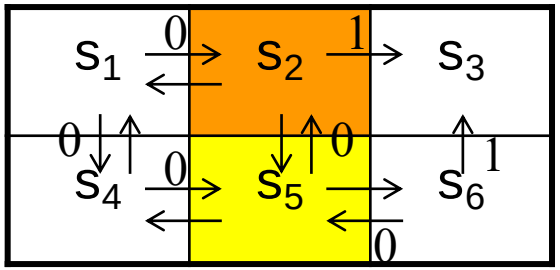
	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

γ = 0.9 - Episode 4

$s = s_5,\ s' = s_2$

$$\hat{Q}(s_5, North) = 0 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_5, a')\right) = 0.9 \cdot 1 = 0.9$$



	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0,9**	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

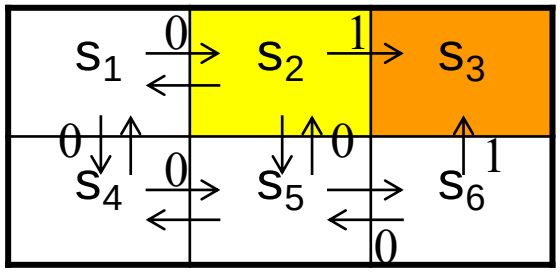
γ = 0.9 - Episode 4

$s = s_2,\ s' = s_3$

$$\hat{Q}(s_2, East) \leftarrow 1 + 0.9 \cdot max_{a'}\left(\hat{Q}(s_3, a')\right) = 1 + 0.9 \cdot 0 = 1$$

- GOAL REACHED:

  END OF 4th EPISODE

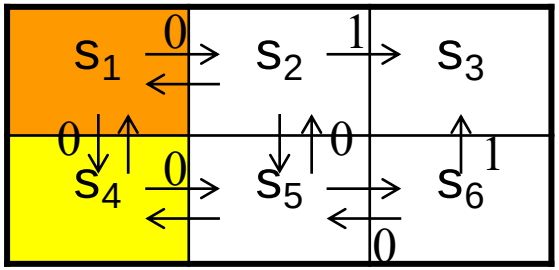


	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1+0**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

$\gamma = 0.9$ - Episode 5

$s = s_4,\ s' = s_1$

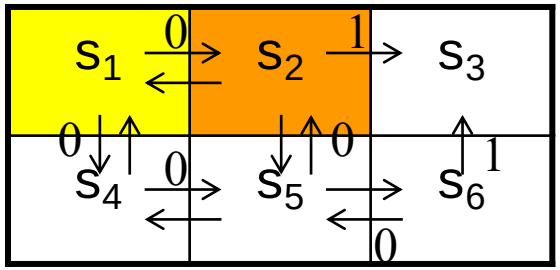


	N	S	O	E
s_1	0	0	0	**0**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0+0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

$\gamma = 0.9$ - Episode 5

$s = s_1,\ s' = s_2$



	N	S	O	E
s_1	0	0	0	**0.9**
s_2	0	0	0	**1**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

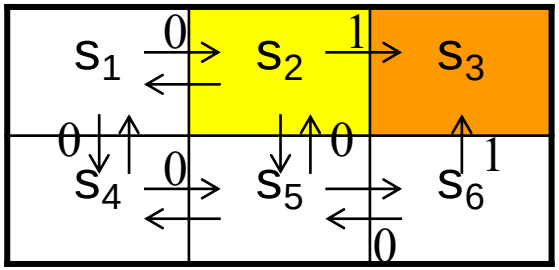
Deterministic Q-Learning Example:

$\gamma = 0.9$ - Episode 5

$s = s_2,\ s' = s_3$

- GOAL REACHED:

  END OF 5th EPISODE

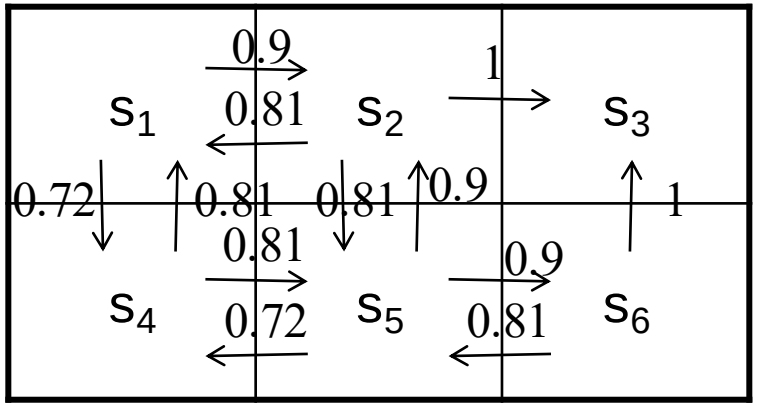


	N	S	O	E
s_1	0	0	0	**0.9**
s_2	0	0	0	**1+0**
s_3	0	0	0	0
s_4	**0**	0	0	**0.81**
s_5	**0.9**	0	0	**0**
s_6	**1**	0	0	0

Deterministic Q-Learning Example:

$\gamma = 0.9$ - Convergence

After several iterations, the algorithm converges to the following table:



	N	S	O	E
s_1	0	**0.72**	0	**0.9**
s_2	0	**0.81**	**0.81**	**1**
s_3	0	0	0	0
s_4	**0.81**	0	0	**0.81**
s_5	**0.9**	0	**0.72**	**0.9**
s_6	**1**	0	**0.81**	0

Non-Deterministic Q-Learning Q-Update Rule

How is the **Q-updating rule** modified for the non-deterministic case? One common formulation is (link):

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r(s,a) + \gamma \cdot max_{a'}\left(\hat{Q}_{n-1}(s',a')\right)]$$

- Where α_n , the learning rate, is:

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

With probability $(1-\alpha_n)$ system stays in current state and gets no reward, with probability α_n it makes a move

visits$_n$(**s,a**) is the total # of times **s**tate-**a**ction pair has been visited upto, and including the **n**th iteration. As n grows, α_n gets smaller thus reducing the updates

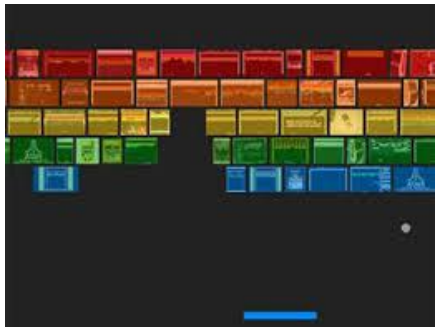
- Rather than replacing the old estimate with the new estimate, we compute a **weighted average of them**: $(1-\alpha_n)$ times your old estimate, plus α_n times your new estimate. This way you average out the *probabilistic fluctuations*, and one can show that it still converges .

RL by Q-functions
Summary

- **Q-learning** is a form of reinforcement learning that **doesn't require** that the learner has **prior knowledge** of how its **actions** affect the environment and their **rewards.**
- An optimal policy can be found applying the **temporal-difference method.**
- Learning occurs by letting the agent randomly move around, and **Q table is iteratively updated until convergence**. Optimal policy is then learned.
- Q values updating rule varies for the deterministic and non-deterministic case.

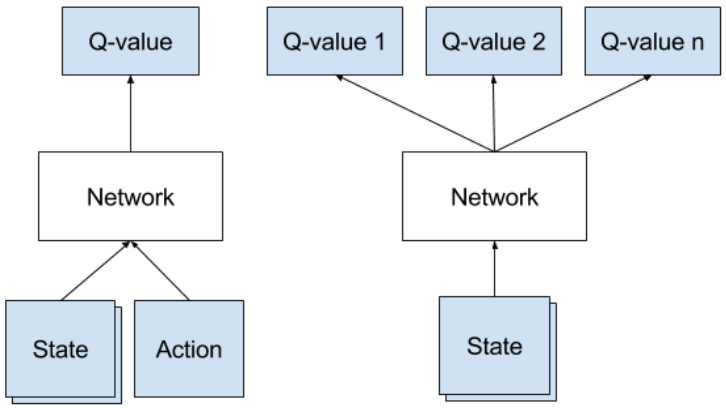
Deep Q-Learning

- In many practical applications, e.g., games, the dimension of a Q-table is **very large**
- For example, consider Atari games: The state of the environment in the Breakout game can be defined by the location of the paddle, location and direction of the ball and the existence of each individual brick.
- If apply a pixel-level processing of board– e.g., take the four last screen images, resize them to $84 \cdot 84$ and convert to gray-scale with 256 gray levels – we would have $25684 \cdot 84 \cdot 4 \approx$ **1067970 possible game states**. This means 1067970 rows in our imaginary $Q - table$ – that is more than the number of atoms in the known universe!



Deep Q-Learning

- The intuition is that, in order to learn Q values, we can use deep **neural networks**. We train for **some** state and action pairs (s,a), and we can then use the trained network to compute Q for any state and action. We can use two alternative architectures:

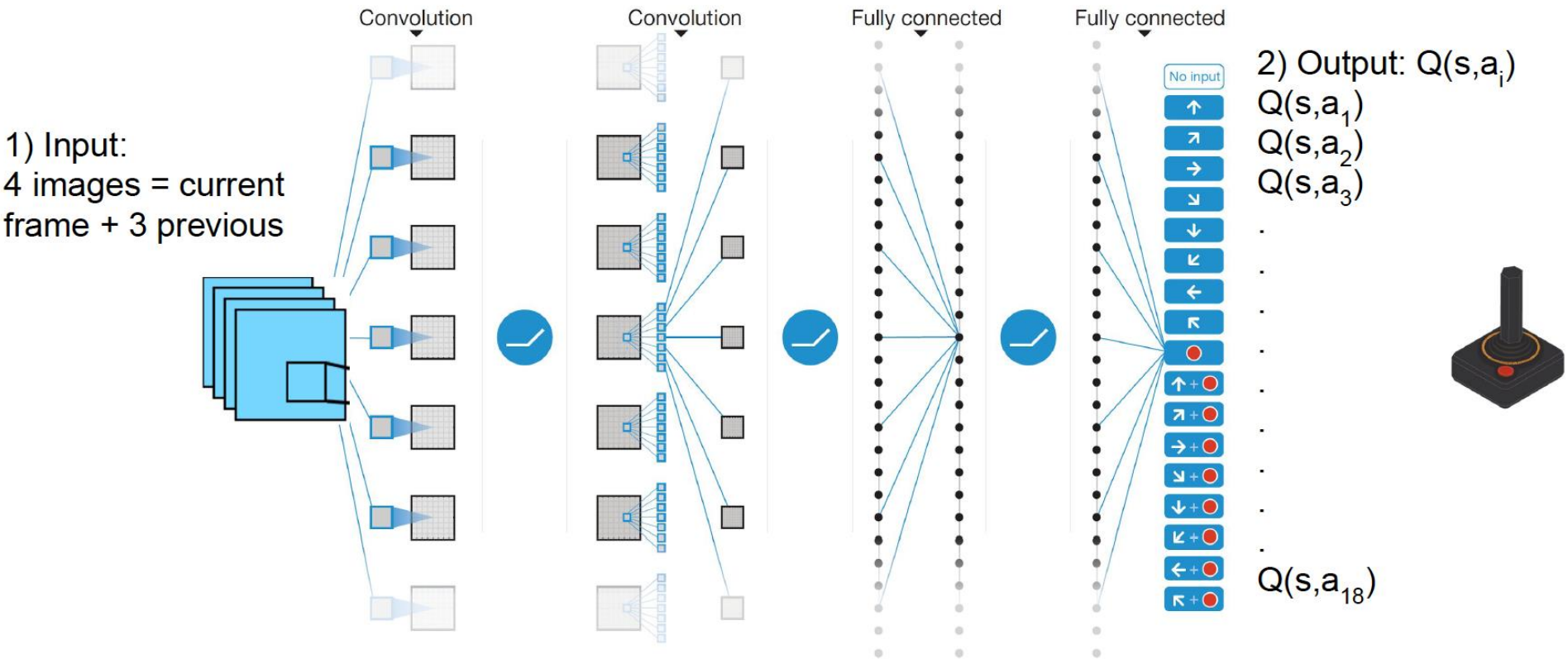


- In the left-hand side formulation, input is a (possibly multi-dimensional) representation of a state s and action a, output is the Q-value of (s, a) (e.g. $Q(s, a)$)

- In the right formulation, input is a state s, output are the Q-values **for all possible actions $\boldsymbol{a_1}$, $\boldsymbol{a_2}$,…**

Example:

Deep Mind network for Atari games (Mnih 2013)



Input to the network are four $84 \cdot 84$ gray-scale game screens (a state).
Outputs of the network are Q-values for each possible action (18 in Atari).

Example:

Deep Mind network for Atari games (Mnih 2013)

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

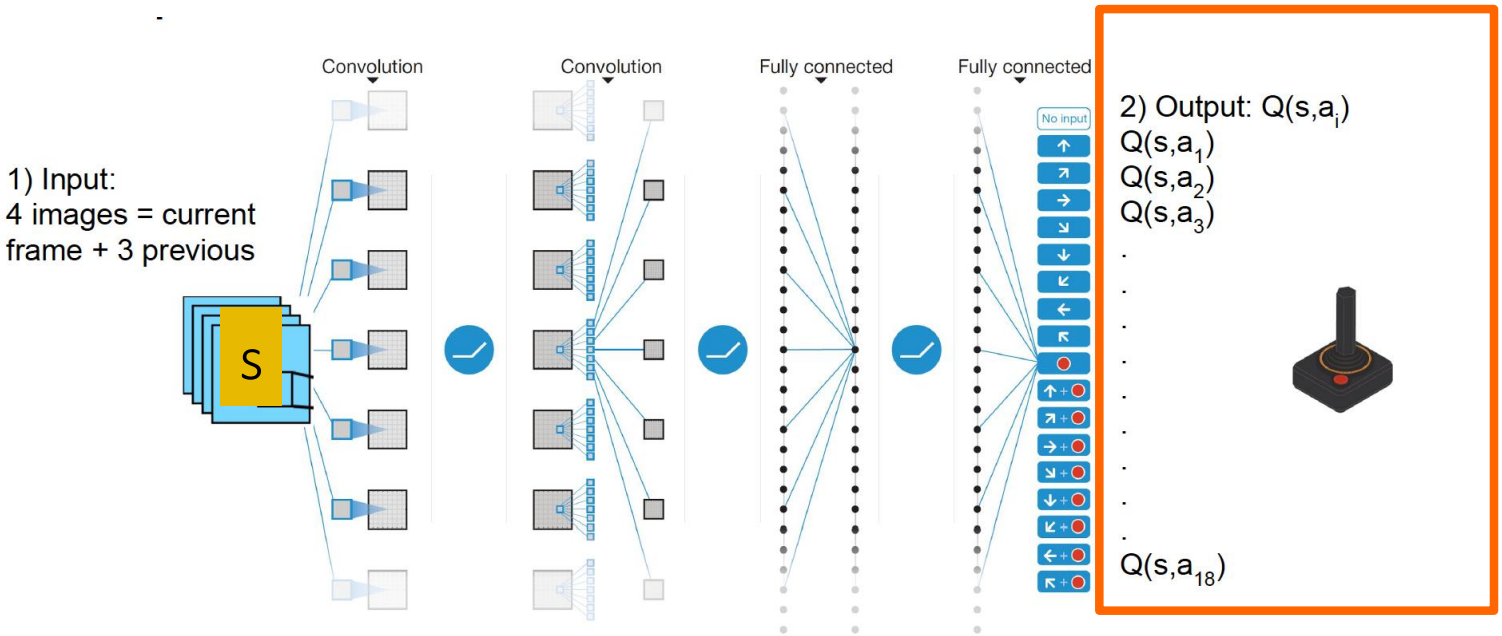
- Notice that there are **no pooling layers**!
- **Pooling layers allow for translation invariance** – the network becomes insensitive to the location of an object in the image.
- That makes perfectly sense for an image classification task, but for games the location of objects (e.g., the ball) is crucial in determining the potential reward and we wouldn't want to discard this information!

Deep Q-Learning:

How does it works?

Given a transition $< \boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s}' >$, the Q-table update rule in the "classic" algorithm must be replaced with the following steps (here is how to estimate $Q(s, a)$ for a given s and a):

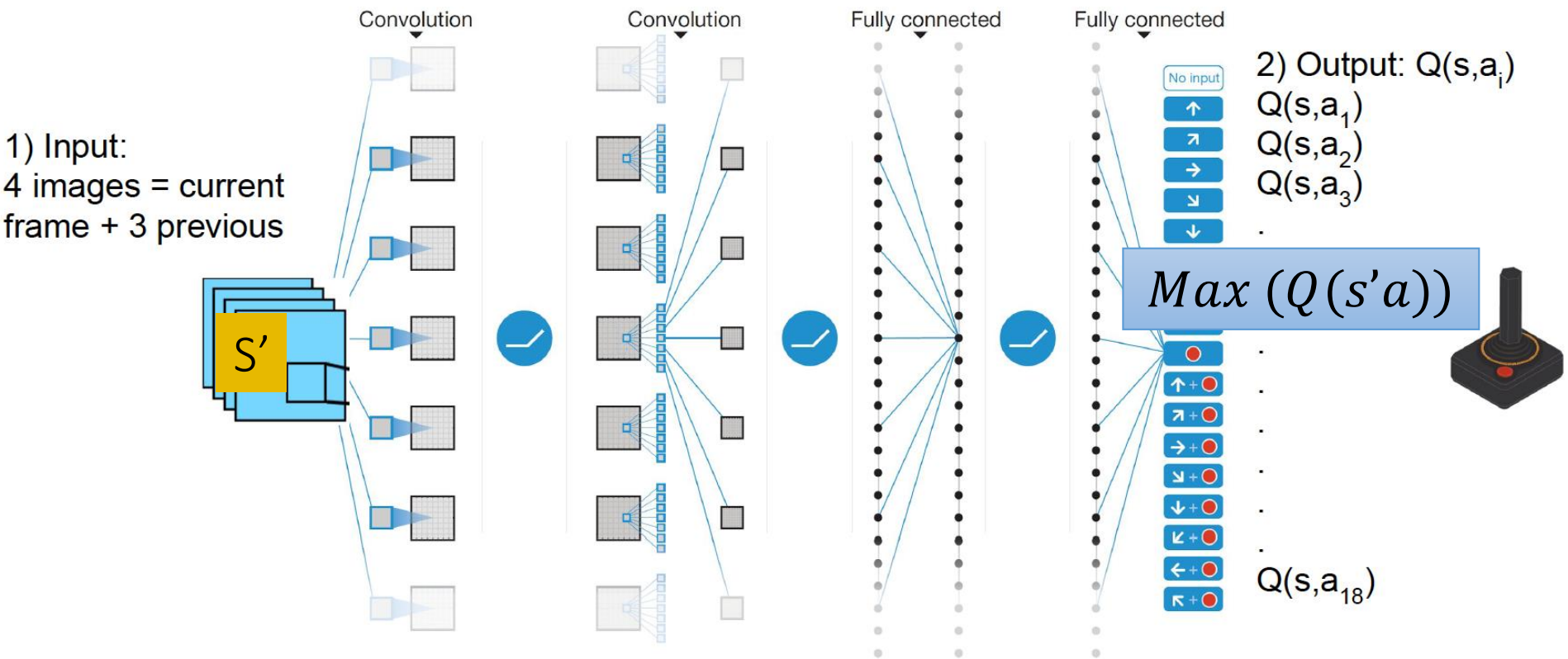
1. Do a feedforward pass on the Deep Neural Network for current state s and get predictions for $Q(s, a)$, for any possible action a:



Deep Q-Learning :
How does it works?

2. Do another feedforward pass **for the next (observed) state** s' given **a**, and calculate maximum over all network outputs $max_{a'}Q(s', a')$

-



Deep Q-Learning :
How does it works?

3. Set **Q-value** "target" (ground truth) for $Q(s,a)$ in step 1 to: $[r + \gamma \cdot max_{a'} Q(s',a')]$ (use the max calculated in step 2)

4. For all other actions in step 1, set the Q-value target to the same as originally returned from step 1, **making the error 0 for those outputs**.

- Therefore after step 4 we only have one error on the prediction for $Q(s,a)$

Deep Q-Learning:
How does it works?

4. To compute the error on $Q(s,a)$, use the standard loss (error) function:

Reward Decay Rate

$$loss = \Big(\underbrace{\mathrm{r} + \gamma \max_{a`} \hat{Q}(s,a`)}_{\text{Target}} - \underbrace{Q(s,a)}_{\text{Prediction}}\Big)^2$$

5. Use **gradient descent with back-propagation** to update network weights and minimize error.

Note we learn one Q(s,a) at a time.

More issues

- We have shown how to estimate the future reward in each state using Q-learning and approximate the Q-function using a convolutional neural network.
- But it turns out that approximation of Q-values using non-linear functions (such as NNs) is **not very stable** and **very slow**, even with conv-nets.

More issues

Several "**tricks**" can be used to **speed the convergence**:

- Most important is **experience replay**. During gameplay all the experiences $< s, a, r, s' >$ are stored in a "replay" memory. When training the network, random samples from the replay memory are used instead of the most recent transition (in other words, we don't follow the sequence of moves of a player).
- This **breaks the similarity of subsequent training samples**, which otherwise might drive the network into a local minimum.

Deep Q-Learning: Summary

- **Deep Q-learning** is a form of Q-Learning that employ NNs to handle problems with a **huge state space**.
- Readings: LINK

RL Summary

- Reinforcement learning is suitable for learning in ***uncertain*** **environments** where **rewards** may be ***delayed*** and subject to chance
- The goal of a reinforcement learning program is to maximise the ***eventual*** reward

Recommended Readings

- Tom Mitchell course: LINK, LINK, LINK, LINK
- RL: LINK, LINK, LINK