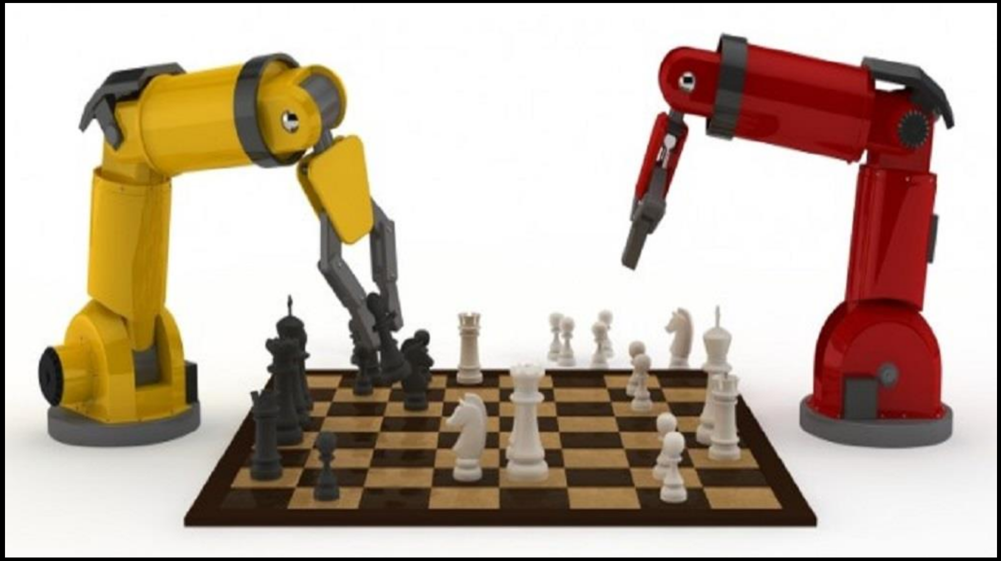
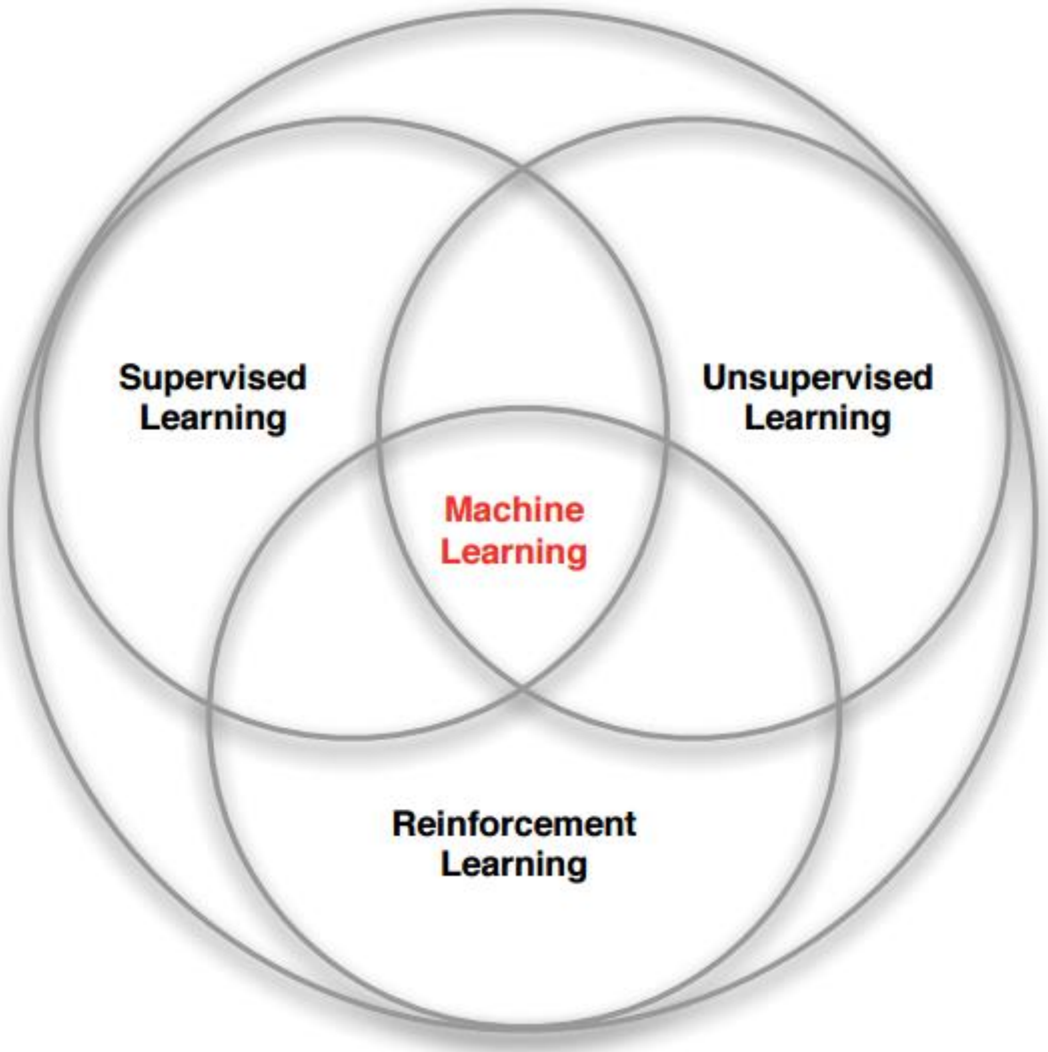
Introduction to Reinforcement Learning

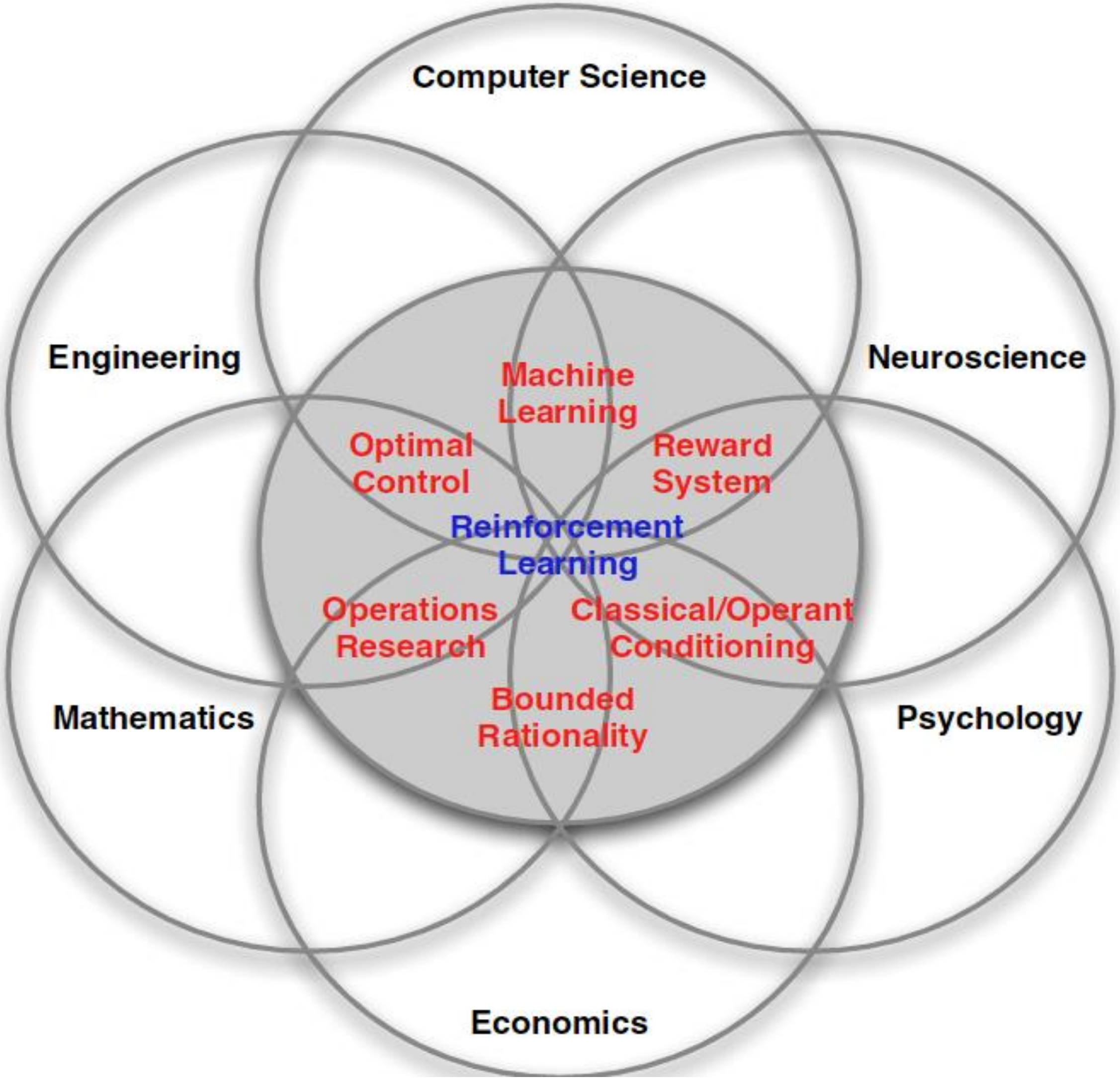


Alexey Gruzdev
alexey.gruzdev@intel.com
CV Camp, July 2019

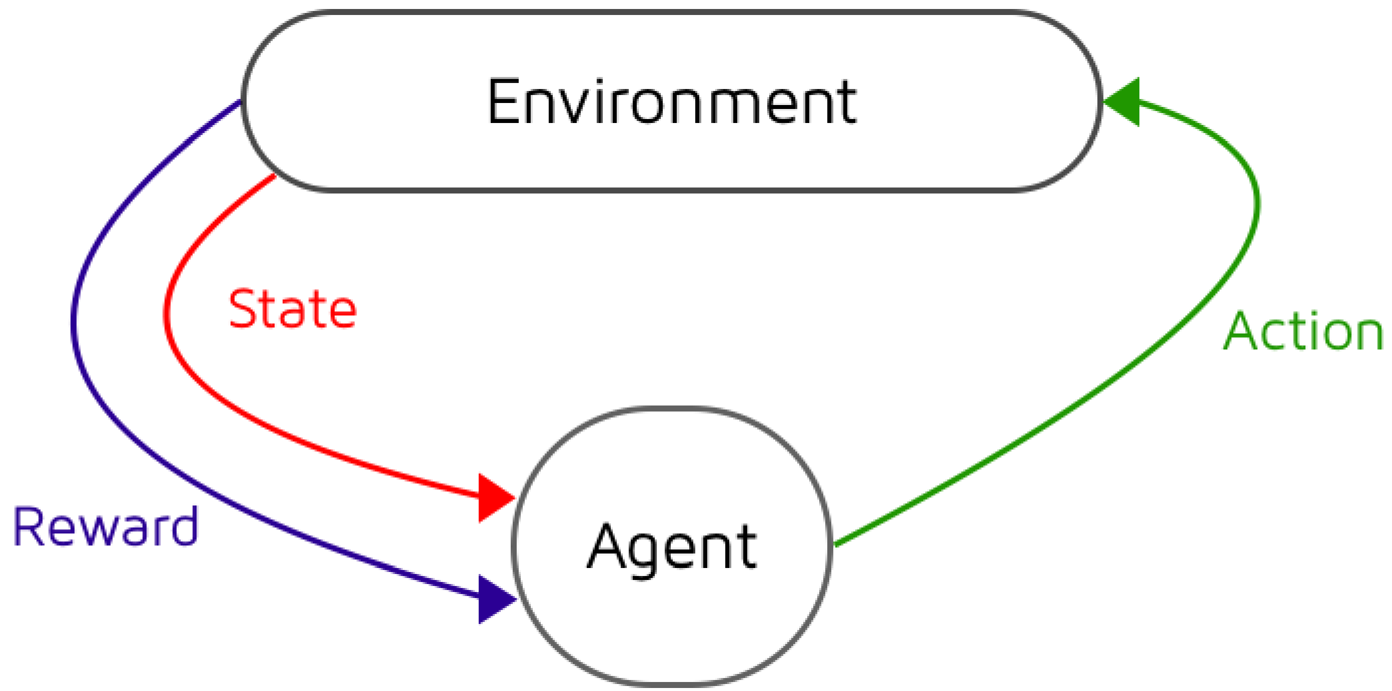
Branches of Machine Learning



Many faces of Reinforcement Learning



RL Mechanics



Reinforcement Learning Peculiarities

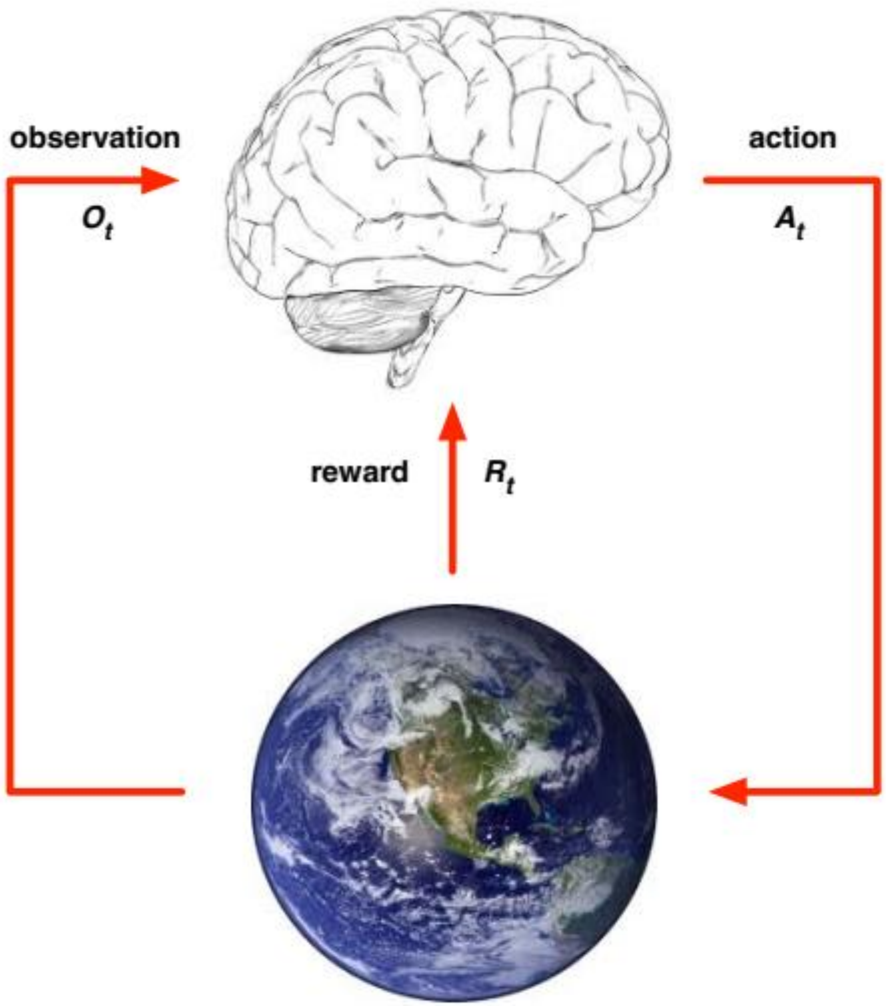
- What makes RL domain different from other machine learning approaches?

  - There is **no** external supervisor, only *reward* signal
  - Feedback can be delayed – not right now!
  - Time matters – sequential, not i.i.d data
  - Agent's actions affect future data it receives

Sequential Decision Making

- Goal: *select actions to maximize total future reward*
- Actions may have long term consequences
- Reward may be delayed
- It may be better to sacrifice immediate reward to gain more long-term reward
  - A financial investment (may take months to mature)
  - Refueling a helicopter (might prevent a crash in several hours)
  - Blocking opponent moves (might help winning chances many moves from now)

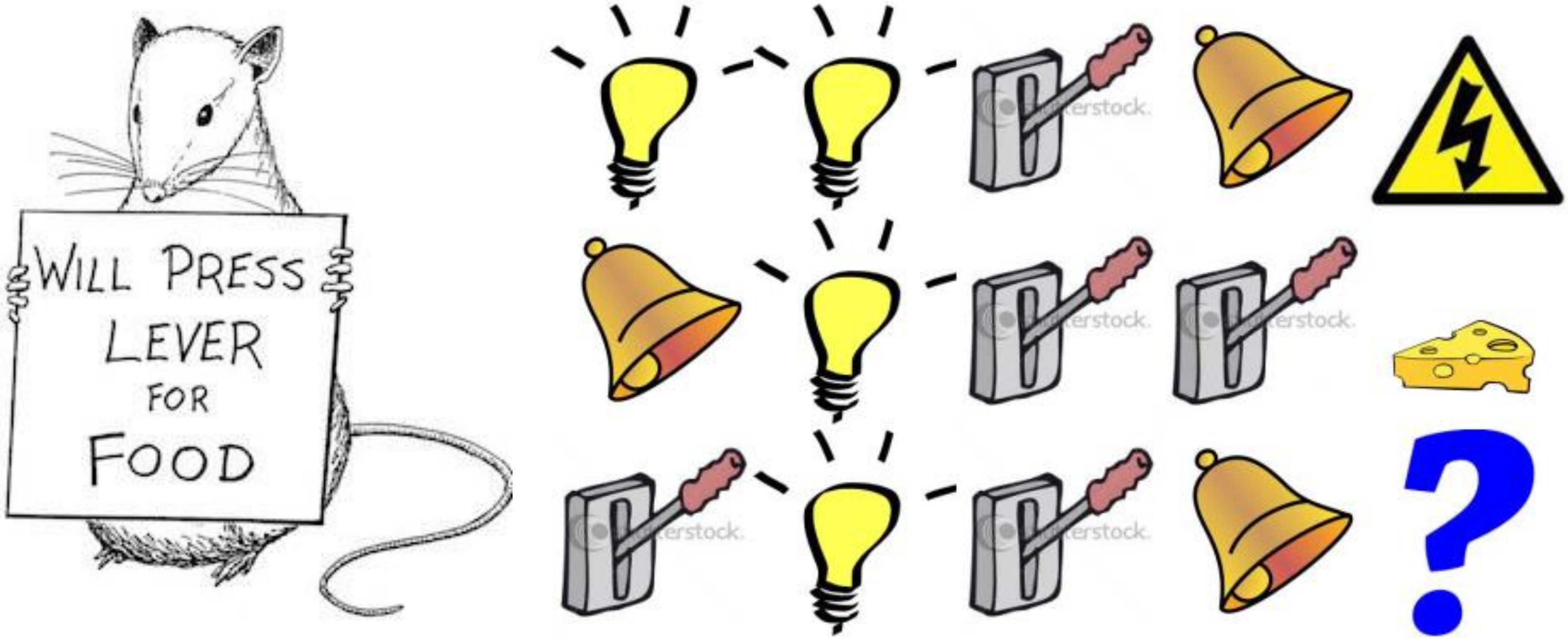
RL Basics



An RL agent may include one or more of these components:

- **Policy:** agent's behavior function
- **Value function:** how good is each state and/or action
- **Model:** agent's representation of the environment

Rat Example



- What if agent state = last 3 items in sequence?
- What if agent state = counts for lights, bells ?
- What if agent state = complete sequence?

Major Components of an RL Agent

- An RL agent may include one or more of these components:
  - Policy: agent's behavior function
  - Value function: how good is each state and/or action
  - Model: agent's representation of the environment

Policy

- A policy is the agent's behavior
- It is a map from state to action, e.g.
- Deterministic policy: $a = \pi(s)$
- Stochastic policy: $\pi(a \mid s) = \mathrm{P}[A_t = a \mid S_t = s]$

Value Function

- Value function is a prediction of future reward
- Used to evaluate the goodness/badness of states
- And therefore to select between actions, e.g.

$$v_\pi(s) = \mathrm{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s]$$

Model

- A model predicts what the environment will do next

- $\boldsymbol{P}$ predicts the next state
- $\boldsymbol{R}$ predicts the next (immediate) reward, e.g.

$$\boldsymbol{P}_{ss'}{}^{a} = \mathrm{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$

$$\boldsymbol{R}_s{}^{a} = \mathrm{E}\,[R_{t+1} \mid S_t = s, A_t = a]$$

Rewards hypothesis revisited

- A reward R_t is a scalar feedback signal
- Indicates how well agent is doing at step t
- The agent's job is to maximize cumulative reward

Reinforcement Learning is based on the reward hypothesis.

<u>The reward hypothesis:</u>

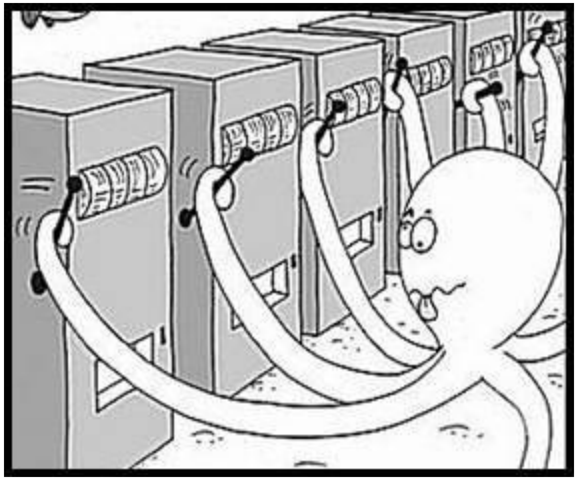
All goals can be described by the maximization of expected cumulative reward .

Exploration & Exploitation

- Reinforcement learning is like *trial-and-error* learning
- The agent should discover a good policy
- From its experiences of the environment
- Without losing too much reward along the way

Exploration & Exploitation

- *Exploration* finds more information about the environment
- *Exploitation* exploits known information to maximize reward
- It is usually important to explore as well as exploit



Examples:

- Bar Selection
  - Exploitation: Go to your favorite bar
  - Exploration: Try a new bar
- Online Banner Advertisements
  - Exploitation: Show the most successful advert
  - Exploration: Show a different advert
- Oil Drilling
  - Exploitation: Drill at the best known location
  - Exploration: Drill at a new location
- Game Playing
  - Exploitation: Play the move you believe is best
  - Exploration: Play an experimental move

Markov Processes Family

- Markov Processes (Markov Chain)
- Markov Reward Processes
- Markov Decision Processes
- Extensions to MDPs:
  - Infinite & Continuous MDP
  - POMDP
  - Undiscounted MDP

Introduction to MDPs

- *Markov decision processes* formally describe an environment for reinforcement learning
- Where the environment is *fully observable*
- i.e. The current *state* completely characterizes the process
- Almost all RL problems can be formalized as MDPs, e.g.
  - Optimal control primarily deals with continuous MDPs
  - Partially observable problems can be converted into MDPs
  - Bandits are MDPs with one state

Markov Property

- Definition: a state S_t is Markov if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \ldots, S_t]$$

- "The future is independent of the past given the present"
  - The state captures all relevant information from the history
  - Once the state is known, the history may be thrown away
  - i.e. The state is a sufficient statistic of the future

State Transition Matrix

- For a Markov state *s* and successor state *so*, the *state transition probability* is defined by

$$\boldsymbol{P}_{ss'} = \mathrm{P}\left[S_{t+1} = s' \mid S_t = s\right]$$

- State transition matrix $\boldsymbol{P}_{ss'}$ defines transition probabilities from all states *s* to all successor states *s'*

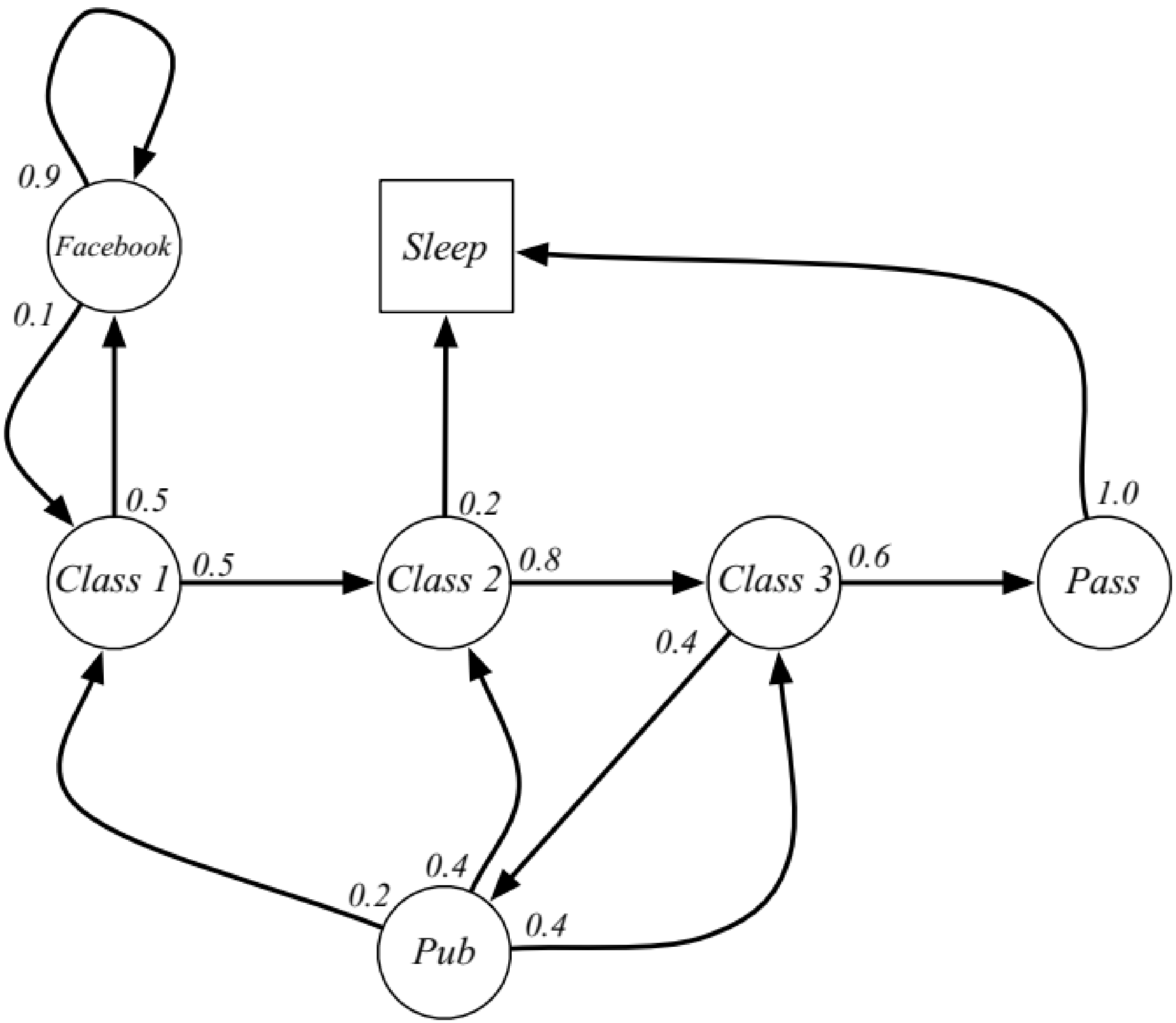
$$\boldsymbol{P}_{ss'} = \begin{pmatrix} \boldsymbol{P}_{11} & \cdots & \boldsymbol{P}_{1n} \\ \vdots & \ddots & \vdots \\ \boldsymbol{P}_{n1} & \cdots & \boldsymbol{P}_{nn} \end{pmatrix}$$

Markov Process

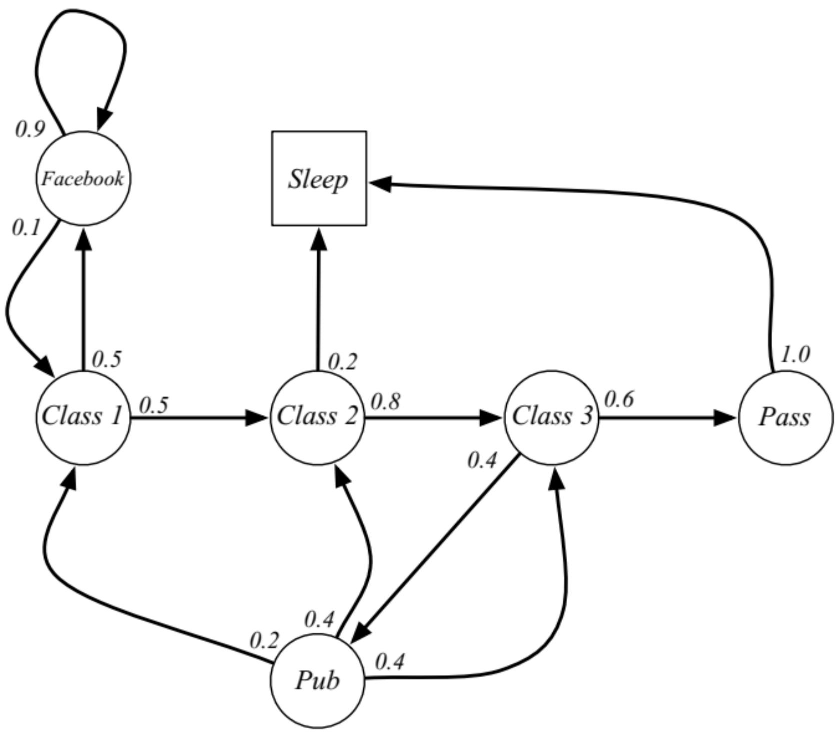
- Markov process is a memoryless random process, i.e. a sequence of random states $\boldsymbol{S_1, \ldots, S_t}$ with the Markov property.

- *A Markov Process* (or *Markov Chain*) is a tuple *(**S**, **P**)*
  - ***S*** is a (finite) set of states
  - $\boldsymbol{P}_{ss'}$ is a state transition probability matrix
  - $\boldsymbol{P}_{ss'} = \mathrm{P}[S_{t+1} = s' \mid S_t = s]$

Markov Process: Student Example



Markov Process: Episodes Sampling

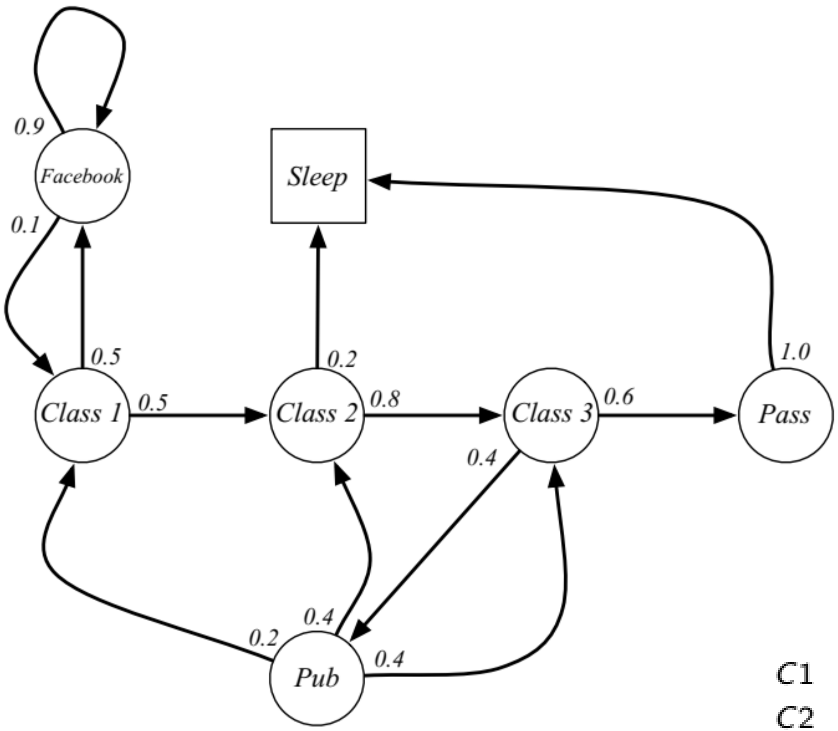


- Sample episodes for Student Markov Chain starting from $S_1 = C1$

$$S_1, \ldots, S_t$$

- C1 C2 C3 Pass Sleep
- C1 FB FB C1 C2 Sleep
- C1 C2 C3 Pub C2 C3 Pass Sleep
- C1 FB FB C1 C2 C3 Pub C1 FB FB FB C1 C2 C3 Pub C2 Sleep

Markov Process: Transition Probabilities



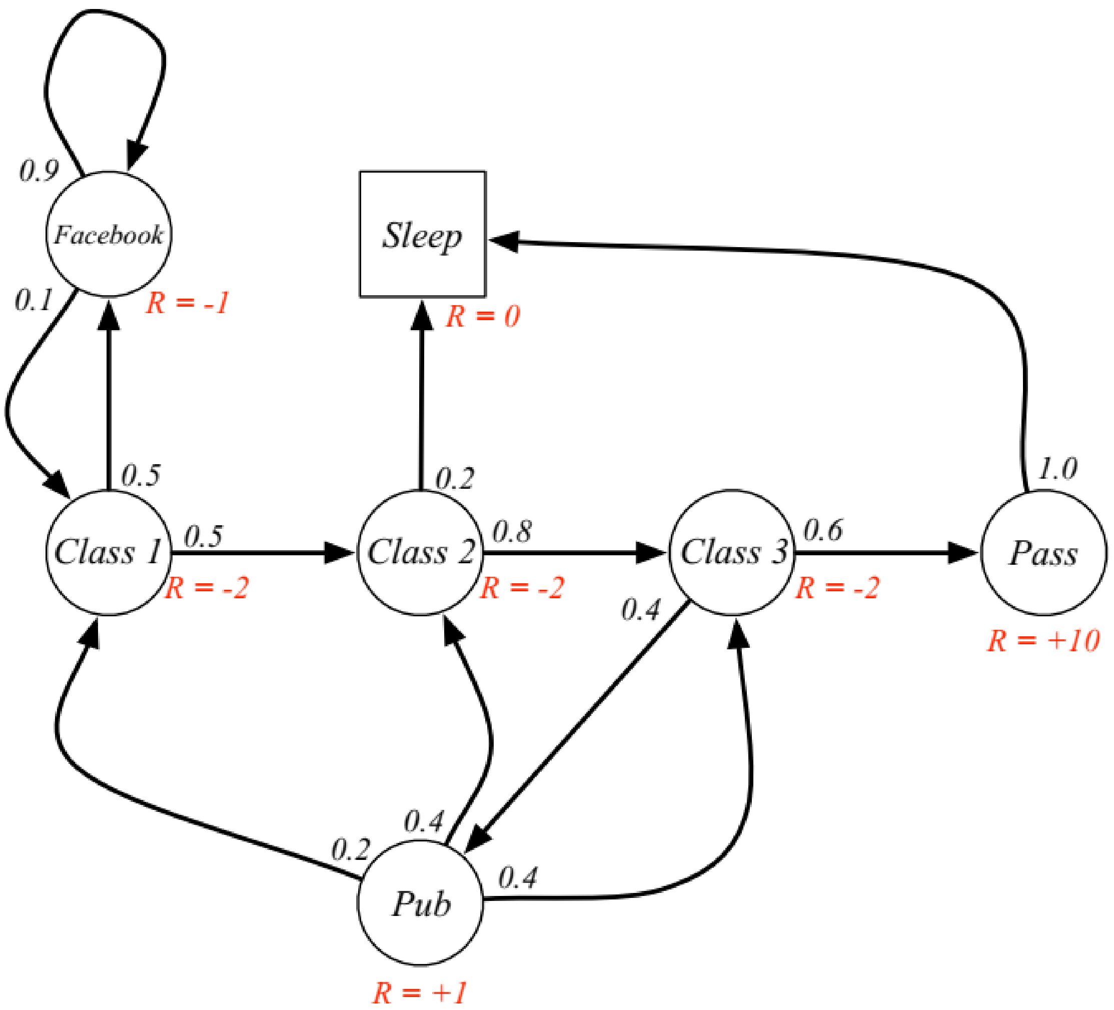
$$
\mathcal{P} = \begin{array}{cc}
 & \begin{array}{ccccccc} \mathit{C1} & \mathit{C2} & \mathit{C3} & \mathit{Pass} & \mathit{Pub} & \mathit{FB} & \mathit{Sleep} \end{array} \\
\begin{array}{c} \mathit{C1} \\ \mathit{C2} \\ \mathit{C3} \\ \mathit{Pass} \\ \mathit{Pub} \\ \mathit{FB} \\ \mathit{Sleep} \end{array} &
\begin{bmatrix}
 & 0.5 & & & & 0.5 & \\
 & & 0.8 & & & & 0.2 \\
 & & & 0.6 & 0.4 & & \\
 & & & & & & 1.0 \\
0.2 & 0.4 & 0.4 & & & & \\
0.1 & & & & & 0.9 & \\
 & & & & & & 1
\end{bmatrix}
\end{array}
$$

Markov Reward Process

- A Markov reward process is a Markov chain with values.

- *A Markov Reward Process* is a tuple *(**S**, **P**, **R**, **γ**)*
  - ***S*** is a (finite) set of states
  - $\boldsymbol{P}_{ss'}$ is a state transition probability matrix
  - $\boldsymbol{P}_{ss'} = \mathrm{P}[S_{t+1} = s' \mid S_t = s]$
  - ***R*** is a reward function, $\boldsymbol{R_s = \mathrm{E}\,[R_{t+1} \mid S_t = s]}$
  - ***γ*** is a discount factor, $\gamma \in [0, 1]$

MRP: Student Example



Return

- The *return* G_t is the total discounted reward from time-step t.

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

- The *discount* $\gamma \in [0, 1]$ is the present value of future rewards
- The value of receiving reward R after $k + 1$ time-steps is $\gamma^k R$.
- This values immediate reward above delayed reward.
  - γ close to 0 leads to "myopic" evaluation
  - γ close to 1 leads to "far-sighted" evaluation

Discount intuition

Most Markov reward and decision processes are discounted. Why?

- Mathematically convenient to discount rewards
- Avoids infinite returns in cyclic Markov processes
- Uncertainty about the future may not be fully represented
- If the reward is financial, immediate rewards may earn more interest than delayed rewards
- Animal/human behavior shows preference for immediate reward
- It is sometimes possible to use *undiscounted* Markov reward processes (i.e. $\gamma = 1$), e.g. if all sequences terminate

Value Function

- The value function $v(s)$ gives the long-term value of state s

- The *state value function* $v(s)$ of an MRP is the expected return starting from state s

$$v(s) = \mathrm{E}[G_t \mid S_t = s]$$

Student MRP returns

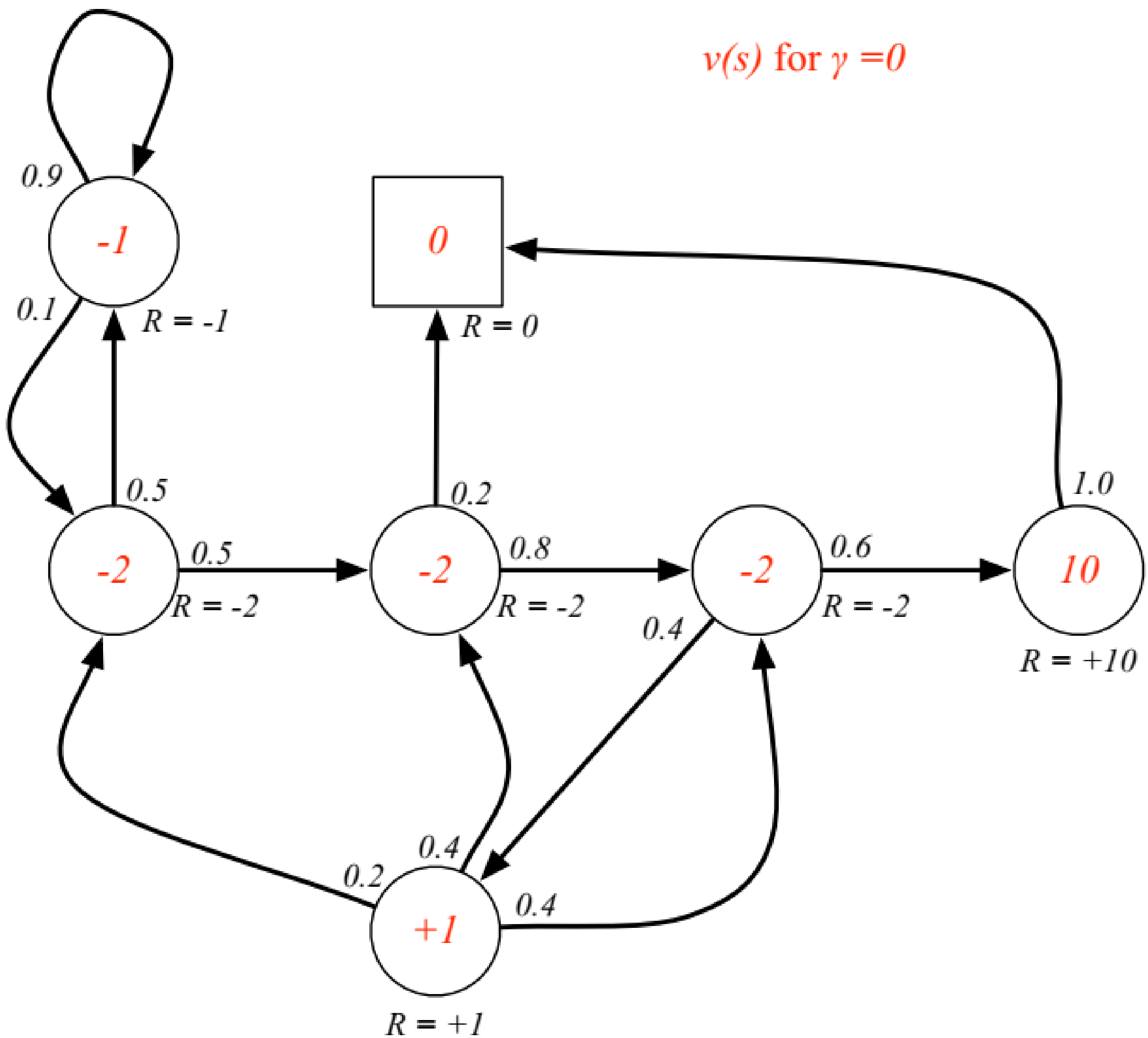
Sample returns for Student MRP with:

- $S_1 = C1$
- $\gamma = 0.5$

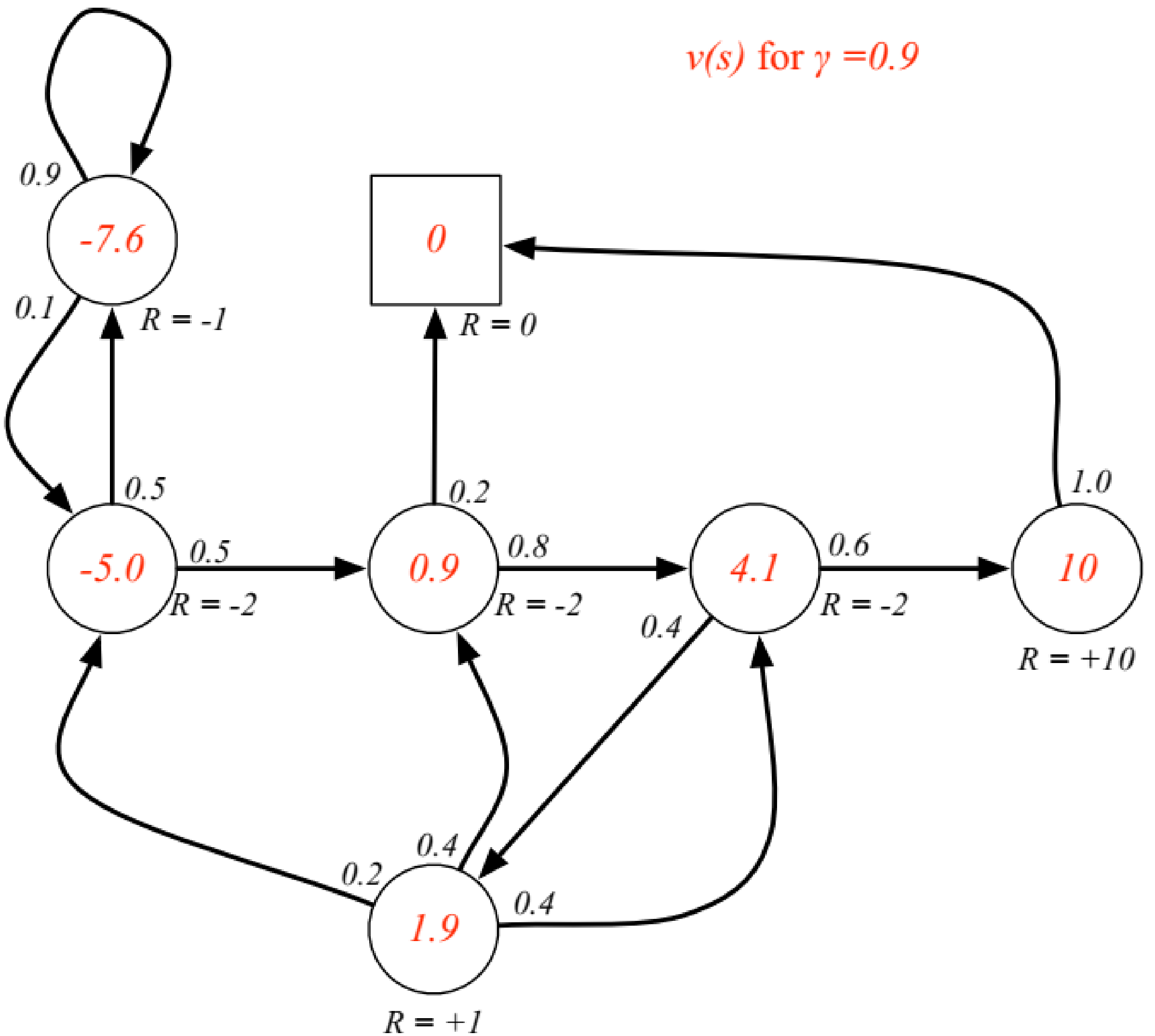
$$G_1 = R_2 + \gamma R_3 + ... + \gamma^{T-2} R_T$$

C1 C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 10 * \frac{1}{8}$	$=$	-2.25
C1 FB FB C1 C2 Sleep	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16}$	$=$	-3.125
C1 C2 C3 Pub C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 1 * \frac{1}{8} - 2 * \frac{1}{16} \dots$	$=$	-3.41
C1 FB FB C1 C2 C3 Pub C1 ...	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16} \dots$	$=$	-3.20
FB FB FB C1 C2 C3 Pub C2 Sleep			

State-Value function for student MRP



State-Value function for student MRP



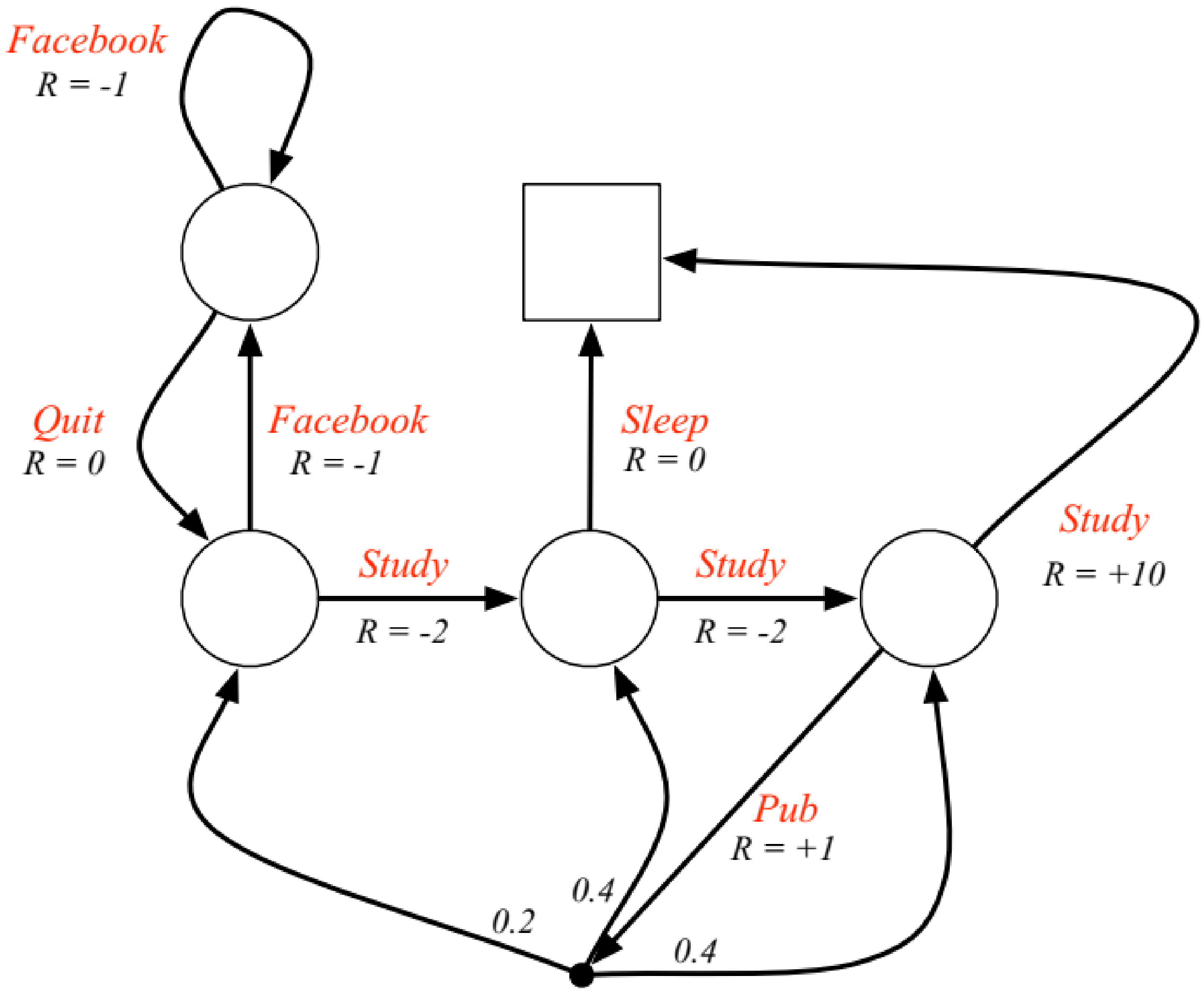
Markov Decision Process

- A Markov decision process (MDP) is a Markov reward process with
decisions. It is an *environment* in which all states are Markov.

- A Markov reward process is a Markov chain with values.

- *A Markov Decision Process* is a tuple $(\boldsymbol{S}, A, \boldsymbol{P}, \boldsymbol{R}, \boldsymbol{\gamma})$
  - $\boldsymbol{S}$ is a (finite) set of states
  - $\boldsymbol{A}$ is a finite set of actions
  - $\boldsymbol{P}^{\boldsymbol{a}}_{ss'}$ is a state transition probability matrix
  - $\boldsymbol{P}^{\boldsymbol{a}}_{ss'} = \mathrm{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$
  - $\boldsymbol{R}$ is a reward function, $\boldsymbol{R^a_s = \mathrm{E}\,[R_{t+1} \mid S_t = s, A_t = a]}$
  - $\boldsymbol{\gamma}$ is a discount factor, $\gamma \in [0, 1]$

Markov Decision Process: Example



MDP Policies

A *policy* π is a distribution over actions given states

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

- A policy fully defines the behavior of an agent
- MDP policies depend on the current state (not the history)
- i.e. Policies are *stationary* (time-independent), $A_t \sim \pi(\cdot \mid S_t), \forall t > 0$

MDP Policies

- Given an MDP $M = (\boldsymbol{S}, \mathbf{A}, \boldsymbol{P}, \boldsymbol{R}, \boldsymbol{\gamma})$ and a policy π
- The state sequence $\boldsymbol{S_1, \ldots, S_t}$ is a Markov process (S, P^{π})
- The state and reward sequence $S_1, R_2, S_2, \ldots,$ is a Markov reward process $(S, P^{\pi}, R^{\pi}, \gamma)$
- where

$$P^{\pi}_{s,s'} = \sum_{a \in A} \pi(a \mid s) P^{a}_{s,s'}$$

$$R^{\pi}_{s} = \sum_{a \in A} \pi(a \mid s) R^{a}_{s}$$

Updated Value functions

- The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π

$$v_\pi(s) = \mathrm{E}_\pi[G_t \mid S_t = s]$$

- The *action-value function* $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π

$$q_\pi(s, a) = \mathrm{E}_\pi[G_t \mid S_t = s, A_t = a]$$

Optimal Value Function

- The *optimal state-value function* $v_*(s)$ is the maximum value function over all policies

$$\boldsymbol{v_*(s) = \max_\pi (v_\pi(s))}$$

- The *optimal action-value function* $q*(s; a)$ is the maximum action-value function over all policies

$$\boldsymbol{q_*(s, a) = \max_\pi (q_\pi(s, a))}$$

- The optimal value function specifies the best possible performance in the MDP.

- An MDP is "solved" when we know the optimal value fn.

Optimal Policy

- Define a partial ordering over policies:

$$\pi \geq \pi' \text{ if } v_{\pi}(s) \geq v_{\pi'}(s) \;\; \forall s$$

<u>Theorem:</u> *For any Markov Decision Process*

- *There exists an optimal policy π_* that is better than or equal to all other policies:* $\pi_* \geq \pi \;\; \forall \pi$
- *All optimal policies achieve the optimal value function,*

$$v_{\pi*}(s) = v_*(s)$$

- *All optimal policies achieve the optimal action-value function*

$$q_{\pi*}(s, a) = q_*(s, a)$$

Recap: Dynamic Programming

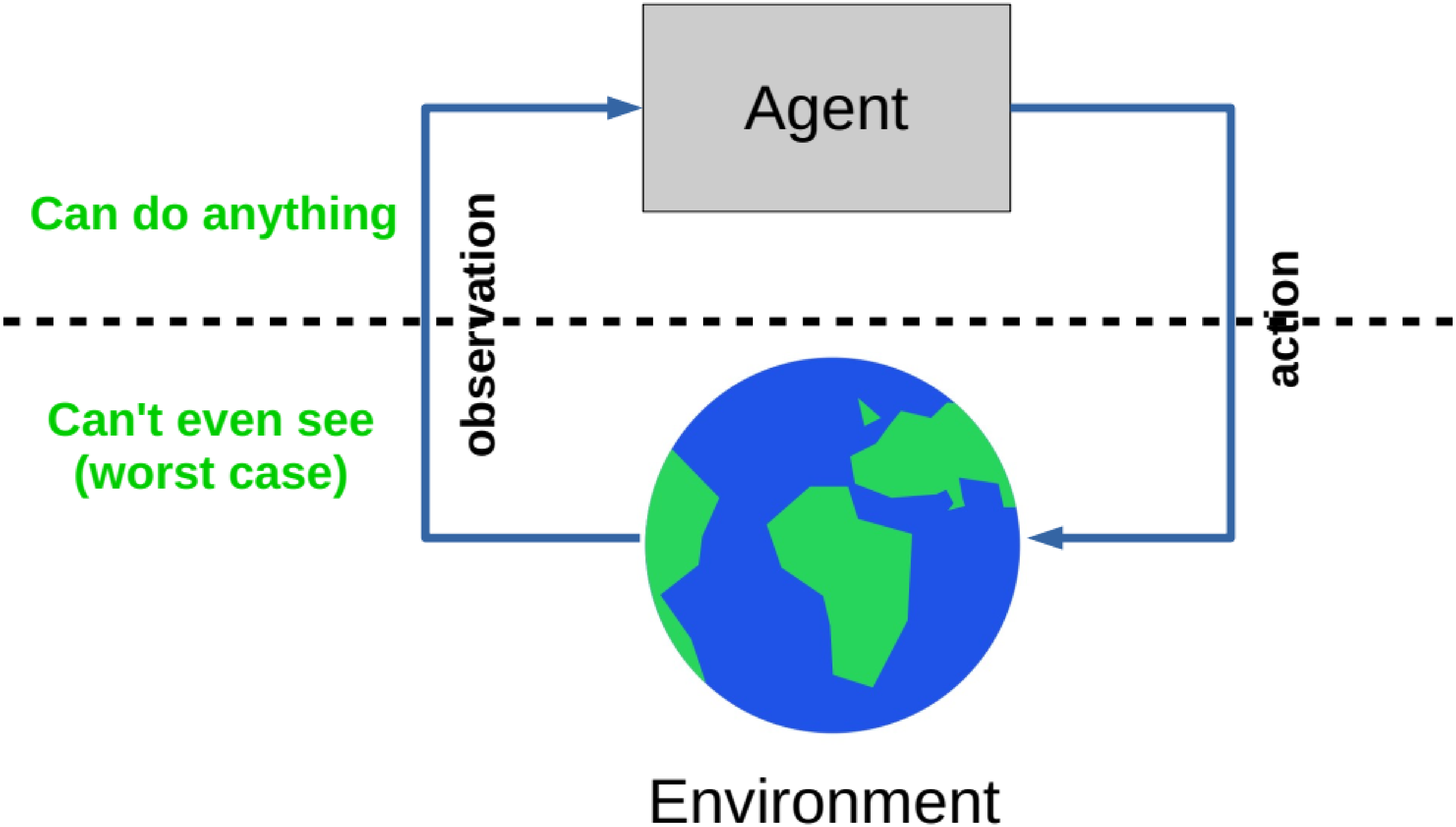
- $v_\pi(s)$, $v_*(s)$
- If you know $v_*(s)$, $p(r, s' \mid s, a) \rightarrow$ **know optimal policy**
- We can learn $v_*(s)$ with Dynamic Programming:

$$v_*(s) = \max_a \sum_{r,s'} p(r, s' \mid s, a)[r + \gamma v_*(s')]$$

- $q_\pi(s, a)$, $q_*(s, a)$

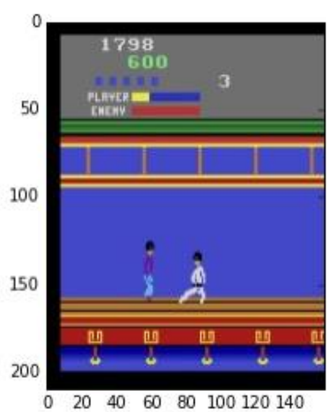
$$q_*(s, a) = \sum_{r,s'} p(r, s' \mid s, a)[r + \gamma \max_{a'} q_*(s', a')]$$

Decision making: reality check



Decision making: reality check

Can do anything



Model-Free Setup

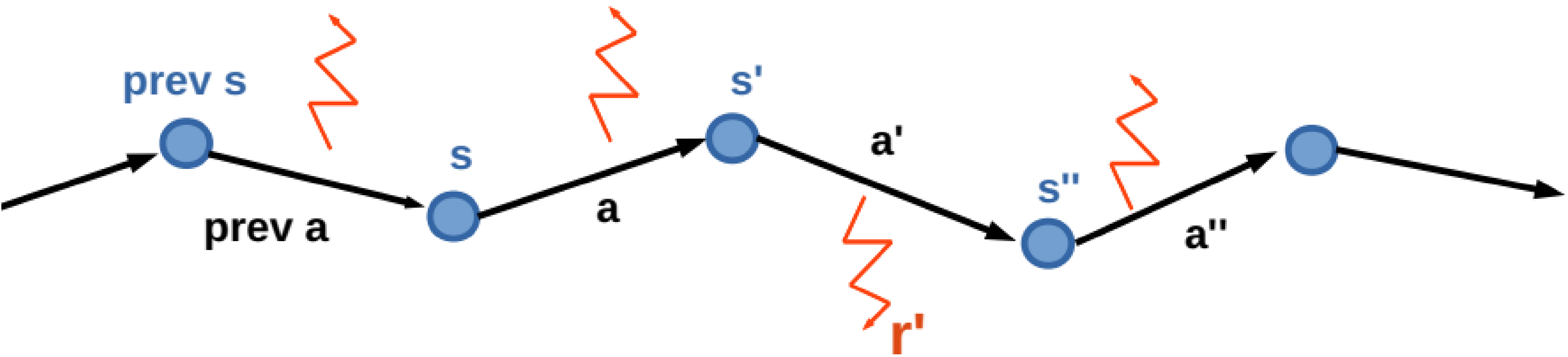
- We don’t know internal environment representation, e.g.

$$\boldsymbol{p}(\boldsymbol{r}, \boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a}) \text{ - unknown}$$

What should we do?

Learning from trajectories

- s_1 -> a_1 -> r_1 -> s_2 -> .... -> s_n – *trajectory*



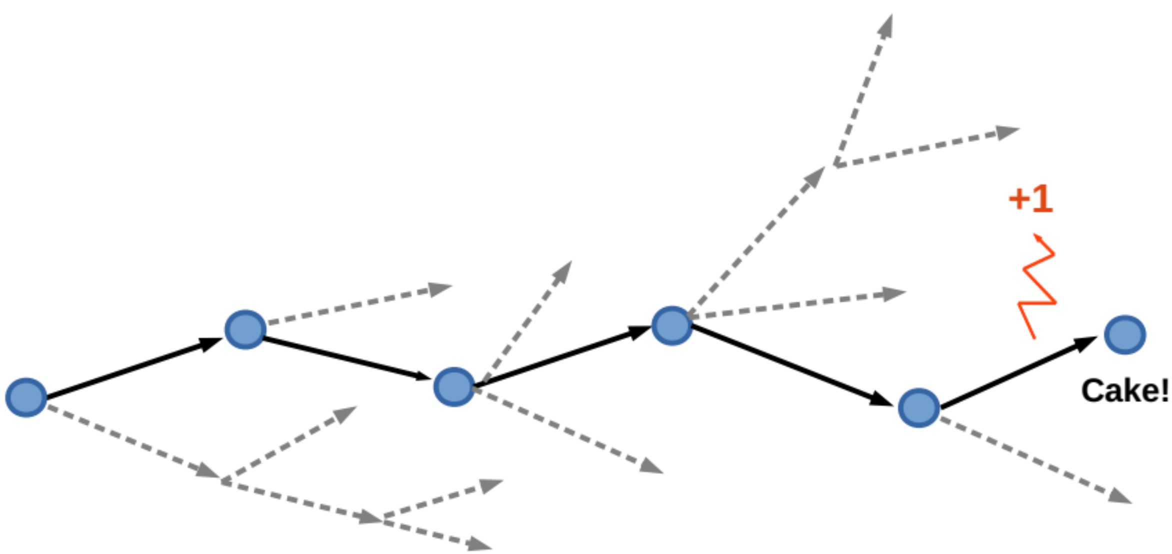
- Model-based setup:
  - you can apply Dynamic Programming
  - you can plan (!)
- Model-free setup:
  - you can experiment with different actions
  - no guaranties (!!!)

Learning from trajectories

- $s_1 \to a_1 \to r_1 \to s_2 \to \ldots \to s_n$ – *trajectory*
- *We can sample trajectories (a lot of trajectories!)*
- *What should we learn ?*
  - $p(r, s' \mid s, a)$
  - $v_\pi(s)$
  - $q_\pi(s, a)$

Monte-Carlo RL

- Just like N+1 heuristic:
  - Get all trajectories containing particular (s, a)
  - Estimate $G_t(s, a)$ for each trajectory
  - Average them to get *estimation* of expectation



Monte-Carlo RL

- MC methods learn directly from episodes of experience
- MC is *model-free*: no knowledge of MDP transitions / rewards
- MC learns from *complete* episodes: no bootstrapping
- MC uses the simplest possible idea: value = mean return
- <u>Note:</u> can only apply MC to *episodic* MDPs
    - All episodes must terminate

Incremental Mean

- The mean $\mu_1, \mu_2, \ldots, \mu_k$ of a sequence $x_1, x_2, \ldots, x_k$ can be computed incrementally:

$$
\begin{aligned}
\mu_k &= \frac{1}{k} \sum_{j=1}^{k} x_j \\
&= \frac{1}{k} \left(x_k + \sum_{j=1}^{k-1} x_j \right) \\
&= \frac{1}{k} \left(x_k + (k-1)\mu_{k-1} \right) \\
&= \mu_{k-1} + \frac{1}{k} \left(x_k - \mu_{k-1} \right)
\end{aligned}
$$

Temporal Difference Learning

- TD methods learn directly from episodes of experience
- TD is *model-free*: no knowledge of MDP transitions / rewards
- TD learns from *incomplete* episodes, by *bootstrapping*
- TD updates a guess towards a guess

Temporal Difference

- Just like in the 'incremental mean' example we can improve $\boldsymbol{q}_{\pi}(\boldsymbol{s}, \boldsymbol{a})$ iteratively:

$$\boldsymbol{q}_{*}(\boldsymbol{s}, \boldsymbol{a}) = \sum_{r,s'} \boldsymbol{p}(\boldsymbol{r}, \boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a})[\boldsymbol{r} + \gamma \, \mathbf{max}_{a'} \boldsymbol{q}_{*}(\boldsymbol{s}', \boldsymbol{a}')]$$

- We don't have $\boldsymbol{p}(\boldsymbol{r}, \boldsymbol{s}' \mid \boldsymbol{s}, \boldsymbol{a})$ to compute 'fair' expectation, so what should we do?

Temporal Difference

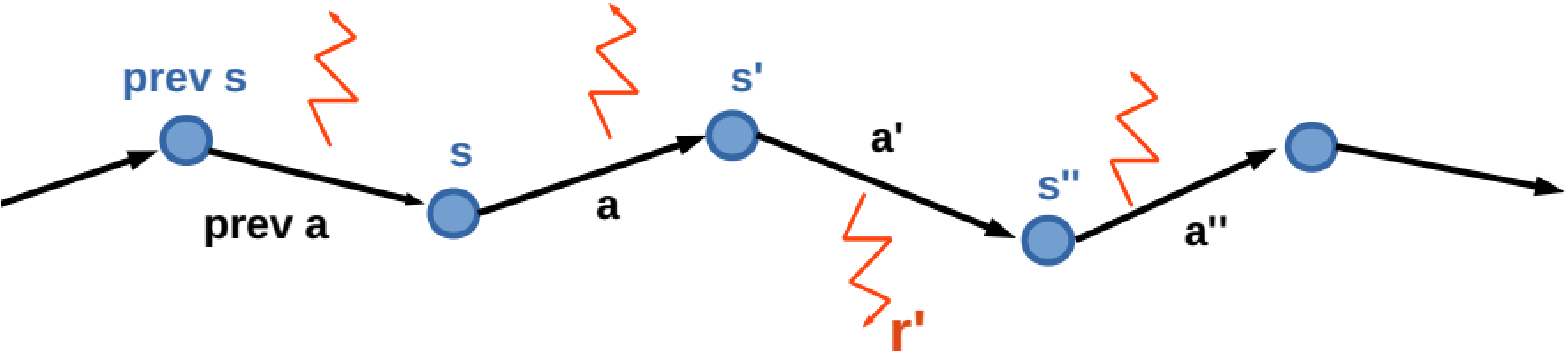
$$\sum_{r,s'} p(r, s' \mid s, a)[r + \gamma \max_{a'} q_*(s', a')] \approx$$

$$\approx \frac{1}{N} \sum_i r_i + \gamma \max_{a'} Q(s_i', a')$$

- One more trick: use incremental averaging with 1 sample.

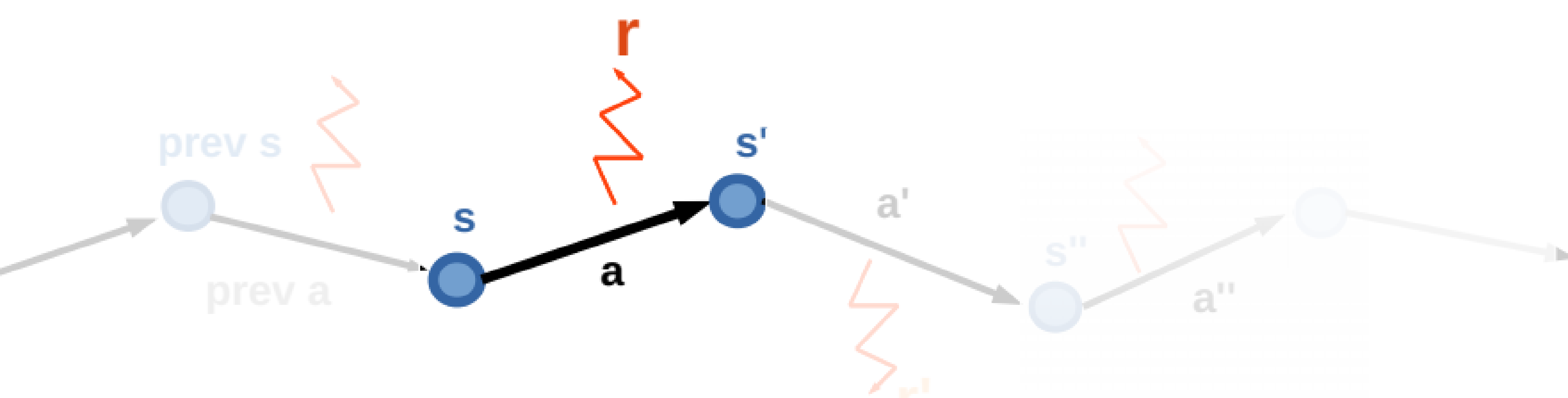
$$Q(s_t, a_t) = \alpha\,(r_t + \gamma\, max_{a'}\, Q(s_{t+1}, a')) + (1 - \alpha)\, Q(s_t, at)$$

Q-learning



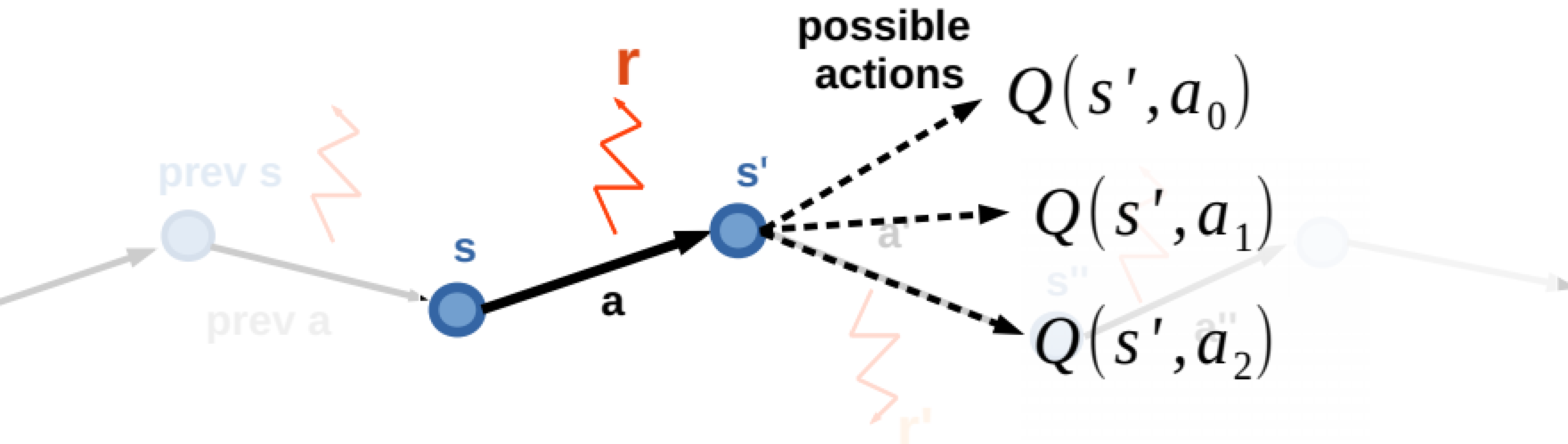
- Works on a sequence of
  - states (s)
  - actions (a)
  - rewards (r)

Q-learning



- Initialize $Q(s, a)$ with zeros
- Cycle:
  - Sample **<s, a, r, s'>** from environment

Q-learning



- Initialize $\boldsymbol{Q(s,a)}$ with zeros
- Cycle:
  - Sample **<s, a, r, s'>** from environment
  - Compute $\widehat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a}) = \boldsymbol{r}(\boldsymbol{s},\boldsymbol{a}) + \boldsymbol{\gamma} \, \boldsymbol{max}_{\boldsymbol{a_i}} \, \boldsymbol{Q}(\boldsymbol{s'},\boldsymbol{ai})$
  - Update: $\boldsymbol{Q}(\boldsymbol{s_t},\boldsymbol{at}) = \ \boldsymbol{\alpha} \, \widehat{\boldsymbol{Q}}(\boldsymbol{s},\boldsymbol{a})$ **+ (1 -** $\boldsymbol{\alpha}$**)** $\boldsymbol{Q}(\boldsymbol{s_t},\boldsymbol{at})$

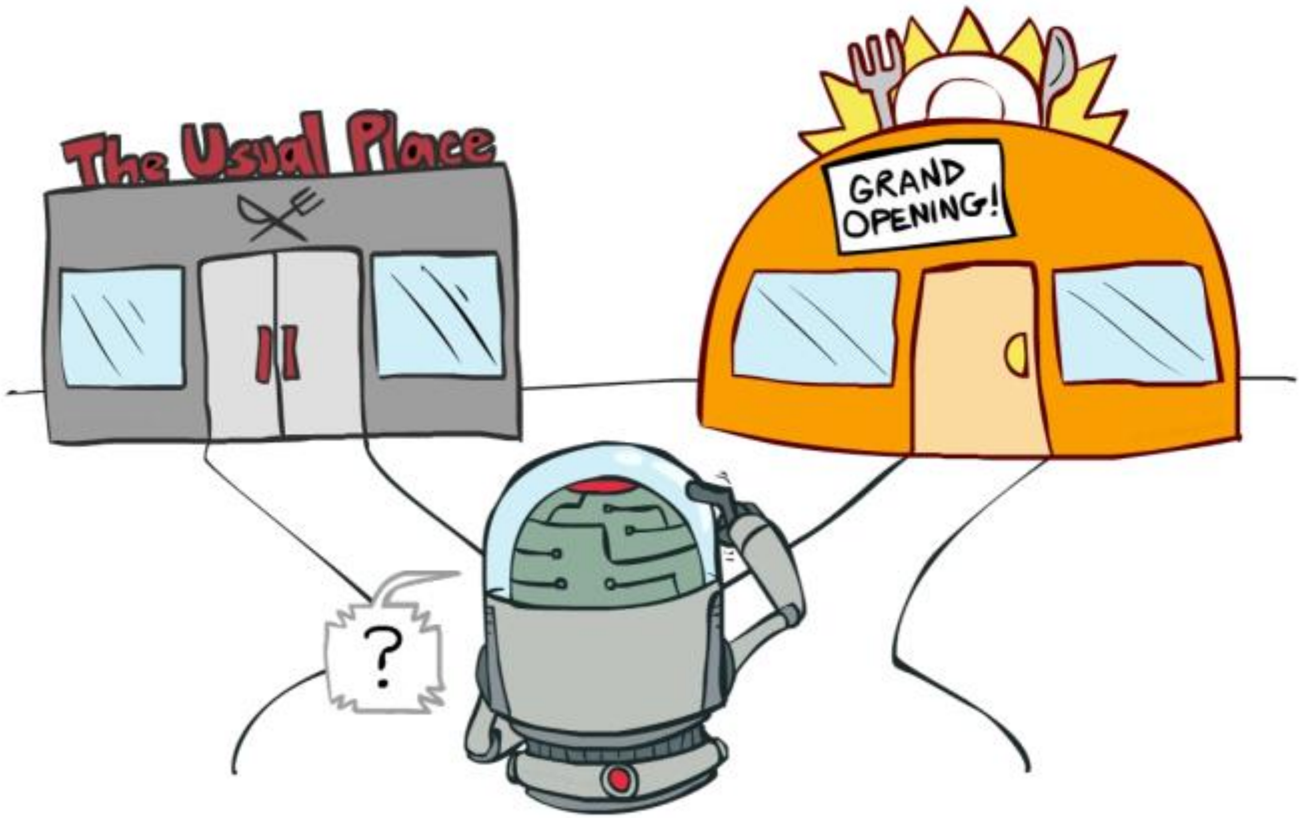
MC vs TD

- TD can learn *before* knowing the final outcome
  - TD can learn online after every step
  - MC must wait until end of episode before return is known

- TD can learn *without* the final outcome
  - TD can learn from incomplete sequences
  - MC can only learn from complete sequences
  - TD works in continuing (non-terminating) environments
  - MC only works for episodic (terminating) environments

Exploration/Exploitation Revisited

- Balance between using what you learned and trying to find something even better



Exploration/Exploitation Revisited

- Strategies:
  - ε-greedy
    With probability ε take random action,
    otherwise take optimal action.

  - Softmax
    Pick action proportional to softmax of shifted normalized Q-values.

$$\pi(a \mid s) = softmax(\, Q(s,a) \,/\, \tau)$$

  - ε- dithering

    Adding random noise to Q-values with ε probability

Exploration/Exploitation over time

- If you want to converge to optimal policy you need to gradually reduce exploration.

- <u>Example:</u>
  Initialize ε-greedy $\varepsilon = 0.5$, then gradually reduce it

  - If $\varepsilon \to 0$, it's **greedy in the limit**
  - Be careful with non-stationary environments

Reinforcement Learning in the Wild

- Reinforcement learning can be used to solve *large* problems, e.g.

  - Backgammon: ***1020 states***
  - Computer Go: **10170 states**
  - Helicopter: **continuous state space**

- How can we scale up the model-free methods for *prediction* and *control ?*

Curse of dimensionality in RL

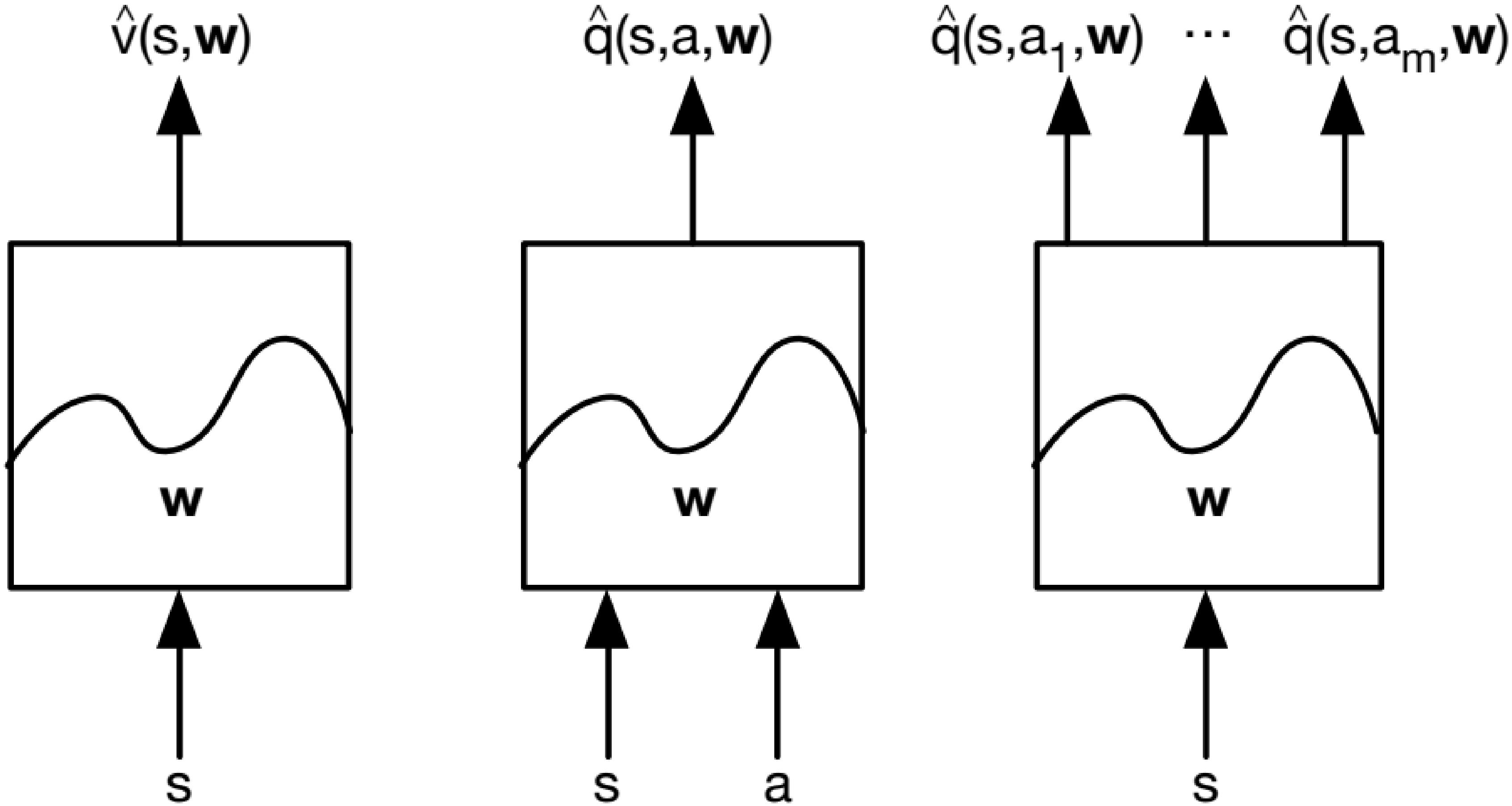
Problem:

- State space is usually large, sometimes continuous.
- How about action space ?

- However, states do have a structure, similar states have similar action outcomes

What should we do?

Types of Value Function Approximation



Which class of function to choose?

- There are many function approximators, e.g.
  - *Linear combinations of features*
  - *Neural network*
  - Decision tree
  - Nearest neighbor
  - Fourier / wavelet bases
  - ...

Gradient Descent

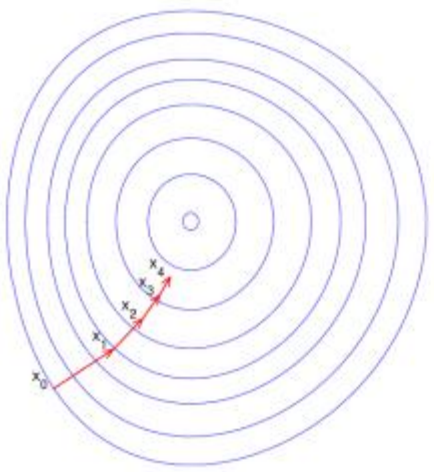
- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$
- Define the *gradient* of $J(\mathbf{w})$ to be

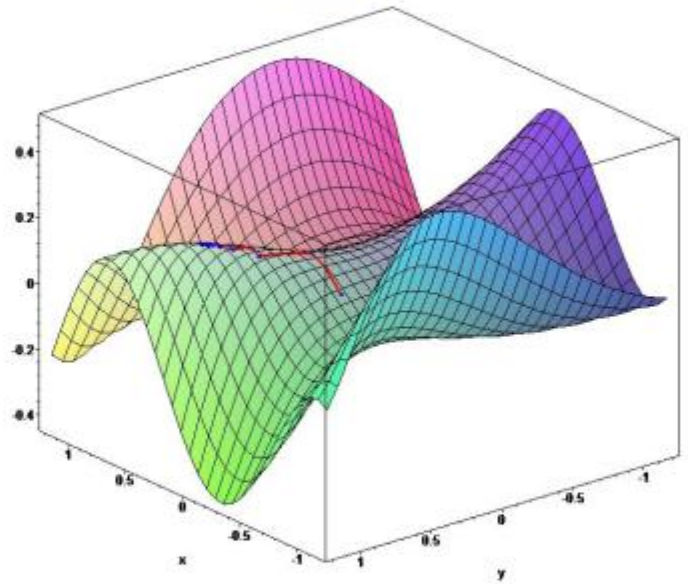
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$

- To find a local minimum of $J(\mathbf{w})$:
  - Adjust $\mathbf{w}$ in direction of -ve gradient

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where α is a step-size parameter





SGD for Value Function approximation

- Goal: find parameter vector **w** minimizing mean-squared error between approximate value $v'(s, \mathbf{w})$ and true value $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2 \right]$$

- Gradient descent finds a local minimum

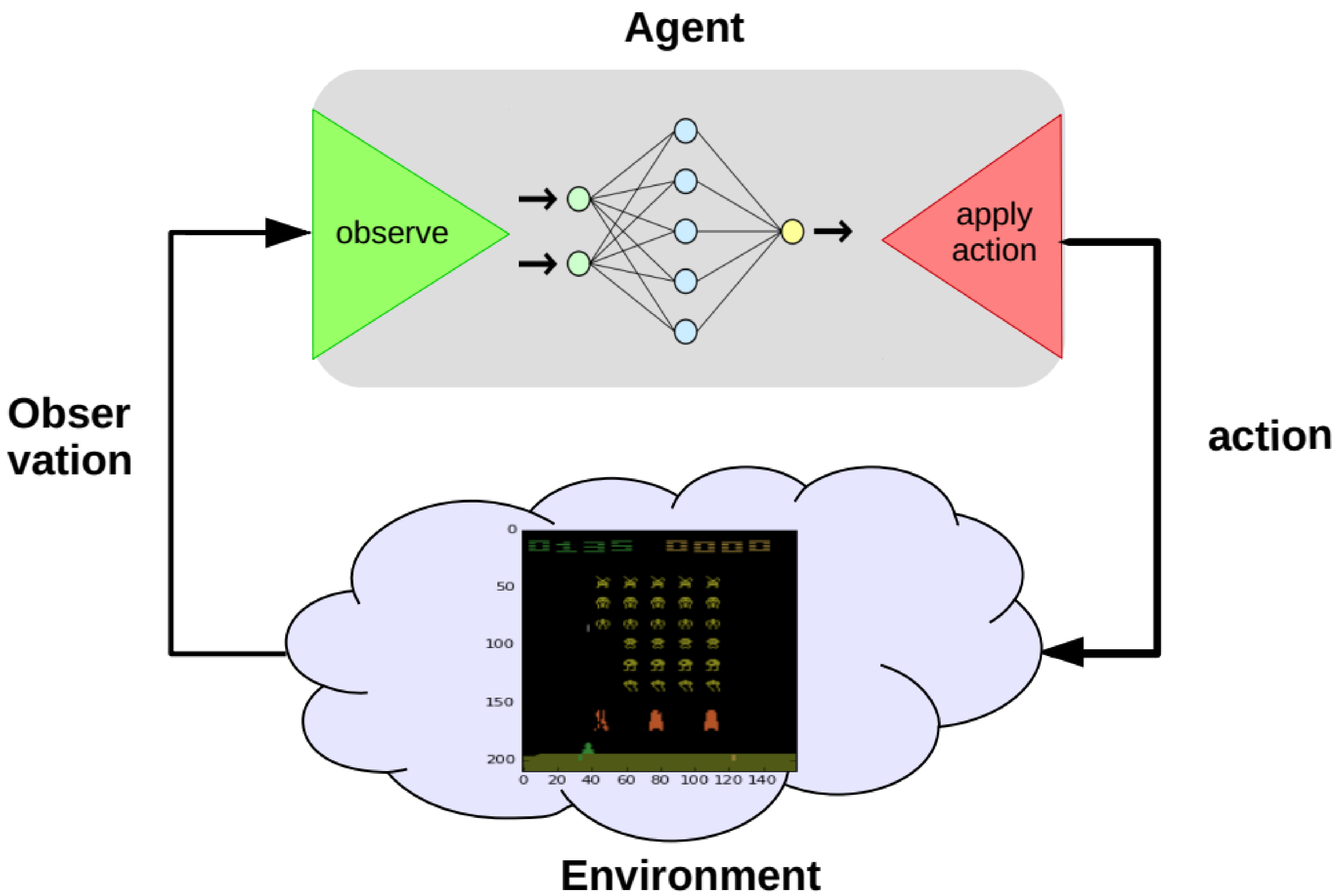
$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right] \end{aligned}$$

- Stochastic gradient descent *samples* the gradient

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

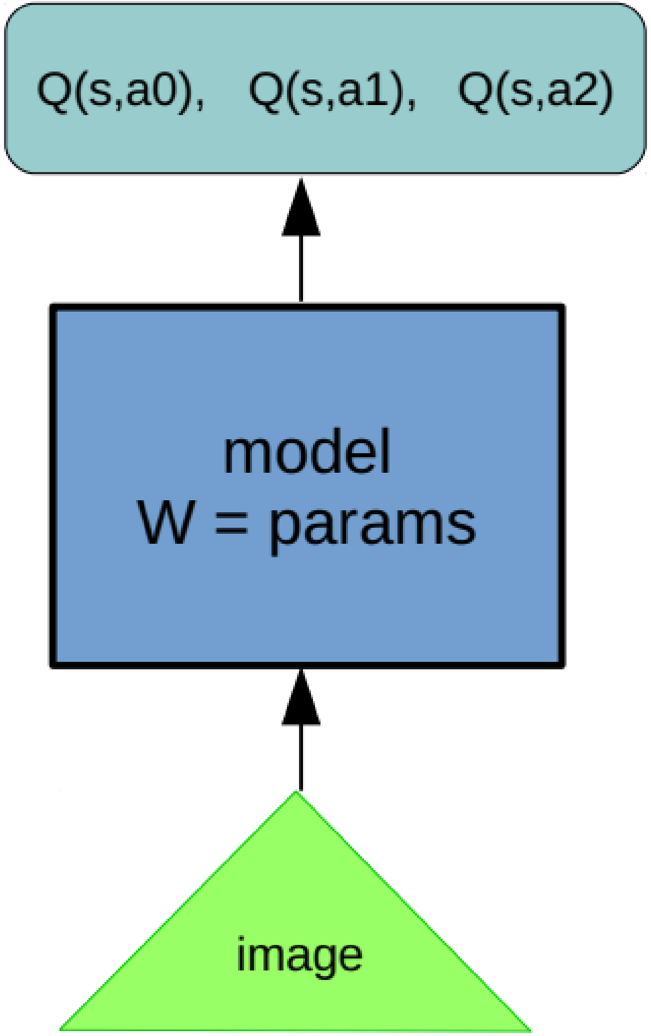
- Expected update is equal to full gradient update

Atari again



Agent
observe
apply action
Obser vation
action
0
50
100
150
200
0 20 40 60 80 100 120 140
Environment

Approximate Q-learning



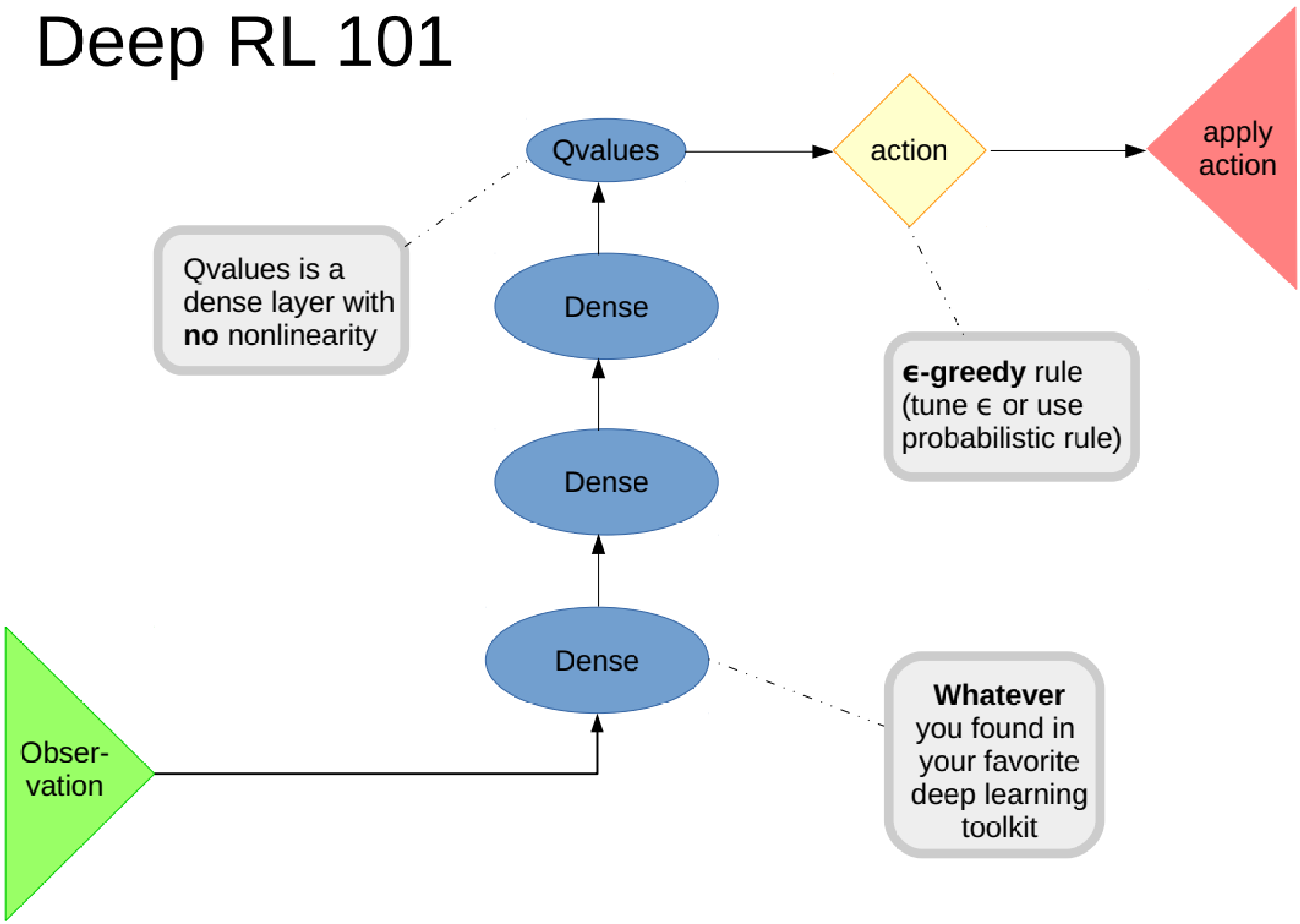
- Initialize **W.**
- Cycle:
  - Sample **<s, a, r, s'>** from environment
  - Compute $\hat{\boldsymbol{Q}}(\boldsymbol{s}, \boldsymbol{a}) = \boldsymbol{r}(\boldsymbol{s}, \boldsymbol{a}) + \boldsymbol{\gamma} \, \boldsymbol{max}_{\boldsymbol{a_i}} \, \boldsymbol{Q}(\boldsymbol{s}', \boldsymbol{ai})$
  - Objective:

$$\boldsymbol{L} = [\boldsymbol{Q}(\boldsymbol{s_t}, \boldsymbol{at}) - \hat{\boldsymbol{Q}}(\boldsymbol{s_t}, \boldsymbol{at})]^2$$

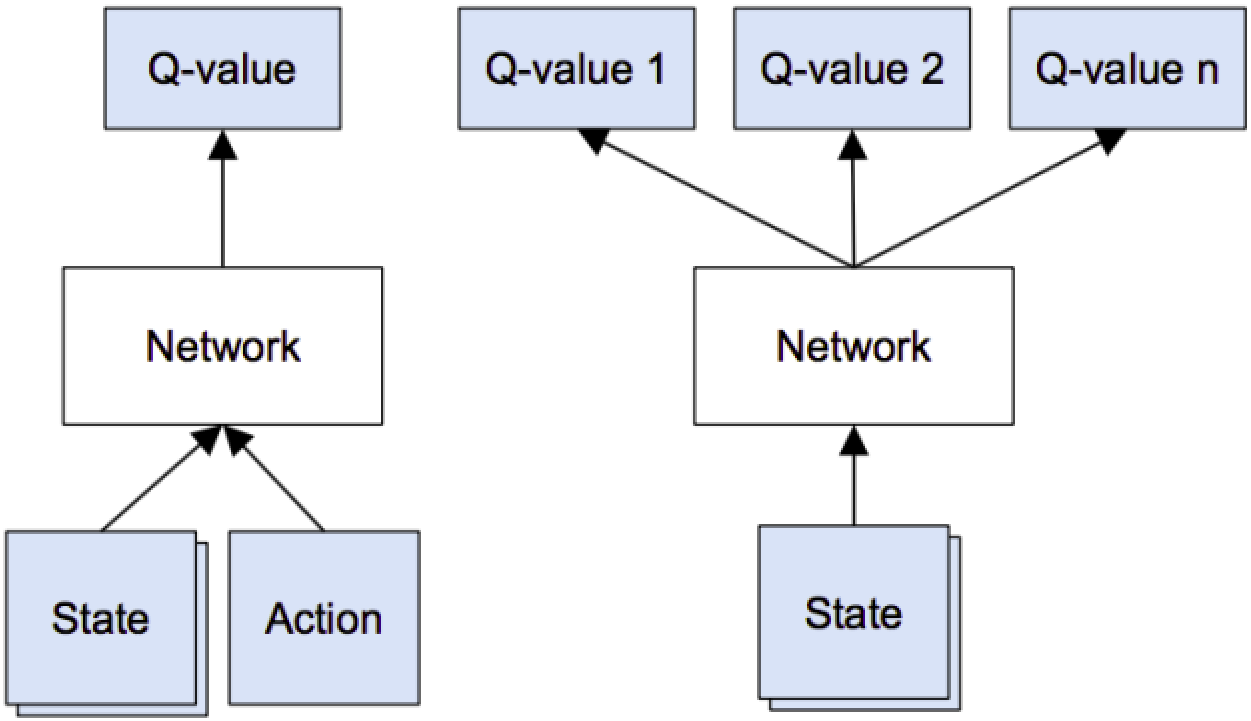
  - SGD Update:

$$W_{t+1} = W_t - \alpha \frac{\partial L}{\partial wt}$$

RL Mechanics



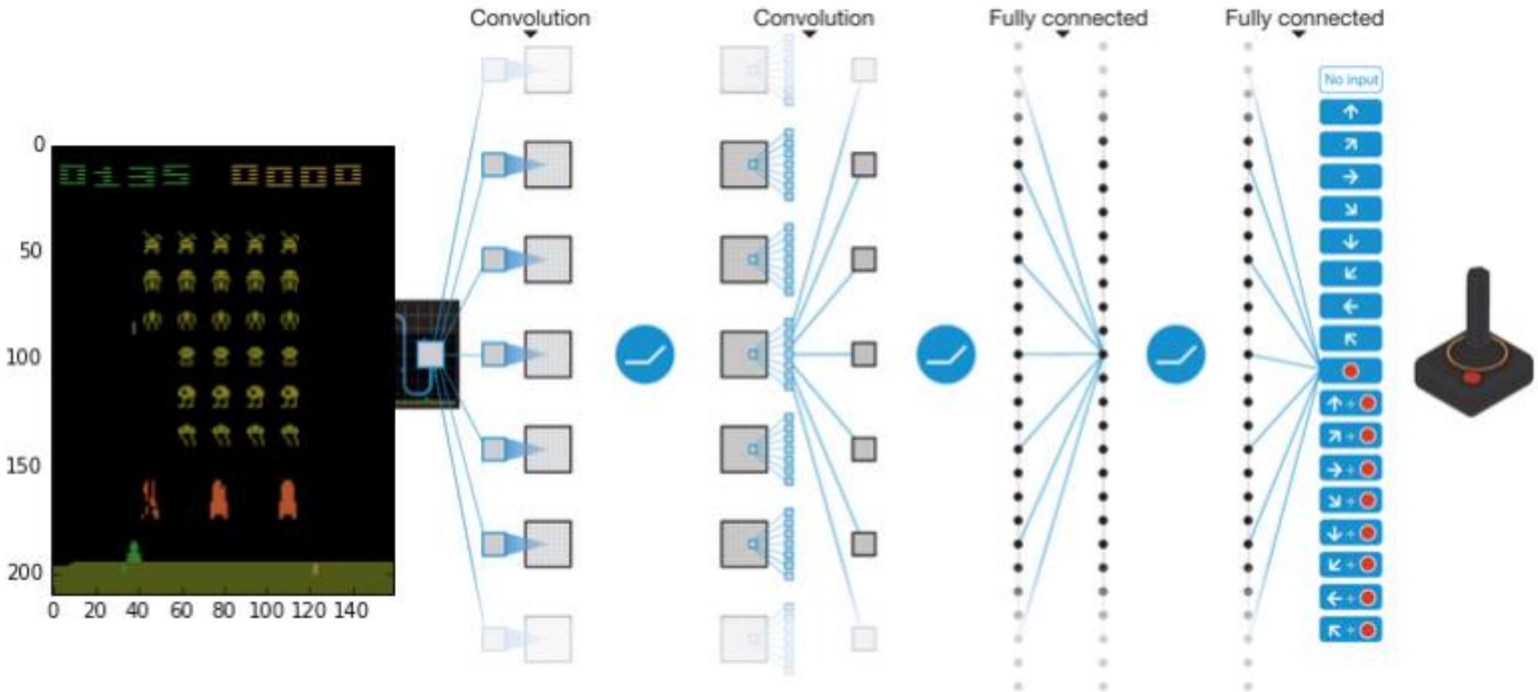
Architectures



Given **(s,a)**
Predict Q(s,a)

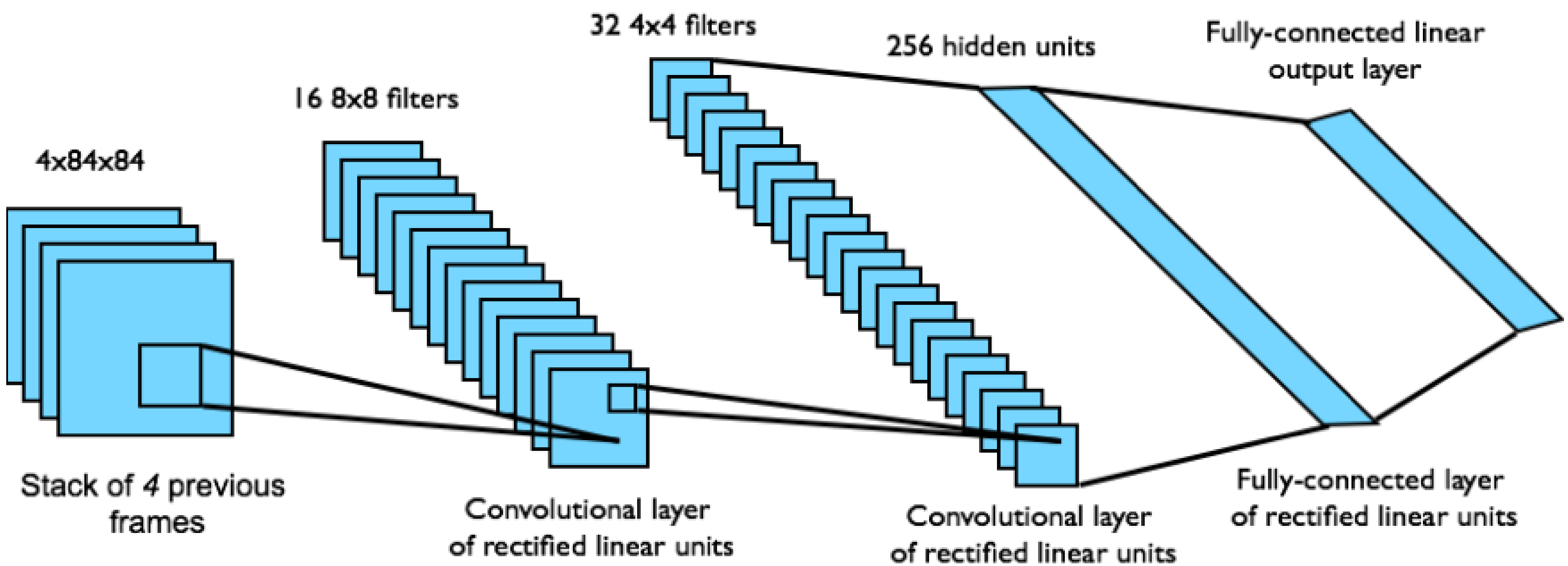
Given **s** predict all q-values
Q(s,a0), Q(s,a1), Q(s,a2)

From theory to practice: DQN case

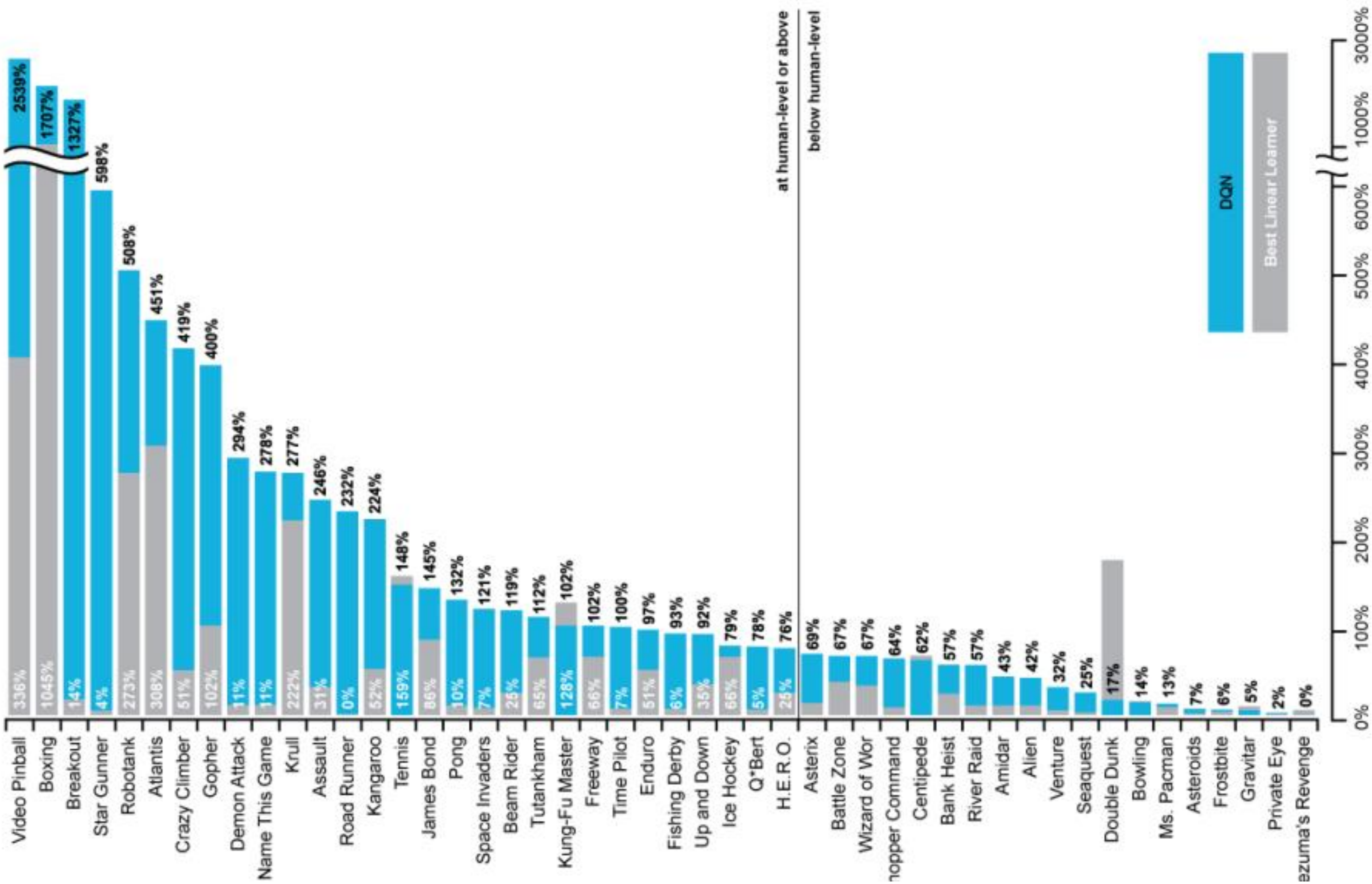


DQN: Atari

- End-to-end learning of values ***Q*(*s*, *a*)** from pixels ***s***
- Input state *s* is stack of raw pixels from last **4** frames
- Output is *Q*(*s*, *a*) for **18** joystick/button positions
- Reward is change in score for that step



DQN results on Atari



DQN: under the hood

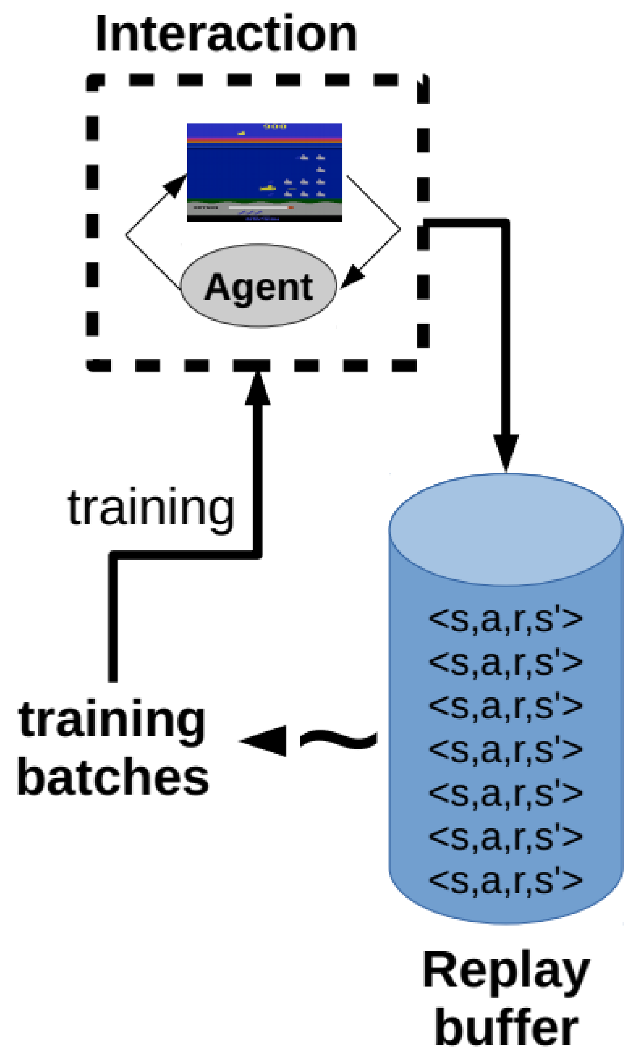
- DQN uses experience replay and fixed Q-targets:
  - Take action a_t according to *e*-greedy policy
  - Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
  - Sample random mini-batch of transitions (s, a, r, s') from D
  - Compute Q-learning targets w.r.t. old, fixed parameters w^-
  - Optimize MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

  - Using variant of stochastic gradient descent

Experience Replay

- **Idea:** store several past interactions <*s*,*a*,*r*,*s'*>
- Train on random subsamples
- **Any +/-** ?



DQN: Atari Breakout

- https://www.youtube.com/watch?v=TmPfTpjtdgg

RL Literature

- **An Introduction to Reinforcement Learning, Sutton and Barto, 1998**

  Available free online!

  last update: March 21, 2018

  http://incompleteideas.net/book/bookdraft2018mar21.pdf

- **Algorithms for Reinforcement Learning, Szepesvari, Morgan and Claypool, 2010**

  Available free online!

  last update: July 8, 2017

  https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf