Lecture 2.0

Reinforcement Learning: Q-learning

Alexey Gruzdev
alexey.s.gruzdev@gmail.com
HSE, Winter 2020

Recap: Dynamic Programming

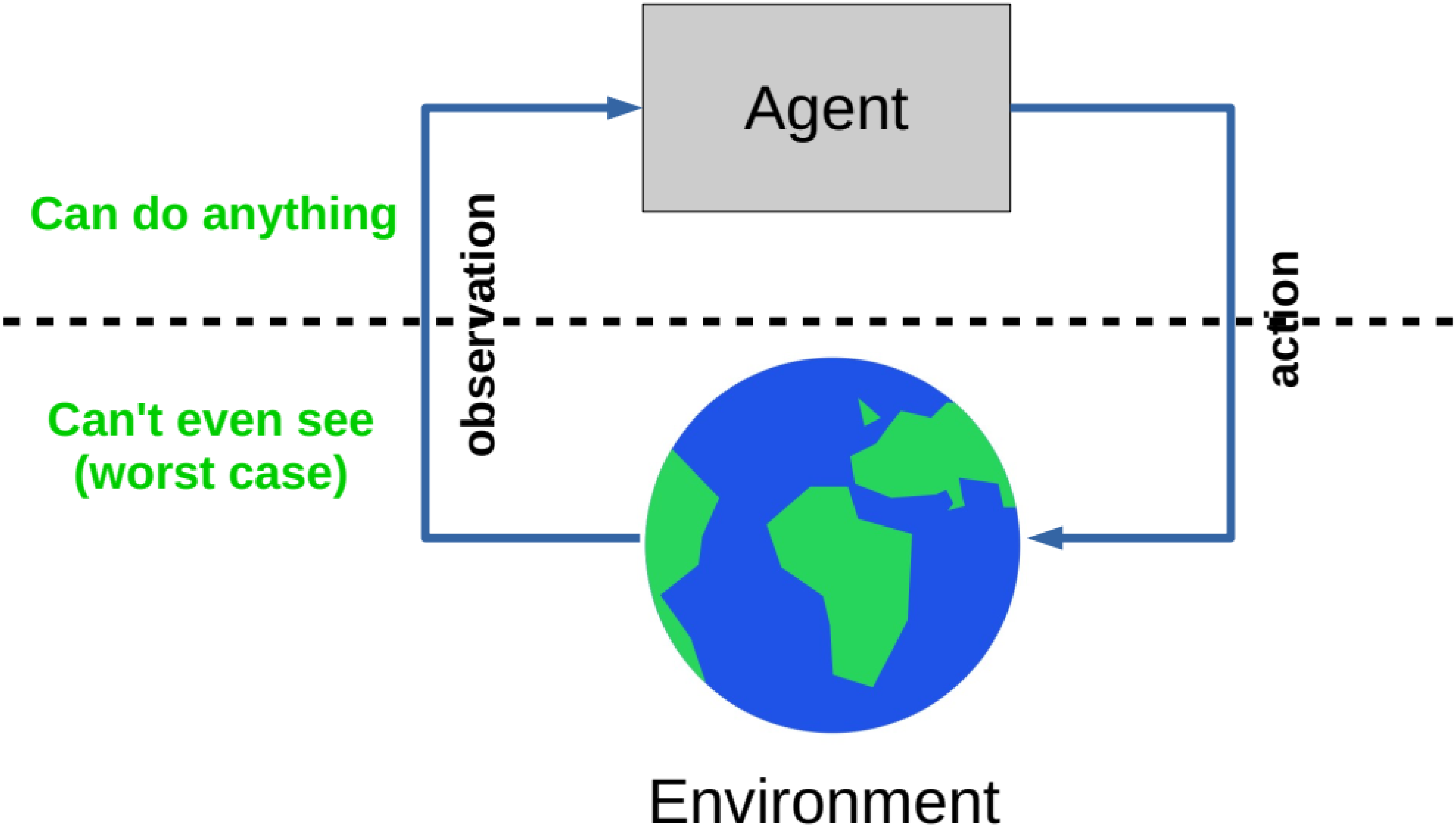
- $v_\pi(s)$, $v_*(s)$
- If you know $v_*(s)$, $p(r, s' \mid s, a) \rightarrow$ **know optimal policy**
- We can learn $v_*(s)$ with Dynamic Programming:

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

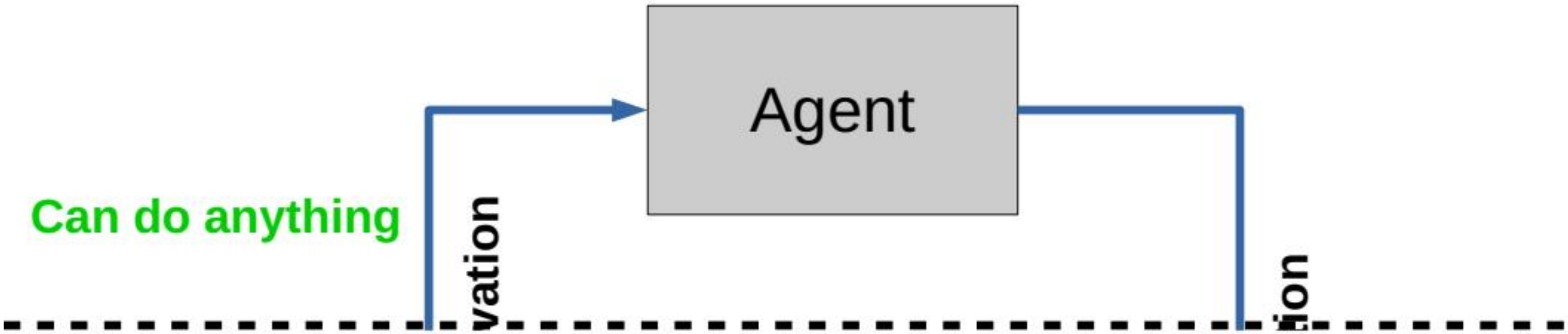
- $q_\pi(s, a)$, $q_*(s, a)$

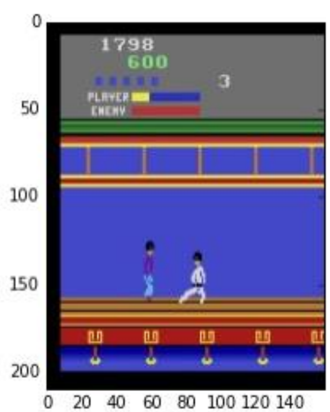
$$q_*(s, a) = \sum_{r,s'} p(r, s' \mid s, a)[r + \gamma \max_{a'} q_*(s', a')]$$

Decision making: reality check



Decision making: reality check





Model-Free Setup

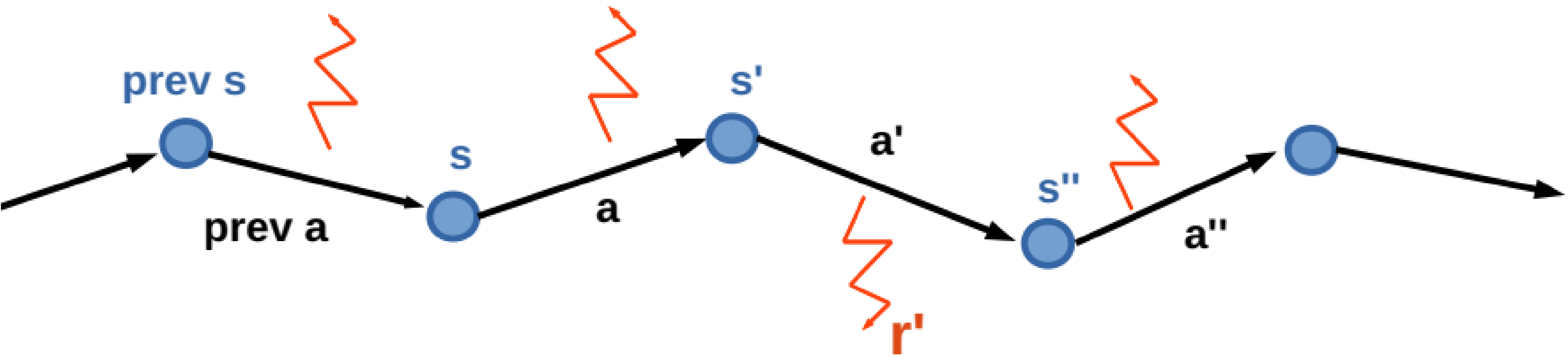
- We don't know internal environment representation, e.g.

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

What should we do?

Learning from trajectories

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



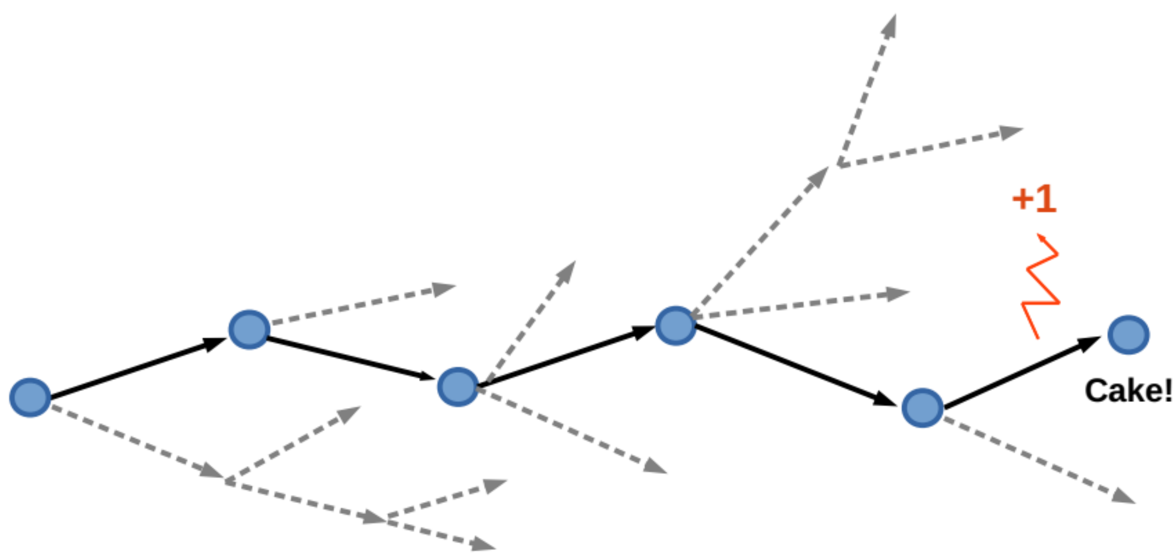
- Model-based setup:
  - you can apply Dynamic Programming
  - you can plan (!)
- Model-free setup:
  - you can experiment with different actions
  - no guaranties (!!!)

Learning from trajectories

- s_1 -> a_1 -> r_1 -> s_2 -> .... -> 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
- Note: can only apply MC to *episodic* MDPs
  - All episodes must terminate

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

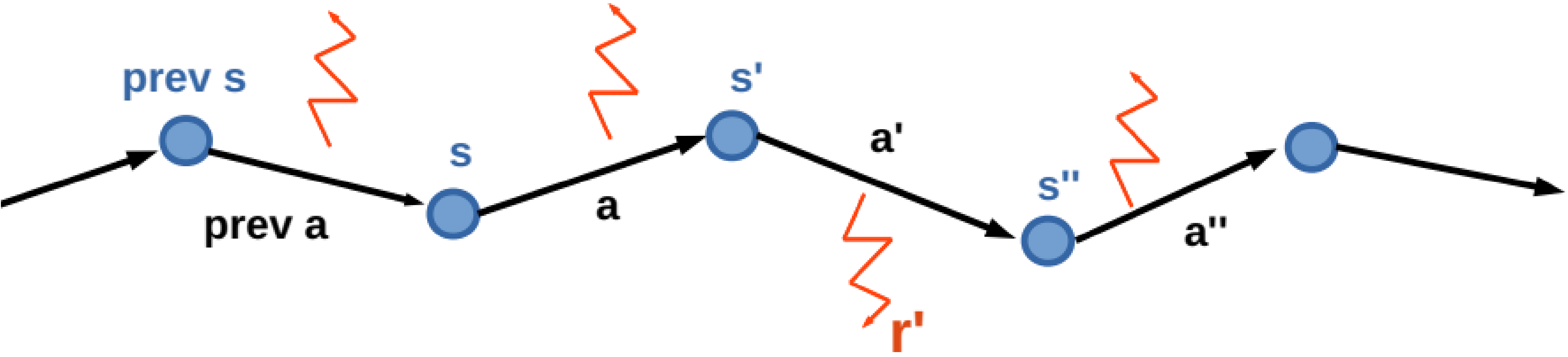
$$\sum_{r,s'} p(r, s' \mid s, a)[r + \gamma \max_{a'} q_*(s', a')] \approx$$

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

- One more trick: use alpha-smoothing for updating Q-values.

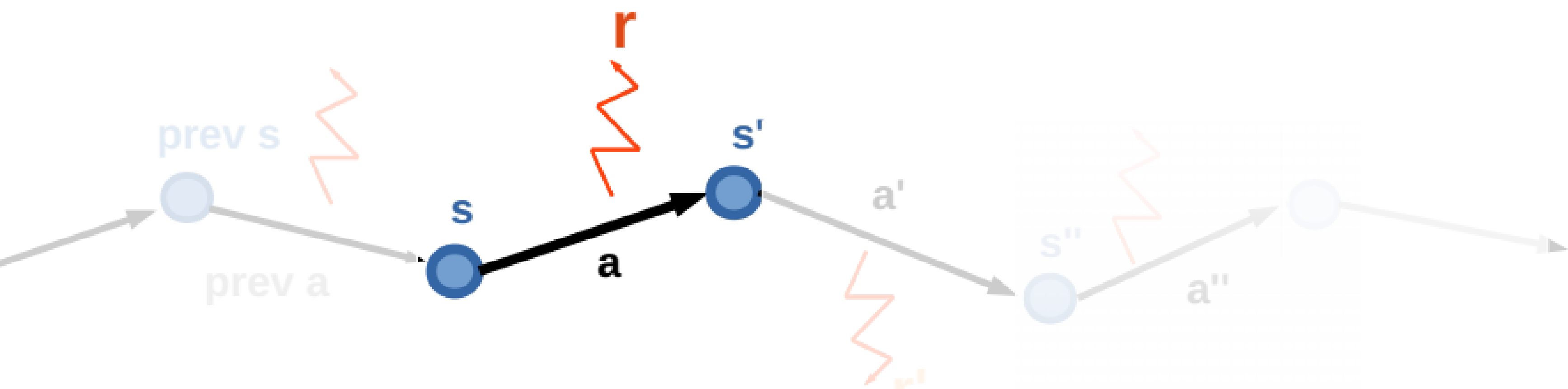
$$Q(s_t, a_t) = \alpha \left(r_t + \gamma \, max_{a'} Q(s_{t+1}, a')\right) + (1 - \alpha) \, Q(s_t, at)$$

Q-learning



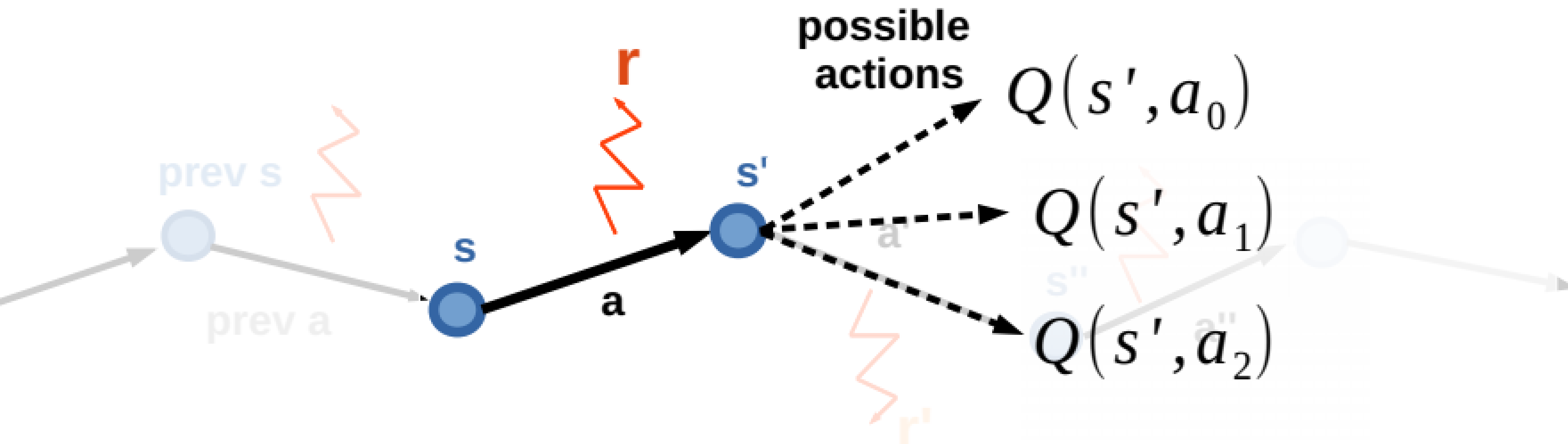
- Works on a sequence of
  - states (s)
  - actions (a)
  - rewards (r)

Q-learning



- Initialize $\boldsymbol{Q(s, a)}$ 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 $\hat{Q}(s,a) = r(s,a) + \gamma \max_{a_i} Q(s',ai)$
  - Update: $Q(s_t,at) = \ \alpha\,\hat{Q}(s,a)$ **+ (1 -** α**)** $Q(s_t,at)$

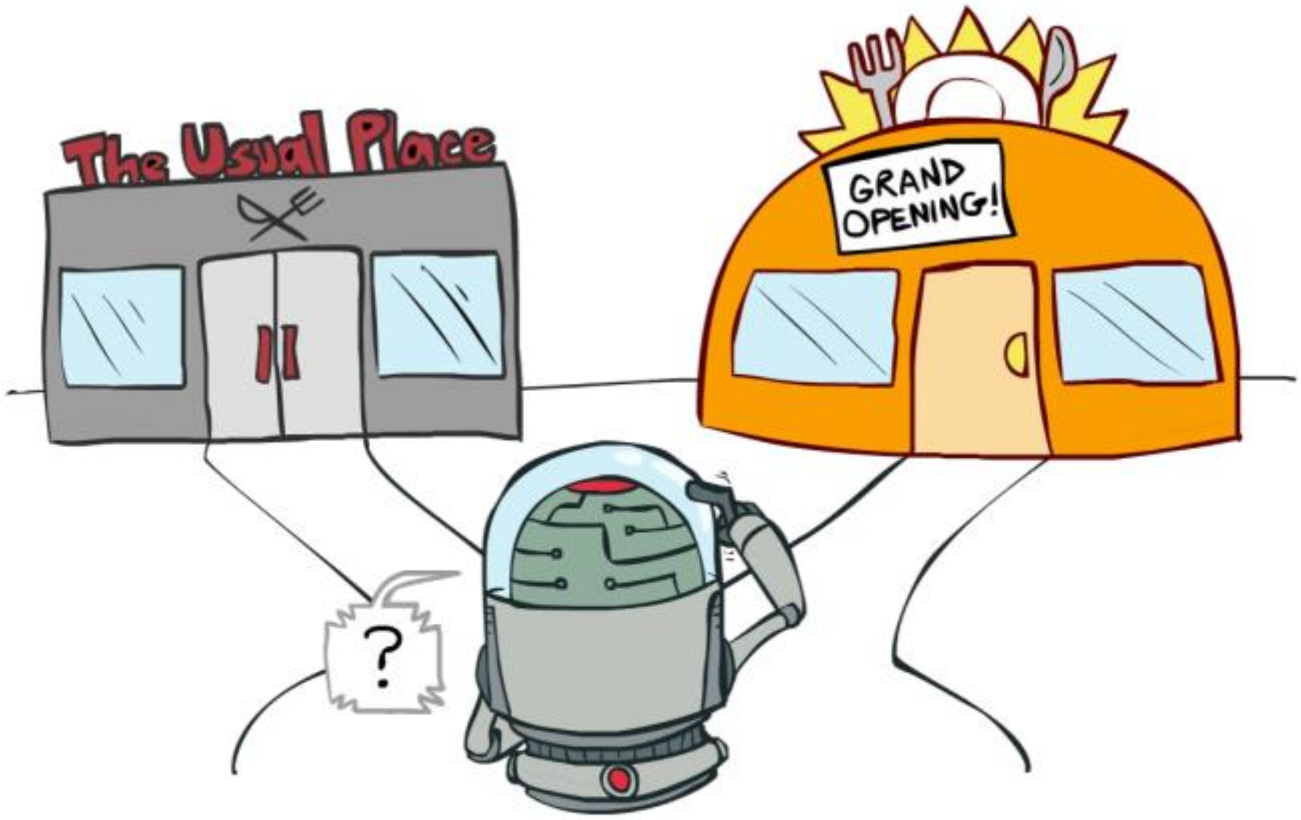
MC vs TD

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

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

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

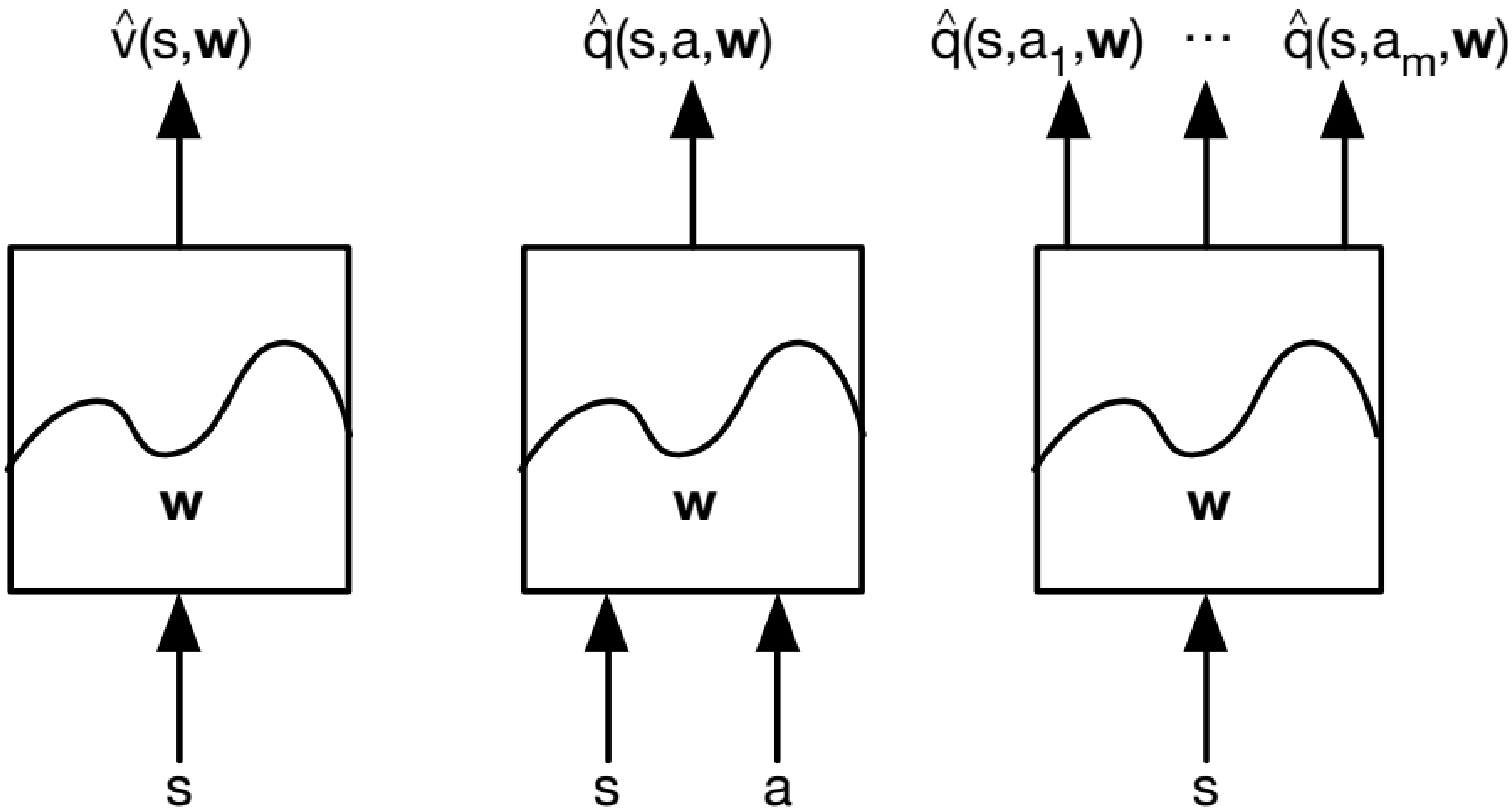
Problem:

- 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



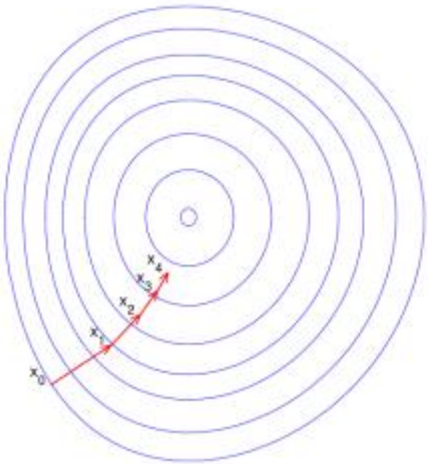
Which class of function to choose?

- 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

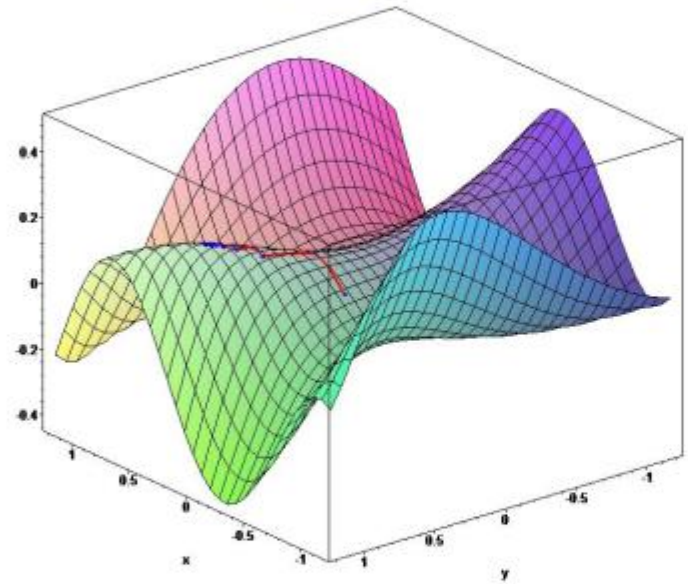
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$



- 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

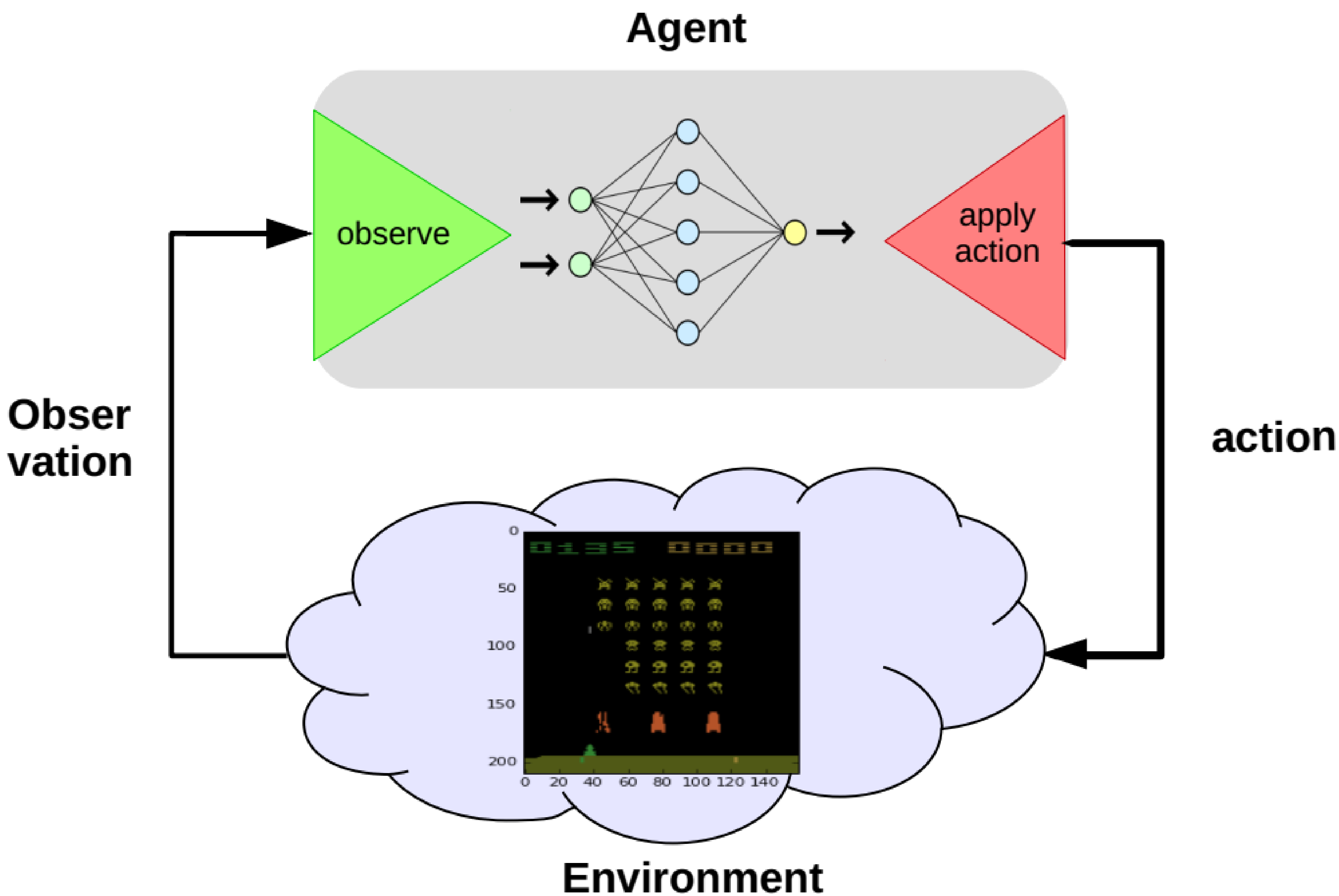
$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right] \end{aligned}$$

- Stochastic gradient descent *samples* the gradient

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

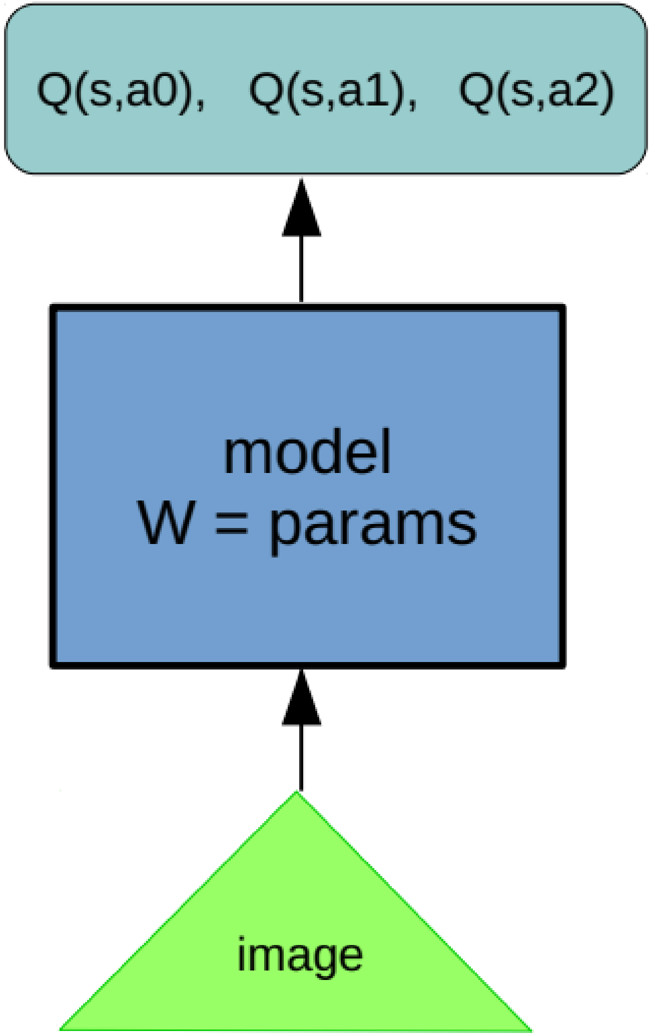
- Expected update is equal to full gradient update

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:

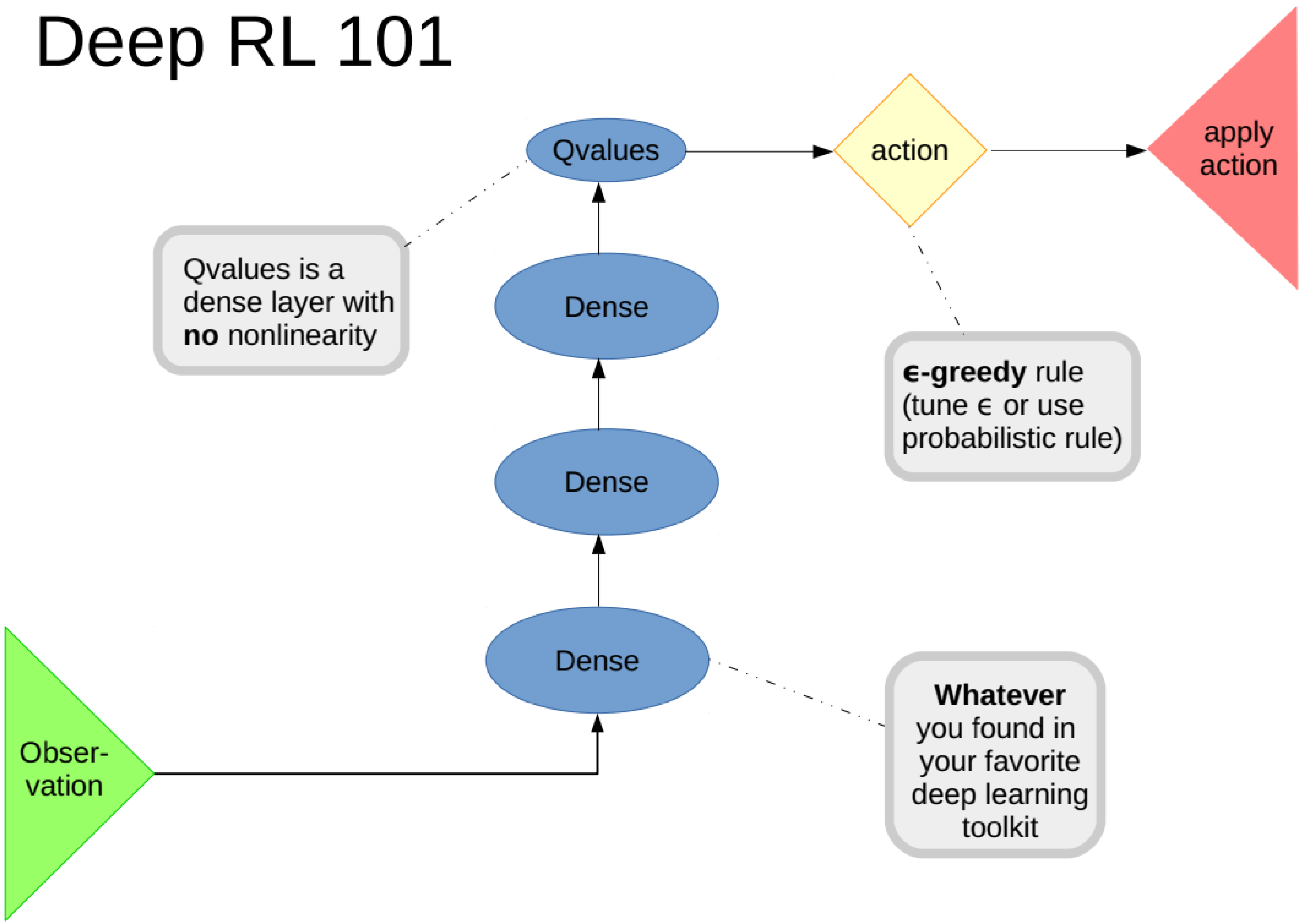
$$\boldsymbol{L} = [\boldsymbol{Q}(\boldsymbol{s_t},\boldsymbol{at}) - \hat{\boldsymbol{Q}}(\boldsymbol{s_t},\boldsymbol{at})]^2$$

  - SGD Update:

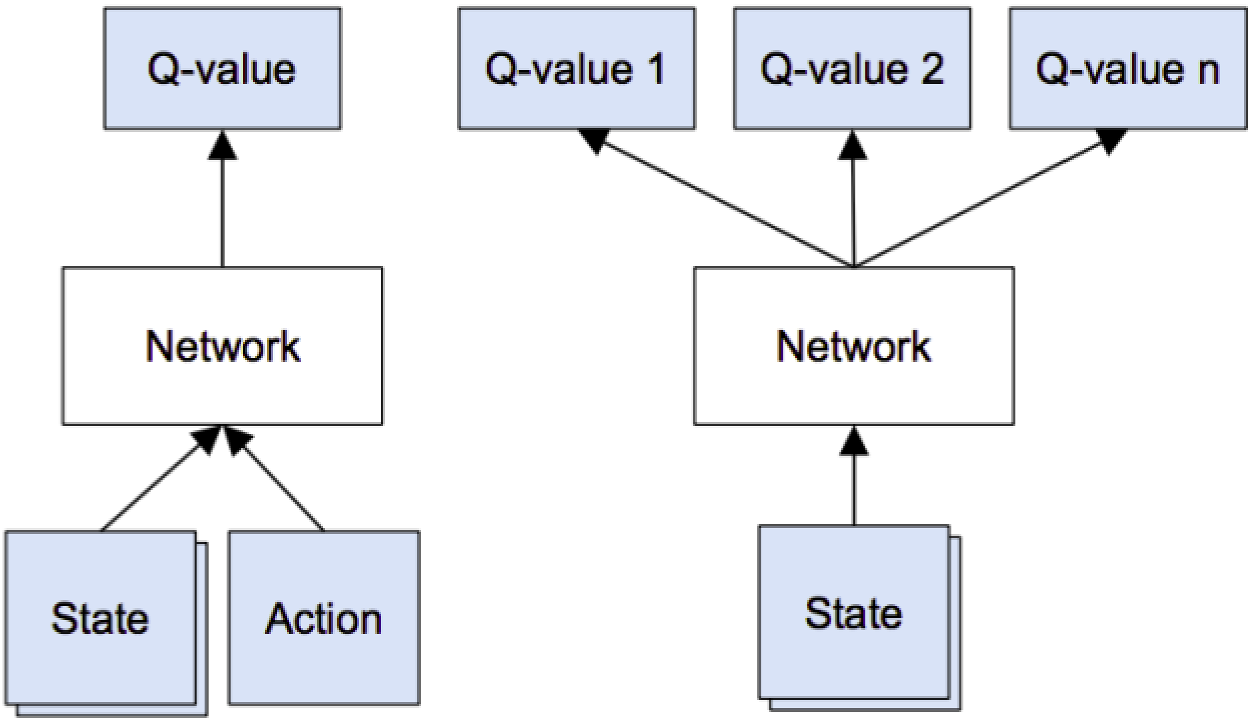
$$W_{t+1} = W_t - \alpha \frac{\partial L}{\partial wt}$$

RL Mechanics

Deep RL 101



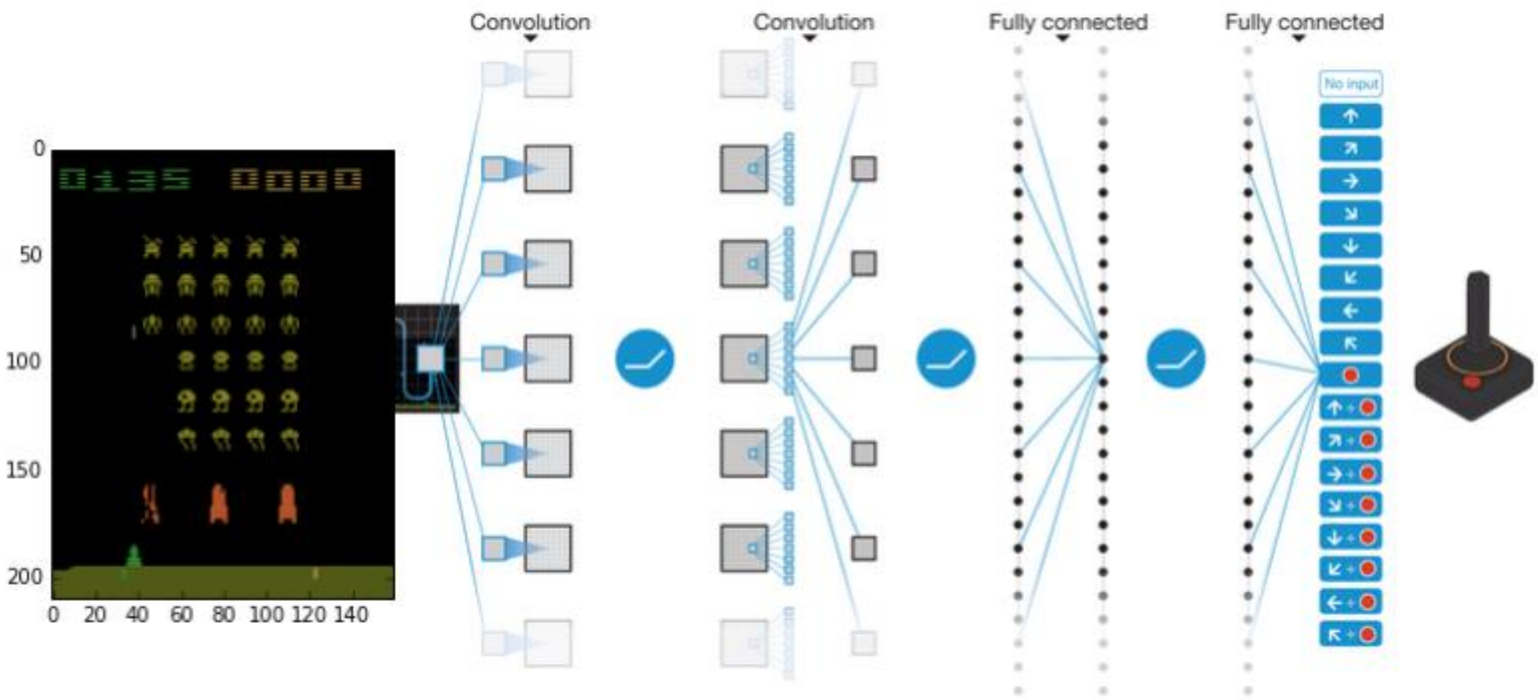
Architectures



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

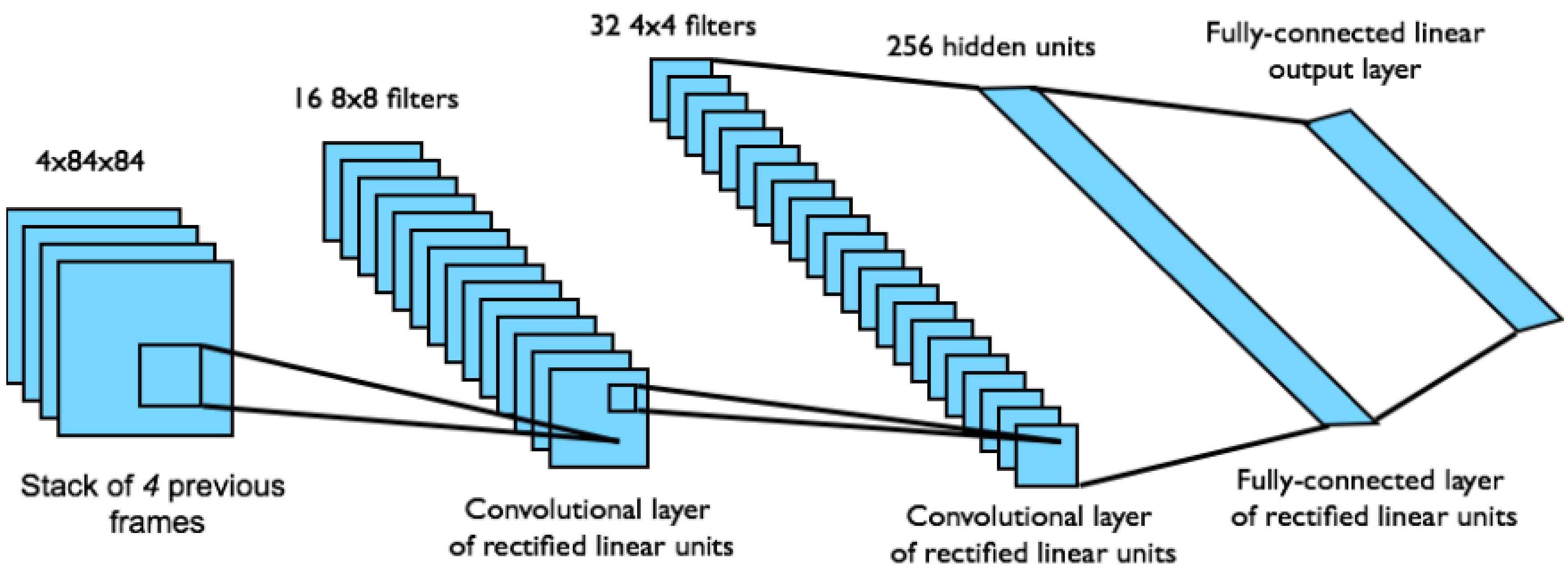
Given **s** predict all q-values
Q(s,a0), Q(s,a1), Q(s,a2)

From theory to practice: DQN case

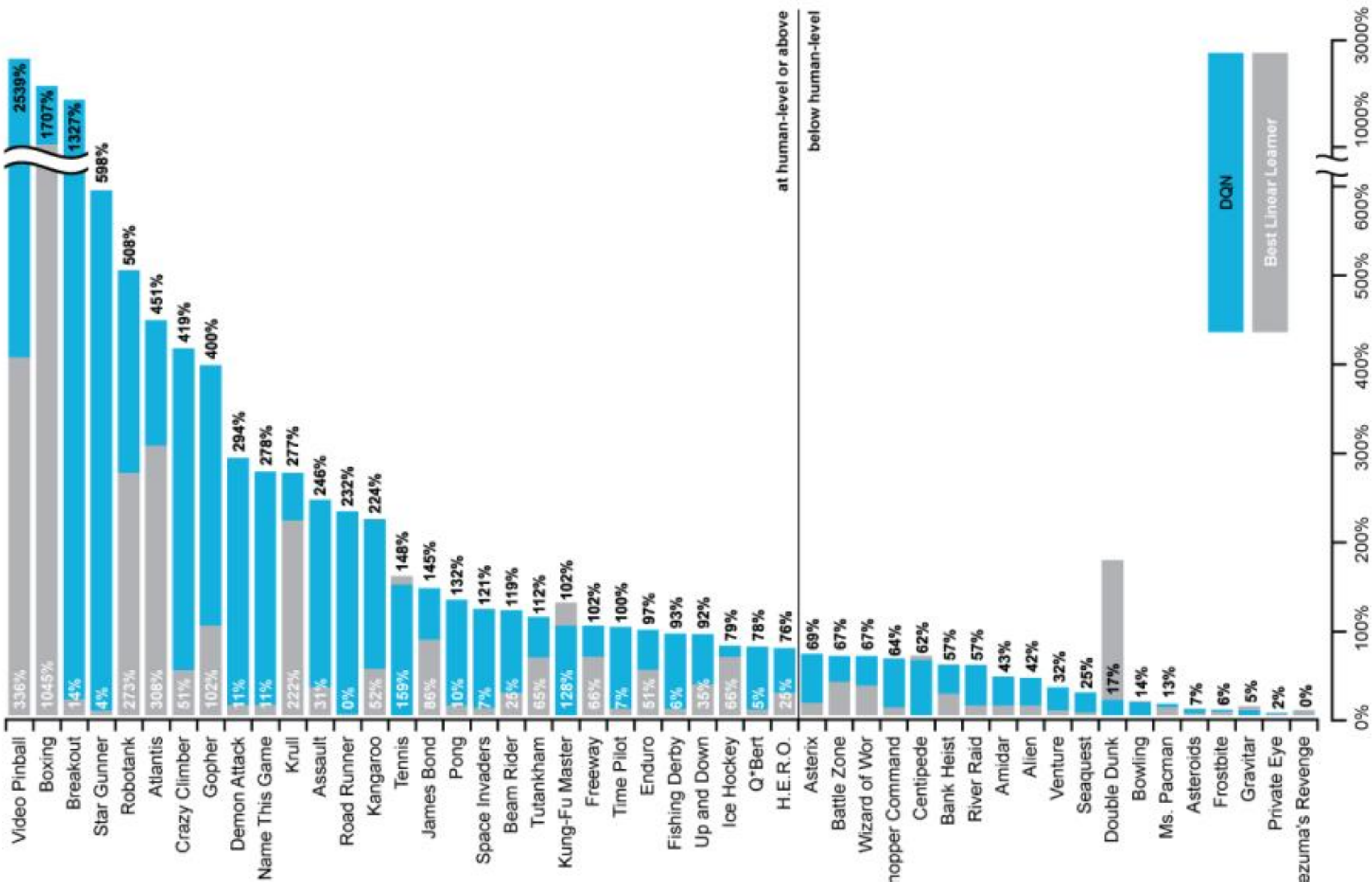


DQN: Atari

- 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

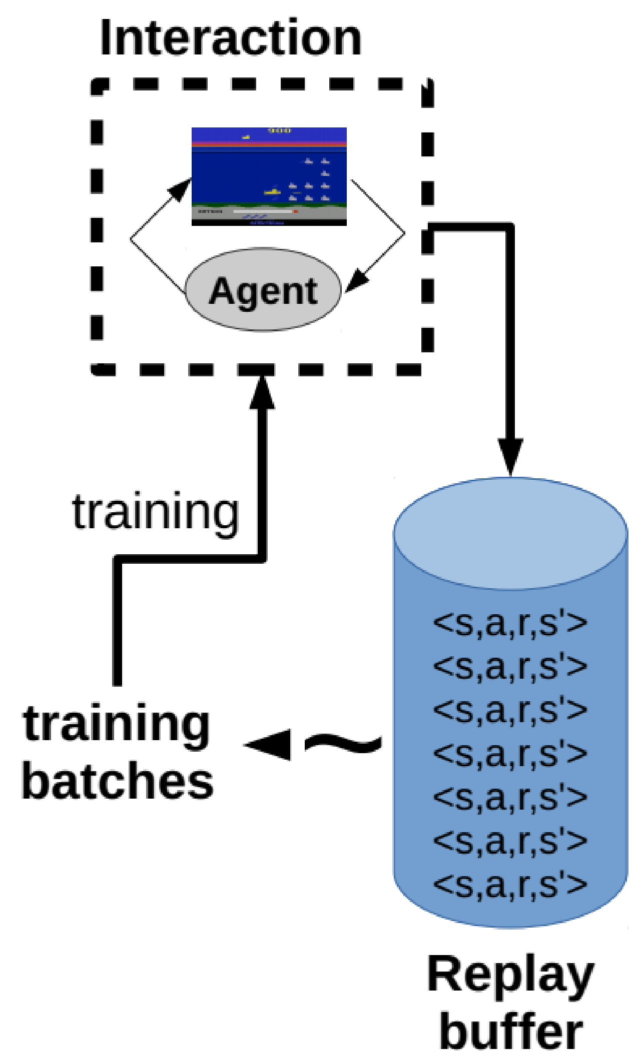
- DQN uses experience replay and fixed Q-targets:
  - Take action $\boldsymbol{a_t}$ according to *e*-greedy policy
  - Store transition $(\boldsymbol{s_t}, \boldsymbol{a_t}, \boldsymbol{r_{t+1}}, \boldsymbol{s_{t+1}})$ in replay memory ***D***
  - Sample random mini-batch of transitions $(\boldsymbol{s},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