Deep RL

CS 175: Projects in AI (in Minecraft)

PROF: SAMEER SINGH

Projects in AI in Minecraft

RL Overview

Tabular Q-Learning

Q-Values

$q(s, a)$: Best possible returns for taking action a in state s

number

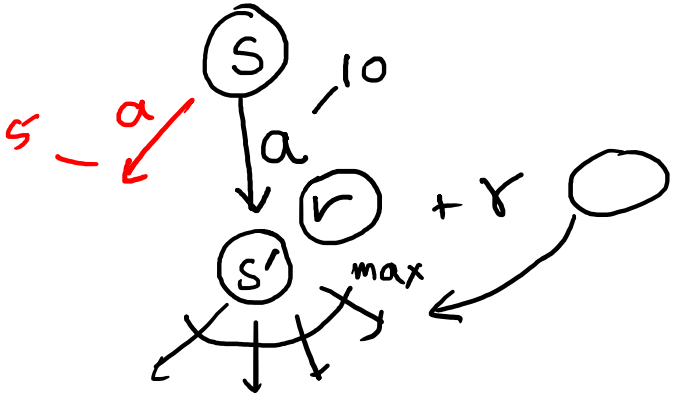
- $q(s, a) = r + \gamma \max_{(a')} q(s', a')$

If you know the q-values, the policy should follow it!

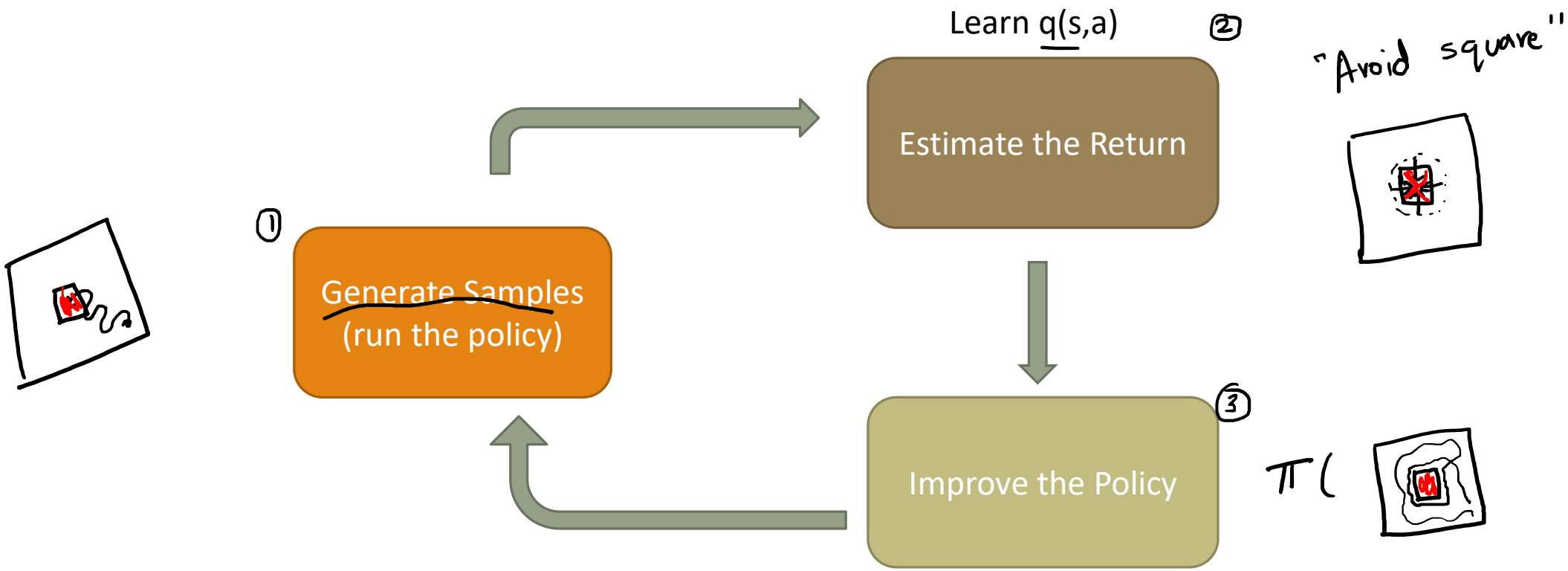
- $\pi(s, a) = \underset{a'}{\operatorname{argmax}}\, q(s, a')$

It's not so simple!

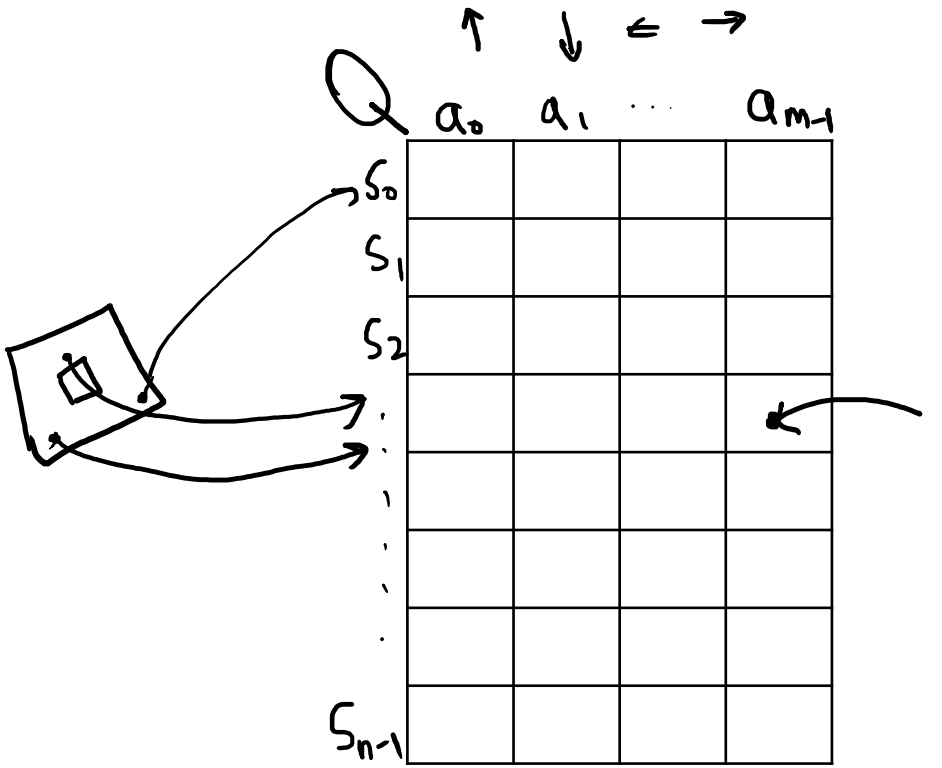
- Q-values need to be estimated from a reasonably good policy
- A policy can be improved from a Q-value!



Value-Function Based



Tabular Q-values



$Q(s,a) \in \mathbb{R}$

States, n

- Rows in the table

Actions, m

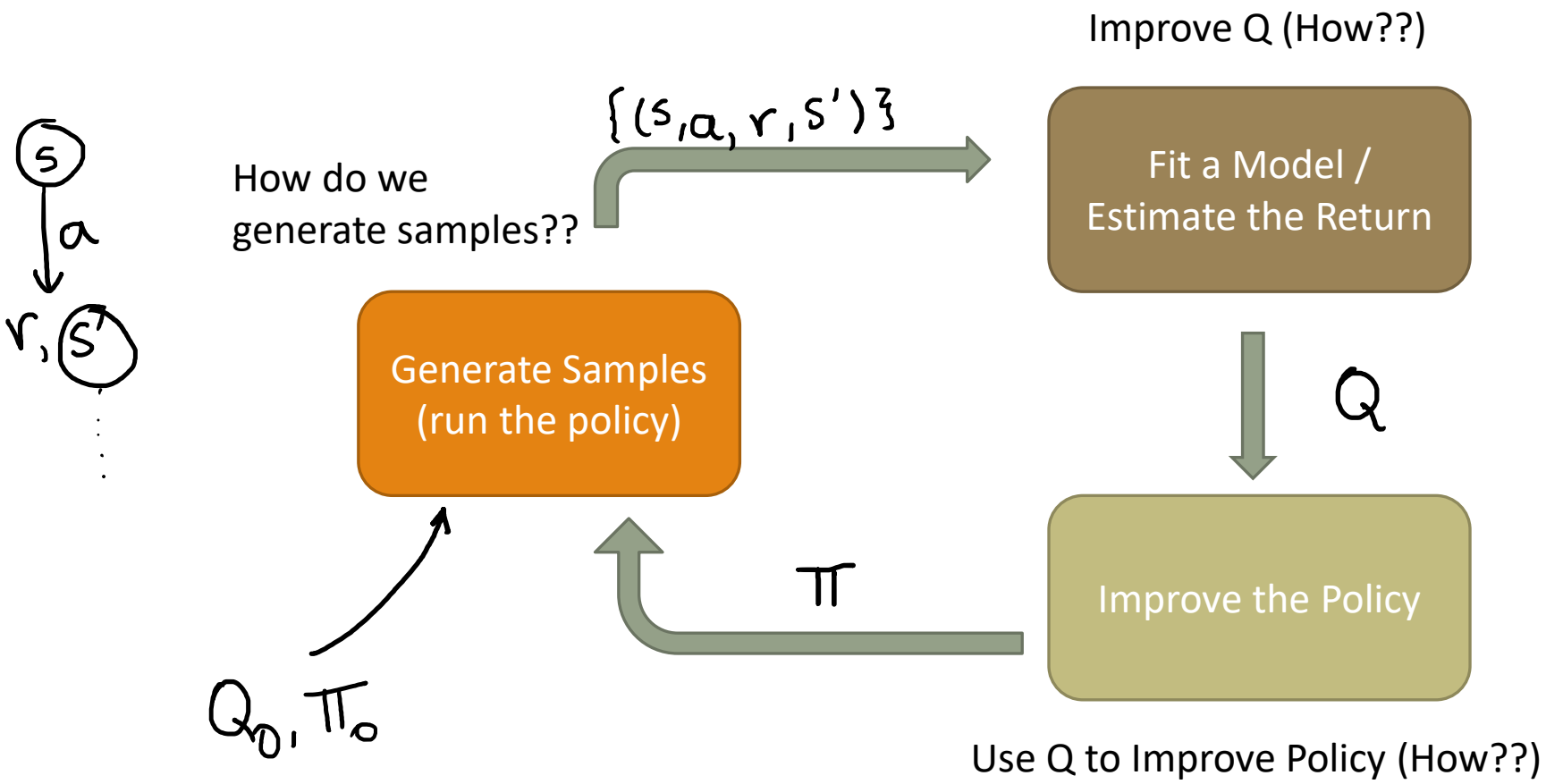
- Columns in the table

Q-values

- Think of it as a "dictionary" / lookup table
- Given a (state, action), returns a value
- Given a state, returns values of all actions

$Q(s) \in \mathbb{R}^m$

Q-Learning Algorithm , $\boxed{Q}$



Improve the Policy

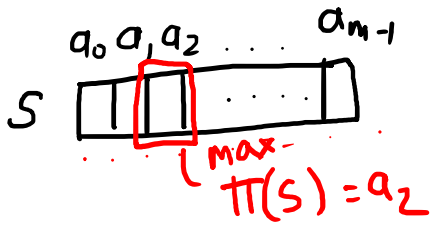
Improving the Policy

Goal: We have current Q-values, how can we use it to improve/get a policy

If Optimal Q-value, Optimal π is Easy

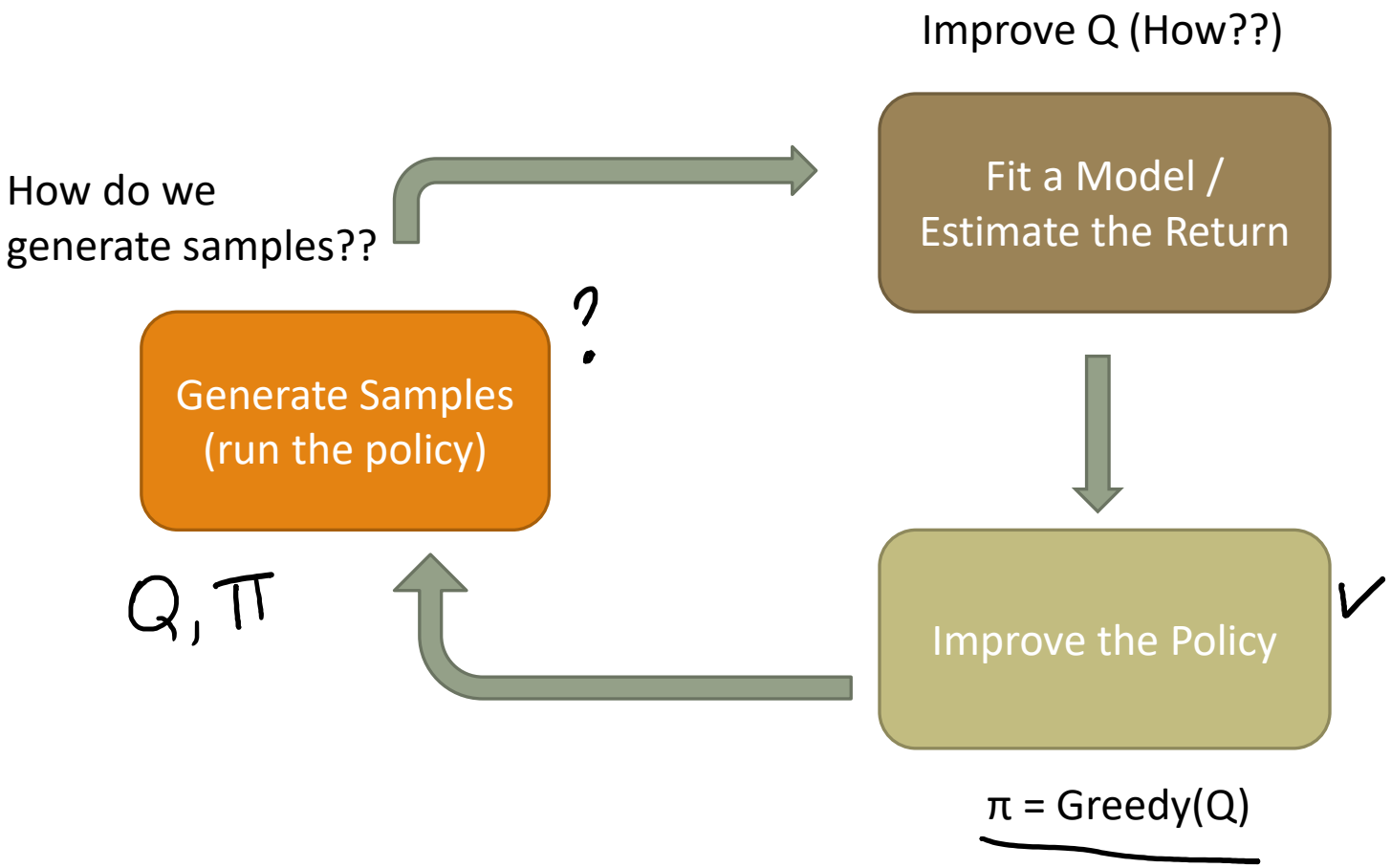
An optimal policy can be found by maximising over $q_*(s, a)$,

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s, a) \\ 0 & \textit{otherwise} \end{cases}$$



$\pi = \text{Greedy}(Q)$

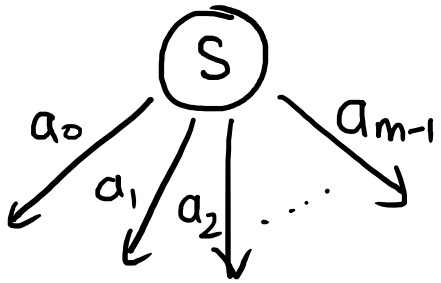
Q-Learning Algorithm



Generate Samples
(run the policy)

Following the Policy

Goal: We need a policy to follow to improve our Q-estimate



$Q(s,a)$

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

Obvious choice: Follow the greedy policy?

∈-Greedy Exploration

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability

0.1

$0 < \epsilon < 1$

ϵ-Greedy Exploration

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability
- With probability $1 - \epsilon$ choose the greedy action
- With probability ϵ choose an action at random

ϵ-Greedy Exploration

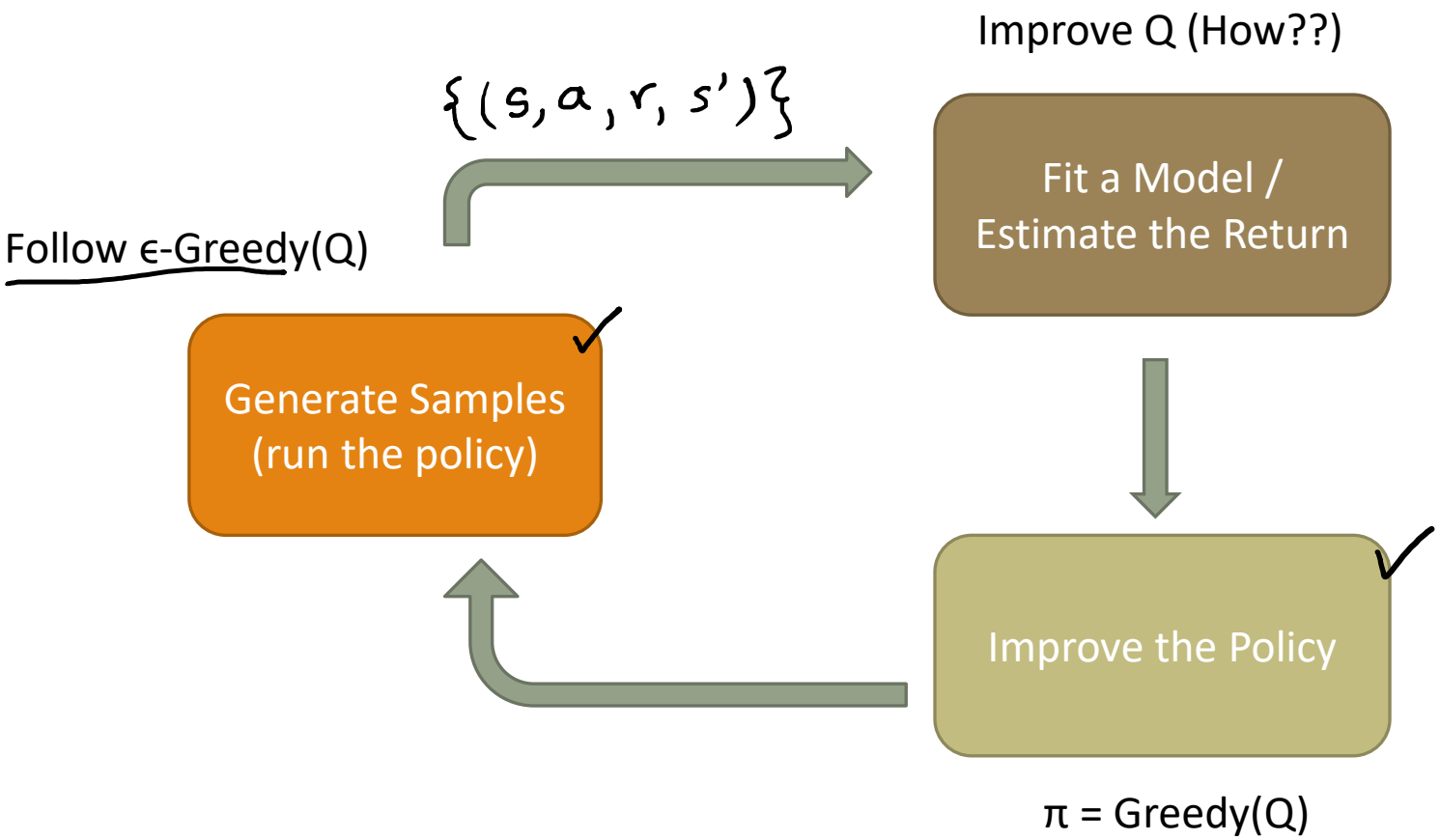
- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability
- With probability $1 - \epsilon$ choose the greedy action
- With probability ϵ choose an action at random

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \operatorname*{argmax}_{a \in \mathcal{A}} Q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

4 actions
a_2 is max

	a_0	$0 + 2.5$
P	a_1	$0 + 2.5$
	a_2	$90 + 2.5$
	a_3	$0 + 2.5$
		10

Q-Learning Algorithm



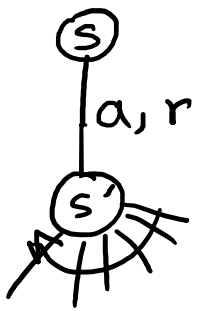
One-step Look-ahead

s, a, r, s'

$\hat{q}$

If you have computed it correctly:

$$q(s,a) = r + \gamma \max_{a'} q(s',a')$$



Current LHS estimate: $\hat{q}(s,a)$

Slightly better estimate: $r + \gamma \max_{a'} \hat{q}(s',a')$

$\hat{q} = 10 + \alpha(-5)$ ($\alpha = 0.1$)

$= 9.5$

Update estimate:

learning rate

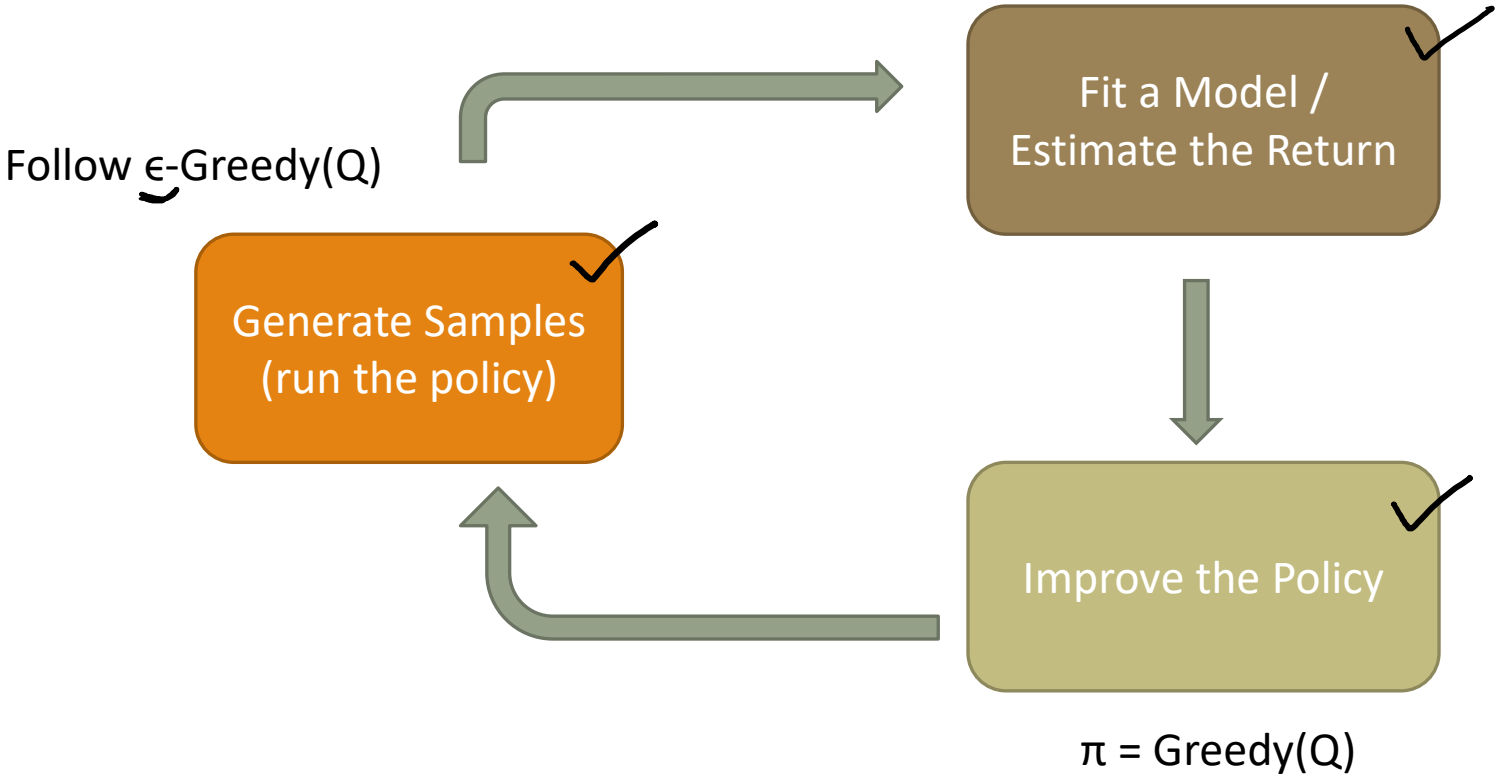
$$\hat{q}(s,a) = \hat{q}(s,a) + \alpha\,(r + \gamma \max_{a'} \hat{q}(s',a') - \hat{q}(s,a))$$

5

10

Q-Learning Algorithm

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$$

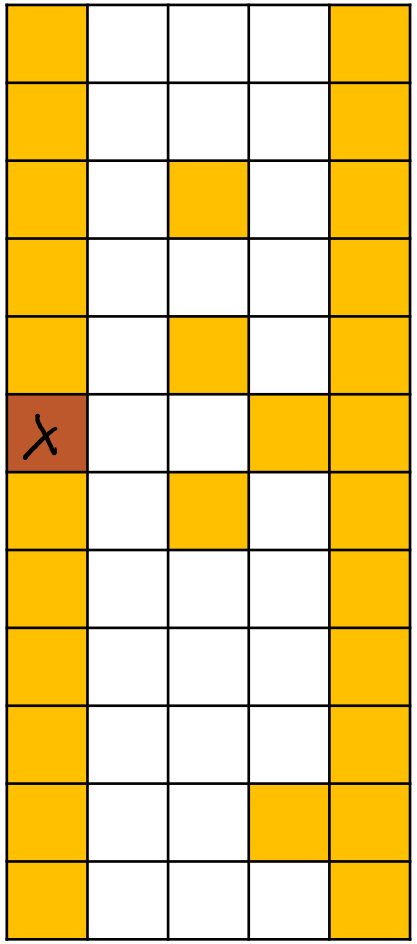


Q-Learning Algorithm

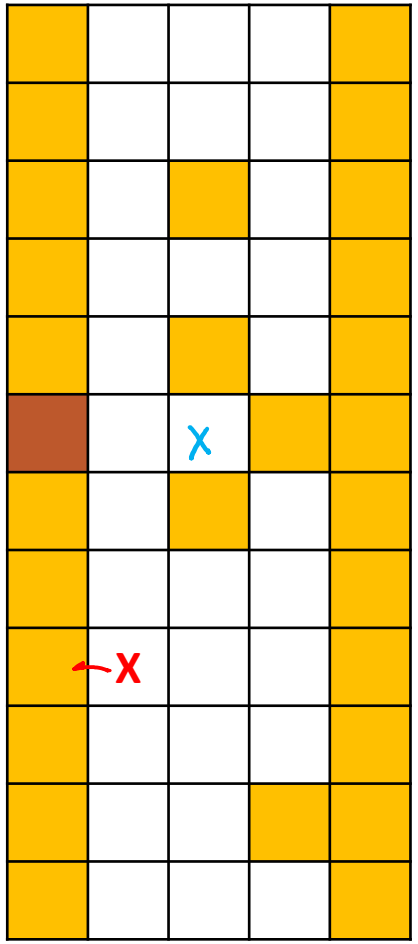
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Task: Navigation (Demo; Tutorial 6)

5x12 grid



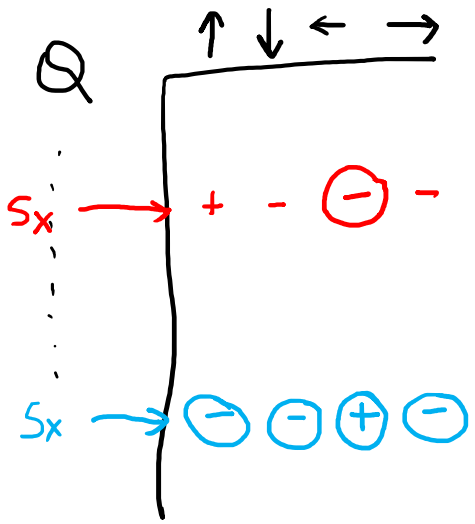
Obvious choice: Cell Position



$S = (i, j)$

$S_x = (3, 1)$

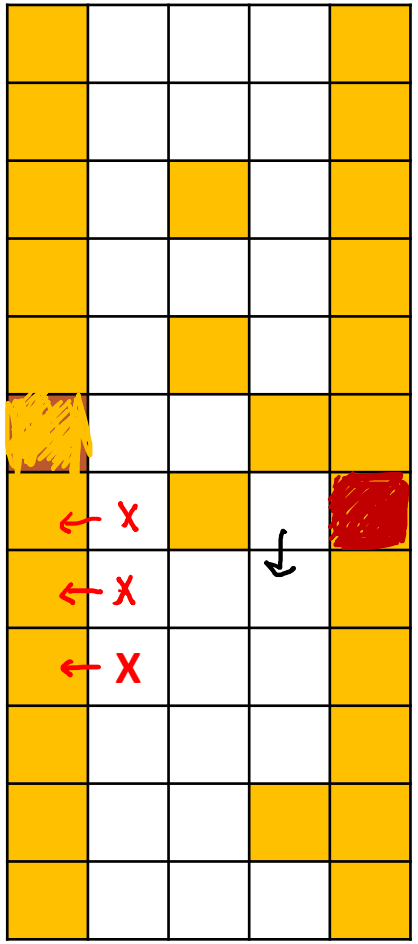
$S_x = (6, 2)$



Positives

- Number of states: 60
- Best action for every cell (eventually, optimal)

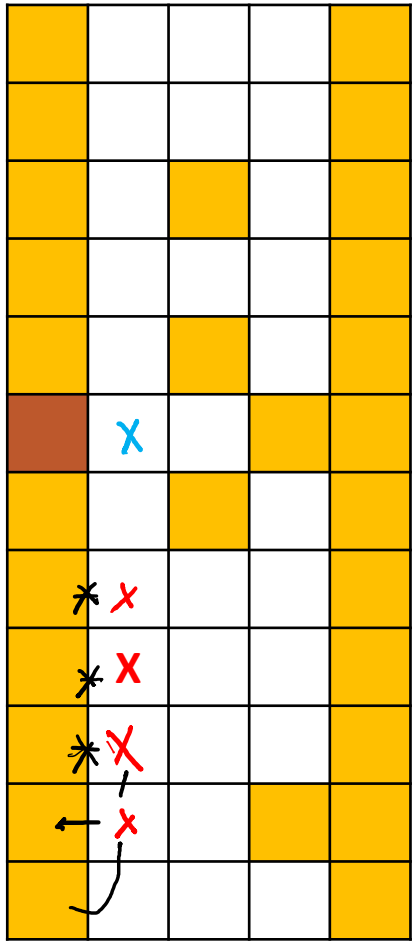
Obvious choice: Cell Position

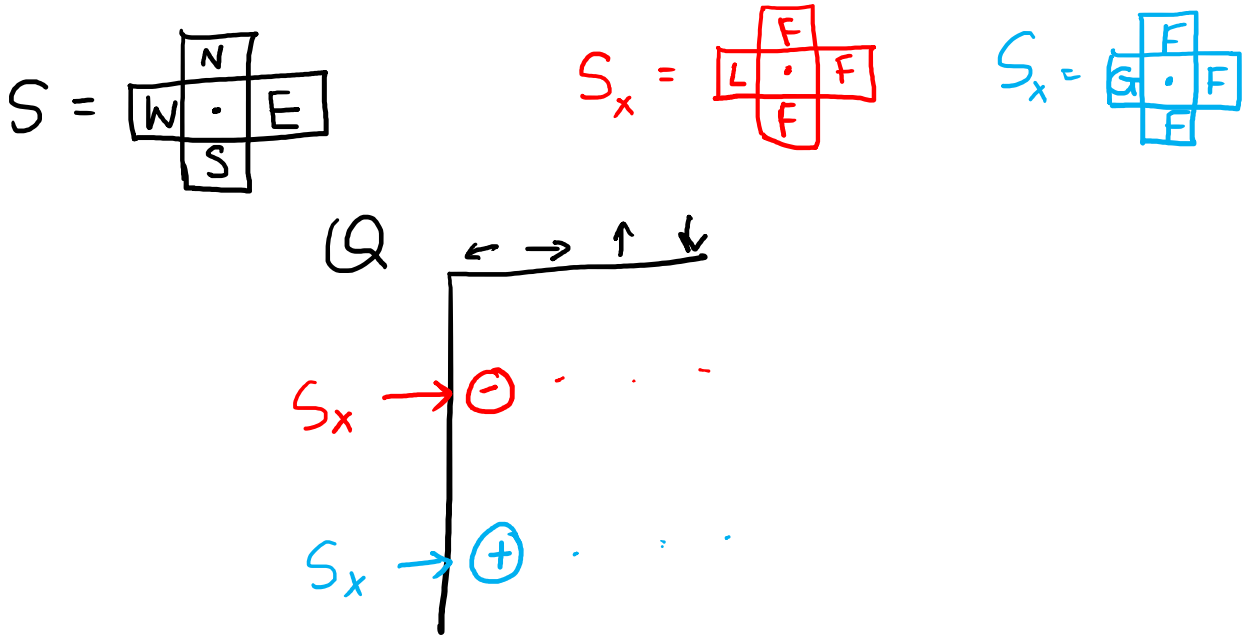


Negatives

- Basically blind!
  - No idea what is in the surrounding cells
  - No idea that lava is bad
- Learning is specific to the maze
  - Needs to learn again if map changes

Alternate choice: Neighbors

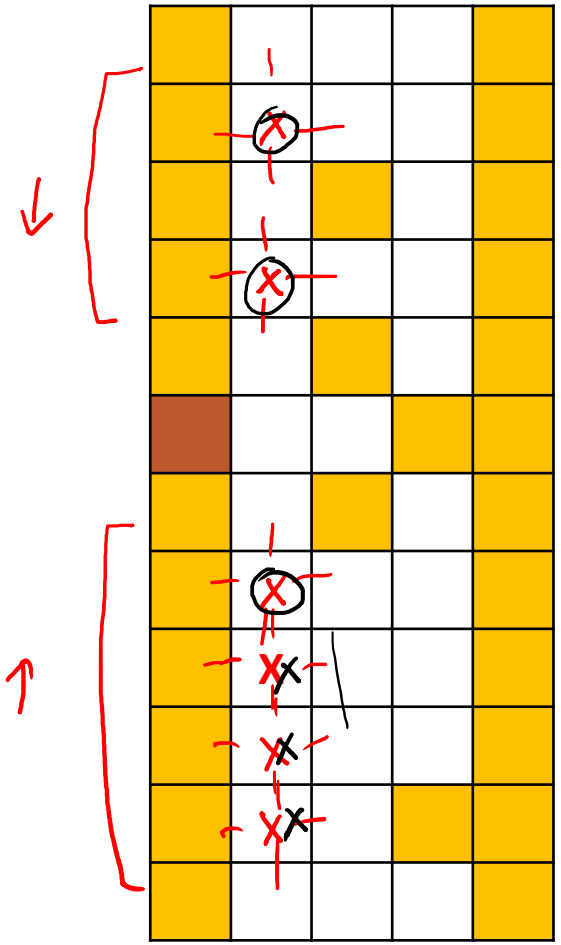




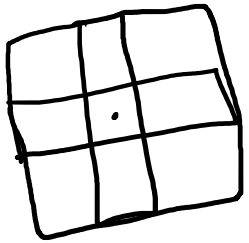
Positives

- Number of states: 2^4 + 4x2^3
- Learns to avoid lava, go towards goal, etc.
- Generalizes, to some degree, across mazes

Alternate choice: Neighbors



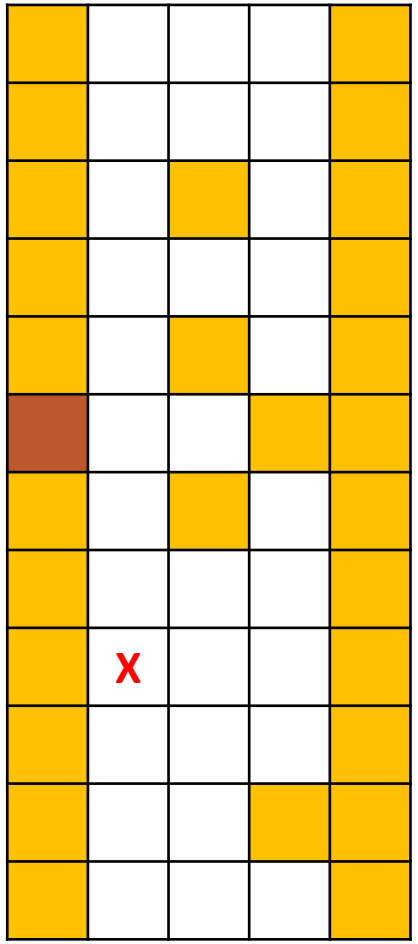
S =

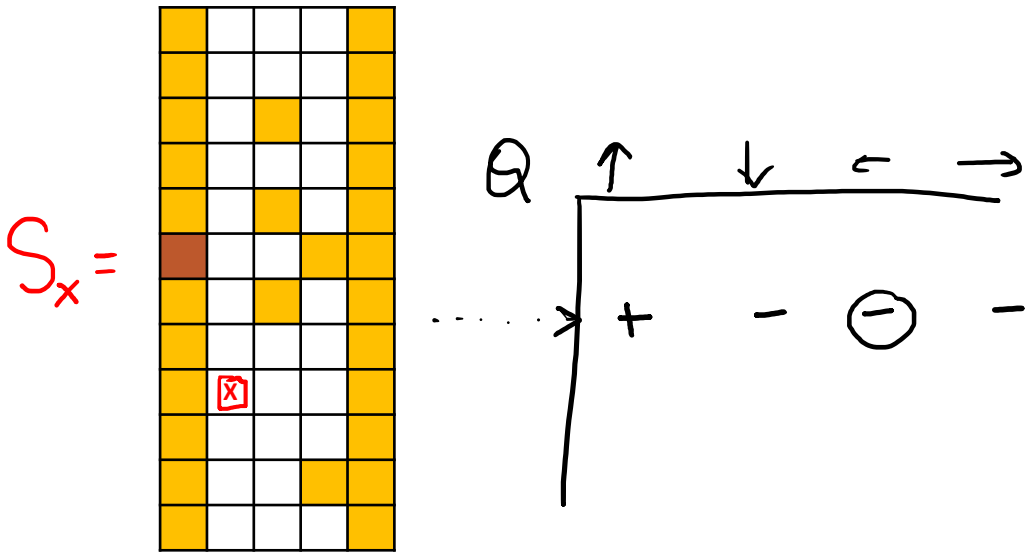


Negatives

- Doesn't know where it is on the maze
  - Policy will be suboptimal, maybe impossible
- Increasing the "view radius" blows up the space
  - 4 cells to 8 would square the no. of states

Why not the full map?

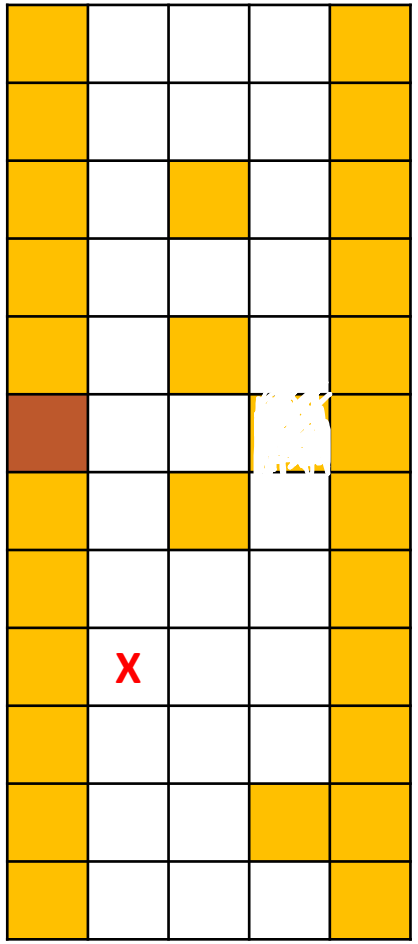




Positives

- Maximum information
- Optimal policy for every map

Why not the full map?



Negatives

- Number of states: ~2^48
- Need to see all possible maps to “learn”
- Basically, impossible

Solution: Non-tabular Q-values

Large-Scale RL

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

- Backgammon: 10^{20} states
- Computer Go: 10^{170} states
- Helicopter: continuous state space

How can we scale up these algorithms to real-world?

Projects in AI in Minecraft

Deep Q-Learning

Policy Gradient

Example: Atari

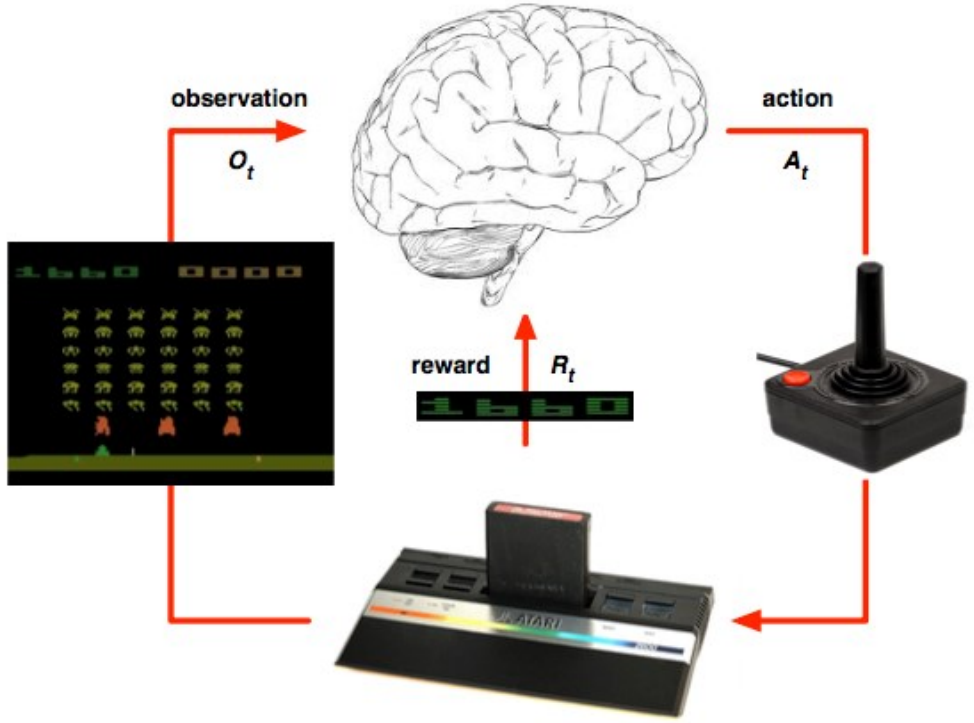
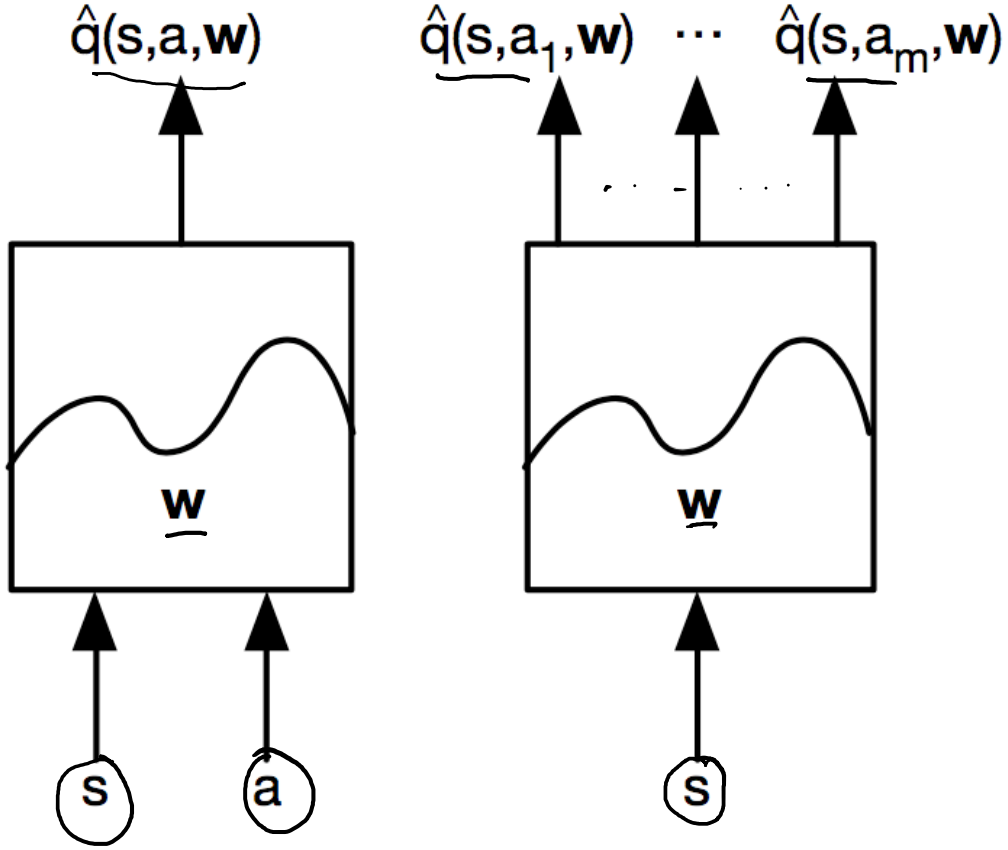


Image size: 84x84
Consecutive: 4 frames
Grayscale with 256 colors

$256^{84x84x4}$ rows in the Q-table
~ $10^{69,970}$

Let's not keep so many values!



$$Q(s,a) \to R$$

$$\hat{Q}_w(s,a) \to R$$

$84 \times 84 \times 4$

Which Approximator?

There are many function approximators, e.g.

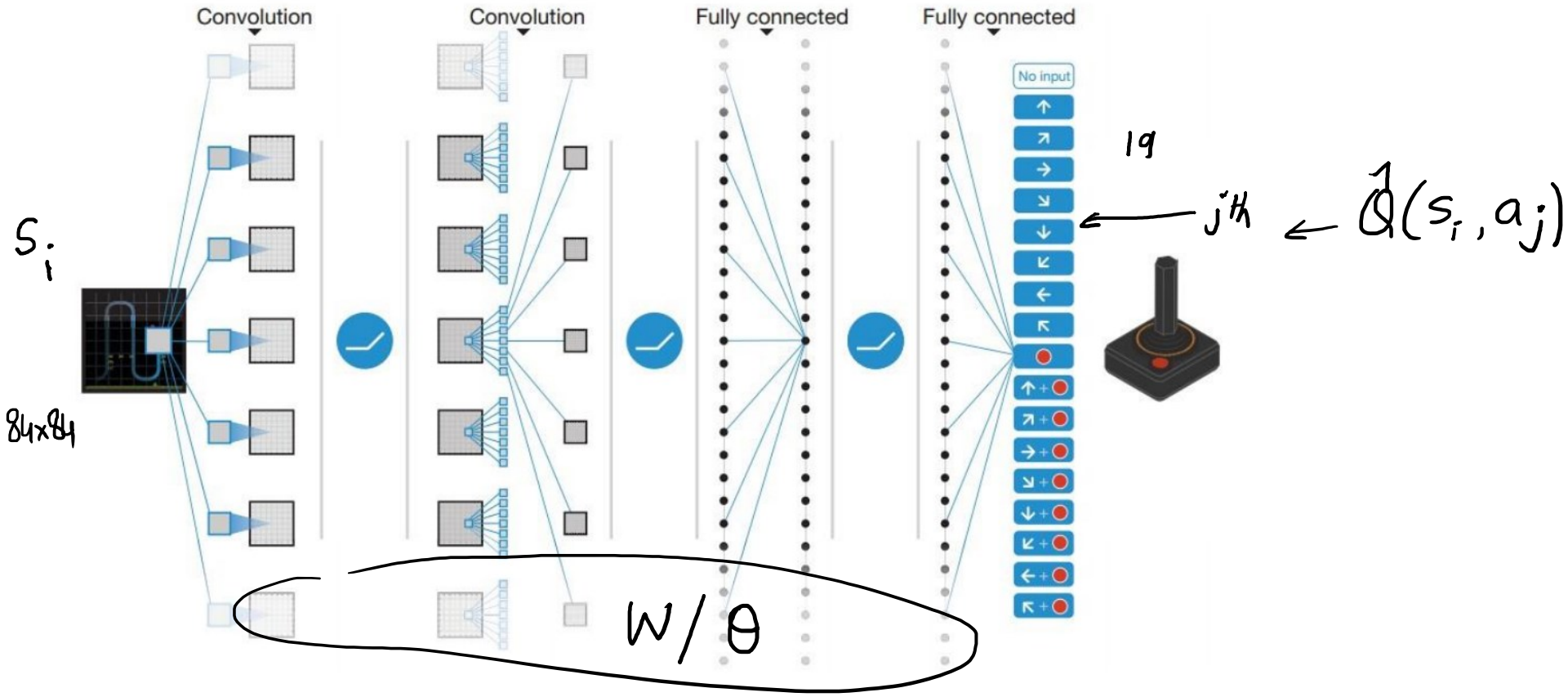
- Linear combinations of features $\longrightarrow$ $w \cdot \phi(s, a)$ (features)
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Which Approximator?

We consider differentiable function approximators, e.g.

- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

Atari Deep Q-Network



Deep Q-Network Algorithm

LR reward

$$\underline{Q(S,A)} \leftarrow \underline{Q(S,A)} + \alpha\left[\underbrace{R + \gamma \max_a Q(S',a)}_{\text{new estimate}} - Q(S,A)\right]$$

$$\min_w J(w) = \left(Q_w(s,a) - \left[R + \gamma \max_a Q_w(s',a)\right]\right)^2$$

$$\nabla_w J(w)$$

$(s, a, r, s') + Q_w$

Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N

Initialize action-value function Q with random weights θ θ^- θ^a

for episode $= 1, M$ **do**

Initialise state s_t

for $t = 1, T$ **do** , $\theta^- \leftarrow \theta$

With probability ϵ select a random action a_t

otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$ θ^a — ϵ-greedy

$(s, a, r, s')_t$

Execute action a_t and observe reward r_t and state s_{t+1}

Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$

Set $s_{t+1} = s_t$

Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$

Set $y_j = \begin{cases} r_j & \text{for terminal } s_{t+1} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{for non-terminal } s_{t+1} \end{cases}$ θ^-

Perform a gradient descent step on $(y_j - Q(s_t, a_j; \theta))^2$

end for

end for

https://towardsdatascience.com/self-learning-ai-agents-part-ii-deep-q-learning-b5ac60c3f47

Stabilizing Deep Q-Learning

- Changing the value of one action will change the value of other actions and similar states.
- The network can end up chasing its own tail because of bootstrapping.

$\theta_i^- \leftarrow \theta_i$

$$Q(s,a,\theta_i) \leftarrow \left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i) \right)^2$$

θ_i^a

ϵ-greedy

"Double Q Learning" takes it a step further!
Action selected using one, evaluated using the other

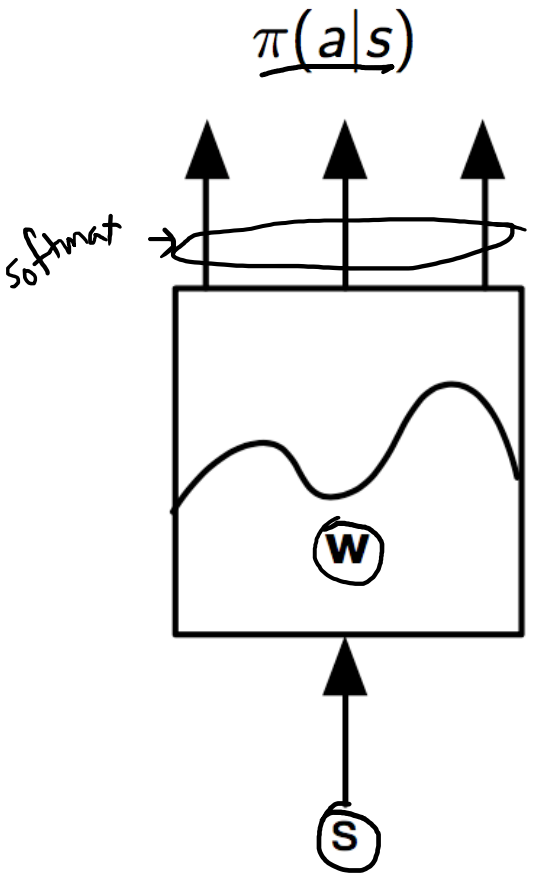
Projects in AI in Minecraft

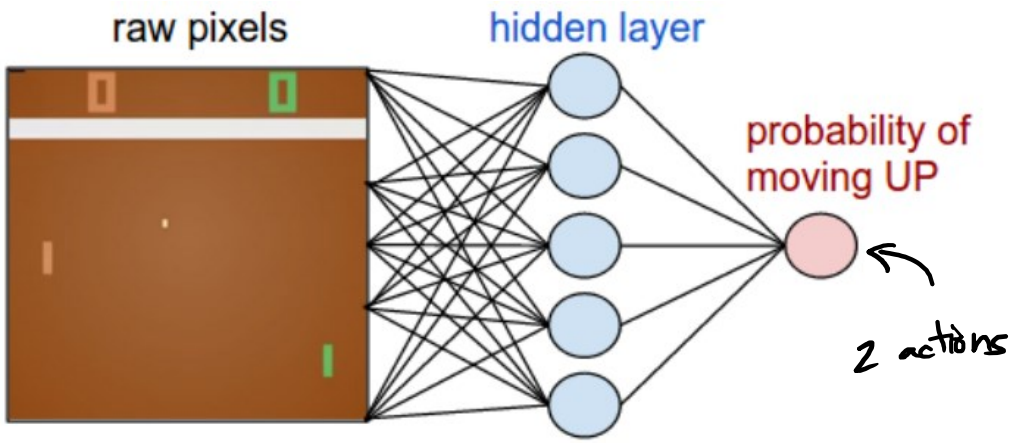
Deep Q-Learning

Policy Gradient

$\pi(a|s) = \epsilon\text{Greedy}(Q) = \max_a Q(s,a)$

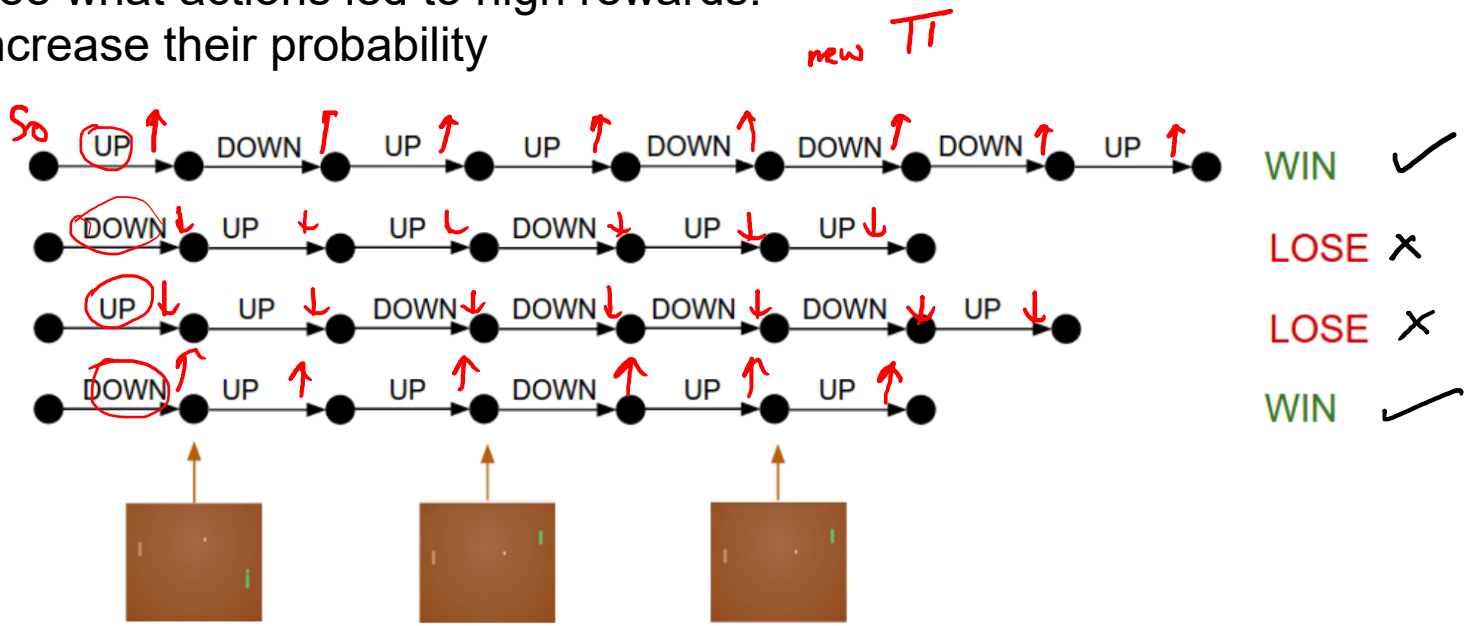
Policy Networks





Policy Gradient (Training)

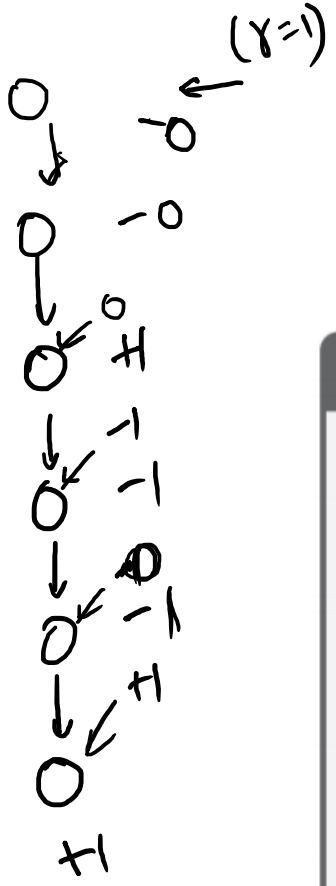
- Run a policy for a while.
- See what actions led to high rewards.
- Increase their probability



Needs more data, slower/unstable convergence

https://medium.com/@thechrisyoon/deriving-policy-gradients-and-implementing-reinforce-f887949bd63

Policy Gradient



$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} r_{t+1} | \pi_\theta]$$

total reward

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log \pi_\theta(a_t|s_t) G_t$$

total return

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$

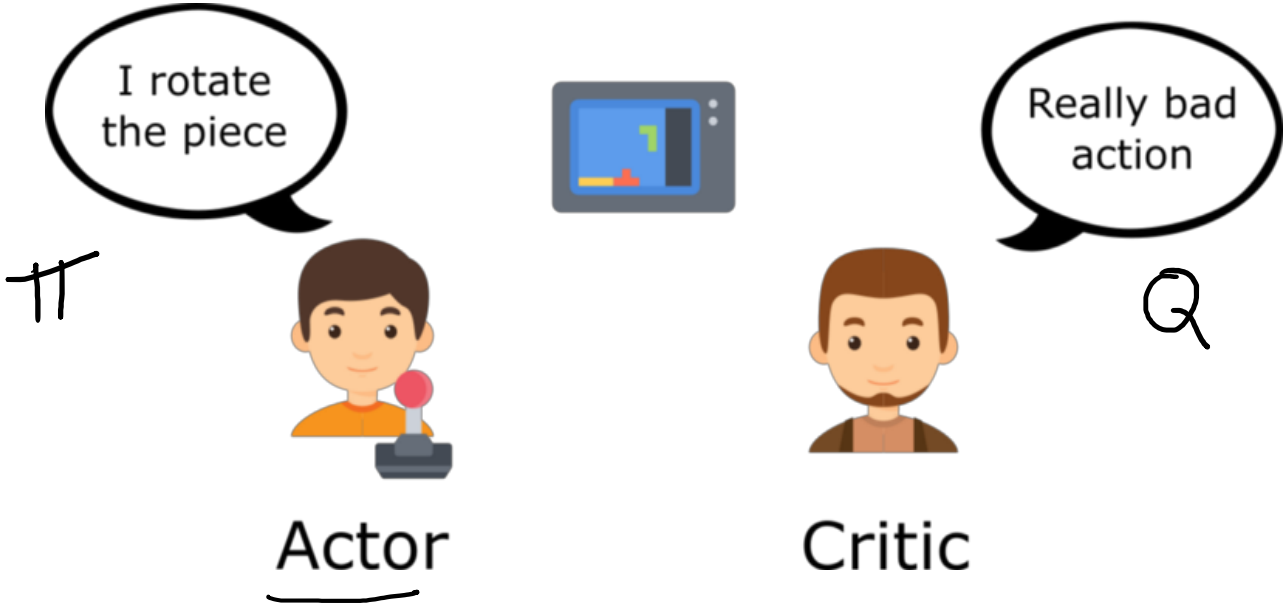
s,a,r,s'

  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\qquad (G_t)$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Actor Critic Policy Gradient

States look good/bad by chance, leading to slow convergence
Let's estimate the Q-values using another network!

Critic



$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(a_t|s_t)G_t$$

P.G.

Total return

Actor Critic Policy Gradient

States look good/bad by chance, leading to slow convergence
Let's estimate the Q-values using another network!

Critic

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) Q_w(s_t, a_t)$$

The "Critic" estimates the Q value function.

→ Updating Q looks a lot like Q-Learning

- One-step look-ahead, and gradient step to update Q

The "Actor" updates the policy as suggested by the Critic

→ Policy gradient, but use Q instead of reward

Actor Critic Policy Gradient

Algorithm 1 Q Actor Critic

Initialize parameters s, θ, w and learning rates α_θ, α_w; sample $a \sim \pi_\theta(a|s)$.
for $t = 1 \ldots T$: **do**
  Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$
  Then sample the next action $a' \sim \pi_\theta(a'|s')$
  Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \log \pi_\theta(a|s)$; Compute the correction (TD error) for action-value at time t:
    $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
  and use it to update the parameters of Q function:
    $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
  Move to a $\leftarrow a'$ and s $\leftarrow s'$
end for

q-learning

https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html