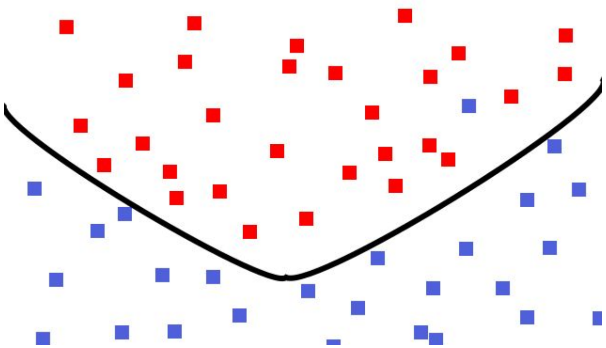
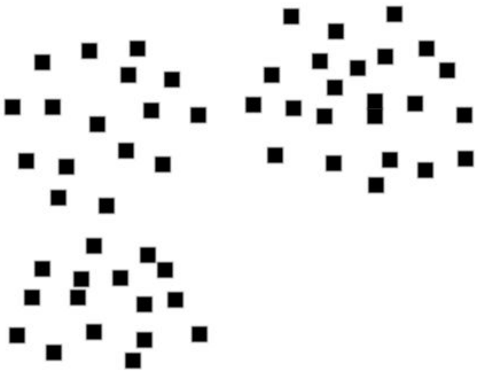
Reinforcement Learning

Slides by Ekapol Chuangsuwanich and
Nat Dilokthanakul, researcher at VISTEC

3 Modes of Learning

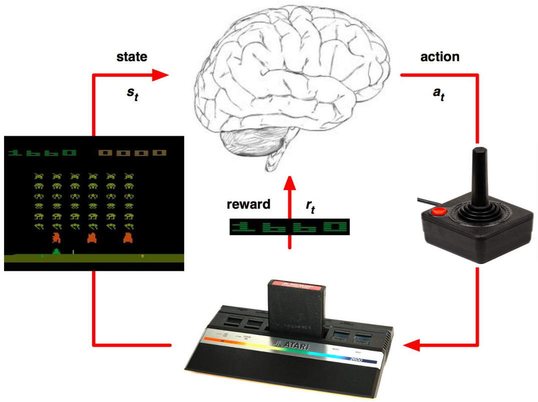




Supervised Learning

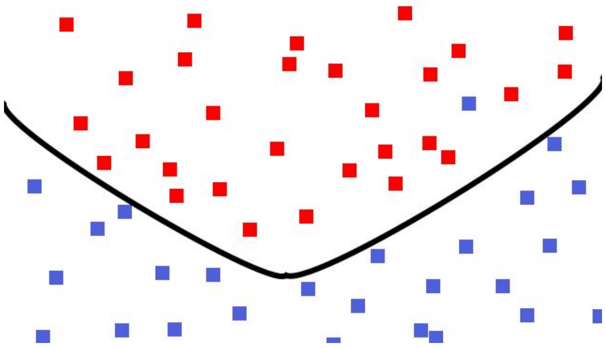
Reinforcement Learning

Unsupervised Learning



3 Modes of Learning

Supervised Learning

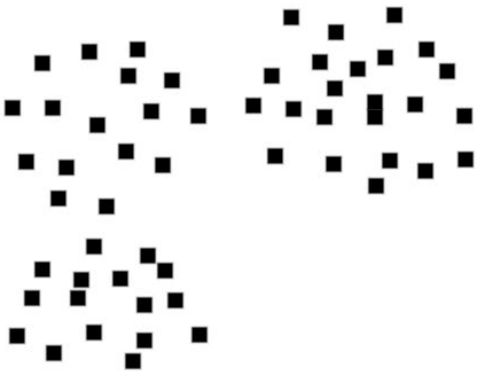


- Observe:
  - $(x_1, y_1), (x_2, y_2), \ldots$
- Objective:
  - Input an unseen x_{new}
  - What is y_{new}?

3 Modes of Learning

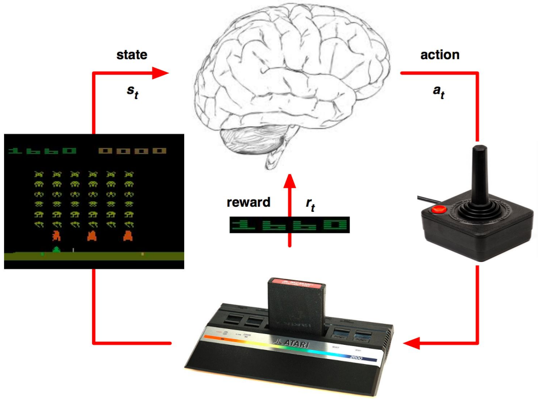
Unsupervised Learning

- Observe:
  - $x_1, x_2, x_3, x_4, \ldots$
- Objective:
  - What is P(x) ?
  - What is a *good* representation of x ?
  - What can we learn from P(x) ?



3 Modes of Learning

Reinforcement Learning (RL)

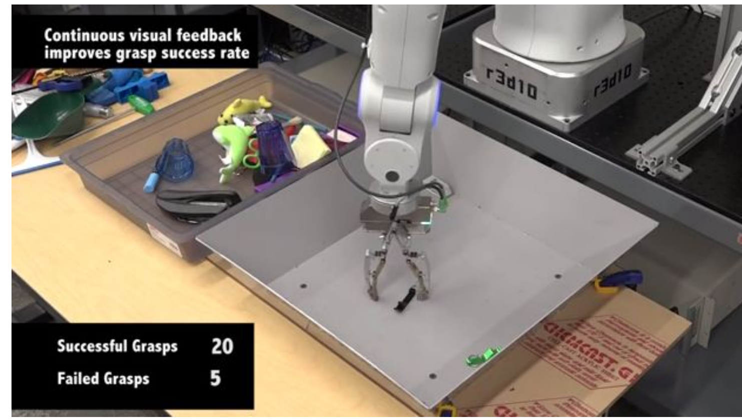


- Observe:
  - The states (x_1, x_2, x_3, …)
  - The reward (r_1, r_2, r_3, …)
- Can also take actions
  - a_1 , a_2 , a_3 , …
- What are the best actions?
  - Such that we will receive highest accumulative rewards

Applications

https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/

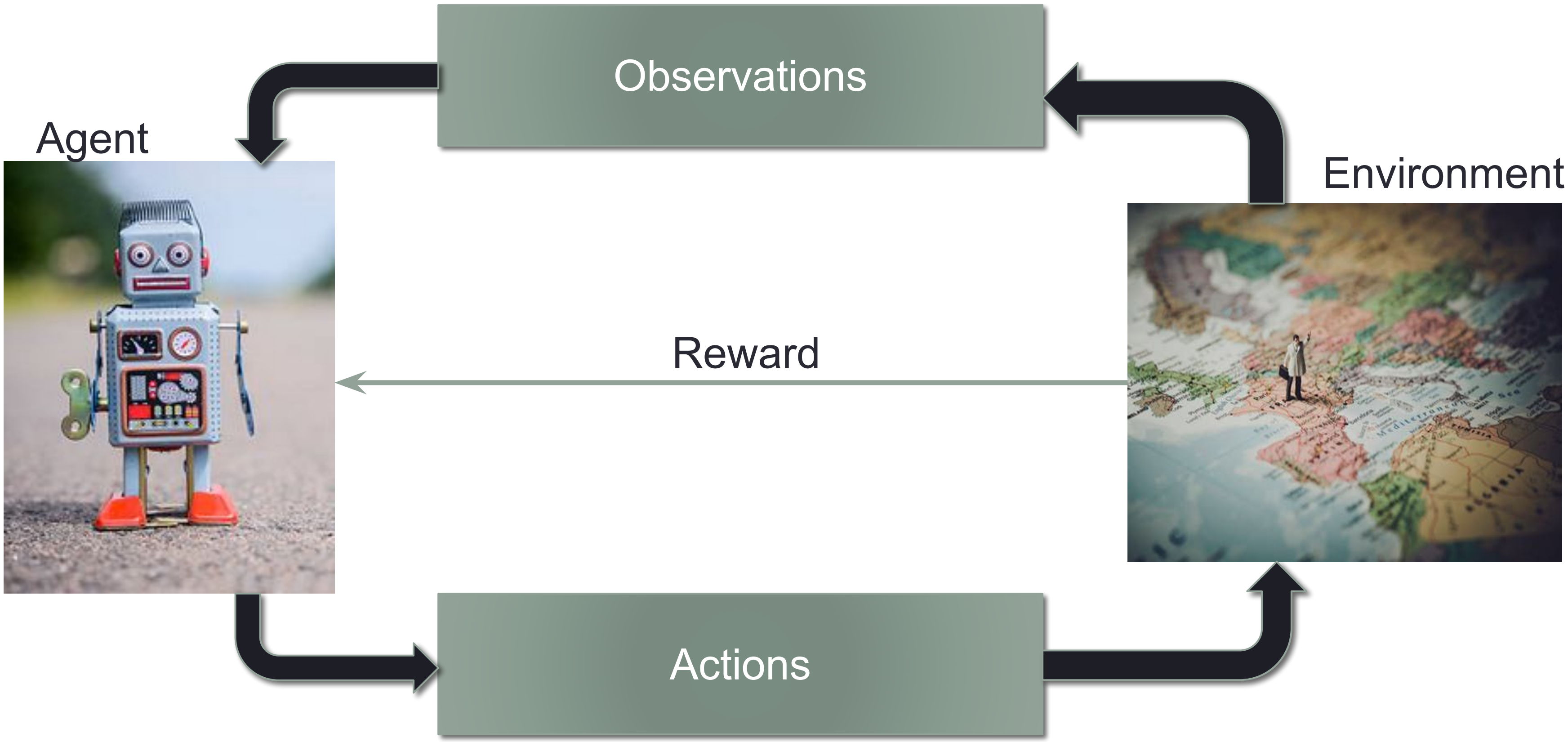
- Robotic
- Games
- Cooling system
- Autonomous vehicle
- etc.



https://research.googleblog.com/2016/03/deep-learning-for-robots-learning-fro

http://spectrum.ieee.org/automaton/robotics/drones/drone-uses-ai-and-11500-crashes-to-learn-how-to-fly

RL framework



Learning through trial and error

RL framework

Reward (r_t)

State (s_t)

Action (a_t)

$$\begin{tikzcd}
{} \arrow[r] & \text{Env.} \arrow[r, "s_t"] \arrow[d] & \text{Agent} \arrow[r, "a_t"] & \text{Env} \arrow[r, "s_{t+1}"] \arrow[d] & \text{Agent} \arrow[r, "a_{t+1}"] & \text{Env} \arrow[r, "s_{t+2}"] \arrow[d] & {} \\
& r_t & & r_{t+1} & & r_{t+2} &
\end{tikzcd}$$

Markov property

$$a_t = A(s_t)$$

$$r_{t+1} = R(s_t, a_t)$$

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

Rewards-based learning

- Maximise the rewards
- Can we design any desired behaviour with reward?

R_t = Δdistance

R_t = score

$$R_T = \begin{cases} 1, & \text{win} \\ -1, & \text{lose} \end{cases}$$

The Environment

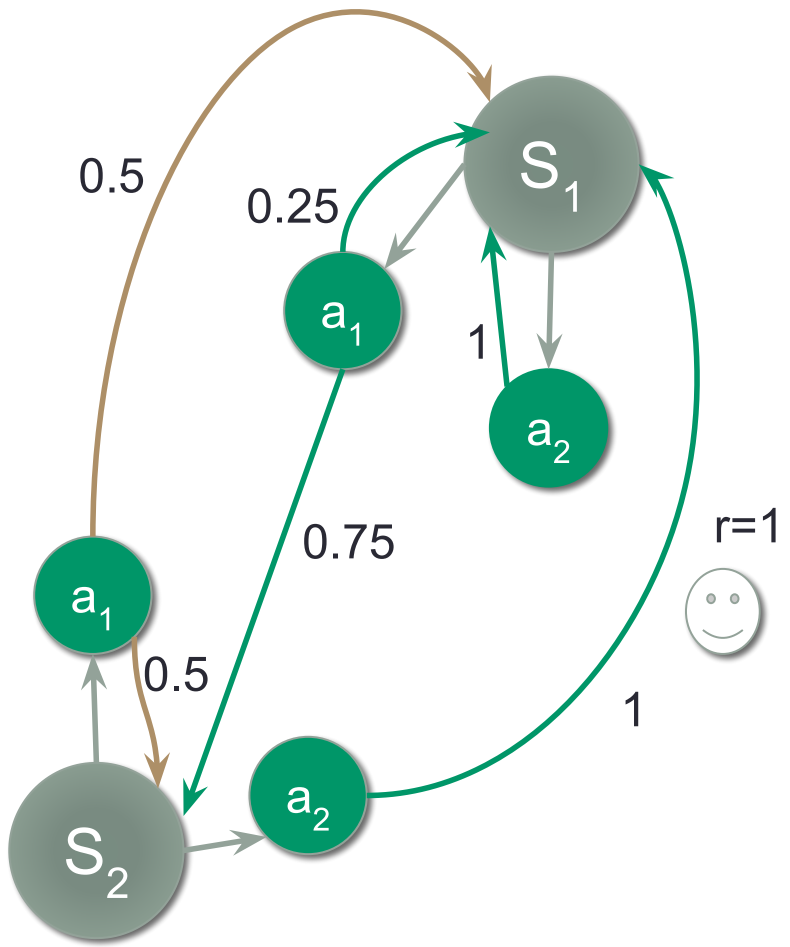
How can we model the environment?

Markov Decision Process (MDP)

- **S,A,P,R**,γ
- **S** – Set of states
- **A** – Set of actions
- **P** – Transition between states given an action

$$P^{a}_{s,s'} = \mathrm{Prob}[s_{t+1}=s' \mid s_t=s,\ a_t=a]$$

- **R** – Rewards associated with actions and states
- γ – Discount factor



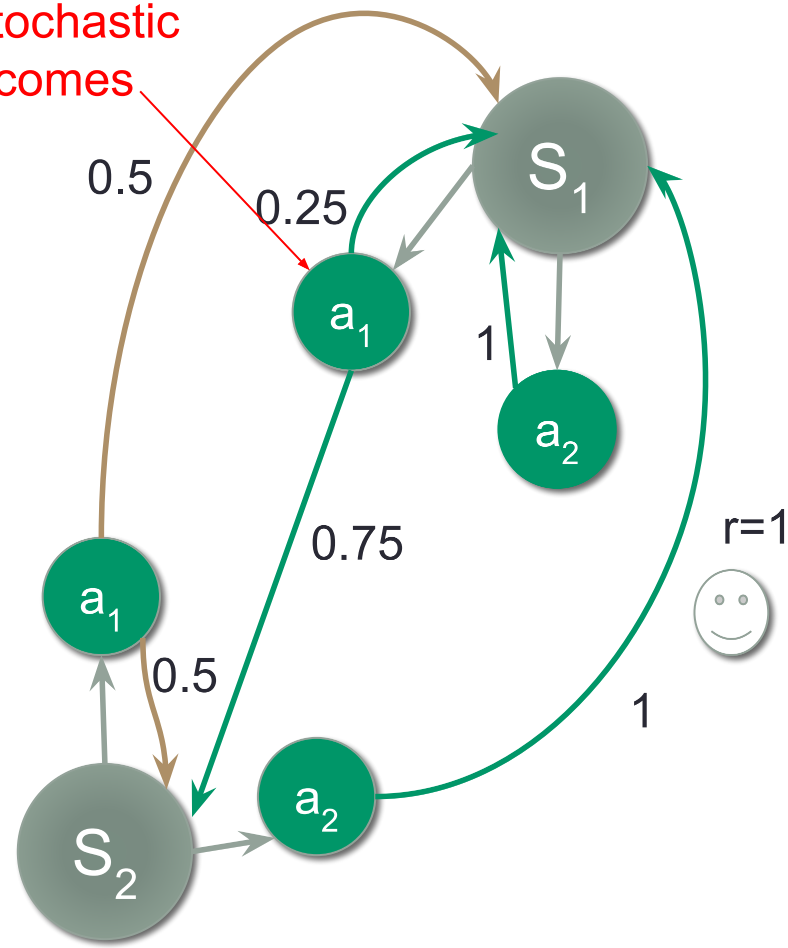
Markov Decision Process (MDP)

Environment could be stochastic
One action, multiple outcomes

- **S,A,P,R,**γ
- **S** – Set of states
- **A** – Set of actions
- **P** – Transition between states given an action

$$P^{a}_{s,s'} = \mathrm{Prob}[s_{t+1}=s' \mid s_t=s,\ a_t=a]$$

- **R** – Rewards associated with actions and states
- γ – Discount factor



Markov Property

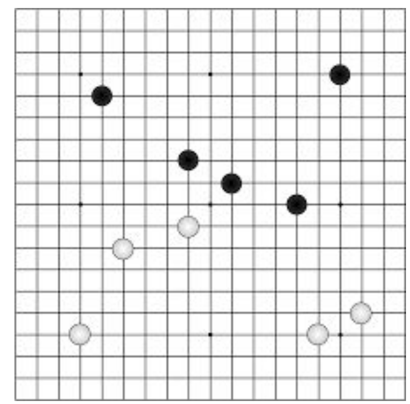
$$p(s_{t+1} | s_t, a_t)$$

- s_{t+1} depends only on s_t
- not s_{t-1}, not anything before
- this simplifies our situation!

But, is it true in every case?

- It depends on your observed state
  - Fully observable state
  - Partially observable state

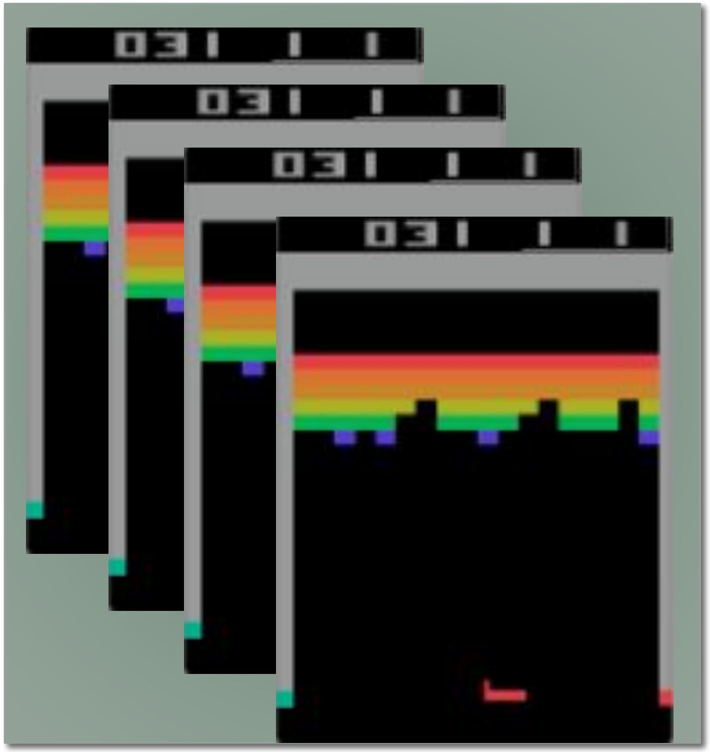
Fog of war



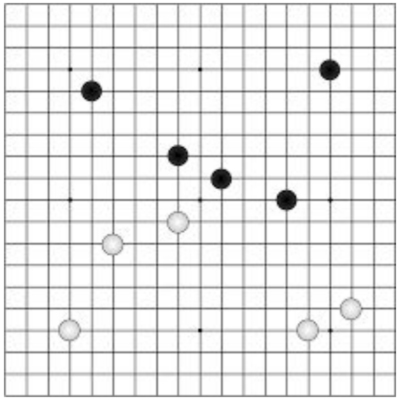
Fully observable

Fully Observable State

- Fully observable state: All information from the past is captured in the current state



For Go, a board position
For simple video games, stack multiple frames



The Agent

Policy

- Policy = a mapping from a state to an action

- Objective of RL is to find the "*optimal*" policy!

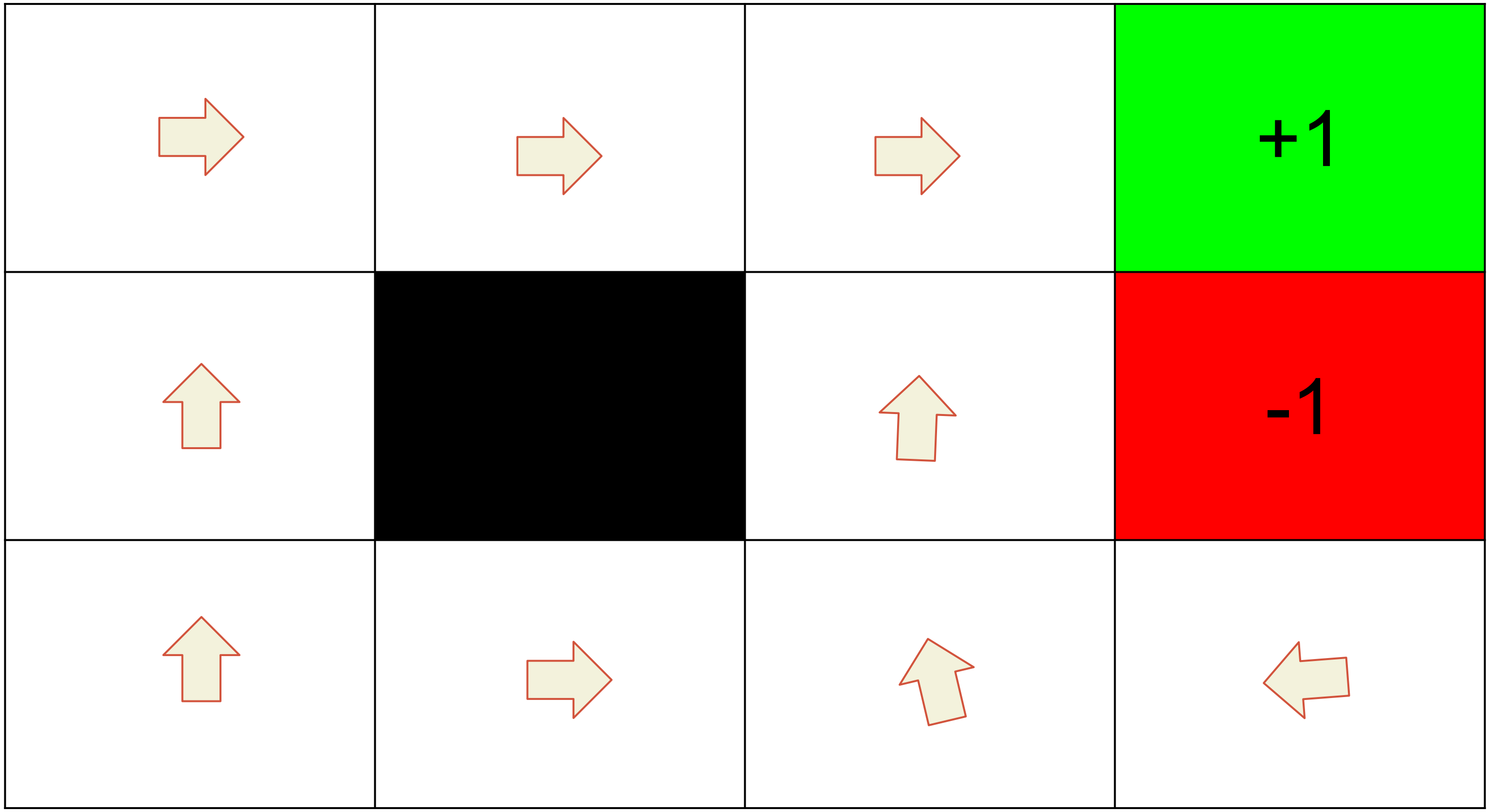
$$a = \pi(s)$$

Can be either deterministic or stochastic

$$a \sim \pi(s)$$

Policy

Example: tabular policy



Learning

How do we find the best policy ?

Rollout (Our data)

Time	0	1	2	3	...	T-1	T
S	S_0	S_1	S_2	S_3	...	S_{T-1}	S_T
A	A_0	A_1	A_2	A_3	...	A_{T-1}	A_T
R	R_0	R_1	R_2	R_3	...	R_{T-1}	R_T
Done	0	0	0	0	...	0	1

Don't care

END! DON'T CARE

$s_0 \longrightarrow$ Agent (π) $\longrightarrow a_0 \longrightarrow \dots \longrightarrow s_{T-1} \longrightarrow$ Env $\longrightarrow s_T \longrightarrow$ Agent (π) $\longrightarrow a_T$

Env $\longrightarrow r_T$

Env $\dashrightarrow$ Terminated

Return (Cumulative rewards)

- Return = cumulative rewards with discount

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{t'=t}^{\infty} \gamma^{t'-t} r_t$$

$r_t = 0$

$r_{t+10} = 1$

$r_{t+34} = -1$

What is learning?

- Use data to find/search for the best policy

What is the best policy?

- Policy that give us the highest expected return!

$$G = r_0 + \gamma r_1 + \gamma^2 r_2 + \dots$$

$$J(\pi) = E_\pi[G]$$

Return under a policy (G^{π})

$$s_0,\ \overset{\pi(s_0)}{\overset{\downarrow}{a_0}},\ r_1,\ s_1,\ \overset{\pi(s_1)}{\overset{\downarrow}{a_1}},\ r_2,\ s_2,\ \overset{\pi(s_2)}{\overset{\downarrow}{a_2}},\ r_3,\ s_3,\ \overset{\pi(s_3)}{\overset{\downarrow}{a_3}},\ r_4,\ s_4,\ \overset{\pi(s_4)}{\overset{\downarrow}{a_4}}$$

$G_0 = r_1 + r_2 + r_3 + r_4 + \dots$

$G_1 = r_2 + r_3 + r_4 + \dots$

$G_2 = r_3 + r_4 + \dots$

$G_3 = r_4 + \dots$

* THIS IS JUST ONE TRY!
ANOTHER MIGHT YIELD DIFFERENTLY

Expected Return

How good "on average" is our return?

Statistical expectation

For value G_0

$$\mathbb{E}_{\pi}[G_0]$$

Following policy π

- Start
- Play with π
- $G_0 = \sum_{t=0}^{T} r_t$
- Retry many times (∞)
- Average G_0

A naive learning method

1. Initialise a policy randomly
2. Evaluate the policy by running that policy multiple times
   a. which we then collect the returns of all the runs
3. Randomly initialise another policy
4. Evaluate the new policy
5. Keep the policy that have a higher expected return
6. Repeat 3-5

Intuitive. But very inefficient!

How to make it more efficient?

Model-free RL

Q-learning algorithm

- Let's define a state value as
  - Expectation of the return after visit s and follow π

$$V^{\pi}(s) = E_{\pi}[G_t | s_t = s]$$

- Let's define a state-action value (Q-value) as
  - Expectation of the return after visit a state s, take action a

$$Q^{\pi}(s, a) = E_{\pi}[G_t | s_t = s, a_t = a]$$

$$V^{\pi}(s) = E_{\pi(s)}[Q^{\pi}(s, \pi(s))]$$

Q-learning algorithm

- There exist an optimal value function associate with an optimal policy,

$$V^*(s) = \max_\pi V^\pi(s) \quad \forall s \in S$$

- The optimal policy is the policy that achieves the highest value for every state

Q-learning algorithm

- It follows that

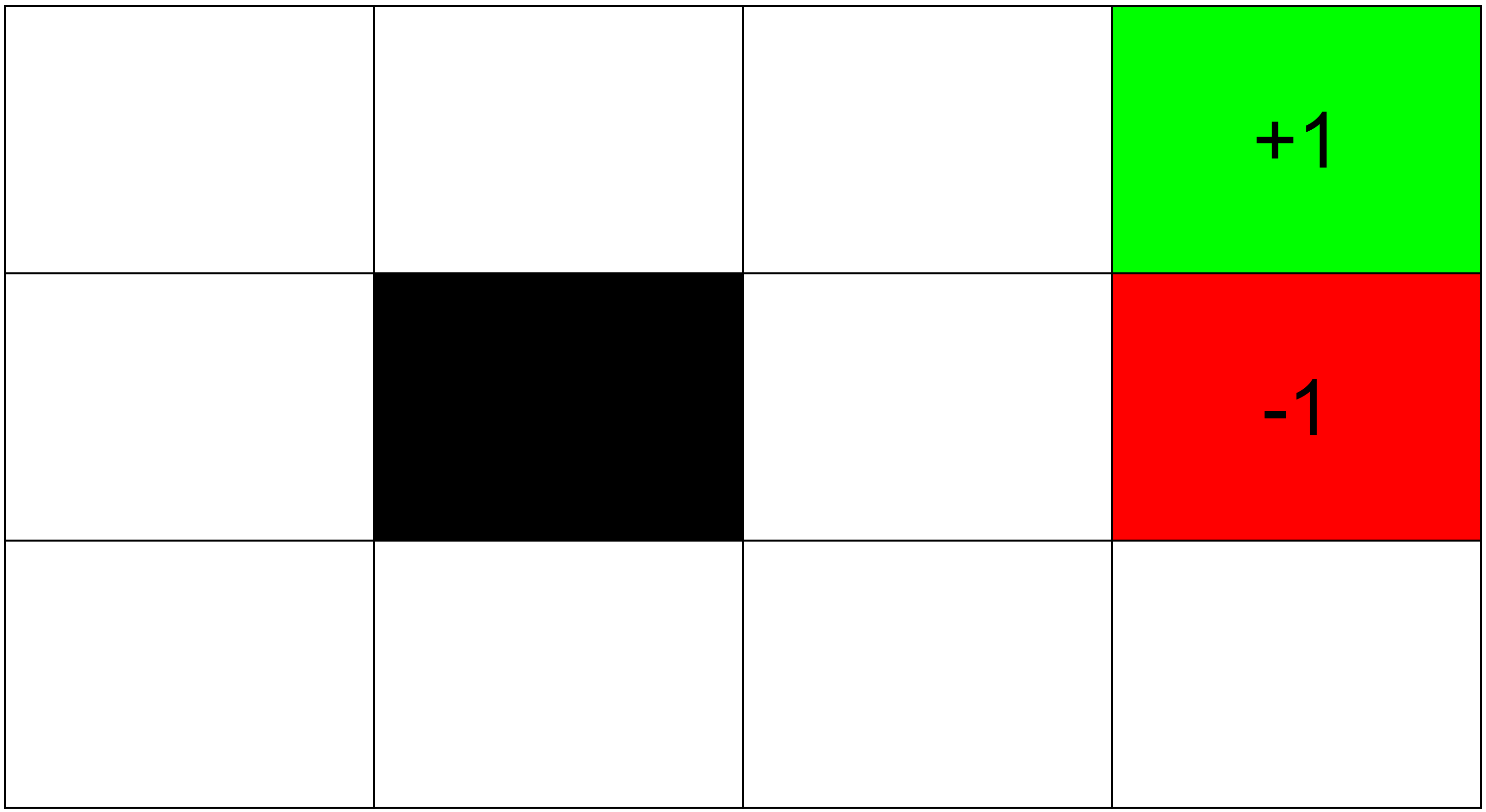
$$V^*(s) = \max_a [Q^*(s, a)]$$

- and ..

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

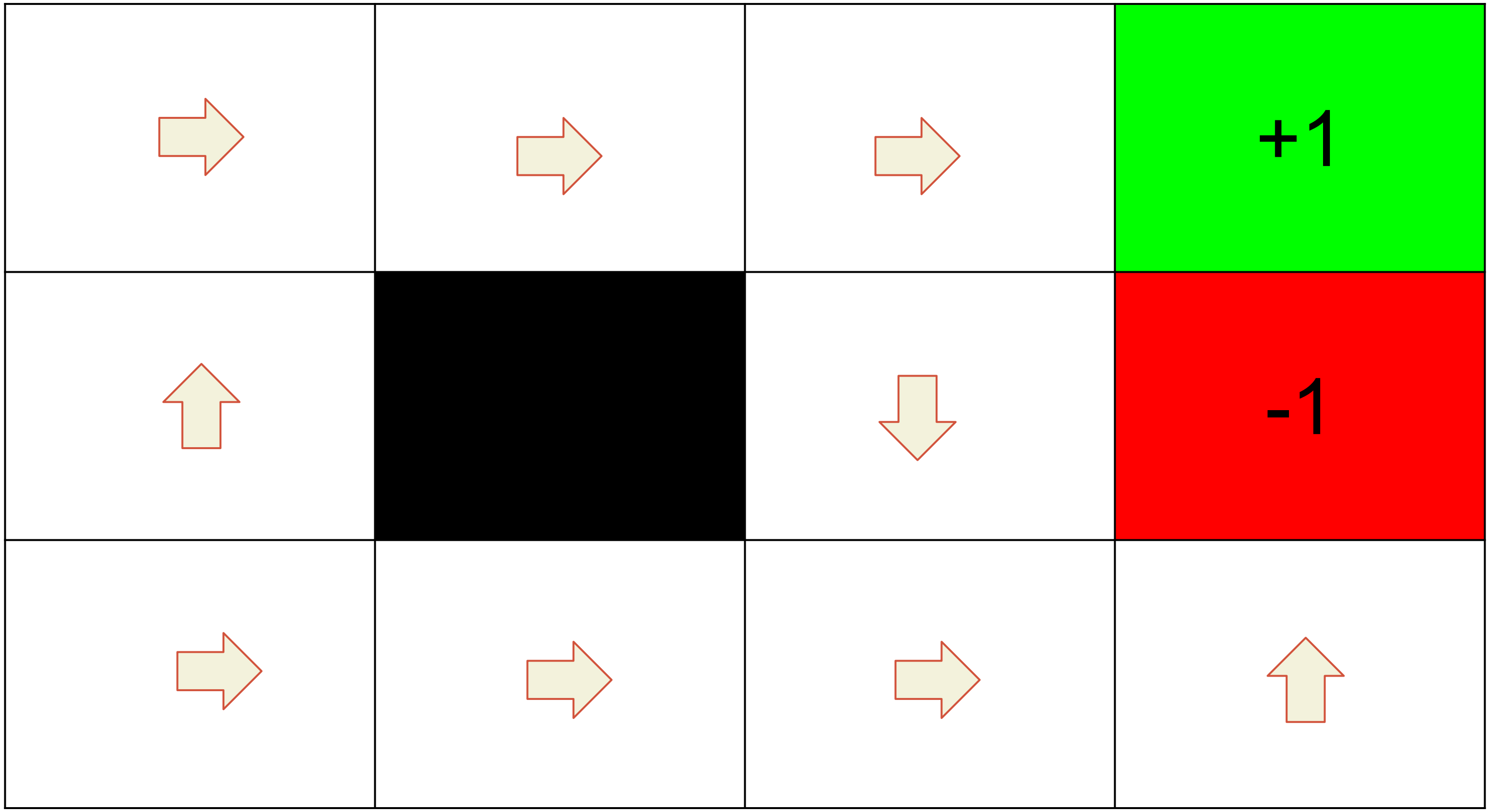
- Optimal actions can be found indirectly through Q-value

Example, Tabular Q-learning



Monte-Carlo Estimator

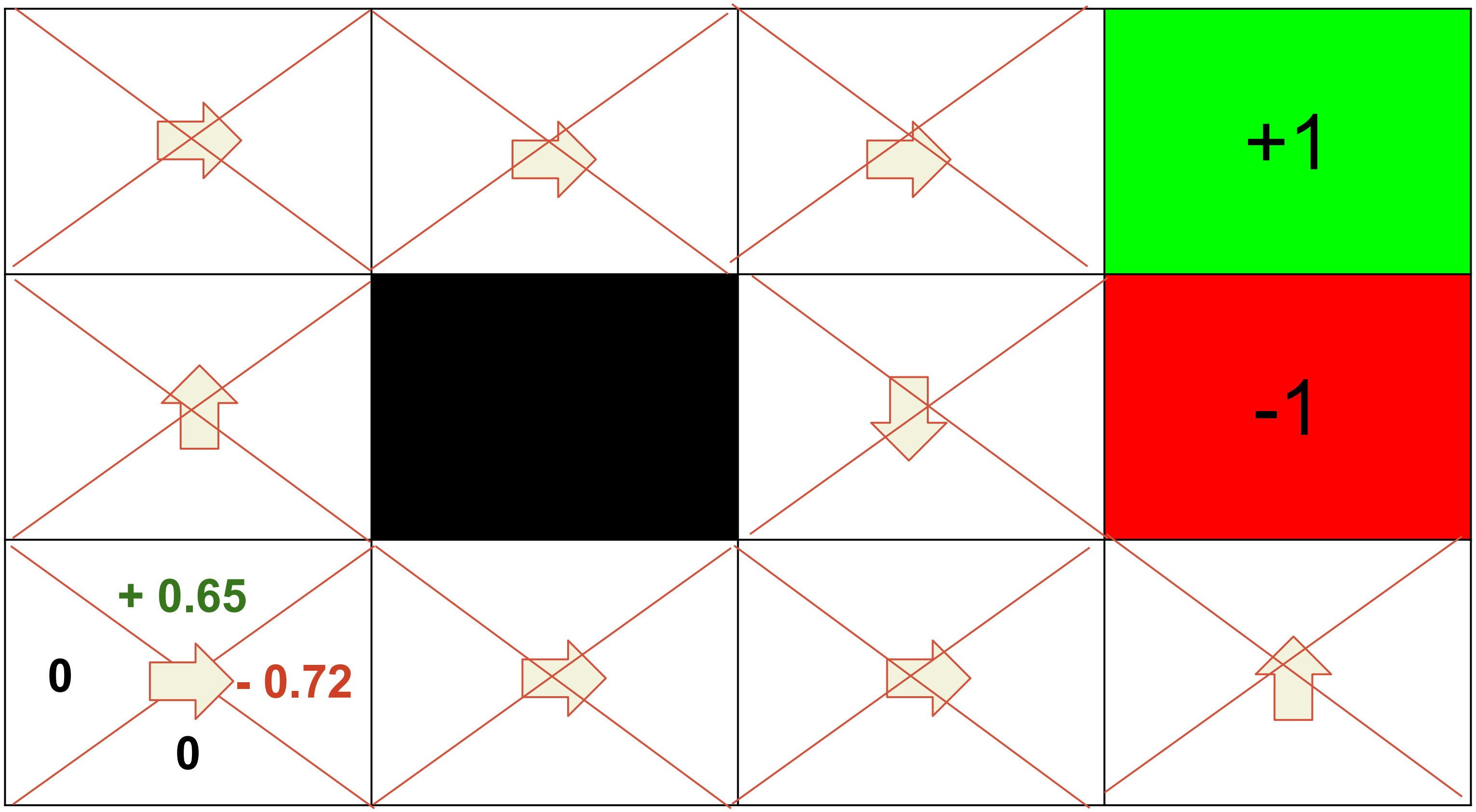
- Initialise π



Monte-Carlo Estimator

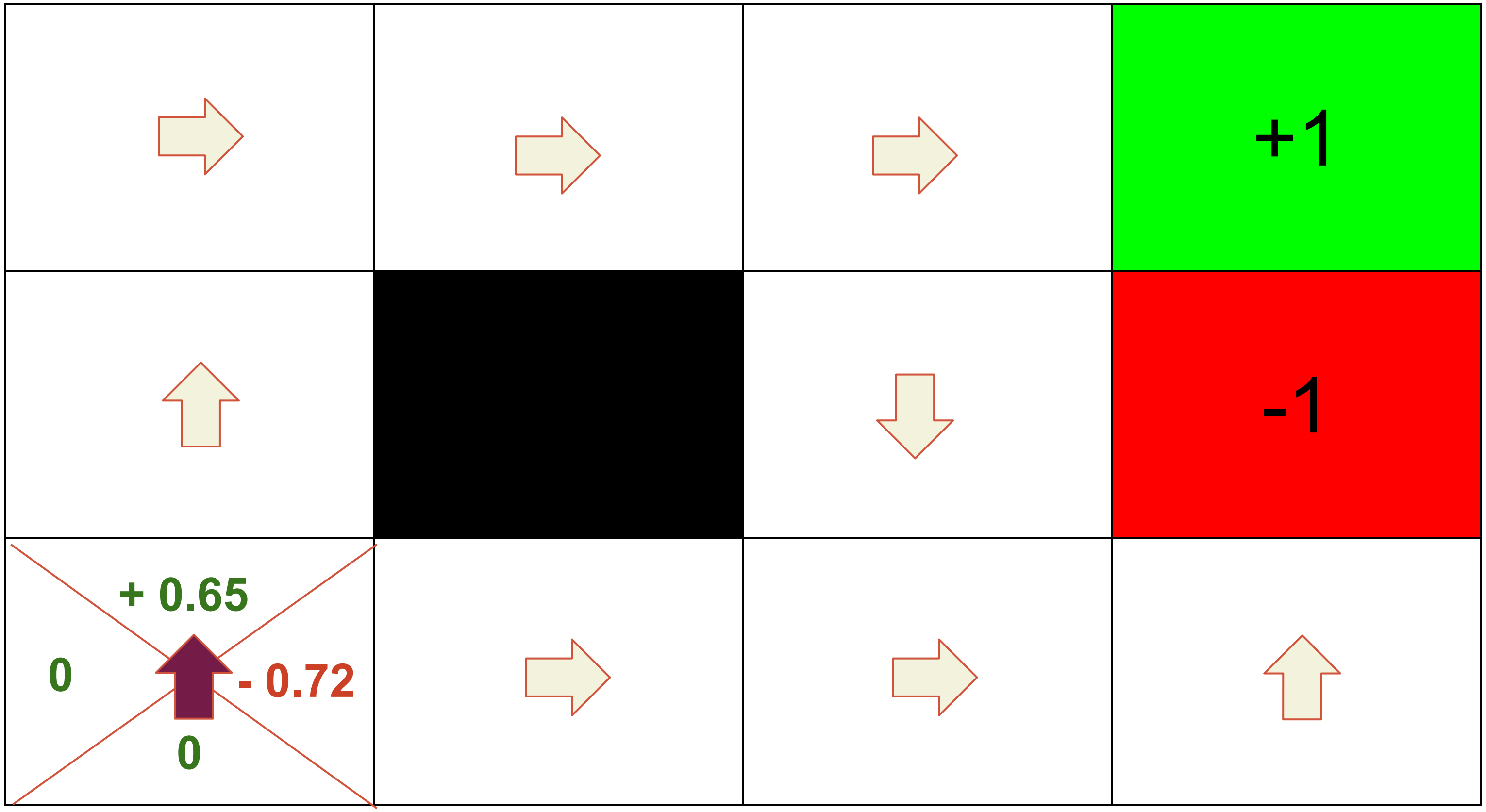
$$Q^{\pi}(s,a) \approx r_0 + \gamma r_1 + \gamma^2 r_2 + .. + \gamma^n r_n$$

$$\gamma = 0.9$$



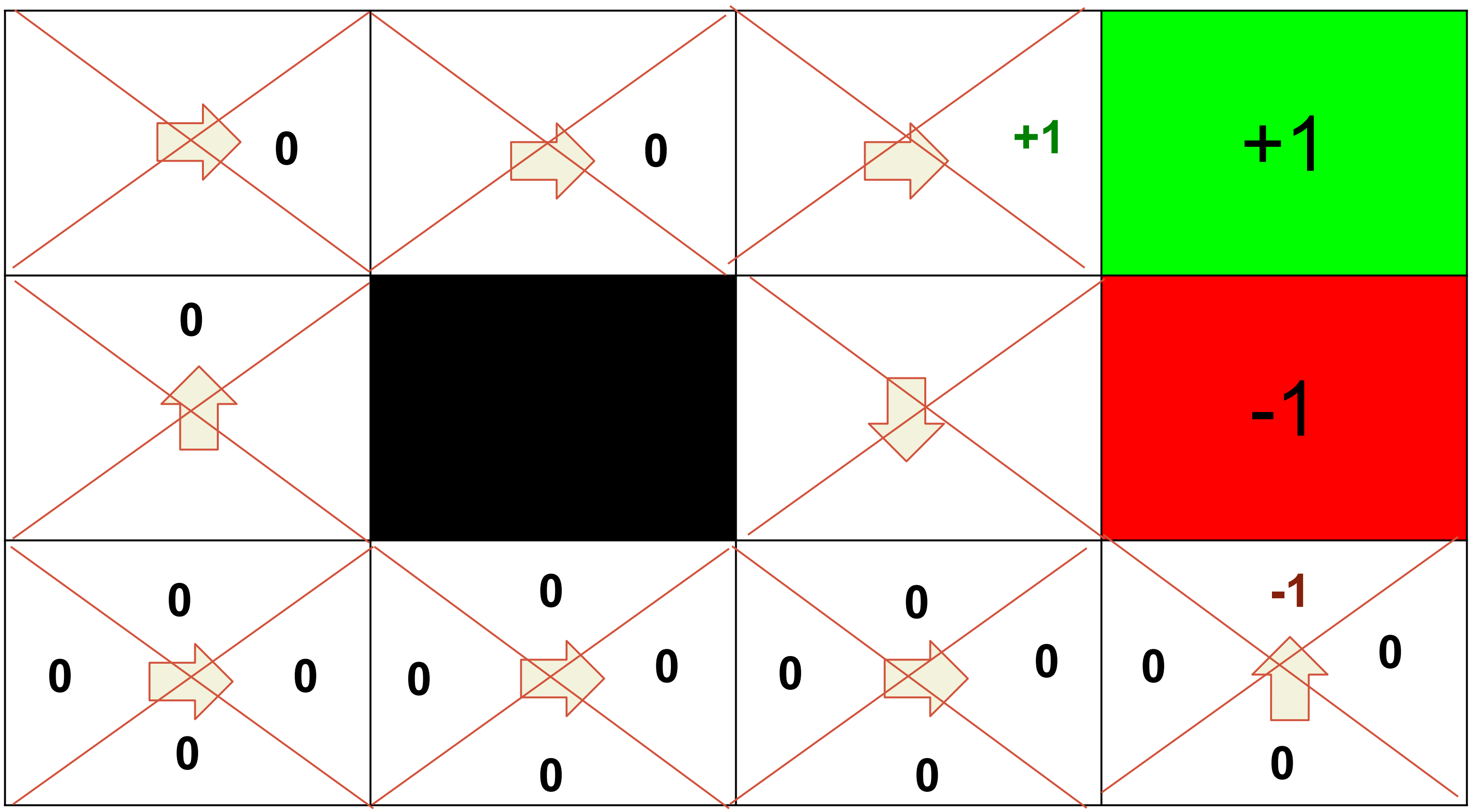
Policy Improvement

$$\pi'(s) = argmax_a Q(s,a)$$



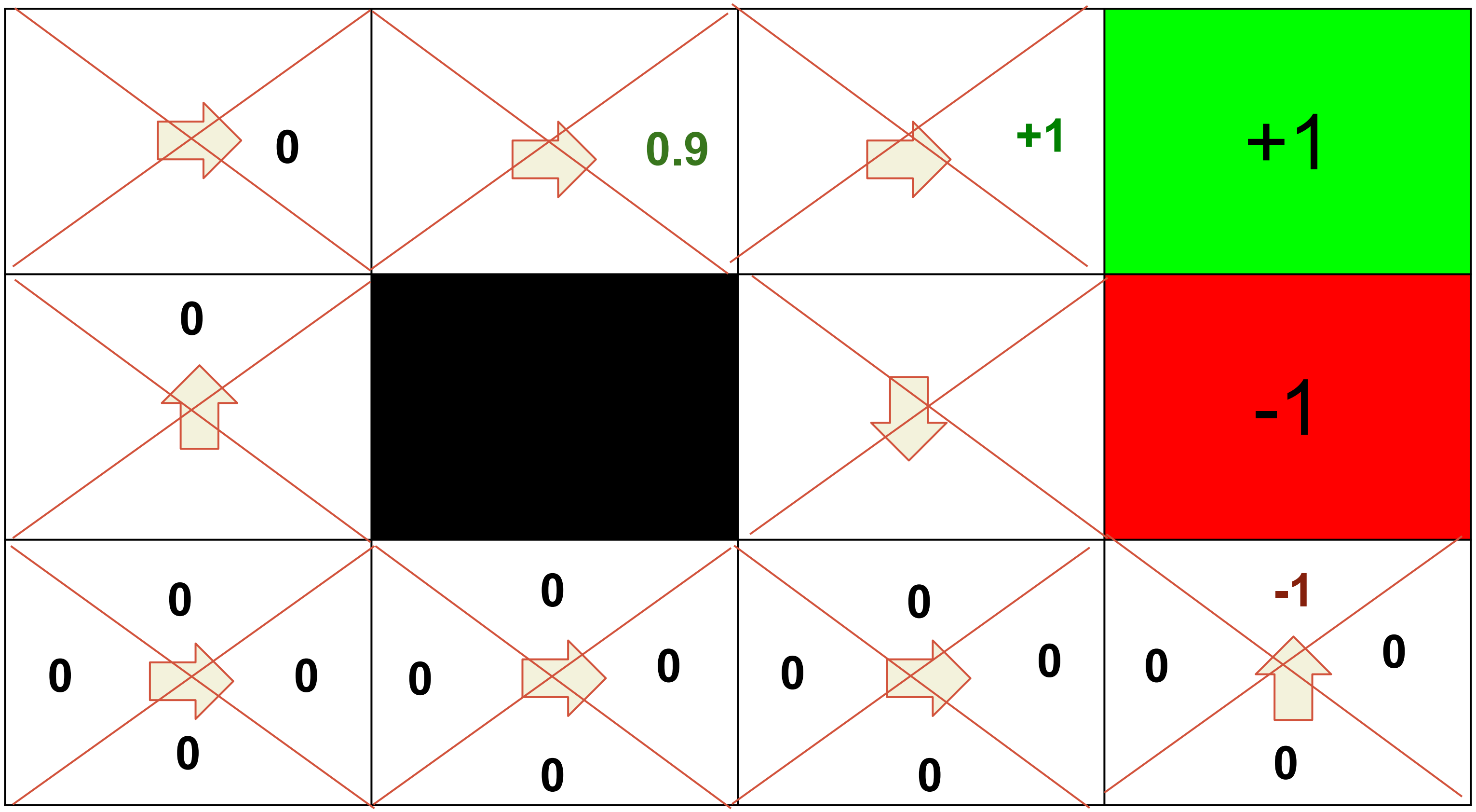
Bootstrap Estimator

$$Q(s,a) \approx r_0 + \gamma \max_b Q(s',b)$$



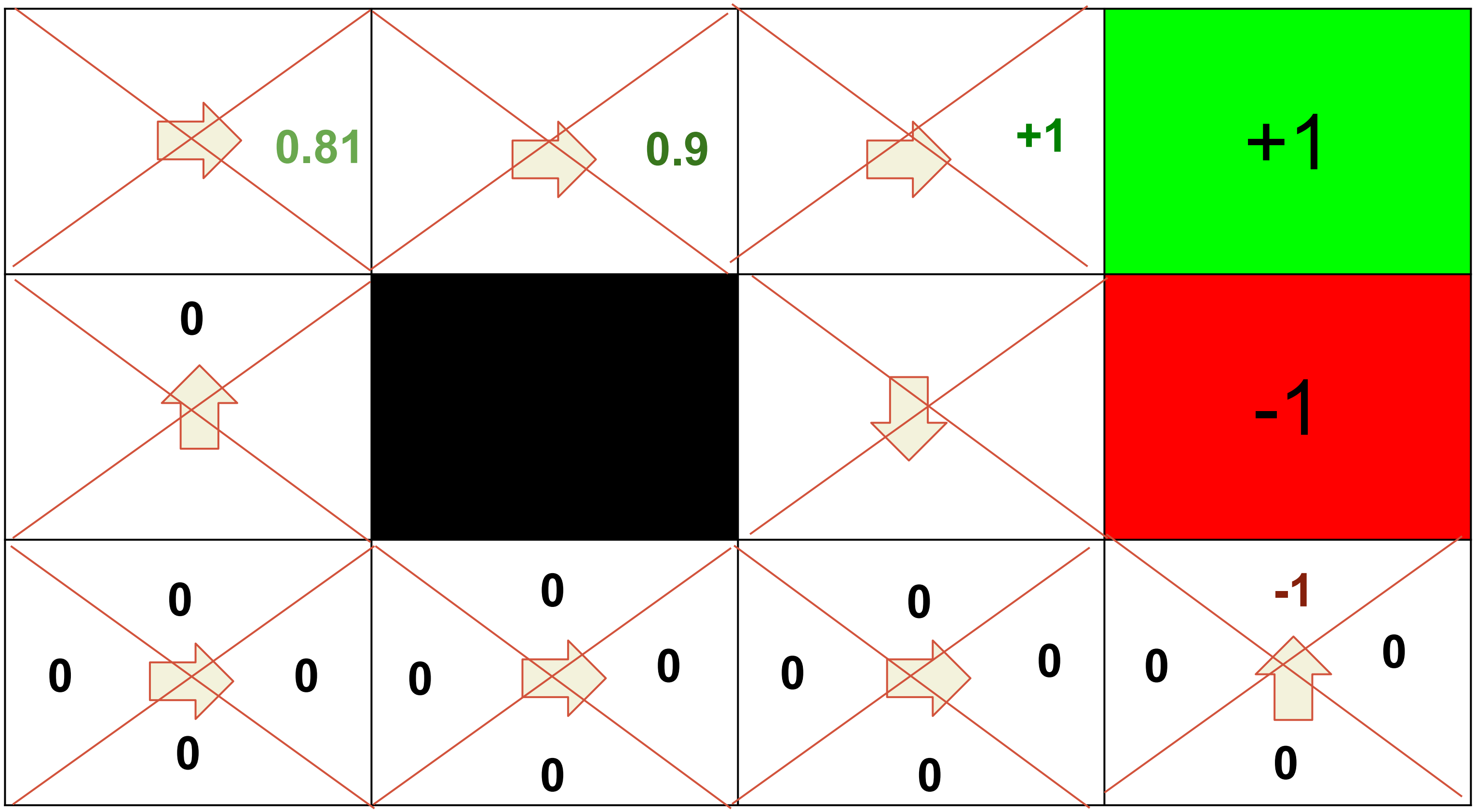
Bootstrap Estimator

$$Q(s,a) \approx r_0 + \gamma \max_b Q(s',b)$$



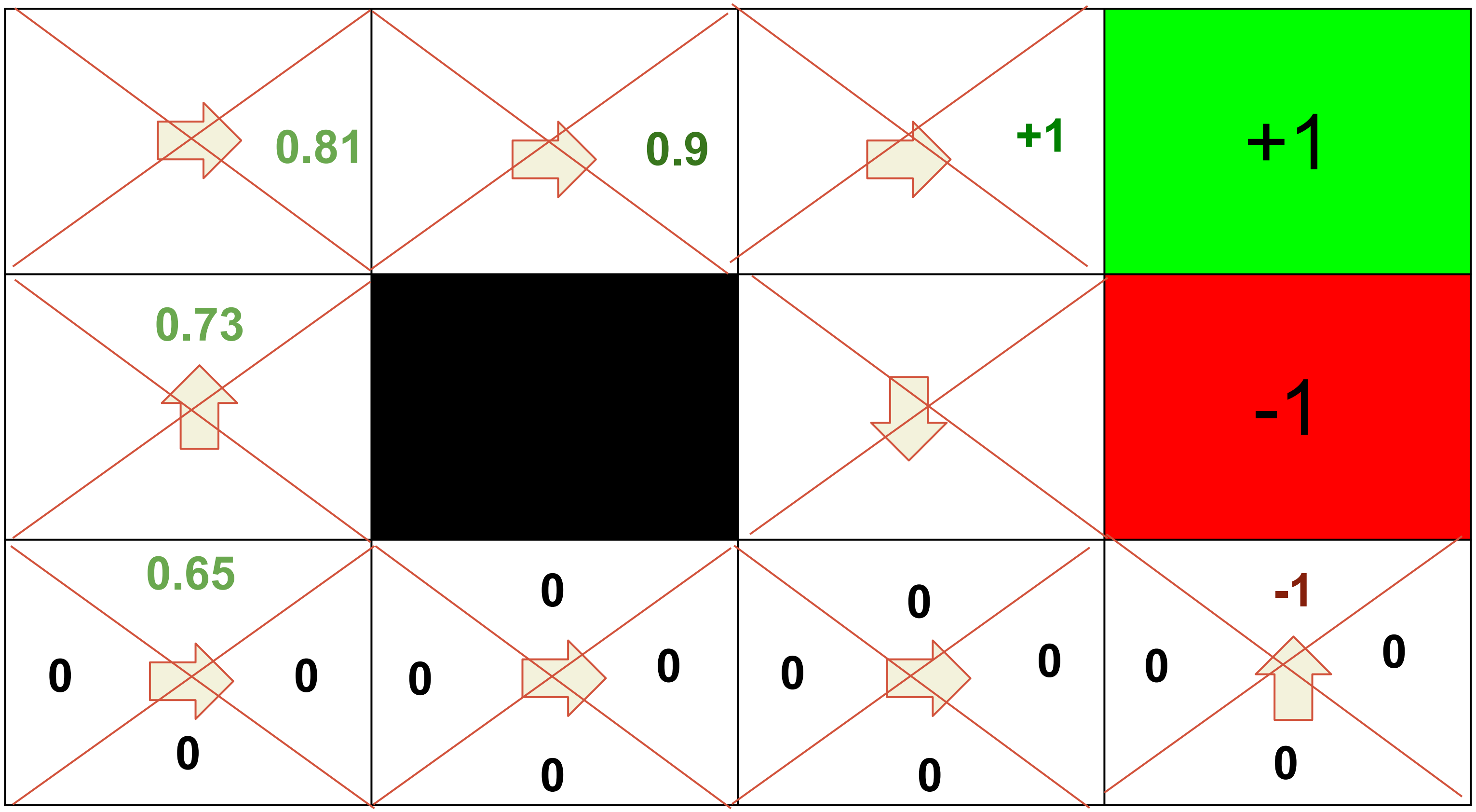
Bootstrap Estimator

$$Q(s,a) \approx r_0 + \gamma \max_b Q(s',b)$$

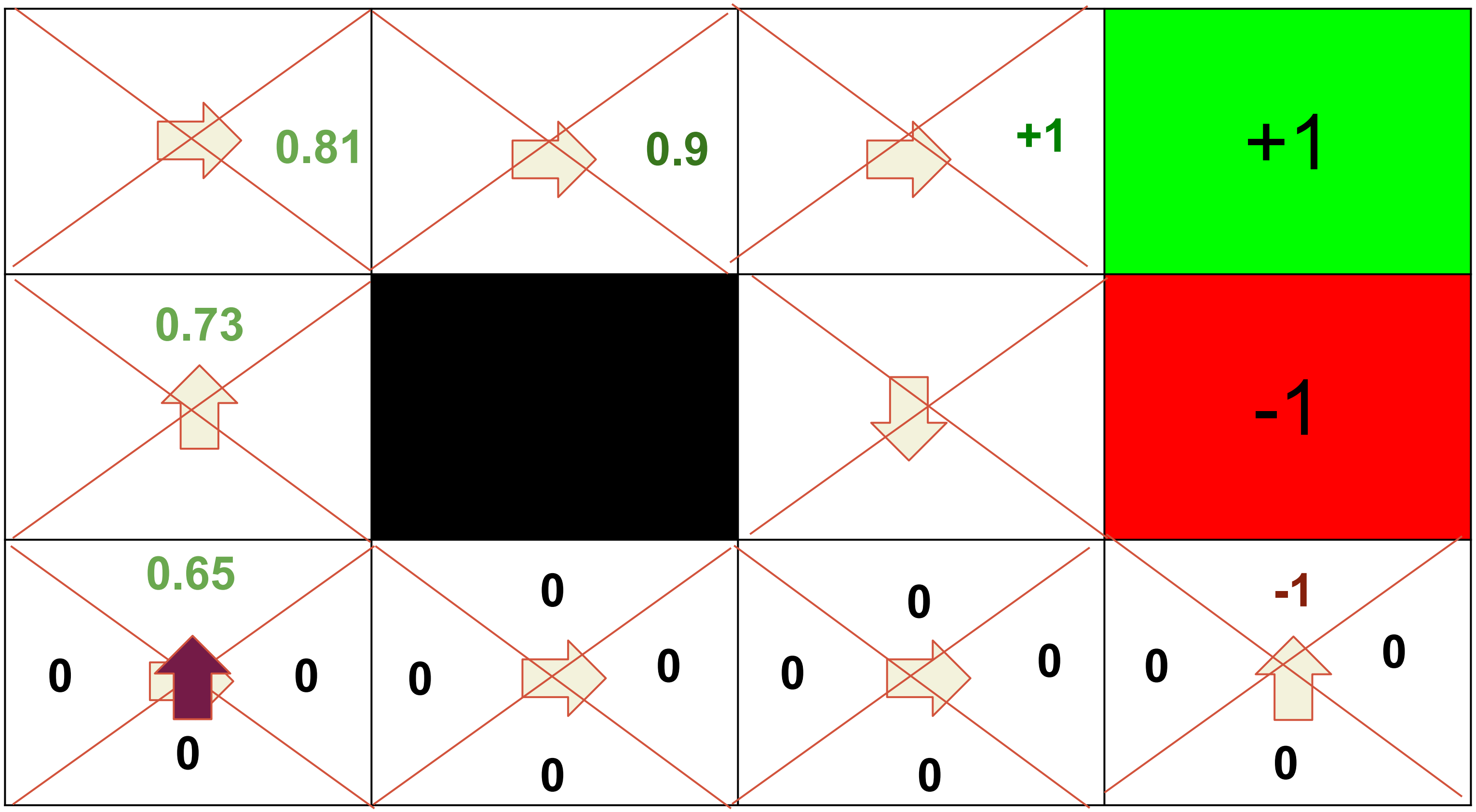


Bootstrap Estimator

$$Q(s, a) \approx r_0 + \gamma \max_b Q(s', b)$$



Policy Improvement



Bias and Variance in RL

What is bias of V_π estimation?

- Let $\hat{V}_\pi(s)$ be an estimate of $V_\pi(s)$
- $\hat{V}_\pi(s)$ is unbiased if:

$$\mathbb{E}_s\left[\hat{V}_\pi(s) - V_\pi(s)\right] = 0$$

What is variance of $\hat{V}_\pi(s)$ estimation?

- $\mathbb{V}\mathrm{ar}\left[\hat{V}_\pi(s)\right] = \mathbb{E}_s\left[(\hat{V}_\pi(s) - \mathbb{E}_s\left[\hat{V}_\pi(s)\right])^2\right]$
- High if $\hat{V}_\pi(s)$ fluctuates a lot

Bias and Variance

- Monte-Carlo estimate has high variance and low bias.
- Bootstrap estimate has higher bias but lower variance.

Function Approximator (FA)

- Tabular Q-value is impractical when the state-action space is large!
  - Need large memory
  - Impractical to fill up every cell

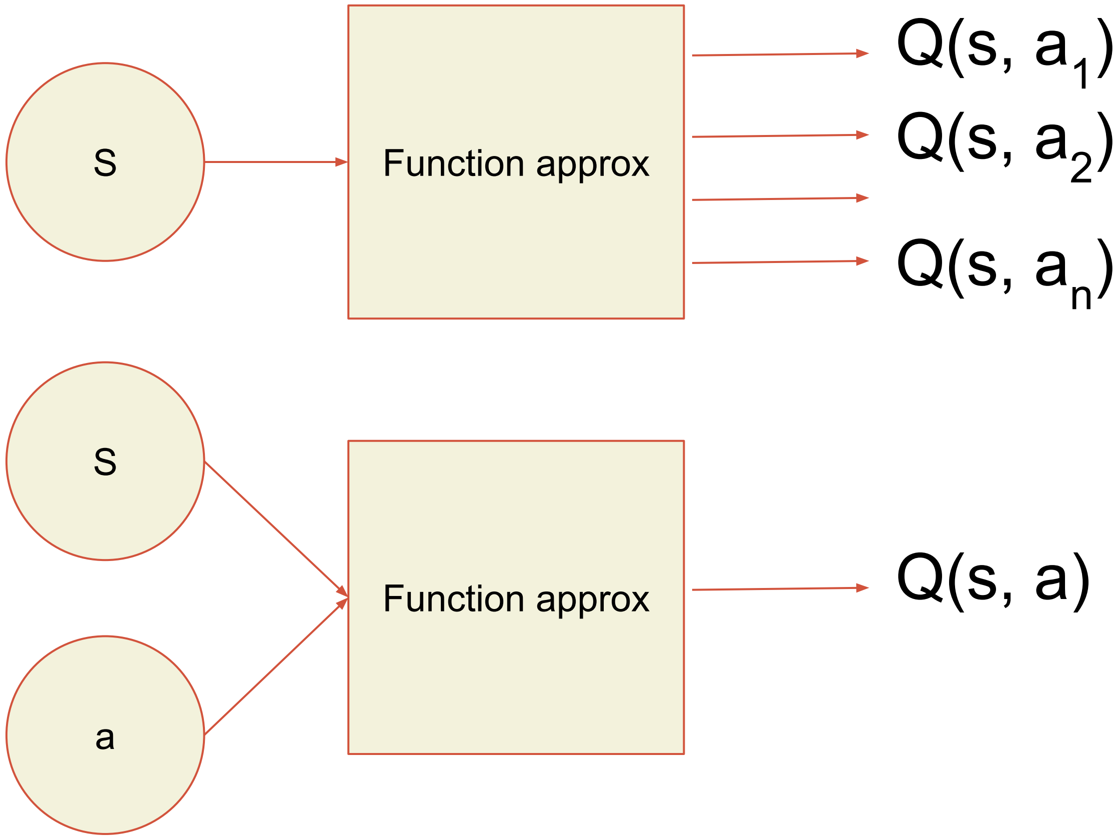
- Enter .. a function approximator

Function Approximator (FA)

- Tabular Q-value is impractical when the state-action space is large!
  - Need large memory
  - Impractical to fill up every cell
- Enter .. a function approximator

Function Approximator (FA)

- Instead of a table containing Q-value for every state and action, use a function that output Q-values.



Learning with FA

- With tabular Q-learning,
  - the act of learning = putting Q-value in the table
- With function approximator,
  - the act of learning = searching for the optimal parameters of the FA

Learning with FA

- How to adapt the parameters (weights) of the FA?
- Step 1: Define a loss function.
- Step 2: Optimise the weights to minimise the loss

Loss function

- What should be the loss function?
- Introducing Bellman's equations

$$V^{\pi}(s_t) = E_{\pi,P}[r_t + \gamma V^{\pi}(s_{t+1})]$$

$$Q^{\pi}(s_t) = E_{\pi,P}[r_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1})]$$

- Bellman's optimality equations

$$Q^{*}(s_t) = E_{P}[r_t + \gamma \max_b Q^{*}(s_{t+1}, b)]$$

Loss function

- The Bellman's equation must hold for correct Q-value
- Rewrite the Bellman's optimality with our estimator (FA)
-

$$\hat{Q}(s_t) = E_P[r_t + \gamma \max_b \hat{Q}(s_{t+1}, b)]$$

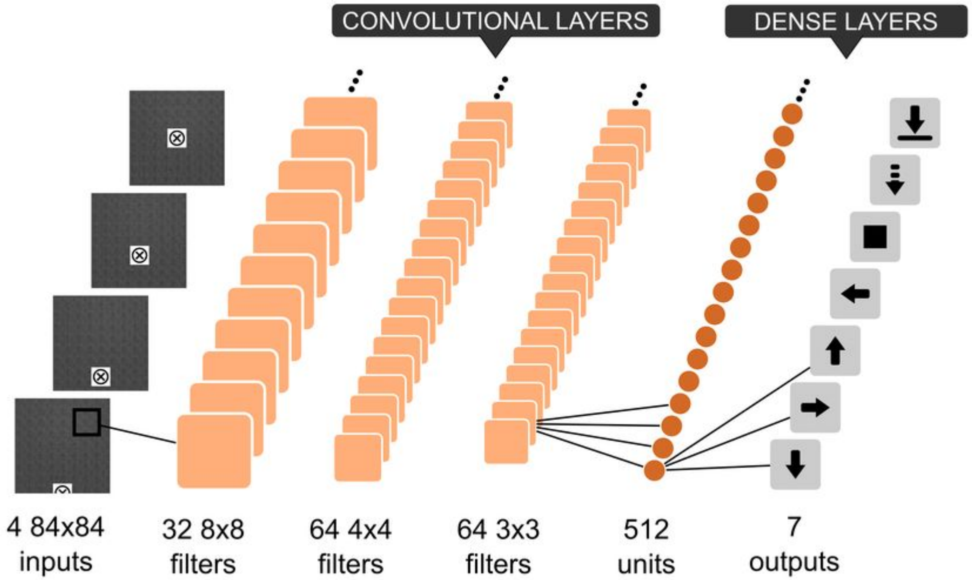
The estimator is correct if the left hand side = right hand side

$$TD = r_t + \gamma \max_b \hat{Q}_\theta(s_{t+1}, b) - \hat{Q}_\theta(s_t)$$

``Temporal Difference error''

Temporal Difference Learning

- Use TD-error to guide learning
- Example
  - Deep Q-Neworks (DQN)
  - Deep convolutional neural network as a function approximator
  - Optimise square TD-error



$$L(\theta) = (r_t + \gamma \max_b \hat{Q}_\theta(s_{t+1}, b) - \hat{Q}_\theta(s_t))^2$$

Policy Gradient methods

Policy gradient

Q Learning

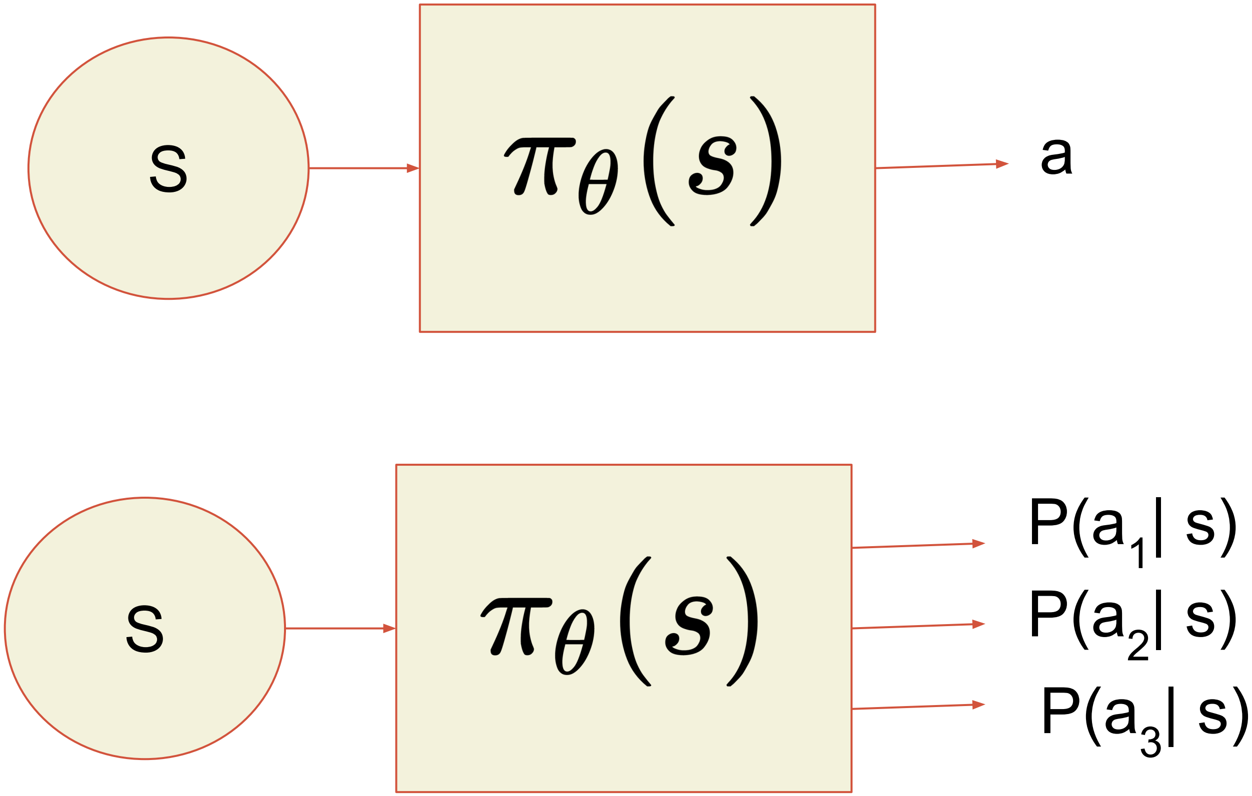
- policy is implicit
- if we already have Q, we have policy
- we just look at Q to get π

Policy gradient

- learns π directly explicitly
- use Q, V as a helper for learning π

Policy gradient

- Use Function Approximator to represent policy directly



Loss function

- Start-state objective

$$J(\theta) = E_{\pi(\theta)}[G|s_0] = V(s_0)$$

- Average-reward objective

$$J(\theta) = \sum_s d^\pi(s) \sum_a \pi(s,a) r(s,a)$$

* **d** is a stationary distribution of a Markov chain.

- One way to optimise these objectives is to use SGD.

Computing the gradient

Let’s try to compute the gradient of the start-state objective

$$J(\theta) = E_{\pi_\theta}[G|s_0]$$

To evaluate this expectation, maybe we could try
a one-sample Monte-Carlo estimator:

$$J(\theta) \approx r_0 + \gamma r_1 + \gamma^2 r_3 + \ldots$$

$$\nabla J(\theta) \approx \nabla_\theta [r_0 + \gamma r_1 + \gamma^2 r_3 + \ldots]$$

Doesn’t quite work? The evaluated value does not depend on θ . Gradient can’t be computed.

Computing the gradient

Maybe we can try change θ a little bit and find the difference?

$$J(\theta) \approx r_0 + \gamma r_1 + \gamma^2 r_3 + \dots$$

$$J(\theta + \delta\theta) \approx r'_0 + \gamma r'_1 + \gamma^2 r'_3 + \dots$$

$$\nabla J = \frac{J(\theta) - J(\theta + \delta)}{\delta}$$

Could work? But...

Looks very expensive and noisy to compute!

Maybe there is a better way?

Policy gradient

Let's start from the average-reward objective

$$J(\theta) = \sum_s d^{\pi}(s) \sum_a \pi_\theta(s,a) r(s,a)$$

For simplicity let's assume d(s) does not depend on θ

$$J(\theta) = \sum_s d(s) \sum_a \pi_\theta(s,a) r(s,a)$$

$$\nabla_\theta J(\theta) = \sum_s d(s) \sum_a \nabla_\theta \pi_\theta(s,a) r(s,a)$$

Almost there...

Policy gradient

REINFORCE trick!

$$\nabla_\theta J(\theta) = \sum_s d(s) \sum_a \nabla_\theta \pi_\theta(s,a) r(s,a)$$

$$\nabla_\theta J(\theta) = \sum_s d(s) \sum_a \pi_\theta \nabla_\theta \log \pi_\theta(s,a) r(s,a)$$

$$\nabla_\theta J(\theta) = E_\pi[\nabla_\theta \log \pi_\theta(s,a) r(s,a)]$$

$$\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(s,a) r(s,a)$$

Policy gradient theorem

There is a theorem…called policy gradient theorem say that we can replace $r(s, a)$ with $Q(s, a)$

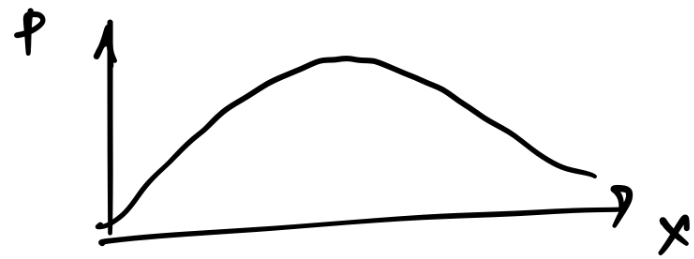
$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log\pi_\theta(s, a)Q^{\pi_\theta}(s, a)]$$

Encourage Exploration

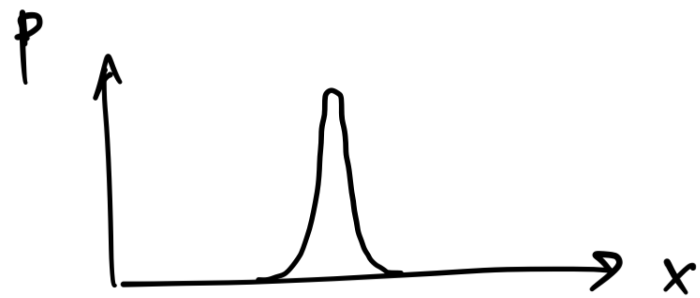
- policy $\pi_\theta(a|s)$ could be too confident early
- Like, $\pi_\theta(a=a|s)=1$
- this could lead to insufficient exploration
- encourage exploration by “entropy term” $H(\pi_\theta)$
- we want to punish too low entropy
- New gradient rule: $\nabla_\theta J(\theta) \rightarrow \nabla_\theta J(\theta) + \nabla_\theta H(\pi_\theta)$

Entropy (H)

high entropy



low entropy



On policy and off policy algorithms

Q Learning: $Q^*(s_t, a_t) = r_{t+1} + \max_a Q^*(s_{t+1}, a)$

- you need a_t, s_t, r_{t+1}, s_{t+1} to satisfy the above equation
- you can get (a, s, r, s') from <u>any</u> *policy*
- Q learning is *off-policy*

Policy gradient: $\nabla_\theta J(\theta) = \mathbb{E}\left[Q_\pi(s, a)\nabla_\theta \log \pi_\theta(a|s)\right]$

- you need s, a and $Q_\pi(s, a)$
- Q_π needs to be from the current policy
- **s, a** needs to come from <u>current</u> policy
- policy gradient is *on-policy*

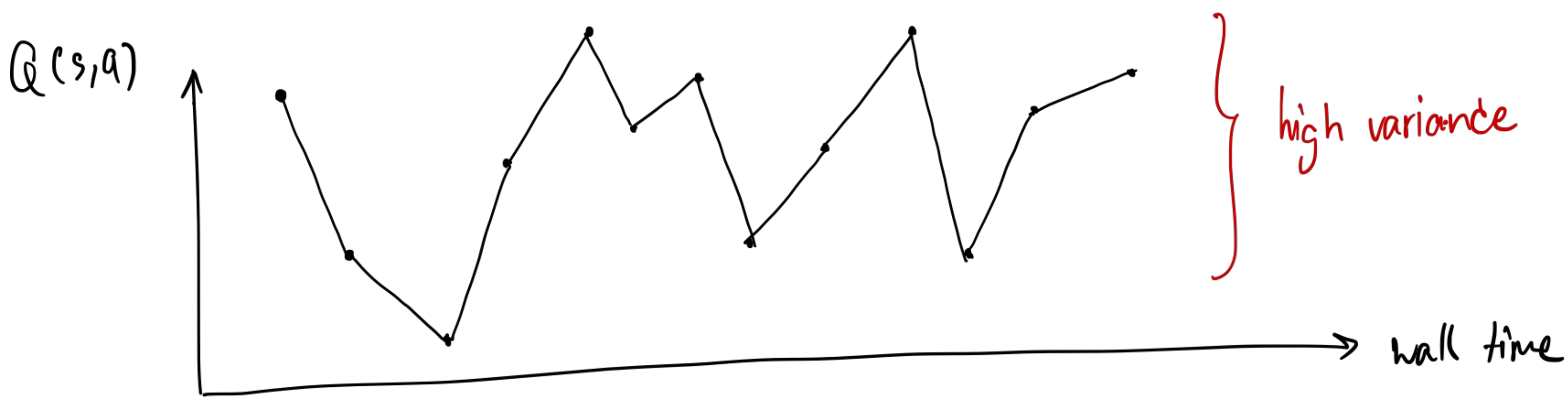
Baselines

$Q_\pi(s,a)$ has high variance (Monte Carlo)

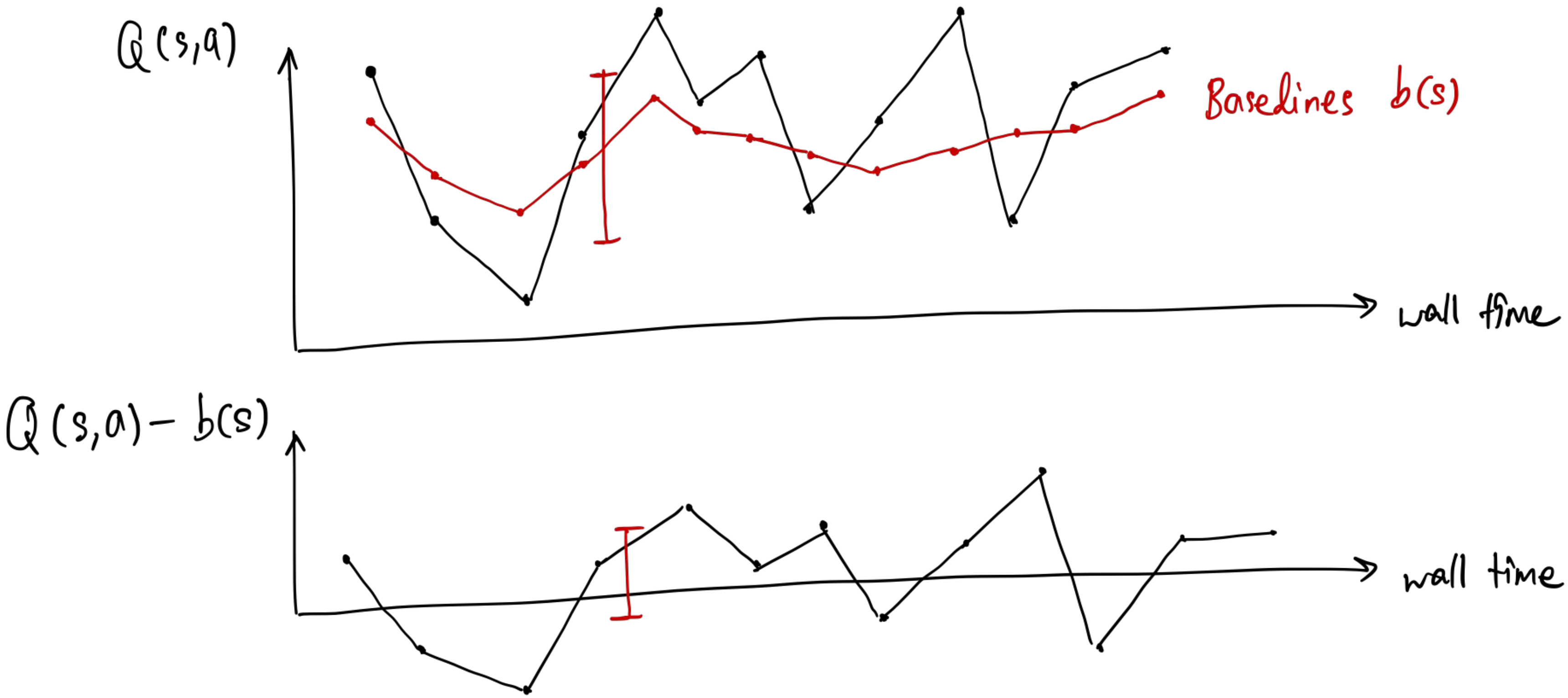
$Q_\pi(s,a)\nabla_\theta \log \pi_\theta(a|s) = \nabla_\theta J(\theta)$ is very noisy

Slow down training a lot!

We need to reduce variance to speed up the training



Baselines reduce variance



What is a good b(s)?

$V_\pi(s)$ is a convenient choice

Why baseline use b(s) not b(s, a)

- b(s, a) has potential to reduce more variance, but harder to incorporate to the policy gradient theorem
- b(s) can be added without changing the objective

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{s\sim d^\pi, a\sim\pi}\left[\left(Q_\pi(s,a) - b(s)\right)\nabla_\theta \log \pi_\theta(a|s)\right] \\ &= \mathbb{E}\left[Q_\pi(s,a)\nabla_\theta \log \pi_\theta(a|s)\right] - \mathbb{E}_{s\sim d^\pi, a\sim\pi}\left[b(s)\nabla_\theta \log \pi_\theta(a|s)\right]\end{aligned}$$

Consider: $\mathbb{E}_{s\sim d^\pi, a\sim\pi}\left[b(s)\nabla_\theta \log \pi_\theta(a|s)\right]$

$$\begin{aligned} &= \mathbb{E}_{s\sim d^\pi}\left[\int_a \pi(a|s) b(s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi(a|s)}\right] \\ &= \mathbb{E}_{s\sim d^\pi}\left[b(s)\nabla_\theta \int_a \pi_\theta(a|s)\right] \\ &= \mathbb{E}_{s\sim d^\pi}\left[b(s)\nabla_\theta 1\right] = 0\end{aligned}$$

b(s) doesn't affect the objective

Advantage Function

When we use V(s) as baseline:

$$A(s,a) = Q(s,a) - V(s)$$

We call this the **advantage function**.

It tells the relative value of the actions.

Lower variance than using absolute value of the actions.

Model-based RL

Model-based RL

- In model-based RL, we first build the model of the environment
- Then use that model to directly search for the answer.
- The problem is … inaccurate model can give us bad policies…
- It is believed that if we can treat the uncertainty in the model correctly...model-based RL is the most efficient method!
- However, measuring uncertainty in the model is also very difficult.

Things to consider

When do we need RL?

- Your action affects the observation.
  - x_1 , $a_1 \Rightarrow x_2$
- The target behaviour is difficult to be directly hard-coded.
- Collection of the data of a target behaviour is difficult.

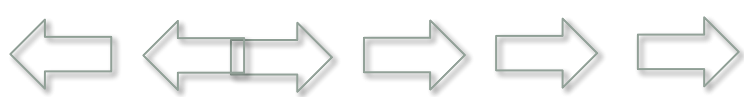
Credit assignment problem

- An action can have consequences further away in time
- Some movements might not have any effect on the outcome

10 time steps later

$r_t = 0$

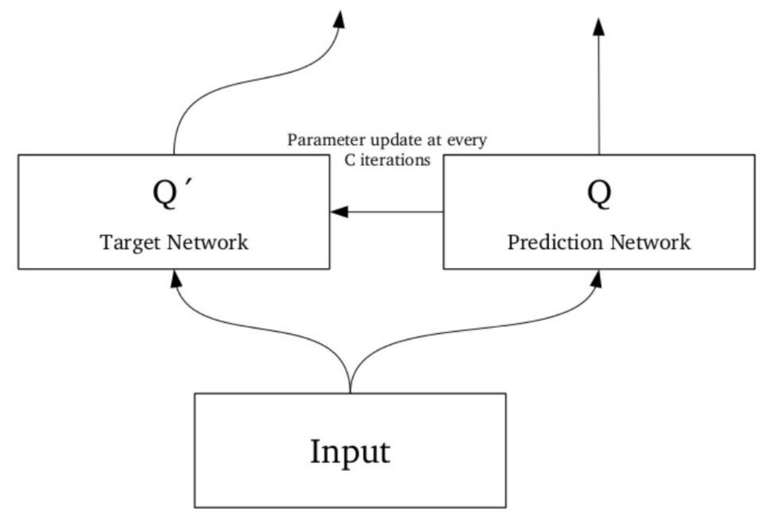
$r_{t+10} = 1$



Bias and Variance

- Bias and variance are really important in RL
- We want to reduce both of them as much as possible
  - DQN uses experience replay to reduce bias
  - DQN uses target network to reduce variance
  - Actor-Critic method uses baseline to reduce variance
  - Actor-Critic method uses parallel worker to reduce bias
  - etc.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim \mathrm{U}(D)}\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$



https://www.slideshare.net/MuhammedKocaba/humanlevel-control-through-deep-reinforcement-learning-presentation

Exploration and Exploitation

- Many RL results assume that the all of the states are visited infinitely often.
- Also, many RL algorithms are reduced into just an optimisation problem.
- Therefore, nicely spread/informative data can help a lot!

- DQN uses epsilon-greedy for exploration
- Policy gradient uses entropy regulariser to encourage exploration

Sparse reward problem

- Another problem is when rewards are sparse.
- Since, model-free RL is just learning the correlations of trajectories and rewards… when there is no reward, RL cannot learn.
- Can we make it better?
  - Curiosity + intrinsic motivation?
  - Curriculum learning?
  - Hierarchical RL?

Optimisation problem

- Initialization problems
- Is SGD the best we can do?
- Natural gradient?
- Adaptive learning rate?
- Catastrophic forgetting?

Designing the reward signal

- Reward design can be quite challenging..
- Naive reward design can lead to unexpected (cheating) behaviours!
- Example:
  - 

Current trends & open problems

- Intrinsic motivation, reward-bonus
- Imitation learning
- Multi-agent system and self-play
- Curriculum learning
- Model-based RL
- Robot learning + sim-to-real transfer learning
- etc...

Reinforcement learning

- Elements of RL
  - Environment, Agent, State, and MDP
- Estimating Q
  - Monte-Carlo, Bootstrap
  - Deep learning as a function approximator
- Policy learning
  - Q-learning
  - TD learning
  - Policy gradient
- Concepts
  - Exploration vs Exploitation