UNIVERSITY of WASHINGTON

Introduction to Reinforcement Learning

Jiafei Duan, 17 February 2026

CSE 571

Used Materials

- Acknowledgement: Some of the material and slides for the lecture were borrowed from Deep RL Bootcamp at UC Berkeley, 10-703 course at CMU by Katerina Fragkiadaki and Ruslan Salakhutdinov, whom originated from Rich Sutton and David Silver class and from Paul Laing CMU Lecture 9.1 and 9.2. Some advance material is also borrowed from Stanford CS234 and CIS 522 UPenn. Some animation are from MUTALINFORMATION YouTube.

Things to cover today:

- Introduction and motivation to RL
- Markov Decision Processes (MDPs)
- Solving known MDPs using value and policy iteration
- Temporal Difference & Q-Learning
- Tabular Q-Learning & Deep Q-Learning
- Policy Gradient Methods
- PPO

What is Reinforcement Learning?

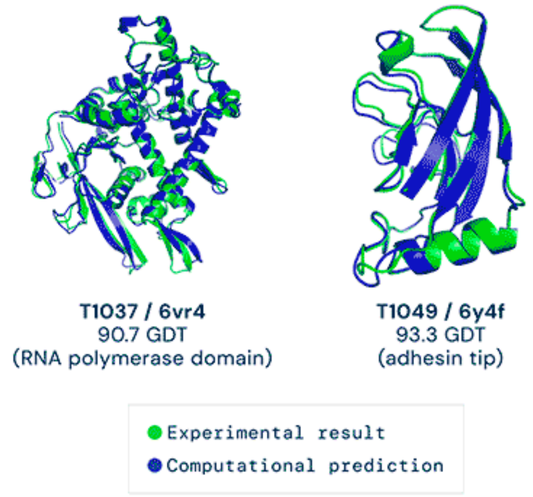
- Learning through experience/data to make good decisions under uncertainty.
- Essential part of intelligence.
- Builds strongly from theory and ideas starting in the 1950s with Richard Bellman.
- A number of impressive successes in the last decade.

Thorndike’s Puzzle Box (Law of effect, 1898)

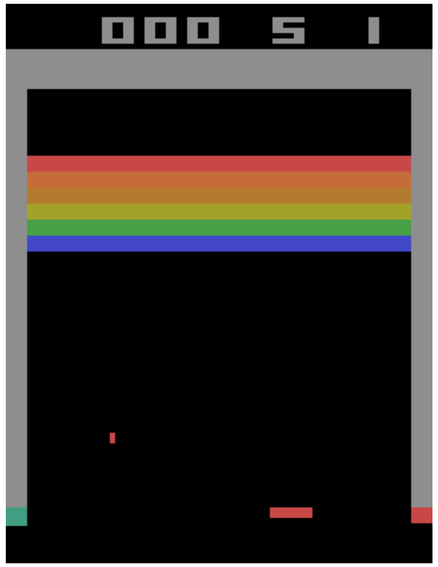
Motivation for RL

Self-driving

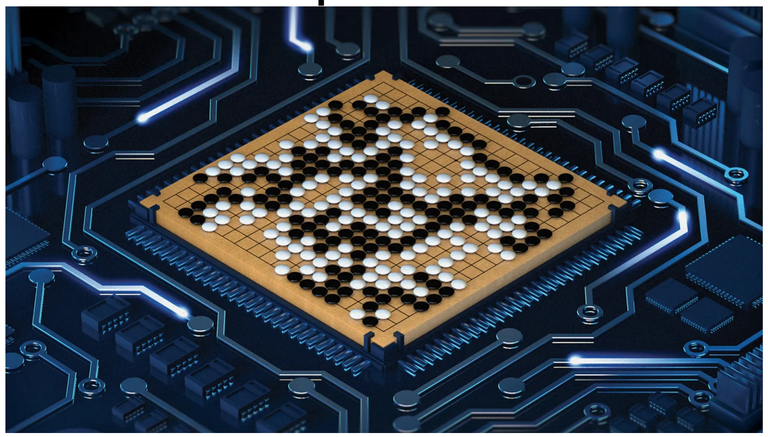
AlphaFold



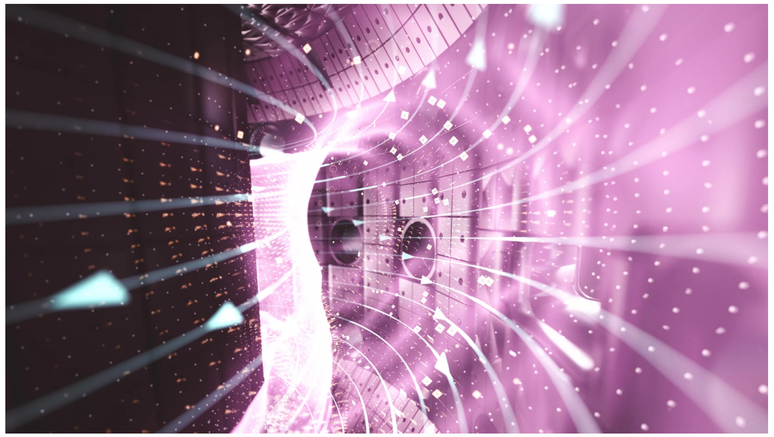
Atari



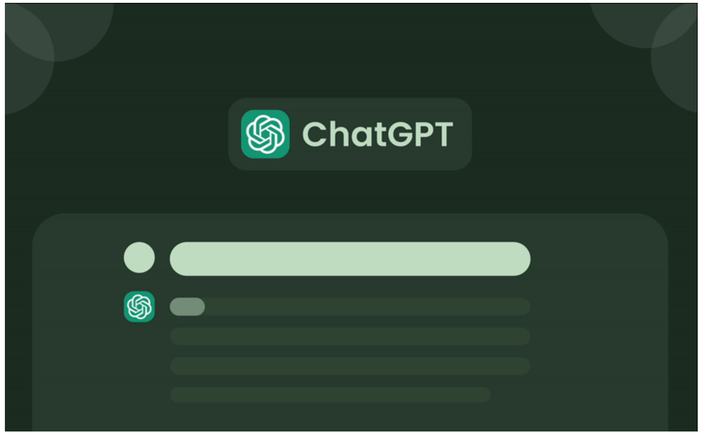
AlphaGo



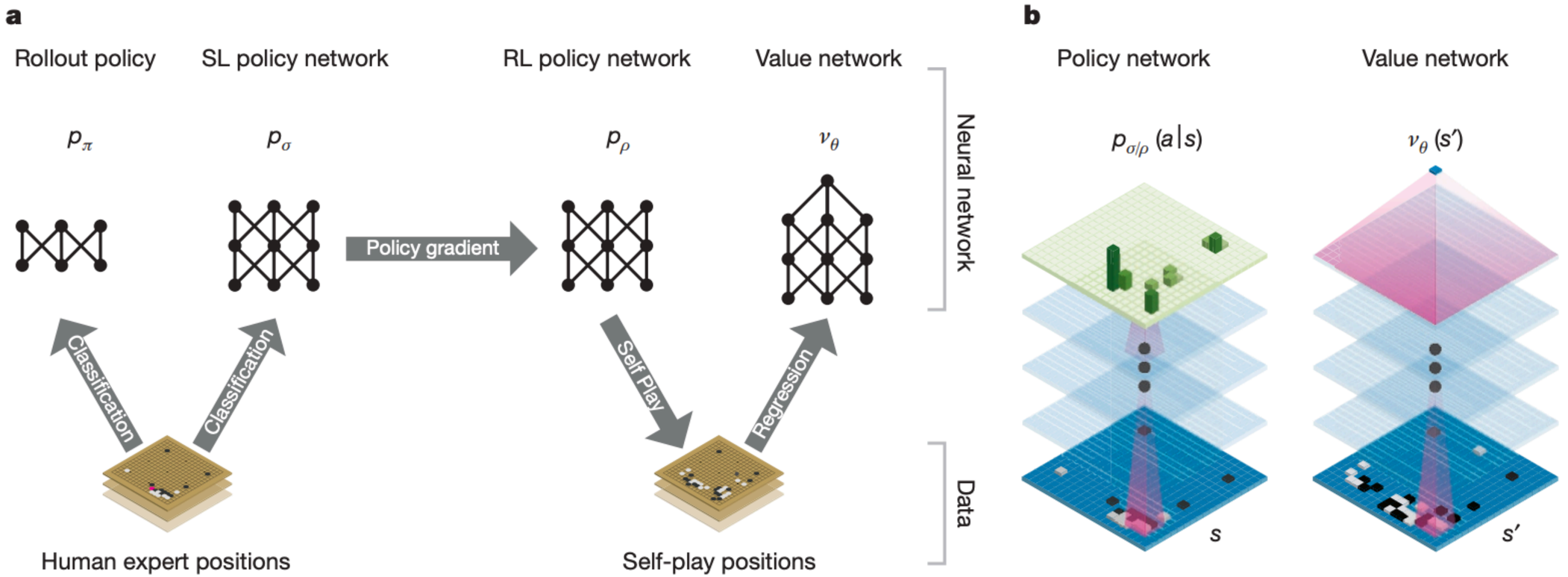
Plasma control



Chatbots



Beyond human performance on Go



[Mastering the game of Go with deep neural networks and tree search, Silver et al. Nature 2017]

ChatGPT with RL

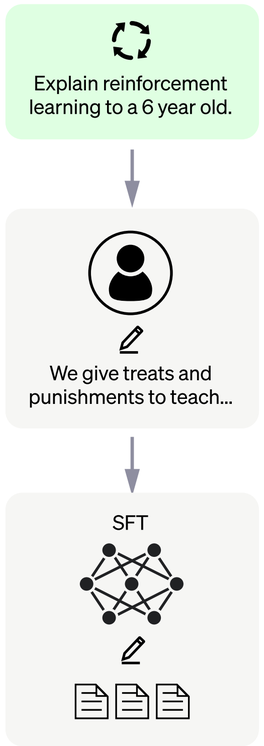
Step 1

Collect demonstration data and train a supervised policy.

A prompt is sampled from our prompt dataset.

A labeler demonstrates the desired output behavior.

This data is used to fine-tune GPT-3.5 with supervised learning.



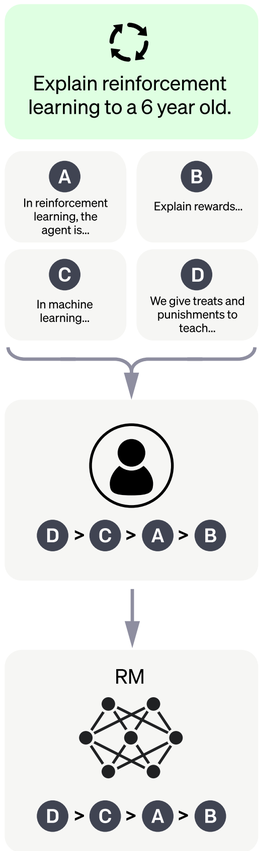
Step 2

Collect comparison data and train a reward model.

A prompt and several model outputs are sampled.

A labeler ranks the outputs from best to worst.

This data is used to train our reward model.



Step 3

Optimize a policy against the reward model using the PPO reinforcement learning algorithm.

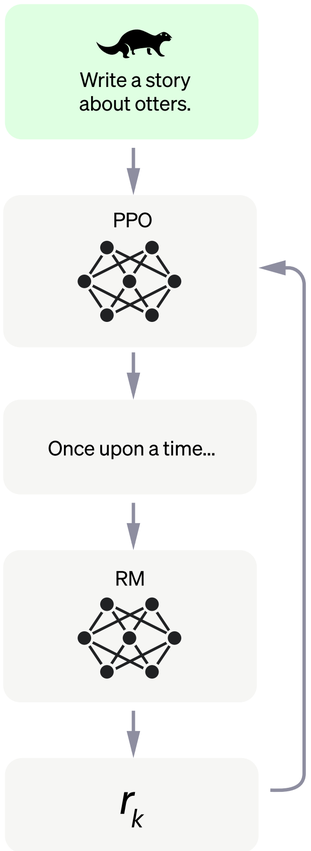
A new prompt is sampled from the dataset.

The PPO model is initialized from the supervised policy.

The policy generates an output.

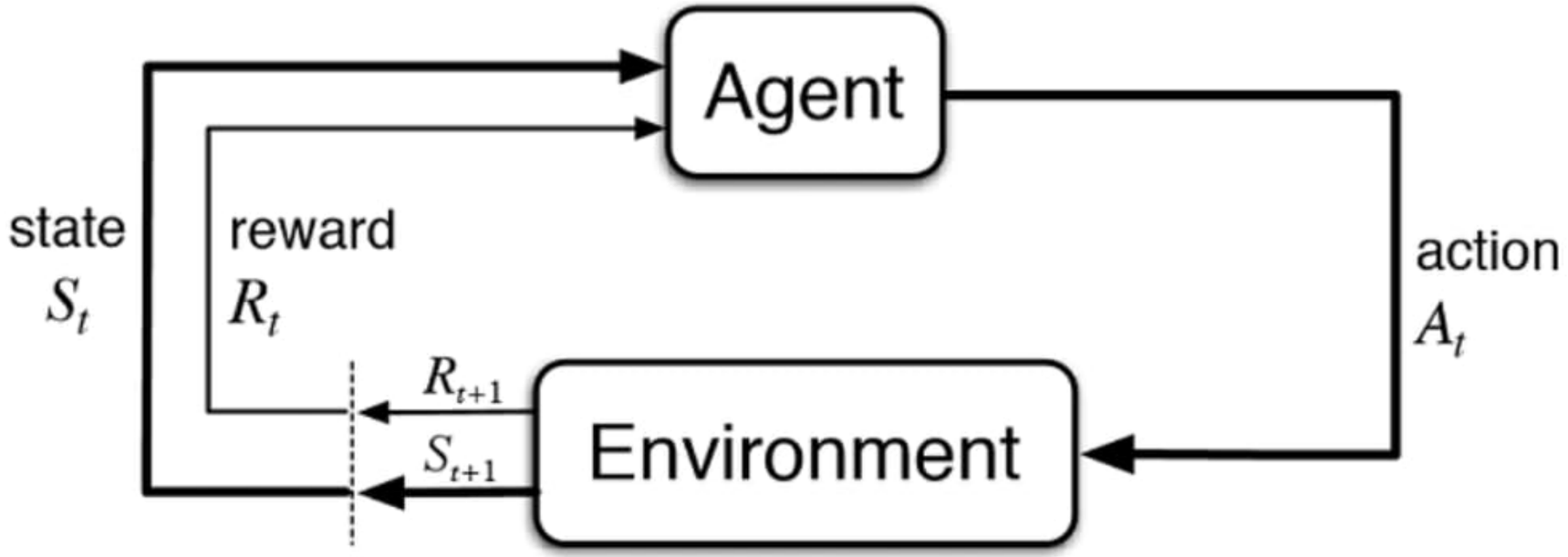
The reward model calculates a reward for the output.

The reward is used to update the policy using PPO.



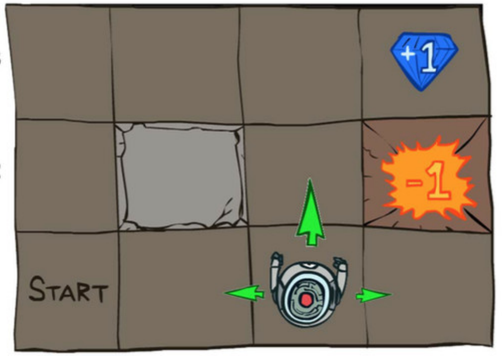
[ChatGPT blogpost]

Reinforcement Learning



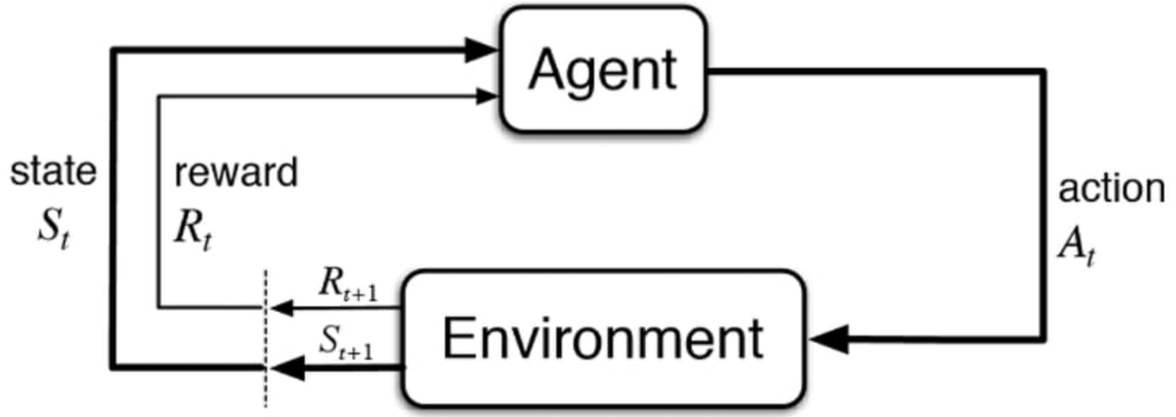
$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2 \ldots$$

Markov Decision Process (MDPs)



An MDP consist of 7 terms:

- Set of states, S
- Set of actions, A
- Transition function P(s' | s, a)
- Reward function R(s, a, s')
- Start state s_0
- Discount factor γ
- Horizon H



$$s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2 \ldots$$

Andrey Andreyevich Markov (1856-1922)

Russian mathematician known for his work on stochastic processes.

Markov assumption + fully observable

A state should summarize all past information and have the Markov property.

$$P(S_{t+1} \mid S_t, S_{t-1}, S_{t-2}, \dots) = P(S_{t+1} \mid S_t)$$

This means the current state S_t contains all necessary information to predict the next state. You don't need the full history.

Suppose your state is s_t = current RGB image

Now imagine a pedestrian stepping into the road.

Is this Markov?

Markov assumption + fully observable

A state should summarize all past information and have the Markov property.

$$P(S_{t+1} \mid S_t, S_{t-1}, S_{t-2}, \dots) = P(S_{t+1} \mid S_t)$$

This means the current state S_t contains all necessary information to predict the next state. You don't need the full history.

From a single image, you **cannot know**:

- Their velocity
- Whether they're accelerating
- Whether they just started moving

Two identical images could lead to very different futures depending on motion history.

So:

$$P(S_{t+1} \mid S_t) \neq P(S_{t+1} \mid S_t, S_{t-1})$$

What is the goal? Return.

We aim to maximize the total discounted reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \qquad \gamma : [0 - 1]$$

Discount factor

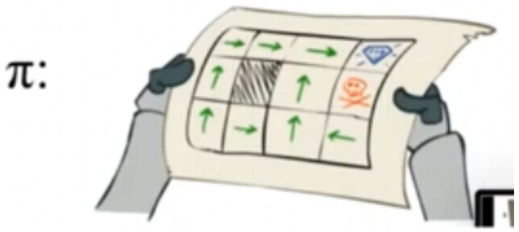
γ close to 0 leads to “myopic” evaluation.

γ close to 1 leads to “far-sighted” evaluation.

What is the solution? A Policy

A policy is a distribution over actions given states

$$\pi(a \mid s) = P(A_t = a \mid S_t = s)$$



- A policy fully defines the behavior of an agent
- The policy is stationary (time-independent)
- During learning, the agent changes its policy as a result of experience.

A deterministic policy is used when agent always chooses exactly one action for each state instead of sampling from a distribution.

$$a = \pi_\theta(s)$$

Sequential Decision Making-Problem examples

Action: Muscle contractions

Observation: sight, smell

Rewards: food

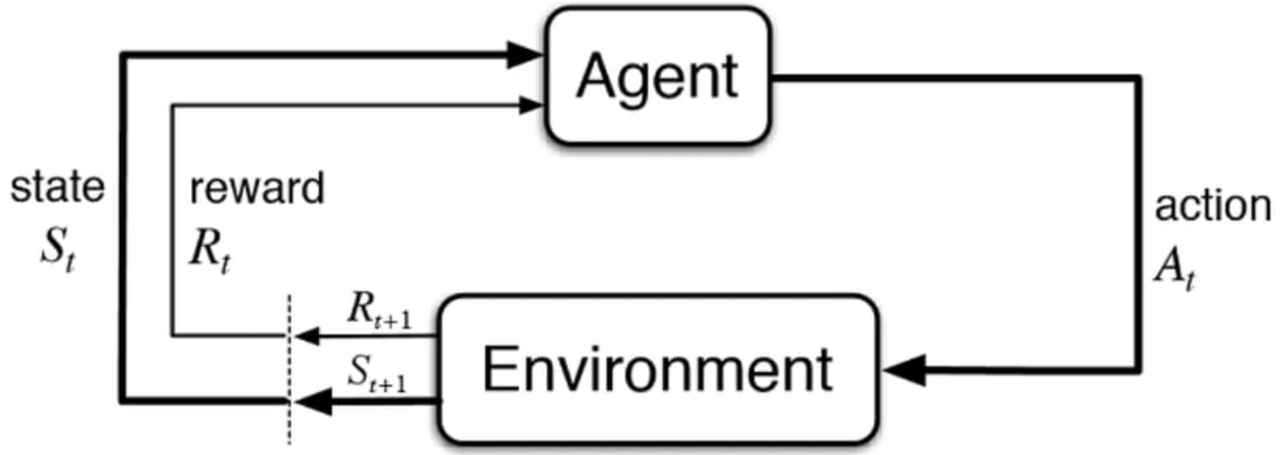
Actions: Motor current or torque

Observations: Camera images

Rewards: task success metrics

Learning the optimal policy to maximize return

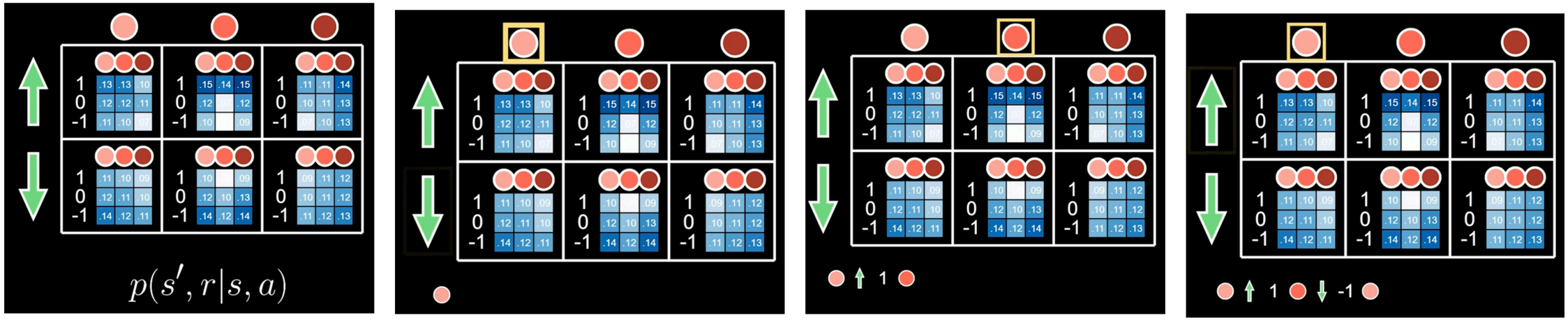
MDP



Return

Goal

Example MDP



Policy: $\pi(a \mid s) = \frac{1}{2}$

Starting Distribution: $\pi(a \mid s) = p_o(s) = \frac{1}{3}$

How to solve MDPs: State Value Functions of Policies.

When given a MDP (S, A, P, R, γ):

Value of a state s under policy π:

$V^{\pi}(s) =$ Expected utility starting in s and acting according to π

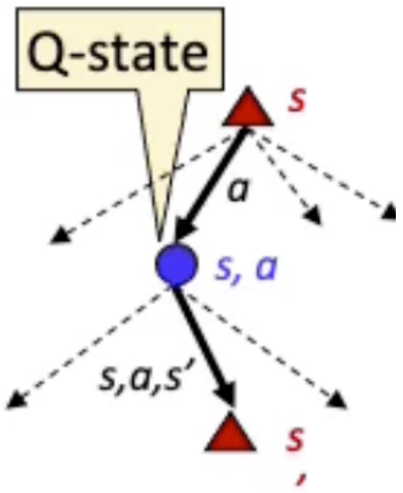
$V^{\pi}(s) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid S_0 = s\right)$ Sequence of rewards generated by following π

$V^{*}(s) =$ Expected utility starting in s and acting optimally

$V^{\pi^*}(s) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid S_0 = s\right)$ Rewards generated by following π^*

How to solve MDPs: Action Value Function of Policies

It is also helpful to define action-value functions



Q-value of taking action a in state s then following policy π

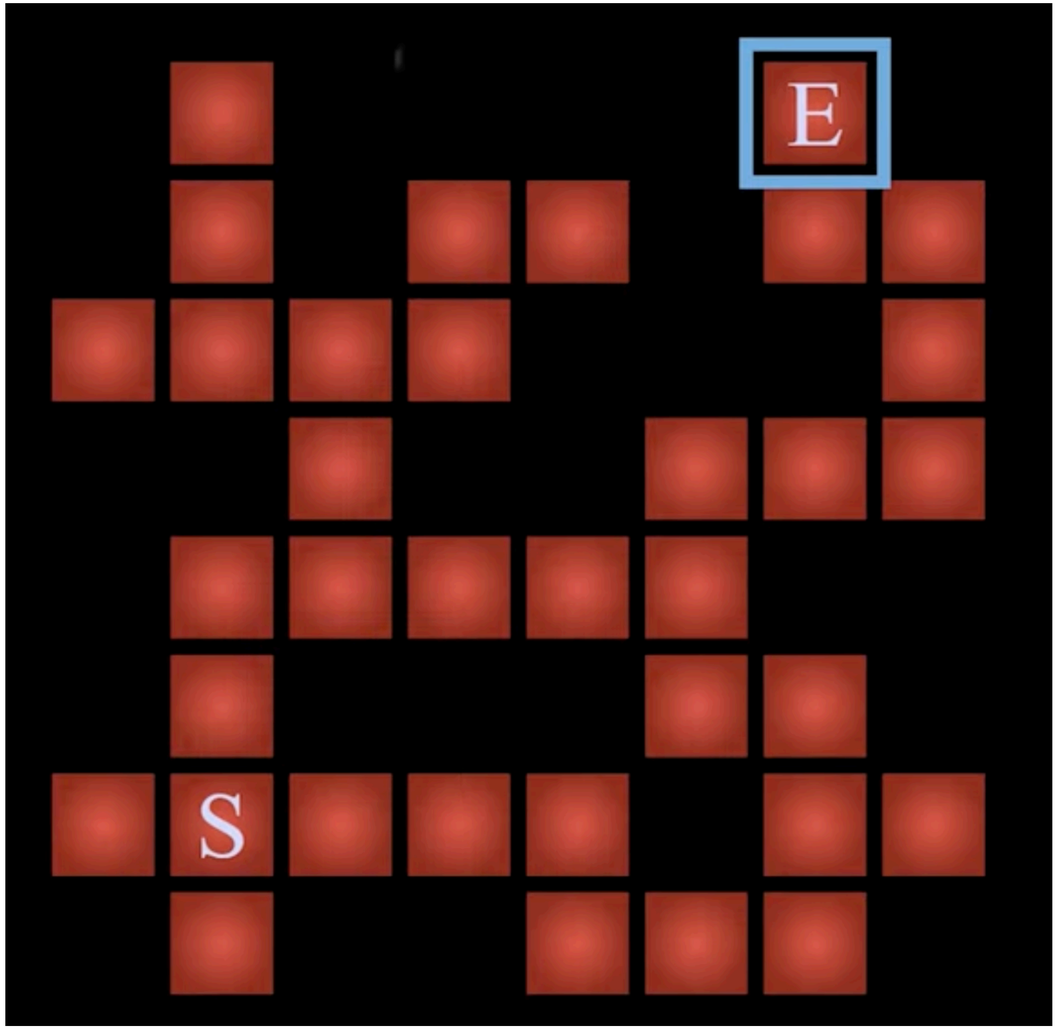
$Q^{\pi}(s, a) =$ Expected utility taking a in s and then following π

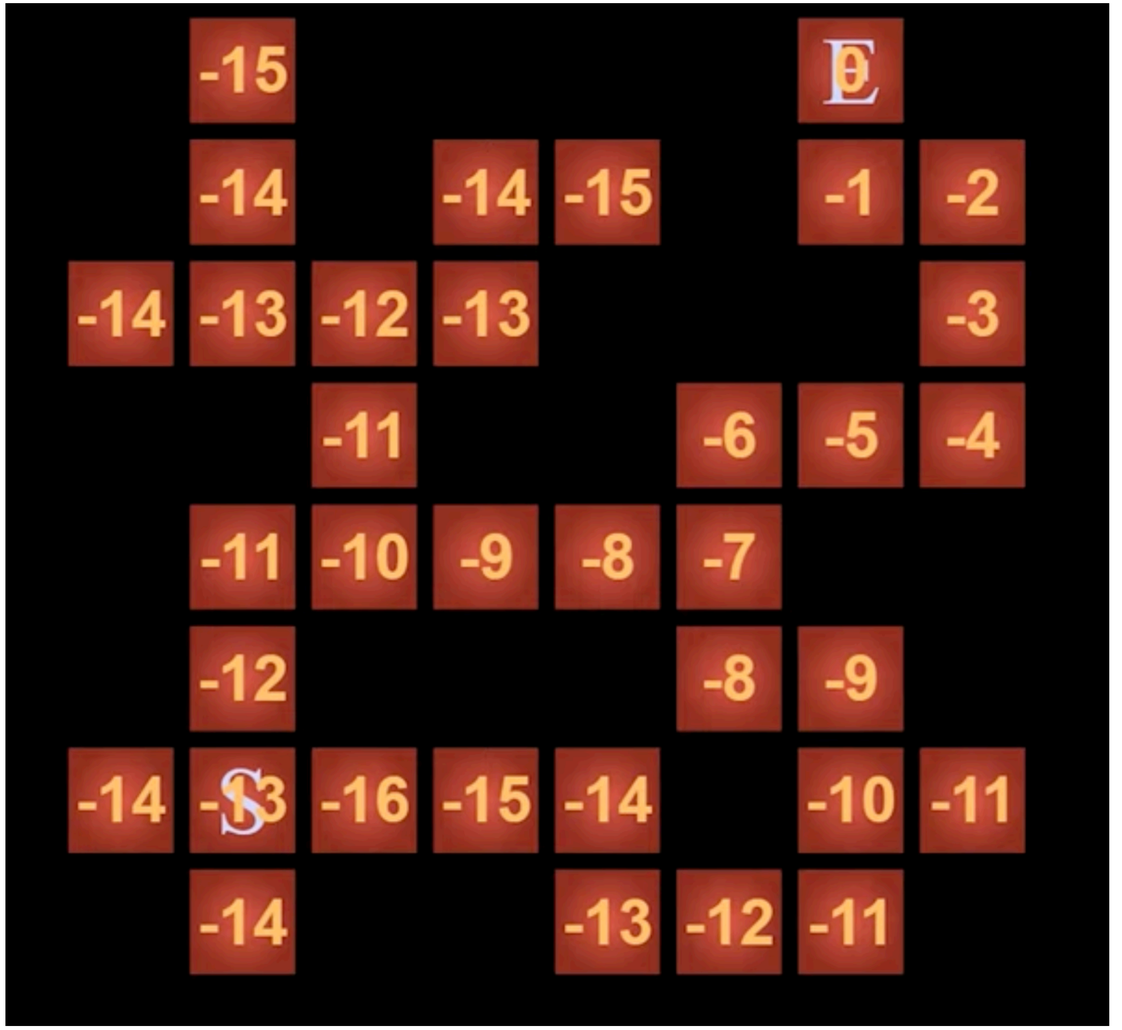
$$Q^{\pi}(s, a) = \mathbb{E}\left(\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid S_0 = s, A_0 = a \right)$$

Optimal Q-value of taking action a in state s : $Q^*(s, a) = Q^{\pi^*}(s, a)$

π^* can be greedily determined from Q^* : $\pi^*(s) = \arg\max_a Q^*(s, a)$

Example of State Value Function

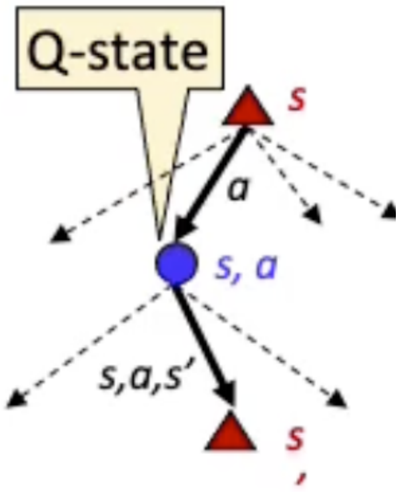




If agent get state value function of the optimal policy, can agent solve this?

Solving MDPs: Bellman Equations

The Bellman equations connect values functions at consecutive time steps:



$$V^*(s) = \max_{a \in A} Q^*(s, a)$$

Optimal value of s is what we get by picking the optimal action

$$Q^*(s, a) = \sum_{s' \in S} P(s' \mid s, a)\left[r(s, a, s') + \gamma V^*(s')\right]$$

Expected value over successor state s'

Current reward + discounted future reward

$$V^*(s) = \max_{a \in A}\left(\sum_{s' \in S} P(s' \mid s, a)\left[r(s, a, s') + \gamma V^*(s')\right)\right.$$

Solving MDPs with known P and R: Value Iteration

Bellman equation give us a recursive definition of the optimal value:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a) \Big[R(s, a, s') + \gamma V^*(s') \Big]$$

Idea: solve iteratively via dynamic programming (DP)

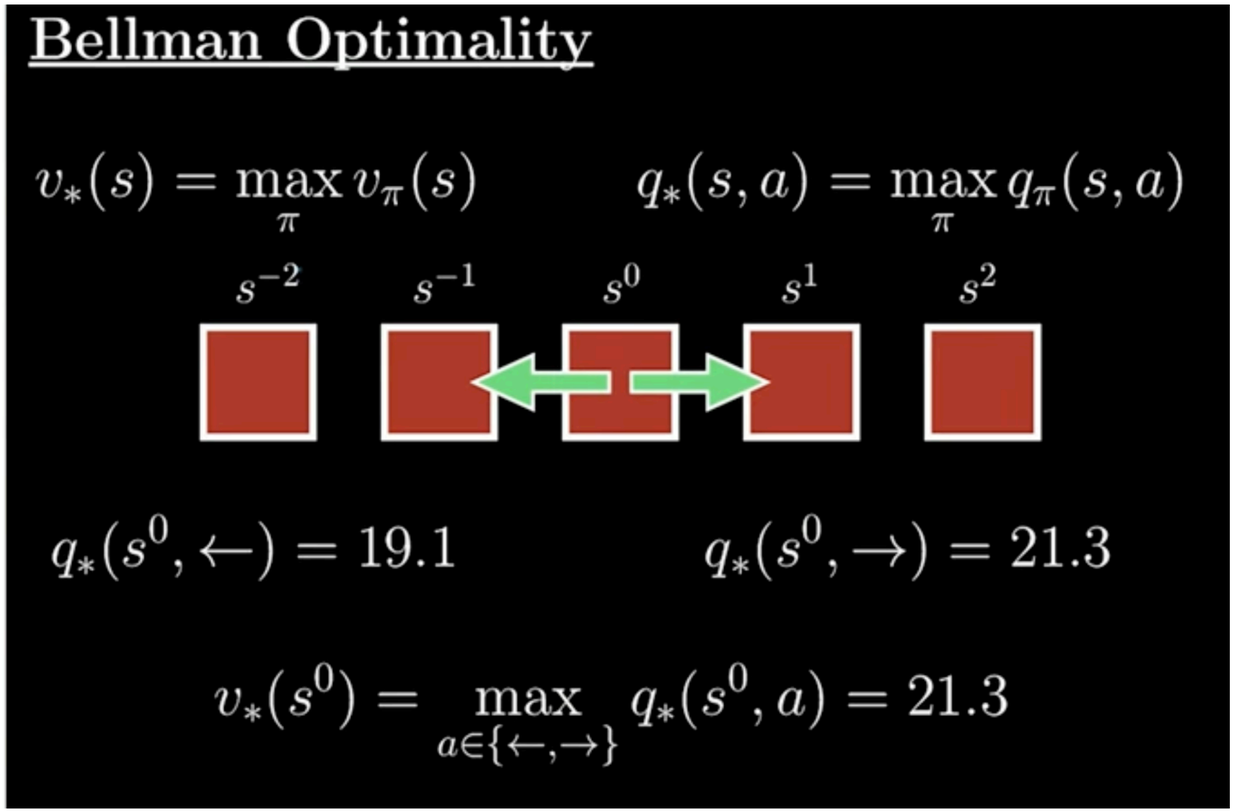
Start with V_0 (s) = 0 for all states s

DP: refers to algorithms used to find optimal policies which have complete knowledge of the environment as an MDP.

Iterate the Bellman update until convergence:

$$V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a) \Big[R(s, a, s') + \gamma V_i(s') \Big]$$

Example Bellman Optimality

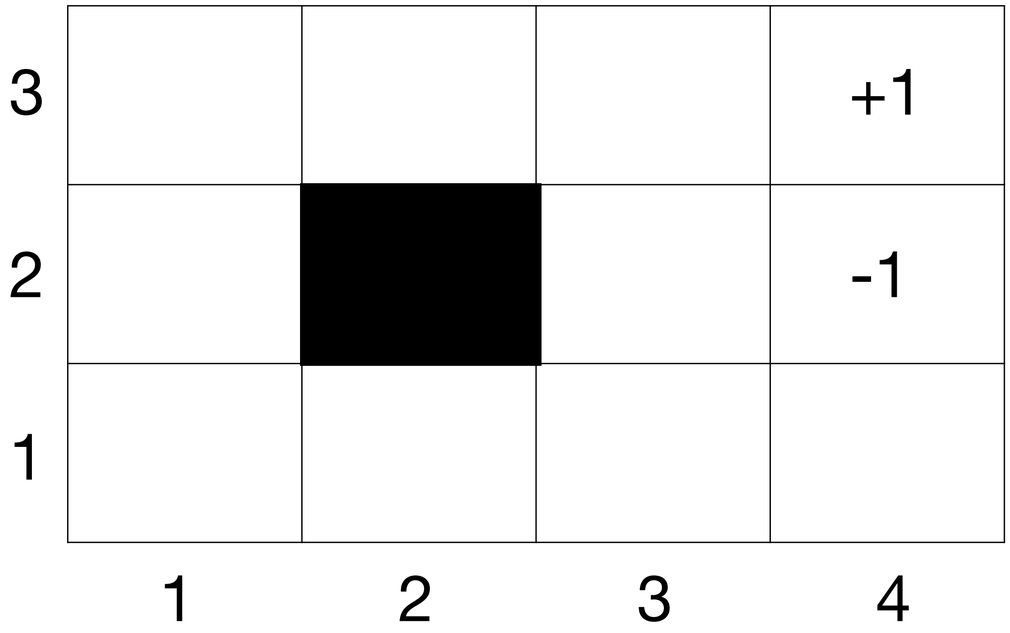


* Because there may be multiple actions which achieve the highest action-value, the choose highest action value rule cannot differentiate among them. This means any policy that assigns non-zero probabilities only to these actions is an optimal policy.

Example: Value iteration

Bellman update rule: $V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a) \left[R(s, a, s') + \gamma V_i(s') \right]$

Example MDP



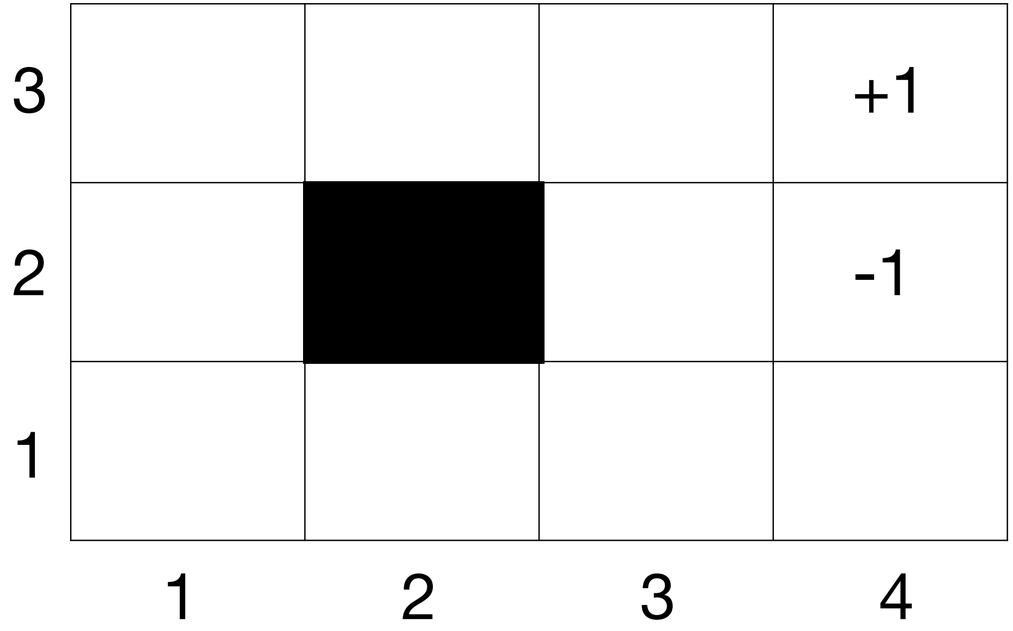
Rewards given in terminal states

$\gamma = 0.9$, living reward = 0, noise = 0.2

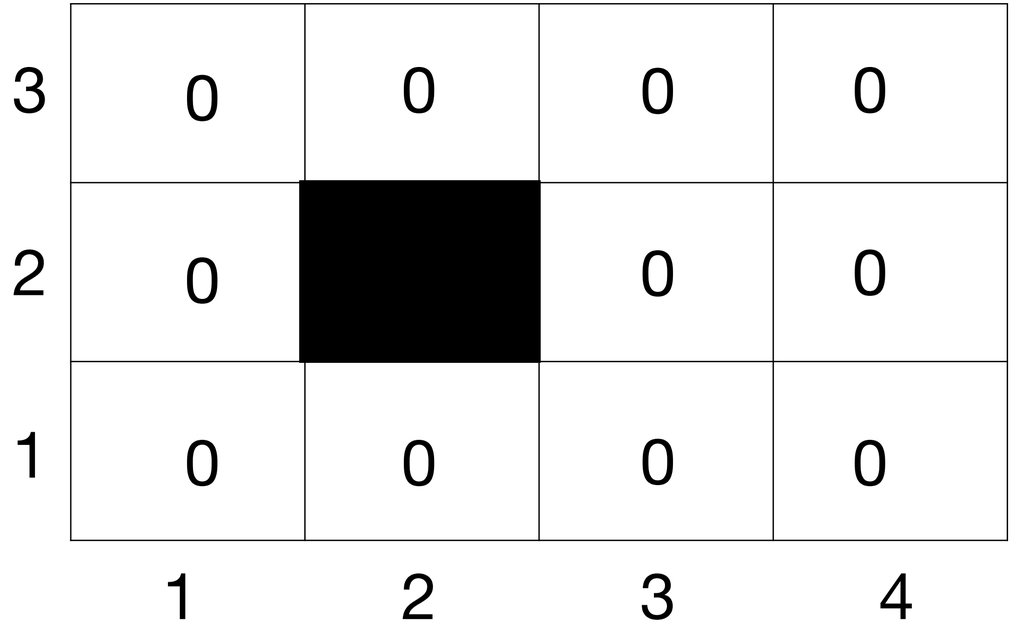
Example: Value iteration

Bellman update rule: $V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a) \left[R(s, a, s') + \gamma V_i(s') \right]$

Example MDP



V_0



$\gamma = 0.9$, living reward = 0, noise = 0.2

$V_0(s) = 0$

Example: Value iteration

Bellman update rule: $V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a) \left[R(s, a, s') + \gamma V_i(s') \right]$

V_0

3	0	0	0	0
2	0		0	0
1	0	0	0	0
	1	2	3	4

$V_0(s) = 0$

V_1

3	0	0	0	+1
2	0		0	-1
1	0	0	0	0
	1	2	3	4

$V_1(s) = 0$

$\gamma = 0.9$, living reward = 0, noise = 0.2

Example: Value iteration

Bellman update rule: $V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a)\left[R(s, a, s') + \gamma V_i(s')\right]$

V_1

3	0	0	0	+1
2	0		0	-1
1	0	0	0	0
	1	2	3	4

$V_1(s) = 0$

V_2

3	0	0	0	+1
2	0		0	-1
1	0	0	0	0
	1	2	3	4

Assume you move ->

0.8 [0+0.9x1] + 0.1 [0+0.9x0] + 0.1[0+0.9x0] =0.72

$\gamma = 0.9$, living reward = 0, noise = 0.2

Example: Value iteration

Bellman update rule: $V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a)\left[R(s, a, s') + \gamma V_i(s')\right]$

V_1

3	0	0	0	+1
2	0		0	-1
1	0	0	0	0
	1	2	3	4

$V_1(s) = 0$

V_2

3	0	0	0.72	+1
2	0		0	-1
1	0	0	0	0
	1	2	3	4

Assume you move ->

0.8 [0+0.9x1] + 0.1 [0+0.9x0] + 0.1[0+0.9x0] =0.72

$\gamma = 0.9$, living reward = 0, noise = 0.2

Example: Value iteration

Bellman update rule: $V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a)\left[R(s, a, s') + \gamma V_i(s')\right]$

V_1

3	0	0	0.72	+1
2	0		0	-1
1	0	0	0	0
	1	2	3	4

V_2

3	0	?	?	+1
2	0		?	-1
1	0	0	0	0
	1	2	3	4

$\gamma = 0.9$, living reward = 0, noise = 0.2

Example: Value iteration

Bellman update rule: $V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a)\left[R(s, a, s') + \gamma V_i(s')\right]$

V_1

3	0	0	0.72	+1
2	0		0	-1
1	0	0	0	0
	1	2	3	4

V_2

3	0	0.52	0.78	+1
2	0		0.43	-1
1	0	0	0	0
	1	2	3	4

$\gamma = 0.9$, living reward = 0, noise = 0.2

Information propagates outward from terminal states

Eventually all states will have correct value estimates

Policy iteration: policy evaluation

How do we compute V's for a fixed policy?

Key idea: Bellman updates for arbitrary policy

$$V_0^\pi(s) = 0$$

$$V_{i+1}^\pi(s) \leftarrow \sum_{s'} P(s' \mid s, \pi(s)) \Big[R(s, \pi(s), s') + \gamma V_i^\pi(s') \Big]$$

Value iteration update rule

$$V_{i+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s' \mid s, a) \Big[R(s, a, s') + \gamma V_i(s') \Big]$$

Generalized Policy Iteration: Policy Iteration

Repeat two steps until convergence

1. Policy evaluation: keep current policy π fixed, find value function V^{π}

Iterate simplified Bellman update until values converge:

$$V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} P(s' \mid s, \pi_k(s)) \Big[R(s, \pi_k(s), s') + \gamma V_i^{\pi_k}(s') \Big]$$

2. Policy improvement: find the best action for V^{π} via one-step lookahead

$$\pi_{k+1}(s) = \arg\max_a \sum_{s'} P(s' \mid s, a) \Big[R(s, a, s') + \gamma V^{\pi_k}(s') \Big]$$

Policy iteration is optimal too!

>Faster than value iteration in terms of number of (outer) loops, but remember that step 1 has an inner loop too.

Find the optimal Pi and V: $\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$

Temporal Differencing (TD)

Policy evaluation: Start V_0 (s) =0

Iterate until convergence: $V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} P(s' \mid s, \pi_k(s)) \left[R(s, \pi_k(s), s') + \gamma V_i^{\pi_k}(s') \right]$

How can we extend this to when P and R is not known, and only revealed gradually through experience?

Every time you take action a from state s, you get a sample from the unknown P and corresponding reward R.

Temporal Differencing (TD)

Policy evaluation: Start V_0 (s) =0

Iterate until convergence: $V_{i+1}^{\pi_k}(s) \leftarrow \sum_{s'} P(s' \mid s, \pi_k(s)) \Big[R(s, \pi_k(s), s') + \gamma V_i^{\pi_k}(s') \Big]$

Key idea: Treat single sample you get as representative of the distribution, and apply an incremental update to reduce the “Bellman error”:

One sample of V(S): $\text{sample} = R(s, \pi(s), s') + \gamma V_i^{\pi}(s')$

TD update: $V^{\pi}(s) \leftarrow V^{\pi}(s) + \alpha\big(\text{sample} - V^{\pi}(s)\big)$

Doing this for each sample = computing the running average over samples.

Learning the optimal Q* function

Recall Bellman equation for optimal Q*

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right]$$

The corresponding Q-value iteration equation (analogous to the state value iteration would be:

$$Q_{i+1}(s,a) \leftarrow \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \left[R(s,a,s') + \gamma \max_{a'} Q_i(s',a') \right]$$

Again, this require access to P and R of which only have samples from experience.

Applying the TD to previous one

Q-value iteration $$Q_{i+1}(s,a) \leftarrow \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\left[R(s,a,s') + \gamma \max_{a'} Q_i(s',a')\right]$$

TD: treat single sample as representative of the distribution and apply an incremental update to reduce the “Bellman error”:

1. Execute a single action a from state s and observe s’ and R:
   sample $= R(s,a,s') + \gamma \max_{a'} Q^*(s',a')$
2. Then incremental TD update is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(R(s,a,s') + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

Bellman error

This is Q-Learning.

Example of Q-Learning

Solving unknown MDPs using function approximation

Q*(s,a) = expected utility starting in s, taking action a, and thereafter acting optimally.

Bellman Equation: $$Q^*(s,a) = \sum_{s'} P(s' \mid s,a)\left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right]$$

Q-value iteration: $$Q^*_{k+1}(s,a) \leftarrow \sum_{s'} P(s' \mid s,a)\left(R(s,a,s') + \gamma \max_{a'} Q^*_k(s',a')\right)$$

This is problematic when we do not known this transition function.

Slides from Fragkiadaki

Tabular Q-Learning

- Q-value iteration: $Q_{k+1}(s,a) \leftarrow \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma \max_{a'} Q_k(s',a'))$
- Rewrite as expectation: $Q_{k+1} \leftarrow \mathbb{E}_{s' \sim P(s'|s,a)} \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$
- (Tabular) Q-Learning: replace expectation by samples
  - For an state-action pair (s,a), receive: $s' \sim P(s'|s,a)$ simulation and exploration
  - Consider your old estimate: $Q_k(s,a)$
  - Consider your new sample estimate:

$$\text{target}(s') = r(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

$$\text{error}(s') = \left(r(s,a,s') + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a) \right)$$

Slides from Fragkiadaki

Tabular Q-Learning update

learning rate

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha \ \text{error}(s')$$
$$= Q_k(s,a) + \alpha \left(r(s,a,s') + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a) \right)$$

Key idea: implicitly estimate the transitions via simulation

Slides from Fragkiadaki

Tabular Q-Learning overview

Bellman optimality

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right]$$

Algorithm:

Start with $Q_0(s,a)$ for all s, a.

Get initial state s

For k = 1, 2, … till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = r(s,a,s')$$

Sample new initial state s'

else:

$$\text{target} = r(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha\left(r(s,a,s') + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a)\right)$$

$$s \leftarrow s'$$

Slides from Fragkiadaki

Tabular Q-Learning overview

Bellman optimality

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right]$$

Algorithm:

Start with $Q_0(s,a)$ for all s, a.

Get initial state s

For k = 1, 2, … till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = r(s,a,s')$$

Sample new initial state s'

else:

$$\text{target} = r(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha\left(r(s,a,s') + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a)\right)$$

$$s \leftarrow s'$$

Slides from Fragkiadaki

Tabular Q-Learning overview

Algorithm:

Start with $Q_0(s, a)$ for all s, a.

Get initial state s

For k = 1, 2, … ~~till convergence~~

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = r(s, a, s')$$

Sample new initial state s'

else:

$$\text{target} = r(s, a, s') + \gamma \max_{a'} Q_k(s', a')$$

$$Q_{k+1}(s, a) = Q_k(s, a) + \alpha \left(r(s, a, s') + \gamma \max_{a'} Q_k(s', a') - Q_k(s, a) \right)$$

$$s \leftarrow s'$$

Bellman optimality

$$Q^*(s, a) = \mathbb{E}_{s'} \left[r(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right]$$

How do we sample these actions?

>Choose random action all the time.

>Choose action that maximize Qk(s,a) greedily.

>Choose action sometimes randomly with prob of E, otherwise choose greedily..

Slides from Fragkiadaki

Epsilon-greedy

Initially, Q(s,a) is poor at the beginning, bad initial estimates in the first few cases can drive policy into sub-optimal region and never explore further.

$$\pi(s) = \begin{cases} \max_a \hat{Q}(s,a) & \text{with probability } 1-\epsilon \\ \text{random action} & \text{otherwise} \end{cases}$$

Gradually decrease epsilon as policy is learned.

Slides from Fragkiadaki

Tabular Q-Learning overview

Algorithm:

Start with $Q_0(s,a)$ for all s, a.

Get initial state s

For k = 1, 2, … till convergence

Sample action a, get next state s'

If s' is terminal:

$$\text{target} = r(s,a,s')$$

Sample new initial state s'

else:

$$\text{target} = r(s,a,s') + \gamma \max_{a'} Q_k(s',a')$$

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha \left(r(s,a,s') + \gamma \max_{a'} Q_k(s',a') - Q_k(s,a) \right)$$

$$s \leftarrow s'$$

Bellman optimality

$$Q^*(s,a) = \mathbb{E}_{s'} \left[r(s,a,s') + \gamma \max_{a'} Q^*(s',a') \right]$$

How do we sample these actions?

>Choose random action all the time.

>Choose action that maximize Qk(s,a) greedily.

>Choose action sometimes randomly with prob of E, otherwise choose greedily..

Slides from Fragkiadaki

Convergence

- Amazing result: Q-learning converges to optimal policy -- even if you're acting suboptimally!
- This is called off-policy learning
- Caveats:
  - You have to explore enough
  - You have to eventually make the learning rate small enough
  - … but not decrease it too quickly

Limitation: Still requires small and discrete state and action space, as it is tabular learning, it keeps a |S| X |A| table of Q(s,a).

Slides from Abbeel

Problems with Q-Learning

In many real situations, we cannot possibly learn about every single state+action!

>Too many state-action pairs to visit them all in training

>Too many state-action pairs to hold the q-tables in memory

Ideally what we want:
-Generalize by learning about some small number of training q-states from experience

-Generalize that experience to new, similar q-states

-Core idea in ML, and we will see it over and over again.

Example with Pacman

>We discover through experience that this is a bad state:

>In naive Q-learning, we know nothing about this state or its Q states:

>Or even this:

Deep Q-Learning

Solution: Describe a state using a vector of features

Predict Q-values with a deep neural network

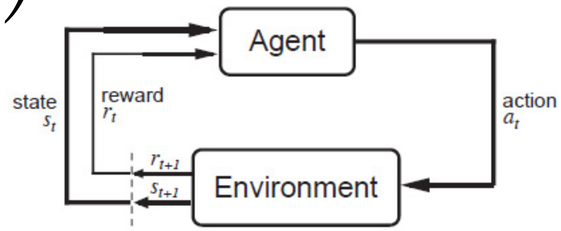
Input: the state

Output: Q-value of various actions

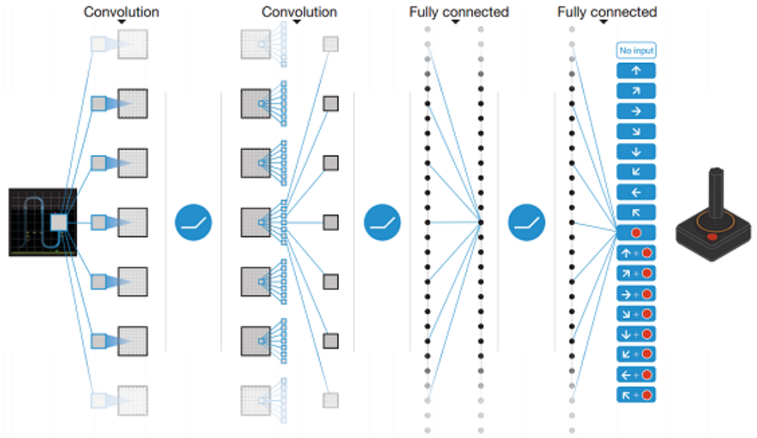
Learning: gradient descent with the squared Bellman error loss:

$$\left(\left(R(s,a,s') + \gamma \max_{a'} Q(s',a') \right) - Q(s,a) \right)^2$$

The policy action is the one with the highest predicted Q-value.



AlphaGo



Deep Q-Learning

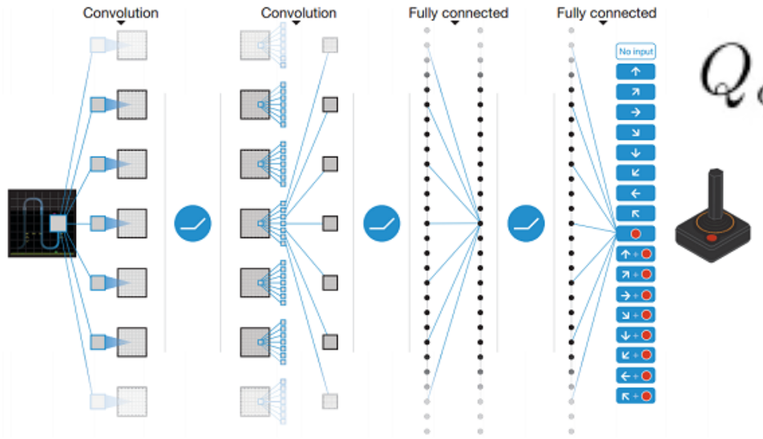
1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r_i + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi(\mathbf{s}_i, \mathbf{a}_i)}{d\phi} \left(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\right)$

$$= \frac{d}{d\phi}(Q_\phi - y_i)^2$$

Incremental update step → gradient descent* on the squared Bellman error loss!

S

A



$Q_\phi(\mathbf{s}, \mathbf{a})$

Value-based VS Policy-based RL

>Value based : Learned value function, Implicit policy (e.g E-greedy)

>Policy based : No value function, just directly learn a policy.

Why do we want to learn directly the policy?

- Often π can be simpler than Q or V
  - E.g., robotic grasp

Q(s,a) and V(s) very high-dimensional
But policy could be just 'open/close hand'

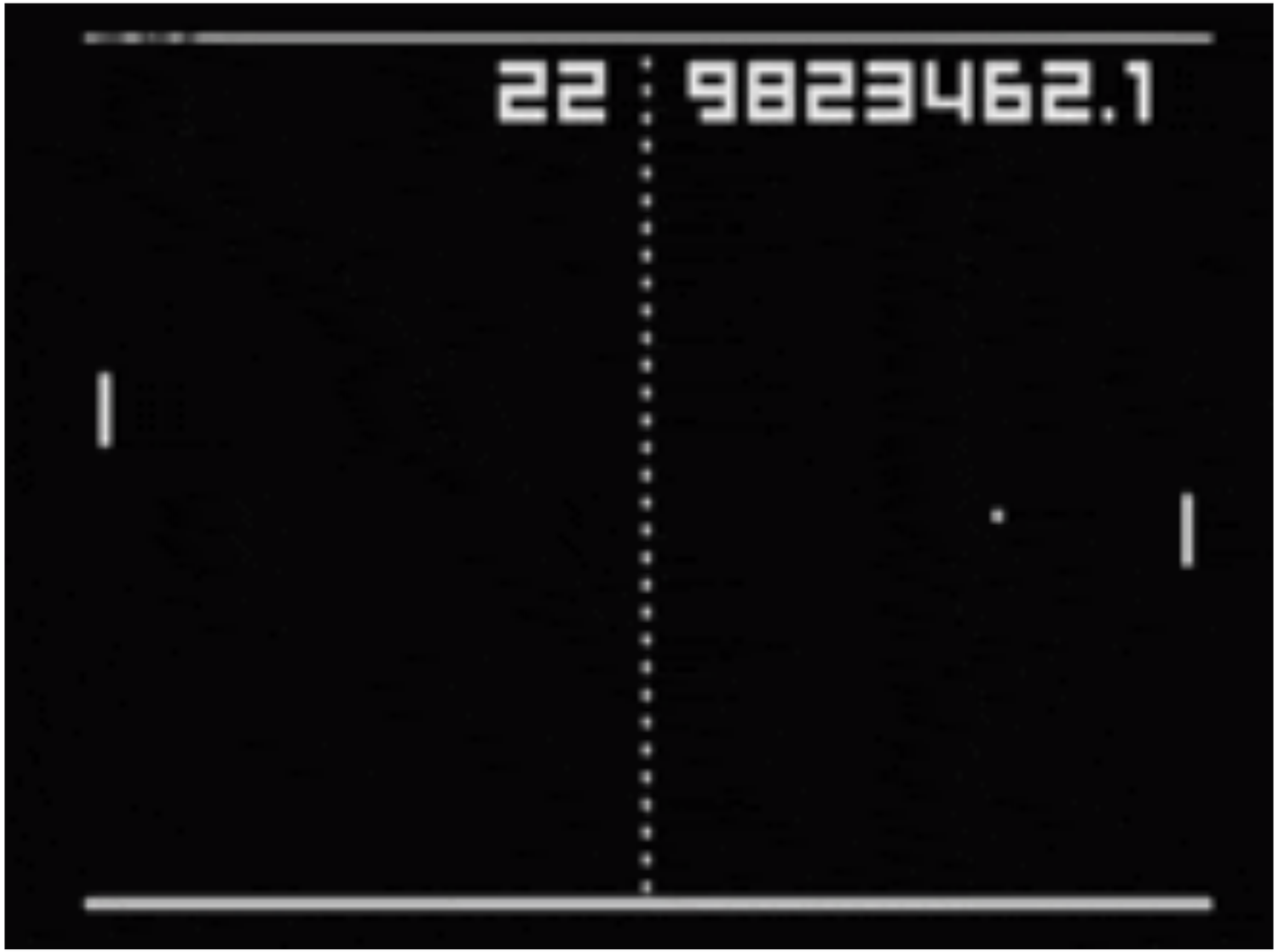
- V: doesn't prescribe actions
  - Would need dynamics model (+ compute 1 Bellman back-up)
- Q: need to be able to efficiently solve $\arg\max_u Q_\theta(s,u)$
  - Challenge for continuous / high-dimensional action spaces*

$$\pi^*(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \arg\max_a \mathbb{E}_{s'}\left[r(s,a,s') + \gamma V^*(s')\right] \\ \epsilon, & \text{else} \end{cases} \qquad \pi^*(a|s) = \begin{cases} 1-\epsilon, & \text{if } a = \arg\max_a Q^*(s,a) \\ \epsilon, & \text{else} \end{cases}$$

Value-based VS Policy-based RL

	Policy-based	**Value-based**
Conceptually:	Optimize what you care about	Indirect, exploit the problem structure, self-consistency
Empirically:	More compatible with rich architectures	More compatible with exploration and off-policy learning
	More versatile	More sample-efficient when they work.
	More compatible with auxiliary objectives	

Policy Gradient Methods

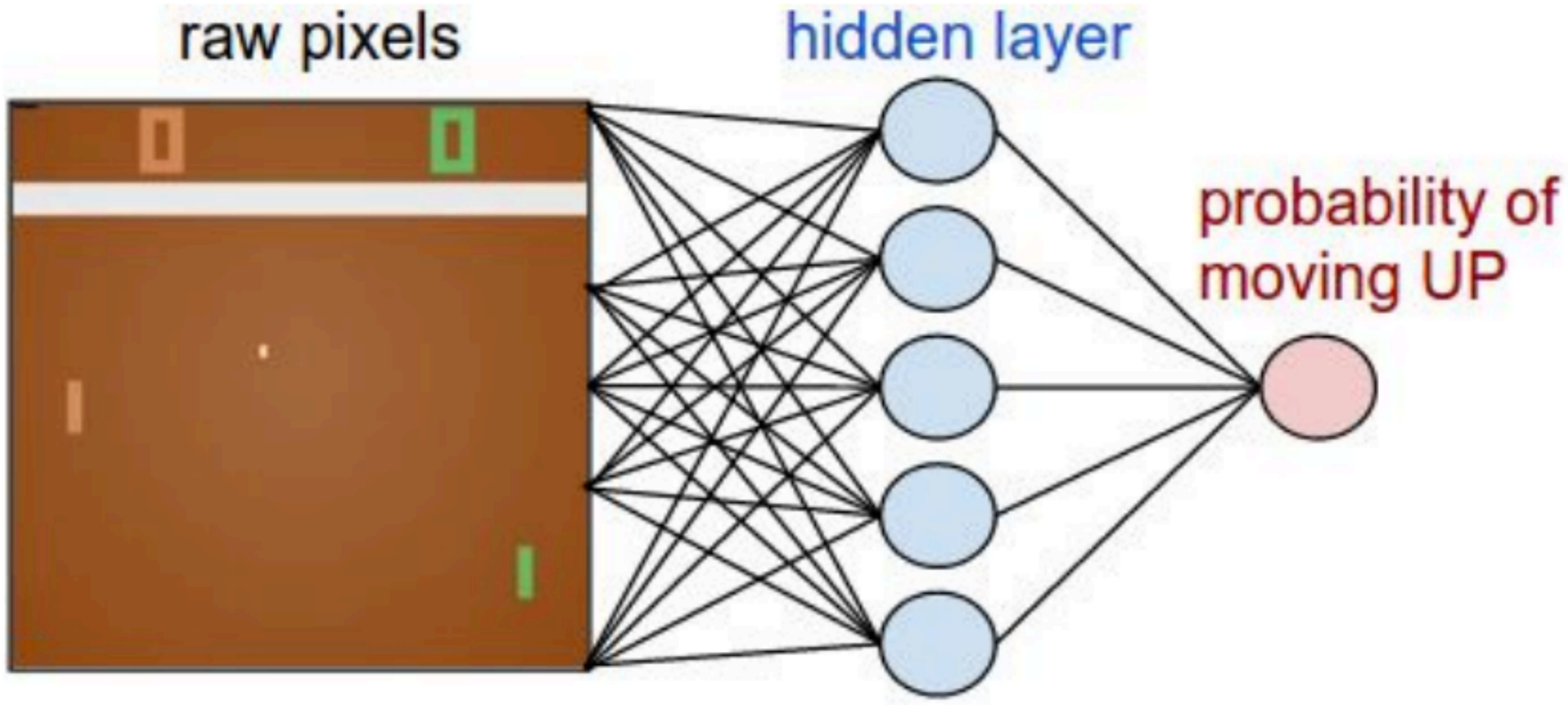


Pong from pixels

e.g.,

height width

[80 x 80]
array of



NN see +1 if it scored a point, -1 if it was scored against. How do we learn these parameters?

Slides from Karpathy

Pong from pixels

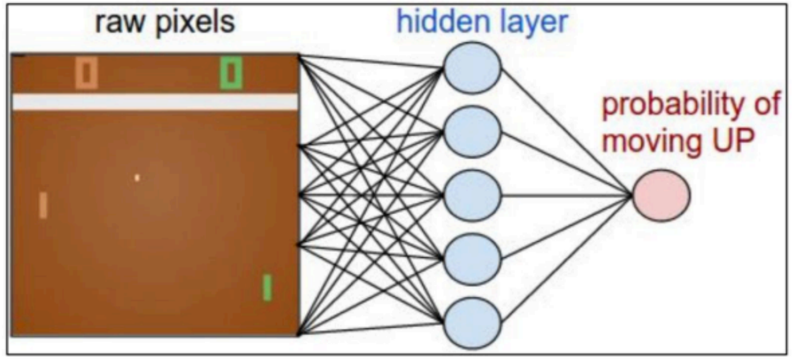
Given training labels from a human expert.

(x1,UP)
(x2,DOWN)
(x3,UP)
...

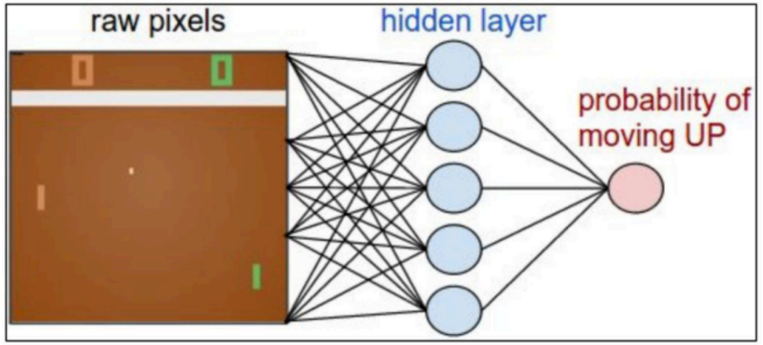
Maximum Likelihood Estimation (MLE).

maximize:

$$\sum_i \log p(y_i | x_i)$$



Except, we don't have labels...

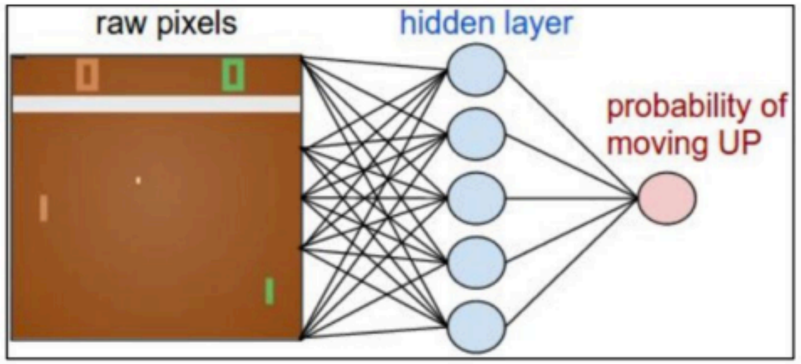


Should we go UP or DOWN?

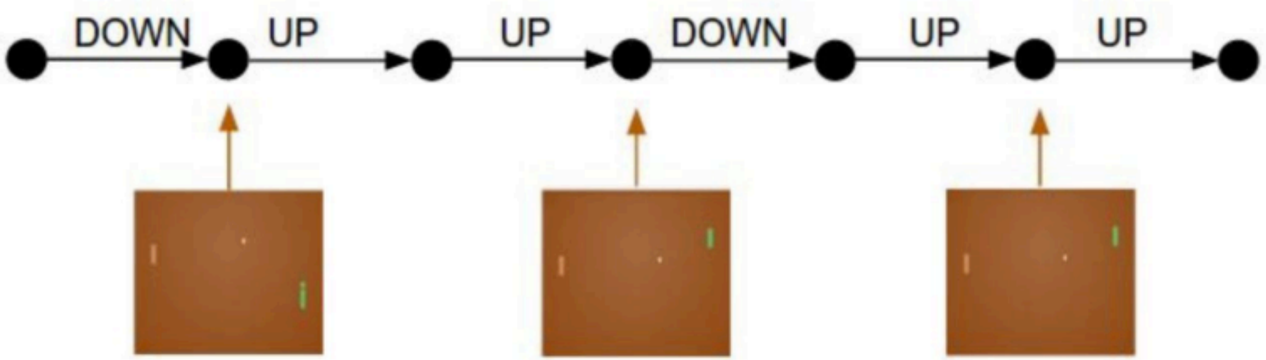
Slides from Karpathy

Pong from pixels

No data, lets just act according to our current policy.



Rollout the policy and collect an episode



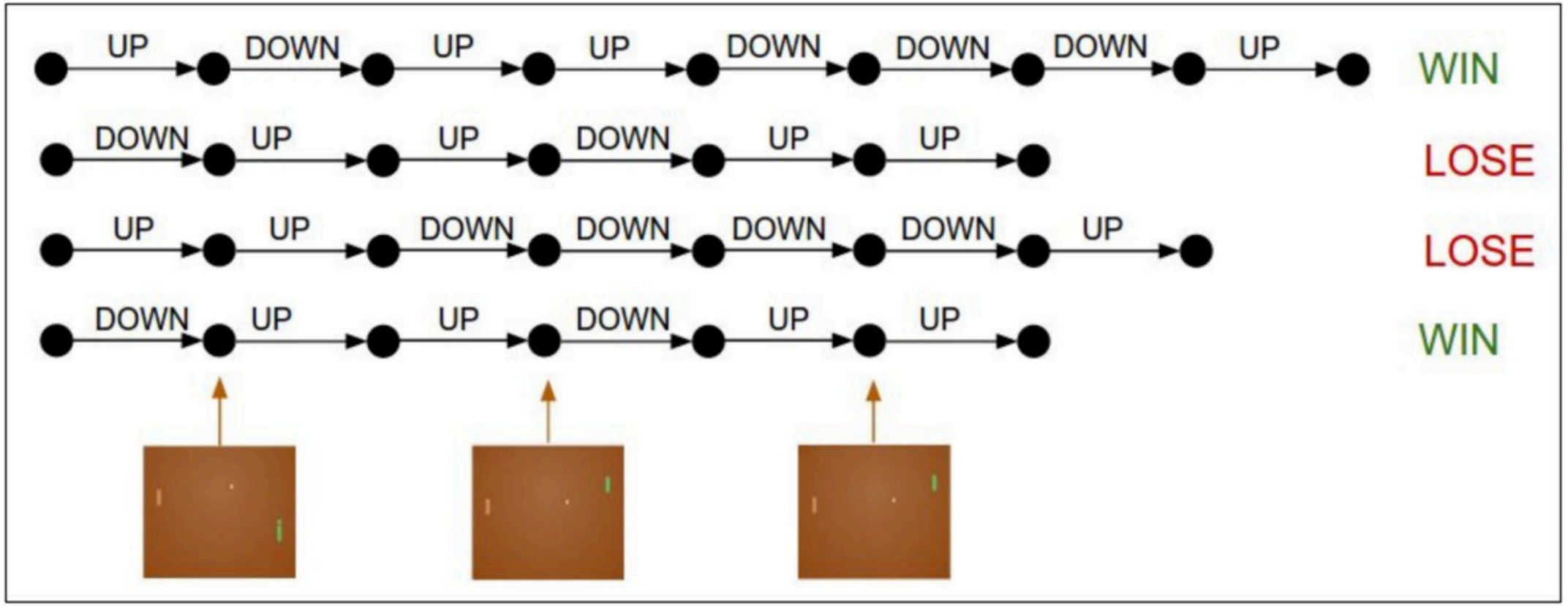
WIN

Slides from Karpathy

Pong from pixels

Collect many rollouts...

4 rollouts:

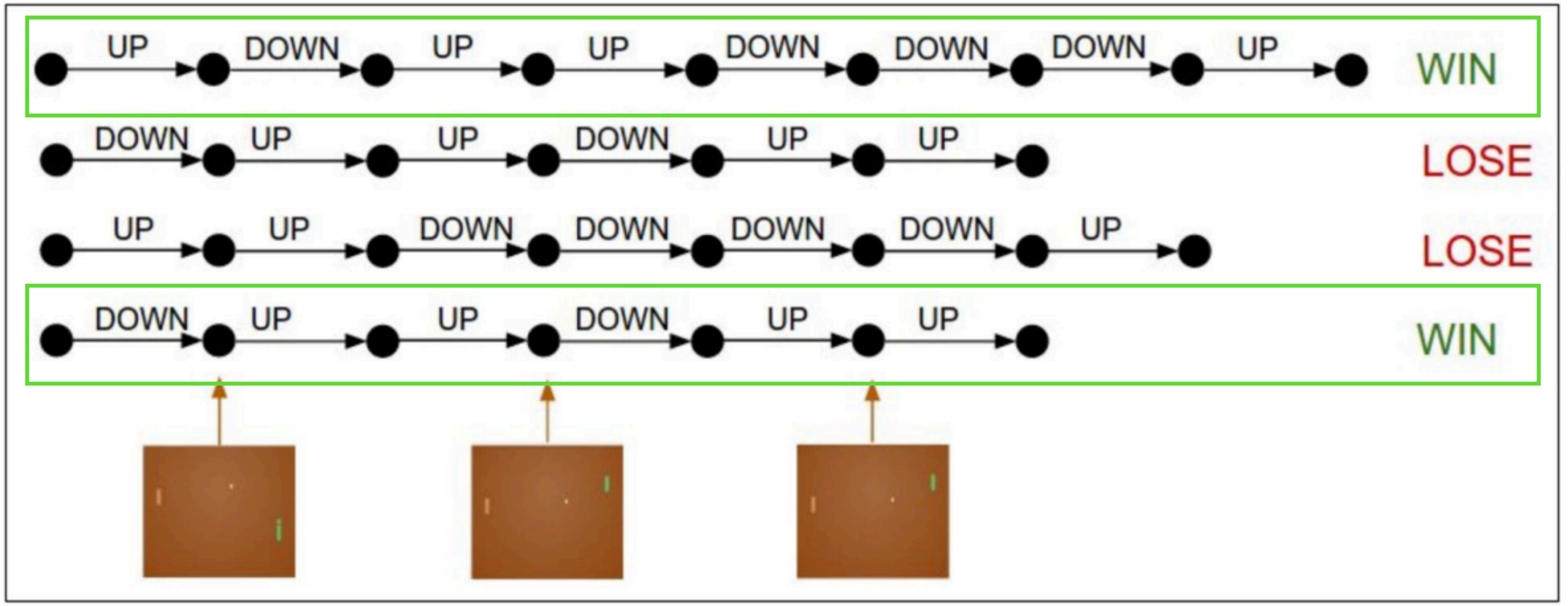


Slides from Karpathy

Pong from pixels

I am not sure what we did here, but these are good actions cause we won.

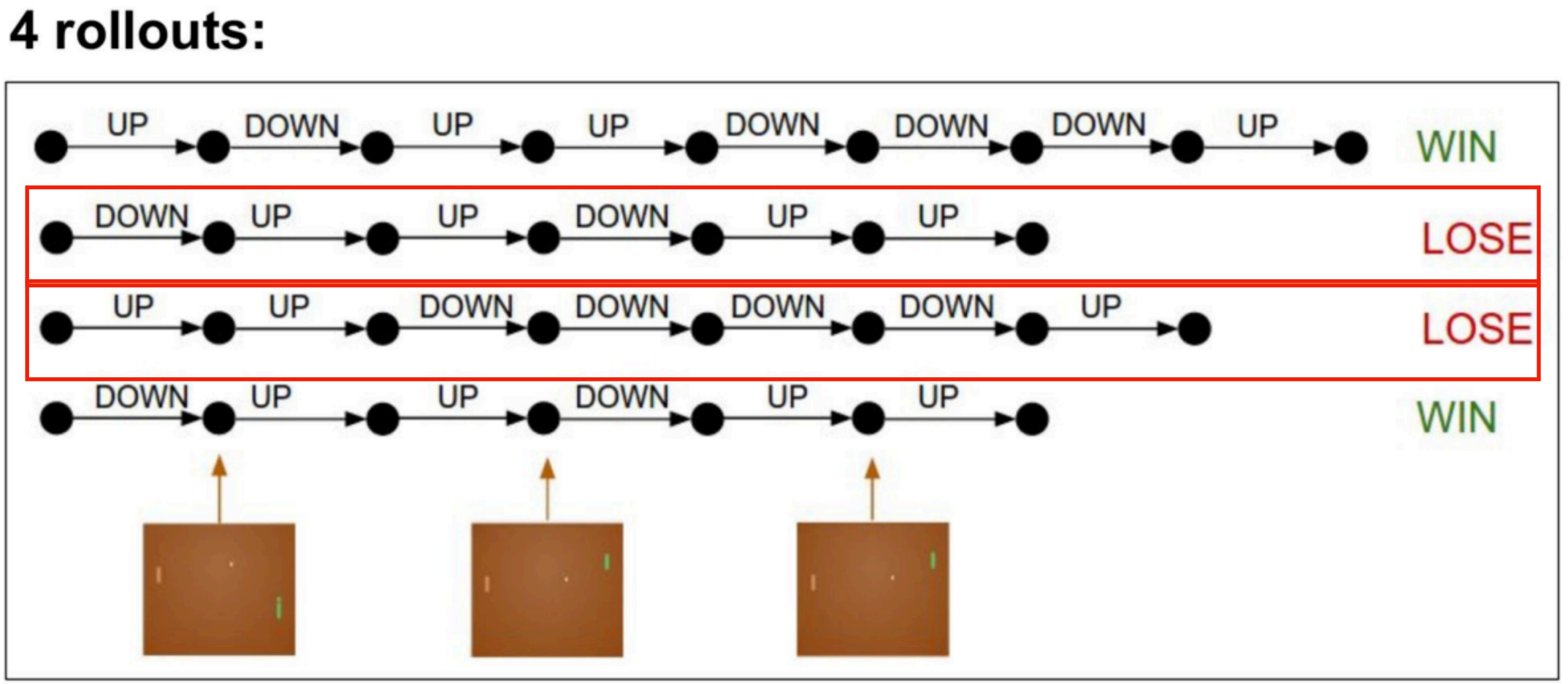
4 rollouts:



Slides from Karpathy

Pong from pixels

Not sure whatever we did here, but it was bad.

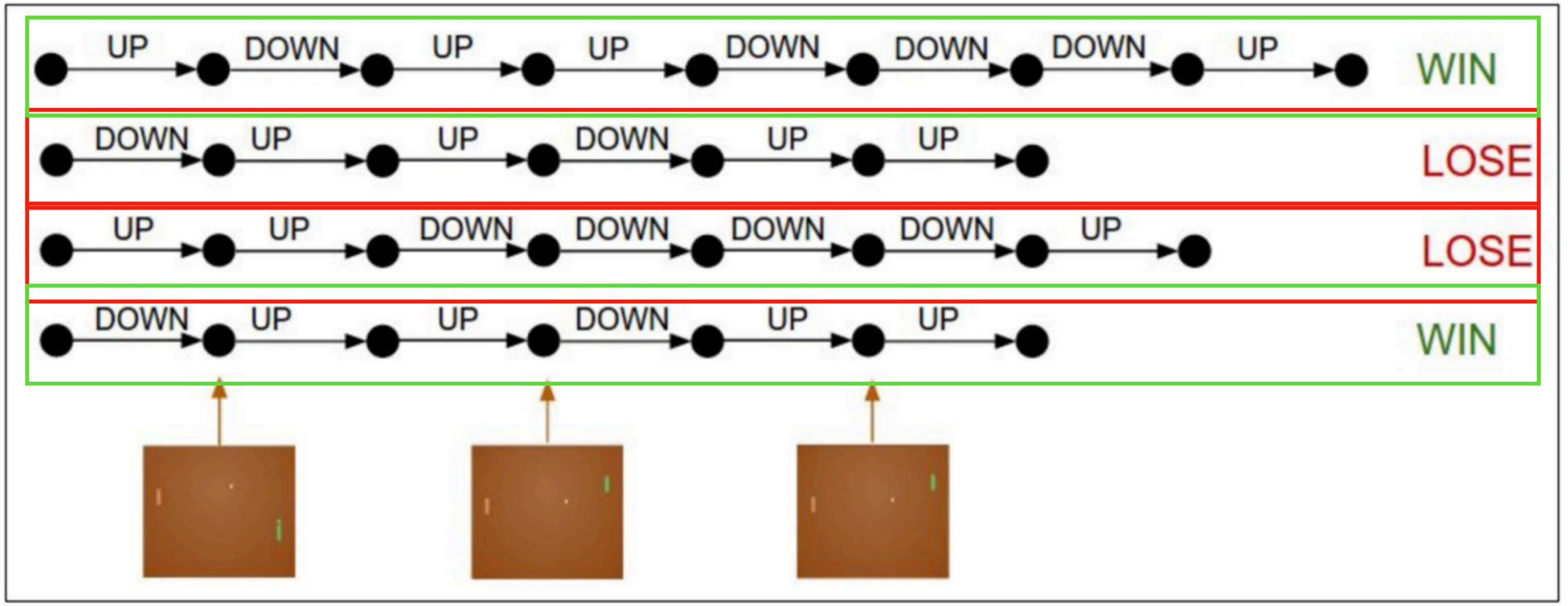


Slides from Karpathy

Pong from pixels

Pretend every action we took here was the correct label or wrong label based on the outcome.

4 rollouts:



Slides from Karpathy

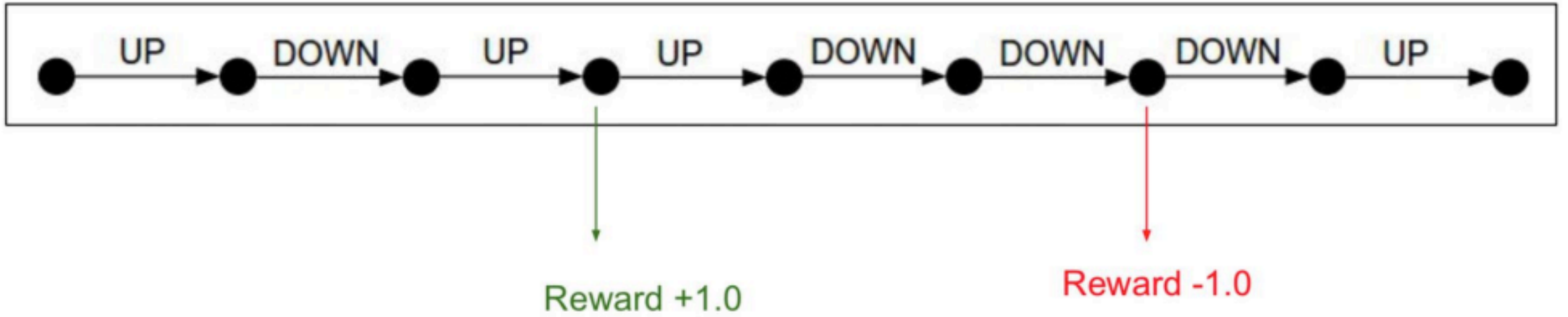
Discounting

Blame each action assuming that its effects have exponentially decaying impact into the future.

$$\sum_i A_i * \log p(y_i|x_i)$$

Discounted rewards

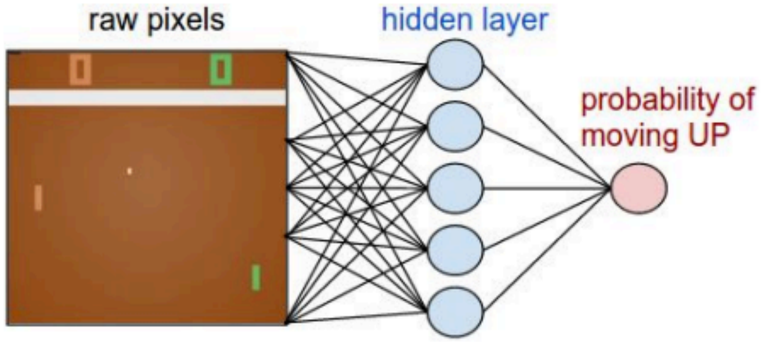
0.21 0.24 0.27 -0.81 -0.9 -1 0 0



\gamma = 0.9

Slides from Karpathy

$\pi(a \mid s)$



1. Initialize a policy network at random
2. **Repeat Forever:**
3. Collect a bunch of rollouts with the policy
4. Increase the probability of actions that worked well

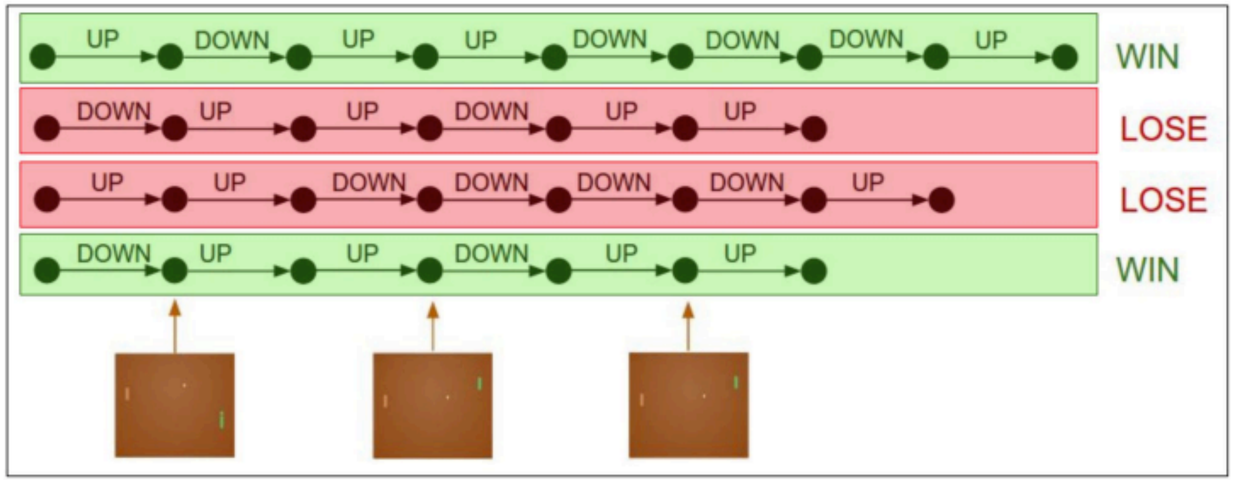
epsilon greedy!

Pretend every action we took here was the correct label.

maximize: $\log p(y_i \mid x_i)$

Pretend every action we took here was the wrong label.

maximize: $(-1) * \log p(y_i \mid x_i)$



$$\sum_i A_i * \log p(y_i \mid x_i)$$

Does not require transition probabilities
Does not estimate Q(), V()
Predicts policy directly

Policy gradient

Slides from Karpathy

Does this algorithm actually work well?

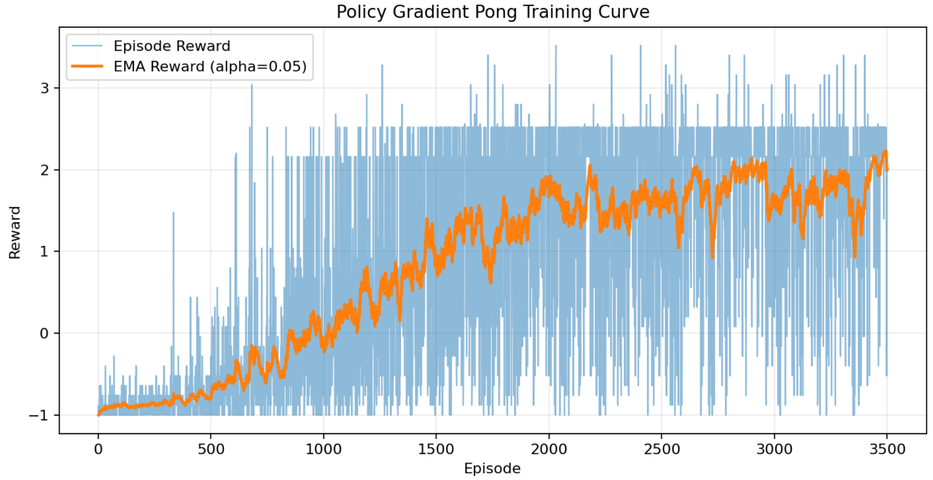
Ep 100

Ep 500

Ep 1500

Ep 2500

Ep 3500



But why does it work?

Policy gradients

Let's define a class of parameterized policies: $\Pi = \{\pi_\theta \mid \theta \in \mathbb{R}^m\}$.

For each policy, define its value:

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t \,\middle|\, \pi_\theta\right].$$

Writing the term of trajectories: $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$

Probability of a trajectory $p(\tau;\theta) = \pi_\theta(a_0 \mid s_0)\, p(s_1 \mid s_0, a_0) \times \pi_\theta(a_1 \mid s_1)\, p(s_2 \mid s_1, a_1) \times \pi_\theta(a_2 \mid s_2)\, p(s_3 \mid s_2, a_2) \times \ldots$

Reward of a trajectory $r(\tau) = \sum_{t\geq 0} \gamma^t r_t$

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t \,\middle|\, \pi_\theta\right] = \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\right].$$

Adapted from Paul liang

But why does it work?

Policy gradients

Let's define a class of parameterized policies: $\Pi = \{\pi_\theta \mid \theta \in \mathbb{R}^m\}$.

For each policy, define its value:

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t \,\middle|\, \pi_\theta\right].$$

Writing the term of trajectories: $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$

Probability of a trajectory $p(\tau;\theta) = \pi_\theta(a_0 \mid s_0)\, p(s_1 \mid s_0, a_0) \times \pi_\theta(a_1 \mid s_1)\, p(s_2 \mid s_1, a_1) \times \pi_\theta(a_2 \mid s_2)\, p(s_3 \mid s_2, a_2) \times \ldots$

Reward of a trajectory $r(\tau) = \sum_{t\geq 0} \gamma^t r_t$

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t \,\middle|\, \pi_\theta\right] = \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\right].$$

Adapted from Paul liang

But why does it work?

Formally, let's define a class of parameterized policies $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$

For each policy, define its value:

$$J(\theta) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | \pi_\theta\right]$$

We want to find the optimal policy $\theta^* = \arg\max_\theta J(\theta)$

How can we do this?

Gradient ascent on policy parameters

Adapted from Paul liang

REINFORCE algorithm

Expected reward: $J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)} [r(\tau)]$

$$= \int_\tau r(\tau) p(\tau;\theta) \, d\tau$$

$$p(\tau;\theta) = \prod_{t \geq 0} p(s_{t+1}|s_t, a_t) \pi_\theta(a_t|s_t)$$

Now let's differentiate this: $\nabla_\theta J(\theta) = \int_\tau r(\tau) \nabla_\theta p(\tau;\theta) \, d\tau$ Intractable! Gradient of an expectation is problematic when p depends on θ

However, we can use a nice trick: $\nabla_\theta p(\tau;\theta) = p(\tau;\theta) \frac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta) \nabla_\theta \log p(\tau;\theta)$

If we inject this back:

$$\nabla_\theta J(\theta) = \int_\tau (r(\tau) \nabla_\theta \log p(\tau;\theta)) \, p(\tau;\theta) \, d\tau$$

$$= \mathbb{E}_{\tau \sim p(\tau;\theta)} [r(\tau) \nabla_\theta \log p(\tau;\theta)]$$

Tractable :-)

Adapted from Paul liang

REINFORCE algorithm

Can we compute these without knowing the transition probabilities?

We have: $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$

Thus: $\log p(\tau;\theta) = \sum_{t\geq 0} (\log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t))$

And when differentiating: $\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$

Doesn't depend on transition probabilities

Therefore when sampling a trajectory, we can estimate gradients:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p(\tau;\theta)} [r(\tau)\nabla_\theta \log p(\tau;\theta)] \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Adapted from Paul liang

Intuition for Policy Gradient

Gradient estimator:

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

Interpretation:

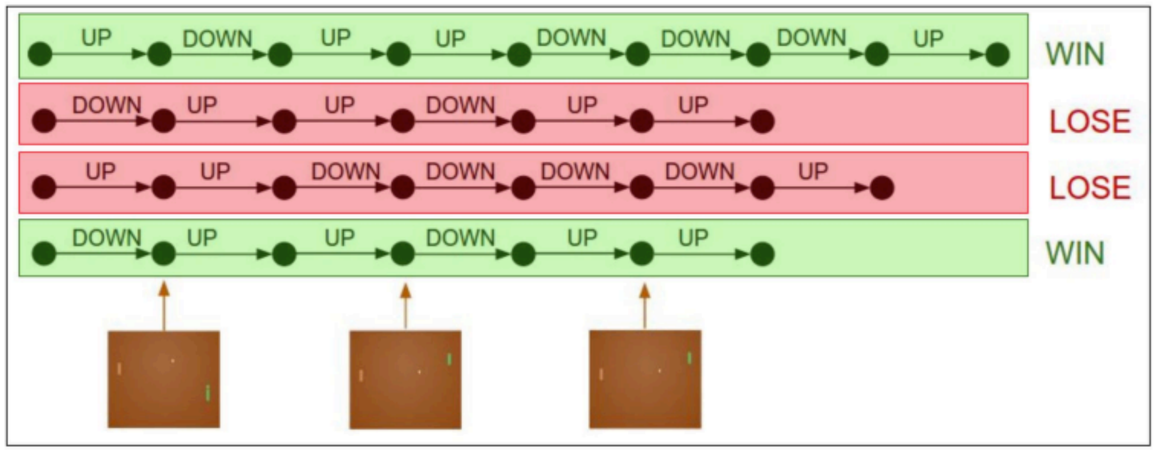
- If **r(trajectory)** is high, push up the probabilities of the actions seen
- If **r(trajectory)** is low, push down the probabilities of the actions seen

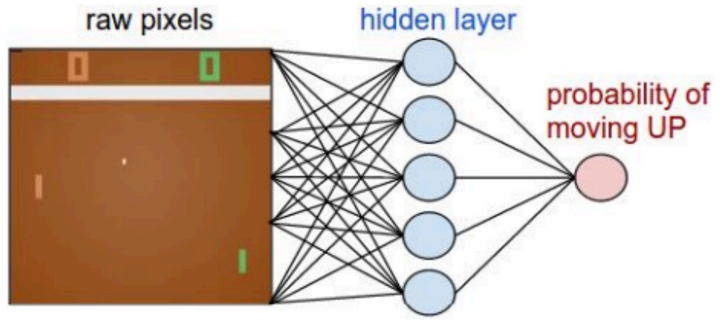
Pretend every action we took here was the correct label.

maximize: $\log p(y_i \mid x_i)$

Pretend every action we took here was the wrong label.

maximize: $(-1) * \log p(y_i \mid x_i)$





$$\sum_i A_i * \log p(y_i | x_i)$$

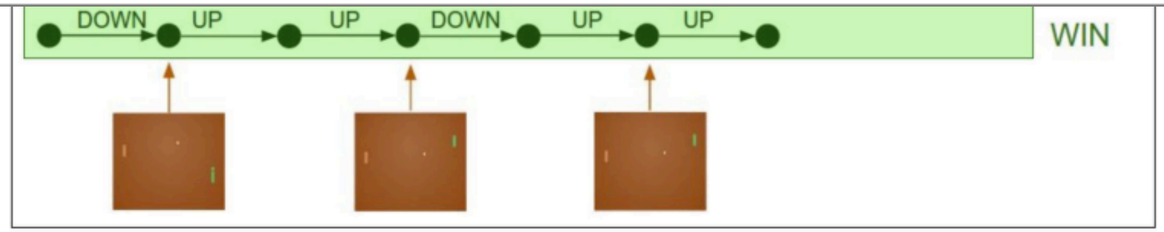
Adapted from Paul liang

Intuition for Policy Gradient

Gradient estimator: $\nabla_\theta J(\theta) \approx \sum r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$

Algorithm 1 REINFORCE Algorithm

1: Initialize a policy π_θ
2: **while** NOT stopping criterion **do**
3: Collect N trajectories $\{\tau^i\}$ from the environment using π_θ
4: Calculate $\nabla_\theta \mathcal{J}(\pi_\theta)$ using Equation (16)
5: Update $\theta = \theta + \alpha \nabla_\theta \mathcal{J}(\pi_\theta)$
6: **end while**



$$\sum_i A_i * \log p(y_i|x_i)$$

Adapted from Paul liang

Policy Gradient Method- Proximal Policy Optimization

Proximal Policy Optimization Algorithms

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov
OpenAI
{joschu, filip, prafulla, alec, oleg}@openai.com

Abstract

We propose a new family of policy gradient methods for reinforcement learning, which alternate between sampling data through interaction with the environment, and optimizing a "surrogate" objective function using stochastic gradient ascent. Whereas standard policy gradient methods perform one gradient update per data sample, we propose a novel objective function that enables multiple epochs of minibatch updates. The new methods, which we call proximal policy optimization (PPO), have some of the benefits of trust region policy optimization (TRPO), but they are much simpler to implement, more general, and have better sample complexity (empirically). Our experiments test PPO on a collection of benchmark tasks, including simulated robotic locomotion and Atari game playing, and we show that PPO outperforms other online policy gradient methods, and overall strikes a favorable balance between sample complexity, simplicity, and wall-time.

Policy Gradient updates:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a|s) A(s,a)$$

Issue:

- Large update → policy changes too much
- Training becomes unstable
- Performance can collapse

Key idea of PPO:
From "Increase probability of good actions." To "increase probability of good action but don't move too far from the old policy."

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

Clipping prevents large destructive updates

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right]$$