Reinforcement Learning Introduction

Motivation: we want to control a helicopter to fly upside down. There are 4 options.

- Find an expert to control a helicopter with a manual control stick.
- Use conventional control algorithms. Define dynamic model, measurement model.
  But it's difficult to tell machines what/how to do. E.g., when to speed up/slow down main rotor.
- Use conventional supervised learning. Map $X \longrightarrow Y$ (X: system states; Y: control signal) It's unrealistic. There're too many possible states.
- Use Reinforcement Learning.

  Reward function: incentivize the machine to figure out controling algorithm by itself.

  Positive reward: helicopter flying well +1

  Negative reward: helicopter flying poorly -1000

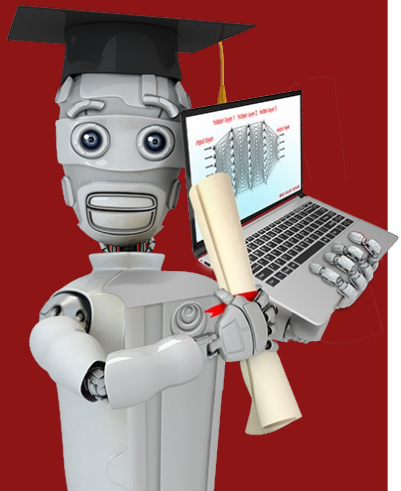
  It's like training a dog. I need to tell it what the goal is and reward my dog's action rather than how to do it. And dogs will figure it out themselves.

Application

- Controlling robots, drones.
- Factory Optimization
- Financial (stock) trading → It's an interesting application. I always think linear regression is used in algorithms to predict stock market. Explain why this is the case.
- AlphaGo

DeepLearning.AI

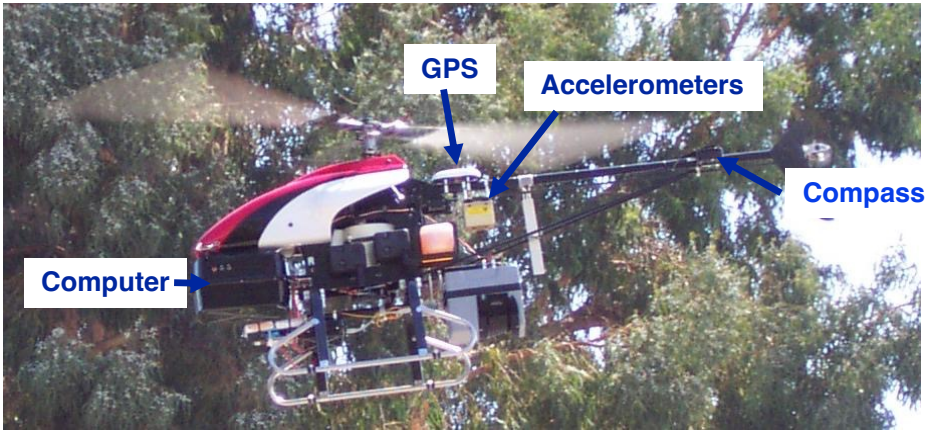
Stanford ONLINE



Reinforcement Learning Introduction

What is Reinforcement Learning?

Autonomous Helicopter



How to fly it?

Autonomous Helicopter

[Thanks to Pieter Abbeel, Adam Coates and Morgan Quigley]

→ For more videos: http://heli.stanford.edu.

Robotic Dog Example

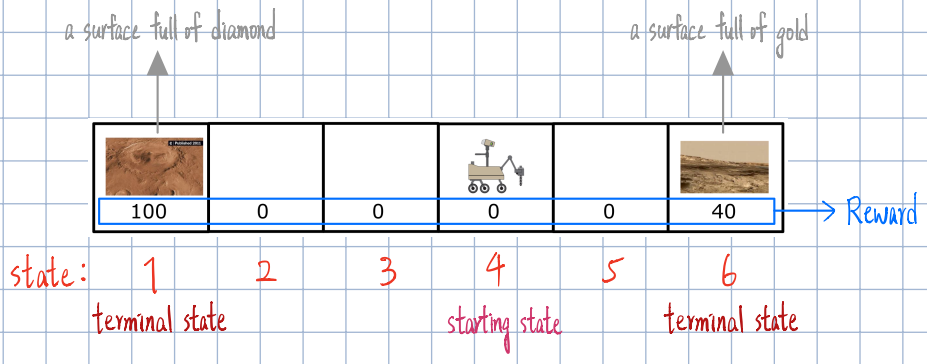
[Thanks to Zico Kolter]

Mars Rover Example

credit: Jagriti Agrawal, Emma Brunskill

Given a Mars rover, it can be in any of the 6 states in any time step.

This rover can only go either left ← or right →.



Now, the robot has many options to go from the starting state toward the terminal state. After reaching the terminal state, the rover cannot go any further.

- Option 1:

  state: 4 3 2 1
  action: ← ← ←
  Reward: 0 0 0 100

- Option 2:

  state: 4 5 6
  action: → →
  Reward: 0 0 40

- Option 3:

  state: 4 5 4 3 2 1
  action: → ← ← ← ←
  Reward: 0 0 0 0 0 100

At every time step, the rover (state, action, Reward, new state)
(s, a, R(s), s')

For instance, in Option 1., at the first step, it's (4, ←, 0, 3) (R(4) = 0)

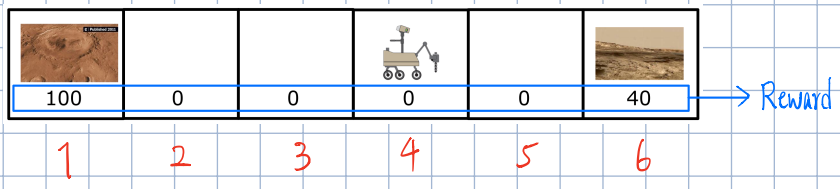
Define Return in reinforcement learning.

First, define discount factor γ. In practice, γ is around 0.9, 0.99, or 0.999.

Then, Return $= R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots$ (until terminal state)

$R_1, R_2, R_3, \ldots, R_i$: Reward at time step i.

In this example, $\gamma = 0.5$ is used for demonstration purposes.



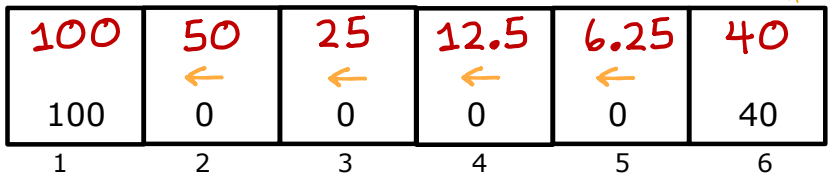
Thus, for Option 1, • Option 1:

state:	4	3	2	1
action:	←	←	←	
Reward:	0	0	0	100

Return $= 0 + 0.5 \cdot 0 + 0.5^2 \cdot 0 + 0.5^3 \cdot 100 = 12.5$

Example of Return

Always go left



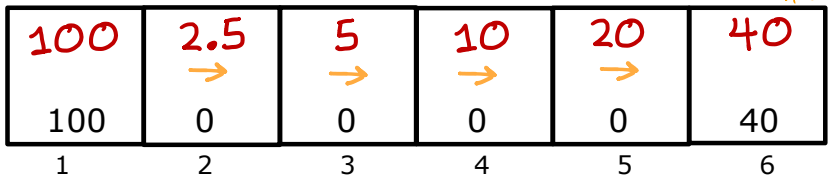
← return

← reward

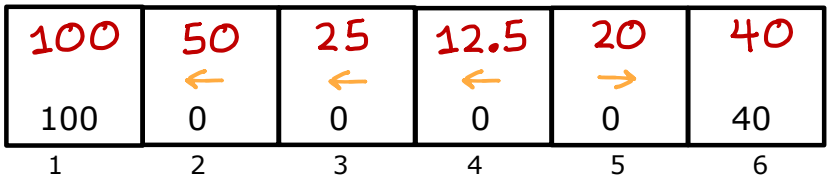
$\gamma = 0.5$

The return depends on the actions you take.

Always go right

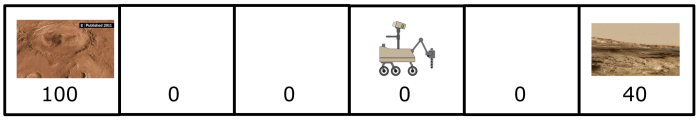


$0 + (0.5)0 + (0.5)^2 40 = 10$



$0 + (0.5)40 = 20$

Making decisions: Policies in reinforcement learning



Given a system, how can we maximize the return?

What policies should be used to determine the action?

For example, the rover can always go left or always go right or always go for the nearer reward.

Policy π

state s → action a

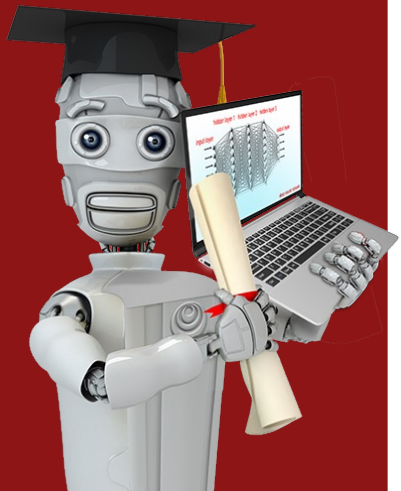
A policy is a function $\pi(s) = a$ mapping from states to actions, that tells you what action a to take in a given state s.

The goal of reinforcement learning:

Find a policy π that tells you what action ($a = \pi(s)$) to take in every state (s) so as to maximize the return.

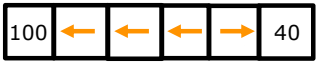
DeepLearning.AI

Stanford ONLINE



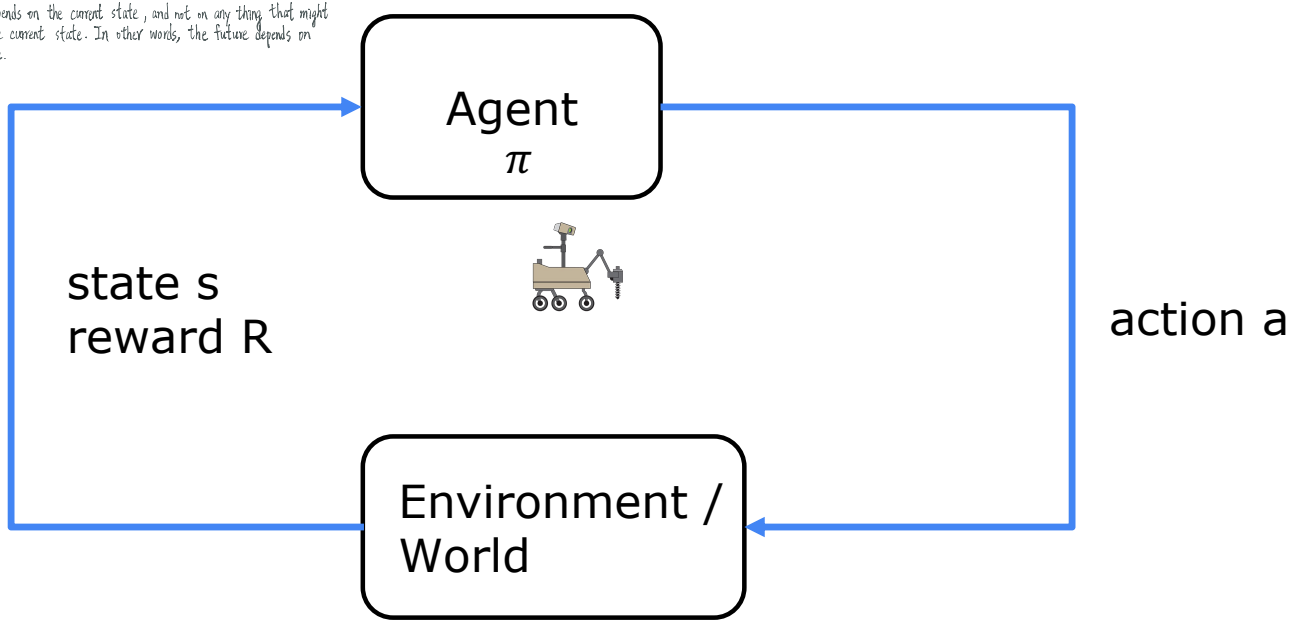
Reinforcement Learning formalism

Review of key concepts

	Mars rover 	Helicopter 	Chess 
states	6 states	position of helicopter	pieces on board
actions	← →	how to move control stick	possible move
rewards	100, 0, 40	flying well +1, −1000 crashing	+1, 0, −1
discount factor γ	0.5	0.99	0.995
return	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$	$R_1 + \gamma R_2 + \gamma^2 R_3 + \cdots$
policy π		Find $\pi(s) = a$	Find $\pi(s) = a$

Markov Decision Process (MDP)

Markov refers to that the future only depends on the current state, and not on any thing that might have occurred prior to getting to the current state. In other words, the future depends on where you are now, not on how you got here.



State-Action Value Function

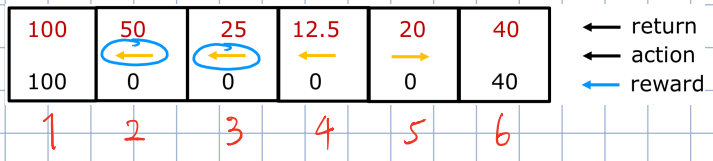
(Q-function)

state-action value function $Q(\underset{\text{state}}{s}, \underset{\text{action}}{a})$ = Return if you

- start in state s.
- take action a (once).
- then behave optimally after that.

Looking at the Mars rover example below. Actually, the policy below is an optimal policy.

discount factor $\gamma = 0.5$



now, we can calculate Q for various states and actions.

For instance,

$Q(2, \rightarrow) = 0 + 0.5 \cdot 0 + 0.5^2 \cdot 0 + 0.5^3 \cdot 100 = 12.5$

Notice that it passes no judgement on if going right is a good idea or not. It's actually not a good idea to go right at state 2, but $Q(2, \rightarrow)$ faithfully reports out the return value.

$Q(2, \leftarrow) = 0 + 0.5 \cdot 100 = 50$

$Q(4, \leftarrow) = 0 + 0.5 \cdot 0 + 0.5^2 \cdot 0 + 0.5^3 \cdot 100 = 12.5$

After calculating all combinations of states and actions, the result is

state	1		2		3		4		5		6	
action	←	→	←	→	←	→	←	→	←	→	←	→
Q-value	100	100	50	12.5	25	6.25	12.5	10	6.25	20	40	40

now, by comparing Q-value and the return value derived from the best policy, we can observe that the best possible return value from any state s is the largest value of $Q(s, a)$ maximizing over action a.

$$\max_a Q(s, a)$$

Therefore, the best possible action in state s is the action a that gives $\max_a Q(s, a)$.

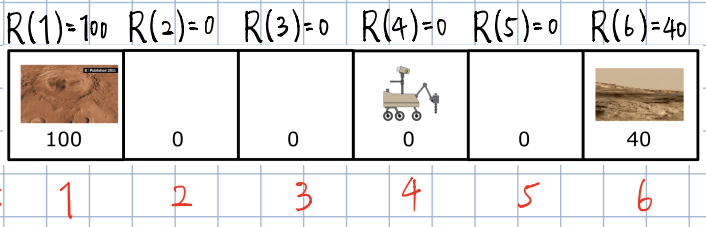
But, how do I know the best policy? Without best policy, I cannot know how to behave optimally.

Without knowing how to behave optimally, I cannot calculate $Q(s, a)$. The solution is using Bellman Equation.

Bellman Equation:

$Q(s, a)$ = Return value if you

- start in state s.
- take action a (once).
- then behave optimally after that.



Definition:

s: current state

$R(s)$: reward of current state

a: current action

s': state you get to after taking action a.

a': action that you take in state s'.

discount factor γ (Gamma)

$$Q(s, a) = R(s) + \gamma \cdot \max_{a'} Q(s', a')$$

Reward you get right away

Return from behaving optimally starting from state s'.

Exeption: if in the terminal state

$$Q(s, a) = R(s)$$

For instance, using the rover example:

state	1		2		3		4		5		6	
action	←	→	←	→	←	→	←	→	←	→	←	→
Q-value	100	100	50	12.5	25	6.25	12.5	10	6.25	20	40	40

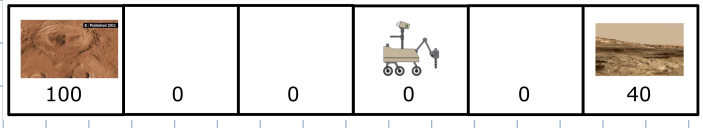
$$Q(2, \rightarrow) = R(2) + 0.5 \cdot \max_{a'} Q(3, a') = 0 + 0.5 \cdot 25 = 12.5$$

$$Q(4, \leftarrow) = R(4) + 0.5 \cdot \max_{a'} Q(3, a') = 0 + 0.5 \cdot 25 = 12.5$$

Random (Stochastic) Environment

In real-world environment, the outcome is not always reliable. For example, if I command the rover to go left, it's possible that the rover is actually going right due to blowing wind.

For instance, let's say there's 10% possibility that the rover won't go to the direction as commanded.



If the command action is ← at state 4, the rover will be at state 3 90% of the time.

state 5 10% of the time.

Thus, the actual return value must take all possibilities into account and then average them.

$$\text{Expected Return} = \text{Average}\left(R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots\right)$$

$$= E\left[R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots\right]$$

Recall the goal of reinforcement learning:

Choose a policy $\pi(s) = a$ that tells us what action ($a = \pi(s)$) to take in state (s) so as to maximize the expected return.

Bellman Equation becomes

$$Q(s, a) = R(s) + \gamma \cdot E\left[\max_{a'} Q(s', a')\right]$$

Learning State-Action Value Function

Finally, we are ready to train the algorithm.

Discrete v.s. Continuous: In the naive Mars rover example, states are all discrete. In reality, states are likely continuous. E.g., the position encompasses latitude and longitude or the orientation is composed of angle.

In the following toy example, a simple Lunar Lander is shown.

The goal is to land the Lunar Lander perfectly.

Lunar Lander Problem

Learn a policy π that, given

Actions:

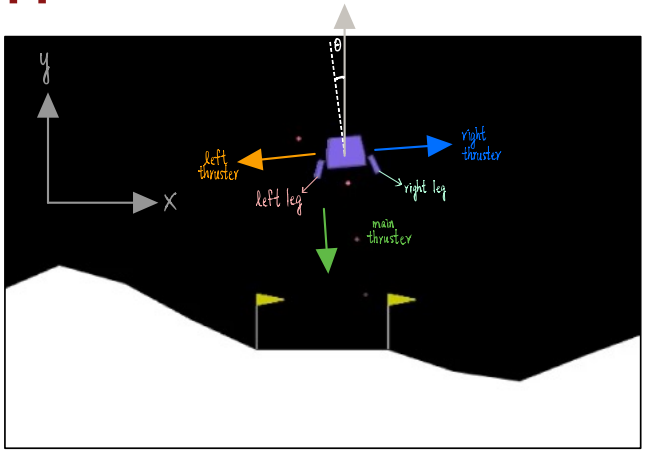
- Do nothing
- Left thruster
- Main thruster
- Right thruster

$$s = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \end{bmatrix}$$

θ → tilting angle

l → if left leg is on the ground

r → if right leg is on the ground



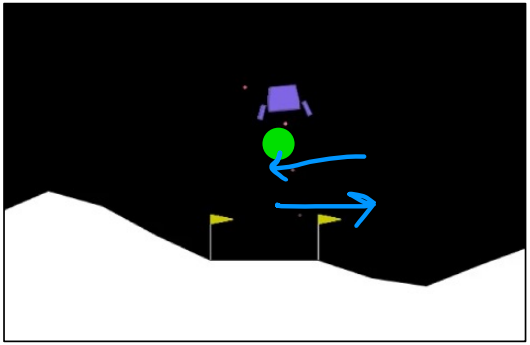
picks action $a = \pi(s)$ so as to maximize the return.

$$\gamma = 0.985$$

Reward Function

- Getting to landing pad: 100 – 140
- Additional reward for moving toward/away from pad.
- Crash: -100
- Soft landing: +100
- Leg grounded: +10
- Fire main engine: -0.3
- Fire side thruster: -0.03

Encourage to use less fuel.

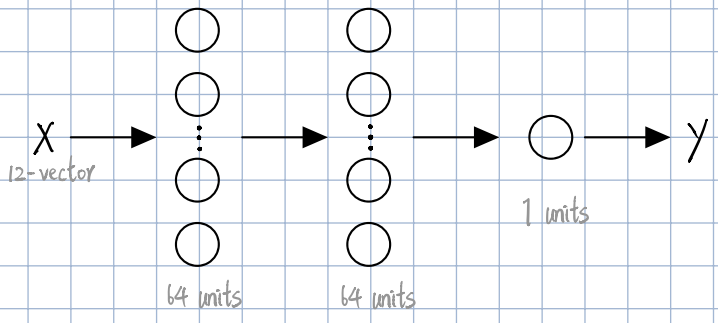


Now, we are ready to use deep reinforcement learning to learn the state-action value function Q.

Input $X = \begin{bmatrix} \text{state} \\ \text{action} \end{bmatrix} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, Output $y = Q(s, a)$

Do nothing ←

→ Must be one-hot vector (cannot fire left and right thruster simultaneously).



In inference, in state s, we can use neural network to compute

$Q(s,$ Do nothing$)$ $Q(s,$ left thruster$)$ $Q(s,$ main thruster$)$ $Q(s,$ right thruster$)$,

and then pick action a that maximizes $Q(s, a)$

Now, the question is how to create training set (X, y) ? Use Bellman Equation.

Recall Bellman Equation.

$$\underbrace{Q(\overbrace{s, a}^{X})}_{} = \underbrace{R(s) + \gamma \cdot \max_{a'} Q(s', a')}_{y}$$

The job of the neural network is to find a function $f_{w,b}$ such that $f_{w,b}(X) \approx y$.

The basic idea of creating training set (X, y) is by taking random action a in state s we can observe rewards for being in that state s, and as result of action a, new state s'.

$$(s, a, R(s), s')$$

Here, $X = \begin{bmatrix} s \\ a \end{bmatrix}$

$$y = R(s) + \gamma \cdot \max_{a'} Q(s', a')$$

The tricky part is that initially we don't know the Q function, and it's OK to take a random guess of Q.

e.g., after doing it 10^4 times, we get 10^4 examples (X, y).

$$(s^{(1)}, a^{(1)}, R(s^{(1)}), s'^{(1)}) \longrightarrow (X^{(1)}, y^{(1)})$$

$$(s^{(2)}, a^{(2)}, R(s^{(2)}), s'^{(2)}) \longrightarrow (X^{(2)}, y^{(2)})$$

$$\vdots$$

$$(s^{(10^4)}, a^{(10^4)}, R(s^{(10^4)}), s'^{(10^4)}) \longrightarrow (X^{(10^4)}, y^{(10^4)})$$

Learning Algorithm for Learning the Q-function (Deep Q-Network DQN)

Step 1:

Initialize neural network randomly as guess of $Q(s, a)$. (just like initialize parameters randomly in linear regression)

Step 2:

Repeat {

- Take actions in Lunar Lander. Get $(s, a, R(s), s')$.
- Store 10^4 most recent $(s, a, R(s), s')$ tuples. ← Replay buffer

- Train neural network:

Create training set of 10^4 examples $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(10^4)}, y^{(10^4)})$ using:

$$x = \begin{bmatrix} s \\ a \end{bmatrix}$$

$$y = R(s) + \gamma \cdot \max_{a'} Q(s', a')$$

Train Q_{new} such that $Q_{new}(s, a) \approx y$. (just like $f_{w,b}(x) \approx y$.)

- Set $Q = Q_{new}$

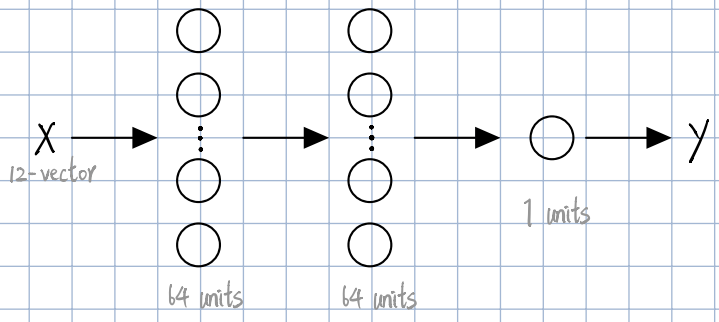
}

Read "Human-level control through deep reinforcement learning"

Algorithm Refinement: Improved neural network architecture

Recall

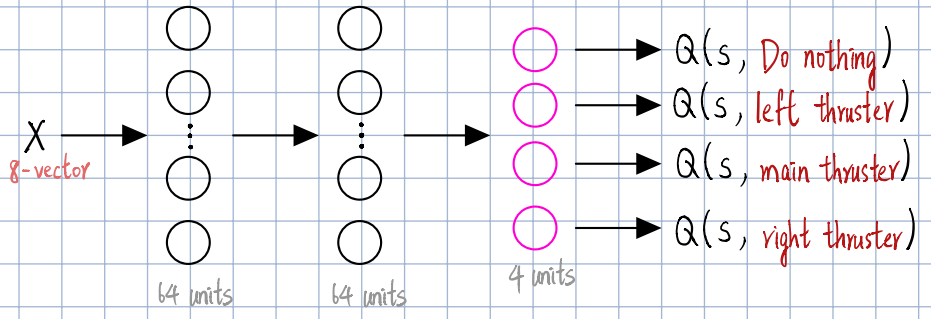
$$\text{Input } X = \begin{bmatrix} \text{state} \\ \text{action} \end{bmatrix} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \text{Output } y = Q(s, a)$$



The problem of this architecture is that it has to perform inference 4 times for each action in every state s.

$Q(s, \text{Do nothing})$
$Q(s, \text{left thruster})$
$Q(s, \text{main thruster})$
$Q(s, \text{right thruster})$,

A better way would be output 4 values simultaneously as shown below:



$$\text{Input } X = \begin{bmatrix} \text{state} \\ \text{action} \end{bmatrix} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \end{bmatrix}$$

$$\text{Output } y = Q(s, a)$$

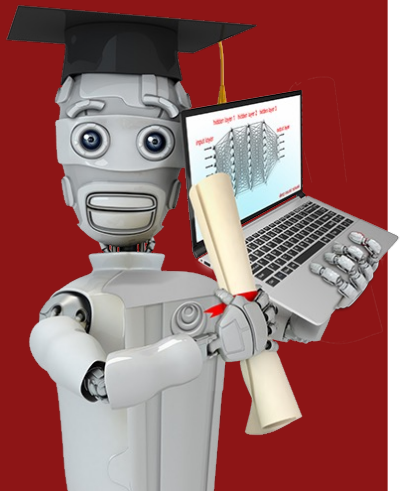
In state s, input X to neural network.

Pick the action a that maximizes $Q(s, a)$.

It turns out it's more efficient because it can get 4 values of $Q(s, a)$ just by running inference once.

DeepLearning.AI

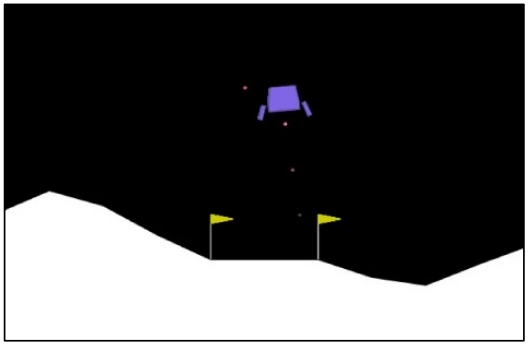
Stanford ONLINE



Continuous State Spaces

Algorithm refinement: ε-greedy policy

Learning Algorithm



Initialize neural network randomly as guess of $Q(s, a)$.

Repeat { How to take actions in this step while learning?

Take actions in the lunar lander. Get $(s, a, R(s), s')$.

Store 10,000 most recent $(s, a, R(s), s')$ tuples.

Train model:

Create training set of 10,000 examples using

$x = (s, a)$ and $y = R(s) + \gamma \max_{a'} Q(s', a')$.

Train Q_{new} such that $Q_{new}(s, a) \approx y$. $f_{w,b}(x) \approx y$

Set $Q = Q_{new}$.

How to choose actions while still learning?

In some state s

what if init $Q(s, \text{main})$ to be low?

Option 1: → will never try other actions ↳ will never fire main thrusters!

Pick the action a that maximizes $Q(s, a)$.

Option 2:

With probability 0.95, pick the action a that maximizes $Q(s, a)$. greedy exploitation

With probability 0.05, pick an action a randomly. exploration

ε-greedy policy ($\varepsilon = 0.05$)

greedy 95% of the time

exploring 5% of the time

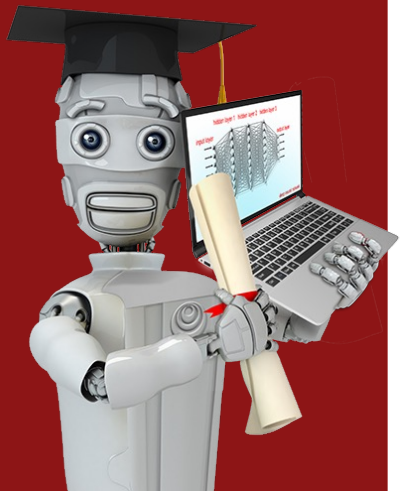
Start ε high

gradually decrease

1.0 → 0.01

DeepLearning.AI

Stanford ONLINE



Continuous State Spaces

Algorithm refinement: Mini-batch and soft update (optional)

How to choose actions while still learning?

x	y
2104	400
1416	232
1534	315
852	178
...	...
3210	870

100,000,000

$$J(w,b) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_{w,b}\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$m = 100,000,000$

$m' = 1,000$

repeat {

$$w = w - \alpha \frac{\partial}{\partial w} \frac{1}{2m'}\sum_{i=1}^{m'}\left(f_{w,b}\left(x^{(i)}\right) - y^{(i)}\right)^2$$

$$b = b - \alpha \frac{\partial}{\partial b} \frac{1}{2m'}\sum_{i=1}^{m'}\left(f_{w,b}\left(x^{(i)}\right) - y^{(i)}\right)^2$$

}

Using the whole batch $m=10^8$ in every step in gradient descent is too slow. Thus, using only a portion $m'=10^3$ in every step is faster. Remember to shuffle the dataset.

Mini-batch

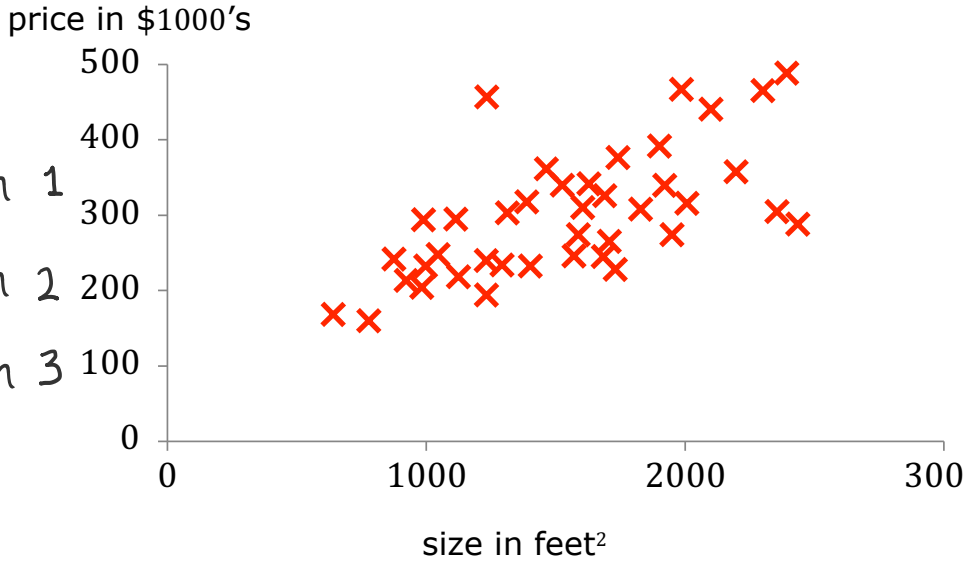
x	y
2104	400
1416	232
1534	315
852	178
...	...
3210	870

batch

iteration 1

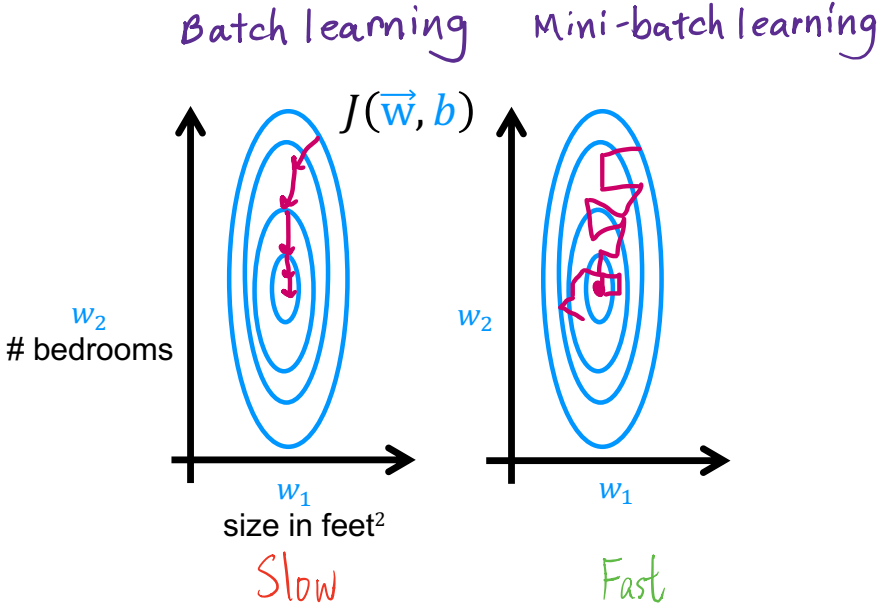
iteration 2

iteration 3

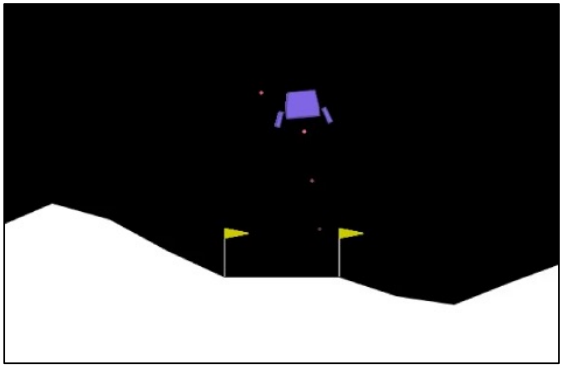


Mini-batch

x	y
2104	400
1416	232
1534	315
852	178
...	...
3210	870



Learning Algorithm



Initialize neural network randomly as guess of $Q(s,a)$.

Repeat {

Take actions in the lunar lander. Get $(s, a, R(s), s')$.

Store 10,000 most recent $(s, a, R(s), s')$ tuples.

Replay Buffer

Only use 10^3 examples

1,000

Train model:

Create training set of ~~10,000~~ examples using

$x = (s, a)$ and $y = R(s) + \gamma \max_{a'} Q(s', a')$

Train Q_{new} such that $Q_{new}(s, a) \approx y$.

Set $Q = Q_{new}$.

This can make every abrupt change to Q.
If unlucky, Q becomes very bad choice. Thus, Q must change gradually.

$x^{(1)}, y^{(1)}$

$x^{(1000)}, y^{(1000)}$

Soft Update

Set $Q = Q_{new}$.

w, b W_{new}, B_{new}

$Q(s,a)$

$W = 0.01 W_{new} + 0.99 W$

$B = 0.01 B_{new} + 0.99 B$

$W = 1 W_{new} + 0 W$

parameters changes too fast

Q now changes gradually and slowly.