Week 3

Reinforcement Learning

Contents: Learn RL, build q-learning, land lander on mars.

There are multiple steps to a certain solution. How do we find it? How do we chech if a certain move in chess will affect your future situation?

It’s a pillar of mach.lear., even though not widely used commercially as recommender systems.

Example: Consider an autonomous helicopter flying. Andrew: “I actually did it but it’s hard. How would you automize flying a helicopter?”

“We used reinforcement learning to make it fly upside down (WOW). ”

Idea: Given position of helicopter, decide how to move. **Basically given state s, decide action a**.

*You could theoretically do this with supervised learning, labeling actions in states by an expert. But when helicopter is flying, there is actually very ambiguous what 1 right action is. It’s also very difficult to get a dataset for this.

For RL, a reward function is used to tell the computer if it’s action is good or bad. It’s just like a dog, if you give it food, it thinks what it’s doing is god. **You tell what to do instead how to do it.** Gives flexibility to system.

A robotic dog, which tries to get over some obstacles on it’s road, works with this same principle. When you look at te dog, you may have no idea how it should position it’s legs in a scenario. But the dog figures them out automatically.

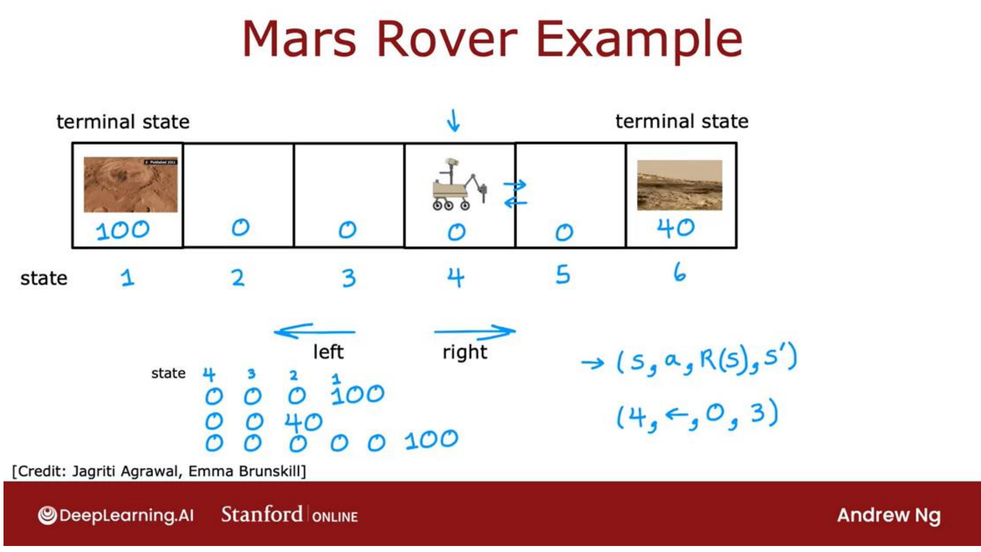
Applications: Controlling robots, factory optimization, stock trading, playing games (including video games).

Mars Rover Example (simplified)

“Actually, people in Stanford has written RL that controls mars rovers.”

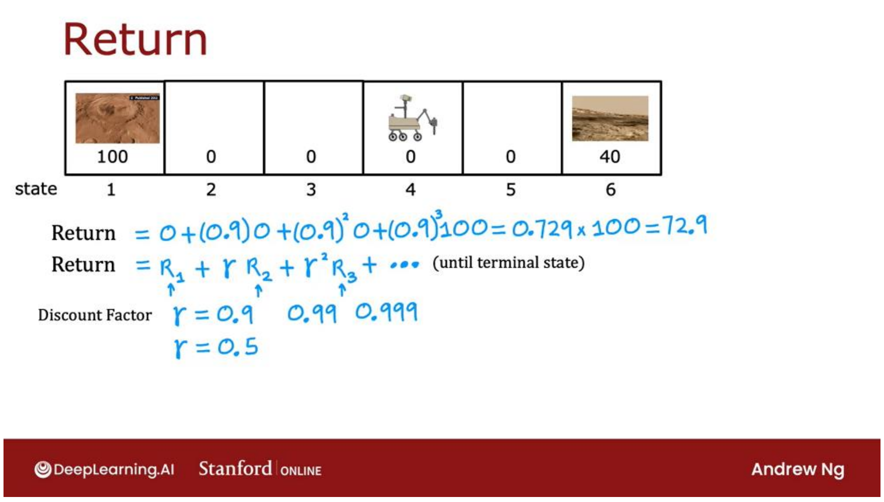
To start with, we have different states that rover can be in from 1 to 6.

Terminal state: After it gets to this state, stop. (that day ends)



The Return in reinforcement learning

If you could reach and pick a 5 dollar bill, or walk a minute and pick a 10 dollar from ground, which one would you choose? **Idea:** You need to specify what is a good act. Quicker gains are preferred.



Return

state	1	2	3	4	5	6
reward	100	0	0	0	0	40

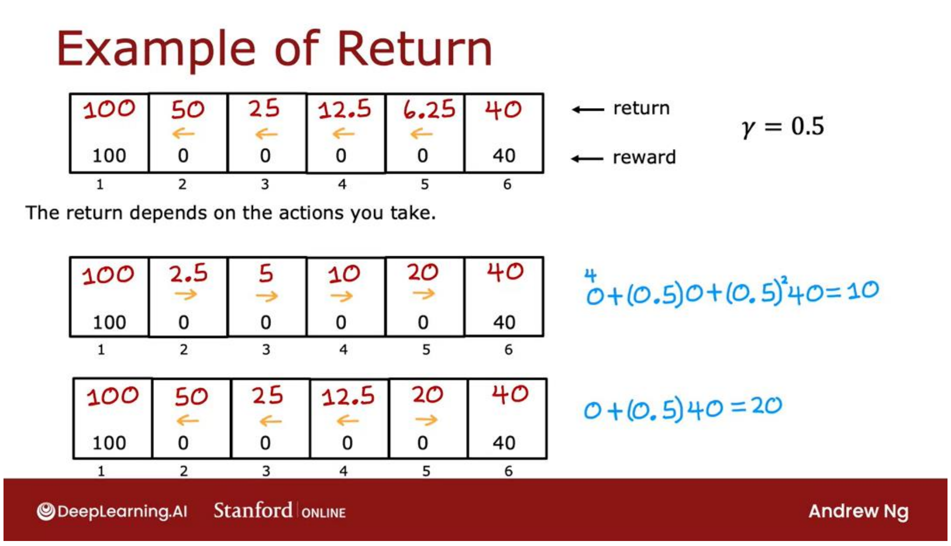
Return $= 0+(0.9)0+(0.9)^2 0+(0.9)^3 100 = 0.729 \times 100 = 72.9$

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

Discount Factor $\gamma = 0.9 \quad 0.99 \quad 0.999$

$\gamma = 0.5$

Discount ratio makes the algorithm a bit impatient. It prioritizes quicker gains and gives later gains a multiplier smaller than 1. In finance, you can think of it as: “Having a dollar today is worth more than having a dollar next year. Both time-wise and interest-wise.”



Example of Return

$\gamma = 0.5$

	1	2	3	4	5	6
return	100	50 ←	25 ←	12.5 ←	6.25 ←	40
reward	100	0	0	0	0	40

The return depends on the actions you take.

	1	2	3	4	5	6
return	100	2.5 →	5 →	10 →	20 →	40
reward	100	0	0	0	0	40

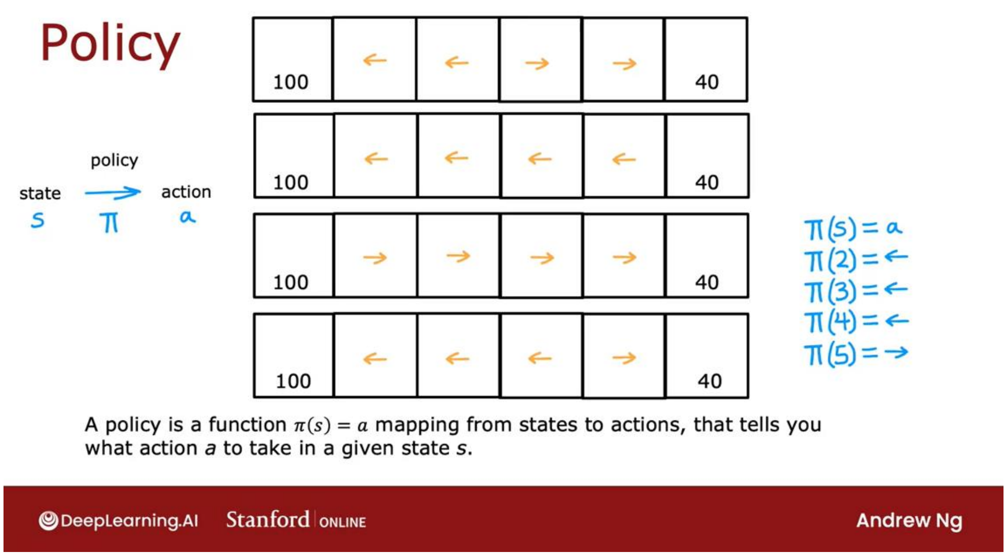
$\overset{4}{0}+(0.5)0+(0.5)^2 40 = 10$

	1	2	3	4	5	6
return	100	50 ←	25 ←	12.5 ←	20 →	40
reward	100	0	0	0	0	40

$0+(0.5)40 = 20$

Return is sum of the rewards in that chain of actions and is weighted by fiscount factor.

This becomes interesting when negative rewars are added to the system. Then the algorithm tries to postpone negatives.



Policy

$$\begin{tikzcd} \text{state } s \arrow[r, "\text{policy } \pi"] & \text{action } a \end{tikzcd}$$

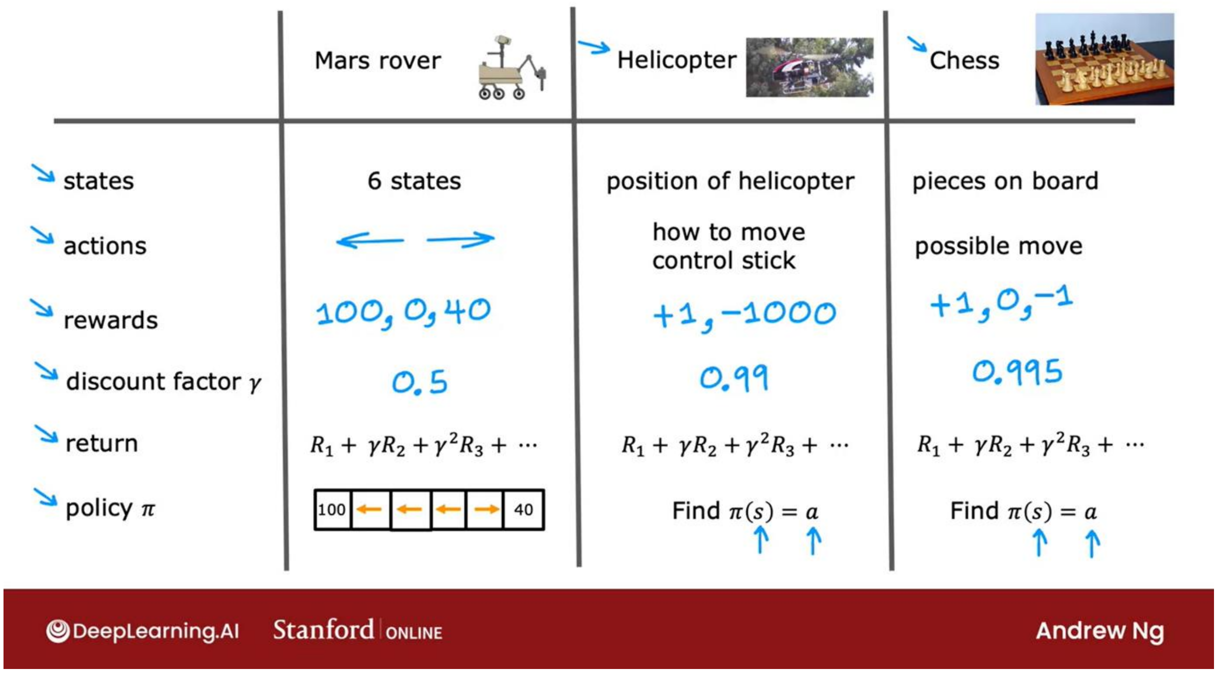
100	←	←	→	→	40
100	←	←	←	←	40
100	→	→	→	→	40
100	←	←	←	→	40

$\pi(s) = a$
$\pi(2) = \leftarrow$
$\pi(3) = \leftarrow$
$\pi(4) = \leftarrow$
$\pi(5) = \rightarrow$

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.

Policy: It is a function that takes s (state) as input and decides what a (action) should be.

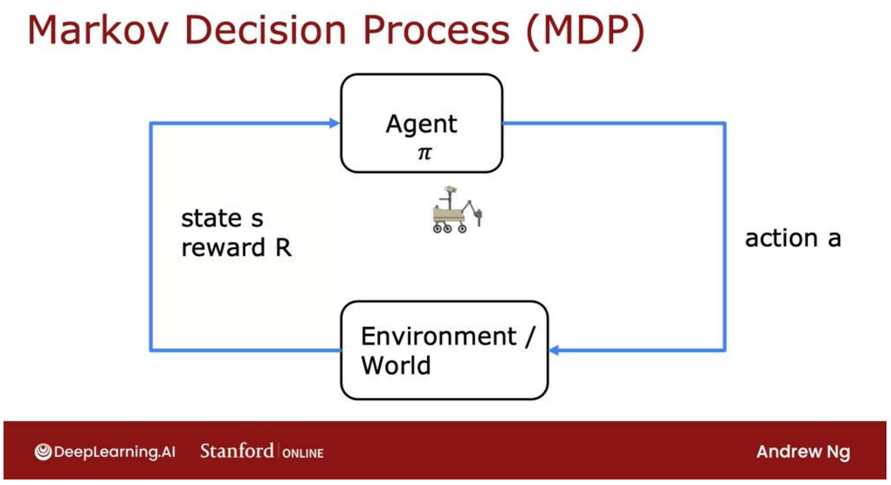
GOAL OF RL: Find a policy that will tell you what action to take in every state **in order to maximize the return**. (be catious, not reward but return)



	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	+1, −1000	+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	100 ← ← ← → 40	Find $\pi(s) = a$	Find $\pi(s) = a$

DeepLearning.AI Stanford ONLINE Andrew Ng

image on different examples



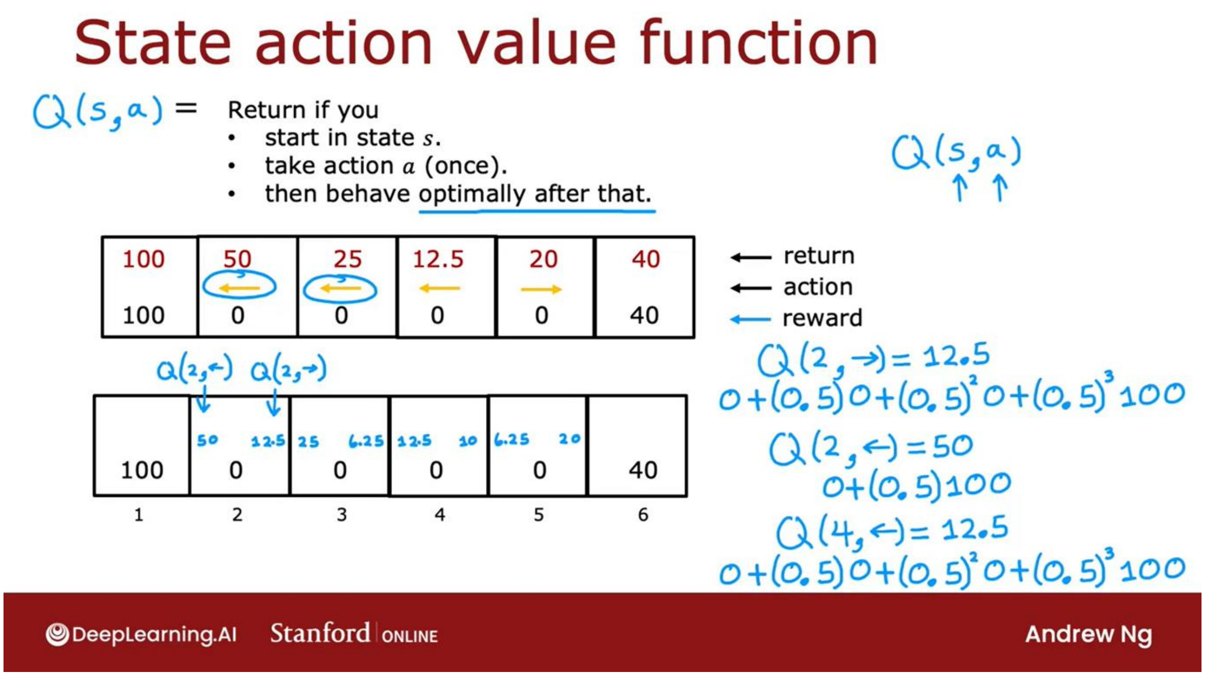
Markov Decision Process (MDP): Formalizm of these steps.

Markov refers to that: future only depends on the current state and not on steps taken before getting to that state.

Agent takes action depending on the policy, world changes,
depending on state (which all have rewards) agent keeps taking steps.

****A key to developing an algorithm is to define a "STATE-ACTION VALUE FUNCTION".** It's a key function that algorithm will try to compute (Just like it applied to cost function in previous parts).

It's represented as **Q(s, a)**. In a state of s, this function's value measures how good each step is in that step. Below it is shown and **we expect agent to move reasonably after this move. (optimal way actually).** It's also called Q-FUNCTION.



State action value function

$Q(s,a) =$ Return if you
- start in state s.
- take action a (once).
- then behave optimally after that.

100	50	25	12.5	20	40	← return
	←	←	←	→		← action
100	0	0	0	0	40	← reward

$Q(2,\leftarrow)$ $Q(2,\rightarrow)$

1	2	3	4	5	6
	50 12.5	25 6.25	12.5 10	6.25 20	
100	0	0	0	0	40

$Q(s,a)$

$Q(2,\rightarrow)=12.5$

$0+(0.5)0+(0.5)^2 0+(0.5)^3 100$

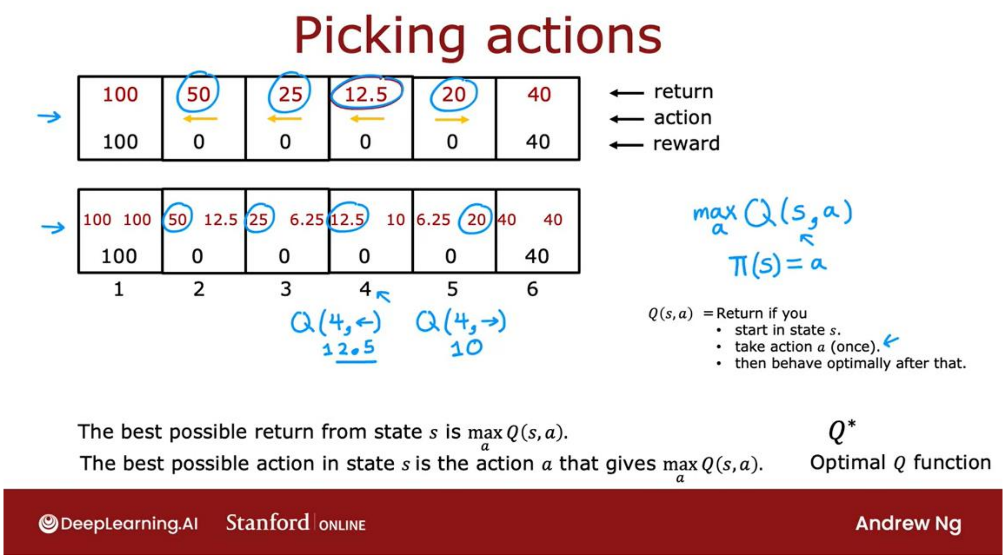
$Q(2,\leftarrow)=50$

$0+(0.5)100$

$Q(4,\leftarrow)=12.5$

$0+(0.5)0+(0.5)^2 0+(0.5)^3 100$

DeepLearning.AI Stanford ONLINE Andrew Ng



Picking actions

100	50	25	12.5	20	40	← return
	←	←	←	→		← action
100	0	0	0	0	40	← reward

1	2	3	4	5	6
100 100	50 12.5	25 6.25	12.5 10	6.25 20	40 40
100	0	0	0	0	40

$Q(4,\leftarrow)$ $\underline{12.5}$ $Q(4,\rightarrow)$ 10

$\max_a Q(s,a)$

$\pi(s)=a$

$Q(s,a)$ = Return if you
- start in state s.
- take action a (once).
- then behave optimally after that.

The best possible return from state s is $\max_a Q(s,a)$.

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

Q^* Optimal Q function

DeepLearning.AI Stanford ONLINE Andrew Ng

*Best possible return in a certain state s, is max of Q(s, a). And best possible action a in a state s, is the acttion that gives max of Q(s, a).

Q* is the same.

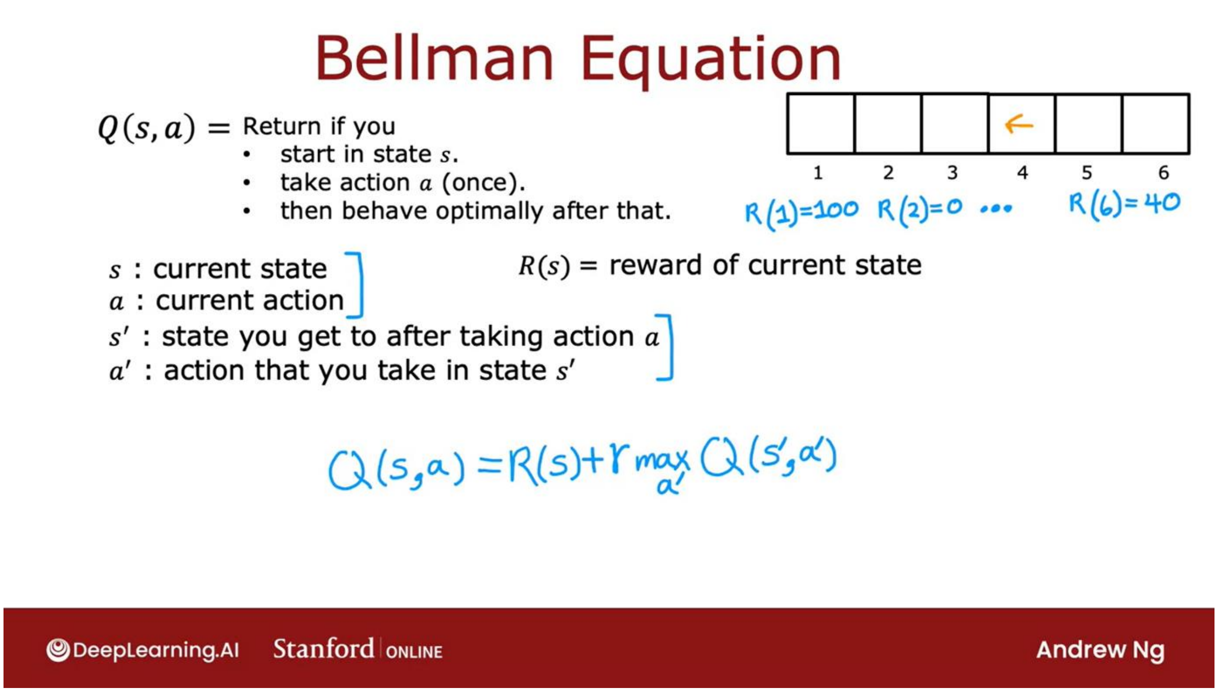
Example:

In lab you can play with values to see how it changes optimal policy. **As gamma decreases, agent becomes impatient**.

Bellman Equation

“Single most important equation for reinforcement learning.”

Until now: If you compute Q(s,a), you’ll be able to pick action a’s. Now only one question remains, how?



Bellman Equation

$Q(s,a) =$ Return if you
- start in state s.
- take action a (once).
- then behave optimally after that.

1	2	3	4	5	6
			$\leftarrow$		

$R(1)=100 \quad R(2)=0 \quad \cdots \quad R(6)=40$

$\left.\begin{array}{l} s : \text{current state} \\ a : \text{current action} \end{array}\right]$

$R(s) =$ reward of current state

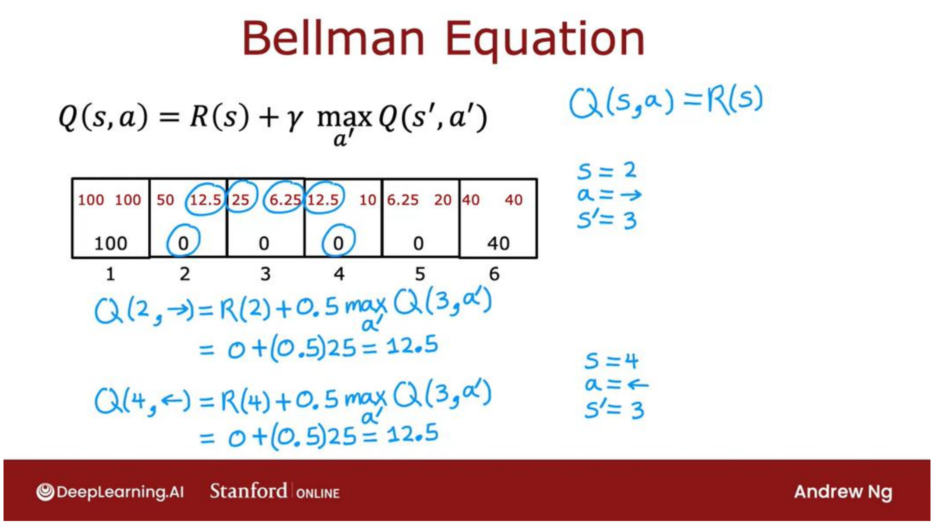
$\left.\begin{array}{l} s' : \text{state you get to after taking action } a \\ a' : \text{action that you take in state } s' \end{array}\right]$

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

Bellman Example →

In a terminal state, q function is taken as just R(s).

When you examine the function, you can see how it is chained.



Bellman Equation

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

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

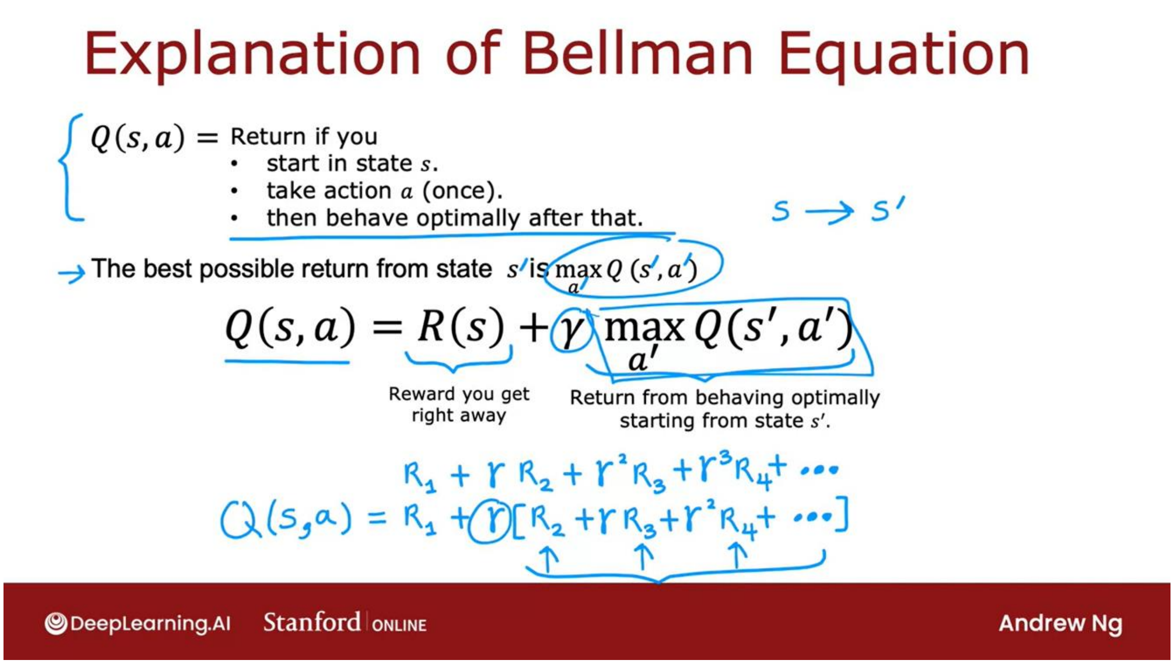
1	2	3	4	5	6
100 100	50 12.5	25 6.25	12.5 10	6.25 20	40 40
100	0	0	0	0	40

$s=2$
$a=\rightarrow$
$s'=3$

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

$s=4$
$a=\leftarrow$
$s'=3$

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



Explanation of Bellman Equation

$Q(s,a) =$ Return if you
- start in state s.
- take action a (once).
- then behave optimally after that.

$s \rightarrow s'$

→ The best possible return from state s' is $\max_{a'} Q(s',a')$

$$Q(s,a) = \underbrace{R(s)}_{\text{Reward you get right away}} + \gamma \underbrace{\max_{a'} Q(s',a')}_{\text{Return from behaving optimally starting from state } s'.}$$

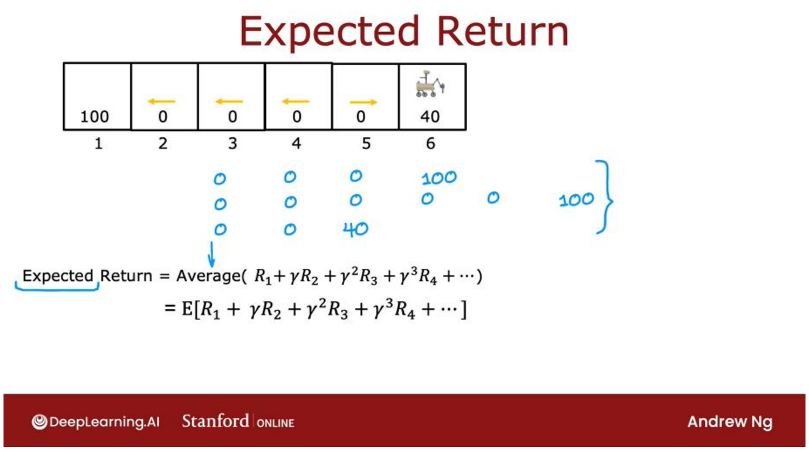
$$R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots$$
$$Q(s,a) = R_1 + \gamma[R_2 + \gamma R_3 + \gamma^2 R_4 + \cdots]$$

Effect of Random Action Selection

In practice, robots can not always do what you tell them to do. Sth can slip or sth bad can happen. A robot can not always go left (remember wrong step probability in the lab).

Stochastic environment example: You goal is to go left but only %90 of the times you do, %10 chance moves you 1 cell to the right.

**In this scenario, you can't just judge return from a situation with one example. You'd run it 1000 or a lot more and you'd get average of those values. Then your objective becomes "maximizing the expected return".


Expected Return

Expected Return = Average($R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots$)

$= \mathrm{E}[R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots]$

In a state of s and action to go left, s' can be 2 but it can also be 4. So bellman equation gets modified as having expected value of Q(s', a').

You'll see that as misstep probability increases, you lose your degree of control over the robot and returns will be lower.

Continuous state spaces

In a mars rover example, states are actually not discrete but continuous. It could be on a line where any point is valid.

**In a real world example, a truct might have various numbers corresponding to it's point in x and y axes, it's speed in these axes, it's angle etc. Meaning state in a vector containing many numbers.

For example in a helicopter: x position, y position, z height; orientation (three numbers: is it rolling, is it pitching forward or backward, which direction is it facing). **And also speed of change in all of these features (how fast is every single number changing)**.

These 12 numbers are fed as input to a policy, that policy looks at these numbers, decides what action to take.

Lunar Lander

There are 4 actions: do nothing, main thruster, left th, right th.

States are combinations of every feature it has: x, y, angle, change of these, l and r (did that leg touch the ground).

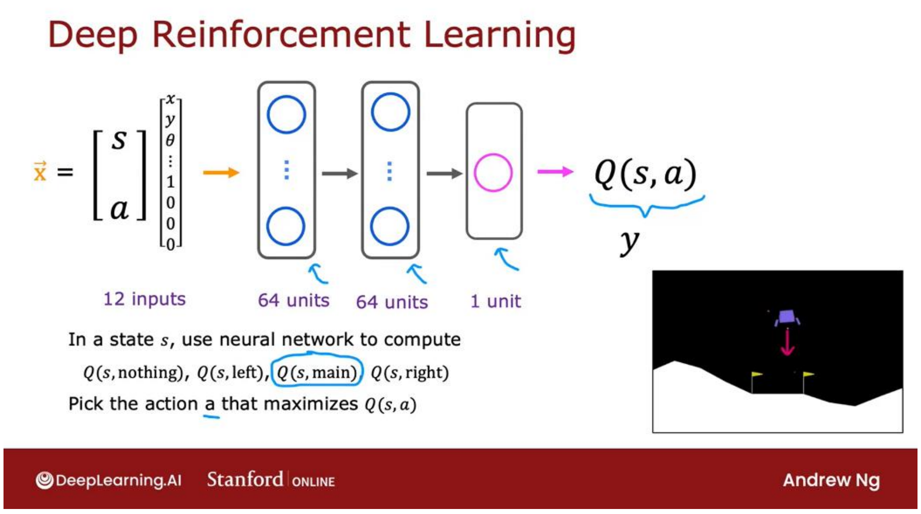
Then consider what you're going to reward (+ and -). Crash, land, soft landing, fire engines (this one's -), leg grounded are some situations you'll be considering.

"You'll see, when you're building your own RL application, that it takes some time to specift what you want or don't want and to codify these in reward." **At least you're just specifying good behaviour, not every good and bad action in every possible state…

**We usually pick a large value of gamma, close to 1, like 0.985.

Goal is to pick a policy that finds actions. For that we'll use deep learning and neural networks. (welcome to deep reinforcement learning)

Learning the state-value function: Q(s, a)



Deep Reinforcement Learning

$\vec{x} = \begin{bmatrix} s \\ a \end{bmatrix}$ $\begin{bmatrix} x \\ y \\ \theta \\ \vdots \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ → 64 units → 64 units → 1 unit → $\underbrace{Q(s,a)}_{y}$

In a state s, use neural network to compute

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

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

DeepLearning.AI Stanford ONLINE Andrew Ng

A neural network will get s and a, s has features and a has different choices so you can one-hot encode them. Using the neural network, we'll compute Q(s, a).

So question becomes: How do you train the neural network?

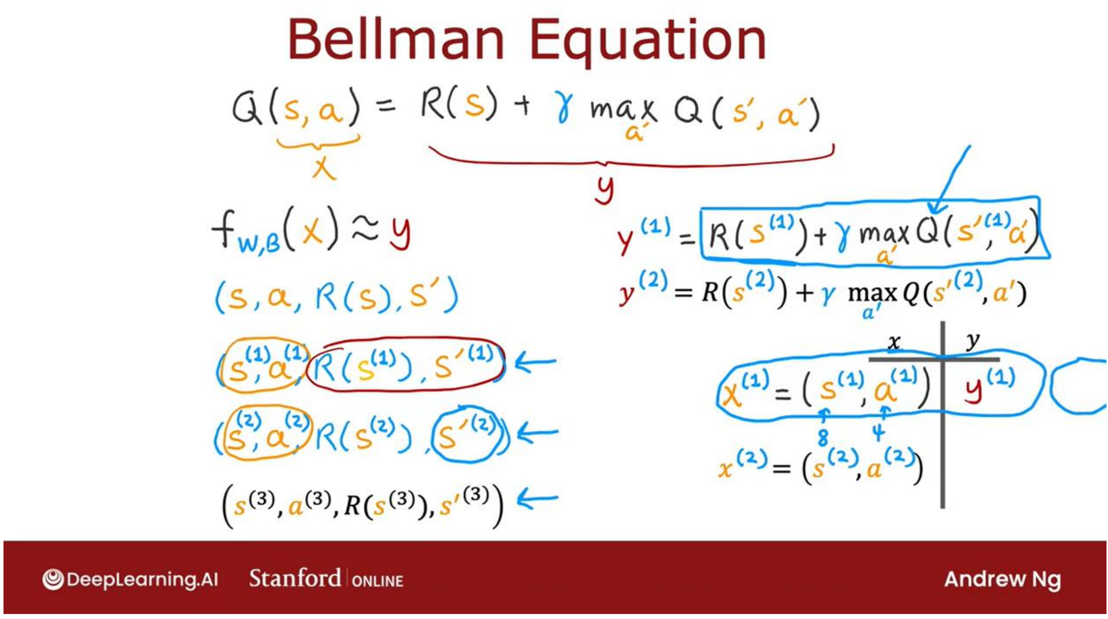
For training, you need a training set. We'll use Bellman's equation to create one with lot's of examples with x's and y's. Then we'll use supervised learning as we did in course 2.

But how do you get a set of pairs of x's and y's? For a neural network to learn from, you need mapping of x's and y's.

In lunar lander, we'll just take actions, good and bad, and obtain **(s, a, R(s), s')** pairs. If we don't have a policy yet, we can do random actions. We'll obtain a lot of tuples this way and **every one of these tuples are going to be enough to create a single training example, x_1 and y_1.**

HOW?

First 2 elements of the tuple gives you x. From last 2, you obtain y. You now R, and get Q from neural network.



Bellman Equation

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

$$f_{W,B}(x) \approx y$$

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

$$(s^{(1)}, a^{(1)}, R(s^{(1)}), s'^{(1)}) \leftarrow$$

$$(s^{(2)}, a^{(2)}, R(s^{(2)}), s'^{(2)}) \leftarrow$$

$$(s^{(3)}, a^{(3)}, R(s^{(3)}), s'^{(3)}) \leftarrow$$

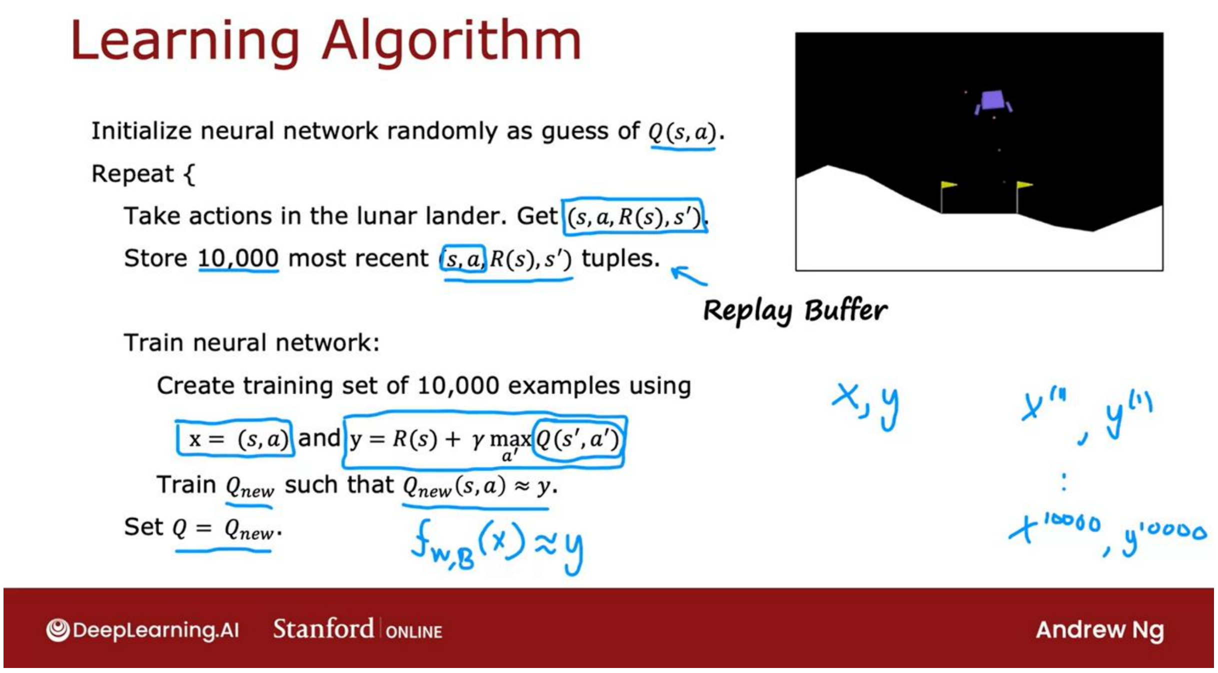
$$y^{(1)} = \boxed{R(s^{(1)}) + \gamma \max_{a'} Q(s'^{(1)}, a')}$$

$$y^{(2)} = R(s^{(2)}) + \gamma \max_{a'} Q(s'^{(2)}, a')$$

x	y
$x^{(1)} = (\underset{8}{s^{(1)}}, \underset{4}{a^{(1)}})$	$y^{(1)}$
$x^{(2)} = (s^{(2)}, a^{(2)})$	

DeepLearning.AI Stanford ONLINE Andrew Ng

To prevent excessive use of memory, in training only last 10.000 (number may vary) number of examples are remembered. This is referred as **replay buffer in reinforcement learning.



Learning Algorithm

```
Initialize neural network randomly as guess of $Q(s,a)$.
Repeat {
    Take actions in the lunar lander. Get $\boxed{(s, a, R(s), s')}$.
    Store $\underline{10{,}000}$ most recent $\boxed{s, a} R(s), s'$) tuples.   ← Replay Buffer

    Train neural network:
        Create training set of 10,000 examples using
        $\boxed{x = (s,a)}$ and $\boxed{y = R(s) + \gamma \max_{a'} \boxed{Q(s',a')}}$
        Train $Q_{new}$ such that $Q_{new}(s,a) \approx y$.      $f_{w,B}(x) \approx y$
    Set $Q = Q_{new}$.
}
```

x, y

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

$\vdots$

x^{10000}, y^{10000}

In every iteration you train a model and every model you obtain becomes a better estimate of the Q function. Run long eonugh and it'll be pretty good as an estimate.

This algorithm is called **DQN, or Deep Q-Network**. (you're using neural network to learn Q function)

Algorithm refinement

1) **Neural network architecture**

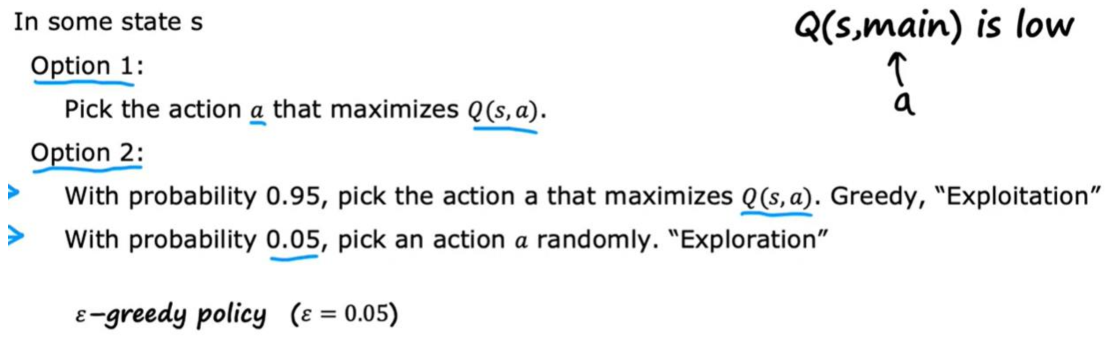
*For every move, what you need to do is to do inference 4 times (for lunar lander, meaning nthg, main th, right, left) and see which one is bigger. This is inefficient. Every state requires 4 inferences.

SOLUTION: Output all these values at the same. Remember what you output is Q for that move.

Your output layer will have 4 neurons now, one for every Q value.

2) **Epsilon-Greedy Policy** Effects how you choose actions, when you're still learning.

You're picking actions when you're learning. How do you choose these actions? First step in the iteration is to take actions but how? Most common way is to use **Epsilon-Greedy Policy**.



In some state s

Option 1:

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* $(\varepsilon = 0.05)$

$Q(s, \text{main})$ is low

$\uparrow$

a

Reason why we should pick random actions as well: Neural networks starts with a random idea about action (it's initialized with random parameters). Think that it has a bias against some actions. Maybe it starts thinking that firing main thruster is a bad idea. To break this bias, it needs to try out new things.

*****You've already heard this as "Exploration – Exploitation" steps.** First one exploits knowlegde, second one explores new things.

One trick in RL is to start with a high epsilon, so initially you can explore more. Then gradualy decrease it and you're more likely to pick good actions.

REINFORCEMENT LEARNING ALGORITHMS ARE **PICKY IN TERMS OF HYPERPARAMTER CHOICES**. In sup.lear. maybe setting learning rate small will result in 3x slower training. But in RL, wrong choice of epsilon could result in 10x or maybe 100x longer trainings.

3) **Mini batch and soft updates**

MINI BATCH

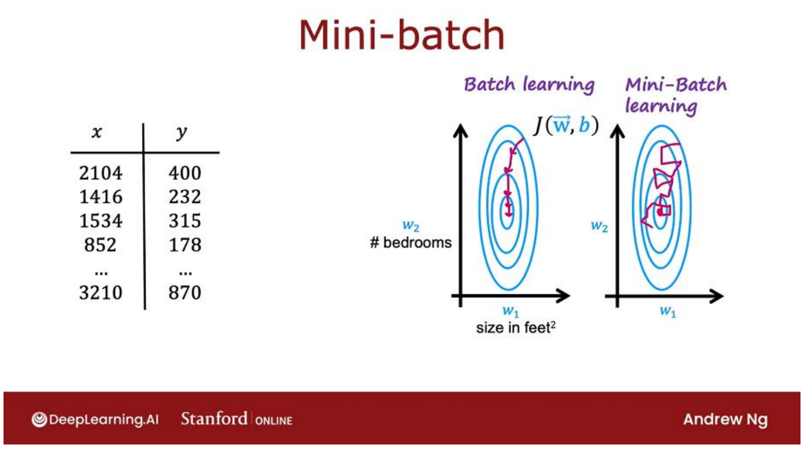
Can also be applied to supervised learning to speed it up.

Consider housing prices, you had 47 examples in the first course but what if you had 100 million houses? In that case, considering that you need to compute average cost to get it's partial derivatives so you can update every parameter, you'll be computing J with 100 million examples in every iteration. This will be slow.

IDEA: Insead of using m examples in every iteration, use m' which is a subset of m and is smaller. These subsets of data can be sequential.

**When applied to gradient descent, it kind of acts like SGD.

But every iteration becomes computationally reasonable.



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

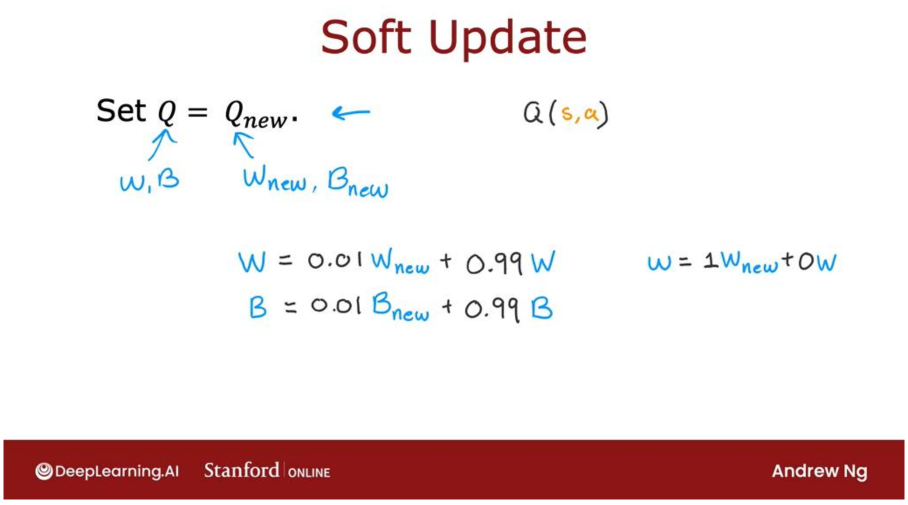
When you have a very large training set mini-batch is used commonly.

To go back into RL, when you stored tuples of last 10.000 tries, you can create a training set of only 1000 training examples. Noise increases but becomes faster, you eventually becomes better.

SOFT UPDATES

In the last row of "Learning algorithm", we set Q = Q_new. Maybe new neural network is not better, even a little bit worse. Soft update prevents Q from getting worse just because of an unlucky iteration.

This makes convergence more reliable and prone to noise.



Soft Update

Set $Q = Q_{new}$. $\leftarrow$ $Q(s,a)$

$\underset{W, B}{Q}$ = $\underset{W_{new}, B_{new}}{Q_{new}}$

$W = 0.01 W_{new} + 0.99 W$ $\quad$ $W = 1 W_{new} + 0 W$

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

DeepLearning.AI Stanford ONLINE Andrew Ng

State of Reinforcement Learning

"There are a lot of hype so it's important for you to really grasp it's place.

Many of today's researches are done in simulations, **but getting to work on a robot is much harder than controlling a robot**. **Real world applications of RL is harder**.

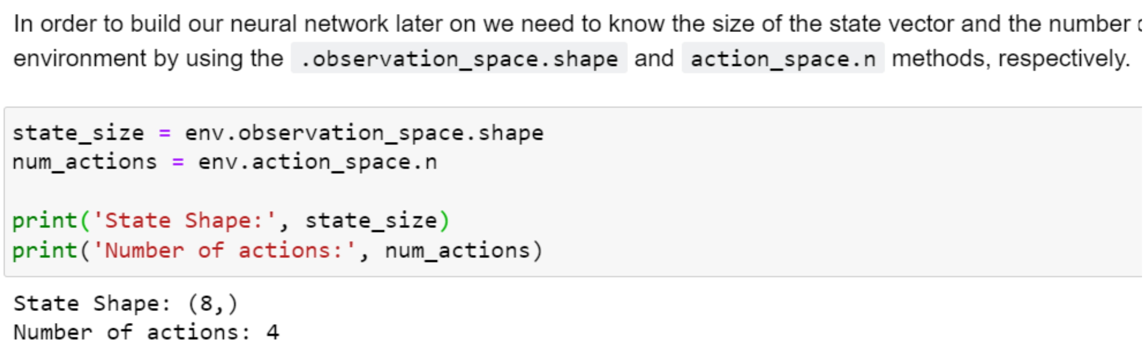
There are far fewer applications than supervised and unsupervised learning. You're more likely to use them for an application you're doing.

Research direction is interesting. **SO FOLLOW IT!!!**"

NOTES FROM LAST PRACTICE LAB OF SPECIALIZATON

- """In this notebook we will be using OpenAI's Gym Library. The Gym library provides a wide variety of environments for reinforcement learning. To put it simply, an environment represents a problem or task to be solved. In this notebook, we will try to solve the Lunar Lander environment using reinforcement learning."""

- Understanding rules of the environment is vital.
- In order to construct out neural network, we need to know **size of the state vector** and **how many valid actions are possible**.



In order to build our neural network later on we need to know the size of the state vector and the number
environment by using the `.observation_space.shape` and `action_space.n` methods, respectively.

```
state_size = env.observation_space.shape
num_actions = env.action_space.n

print('State Shape:', state_size)
print('Number of actions:', num_actions)
```

```
State Shape: (8,)
Number of actions: 4
```

6 - Deep Q-Learning

In cases where both the state and action space are discrete we can estimate the action-value function iteratively by using the Bellman equation:

$$Q_{i+1}(s,a) = R + \gamma \max_{a'} Q_i(s',a')$$

This iterative method converges to the optimal action-value function $Q^*(s,a)$ as $i \to \infty$. This means that the agent just needs to gradually explore the state-action space and keep updating the estimate of $Q(s,a)$ until it converges to the optimal action-value function $Q^*(s,a)$. However, in cases where the state space is continuous it becomes practically impossible to explore the entire state-action space. Consequently, this also makes it practically impossible to gradually estimate $Q(s,a)$ until it converges to $Q^*(s,a)$.

In the Deep Q-Learning, we solve this problem by using a neural network to estimate the action-value function $Q(s,a) \approx Q^*(s,a)$. We call this neural network a Q-Network and it can be trained by adjusting its weights at each iteration to minimize the mean-squared error in the Bellman equation.

Unfortunately, using neural networks in reinforcement learning to estimate action-value functions has proven to be highly unstable. Luckily, there's a couple of techniques that can be employed to avoid instabilities. These techniques consist of using a ***Target Network*** and ***Experience Replay***. We will explore these two techniques in the following sections.

A VEEEEEERY IMPORTANT NOTE

- Read **6.1 target network** section again.
  - It basically uses two neural networks to form the cost function, to compute the error. One of these is the network we use to compute Q's, and other one is used to compute Q^'s. This is called **Target Q-Network**.

6.2 Experience Replay

When an agent interacts with the environment, the states, actions, and rewards the agent experiences are sequential by nature. If the agent tries to learn from these consecutive experiences it can run into problems due to the strong correlations between them. To avoid this, we employ a technique known as **Experience Replay** to generate uncorrelated experiences for training our agent. Experience replay consists of storing the agent's experiences (i.e the states, actions, and rewards the agent receives) in a memory buffer and then sampling a random mini-batch of experiences from the buffer to do the learning. The experience tuples (S_t, A_t, R_t, S_{t+1}) will be added to the memory buffer at each time step as the agent interacts with the environment.

For convenience, we will store the experiences as named tuples.

Algorithm 1: Deep Q-Learning with Experience Replay

```
Initialize memory buffer $D$ with capacity $N$
Initialize $Q$-Network with random weights $w$
Initialize target $\hat{Q}$-Network with weights $w^- = w$
for episode $i = 1$ to $M$ do
    Receive initial observation state $S_1$
    for $t = 1$ to $T$ do
        Observe state $S_t$ and choose action $A_t$ using an $\epsilon$-greedy policy
        Take action $A_t$ in the environment, receive reward $R_t$ and next state $S_{t+1}$
        Store experience tuple $(S_t, A_t, R_t, S_{t+1})$ in memory buffer $D$
        Every $C$ steps perform a learning update:
        Sample random mini-batch of experience tuples $(S_j, A_j, R_j, S_{j+1})$ from $D$
        Set $y_j = R_j$ if episode terminates at step $j+1$, otherwise set $y_j = R_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a')$
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; w))^2$ with respect to the $Q$-Network weights $w$
        Update the weights of the $\hat{Q}$-Network using a soft update
    end
end
```

By using experience replay we avoid problematic correlations, oscillations and instabilities. In addition, experience replay also allows the agent to potentially use the same experience in multiple weight updates, which increases data efficiency.

***Also full cycle of reinforcement learning for this example is shown on figure above.** Note that target network also gets to be updated, but through a soft update.

- **SECTION 9 TELLS ABOUT EVERYTHING WE HAVE DONE SO FAR. Examining is a must.**
- Also see utils.get_action function.

Books on Deep Q Learning:

- Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. Nature 518, 529–533 (2015).
- Lillicrap, T. P., Hunt, J. J., Pritzel, A., et al. Continuous Control with Deep Reinforcement Learning. ICLR (2016).
- Mnih, V., Kavukcuoglu, K., Silver, D. et al. Playing Atari with Deep Reinforcement Learning. arXiv e-prints. arXiv:1312.5602 (2013).

CONCLUSION FOR ENTIRE SPECIALIZATION

Looking back: Course 1 was regression and classification. Course 2 included neural networks, decision trees, **advice for ML**. Course 3 was on uns.lea., recommenders and RL.

These are some broad set of tools which give you a wide range of possible roads for future.

That's it! You did it pal, congrats!

Next up: Deep learning specialization, natural language processing specialization.