Day-100, Mar 10-2025, Falgun 26, 2081 BS.

① Bell Man Equation (12min)

② Making Decisions (2min)

③ State-Action function Definition & Example (15min)

④ Learning the State-value function (16min)

⑤ Algorithm Refinement ε-greedy policy

⑥ The State of Reinforcement Learning.

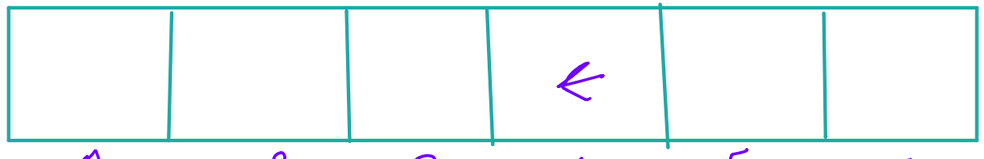
⑦ Acknowledgements:

⑧ What's Next?

Bellman Equation # Q(s,a) → Help to compute Q s of a.

Bellman Equation

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



R(1) = 100 R(2) = 0 ... R(6) = 40

R(s) = reward of current state

s : current state

a: Current action

s': state you get to after taking action a

a': action that you take in state s'

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

Bellman Equation

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

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

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

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

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

$\Rightarrow 0 + 0.5(25)$

$[\, Q(2, \rightarrow) \Rightarrow 12.5 \,]$

$Q(4, \leftarrow) \Rightarrow R(4) + 0.5 \max_{a'} (3, a')$

$\Rightarrow 0 + (0.5) \times 25$

$[\, Q(4, \leftarrow) \Rightarrow 12.5 \,]$

DeepLearning.AI Stanford ONLINE Andrew Ng

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 $\max_{a'} Q(s', a')$

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

$R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \cdots\cdots$

Broken down into Components

$\Rightarrow R_1 + \gamma [R_2 + \gamma R_3 + \gamma^2 R_4 + \cdots]$

$\max_{a'} Q(S', a')$ Best possible Return

$R(S) \rightarrow$ Immediate Reward

$\max_{a'} Q(S', a') \rightarrow$ Return from behaving optimaly Starting from state S'

$Q(S, a) = R_1(S) + \gamma \max_{a'} Q(S', a')$

3. Bellman Equation Explanation

The Bellman equation is:

$$Q(S, A) = R(S) + \gamma \max_{A'} Q(S', A')$$

Where:

- S' is the **next state** after taking action A from state S.
- A' is the **next action** to take from S'.
- $\max_{A'} Q(S', A')$ represents the **best possible future return** from state S'.

Breaking It Down:

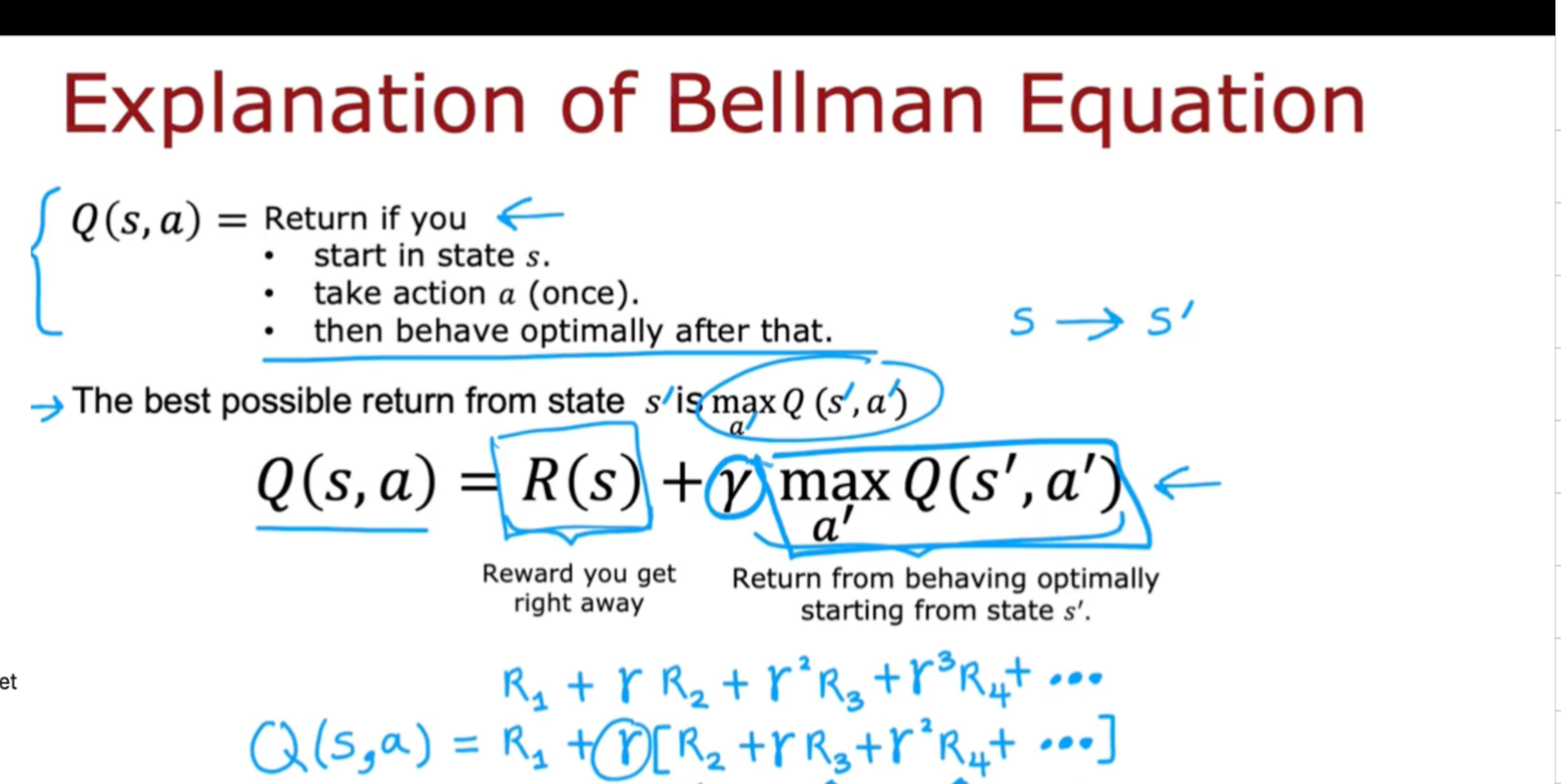
1. **First part:** $R(S)$ is the **immediate reward** for being in state S.
2. **Second part:** $\gamma \max_{A'} Q(S', A')$ represents the **discounted future rewards**—what you will get **starting from S' and behaving optimally**.

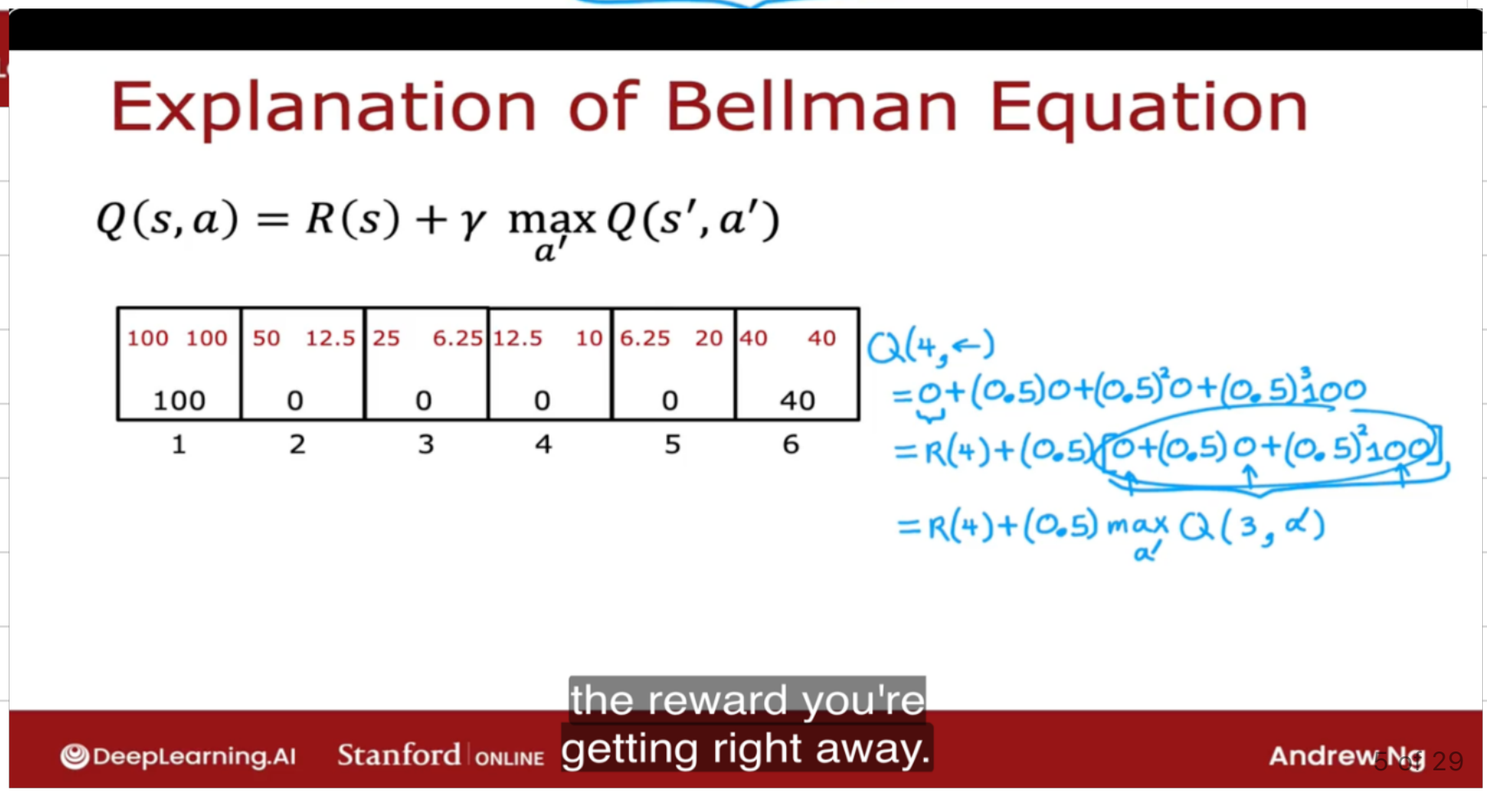
4. Example Calculation

Consider the **Markov Decision Process (MDP)** example in the lecture:

- The reward values for states:
  - $R(1) = 100$
  - $R(2) = 0$
  - $R(3) = 0$
  - $R(4) = 0$
  - $R(5) = 0$
  - $R(6) = 40$
- Suppose the **discount factor** is $\gamma = 0.5$.

Computing $Q(2, \text{right})$:





Computing $Q(2, \text{right})$:

- If you're in **State 2** and go **right**, you move to **State 3**.
- The Bellman equation gives:

$$Q(2, \text{right}) = R(2) + \gamma \max_{A'} Q(3, A')$$

- Since $R(2) = 0$, and the two possible $Q(3, A')$ values are **25** and **6.25**:

$$Q(2, \text{right}) = 0 + 0.5 \times 25 = 12.5$$

Computing $Q(4, \text{left})$:

- If you're in **State 4** and go **left**, you move to **State 3**.

$$Q(4, \text{left}) = R(4) + \gamma \max_{A'} Q(3, A')$$

- Since $R(4) = 0$ and the best possible value from **State 3** is **25**:

$$Q(4, \text{left}) = 0 + 0.5 \times 25 = 12.5$$

Terminal States:

For **terminal states**, the Bellman equation simplifies to:

$$Q(S, A) = R(S)$$

because there's no future state (S') to consider.

For example:

- If $S = 1$, then $Q(1, A) = 100$.
- If $S = 6$, then $Q(6, A) = 40$.

---

5. The Intuition Behind the Bellman Equation

The Bellman equation **breaks down the total expected return into two parts**:

1. **Immediate reward** $R(S)$ (what you get right now).

5. The Intuition Behind the Bellman Equation

The Bellman equation **breaks down the total expected return into two parts**:

1. **Immediate reward** $R(S)$ (what you get right now).
2. **Future rewards**, discounted by γ, assuming optimal actions thereafter.

Mathematically, this breakdown follows from the way rewards accumulate over time:

$$\text{Total Return} = R_1 + \gamma R_2 + \gamma^2 R_3 + \dots$$

Rearranging, we get:

$$\text{Total Return} = R(S) + \gamma(\text{Future Return})$$

This leads to the **Bellman equation**.

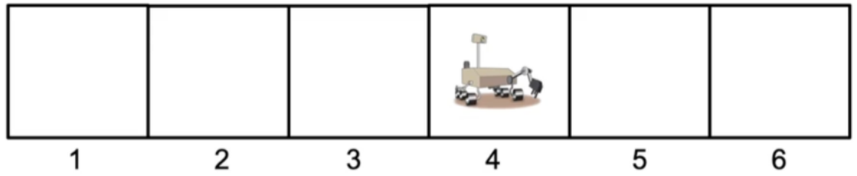
---

6. Key Takeaways

- **Bellman equation recursively defines $Q(S, A)$.**
- **To find the best action in a state, choose the one that maximizes $Q(S, A)$.**
- **The equation helps compute optimal policies in RL.**
- **If you understand the basic intuition, you can apply the equation even if the details seem complex.**

Discrete vs Continuous State

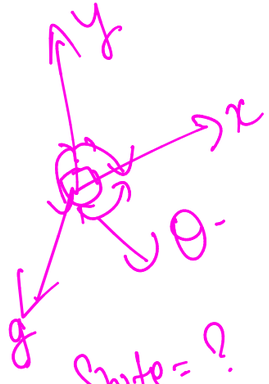
Discrete State:



Continuous State:

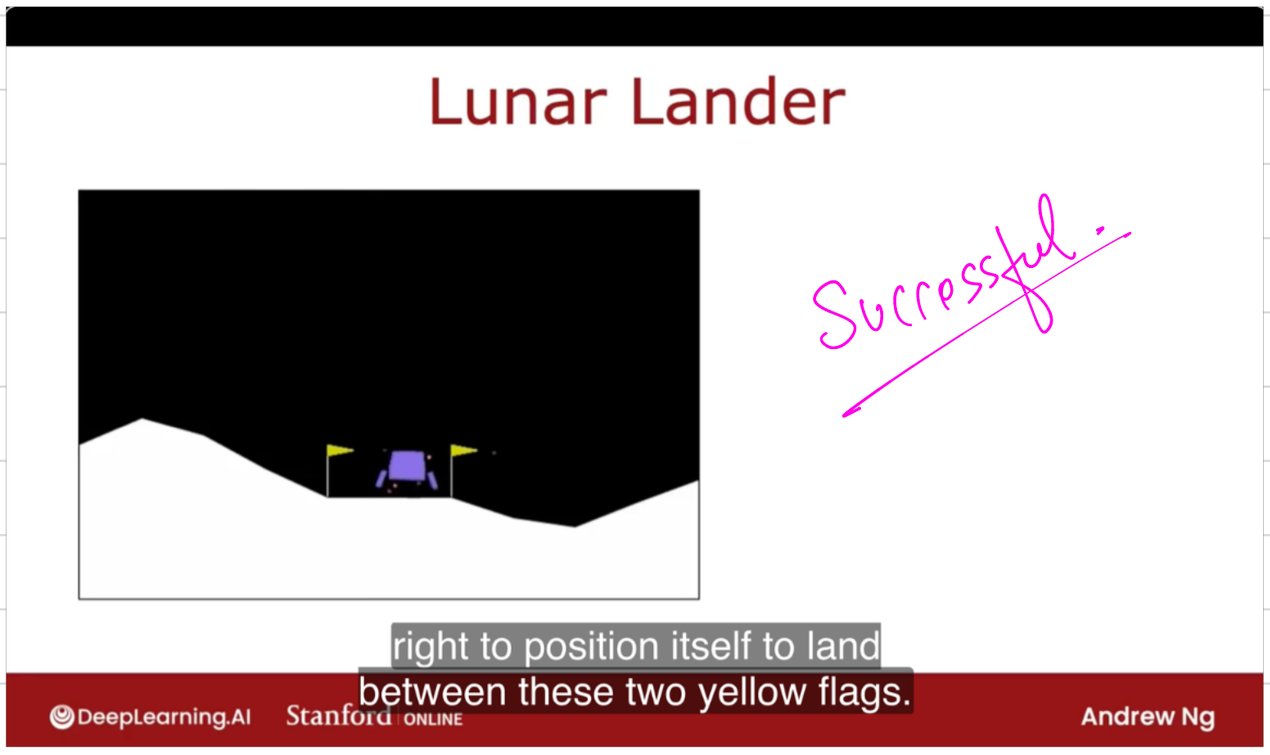
DeepLearning.AI Stanford ONLINE instead, they can be in any of Andrew Ng

Autonomous Helicopter (3D)

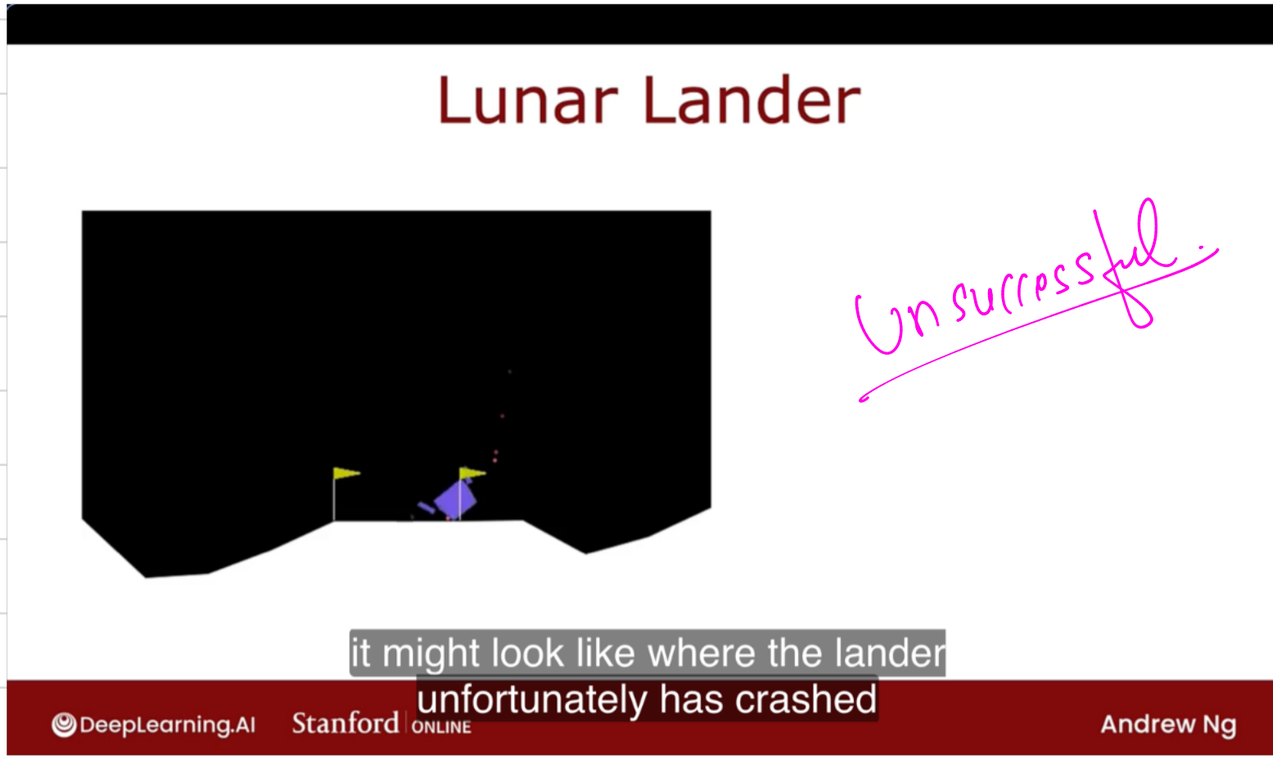


DeepLearning.AI Stanford ONLINE control an autonomous helicopter, Andrew Ng

#1 Lunar Lander (Simulated).



Successful.



Unsuccessful.

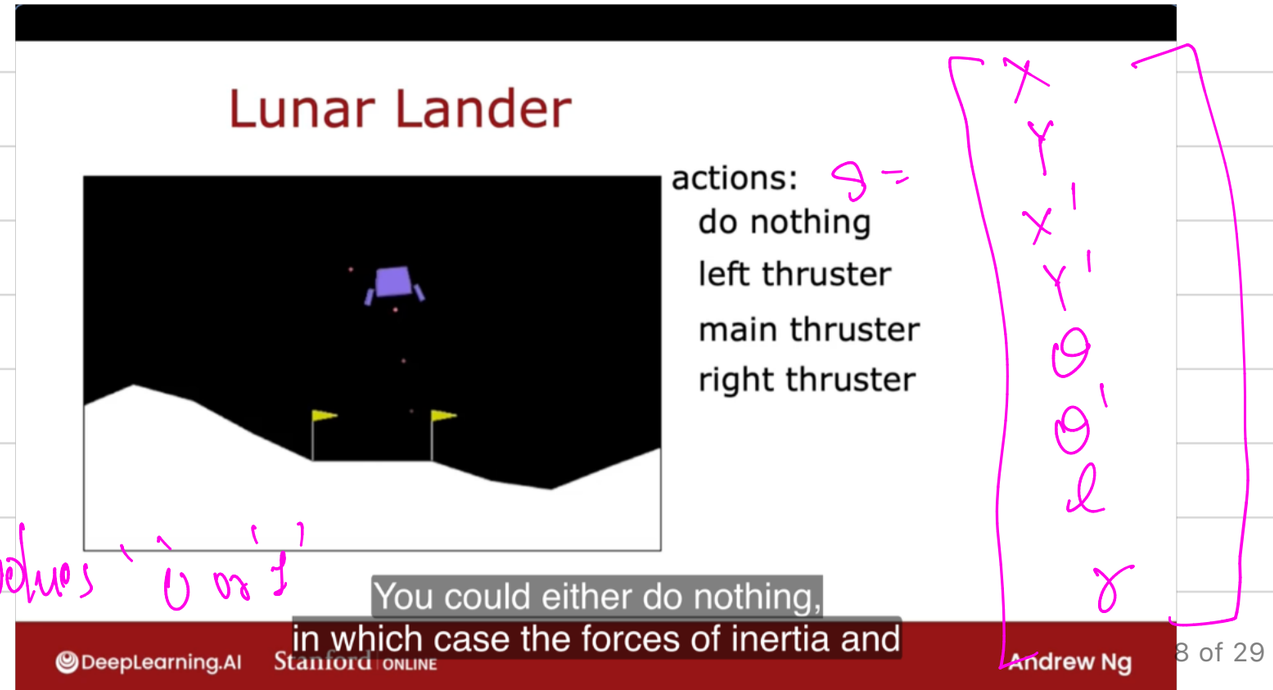
→ Actions that make Safe landing.

→ Means various Correct actions must be triggered at a sequence of time

left → l leg.

r → r right leg

x = y = binary values '0 or 1'



$\vartheta = \begin{bmatrix} x \\ y \\ x' \\ y' \\ \theta \\ \theta' \\ l \\ r \end{bmatrix}$

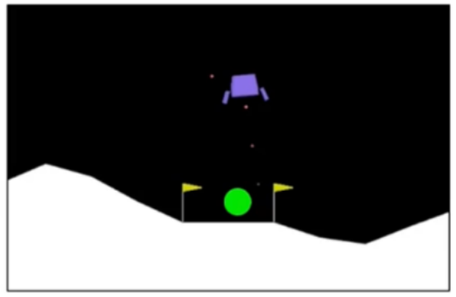
Lunar Lander and Reinforcement Learning

The **Lunar Lander** problem is a reinforcement learning environment where an agent (the lander) learns to land safely on a pad by optimizing its actions.

1. **State Space**:
   - The lander's state is represented by:
     - Position: x, y
     - Velocity: x', y' (horizontal and vertical speeds)
     - Angle: θ (tilt)
     - Angular velocity: θ'
     - Leg contact indicators: L and R (binary values indicating whether the left and right legs are touching the ground).
2. **Action Space**:
   - The agent can take one of four discrete actions:
     - **Do nothing**: Gravity pulls the lander downward.
     - **Fire left thruster**: Pushes the lander to the right.
     - **Fire right thruster**: Pushes the lander to the left.
     - **Fire main thruster**: Slows the descent.
3. **Reward Function**:
   - The goal is to reward good landings and penalize bad behavior:
     - Landing on the pad: **+100 to +140**.
     - Moving toward the pad: **positive reward**.
     - Moving away: **negative reward**.
     - Crashing: **-100**.
     - Landing softly: **+100 per leg**.
     - Landing with left or right leg: **+10 per leg**.
     - Fuel usage penalty:
       - Main engine: **-0.3 per use**.

Reward Function

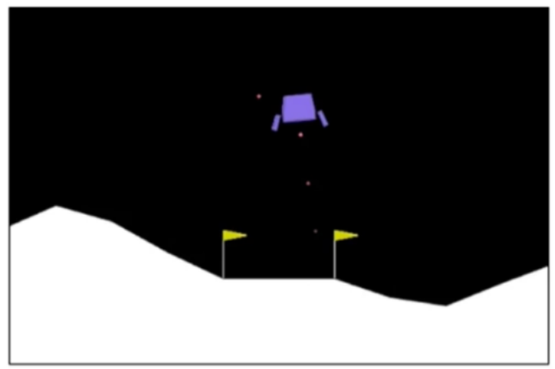
- Getting to landing pad: 100 – 140
- Additional reward for moving toward/away from pad.
- Crash: -100
- Soft landing: +100
- Leg grounded: +10
- Fire main engine: -0.3
- Fire side thruster: -0.03



Lunar Lander Problem

Learn a policy π that, given $s = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \\ \theta \\ \dot{\theta} \\ l \\ r \end{bmatrix}$

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



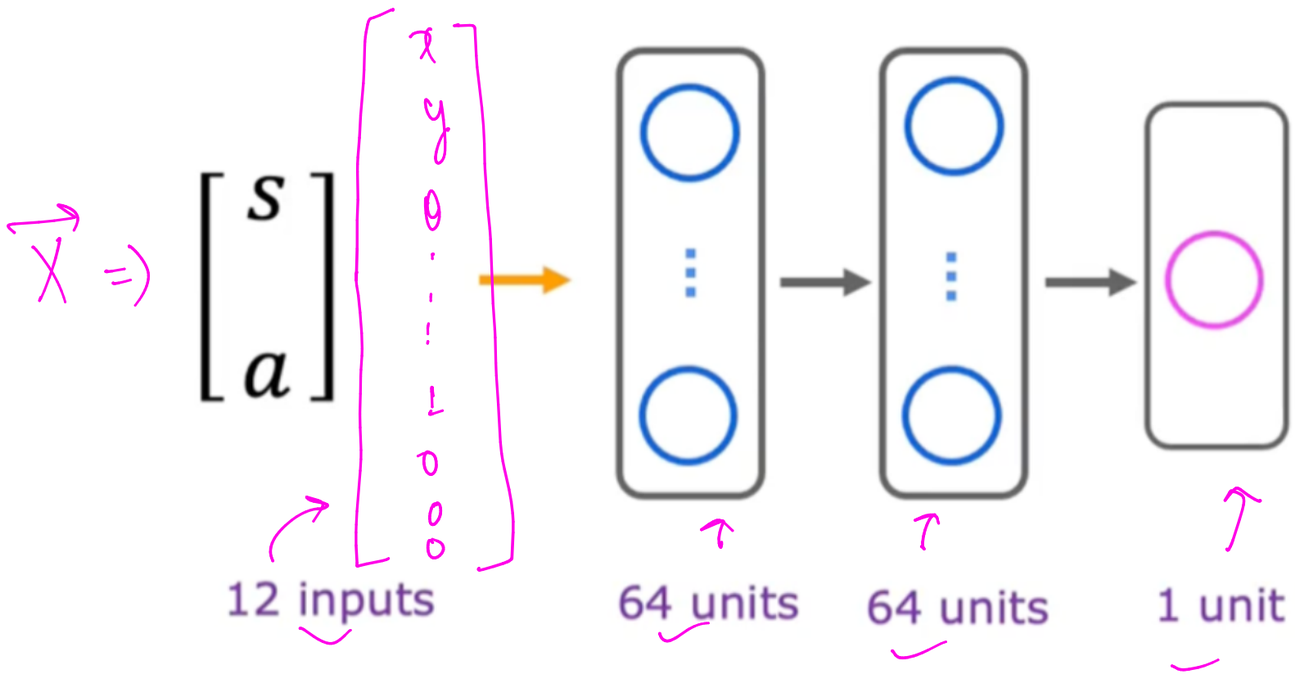
use large value
$\gamma = 0.985$

Our goal is to learn a policy pi.

DeepLearning.AI Stanford ONLINE Andrew Ng

Deep learning (State-Value Q^* function) ⇨ train NN to approximate $Q(s,a)$.

Deep Reinforcement Learning



Q^*

(Input given in pair)

In a state s, use neural network to compute

$Q(s,\text{nothing})$, $Q(s,\text{left})$, $Q(s,\text{main})$, $Q(s,\text{right})$ → One hot feature vector: [0 1 0 0]

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

↑ decides which action

any action a and put them together.

DeepLearning.AI Stanford ONLINE Andrew Ng

Bellman Equation

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

x — y

$f_{w,b}(x) \approx y$

$(S, a, R(S), S')$

$(S^{(2)}, a^{(1)}, R(S^{(1)}), S'^{(1)})$ ←

$(S^{(2)}, a^{(2)}, R(S^{(2)}), S'^{(2)})$ ←

$(S^{(3)}, a^{(3)}, R(S^{(3)}), S'^{(3)})$ ←

Input

No policy then
Take Random Actions.

Result of Action gives, S'

Initially it's Random

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

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

x	y
$x^{(1)} = (S^{(1)}, a^{(1)})$ ↑ $(8, 4)$	$y^{(1)}$
$x^{(2)} = (S^{(2)}, a^{(2)})$	$y^{(2)}$

First Training Example

a equals R of s plus Gamma,

Bellman Equation (A piece RL Algorithm)

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

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

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

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

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

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

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

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

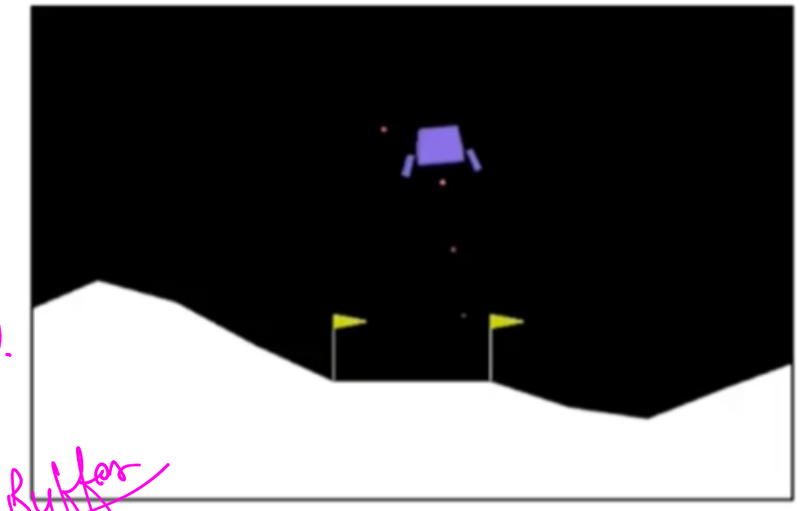
x	y
$x^{(1)} = (s^{(1)}, a^{(1)})$ (8, 4)	$y^{(1)}$
$x^{(2)} = (s^{(2)}, a^{(2)})$	$y^{(2)}$
$x^{(10,000)}$	$y^{(10,000)}$

What I describe here is

DeepLearning.AI Stanford ONLINE Andrew Ng

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

Train Neural Network.

Create training set of 10,000 examples using

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

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

Set $Q = Q_{new}$.

}

$x_1 \quad y \quad x^{(1)}, y^{(1)}$

$\vdots$

$x^{(10000)}, y^{(10000)}$

$f_{w,b}(x) \approx y$

training linear regression and you

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

Train neural network:

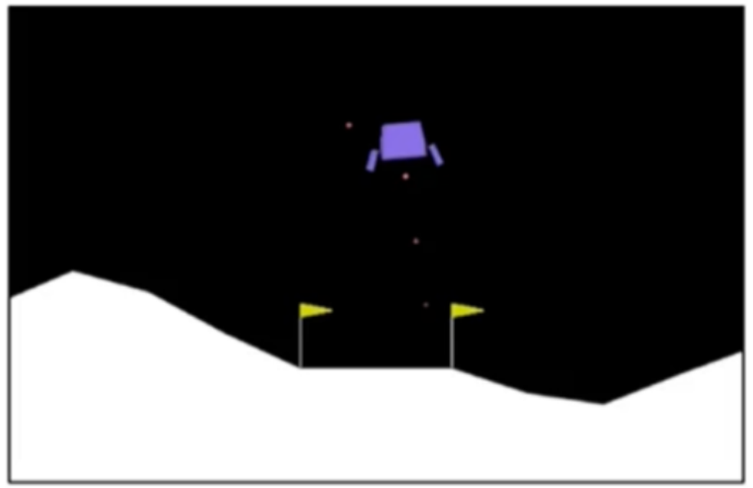
Create training set of 10,000 examples using

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

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

Set $Q = Q_{new}$.

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



x, y

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

$x^{(10000)}, y^{(10000)}$

Then for the next time you train

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

1. Problem Setup

The goal of reinforcement learning is to approximate the **state-action value function** $Q(s, a)$, which helps in selecting optimal actions.

For **Lunar Lander**, the state s consists of:

- (x, y) position
- $(\dot{x}, \dot{y})$ velocity
- $(\theta, \dot{\theta})$ orientation
- Leg contact indicators (Left, Right)

The actions a are **one-hot encoded**:

- **Nothing**: $[1, 0, 0, 0]$
- **Left thruster**: $[0, 1, 0, 0]$
- **Main thruster**: $[0, 0, 1, 0]$
- **Right thruster**: $[0, 0, 0, 1]$

Need to be Rewarded

2. Q-Function Approximation with a Neural Network

A neural network is used to approximate $Q(s, a)$, where:

- **Input**: $X = [s, a]$ (12 features: 8 for state, 4 for action)
- **Architecture**:
  - Hidden layer 1: 64 neurons
  - Hidden layer 2: 64 neurons
  - Output: Single value $Q(s, a)$

2. Q-Function Approximation with a Neural Network

A neural network is used to approximate $Q(s, a)$, where:

- **Input**: $X = [s, a]$ (12 features: 8 for state, 4 for action)
- **Architecture**:
  - Hidden layer 1: 64 neurons
  - Hidden layer 2: 64 neurons
  - Output: Single value $Q(s, a)$

3. Algorithm: Deep Q-Network (DQN)

Step 1: Initialize Neural Network

- Initialize all parameters **randomly**
- This serves as an initial guess for $Q(s, a)$

Step 2: Collect Experience Tuples

- Run the **Lunar Lander** simulator by taking random actions
- Store experience tuples:

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

- Maintain a **Replay Buffer** storing the last 10,00[illegible] tuples

2. Q-Function Approximation with a Neural Network

A neural network is used to approximate $Q(s, a)$, where:

- **Input**: $X = [s, a]$ (12 features: 8 for state, 4 for action)
- **Architecture**:
  - Hidden layer 1: 64 neurons
  - Hidden layer 2: 64 neurons
  - Output: Single value $Q(s, a)$

We train the network to predict $Q(s, a)$ using the **Bellman equation**:

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

where:

- $R(s)$ is the reward at state s
- s' is the next state after taking action a
- γ is the discount factor (typically 0.99)

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

Step 4: Train Neural Network

- Use **Mean Squared Error (MSE) Loss**:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (Q(s^i, a^i) - y^i)^2$$

- Train using **Gradient Descent**

Step 5: Update Q-function

- Replace the old Q-function with the newly trained model

Step 6: Repeat

- Continue collecting new experiences, updating the neural network, and refining $Q(s, a)$
- Over time, $Q(s, a)$ improves, leading to better actions

4. Summary of Key Equations

1. **Bellman Equation:**

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

2. **Target for Supervised Learning:**

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

3. **Loss Function (MSE):**

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (Q(s^i, a^i) - y^i)^2$$

This **DQN Algorithm** allows reinforcement learning to efficiently learn optimal control strategies for the **Lunar Lander** problem. 🚀

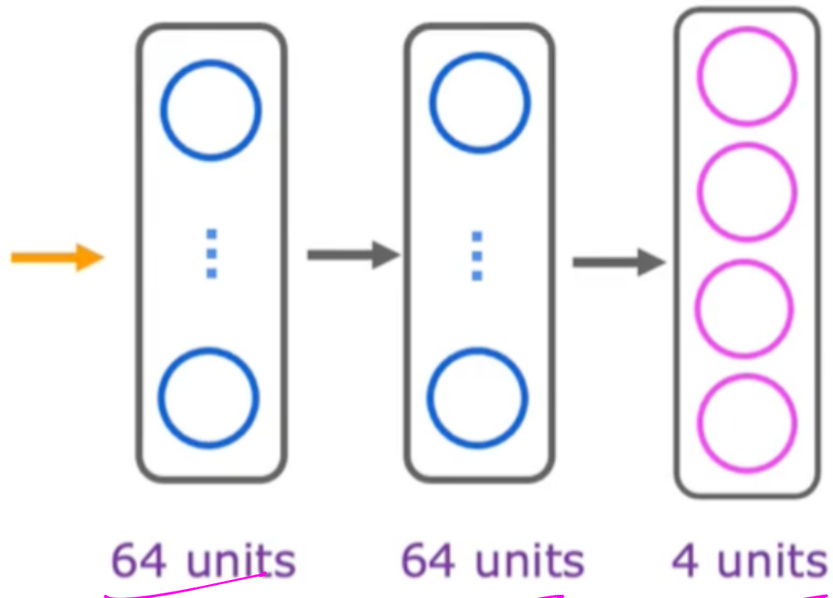
Algorithm Refinement:

Deep Reinforcement Learning

(Modified / Better).

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

8 Inputs



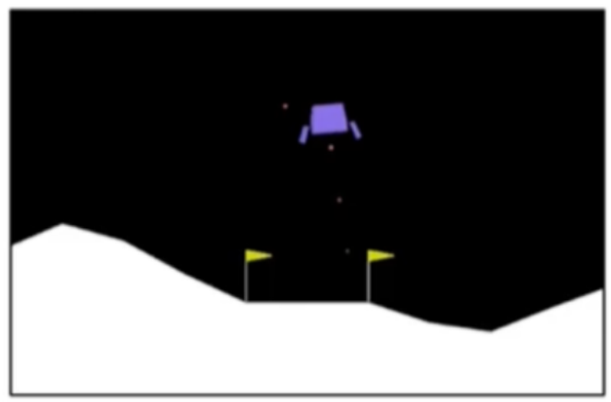
$Q(s, \text{nothing})$
$Q(s, \text{left})$
$Q(s, \text{main})$
$Q(s, \text{right})$

In a state s, input s to neural network.

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

Train S' at a time.

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



Here's a modified neural network

Algorithm Refinement ε-greedy policy:

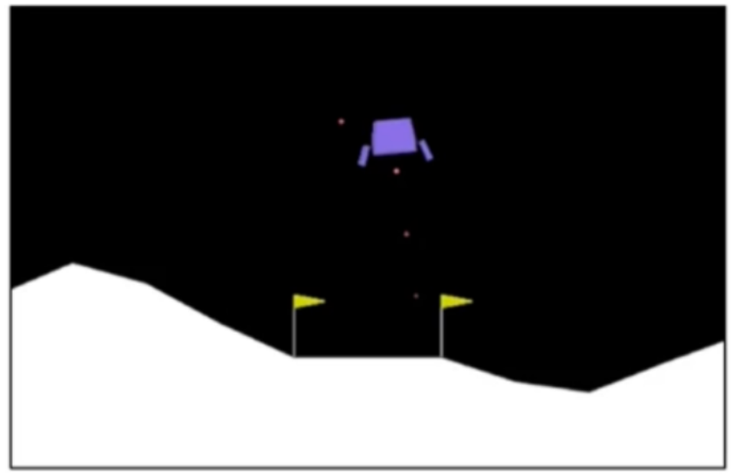
Learning Algorithm

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.



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 $Q = Q_{new}$.

→ NN Algorithm to approximate $Q(s,a)$

Here's the algorithm that you saw earlier.

DeepLearning.AI Stanford ONLINE Andrew Ng

How to choose actions while still learning?

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 $Q(s,a)$.

→ With probability 0.05, pick an action a randomly.

When you're in some state s,

How to choose actions while still learning?

In some state s

Q(s,main) is low

↑

a

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 ($\epsilon = 0.05$)

0.95 ⇒ ϵ

0.05 ⇒

→ Start ϵ high 1.0 ~ 0.01

→ Gradually decrease

versus trying to maximize your return by say,

DeepLearning.AI Stanford ONLINE Andrew Ng

RL is quite flexible in terms parameters compared to Supervised Learning-

Mini-Batch and Soft Updates → (Helps RL Algorithm efficiency)

How to choose actions while still learning?

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

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

$m = 100,000,000$

House Sizes

repeat {

$$w = w - \alpha \frac{\partial}{\partial w} J(w, b)$$

$$b = b - \alpha \frac{\partial}{\partial b} J(w, b)$$

}

Gradient Descent · becomes slower for larger m.

Problem ?
↳ Computing Intensive. Σ, Average and Do Backpropgath and so on.

way back in the first course of this specialization

DeepLearning.AI Stanford | ONLINE Andrew Ng

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 {

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

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

}

Look at 1,000, rather than all.

(Mini-Batch)

we would pick some subset of 1,000 or m prime examples.

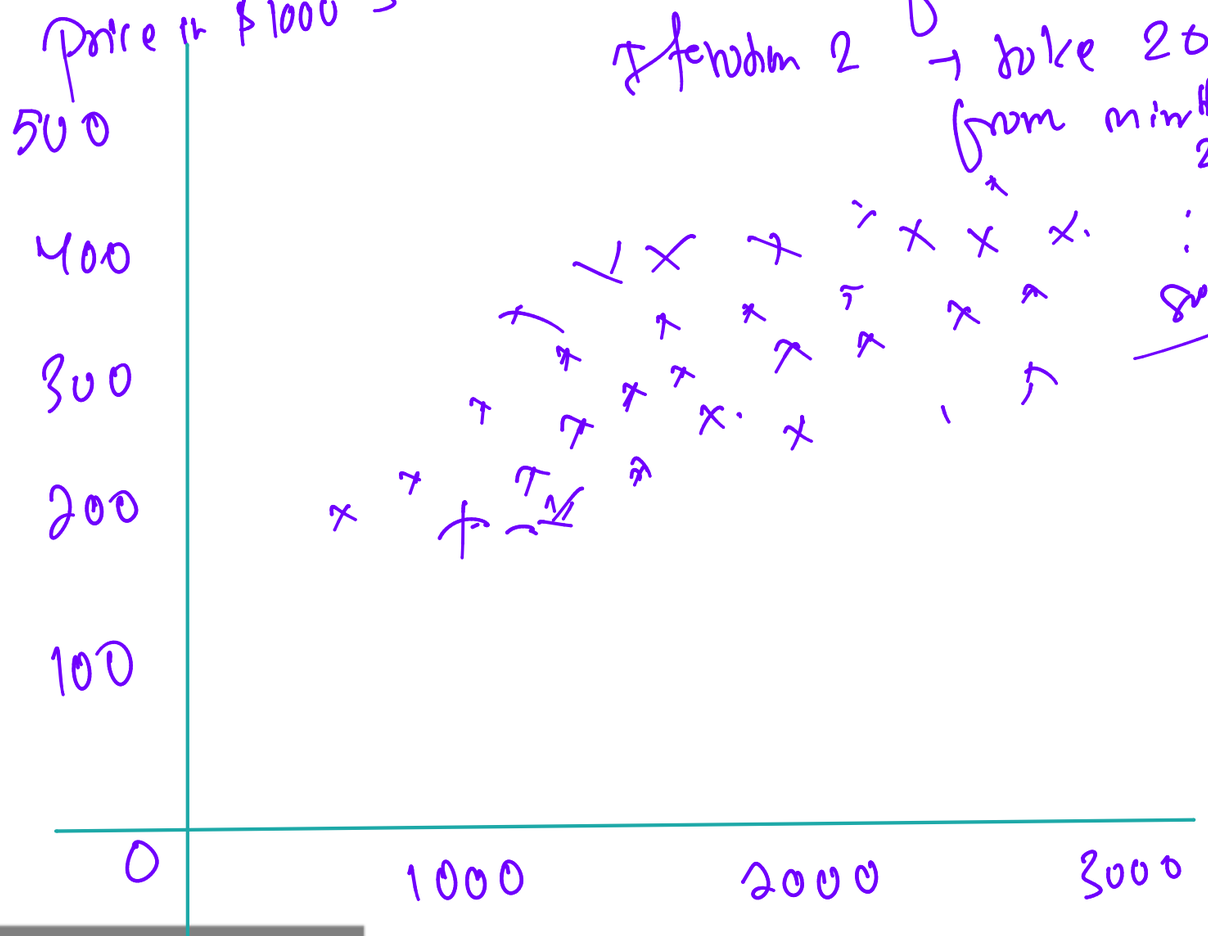
Mini-batch

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

mini-batch 1

mini-batch 2

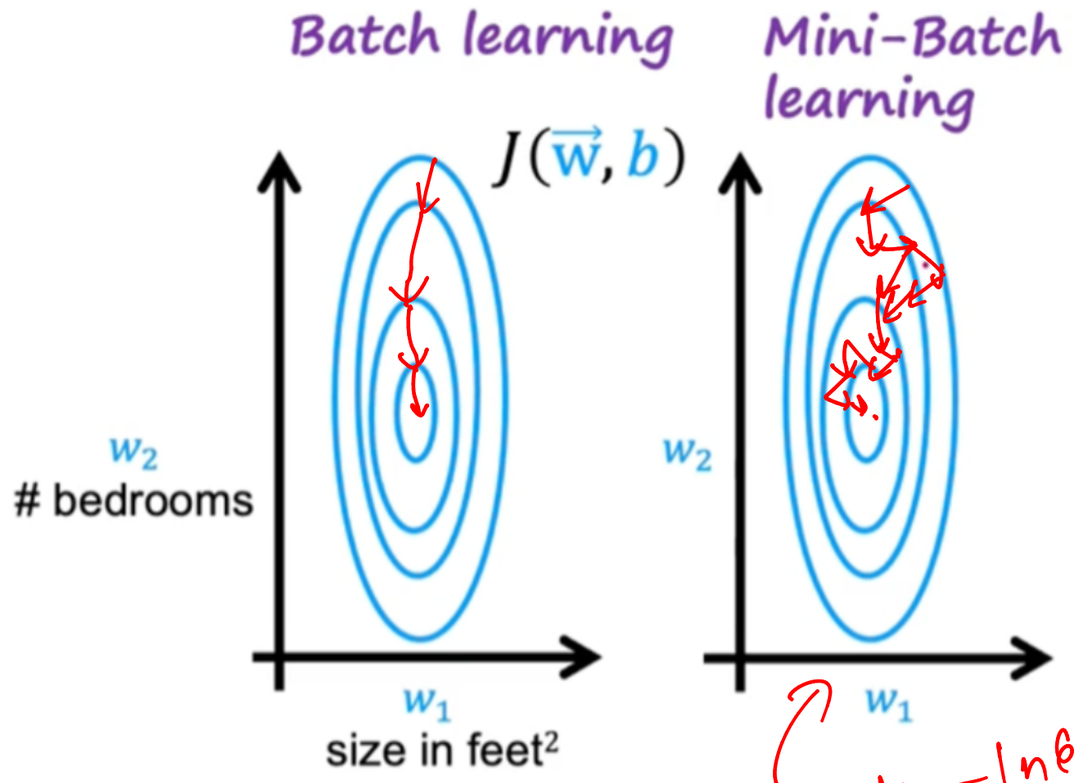
mini-batch 3



so that every iteration
is looking at

Mini-batch

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



Computation – Inexpensive
But may not find Global min as Batch Gradient.

You might remember with batch gradient descent,

DeepLearning.AI Stanford ONLINE Andrew Ng

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.

Train model:

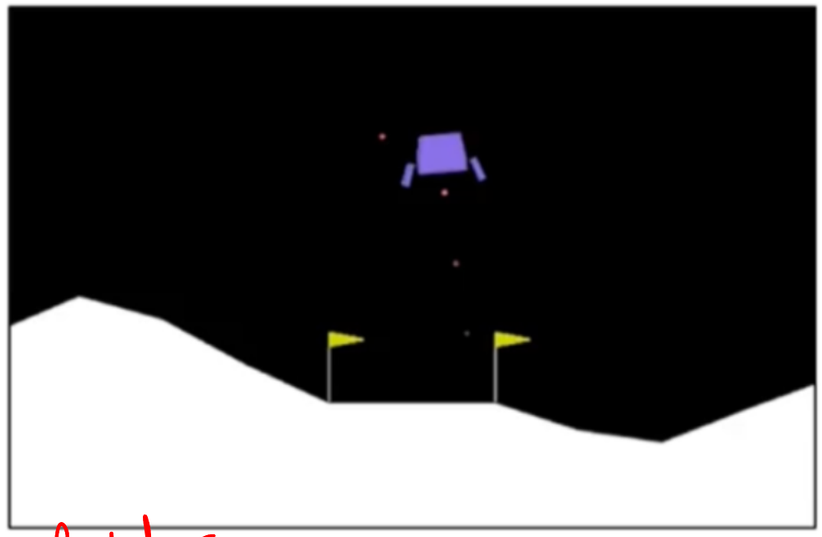
Create training set of 10,000 examples using

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

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

Set $Q = Q_{new}$.

}



recent tuples in the replay buffer,

Helps to Converge faster

Soft Update

↳ Accept only a little bit value

Set $Q = Q_{new}$.

W, B ← Q

W_{new}, B_{new} ← Q_{new}

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

$B = B_{new}$

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

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

$W = \gamma W_{new} + (1-\gamma) W$

$Q(s,a)$ → works well applying Soft update

For Lunar Simulation

When you train the new neural network,

DeepLearning.AI Stanford ONLINE Andrew Ng

Stats of RL

Limitations of Reinforcement Learning

- Much easier to get work in a simulation than a real robot!
- for fewer applications than SL and USL. (Unsupervised Learning)
- But .. exciting Research direction with potential for future applications.

RL is sometimes Hype but still long way to go (2025-03-03)

One of the reasons for some of the hype about reinforcement learning is, it turns

DeepLearning.AI Stanford ONLINE Andrew Ng

This lecture introduces **two key refinements** to the reinforcement learning algorithm for **Deep Q-Networks (DQN)**:

1. **Mini-batch Gradient Descent** – Speeds up training by using a subset of data.
2. **Soft Updates** – Stabilizes learning by gradually updating the Q-network.

---

1. Mini-Batch Gradient Descent in RL

Why Mini-Batches?

- **Batch Gradient Descent**: Computes gradients using **all** training examples.
- **Issue**: If the dataset is very large (e.g., 100M examples), every step is computationally expensive.
- **Mini-Batch Gradient Descent**: Uses a subset m' of training examples (e.g., 1,000).
  - Each step is much **faster** and updates the model with less computation.
  - Works well for **supervised learning** (e.g., linear regression, neural networks) and **reinforcement learning**.

How It Works in Reinforcement Learning?

- Store **10,000 tuples** $(s, a, R(s), s')$ in a **Replay Buffer**.
- Instead of training on all **10,000** experiences, use a mini-batch of **1,000** samples.
- Update the Q-function using:

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

- This introduces some noise but significantly **speeds up learning**.

---

2. Soft Updates for Stable Learning

Why Soft Updates?

- In the original **Q-update step**, we set:

$$Q \leftarrow Q_{\text{new}}$$

- If the new network Q_{new} is **worse** due to a bad update, it can degrade learning.
- Instead, **soft updates** gradually adjust Q using:

$$W \leftarrow \tau W_{\text{new}} + (1 - \tau) W$$

$$B \leftarrow \tau B_{\text{new}} + (1 - \tau) B$$

where τ is a small value (e.g., 0.01).

Effect of Soft Updates

- Reduces **oscillations** and prevents instability.
- Allows **gradual improvements** instead of abrupt changes.
- Ensures **smooth convergence** in reinforcement learning.

Needs to be Reviewed

Conclusion

- **Mini-batch learning** improves speed by training on small subsets.
- **Soft updates** make training more stable.
- These refinements **enhance DQN performance** and help the **Lunar Lander** land safely! 🚀

2. Soft Updates for Stable Learning

Why Soft Updates?

- In the original **Q-update step**, we set:

1. Mini-Batch Gradient Descent in Reinforcement Learning

Problem with Full-Batch Gradient Descent

- In **supervised learning**, batch gradient descent updates parameters using all m training examples:

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

  - Where $h(x^{(i)}) = wx^{(i)} + b$.
  - The gradient descent updates are:

$$w := w - \alpha\frac{\partial J}{\partial w}$$

$$b := b - \alpha\frac{\partial J}{\partial b}$$

- **Issue**: For large datasets (e.g., $m = 100M$), computing gradients for all examples in every step is computationally expensive.

Mini-Batch Gradient Descent Solution

- Instead of using all m samples, pick a **mini-batch** of m' (e.g., 1,000).
- The gradient updates become:

$$J_{\text{mini}}(w,b) = \frac{1}{2m'}\sum_{i=1}^{m'}(h(x^{(i)}) - y^{(i)})^2$$

  - New update rules:

$$w := w - \alpha\frac{\partial J_{\text{mini}}}{\partial w}$$

$$b := b - \alpha\frac{\partial J_{\text{mini}}}{\partial b}$$

- **Advantage**: Faster iterations with some noise, but still converges to the minimum.

Mini-Batch Learning in Reinforcement Learning

- In **Deep Q-Networks (DQN)**, store past experiences (s, a, r, s') in a **Replay Buffer**.
- Instead of training on the entire buffer, sample a **mini-batch** of m' experiences.
- Use **Q-learning update rule** for each mini-batch sample:

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

  - Update the **Q-function** using gradient descent:

$$\theta := \theta - \alpha\nabla_\theta \sum_{i=1}^{m'}(y^i - Q(s^i, a^i; \theta))^2$$

  - Reduces computational cost while still improving the policy.

2. Soft Updates for Stable Learning

Problem with Direct Q-Function Updates

- Original **Q-learning update** directly replaces old weights:

$$W \leftarrow W_{\text{new}}, \quad B \leftarrow B_{\text{new}}$$

- **Issue**: If W_{new} is poorly trained, it may degrade performance.

Soft Update Solution

- Instead of full replacement, use a weighted update:

$$W \leftarrow \tau W_{\text{new}} + (1-\tau)W$$

$$B \leftarrow \tau B_{\text{new}} + (1-\tau)B$$

  - Where τ (e.g., 0.01) controls how much of the new parameters are used.
  - Ensures **smooth transitions** and prevents instability.

Acknowledgement: I want to thank Coursera (finorcial aid) for providing valuable materials.

Date of 100Days of Maths Completion: 2025/03/10 (2081/11/26 B.S)

Resources: ① B.Sc CSIT 1st and 2nd Semester (KEC publication Maths Books) 2078 B.S.

② Coursera - Deeplearning.AI Courses
(Statistics, Linear Algebra, Calculus, Machine Learning)

③ ChatGPT, DeepSeek, Claude, Gemini (Coding & Additional Information)

Completed Courses: Coursera → Mathematics for Machine Learning and Data Science (3 Courses)

Coursera → Machine Learning (3 Courses)

Comprehensive learning, Basic Strong Concepts, fundamental skills, Job works, and many more

More Info? Visit: github.com/Dilli822/100DaysOfMaths/