Continuous state space

Many robotic applications, like the lunar lander, operate in continuous state spaces, allowing for a wide range of positions rather than a limited set.

- For instance, a Mars rover can occupy any position along a line, represented by a continuous range of values (e.g., $0 - 6$ kilometers).

Examples of Continuous States

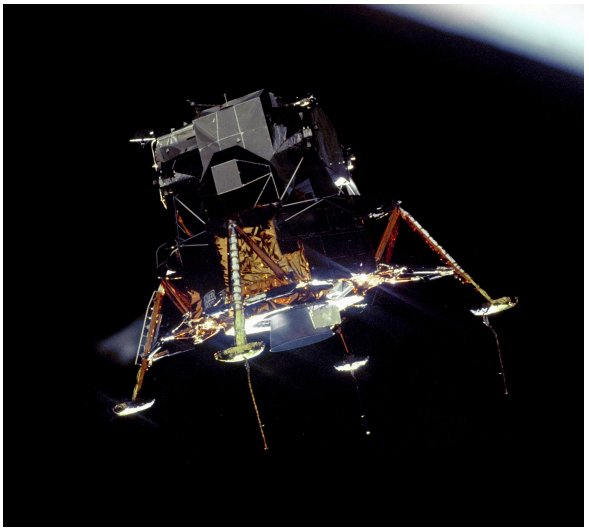
- In controlling a self-driving car or truck, the state includes multiple variables: x position, y position, orientation (θ), and speeds in both x and y directions.

$$s = \begin{bmatrix} x & y & \theta & \dot{x} & \dot{y} & \ddot{x} & \ddot{y} \end{bmatrix}^T$$

- For an autonomous helicopter, the state comprises x, y, z positions, orientation (roll, pitch, yaw), and their respective speeds, resulting in a vector of 12 numbers.

Lunar lander

- The simulation involves controlling a lunar lander that approaches the moon's surface, requiring the user to fire thrusters at the right times to land safely.
- Successful landings involve maneuvering between two flags on a designated landing pad, while crashes result from poor control
- The lander has four possible actions:
  - do nothing
  - fire the left thruster
  - fire the main engine
  - fire the right thruster.

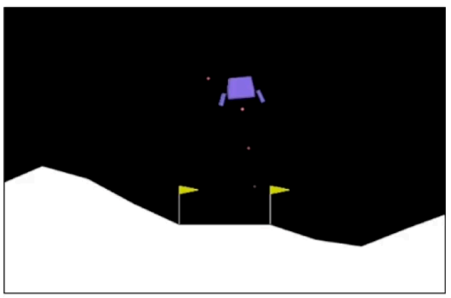


- The state space includes the lander's position (X and Y), velocity (horizontal and vertical), angle, angular velocity, and whether the left or right landing legs are grounded.

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

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

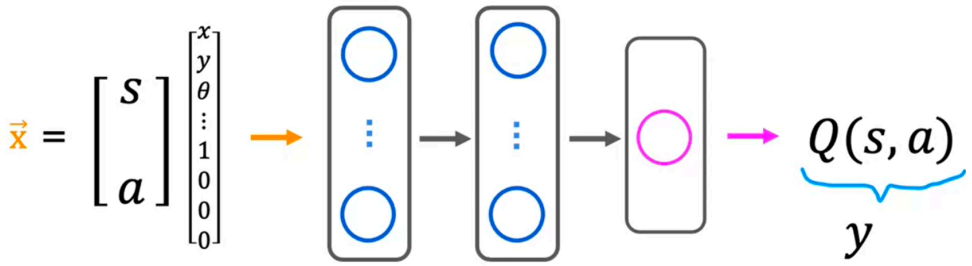


Lunar Lander Problem

Learn a policy π that, given: $s = \begin{bmatrix} x & y & \dot{x} & \dot{y} & \theta & \dot{\theta} & l & r \end{bmatrix}^T$
pick action $a = \pi(s)$ so as to maximize the return

Learning the state-value function

Deep Reinforcement Learning



A neural network estimates the expected return for each possible action in a given state and selects the one with the highest value. The network is trained to approximate the $Q - function$ by leveraging the Bellman equation, which generates a collection of training pairs (x, y). Using these pairs, supervised learning techniques are applied so the network learns the mapping from states and actions to their corresponding $Q - values$.

A neural network is trained to compute $Q(s, a)$ using the current state s and action a as input $\vec{x} = \begin{bmatrix} s \\ a \end{bmatrix}$.

- The input consists of 8 state variables and a one-hot encoding of 4 possible actions, totaling 12 input features, e.g, $\begin{bmatrix} x & y & \theta & \dots & 1 & 0 & 0 & 0 \end{bmatrix}^T$, $\begin{bmatrix} x & y & \theta & \dots & 0 & 1 & 0 & 0 \end{bmatrix}^T, \dots$

In a states, use neural network to compute: $Q(s, nothing),\ Q(s, left),\ Q(s, main),\ Q(s, right)$

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

This method is called **Deep Q-Network (DQN)**.

Bellman's Equation

- The Bellman equation is used to create a training set, where $Q(s, a)$ is defined in terms of rewards and future state values.
- The right-hand side of the equation provides the target value Y for the neural network to predict.

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

The objective of a neural network is to learn a function that can accurately produce the desired output.

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

To train the neural network it is necessary to build a training set with pairs (x, y) for the NN, the model begins by exploring different states and actions—sometimes randomly, sometimes in a guided way.

- Every time an action is chosen, the experience is recorded into the training set.
- Here, the combination of the state and action forms the input x, while the reward and the resulting next state are used to compute the target output y.

$$(\underbrace{s^{(i)}, a^{(i)}}_{x^{(i)}}, \underbrace{R(s^{(i)}), s'^{(i)}}_{y^{(i)}})$$

Therefore, with each trial $(s^{(i)}, a^{(i)}, R(s^{(i)}), s'^{(i)})$ we can calculate $x^{(i)}$ and $y^{(i)}$ using Bellman equation:

$$x^{(i)} = \begin{bmatrix} s^{(i)} \\ a^{(i)} \end{bmatrix}, \qquad y^{(i)} = R(s^{(i)}) + \gamma \max_{a'} Q(s'^{(i)}, a')$$

Learning algorithm

Algorithm 1 Training Q-function with Neural Network

1: Initialize neural network randomly as an estimate of $Q(s, a)$
2: **repeat**
3: Take actions in the environment (e.g., Lunar Lander) to obtain tuples $(s, a, R(s), s')$
4: Store the 10,000 most recent tuples $(s, a, R(s), s')$
5: Create training set of 10,000 examples with:
6: $x \leftarrow (s, a)$
7: $y \leftarrow R(s) + \gamma \max_{a'} Q(s', a')$
8: Train neural network Q_{new} such that $Q_{\text{new}}(s, a) \approx y$
9: Set $Q \leftarrow Q_{\text{new}}$
10: **until** convergence

In line 4, the recent tuples is usually called Relay buffer.

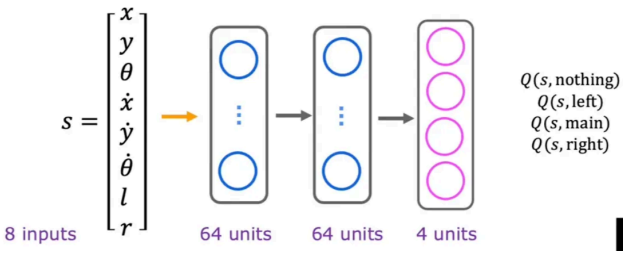
Algorithm refinement

Improved neural network architecture

The previous architecture required **separate inference** for each action, which was **inefficient**.

- To compute $\max_a Q(s, a)$, we would normally need to evaluate the network n times (where n is the total number of actions), once for each action, and then select the maximum.

> A more efficient approach is to redesign the network so that it takes only the state s as input and directly produces n outputs: $\{Q(s, a_1), Q(s, a_2), \ldots, Q(s, a_n)\}$. This modification greatly speeds up the inference process.



The new architecture allows the neural network to output Q values for all possible actions simultaneously.

In a state s, input s to neural network.
Pick the action a that maximizes $Q(s, a)$.

ϵ-greedy policy

Problem statement

Algorithm 1 Training Q-function with Neural Network

1: Initialize neural network randomly as an estimate of $Q(s, a)$
2: **repeat**
3: Take actions in the environment (e.g., Lunar Lander) to obtain tuples $(s, a, R(s), s')$
4: Store the 10,000 most recent tuples $(s, a, R(s), s')$
5: Create training set of 10,000 examples with:
6: $x \leftarrow (s, a)$
7: $y \leftarrow R(s) + \gamma \max_{a'} Q(s', a')$
8: Train neural network Q_{new} such that $Q_{\text{new}}(s, a) \approx y$
9: Set $Q \leftarrow Q_{\text{new}}$
10: **until** convergence

Look at line 3 of the learning algorithm pseudocode, we need to **"take action"**

⇒ The question is **"How to take the good action?"**

How to choose actions while still learning?

In some state s, with $\epsilon = 0.05$:

1. **Option** 1**:**
   - Pick the action a that maximizes $Q(s, a)$. (It is not good)

   Why not always pick the best action?
   - Early in training, Q-values may be inaccurate.
   - If we always exploit, the agent may get stuck in a **local optimum**.
   - Occasional exploration allows the agent to discover better actions and improve its estimates.

2. **Option** 2**:**
   - With probability $1 - \epsilon = 0.95$, pick the action a that **maximizes** $Q(s, a)$. **Greedy** (**Exploitation**)
   - With probability $\epsilon = 0.05$, pick an action a **randomly**. (**Exploration**)

- Exploration involves trying out different actions to gather more information, while exploitation focuses on maximizing returns based on current knowledge.
- The trade-off between these two strategies is crucial for effective learning in reinforcement learning algorithms.

This randomness helps the learning algorithm discover potentially beneficial actions that may initially seem suboptimal.

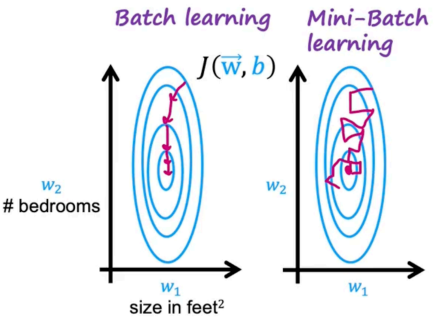
Adjusting Epsilon

- It is common to start with a high ϵ value (e.g., 1.0) to encourage exploration and gradually decrease it to promote exploitation as the learning progresses.
- Proper tuning of ϵ and other hyperparameters is essential, as reinforcement learning can be sensitive to these choices compared to supervised learning.

Mini-batch

Mini-batch gradient descent improves efficiency by using a smaller subset of training examples (e.g., 1,000) instead of the entire dataset (e.g., 100 million) for each iteration, speeding up the training process.

This method allows for quicker iterations while still tending toward the global minimum of the cost function, making it more suitable for large datasets.



Algorithm 1 Training Q-function with Neural Network with Mini-Batch

Initialize neural network randomly as an estimate of $Q(s, a)$
repeat
  Take actions in the environment (e.g., Lunar Lander) to obtain tuples $(s, a, R(s), s')$
  Store the 10,000 most recent tuples $(s, a, R(s), s')$
  Create training set of 1,000 (instead of 10,000) examples with:
    $x \leftarrow (s, a)$
    $y \leftarrow R(s) + \gamma \max_{a'} Q(s', a')$
  Train neural network Q_{new} such that $Q_{\text{new}}(s, a) \approx y$
  $Q \leftarrow Q_{\text{new}}$
until convergence

1. Store experiences in a **replay buffer** (a memory).
2. When updating, **sample a random mini-batch** of experiences from this buffer.

Soft Updates

It is like gradually blending in new information with what you already know, instead of completely replacing it at once.

- Soft updates prevent abrupt changes to the Q function by gradually incorporating new neural network parameters, using a weighted average (e.g., $1 - p = 99\%$ old parameters and $p = 1\%$ new).

$$w = p \cdot w_{new} + (1 - p) \cdot w$$
$$b = p \cdot b_{new} + (1 - p) \cdot b$$

> This approach **enhances the convergence of the reinforcement learning algorithm**, **reducing** the **likelihood** of oscillations or undesirable behaviors during training.

This percentage is usually quite small (around 1%), and it controls how strongly the updates are applied. For instance, if $w = 1 \cdot w_{\text{new}} + 0 \cdot w_{\text{old}}$, the result is **identical** to simply using the new function $Q = Q_{\text{new}}$, since all the weight is assigned to the updated parameters.

Limitations of Reinforcement Learning

- Much easier to get to work in a simulation than a real robot!
- Far fewer applications than supervised and unsupervised
- But ... exciting research direction with potential for future applications.

The Lunar Lander is a continuous state Markov Decision Process (MDP) because:

- The state contains numbers such as position and velocity that are continuous valued. 👍👍
- The state has multiple numbers rather than only a single number (such as position in the x-direction)
- The reward contains numbers that are continuous valued
- The state-action value $Q(s, a)$ function outputs continuous valued numbers

In the learning algorithm described in the videos, we repeatedly create an artificial training set to which we apply supervised learning where the input $x = (s, a)$ and the target, constructed using Bellman's equations, is $y =$ ____?

- $y = R(s)$
- $y = \max_{a'} Q(s', a')$ where s' is the state you get to after taking action a in state s
- $y = R(s) + \gamma \max_{a'} Q(s', a')$ where s' is the state you get to after taking action a in state s 👍
- $y = R(s')$ where s' is the state you get to after taking action a in state s