




# Robot Path Planning

Nguyen Van Luong 20173249

# Robot Path Planning

- PSO
- Q-Learning
- PRM + Dijkstra
- TD3



# PSO - Particles Swarm Optimization

---

# What ?

- Inventors: James Kennedy & Russell Eberhart // IEEE – 1995.
- Optimization of nonlinear functions using particle swarm methodology.
- Based on Swarm Intelligence.
- The movement of organisms in a bird flock or fish school.
- Main elements:
  - Global Best
  - Particle Best

# Why ?

- PSO is a metaheuristic – few assumptions about the problem
- Search very large spaces of candidate solutions
- Not Gradient
- Bioinspired algorithms
- Swarm Intelligence
- Simple formulae

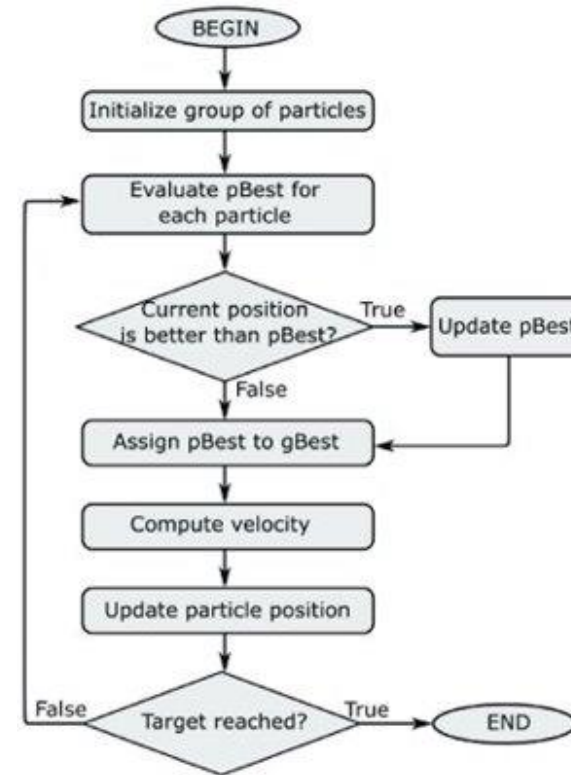
# How ?

- Swarm:
  - Fitness
  - Global Best
- Particles:
  - Position
  - Velocity
  - Particle Best



# How ?

- Initialize Swarm
- Loop:
  - Check + update pBest / gBest
  - Compute velocity
  - Update position
  - Check Target



# Formulae

- Velocity:

The diagram illustrates the velocity update formula for a particle  $i$  at time  $k+1$ . The formula is 
$$v_{k+1}^i = w v_k^i + c_1 \text{rand} \frac{(p^i - x_k^i)}{\Delta t} + c_2 \text{rand} \frac{(p_k^g - x_k^i)}{\Delta t}$$
 Annotations include: 

- An arrow from "velocity of particle  $i$  at time  $k+1$ " points to  $v_{k+1}^i$ .
- A bracket under  $w v_k^i$  is labeled "current motion". An arrow from "inertia factor range: 0.4 to 1.4" points to  $w$ .
- A bracket under  $c_1 \text{rand} \frac{(p^i - x_k^i)}{\Delta t}$  is labeled "particle memory influence". An arrow from "self confidence range: 1.5 to 2" points to  $c_1$ .
- A bracket under  $c_2 \text{rand} \frac{(p_k^g - x_k^i)}{\Delta t}$  is labeled "swarm influence". An arrow from "swarm confidence range: 2 to 2.5" points to  $c_2$ .

- Position:

$$X_i^{t+1} = X_i^t + V_i^{t+1}$$

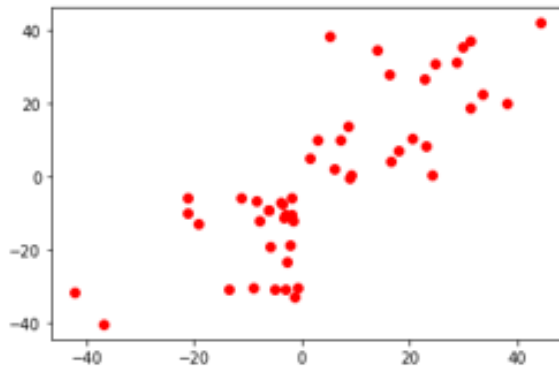


# Pseudocode

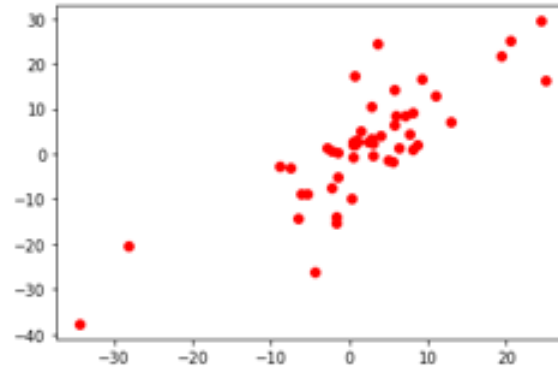
- $p = \text{particle\_initialize}()$
- for  $i = 1$  to  $it\_max$ :
  - for each particle  $p$  in  $P$ :
    - $fp = f(p)$
    - if  $fp$  is better than  $f(pBest)$ :
      - $pBest = p$
  - end
- end
- $gBest = \text{best } p \text{ in } P$
- for each particle  $p$  in  $P$ :
  - $v = v + c1 * r1 * (pBest - p) + c2 * r2 * (gBest - p)$
  - $p = p + v$
- end
- end

# Simulation

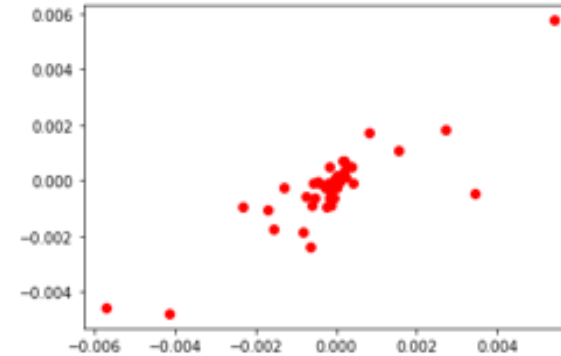
- Problem:  $\min f = x^2 + y^2 + 1$



1st iterations



2nd iterations



32nd iterations

# Implementation

- Youtube: [https://www.youtube.com/watch?v=vNIX31QtQOw&ab\\_channel=LuongNguyen](https://www.youtube.com/watch?v=vNIX31QtQOw&ab_channel=LuongNguyen)
- Colab: [https://drive.google.com/file/d/1fCzRZdsgH\\_z\\_61bJuHFpzSz-\\_HMSYope/view?usp=sharing](https://drive.google.com/file/d/1fCzRZdsgH_z_61bJuHFpzSz-_HMSYope/view?usp=sharing)

# References

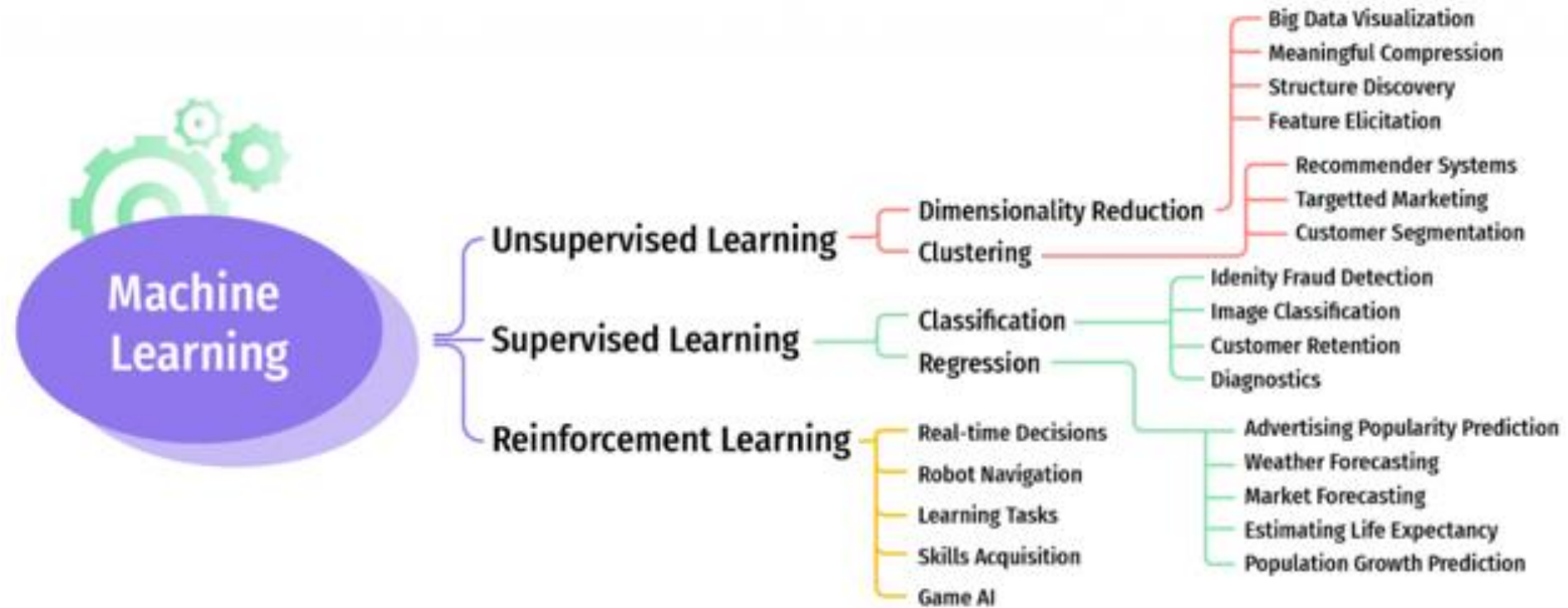
- *Kennedy, J.; Eberhart, R. (1995). "Particle Swarm Optimization". Proceedings of IEEE International Conference on Neural Networks. IV. pp. 1942–1948.*



# Reinforcement Learning

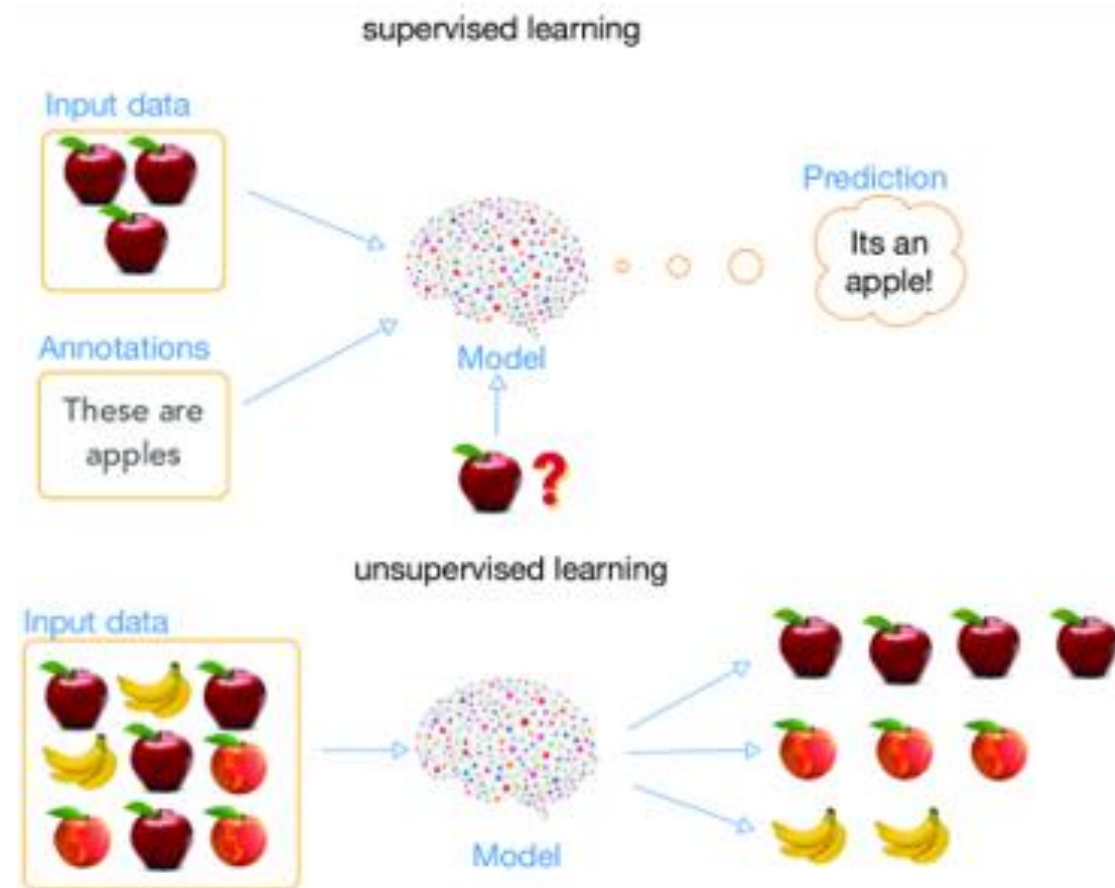
---

# Machine Learning



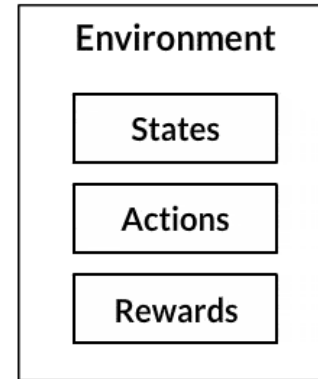


# Supervised Learning & Unsupervised Learning



# What ?

- Machine Learning technique
- Experiences
- Maximize reward
- AlphaGo



# Terminologies

- **Agent** — the learner and the decision maker.
- **Environment** — where the agent learns and decides what actions to perform.
- **Action** — a set of actions which the agent can perform.
- **State** — the state of the agent in the environment.
- **Reward** — for each action selected by the agent the environment provides a reward. Usually a scalar value.
- **Policy** — the decision-making function (control strategy) of the agent, which represents a mapping from situations to actions.
- **Value function** — mapping from states to real numbers, where the value of a state represents the long-term reward achieved starting from that state and executing a particular policy.
- **Function approximator** — refers to the problem of inducing a function from training examples. Standard approximators include decision trees, neural networks, and nearest-neighbor methods

# Terminologies

- **Markov decision process (MDP)** — A probabilistic model of a sequential decision problem, where states can be perceived exactly, and the current state and action selected determine a probability distribution on future states. Essentially, the outcome of applying an action to a state depends only on the current action and state (and not on preceding actions or states).
- **Dynamic programming (DP)** — is a class of solution methods for solving sequential decision problems with a compositional cost structure. Richard Bellman was one of the principal founders of this approach.
- **Monte Carlo methods** — A class of methods for learning of value functions, which estimates the value of a state by running many trials starting at that state, then averages the total rewards received on those trials.
- **Temporal Difference (TD) algorithms** — A class of learning methods, based on the idea of comparing temporally successive predictions. Possibly the single most fundamental idea in all of reinforcement learning.
- **Model** — The agent's view of the environment, which maps state-action pairs to probability distributions over states. Note that not every reinforcement learning agent uses a model of its environment

# Analogy

- Teaching a dog
  - Agent: dog // don't understand language
  - Environment: house
  - State: standing
  - Action: sit
  - Reward: snacks
- Note:
  - Being greedy doesn't always work
  - Sequence matters in Reinforcement Learning

# Why ?

- Generality
- Not needing labeled input/output
- Finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge)
- No mathematical model of the environment



# Where ?

- Many disciplines
  - Game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, and statistics
- AI games (AlphaGo, ATARI games, Backgammon, ...)
- In robotics and industrial automation
  - RL is used to enable the robot to create an efficient adaptive control system for itself which learns from its own experience and behavior.
- In economics and game theory
  - RL may be used to explain how equilibrium may arise under bounded rationality.
- Text summarization engines, dialog agents (text, speech)
  - which can learn from user interactions and improve with time, learning optimal treatment policies in healthcare and RL based agents for online stock trading.

# How ?

1. Observation of the environment
2. Deciding how to act using some strategy
3. Acting accordingly
4. Receiving a reward or penalty
5. Learning from the experiences and refining our strategy
6. Iterate until an optimal strategy is found

# Algorithms

Algorithm	Description	Model	Policy	Action Space	State Space	Operator
<a href="#">Monte Carlo</a>	Every visit to Monte Carlo	<a href="#">Model-Free</a>	Either	Discrete	Discrete	Sample-means
<a href="#">Q-learning</a>	State-action-reward-state	Model-Free	Off-policy	Discrete	Discrete	Q-value
<a href="#">SARSA</a>	State-action-reward-state-action	Model-Free	On-policy	Discrete	Discrete	Q-value
<a href="#">DQN</a>	Deep Q Network	Model-Free	Off-policy	Discrete	Continuous	Q-value
DDPG	Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
A3C	Asynchronous Advantage Actor-Critic Algorithm	Model-Free	On-policy	Continuous	Continuous	Advantage
TRPO	Trust Region Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
<a href="#">PPO</a>	Proximal Policy Optimization	Model-Free	On-policy	Continuous	Continuous	Advantage
TD3	Twin Delayed Deep Deterministic Policy Gradient	Model-Free	Off-policy	Continuous	Continuous	Q-value
SAC	Soft Actor-Critic	Model-Free	Off-policy	Continuous	Continuous	Advantage

# References

- Kaelbling, Leslie P.; Littman, Michael L.; Moore, Andrew W. (1996). "Reinforcement Learning: A Survey". *Journal of Artificial Intelligence Research*. 4: 237–285
- Robert Moni (2019). "Reinforcement Learning algorithms — an intuitive overview". *SmartLab AI*.



# Q-Learning

---

# Self-Driving Cab

- Pick up the passenger at one location and drop them off in another
  - Drop off the passenger to the right location.
  - Save passenger's time by taking minimum time possible to drop off
  - Take care of passenger's safety and traffic rules



# Self-Driving Cab

- 1. Reward
  - The agent should receive a high positive reward for a successful drop-off because this behavior is highly desired
  - The agent should be penalized if it tries to drop off a passenger in wrong locations
  - The agent should get a slight negative reward for not making it to the destination after every time-step.

# Self-Driving Cab

- 2. State space
  - 5x5 grid, which gives us 25 possible taxi locations
  - R, G, Y, B or [(0,0), (0,4), (4,0), (4,3)]
  - (4) destinations and (4 + 1) passenger locations.
  - So, our taxi environment has  $5 \times 5 \times 5 \times 4 = 5 \times 5 \times 5 \times 4 = 500$  total possible states.



# Q-Learning

- Essentially, Q-learning lets the agent use the environment's rewards to learn, over time, the best action to take in a given state.
- Receiving a reward for taking an action, then updating a *Q-value*.
- Q-value = (state, action)

$$Q(\text{state}, \text{action}) \leftarrow (1 - \alpha)Q(\text{state}, \text{action}) + \alpha \left( \text{reward} + \gamma \max_a Q(\text{next state}, \text{all actions}) \right)$$

- Where:
  - $\alpha$  (alpha) is the learning rate ( $0 < \alpha \leq 1$ ) - Just like in supervised learning settings,  $\alpha$  is the extent to which our Q-values are being updated in every iteration.
  - $\gamma$  (gamma) is the discount factor ( $0 \leq \gamma \leq 1$ ) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to **1**) captures the long-term effective award, whereas a discount factor of **0** makes our agent consider only immediate reward, hence making it greedy.

# Q-Table

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
	327	0	0	0	0	0	0
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
499	-	0	0	0	0	0	0
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
	328	-2.30808105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
	-	-	-	-	-	-	-
499	-	9.96984239	4.02706992	12.96022777	29	3.328077873	3.382306603
	499	9.96984239	4.02706992	12.96022777	29	3.328077873	3.382306603

- *Q-Table values are initialized to zero and then updated during training to values that optimize the agent's traversal through the environment for maximum rewards*

# Q-Learning Processing

1. Initialize the Q-table by all zeros.
2. Start exploring actions: For each state, select any one among all possible actions for the current state ( $S$ ).
3. Travel to the next state ( $S'$ ) as a result of that action ( $a$ ).
4. For all possible actions from the state ( $S'$ ) select the one with the highest Q-value.
5. Update Q-table values using the equation.
6. Set the next state as the current state.
7. If goal state is reached, then end and repeat the process.

# Exploiting learned values

- After enough random exploration of actions, the Q-values tend to converge serving our agent as an action-value function which it can exploit to pick the most optimal action from a given state.
- There's a tradeoff between exploration (choosing a random action) and exploitation (choosing actions based on already learned Q-values). We want to prevent the action from always taking the same route, and possibly overfitting, so we'll be introducing another parameter called  $\epsilon$  "epsilon" to cater to this during training.
- Instead of just selecting the best learned Q-value action, we'll sometimes favor exploring the action space further. Lower epsilon value results in episodes with more penalties (on average) which is obvious because we are exploring and making random decisions.



# Implementation

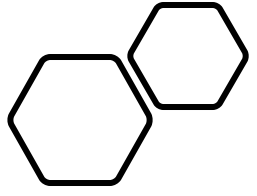
- Kaggle: <https://www.kaggle.com/greyng/basic-q-learning-self-driving-cab>

```
+-----+
|R:  |  :  :G|
|  :  :  :  |
|  :  :  :  |
|  :  :  :  |
|  |  :  |  :  |
|Y|  :  |B:  |
+-----+
      (Dropoff)
```

```
Timestep: 1
State: 328
Action: 5
Reward: -10
```

# References

- Brendan M & Satwik K. (2020) "Reinforcement Q-Learning with OpenAI Gym".  
Learndatasci.



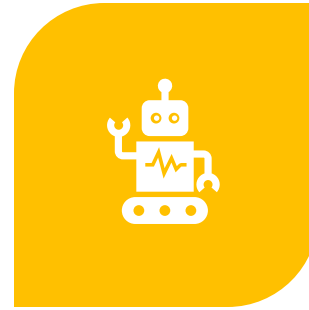
# Robot Path Planning



PROBLEMS



PATH PLANNING



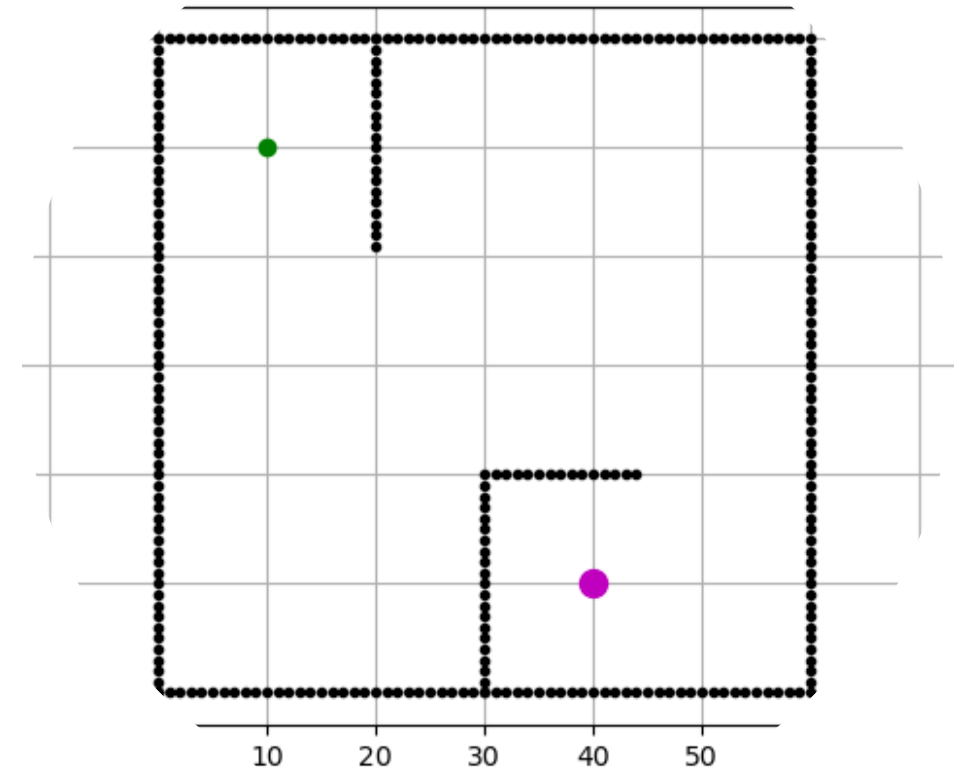
ALGORITHMS



RESULTS

# Problems

- Determining a path between a starting configuration of the robot and a goal configuration
- Avoiding collisions



# Path Planning

- Probabilistic Road-Map (PRM) planning
  - Description
    - The probabilistic roadmap planner is a motion planning algorithm in robotics
    - The basic idea behind PRM is to take random samples from the configuration space of the robot, testing them for whether they are in the free space, and use a local planner to attempt to connect these configurations to other nearby configurations.

# Path Planning

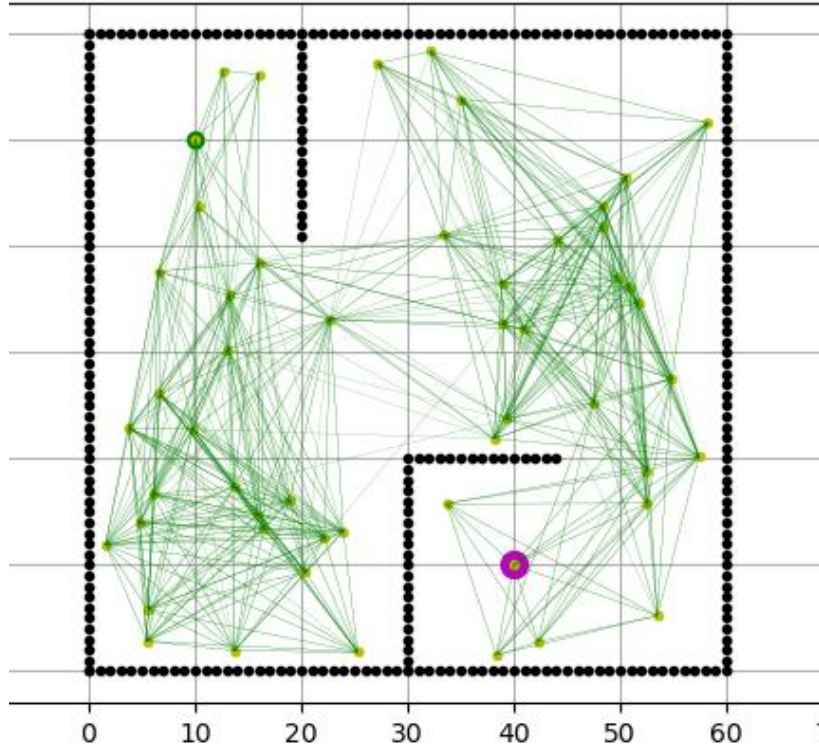
- PRM
  - Steps
    1. Initially empty Graph  $G$
    2. A configuration  $q$  is randomly chosen
    3. If  $q \rightarrow Q_{\text{free}}$  then added to  $G$  (collision detection
    4. needed here)
    5. Repeat until  $N$  vertices chosen
    6. For each  $q$ , select  $k$  closest neighbors
    7. Local planner  $\Delta$  connects  $q$  to neighbor  $q'$
    8. If connect successful (i.e. collision free local path), add edge  $(q, q')$

# Path Planning

- PRM
  - Pseudocode

```
1:  $V \leftarrow \emptyset$ 
2:  $E \leftarrow \emptyset$ 
3: while  $|V| < n$  do
4:   repeat
5:      $q \leftarrow$  a random configuration in  $\mathcal{Q}$ 
6:   until  $q$  is collision-free
7:    $V \leftarrow V \cup \{q\}$ 
8: end while
9: for all  $q \in V$  do
10:   $N_q \leftarrow$  the  $k$  closest neighbors of  $q$  chosen from  $V$  according to dist
11:  for all  $q' \in N_q$  do
12:    if  $(q, q') \notin E$  and  $\Delta(q, q') \neq \text{NIL}$  then
13:       $E \leftarrow E \cup \{(q, q')\}$ 
14:    end if
15:  end for
16: end for
```

---



# Path Planning

- PRM
  - Implementation



# Algorithms

- Dijkstra's Algorithm
  - Description
    - Dijkstra's algorithm to find the shortest path between  $a$  and  $b$ . It picks the unvisited vertex with the lowest distance, calculates the distance through it to each unvisited neighbor, and updates the neighbor's distance if smaller. Mark visited when done with neighbors.

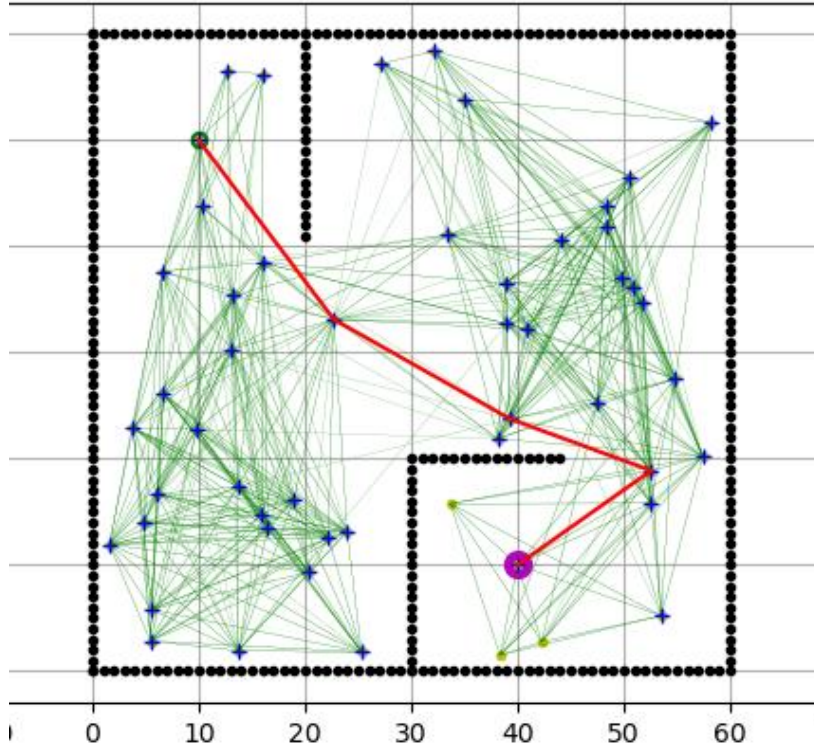
# Algorithms

- Dijkstra's Algorithm
  - Steps
    1. Label the start node with zero and box this label.
    2. Consider the node with the most recently boxed label. Suppose this node to be X and let D be its permanent label. Then, in turn, consider each node directly joined to X but not yet permanently boxed. For each such node, Y say, temporarily label it with the lesser of  $D + (\text{the weight of arc } XY)$  and its existing label (if any).
    3. Choose the least of all temporary labels on the network. Make this label permanent by boxing it.
    4. Repeat Steps 2 and 3 until the destination node has a permanent label.
    5. Go backwards through the network, retracing the path of shortest length from the destination node to the start node.

# Algorithms

- Dijkstra's Algorithm
  - Pseudocode

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:
6          dist[v] ← INFINITY
7          prev[v] ← UNDEFINED
8          add v to Q
9      dist[source] ← 0
10
11     while Q is not empty:
12         u ← vertex in Q with min dist[u]
13
14         remove u from Q
15
16         for each neighbor v of u:           // only v that are still in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]:
19                 dist[v] ← alt
20                 prev[v] ← u
21
22     return dist[], prev[]
```



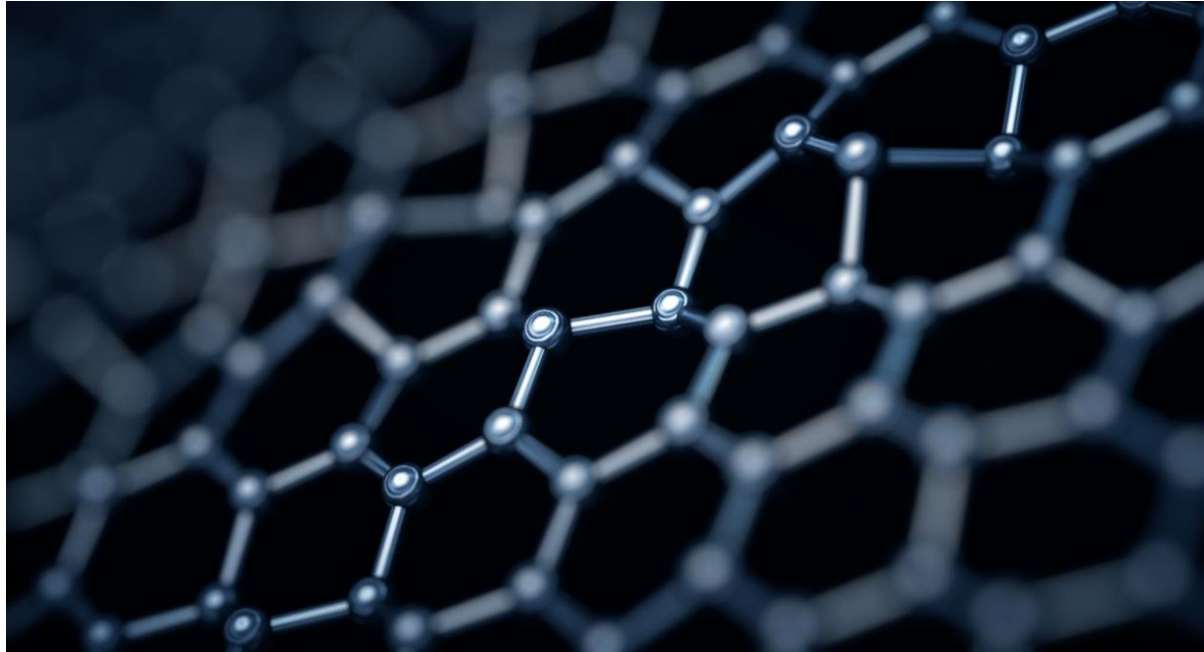
# Algorithms

- Dijkstra's Algorithm
  - Implementation

# References

- Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; Overmars, M. H. (1996), "Probabilistic roadmaps for path planning in high-dimensional configuration spaces", *IEEE Transactions on Robotics and Automation*, 12 (4): 566–580.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Section 24.3: Dijkstra's algorithm". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw–Hill. pp. 595–601.





Deep Reinforcement Learning  
for Mobile Robot Path Planning in 2D Environments

# Deep Reinforcement Learning for Robot Path Planning



Problem



Algorithm



Result



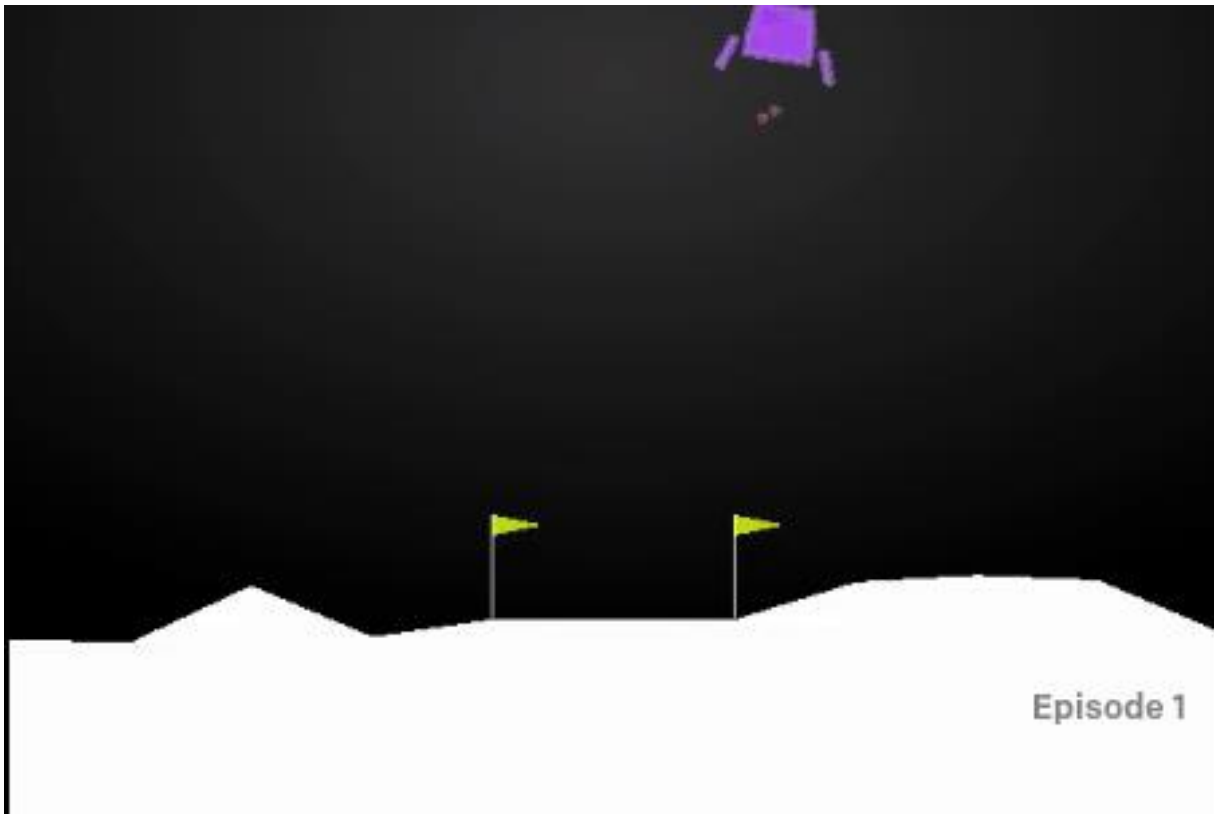
Simulation



References



# Problem



- Lunar Lander Continuous
  - The Lunar Lander example is an example available in the OpenAI Gym (Discrete) and OpenAI Gym (Continuous) where the goal is to land a Lunar Lander as close between 2 flag poles as possible, making sure that both side boosters are touching the ground.
  - We can land this Lunar Lander by utilizing actions and will get a reward in return - as is normal in Reinforcement Learning.

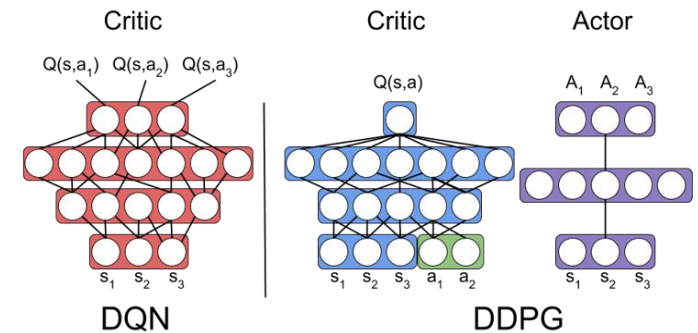
- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>• Observation Space:             <ul style="list-style-type: none"> <li>• The observation space is illustrated by a "Box" containing 8 values between <math>[-\infty, \infty]</math> these values are:                 <ul style="list-style-type: none"> <li>• Position X</li> <li>• Position Y</li> <li>• Velocity X</li> <li>• Velocity Y</li> <li>• Angle</li> <li>• Angular Velocity</li> <li>• Is left leg touching the ground: 0 OR 1</li> <li>• Is right leg touching the ground: 0 OR 1</li> </ul> </li> </ul> </li> </ul>   | <ul style="list-style-type: none"> <li>• Action Space:             <ul style="list-style-type: none"> <li>• Discrete (Discrete Action Space with 4 values):                 <ul style="list-style-type: none"> <li>• 0 = Do Nothing</li> <li>• 1 = Fire Left Engine</li> <li>• 2 = Fire Main Engine</li> <li>• 3 = Fire Right Engine</li> </ul> </li> <li>• Continuous (Box Action Space with 2 values between -1 and +1):                 <ul style="list-style-type: none"> <li>• Value 1: <math>[-1.0, +1.0]</math> for main engine where <math>[-1.0, 0.0]</math> = Off and <math>[0.0, +1.0]</math> = On</li> <li>• Value 2:                     <ul style="list-style-type: none"> <li>• <math>[-1.0, -0.5]</math>: Left Engine</li> <li>• <math>[-0.5, 0.5]</math>: Off</li> <li>• <math>[0.5, 1.0]</math>: Right Engine</li> </ul> </li> </ul> </li> </ul> </li> </ul> |
| <ul style="list-style-type: none"> <li>• Reward Function:             <ul style="list-style-type: none"> <li>• The Reward Function is a bit more complex and consists out of multiple components:                 <ul style="list-style-type: none"> <li>• <math>[100, 140]</math> points for Moving to the landing pad and zero speed</li> <li>• Negative reward for moving away from the landing pad</li> <li>• If lander crashes or comes to rest it gets -100 or +100</li> <li>• Each leg with ground contact gets +10</li> <li>• Firing the main engine is -0.3 per frame</li> <li>• Firing the side engine is -0.03 per frame</li> <li>• Solved is 200 points</li> </ul> </li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>• Constraint             <ul style="list-style-type: none"> <li>• Timesteps</li> <li>• Map</li> <li>• Avoid collision</li> </ul> </li> <li>• Scope             <ul style="list-style-type: none"> <li>• Maximum Reward</li> </ul> </li> </ul>   |

# Algorithm

- TD3 Algorithms

- Description:

- Twin Delayed Deep Deterministic policy gradient -TD3
    - TD3 is the successor to the Deep Deterministic Policy Gradient (DDPG)(Lillicrap et al, 2016). Up until recently, DDPG was one of the most used algorithms for continuous control problems such as robotics and autonomous driving.
    - Although DDPG is capable of providing excellent results, it has its drawbacks. Like many RL algorithms training DDPG can be unstable and heavily reliant on finding the correct hyper parameters for the current task (OpenAI [Spinning Up](#), 2018).
    - This is caused by the algorithm continuously over estimating the Q values of the critic (value) network. These estimation errors build up over time and can lead to the agent falling into a local optima or experience catastrophic forgetting.
    - TD3 addresses this issue by focusing on reducing the overestimation bias seen in previous algorithms. This is done with the addition of 3 key features:
      1. Using a pair of critic networks
      2. Delayed updates of the actor
      3. Action noise regularization



# Algorithm

- Pseudo code [1]

1. Initialize networks
2. Initialize replay buffer
3. Select and carry out action with exploration noise
4. Store transitions
5. Update critic
6. Update actor
7. Update target networks
8. Repeat until sentient

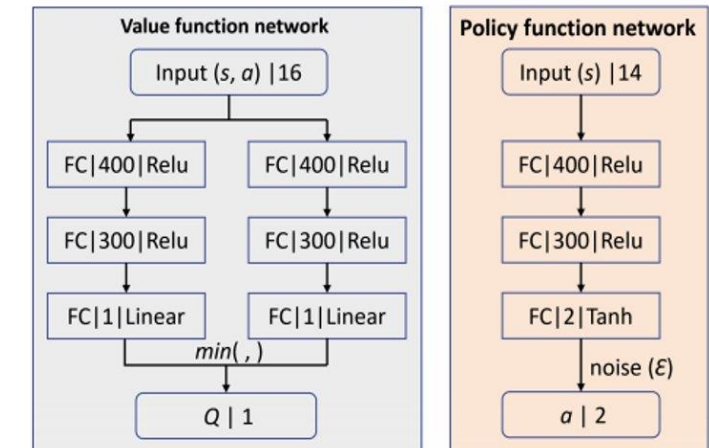
## Algorithm 1 TD3

```

1 Initialize critic networks  $Q_{\theta_1}, Q_{\theta_2}$ , and actor network  $\pi_\phi$ 
  with random parameters  $\theta_1, \theta_2, \phi$ 
2 Initialize target networks  $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$ 
  Initialize replay buffer  $\mathcal{B}$ 
  for  $t = 1$  to  $T$  do
3   Select action with exploration noise  $a \sim \pi_\phi(s) + \epsilon$ ,
    $\epsilon \sim \mathcal{N}(0, \sigma)$  and observe reward  $r$  and new state  $s'$ 
4   Store transition tuple  $(s, a, r, s')$  in  $\mathcal{B}$ 

5   Sample mini-batch of  $N$  transitions  $(s, a, r, s')$  from  $\mathcal{B}$ 
    $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon$ ,  $\epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$ 
    $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$ 
   Update critics  $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$ 
6   if  $t \bmod d$  then
   Update  $\phi$  by the deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$ 
7   Update target networks:
    $\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$ 
    $\phi' \leftarrow \tau \phi + (1 - \tau) \phi'$ 
   end if
  end for
  
```

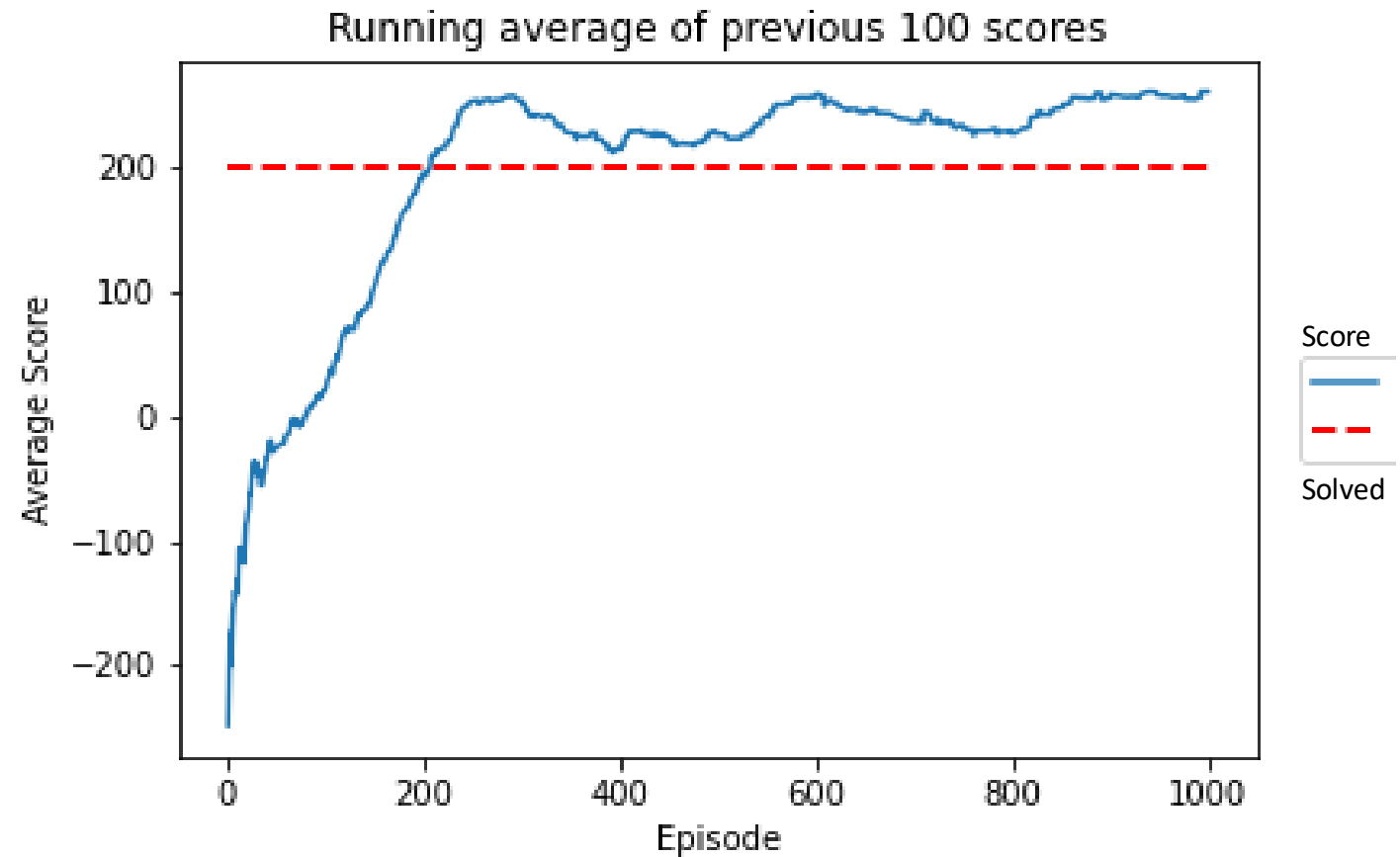
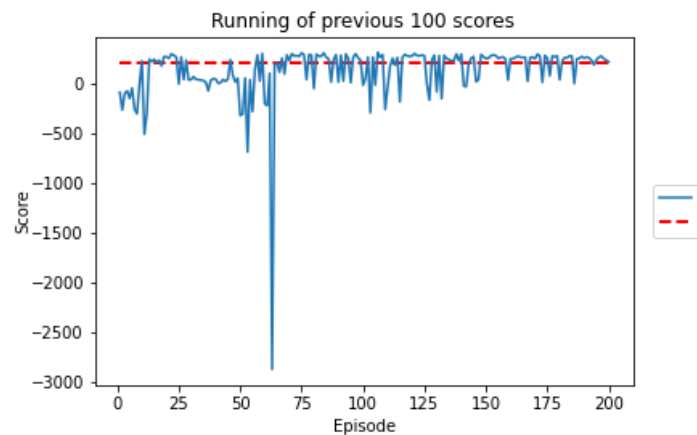
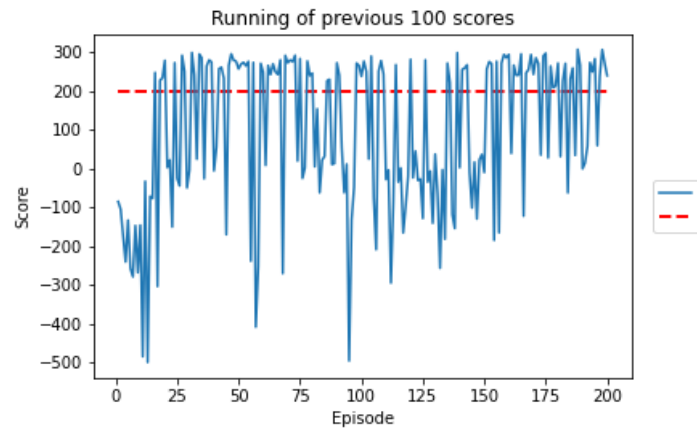
- Network Architecture [2]



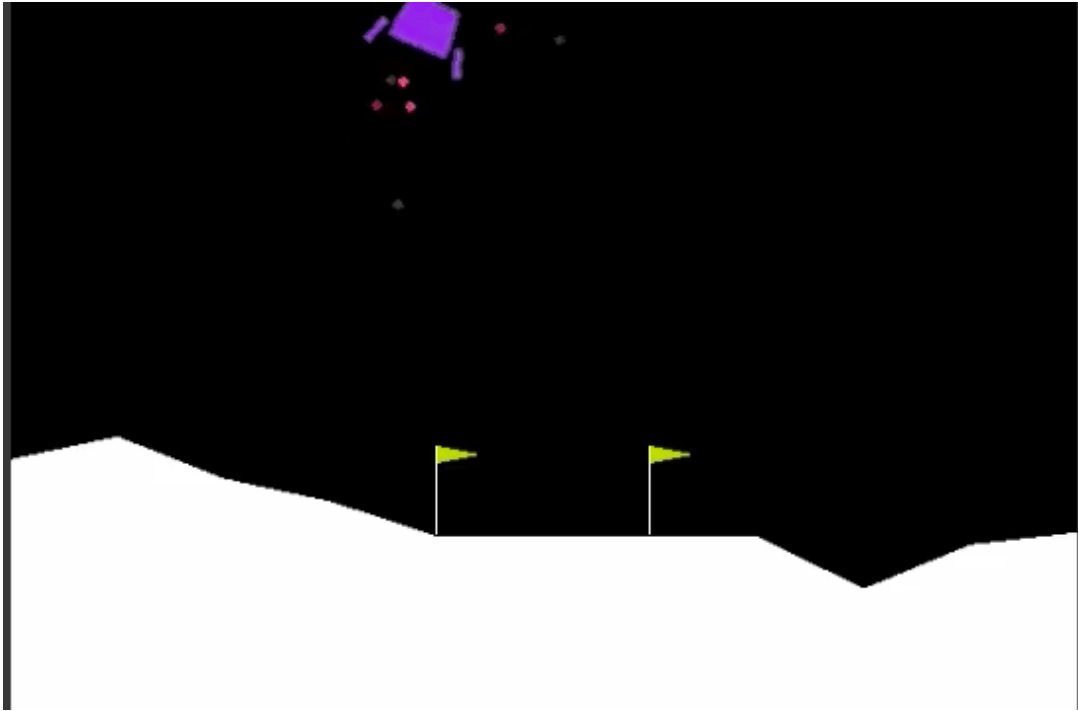
[1] Fujimoto, S., Van Hoof, H., & Meger, D. (2018). "Addressing function approximation error in actor-critic methods". arXiv preprint arXiv:1802.09477.

[2] Gao, J., Ye, W., Guo, J., & Li, Z. (2020). "Deep Reinforcement Learning for Indoor Mobile Robot Path Planning". Sensors, 20(19), 5493.

# Results



# Simulation



Point: 276.74234249152335



Point: 297.34392408211403

# References

- [1] Fujimoto, S., Van Hoof, H., & Meger, D. (2018). "Addressing function approximation error in actor-critic methods". *arXiv preprint arXiv:1802.09477*.
- [2] Gao, J., Ye, W., Guo, J., & Li, Z. (2020). "Deep Reinforcement Learning for Indoor Mobile Robot Path Planning". *Sensors*, 20(19), 5493.



The end.