DELE CA2 Reinforcement Learning

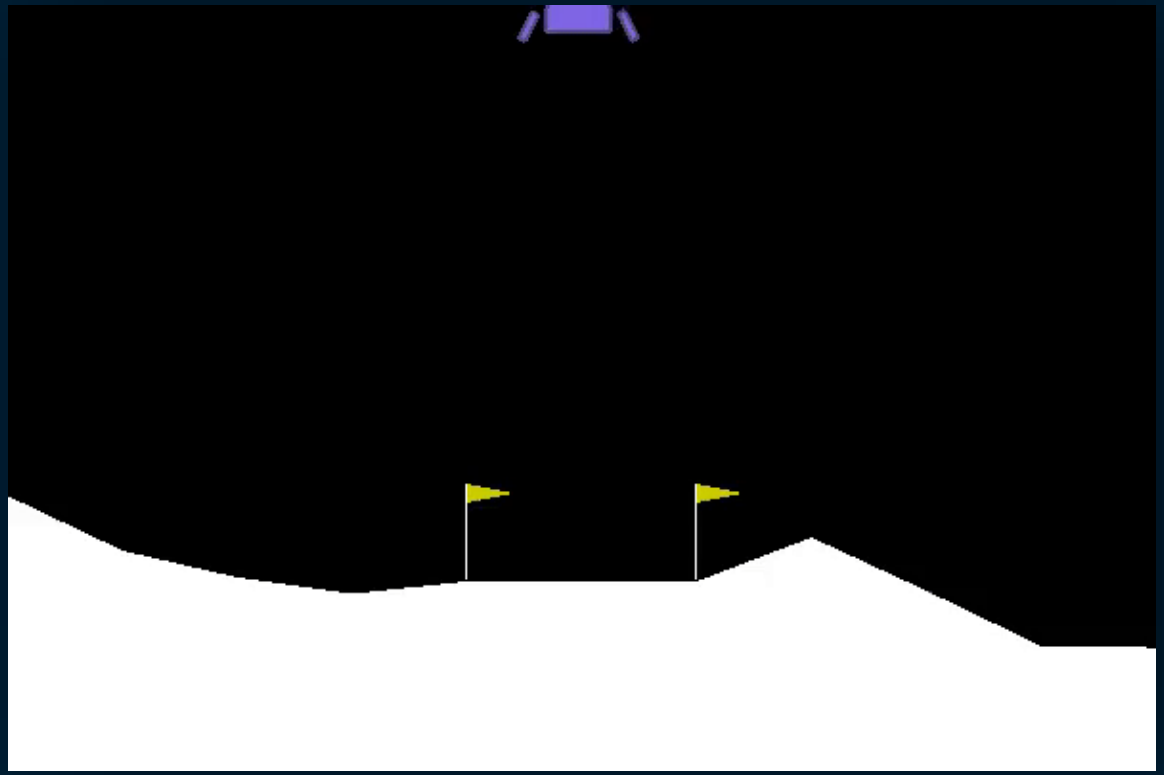
SHI TINGXIAO (P2033444)
Tan Yu Hoe (P2026309)

Exploratory Data Analysis

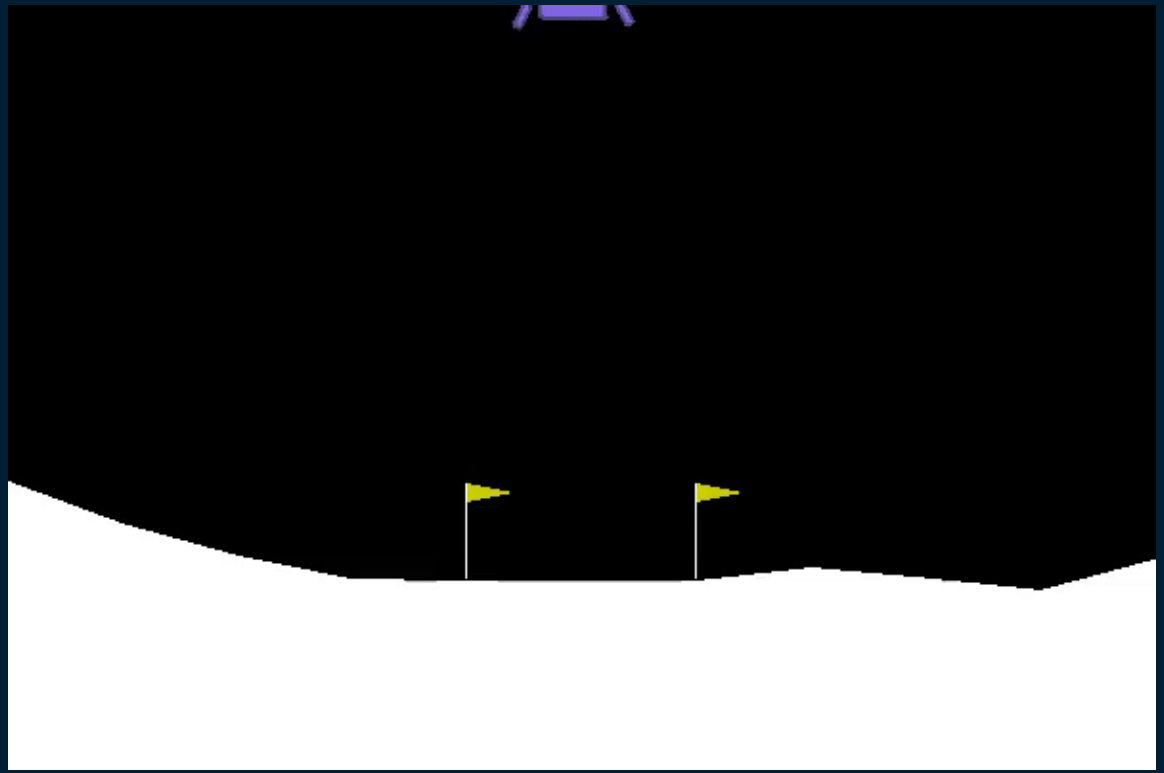
Observation Space

0	Lander’s X Coordinate
1	Lander’s Y Coordinate
2	X Linear Velocity
3	Y Linear Velocity
4	Angle
5	Angular Velocity
6	Left Leg is touching the ground
7	Right Leg is touching the ground

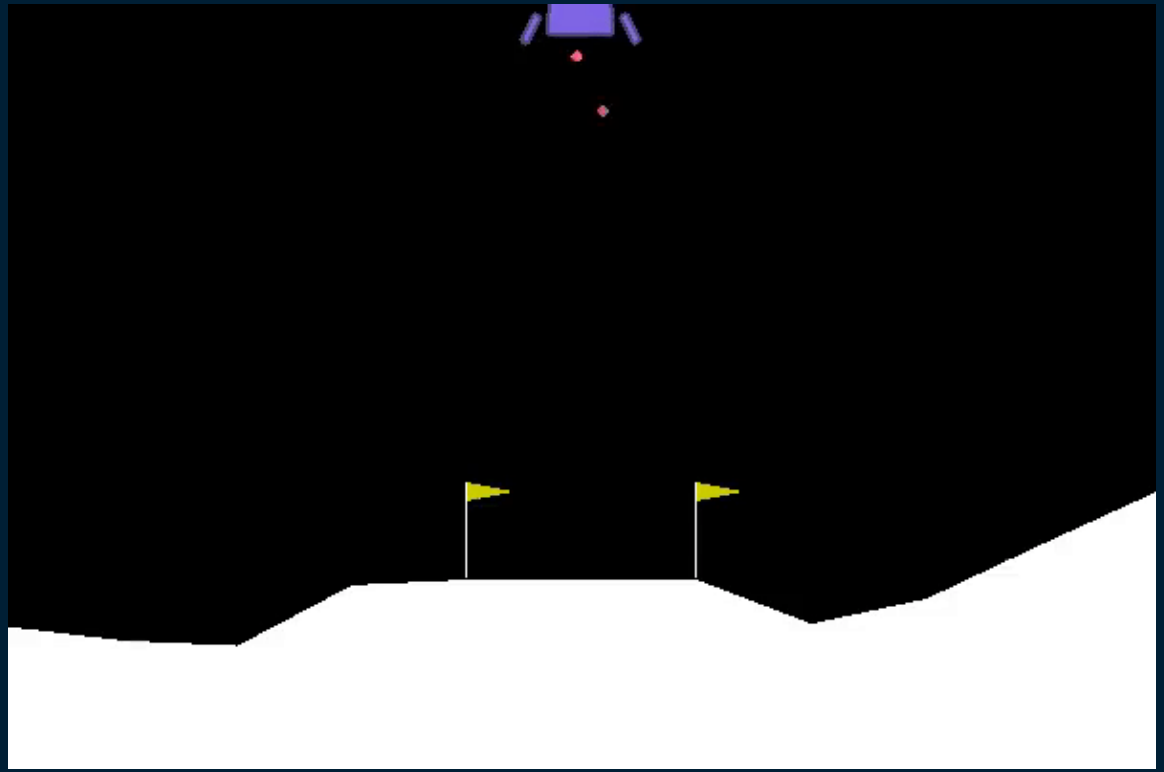
Action 0 Do Nothing



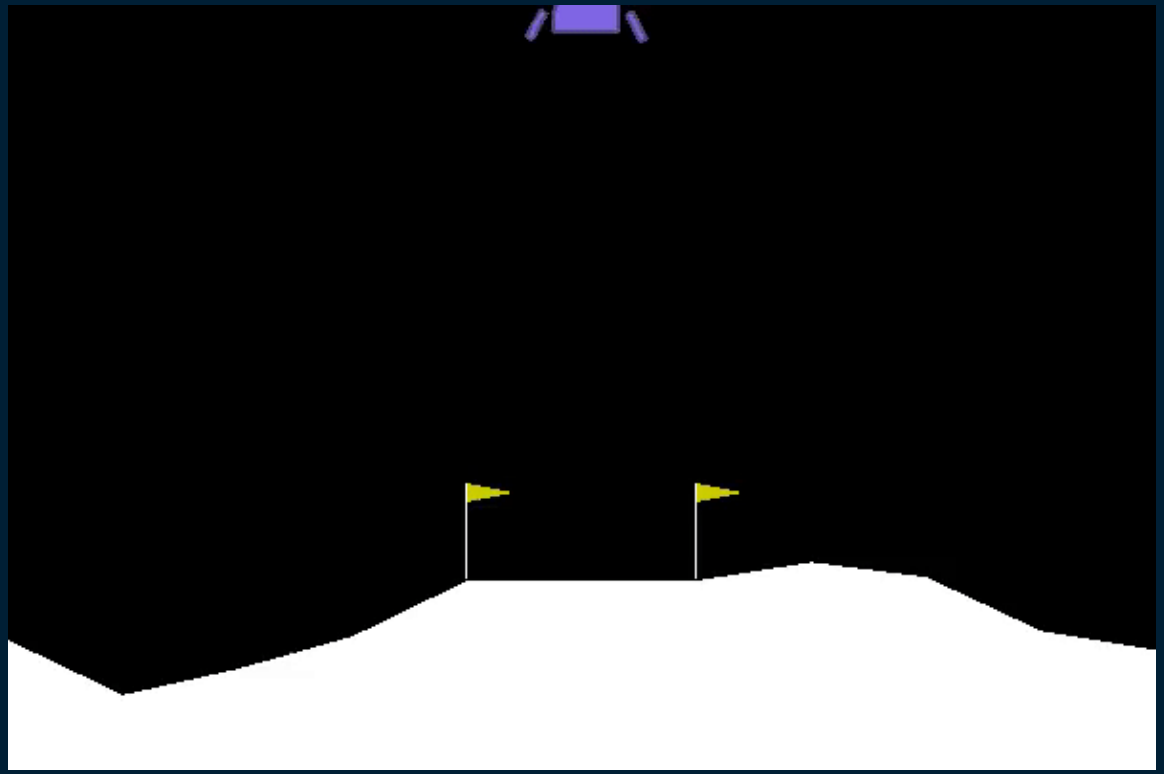
Action 1 Fire Right Engine



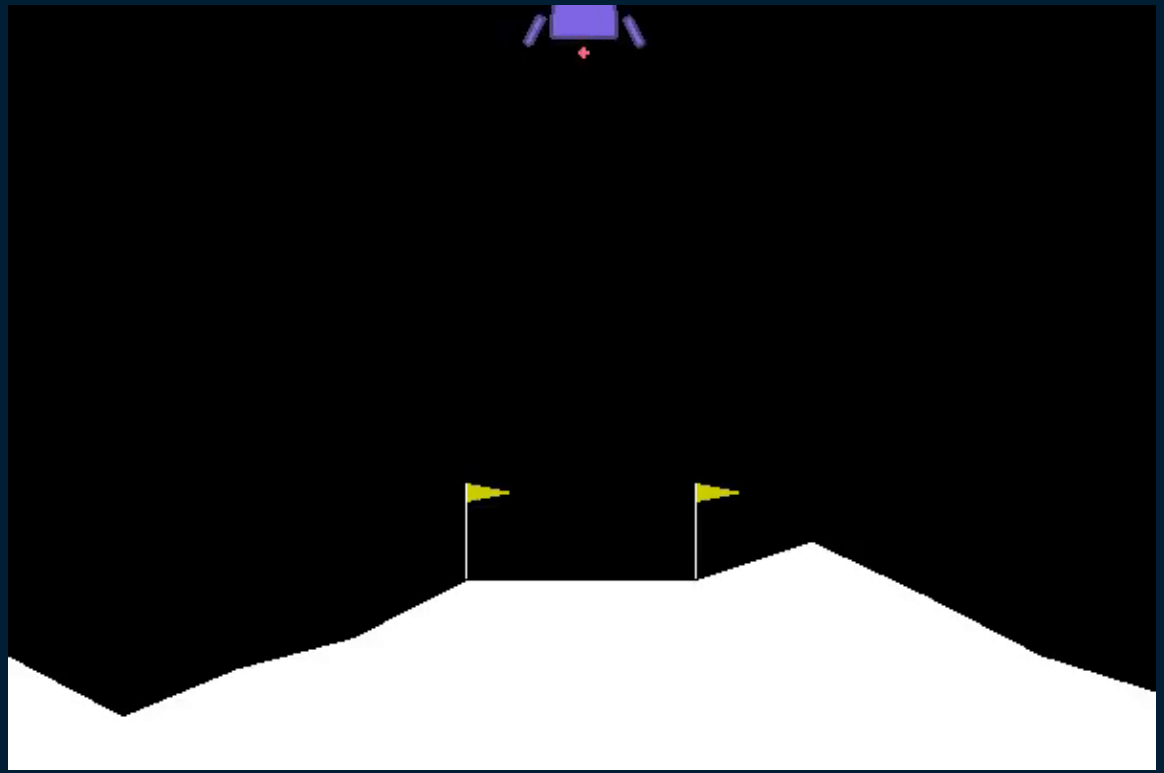
Action 2 Fire Main Engine



Action 3 Fire Left Engine



Random Actions



Reward Scenarios

Moving from top of screen to the landing pad and zero speed	+100-140
Moves away from landing pad	Lose Reward
Lander Crashes	-100
Lander comes to rest	+100
Each leg with ground contact	+10
Firing Main Engine	-0.3 per frame
Firing Side Engine	-0.03 per frame
Solving Scenario	≥ 200 points

Episode Termination

Lander Crashes

In contact with the moon

Lander gets outside of the view port

X coordinate > 1

Lander is not awake

Body doesn't move and doesn't collide with any other body

Models we attempted

Models that we have attempted

Deep Q-Network

Dueling Deep Q-Network

Double Dueling Deep Q-Network

Actor To Critic

Deep Q Learning

Value Based Model

What is Deep Q Learning

- Deep Q Learning is an extended version of Q Learning
- Instead of approximating Q value using a function
- Deep Q Learning uses a neural network to approximate Q

Experience Replay and Replay Buffer

- Store past experiences
- Uses a random subset of these experiences to update the Deep Q-Network

- Contains a collection of experience as a Tuple
- (State, Action, Reward, New State)
- A buffer is added to the end so that it pushes the oldest experience out

Target Network

- When neural networks parameters are updated to Q(s, a)
- Q(s’, a’) and other nearby states are also affected.
- Makes training unstable

- Target Network makes a copy of the Deep Q-Net to serve as stable target every N number of steps.
- The Target Network is used to create Q(s’, a’)

Deep Q Network Architecture

- Input layer taking in 8 states
- Two FC Hidden Layers with 256 nodes with ReLU Activation
- FC Output Layer with 4 nodes
- Compiled with Adam Optimizer and MSE Loss

```
def DeepQNetwork(lr, num_actions, input_dims, fc1, fc2):
    q_net = Sequential()
    q_net.add(Dense(fc1, input_dim=input_dims, activation='relu'))
    q_net.add(Dense(fc2, activation='relu'))
    q_net.add(Dense(num_actions, activation=None))
    q_net.compile(optimizer=Adam(learning_rate=lr), loss='mse')

    return q_net
```

DQN Policy

- Takes the highest Q-value of the current state and chooses the action with the high Q-value. (using `.argmax()` operation)
- Explores the environment by randomly choosing actions for the first few episodes.

$$a \leftarrow argmax\ Q(s, a_s)$$

Training Function

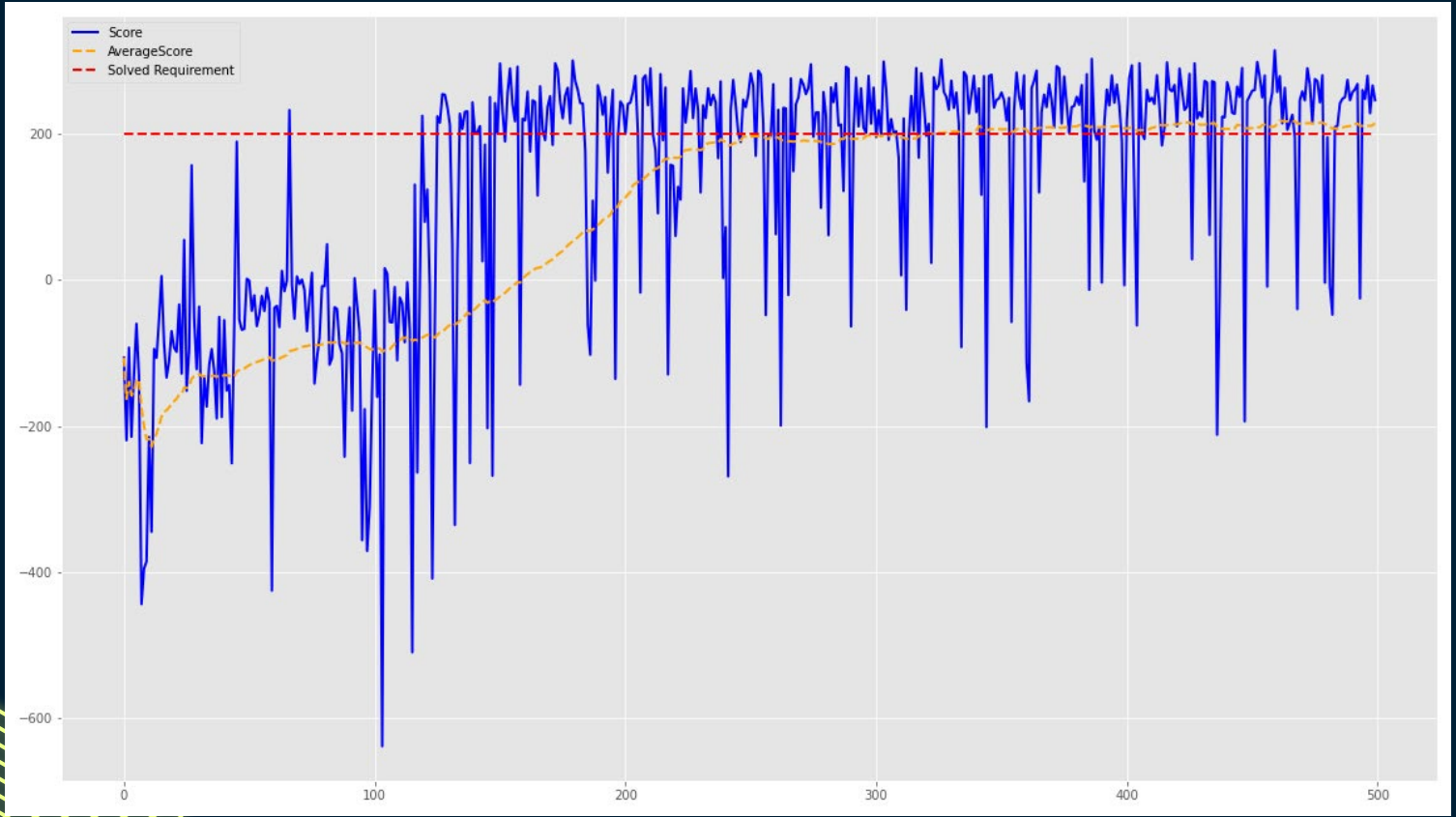
For every State

1. Copy QNet over to Target QNet (Every 120 steps)
2. Sample a batch of experience
3. Get Q value for the current state
4. Calculate Target Q Value using reward, discount factor, and the max Q value
5. Network performs backpropagation using state and target Q value.

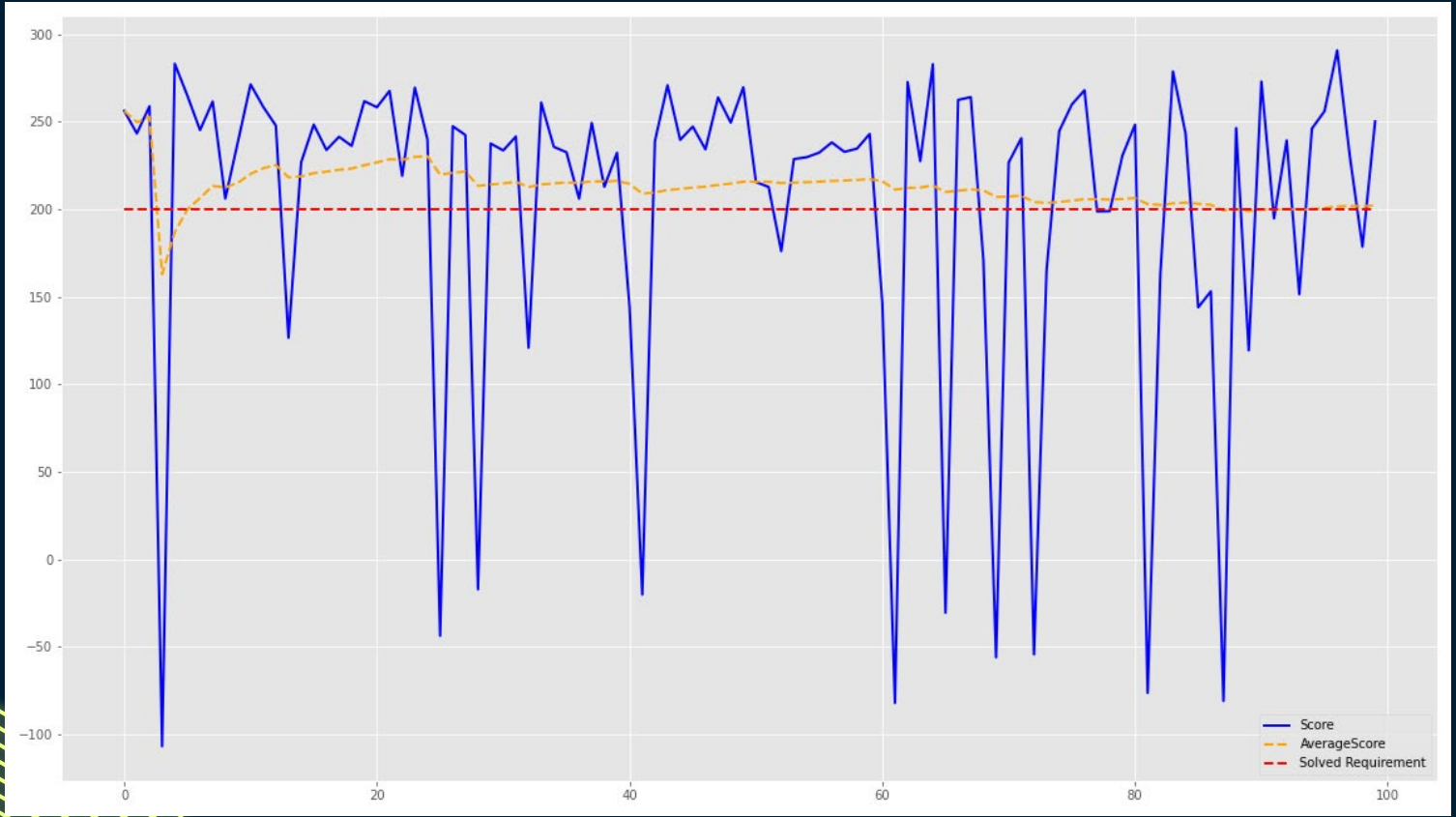
$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

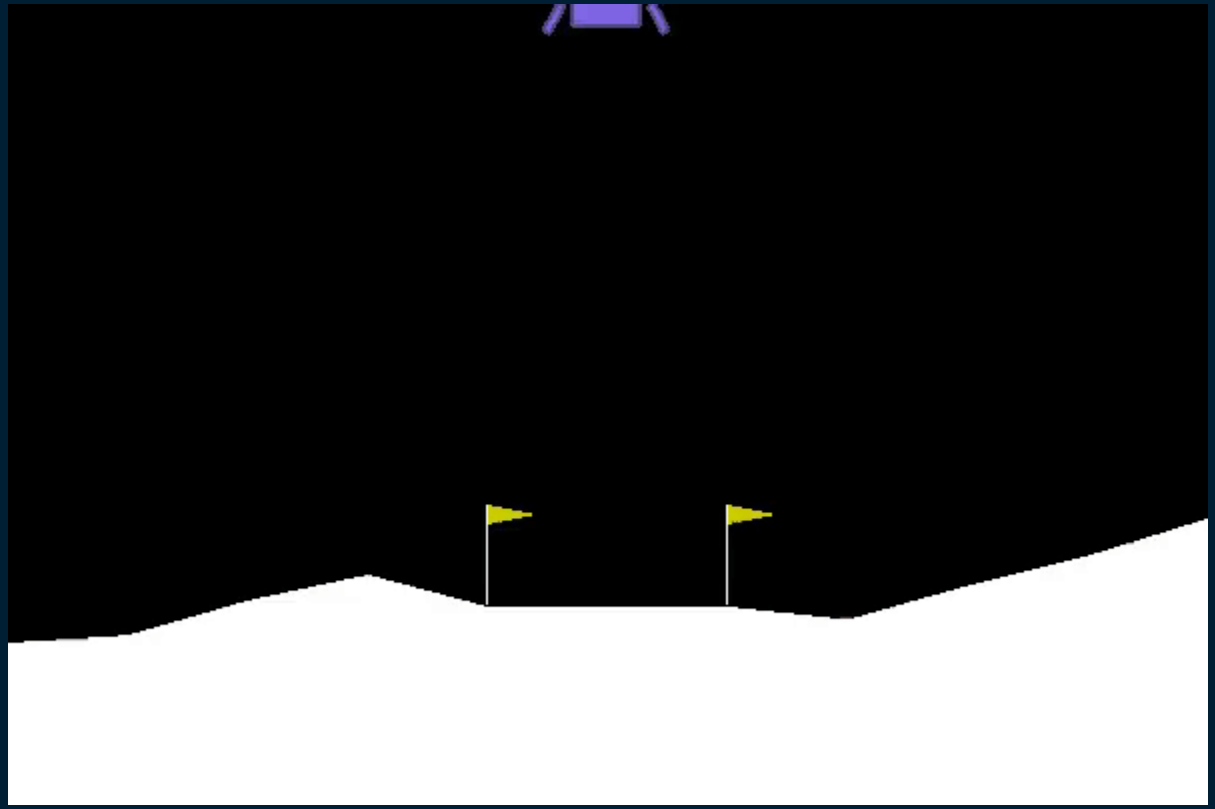
DQN Train 500 Episodes

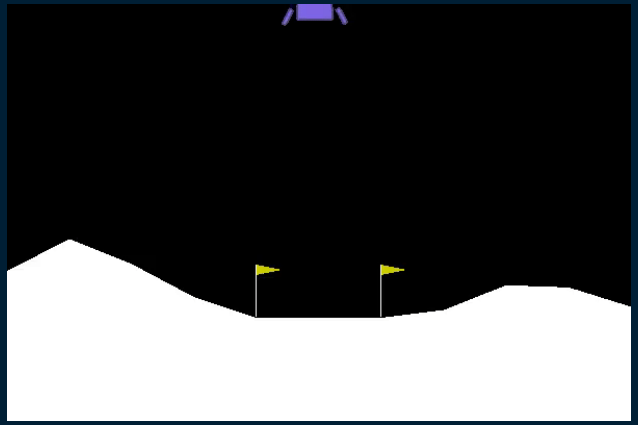


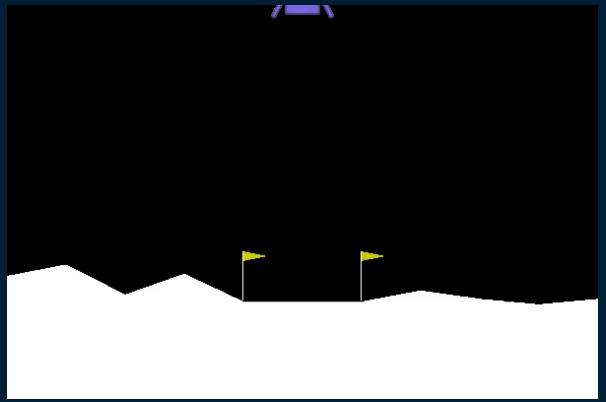
DQN Test 100 episodes



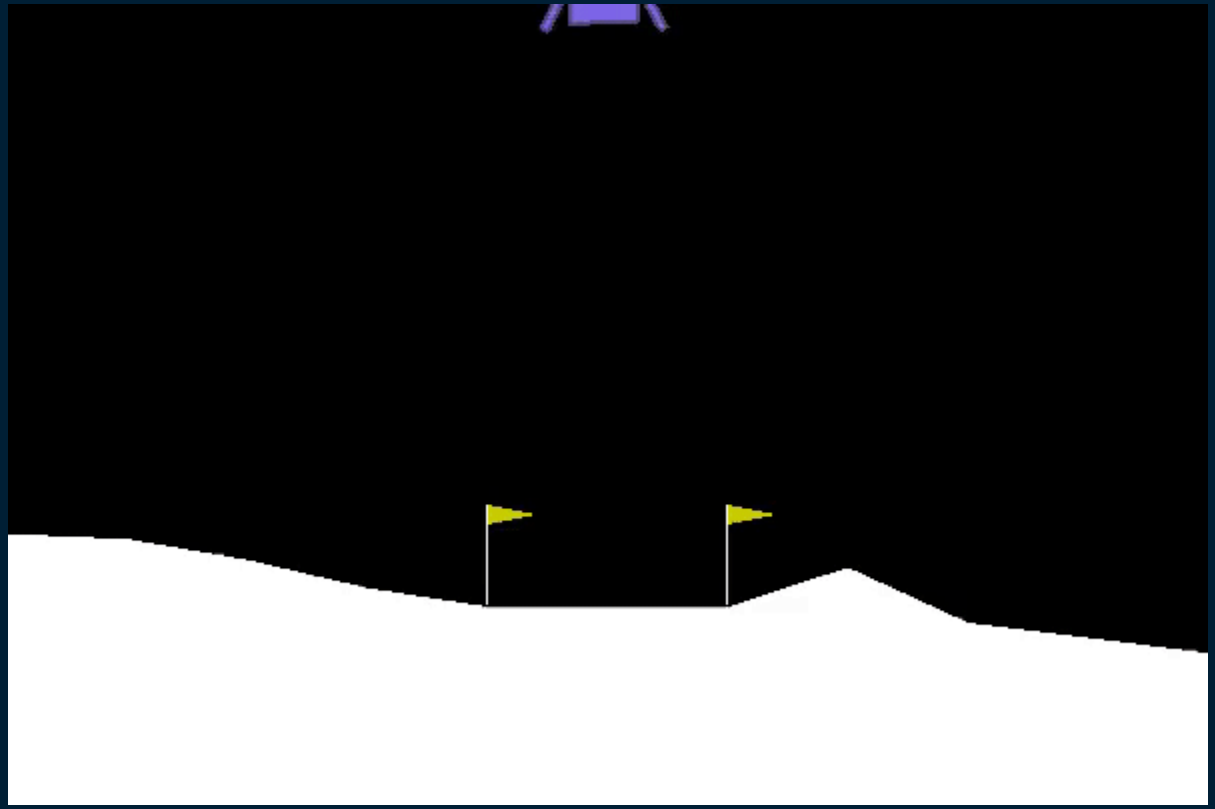
DQN Video Showcase

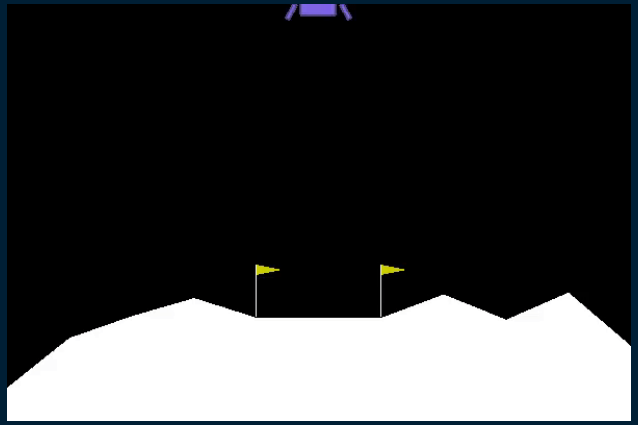






DQN Video Showcase





The problem of Single DQN

Temporal Difference learning makes DQN overestimate action-values.

1) TD target utilised maximization, TD target is bigger than the real action-value

```
for idx in range(done_batch.shape[0]):
    target_q_val = reward_batch[idx]
    if not done_batch[idx]:
        target_q_val += self.discount_factor*q_max_next[idx]
```

1) Since TD target is overestimated, bootstrapping propagates the overestimation
2) Since each overestimation is bootstrapped, once DQN overestimates, it will be a vicious circle of overestimation.

02

Dueling Deep Q Network

Value Based Model

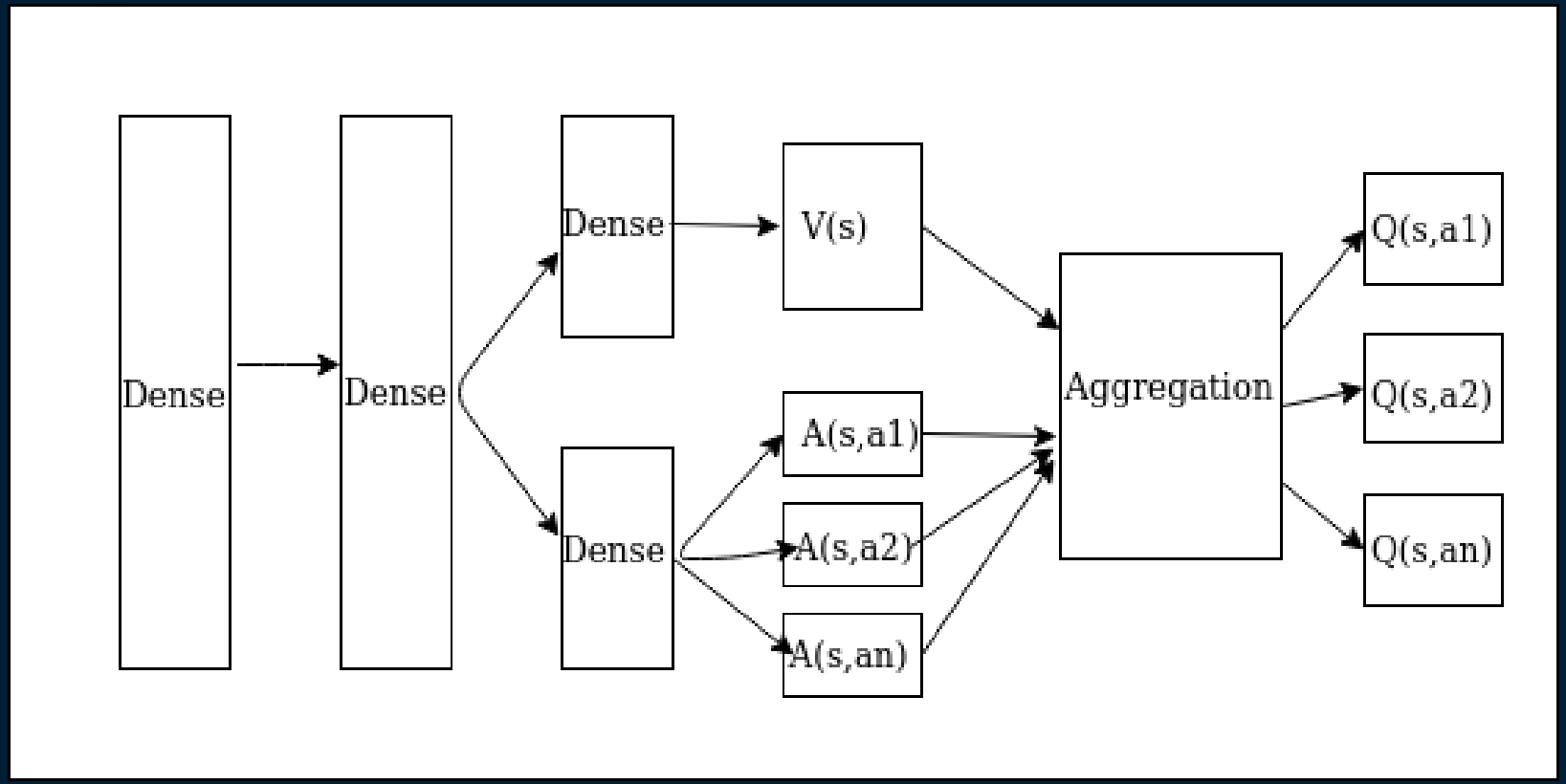
What is Dueling Deep-Q Network

- The model is represented by the equation below,

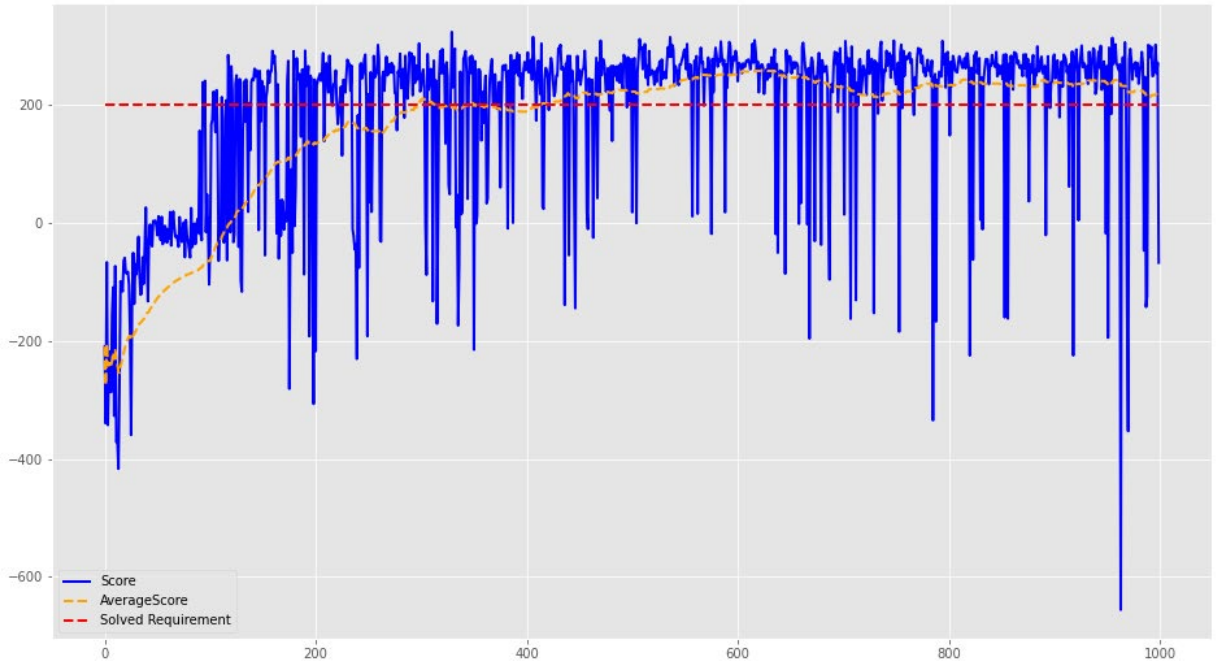
$$Q = V + A - \textit{mean of all } A$$

- The model consists of two estimators
  - State value function *V*
  - State dependent action advantage function *A*
- These two estimators are combined to generate the action-value function
- The benefit is that the state-value function is quicker and easier to learn → Faster Convergence

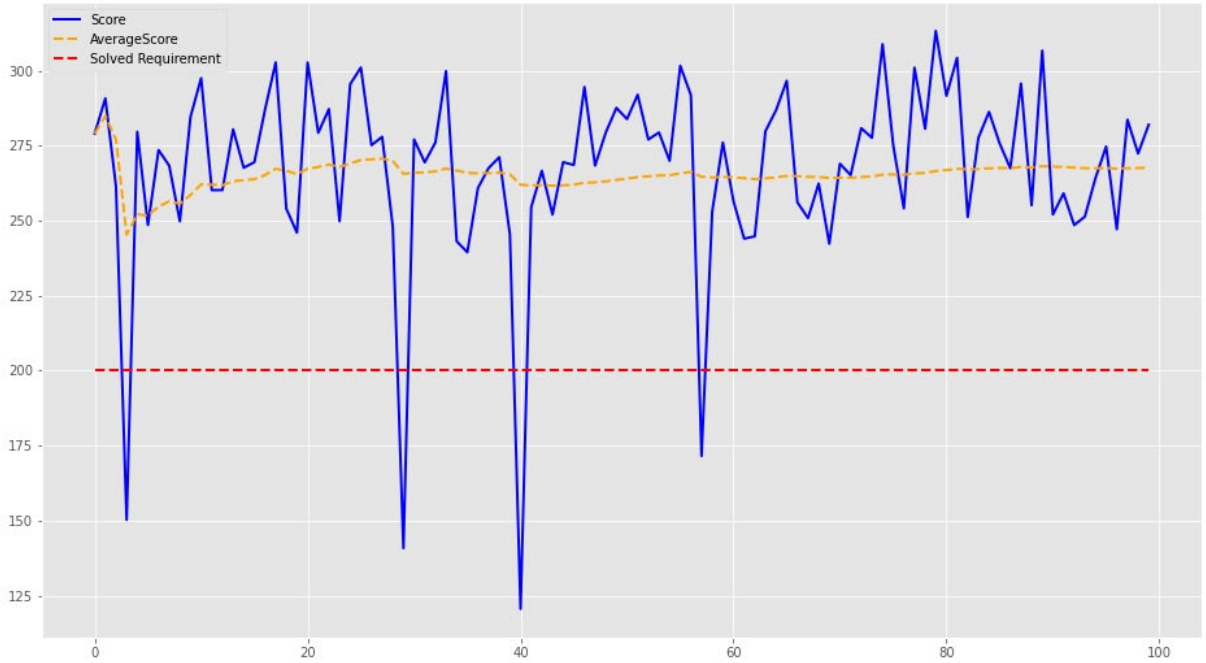
What is Dueling Deep-Q Network



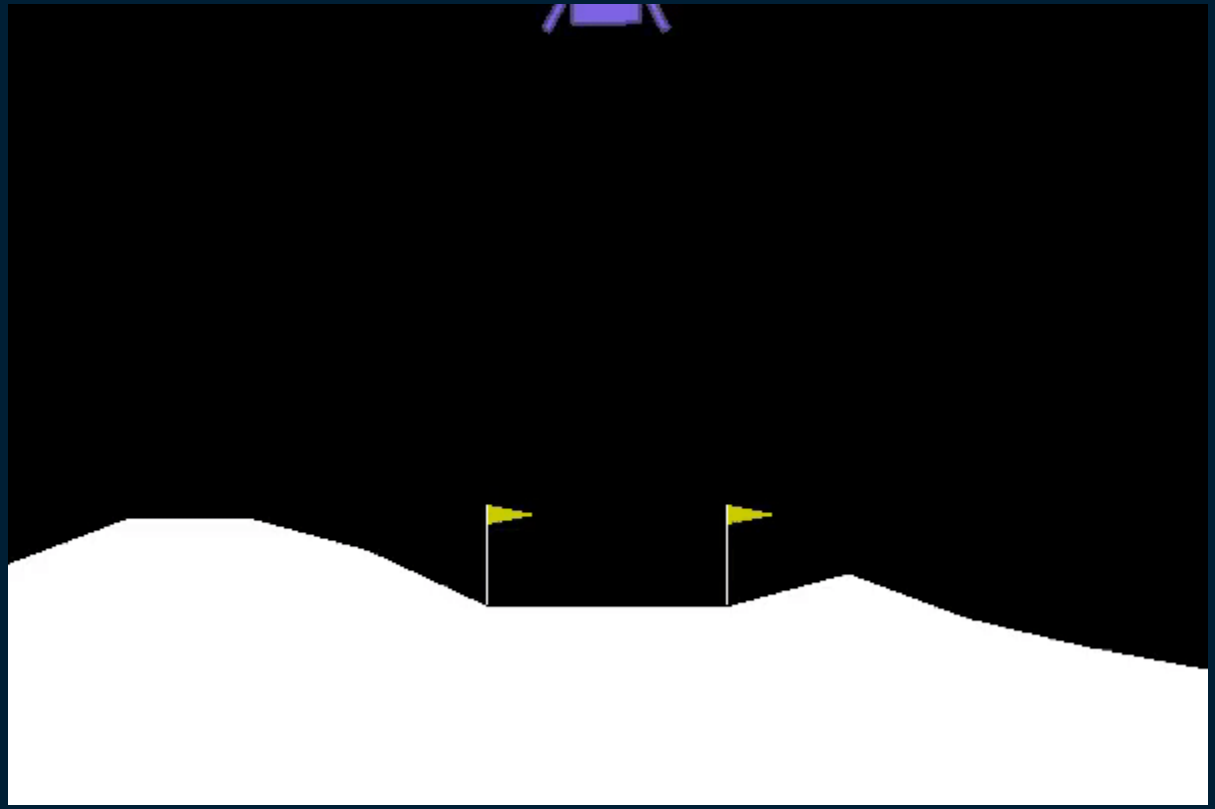
DDQN Train 1000 Episodes

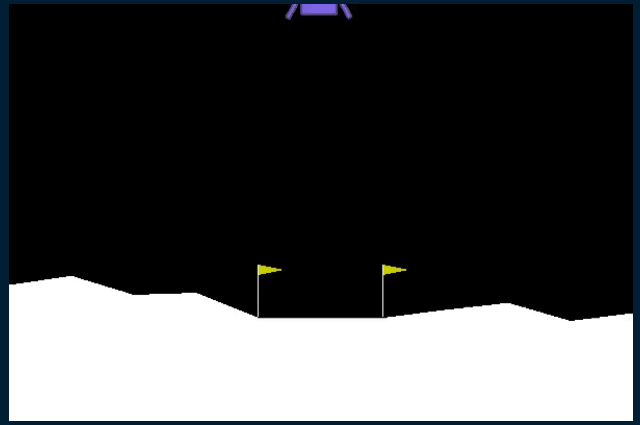


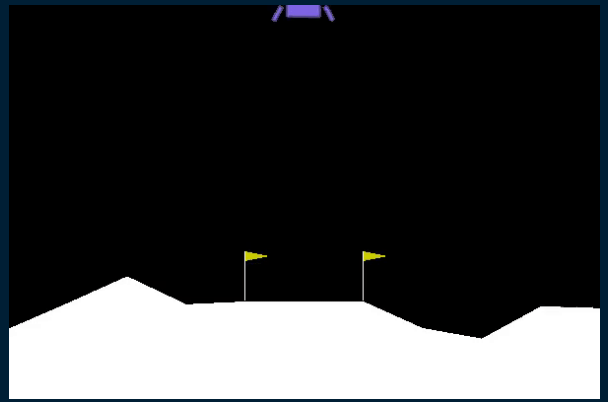
DDQN Test 100 episodes



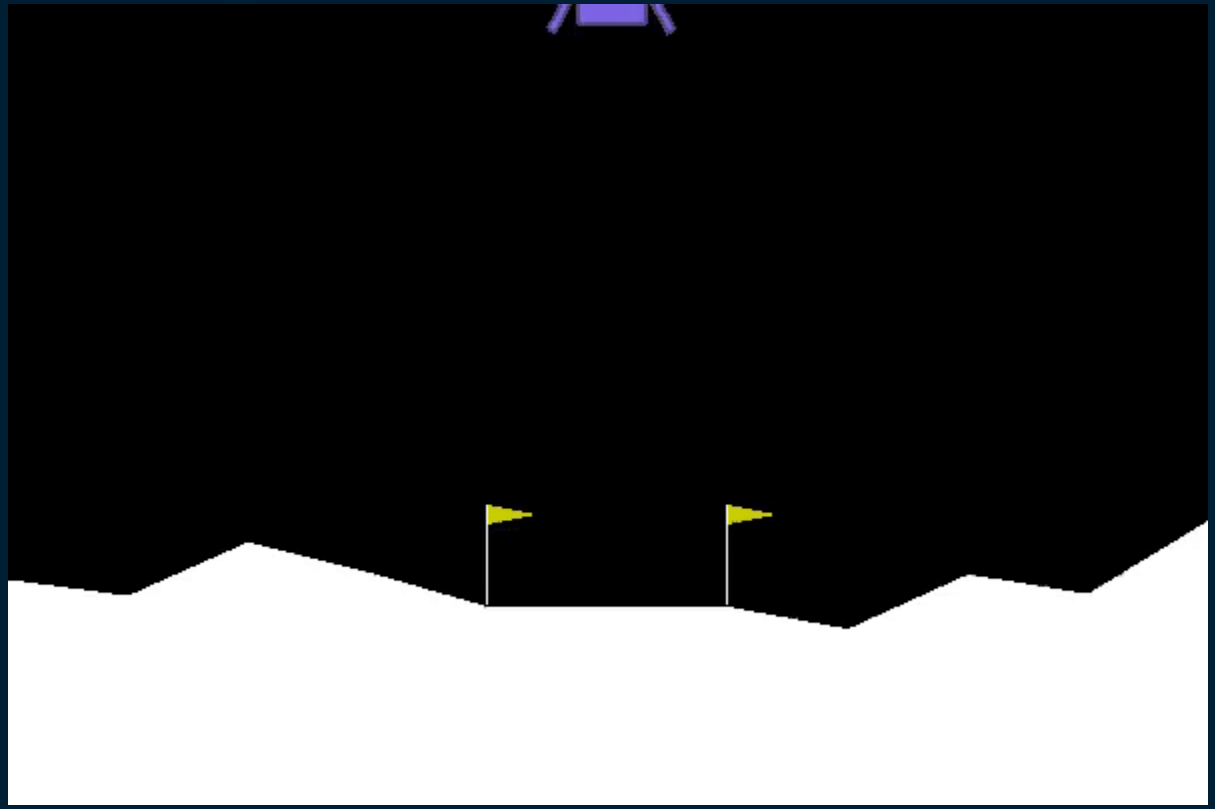
DDQN Video Showcase

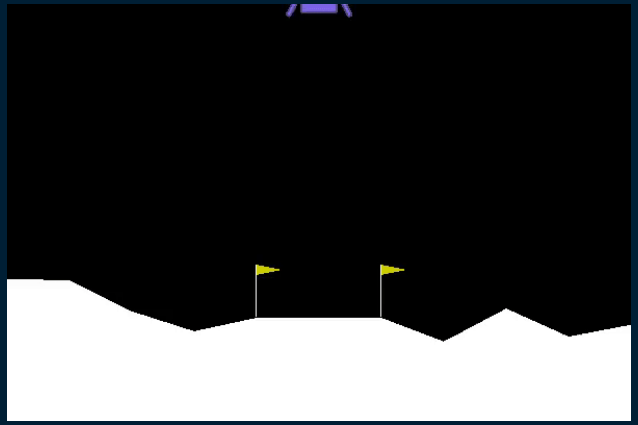




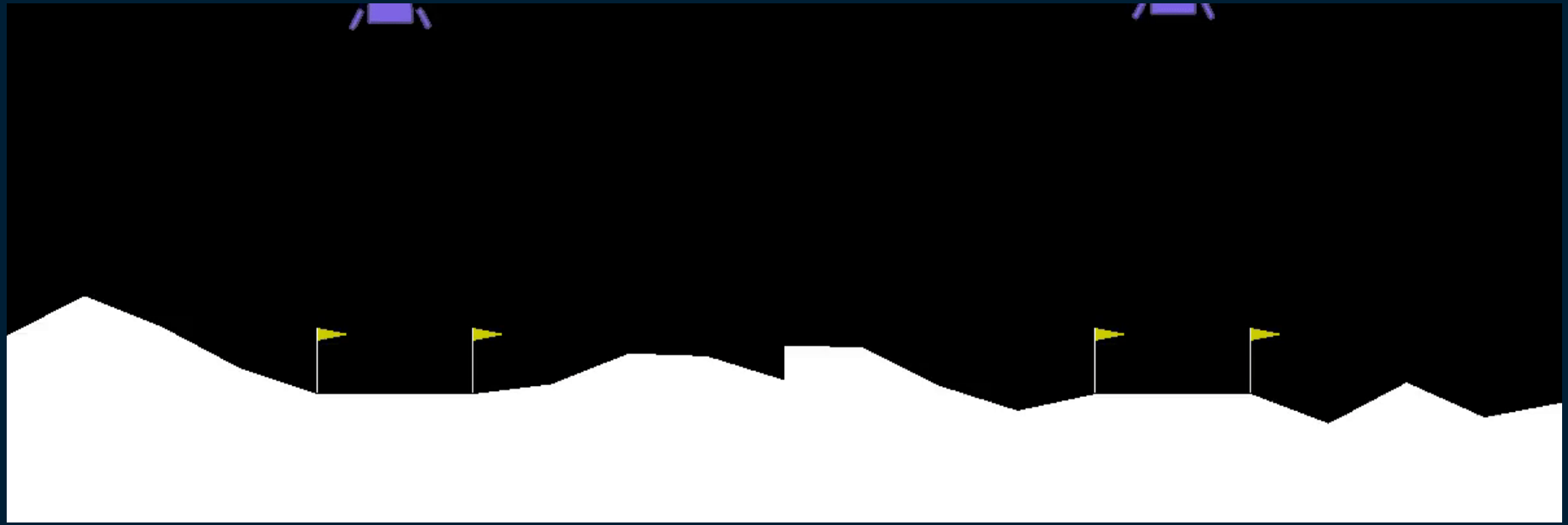


DDQN Video Showcase



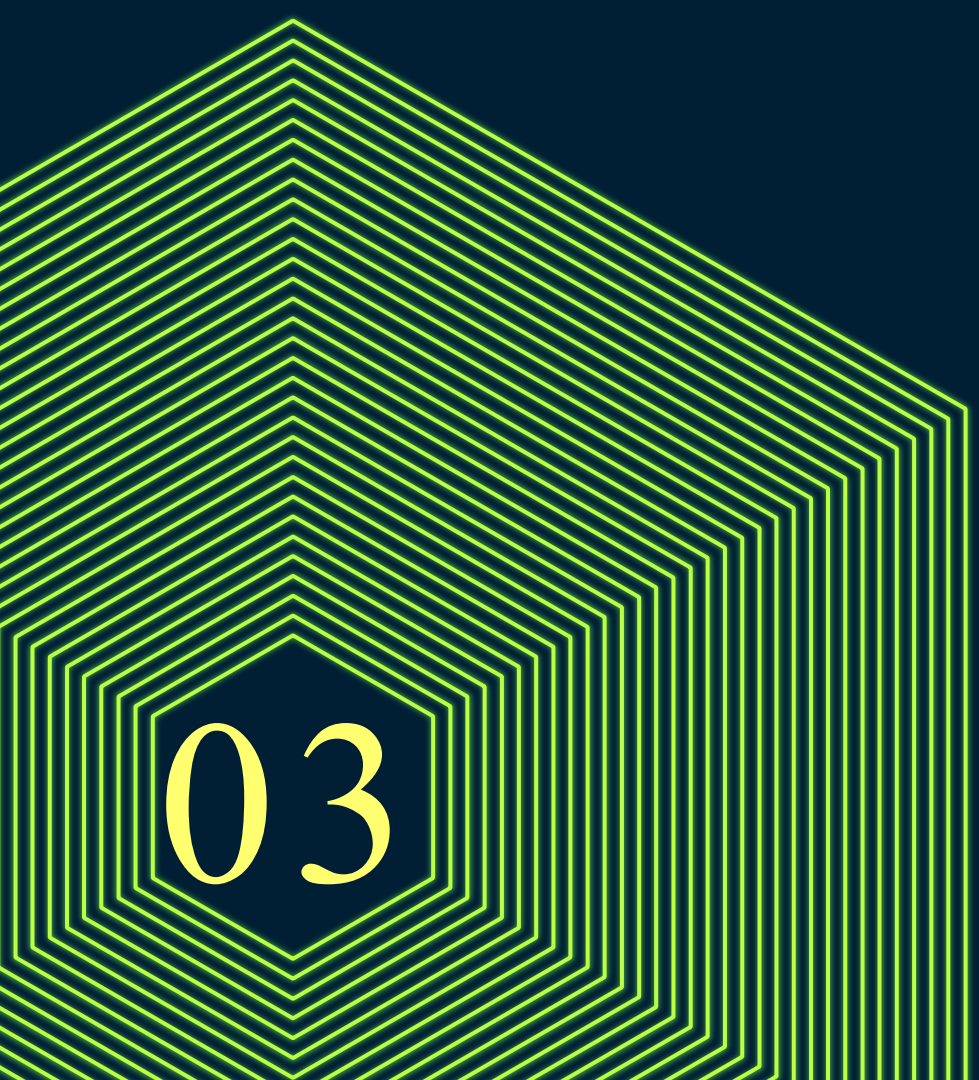


Cross Comparison DQN vs DDQN



Deep Q-Network

Dueling Deep Q-Network



Double Dueling Q-Network

Value Based model

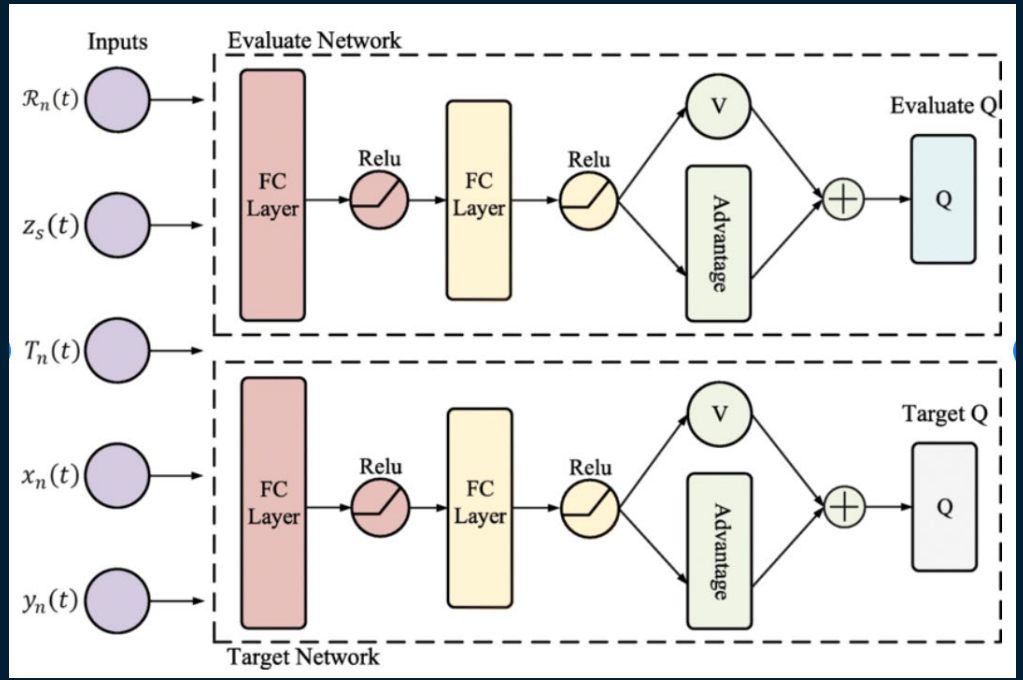
What is Double Dueling Deep Q Learning

- The double dueling DQN is pretty similar to the Dueling DQN, there are two consisting estimator - the state value function, and the state dependent action advantage function.

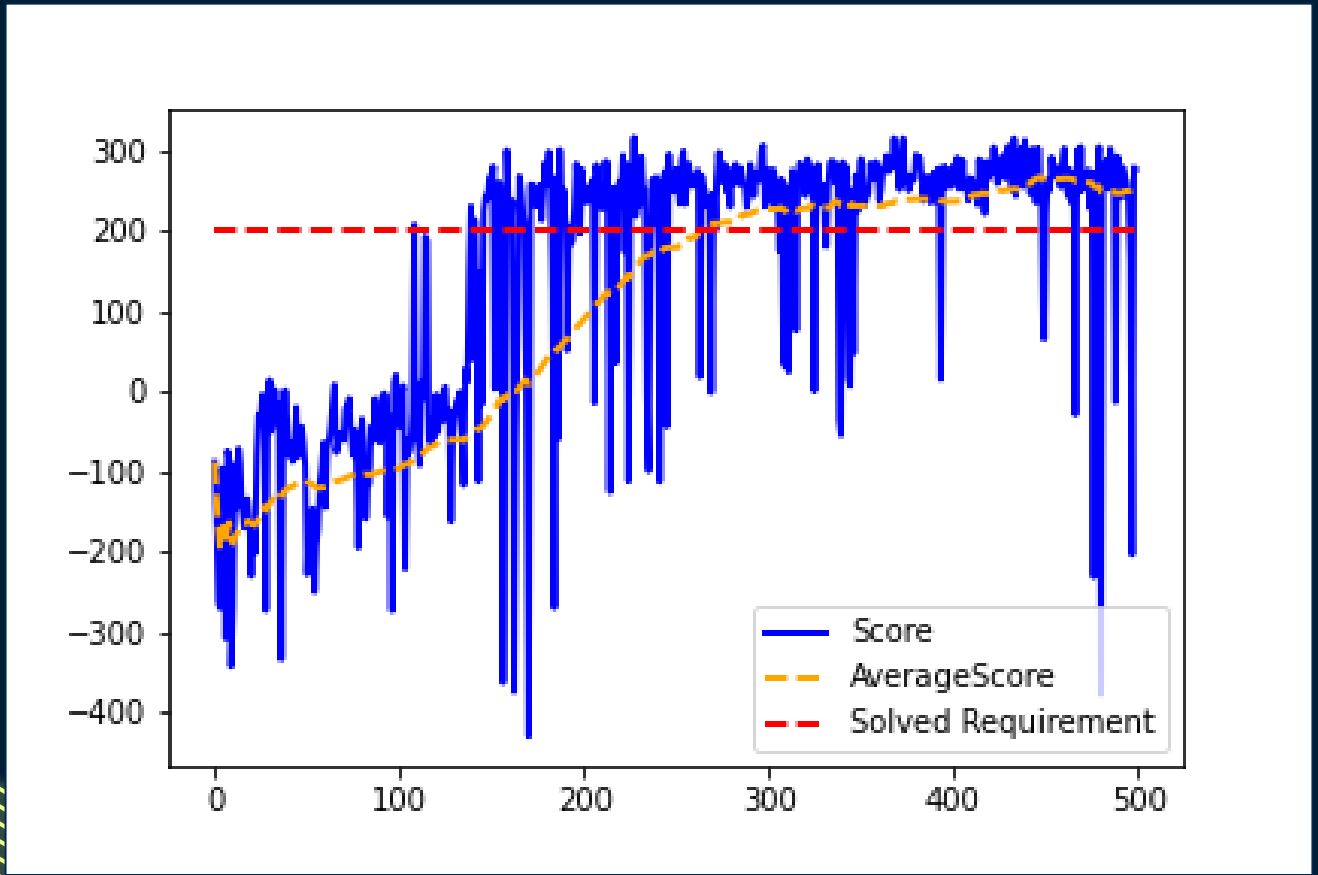
$$Q_1(S, A) \leftarrow Q_q(S, A) + \alpha(R + \gamma Q_2(S', argmax_a Q_1(S, a)) - Q_1(S, A))$$

- However there is an issue with single DQNs. The first time an agent receives a positive reward, it might continue making the same set action. A solution to solve this problem is to use two action value functions, in other words, two lookup tables. The agent uses one to update the other Q-value, providing an unbiased estimate of Q(s, a).

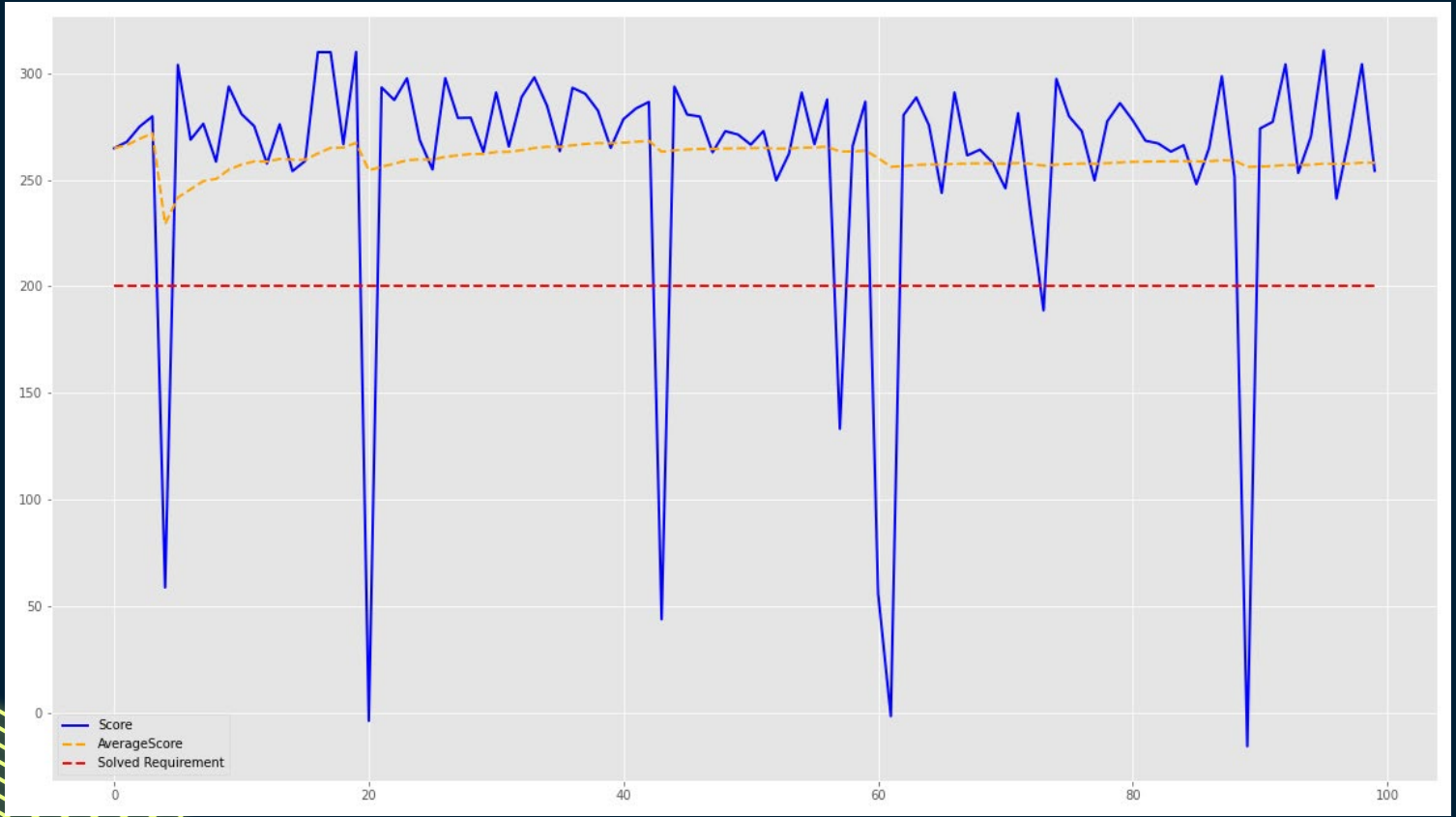
What is Double Dueling Deep Q Learning



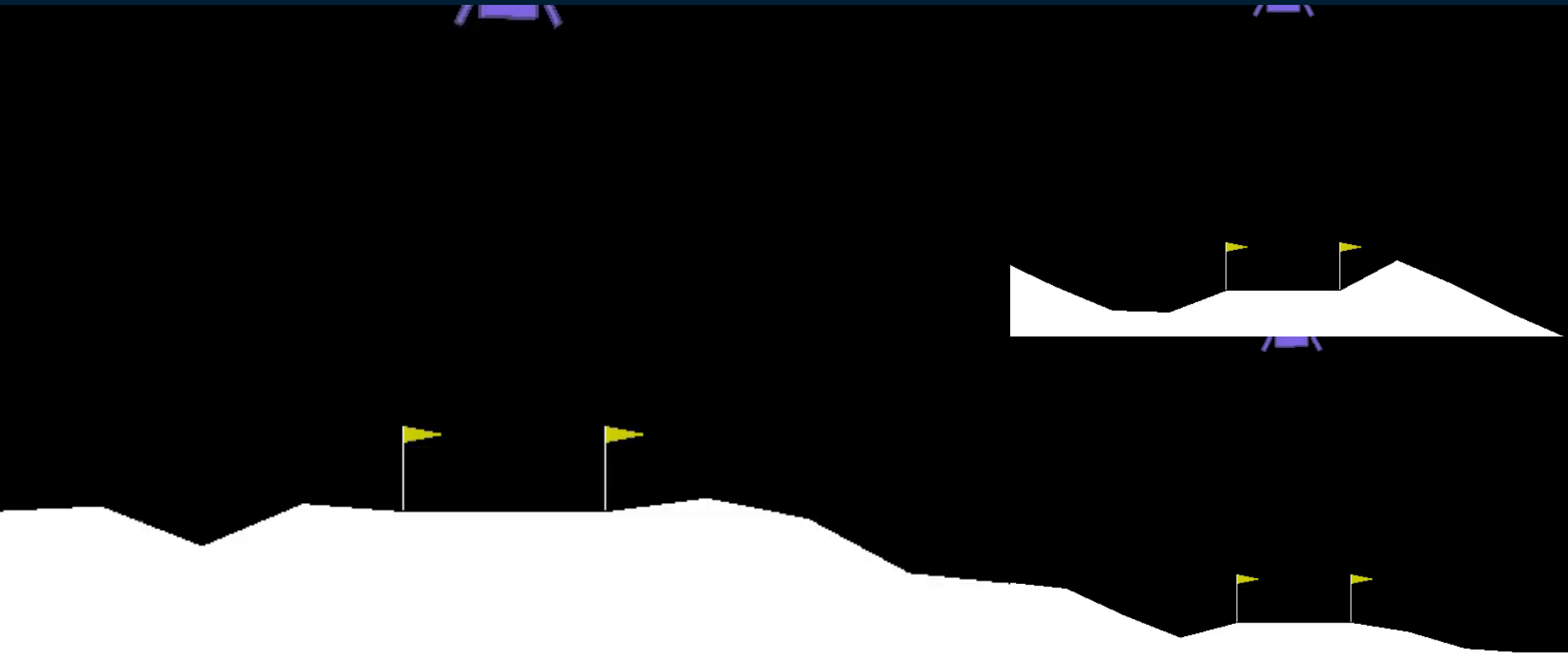
D3QN Train 500 episodes



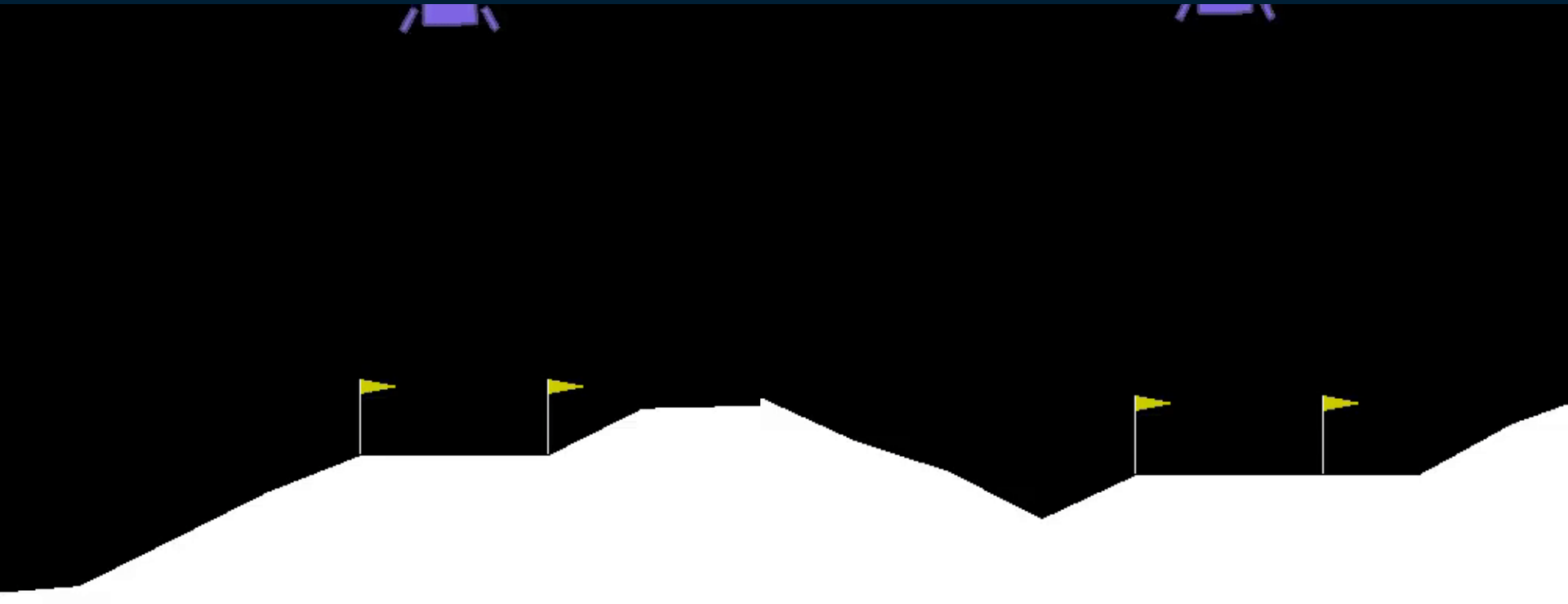
D3QN Test 100 episodes



D3QN Video Showcase



D3QN Video Showcase



Cross Comparison

D3QN

DDQN

Actor to Critic

Value and Policy based model

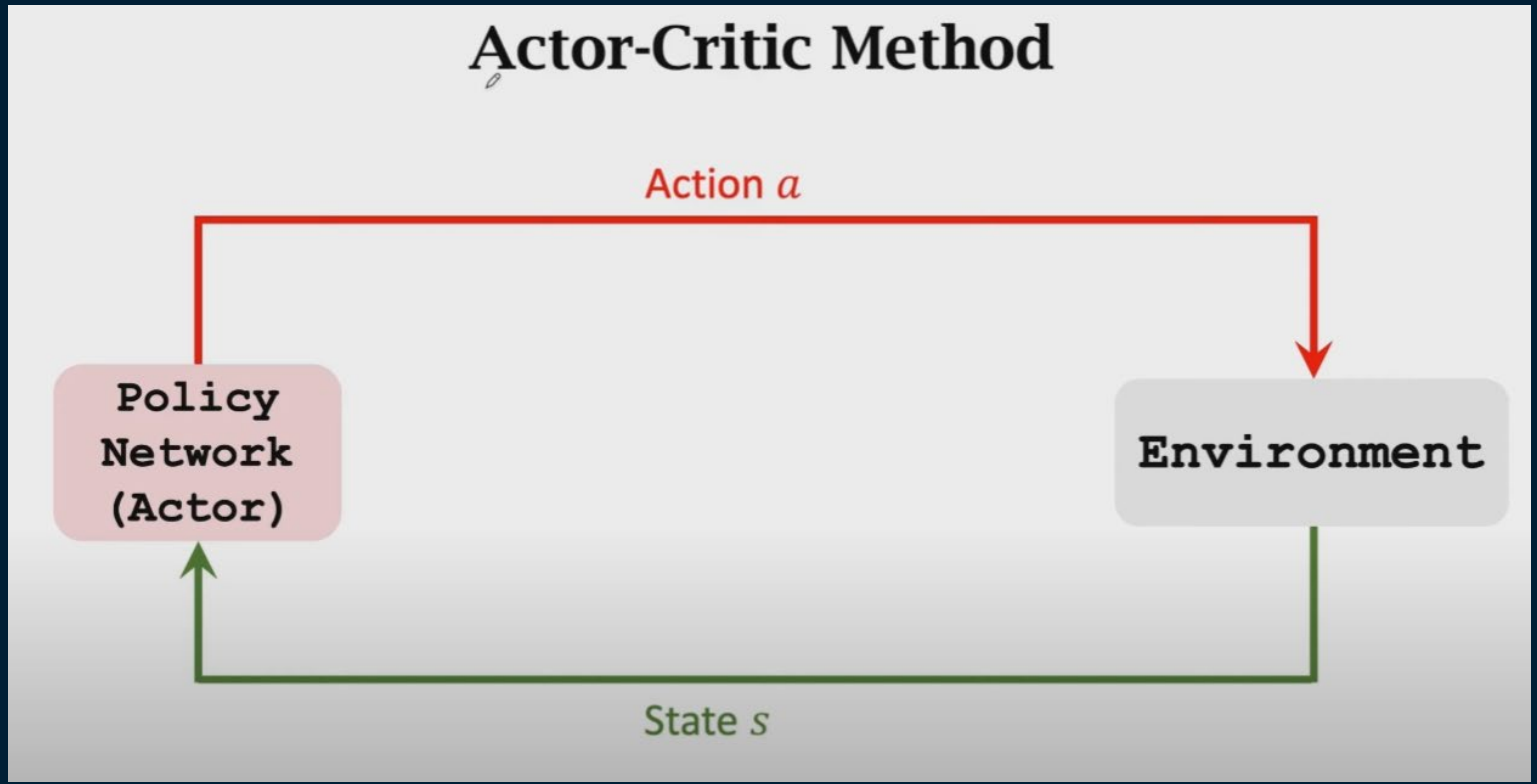
What is Actor to Critic

Actor critic network is a combination of policy-based method and value based method

Actor is a policy network, it actively trying to
Search for the best policy in the action table given state s
Policy(s) = argmax a (Q[s,a])

Critic is a value network, it rates the action, and help actor learn

Actor's Role

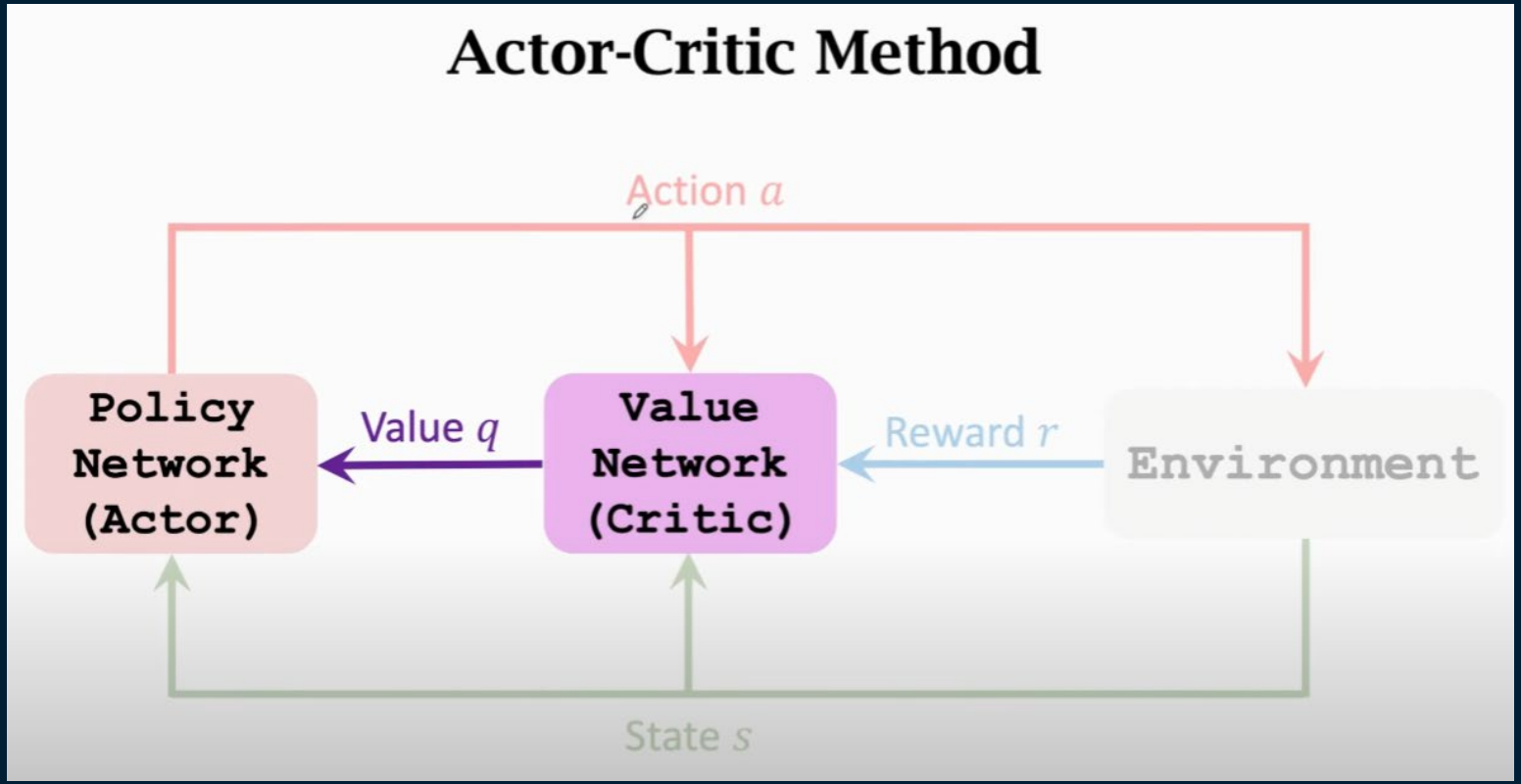

Actor-Critic Method
Action a
Policy Network (Actor)
Environment
State s

Actor network

With the input of 8 states, the actor network would choose which action to take, the dense layers using relu activation while the last dense uses softmax.

```
def build_model_actor(lr = 0.001, size = [128,128,64]):
    # input the state and output the actio
    model = Sequential()
    # dense layers 1
    model.add(Dense(size[0], input_shape = (8,), activation = 'relu'))
    # dense layers 2
    model.add(Dense(size[1], activation = 'relu'))
    # dense layers 3
    model.add(Dense(size[2], activation = 'relu'))
    # choose an action to take
    model.add(Dense(4, activation = 'softmax'))
    adam = optimizers.Adam(lr=lr, beta_1=0.9, beta_2=0.999)
    model.compile(loss = 'categorical_crossentropy', optimizer = adam)
    return model
```

Critic's Role



Critic Network

With the input of 8 states, the critic network will produce a

```
def build_model_critic( lr = 0.001, size = [128,128,64]):
    # input the state and output the rating
    model = Sequential()
    # dense layer 1
    model.add(Dense(size[0], input_shape = (8,), activation = 'relu'))
    # dense layer 2
    model.add(Dense(size[1], activation = 'relu'))
    # dense layer 3
    model.add(Dense(size[2], activation = 'relu'))
    # output layer
    model.add(Dense(1, activation = 'linear'))
    # compile the model with adam optimizer and 'MSE loss'
    adam = optimizers.Adam(lr=lr, beta_1=0.9, beta_2=0.999)
    model.compile(loss = 'mse', optimizer = adam)

    return model
```

Run Episodes

When the state is not done(either success or crash) or action is less than 1000 (to give model time to react to the env) the episode will still be running.

The memory stores the continuous state, reward and action.

```python
def run_episode(env, actor, render = False):
    # creete the meomory list
    memory = []
    # reset the environment
    state = env.reset()
    # get the first action
    episode_reward = 0

    cnt = 0

    done = False

    while not done and cnt <1000:
        cnt += 1
        if render:
            env.render()
        # predict the action
        action = decide_action(actor, state)
        observation, reward, done, _ = env.step(action)
        episode_reward += reward
        state_new = observation
        memory.append((state, action, reward, state_new, done))
        state = state_new

    return(memory, episode_reward)
```

Training Process

1) Actor observe the state S(t)
2) Actor randomly sample action A(t) from the action space
3) Actor perform action A(t) and observe the next state S(t+1) and reward R(t)
4) update the weight of the critic network using temporal difference
5) update the weight of the actor network using the policy gradient

```python
def decide_action(actor, state):   TINGXIAO, Today • a2c
    # flatten the state
    flat_state = np.reshape(state, [1,8])
    # numpy to choose a action with the probability associated with each action
     generated by the actor
    action = np.random.choice(4, 1, p = actor.predict(flat_state)[0])[0]
    return(action)
```

```python
def run_episode(env, actor, render = False):
    # creete the meomory list
    memory = []
    # reset the environment
    state = env.reset()
    # get the first action
    episode_reward = 0

    cnt = 0

    done = False

    while not done and cnt <1000:
        cnt += 1
        if render:
            env.render()
        # predict the action
        action = decide_action(actor, state)
        observation, reward, done, _ = env.step(action)
        episode_reward += reward
        state_new = observation
        memory.append((state, action, reward, state_new, done))
        state = state_new

    return(memory, episode_reward)
```

```python
def train_models(actor, critic, memory, gamma):
    '''
    # Training Process
    # observe the state S(t)
    # randomly samole action A(t) from the action space
    # perform action A(t) and observe the next state S(t+1) and reward R(t)
    # update the weight of the critic network using temporal difference
    # update the weight of the actor netowrk using the policy gradient
    '''
    random.shuffle(memory)

    for i in range(len(memory)):   TINGXIAO, Today • a2c

        state, action, reward, state_new, done = memory[i]

        flat_state_new = np.reshape(state_new, [1,8])
        flat_state = np.reshape(state, [1,8])

        target = np.zeros((1, 1))
        advantages = np.zeros((1, 4))

        value = critic.predict(flat_state)
        next_value = critic.predict(flat_state_new)

        if done:
            advantages[0][action] = reward - value
            target[0][0] = reward
        else:
            advantages[0][action] = reward + gamma * (next_value) - value
            target[0][0] = reward + gamma * next_value

        actor.fit(flat_state, advantages, epochs=1, verbose=0)
        critic.fit(flat_state, target, epochs=1, verbose=0)
```

Performance

The performance was far from ideal, after 700 episode, the model was not improving. As all RL models, uncertainty is prevalent. Especially in this case, we need to find the perfect balance between two models in order to make it perform. And we simply ran out of time and computing power.

```
ALR: 2e-06  CLR: 9e-05 episode  658 of 2000 Average Reward (last 100 eps)=  -195.52599301400278
ALR: 2e-06  CLR: 9e-05 episode  659 of 2000 Average Reward (last 100 eps)=  -191.9333021155691
ALR: 2e-06  CLR: 9e-05 episode  660 of 2000 Average Reward (last 100 eps)=  -191.75499694754973
ALR: 2e-06  CLR: 9e-05 episode  661 of 2000 Average Reward (last 100 eps)=  -193.12594209924998
ALR: 2e-06  CLR: 9e-05 episode  662 of 2000 Average Reward (last 100 eps)=  -191.84445102953157
ALR: 2e-06  CLR: 9e-05 episode  663 of 2000 Average Reward (last 100 eps)=  -190.44990034709605
ALR: 2e-06  CLR: 9e-05 episode  664 of 2000 Average Reward (last 100 eps)=  -191.47033097885253
ALR: 2e-06  CLR: 9e-05 episode  665 of 2000 Average Reward (last 100 eps)=  -191.43658763918566
ALR: 2e-06  CLR: 9e-05 episode  666 of 2000 Average Reward (last 100 eps)=  -189.55633828918744
ALR: 2e-06  CLR: 9e-05 episode  667 of 2000 Average Reward (last 100 eps)=  -187.75440749070046
ALR: 2e-06  CLR: 9e-05 episode  668 of 2000 Average Reward (last 100 eps)=  -188.0288119590154
ALR: 2e-06  CLR: 9e-05 episode  669 of 2000 Average Reward (last 100 eps)=  -188.9932788684279
ALR: 2e-06  CLR: 9e-05 episode  670 of 2000 Average Reward (last 100 eps)=  -190.62176962248495
ALR: 2e-06  CLR: 9e-05 episode  671 of 2000 Average Reward (last 100 eps)=  -190.397969598374
ALR: 2e-06  CLR: 9e-05 episode  672 of 2000 Average Reward (last 100 eps)=  -190.69511110511445
ALR: 2e-06  CLR: 9e-05 episode  673 of 2000 Average Reward (last 100 eps)=  -190.01140096953898
ALR: 2e-06  CLR: 9e-05 episode  674 of 2000 Average Reward (last 100 eps)=  -189.75008438812725
ALR: 2e-06  CLR: 9e-05 episode  675 of 2000 Average Reward (last 100 eps)=  -194.1611820653968
ALR: 2e-06  CLR: 9e-05 episode  676 of 2000 Average Reward (last 100 eps)=  -193.45029528705695
ALR: 2e-06  CLR: 9e-05 episode  677 of 2000 Average Reward (last 100 eps)=  -197.12702651911684
ALR: 2e-06  CLR: 9e-05 episode  678 of 2000 Average Reward (last 100 eps)=  -197.2498457418017
ALR: 2e-06  CLR: 9e-05 episode  679 of 2000 Average Reward (last 100 eps)=  -196.56136256900194
ALR: 2e-06  CLR: 9e-05 episode  680 of 2000 Average Reward (last 100 eps)=  -197.35531800569493
ALR: 2e-06  CLR: 9e-05 episode  681 of 2000 Average Reward (last 100 eps)=  -196.0454408507716
ALR: 2e-06  CLR: 9e-05 episode  682 of 2000 Average Reward (last 100 eps)=  -195.6432167234526
ALR: 2e-06  CLR: 9e-05 episode  683 of 2000 Average Reward (last 100 eps)=  -195.13780059505496
ALR: 2e-06  CLR: 9e-05 episode  684 of 2000 Average Reward (last 100 eps)=  -193.67506448293042
ALR: 2e-06  CLR: 9e-05 episode  685 of 2000 Average Reward (last 100 eps)=  -193.14369150785996
ALR: 2e-06  CLR: 9e-05 episode  686 of 2000 Average Reward (last 100 eps)=  -194.73156345327865
ALR: 2e-06  CLR: 9e-05 episode  687 of 2000 Average Reward (last 100 eps)=  -193.52972566680097
ALR: 2e-06  CLR: 9e-05 episode  688 of 2000 Average Reward (last 100 eps)=  -194.88857263882718
ALR: 2e-06  CLR: 9e-05 episode  689 of 2000 Average Reward (last 100 eps)=  -194.4114778952176
ALR: 2e-06  CLR: 9e-05 episode  690 of 2000 Average Reward (last 100 eps)=  -193.01414867592723
ALR: 2e-06  CLR: 9e-05 episode  691 of 2000 Average Reward (last 100 eps)=  -193.11469268558065
ALR: 2e-06  CLR: 9e-05 episode  692 of 2000 Average Reward (last 100 eps)=  -190.98922799647502
ALR: 2e-06  CLR: 9e-05 episode  693 of 2000 Average Reward (last 100 eps)=  -192.82072964139775
```

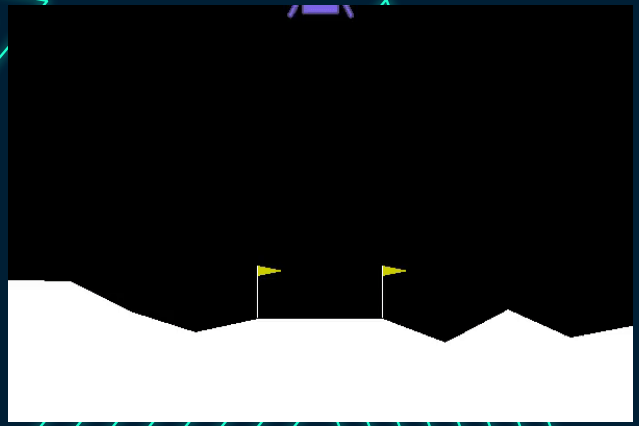
Conclusion

- Relatively, we can see that the different Q-Networks are more consistent in meeting the solved requirements.

- DQN was also found to be overestimating over a single state, hence our decision to improve to DDQN and D3QN

Further Improvement

- Due to time constraint, we could not fully implement the A2C network.
- We could try to implement more RL algorithm such as **Proximal Policy Optimization (PPO)**, **Variational DQN**, or even Google's **Evolving Reinforcement Learning Algorithm**
- We could also try different hyperparameters such as **Learning Rate**, γ Discount Factor, **Batch Size**
- We could also try to change the Loss Function to Huber Loss to minimise the effect of outliers.



Thanks!