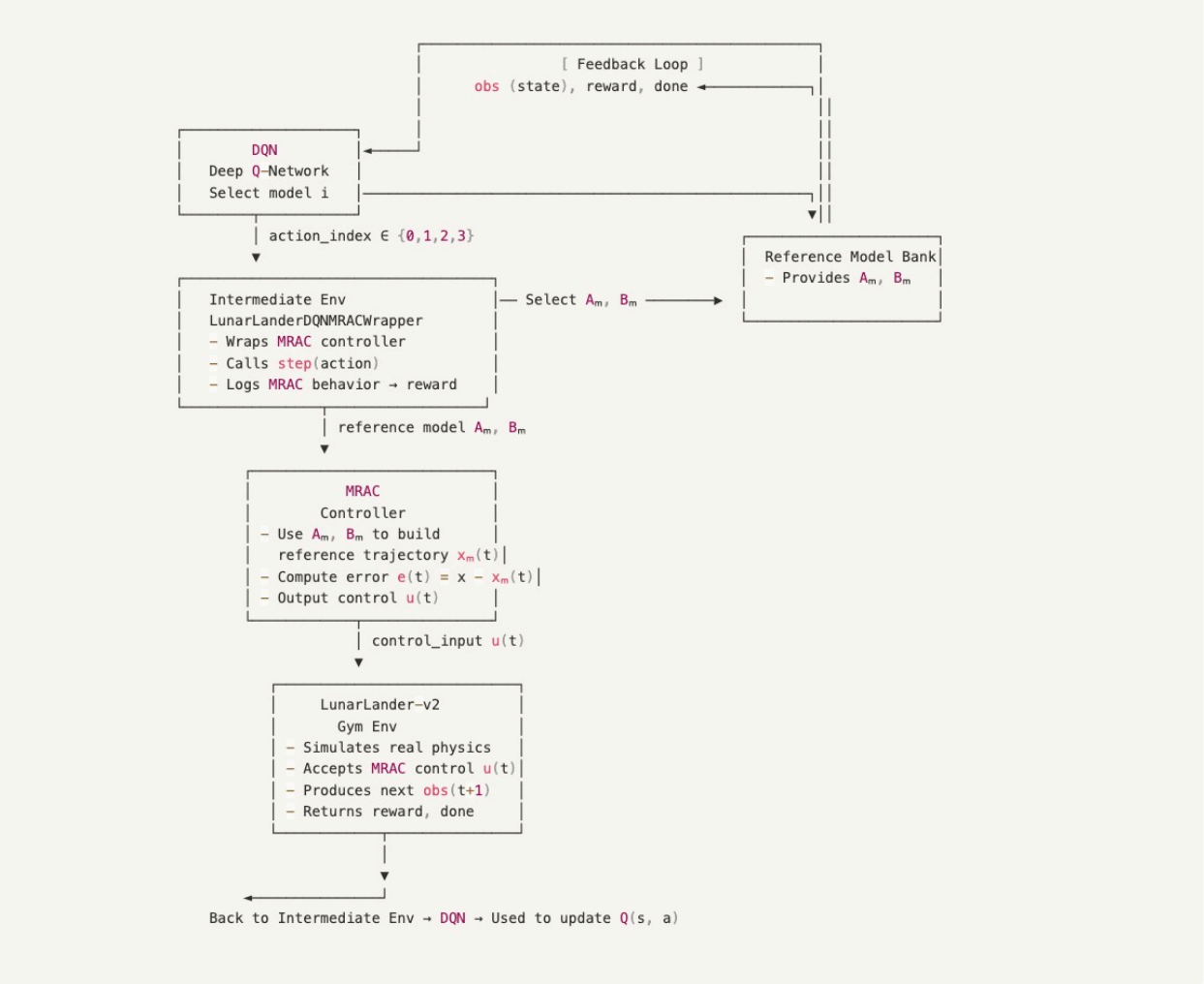
Report: DQN-Augmented Discrete-Time MRAC for Lunar Lander Control

1. Introduction and Problem Statement

The goal is to design a control strategy that combines Model Reference Adaptive Control (MRAC) with Deep Q-Network (DQN) reinforcement learning for Landing a lunar module (as in LunarLander-v3).

MRAC will provide real-time adaptive control to track a desired trajectory or model, while DQN will supply high-level decision-making for optimal action selection or mode switching. This approach leverages the strengths of both optimal control and adaptive control: DQN learns a near-optimal policy for the nominal lander dynamics, and MRAC continuously adapts the control input to account for modeling errors or uncertainties, ensuring stable and accurate tracking.

Objectives of this proposal are: (1)Maintain stability and precise control during landing via MRAC's adaptive law.(2) Achieve optimal performance (fast, fuel-efficient landing) via DQN's learned policy. (3)Ensure Lyapunov stability in discrete time.



2. Environment Description: Lunar Lander Simulation (LunarLander-v2)

Its discrete-time control setting with 2D lunar landing scenario.

2.1 State and Action Spaces

- **Observation Space**: The environment provides an 8-dimensional continuous state vector, representing:
  - Position: (x, y) — coordinates relative to landing pad
  - Velocity: (v_x, v_y) — linear velocities in x and y directions
  - Orientation: θ (theta) — lander's angle

    - Angular Velocity: ω (omega) — rate of rotation
    - Contact Sensors: Two boolean variables indicating whether the left and right legs are in contact with the ground.
- **Action Space** (Discrete Control Modes):
    - Mode 0: No thrust (coasting/falling).
    - Mode 1: Fire left thruster (counterclockwise rotation).
    - Mode 2: Fire main engine (upward thrust).
    - Mode 3: Fire right thruster (clockwise rotation).

2.2 Reward Structure

The **reward function** encourages safe and efficient landings:

- Proximity & Velocity: Reward increases when the lander moves closer to the pad and slows down.
- Stability Penalty: A tilt away from a horizontal position incurs a penalty.
- Landing Points: +10 points per leg ground contact
- Fuel Economy:
    - Side thrusters: -0.03 points/timestep
    - Main engine: -0.3 points/timestep
- Mission Outcomes:
    - Success: +100 points for safe landing
    - Crash: -100 points for crash
- Victory Condition: ≥ 200 points total score required

2.3 Environment Dynamics and Constraints

- Initial Conditions: Random position and velocity at viewport top
- Physics Parameters: g = -10.0 m/s^2, with optional wind effects
- Terminal States:

1. Crash: The lander body contacts the ground outside of its legs.
2. Out of Bounds: The x-coordinate exceeds the viewport.
3. Lander Stability: If the lander stops moving, the simulation ends.

3. Theoretical Foundations

3.1 Model Reference Adaptive Control in Discrete Time

MRAC (Model Reference Adaptive Control) is a control strategy where the lander is commanded to follow a desired behavior given by a reference model despite uncertainties in dynamics. At each discrete time-step k, MRAC adjusts the control input based on the tracking error `e(k) = x(k) - x_ref(k)`, x is the lander's state (like position, velocity, angle, etc) and x_ref is the reference model state.

3.1.1 Reference Model for Discrete-Time System Dynamics:

$$x_r(k+1) = A_r x_r(k) + B_r u_r(k);$$

- where u_r(k) represents the reference input, and $u_r = \pi(x_r)$ denotes the policy used for reinforcement learning training.[*formular 7,page 5,'Integration of AC and RL for RealTime Control and Learning']*
- The matrices A_r, B_r ensure stability and define the desired system behavior

The reference model $x_{\text{ref}}(k)$ is target landing trajectory.

```
class MRACReferenceModel:
    def __init__(self, A_m, B_m):
        self.A_m = np.array(A_m)
```

```
        self.B_m = np.array(B_m)
        self.x_m = np.zeros((self.A_m.shape[0],))

    def reset(self, init=None):
        self.x_m = np.zeros_like(self.x_m) if init is None else init

    def step(self, r):
        self.x_m = self.A_m @ self.x_m + self.B_m @ r
        return self.x_m.copy()
```

3.1.2 Control Law (with Adaptive Term):

To make the actual state $x(k)$ track $x_{\text{ref}}(k)$, a discrete-time control law with an adaptive term is,

$$u(k) = Kx(k) + \theta^T(k)\Psi(x(k))$$

where

- Kx(k) is the baseline controller(pre-designed linear feedback),K is a fixed baseline gain.
- $\theta^T(k)\psi(x(k))$ is Adaptive Compensation Term(unmodeled dynamics.).
- $\Theta(k)$ is a vector of adaptive parameters with $\Phi(x(k))$ by using a **Lyapunov-based update rule** to ensure stability: $\Theta(k+1) = \Theta(k) - \Gamma\Psi(x(k))e(k)$, where Γ is the learning rate and $e(k) = x(k) - x_r(k)$ represents the tracking error.
- adaptive parameters → make tracking error toward zero.

```
class MRACController:
    def __init__(self, state_dim, phi_dim, K=None, Gamma=None):
        self.K = np.array(K) if K is not None else np.zeros((1, state_dim))
        self.Gamma = np.eye(phi_dim) * 0.01 if Gamma is None else Gamma
        self.Theta = np.zeros((phi_dim, 1))
```

```
    def phi(self, x):
        return np.concatenate([x, x**2, np.tanh(x)])

    def get_control(self, x):
        x = np.array(x).reshape(-1)
        phi_x = self.phi(x).reshape(-1, 1)
        u = self.K @ x.reshape(-1, 1) + self.Theta.T @ phi_x
        return u.item(), phi_x

    def update(self, phi_x, error):
        error = np.array([[error]])
        self.Theta = self.Theta - self.Gamma @ phi_x @ error
```

3.1.3 Adaptation Law for Weight Updates(ensure convergence)

$$\Theta(k+1) = \Theta(k) - \Gamma\Phi(x(k))e(k)$$

- This is the update rule for adaptive weights in a discrete-time adaptive control system.
- Γ is a positive definite gain matrix controlling the adaptation rate.
- φ(x(k)) is regression vector.
- $e(k) = x(k) - x_{\text{ref}}(k)$ is the tracking error.

This adjusts Θ in the direction that reduces the error,and makes stability if Γ is satisfied with the discrete-time Lyapunov conditions.

```
            # MRAC control
        u, phi_x = self.controller.get_control(x)
        # Update adaptive parameters
        self.controller.update(phi_x, e.mean())
```

3.2 Stability:

A key feature of MRAC is its ability to assure closed-loop stability despite unknown parameters. Suppose choosing a discrete-time Lyapunov function V(k) = x_1^2+x_2^2, we can prove that V(k) decreses amd bounded at each step.

For example,in adaptive neural controller implementations, research has shown that with appropriate control law and update rules, the system's tracking error is bounded (converges to a small neighborhood of zero) when adaptation gains meet specific conditions.[*'Theorem 1: If the dynamics of the vehicle can be described by (7)...' ,page 1024, 'Adaptive Neural Network Control of AUVs With Control Input Nonlinearities Using Reinforcement Learning'*]

Our design constructs a discrete Lyapunov function V(k) that either decreases at each step or remains bounded, ensuring uniform boundedness (UB) of the closed-loop signals. In summary, the MRAC component guides the lander's state x(k) to follow the reference model x_ref(k) asymptotically, maintaining stability and control accuracy even with uncertainties in the lander's mass, gravity, or engine thrust.

3.3 Deep Q-Network (DQN) and Reinforcement Learning for Optimal Control

3.3.1 DQN (Deep Q-Network) is a model-free reinforcement learning algorithm that learns an optimal policy:

$$\pi^* : \mathcal{S} \mapsto \mathcal{A}$$

It approximates the optimal action-value function $Q^*(s, a)$ using a deep neural network:

$$Q^*(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right]$$

where,it represents the maximum expected cumulative reward from state s taking action a and following the optimal policy, $\gamma \in (0,1)$ is the discount factor.

The optimal policy is then derived as:

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

```
def select_action(self, state, return_index=False):
        state = torch.FloatTensor(state).unsqueeze(0).to(self.device)
        if np.random.rand() < self.epsilon:
            index = np.random.randint(self.action_dim)
        else:
            with torch.no_grad():
                q_values = self.q_net(state)
            index = q_values.argmax().item()

        if return_index:
            return index
        else:
            # action ( 0~3),  model_index = action
            return index
```

DQN Training:

The DQN uses a parameterized Q-network $Q_\theta(s,a)$ to estimate Q-values and is trained by minimizing the Bellman error over experience tuples (s,a,r,s'):

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\left(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a)\right)^2\right]$$

where (s,a,r,s') represents state-action-reward-nextstate samples from the lander's experience.After doing the iterative updates,the DQN converges to the

optimal Q-function, learning the precise timing for thruster activation to minimize fuel consumption and landing error.

The loss function $L(\theta) = (r + \gamma \max_{a'} Q_{\theta^-}(s', a') - Q_\theta(s, a))^2$, where Q_{θ^-} is the target network.

This training procedure, combined with techniques like experience, leads $Q_\theta(s, a)$ to converge to $Q^*(s, a)$, learning the optimal policy for the lander (e.g. when to fire each thruster during landing).

```
for t in range(max_timesteps):
            #action = self.select_action(state)
            model_index = self.select_action(state, return_index=True)  # use DQN
            env.set_reference_model(model_index)  # update MRAC reference mod

            action = model_index  # if action=reference index, you can map to disc

            next_state, reward, terminated, truncated, info = env.step(action)
            done = terminated or truncated
            self.store_transition(state, action, reward, next_state, float(done))
            self.update()

            state = next_state
            total_reward += reward

            if done or truncated:
                break
```

```
def update(self):
        if len(self.replay_buffer) < self.batch_size:
            return

        states, actions, rewards, next_states, dones = self.replay_buffer.sample(self.
```

```
states = torch.FloatTensor(states).to(self.device)
actions = torch.LongTensor(actions).unsqueeze(1).to(self.device)
rewards = torch.FloatTensor(rewards).unsqueeze(1).to(self.device)
next_states = torch.FloatTensor(next_states).to(self.device)
dones = torch.FloatTensor(dones).unsqueeze(1).to(self.device)

# Compute current Q values
curr_q = self.q_net(states).gather(1, actions)

# Compute target Q values with N-step returns
with torch.no_grad():
    next_q = self.target_net(next_states).max(1, keepdim=True)[0]
    target_q = rewards + (self.gamma ** self.n_step) * (1 - dones) * next_q

# Compute loss
loss = nn.MSELoss()(curr_q, target_q)
# Optimize
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()

# Update target network
self.update_count += 1
if self.update_count % self.target_update_freq == 0:
    self.target_net.load_state_dict(self.q_net.state_dict())

# Decay epsilon
self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)
```

4. Integration of DQN and MRAC.

Use the DQN and MRAC.

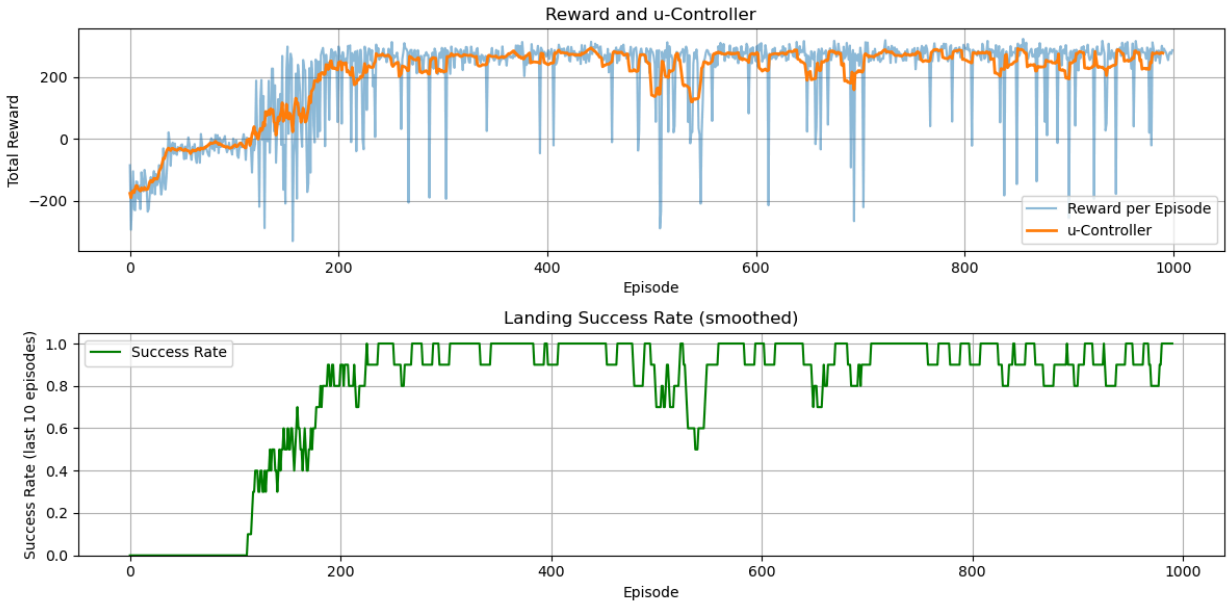
The DQN Policy Selection:

- The DQN observes the current environment state (e.g., position, velocity, angle) and selects a discrete reference model index that corresponds to a desired landing behavior.
- It learns a discrete-time policy $\pi^*(s)$ that selects from multiple reference dynamics (A_m, B_m) model to maximize long-term reward based on landing performance and stability.

MRAC Adaptive Tracking Controller:

- The MRAC controller receives the selected reference model (A_m, B_m) and generates a reference trajectory $x_m(t)$ based on it.
- It calculates the tracking error $e(t) = x(t) - x_m(t)$ and adjusts the control law to reduce this error.
- It produces a discrete time control signal $u(t)$ that guides the lander to follow the desired dynamics while maintaining robustness against model uncertainties and external disturbances.

After training for 1,000 epochs, the following plot shows the success rate and the relationship between the controller and rewards.



The training and testing videos are attached in the videos folder.

5. Conclusion

This shows that integrates Deep Q-Network (DQN) reinforcement learning with Model Reference Adaptive Control (MRAC) can solve the lunar landing problem in a discrete-time control setting. The DQN component learns to select optimal reference models based on environmental observations,it allows decision-making over adaptive dynamics. The MRAC controller, in turn, uses these reference models to generate real-time control inputs that ensure the lander tracks the desired trajectory accurately and robustly even under disturbances.