CART-POLE SYSTEM

1. Yeduru Rupasree - 2020101097
2. Mudireddy Nandini Reddy - 2020101038
3. Tharigopula Manaswini - 2020113015
4. Mandyam Brunda - 2020101057
5. Rohan Madineni - 2020102066

Contribution

DQN - Nandini and Manaswini

DDPG - Rohan Madineni and Rupa

PPO - Rupa and Brunda

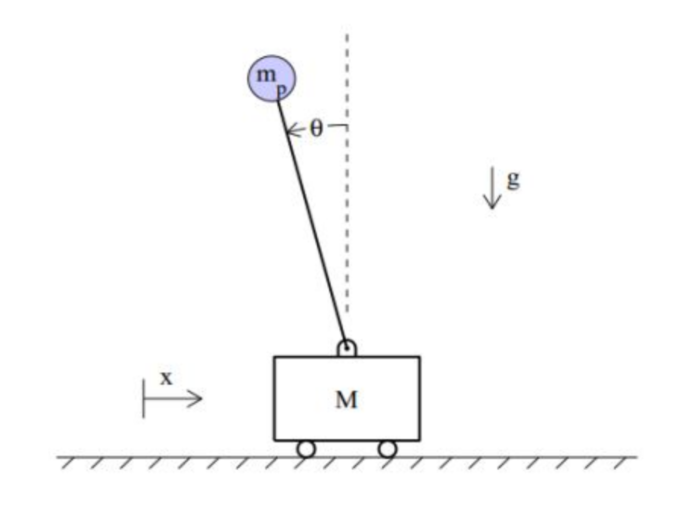
SAC - Nandini, Manaswini, Brunda

Problem Statement

The dynamics of the cart-pole system shown in figure given below. Here M and mp is the mass (kg) of the cart and pole respectively. The linear displacement (in m) of the cart is denoted by x, g is the acceleration due to gravity, θ is the angular displacement (radians) of the pole of length L (in m) and Fx is the input force applied to the cart (in N). Find the optimal control input (Fx) that takes the pole from the initial angular position i) θ(0) = π/3 , ii) θ(0) = π/6 , iii) θ(0) = π/2 to the desired angular position θ(tf) = 0? (Here, the final time tf is a free variable.) M= 40 Kg, mp= 2 kg, L = 0.75 m.

$$\ddot{\theta} = \frac{-m_p L \sin\theta \cos\theta \dot{\theta}^2 + (M + m_p)\, g \sin\theta + \cos\theta F_x}{\left(M + m_p \left(1 - \cos^2\theta\right)\right) L}$$

$$\ddot{x} = \frac{-m_p L \sin\theta \dot{\theta}^2 + m_p g \sin\theta \cos\theta + F_x}{M + m_p \left(1 - \cos^2\theta\right)}$$



Terminologies

State Variables:

- $x(t)$: Linear displacement of the cart (m) at time t.
- $\dot{x}(t)$: Velocity of the cart (m/s) at time t.
- $\theta(t)$: Angular displacement of the pole (radians) at time t.
- $\dot{\theta}(t)$: Angular velocity of the pole (radians/s) at time t.

Control Input:

- $F_x(t)$: Horizontal force applied to the cart (N) at time t.

Objective:

The objective is to find a control policy for $F_x(t)$ that balances the pole in the upright position ($\theta(t) = 0$) for as long as possible. This can be formulated as maximizing the total reward received over time.

Reward:

$$-\theta^2 - \dot{\theta}^2$$

DDPG

Algorithm

Pseudocode

Algorithm 1 Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
    Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
    Execute a in the environment
    Observe next state s', reward r, and done signal d to indicate whether s' is terminal
    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
    If s' is terminal, reset environment state.
    if it's time to update **then**
        for however many updates **do**
            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
            Compute targets

$$y(r, s', d) = r + \gamma(1 - d) Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

            Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d) \right)^2$$

            Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

            Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho \phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho \theta_{\text{targ}} + (1 - \rho)\theta$$

        end for
    end if
until convergence

Critic Network

```python
class CriticNetwork(nn.Module):
    def __init__(self, beta, input_dims, fc1_dims, fc2_dims, n_actions, name):
        super(CriticNetwork, self).__init__()
        self.input_dims = input_dims
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions

        self.fc1 = nn.Linear(*self.input_dims, self.fc1_dims)
        f1 = 1./np.sqrt(self.fc1.weight.data.size()[0])
        T.nn.init.uniform_(self.fc1.weight.data, -f1, f1)
        T.nn.init.uniform_(self.fc1.bias.data, -f1, f1)
        self.bn1 = nn.LayerNorm(self.fc1_dims)

        self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
        f2 = 1./np.sqrt(self.fc2.weight.data.size()[0])
        T.nn.init.uniform_(self.fc2.weight.data, -f2, f2)
        T.nn.init.uniform_(self.fc2.bias.data, -f2, f2)
        self.bn2 = nn.LayerNorm(self.fc2_dims)

        self.action_value = nn.Linear(self.n_actions, self.fc2_dims)
        f3 = 0.003
        self.q = nn.Linear(self.fc2_dims, 1)
        T.nn.init.uniform_(self.q.weight.data, -f3, f3)
        T.nn.init.uniform_(self.q.bias.data, -f3, f3)

        self.optimizer = optim.Adam(self.parameters(), lr=beta)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')

        self.to(self.device)
```

```python
def forward(self, state, action):
    state_value = self.fc1(state)
    state_value = self.bn1(state_value)
    state_value = F.relu(state_value)
    state_value = self.fc2(state_value)
    state_value = self.bn2(state_value)

    action_value = F.relu(self.action_value(action))
    state_action_value = F.relu(T.add(state_value, action_value))
    state_action_value = self.q(state_action_value)

    return state_action_value
```

Hyper-Parameters

Hyper Parameters of CriticNetwork

- Learning_rate = 0.002
- Layer1_size = 30
- Layer2_size = 30

Actor Network

```python
class ActorNetwork(nn.Module):
    def __init__(self, alpha, input_dims, fc1_dims, fc2_dims, n_actions, name):
        super(ActorNetwork, self).__init__()
        self.input_dims = input_dims
        self.fc1_dims = fc1_dims
        self.fc2_dims = fc2_dims
        self.n_actions = n_actions

        self.fc1 = nn.Linear(*self.input_dims, self.fc1_dims)
        f1 = 1./np.sqrt(self.fc1.weight.data.size()[0])
        T.nn.init.uniform_(self.fc1.weight.data, -f1, f1)
        T.nn.init.uniform_(self.fc1.bias.data, -f1, f1)
        self.bn1 = nn.LayerNorm(self.fc1_dims)

        self.fc2 = nn.Linear(self.fc1_dims, self.fc2_dims)
        f2 = 1./np.sqrt(self.fc2.weight.data.size()[0])
        T.nn.init.uniform_(self.fc2.weight.data, -f2, f2)
        T.nn.init.uniform_(self.fc2.bias.data, -f2, f2)
        self.bn2 = nn.LayerNorm(self.fc2_dims)

        f3 = 0.003
        self.mu = nn.Linear(self.fc2_dims, self.n_actions)
        T.nn.init.uniform_(self.mu.weight.data, -f3, f3)
        T.nn.init.uniform_(self.mu.bias.data, -f3, f3)
        self.optimizer = optim.Adam(self.parameters(), lr=alpha)
        self.device = T.device('cuda:0' if T.cuda.is_available() else 'cpu')

        self.to(self.device)
```

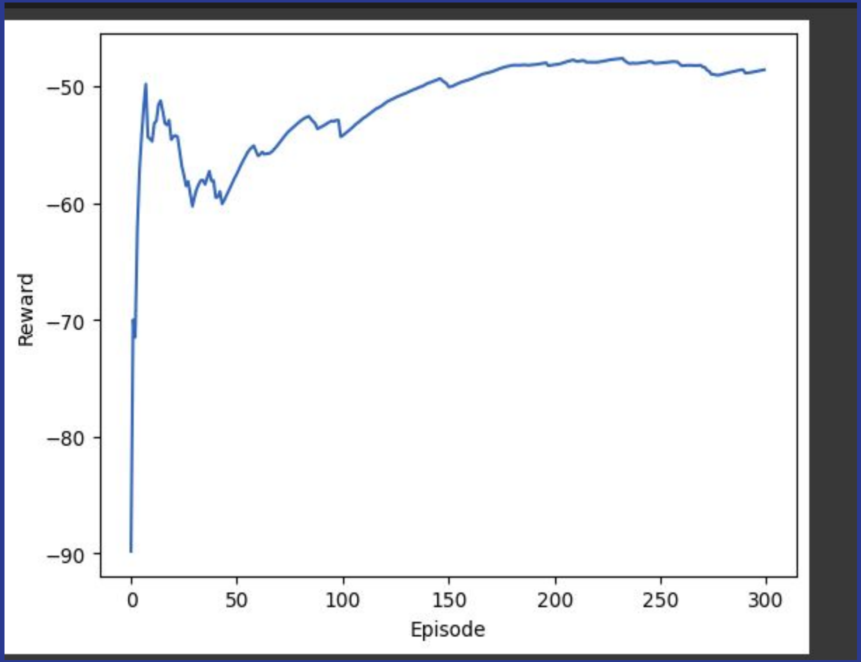
```python
def forward(self, state):
    x = self.fc1(state)
    x = self.bn1(x)
    x = F.relu(x)
    x = self.fc2(x)
    x = self.bn2(x)
    x = F.relu(x)
    x = T.tanh(self.mu(x))

    return x
```

Hyper-Parameters

Hyper Parameters of ActorNetwork

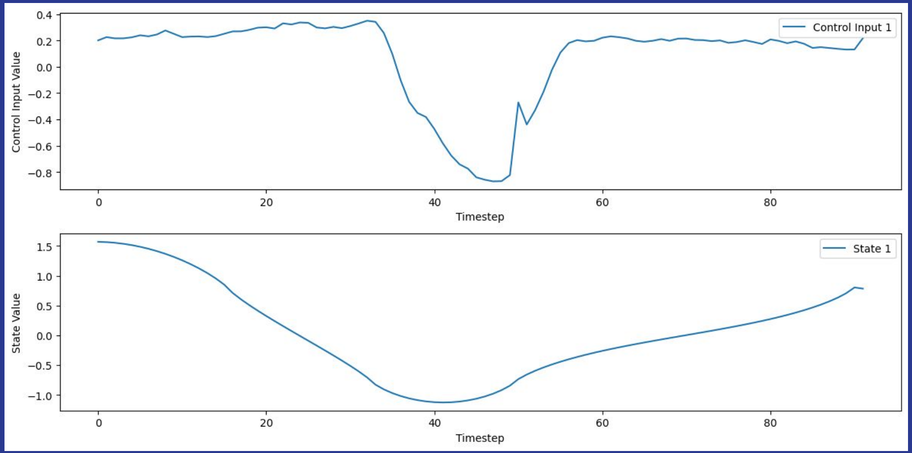
- Learning_rate = 0.001
- Layer1_size = 30
- Layer2_size = 30

Average reward vs episode

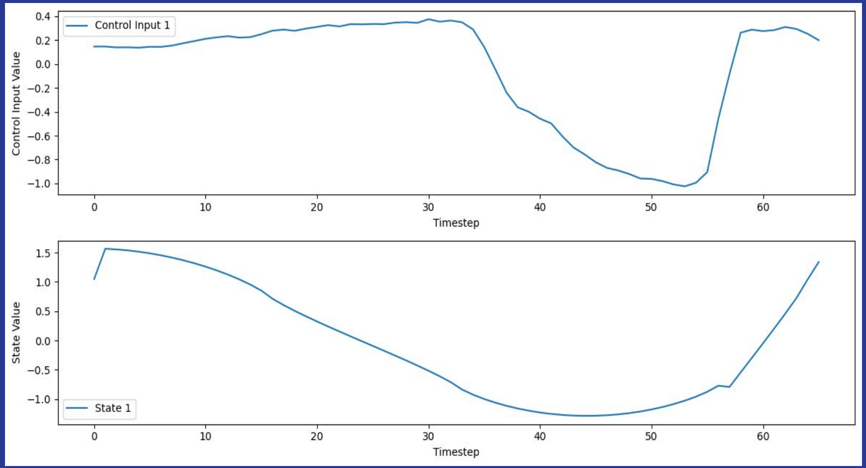


Optimal Input and state Trajectory

theta = pi/2

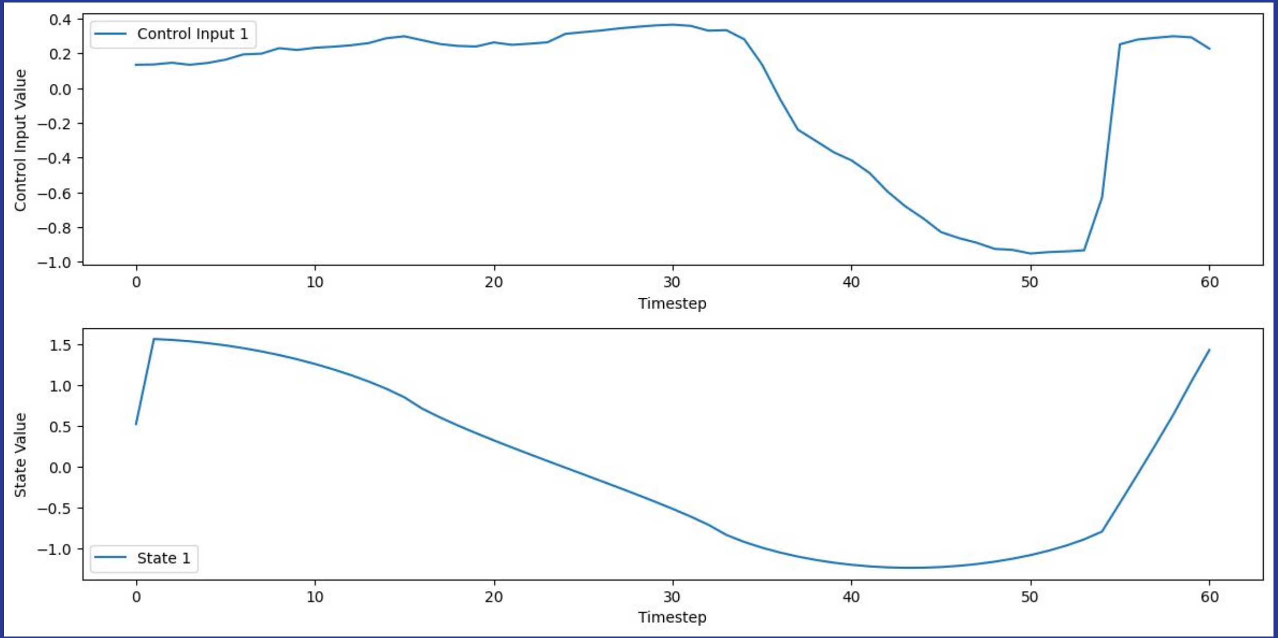


theta = pi/3



Optimal Input and state Trajectory[cont]

theta = pi/6



PPO

Algorithm

Pseudocode

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0

2: **for** $k = 0, 1, 2, ...$ **do**

3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.

4: Compute rewards-to-go $\hat{R}_t$.

5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.

6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \;\; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k| T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

8: **end for**

Policy Network

```python
from torch import nn
class PolicyNetwork(nn.Module):
    def __init__(self, obs_space_size, action_space_size):
        super().__init__()

        self.shared_layers = nn.Sequential(
            nn.Linear(obs_space_size, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU()
        )

        self.policy_mean = nn.Sequential(
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, action_space_size)
        )

        self.policy_logstd = nn.Parameter(torch.zeros(1, action_space_size))

    def forward(self, obs):
        z = self.shared_layers(obs)
        mean = self.policy_mean(z)
        std = torch.exp(self.policy_logstd).to(device)
        return mean, std
```

Value Network

```python
class ValueNetwork(nn.Module):
    def __init__(self, obs_space_size):
        super().__init__()

        self.shared_layers = nn.Sequential(
            nn.Linear(obs_space_size, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU()
        )

        self.value_layers = nn.Sequential(
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )

    def forward(self, obs):
        z = self.shared_layers(obs)
        value = self.value_layers(z)
        return value
```

Hyper-Parameters

Hyper Parameters of Policy Network:

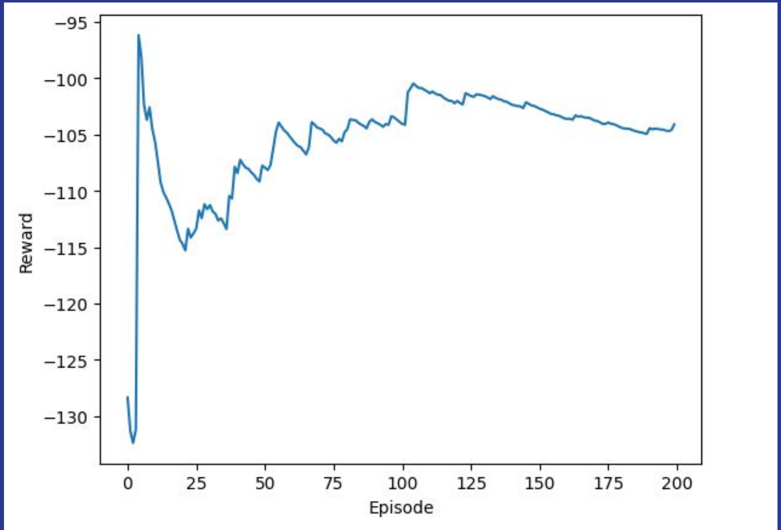
- Policy_lr = 3e-4
- Max_policy_train_iters = 40
- Hidden_layer_dim =64

Hyper Parameters of Value Network:

- Value_lr = 1e-3
- value_train_iters = 40
- hidden_layer_dim= 64

ppo_clip_val =0.2

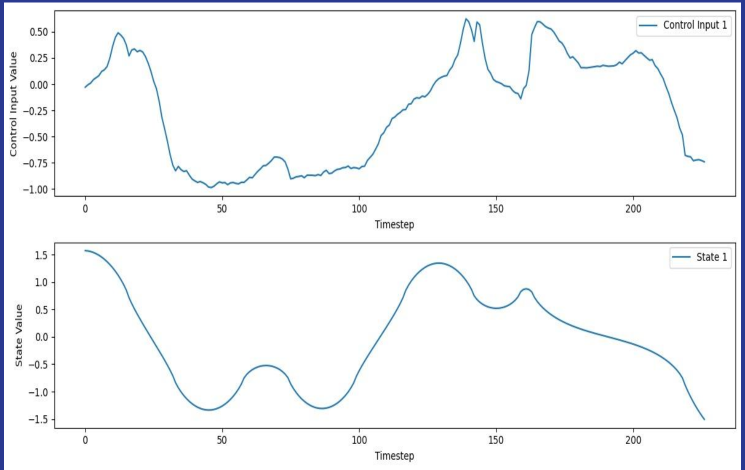
Average reward vs episode



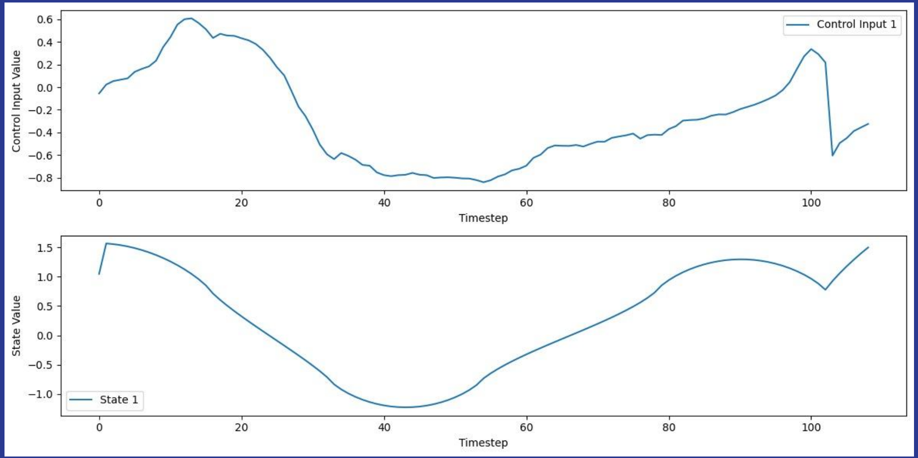
Reward is increasing which indicates that learning is happening

Optimal Input and state Trajectory

theta = pi/2

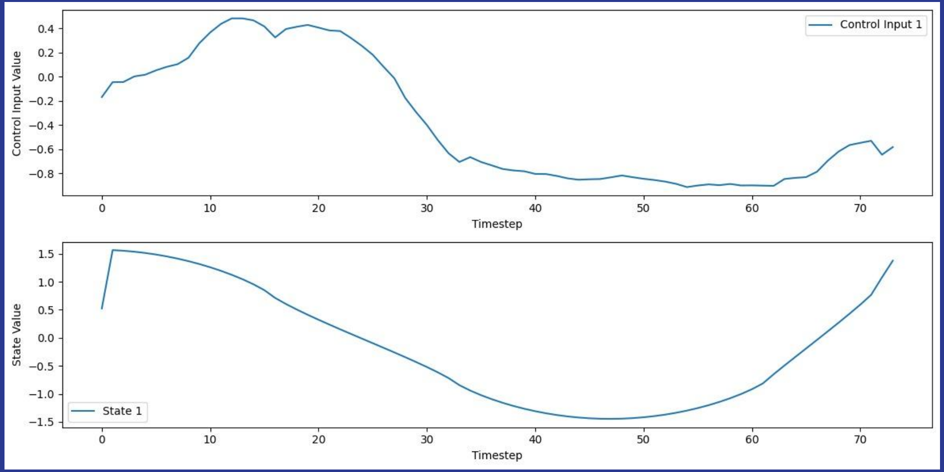


theta = pi/3



Optimal Input and state Trajectory

theta = pi/6



SAC

Algorithm

Pseudocode

Algorithm 1 Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for j in range(however many updates) **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\right),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.
      Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho\phi_{\text{targ},i} + (1 - \rho)\phi_i \qquad \text{for } i = 1, 2$$

    end for
  end if
until convergence

Policy Network

```
class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, actor_lr):
        super(PolicyNetwork, self).__init__()

        self.fc_1 = nn.Linear(state_dim, 128)
        self.fc_2 = nn.Linear(128,128)
        self.fc_mu = nn.Linear(128, action_dim)
        self.fc_std = nn.Linear(128, action_dim)

        self.lr = actor_lr

        self.LOG_STD_MIN = -20
        self.LOG_STD_MAX = 2
        self.max_action = 2
        self.min_action = -2
        self.action_scale = (self.max_action - self.min_action) / 2.0
        self.action_bias = (self.max_action + self.min_action) / 2.0

        self.optimizer = optim.Adam(self.parameters(), lr=self.lr)

    def forward(self, x):

        x = F.leaky_relu(self.fc_1(x))
        x = F.leaky_relu(self.fc_2(x))
        mu = self.fc_mu(x)
        log_std = self.fc_std(x)
        log_std = torch.clamp(log_std, self.LOG_STD_MIN, self.LOG_STD_MAX)
        return mu, log_std
```

Q Network

```
class QNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, critic_lr):
        super(QNetwork, self).__init__()

        self.fc_s = nn.Linear(state_dim, 64)
        self.fc_a = nn.Linear(action_dim, 64)
        self.fc_1 = nn.Linear(128, 128)
        self.fc_out = nn.Linear(128, action_dim)

        self.lr = critic_lr

        self.optimizer = optim.Adam(self.parameters(), lr=self.lr)

    def forward(self, x, a):
        h1 = F.leaky_relu(self.fc_s(x))
        h2 = F.leaky_relu(self.fc_a(a))
        cat = torch.cat([h1, h2], dim=-1)
        q = F.leaky_relu(self.fc_1(cat))
        q = self.fc_out(q)
        return q
```

Hyper-Parameters

Hyper Parameters of Policy Network:

- lr_pi = 0.03
- hidden_layer_dim = 128

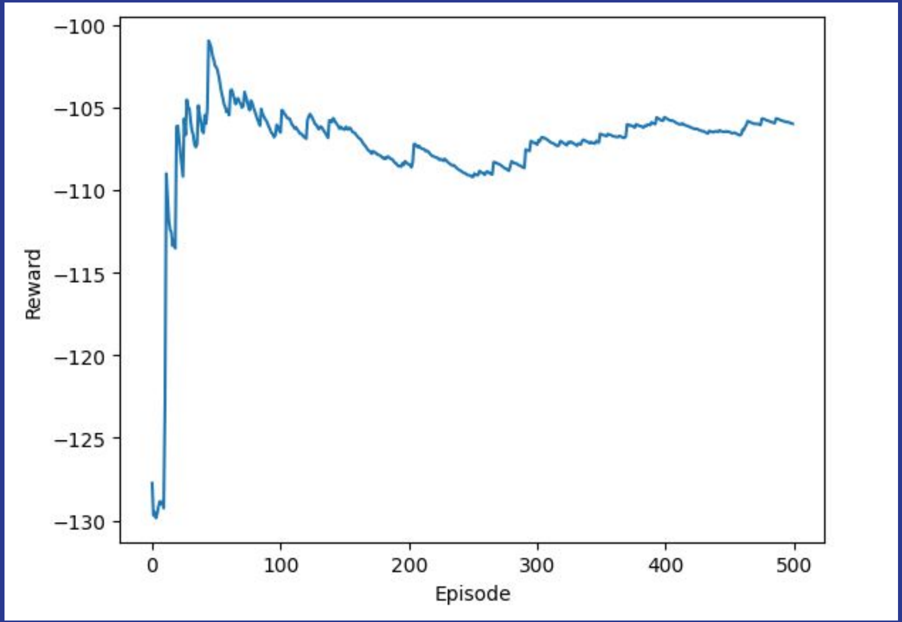
Hyper Parameters of Q Network:

- Lr_q = 0.03
- hidden_layer_dim = 128

Gamma = 0.999

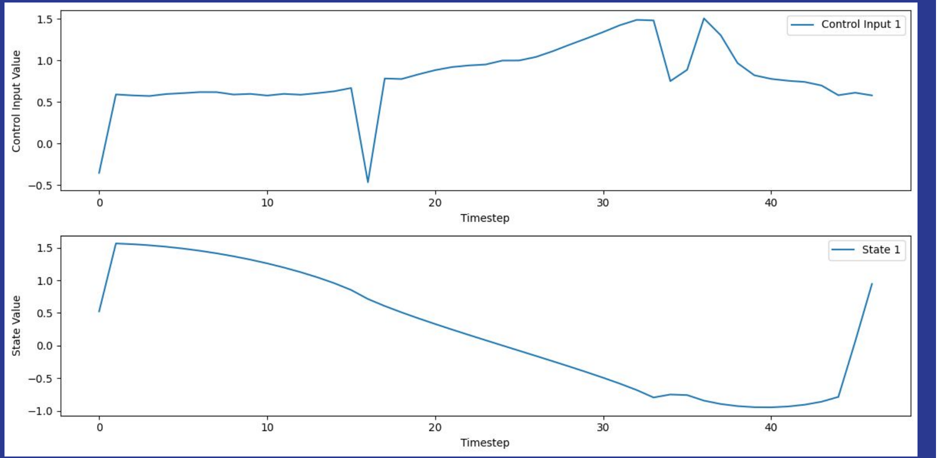
Tau = 0.005

Average reward vs episode

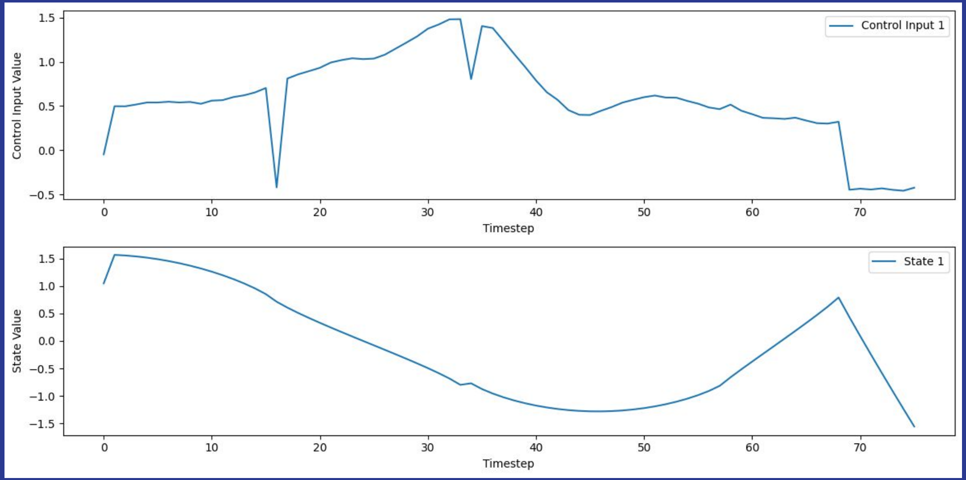


Optimal Input and state Trajectory

theta = pi/6

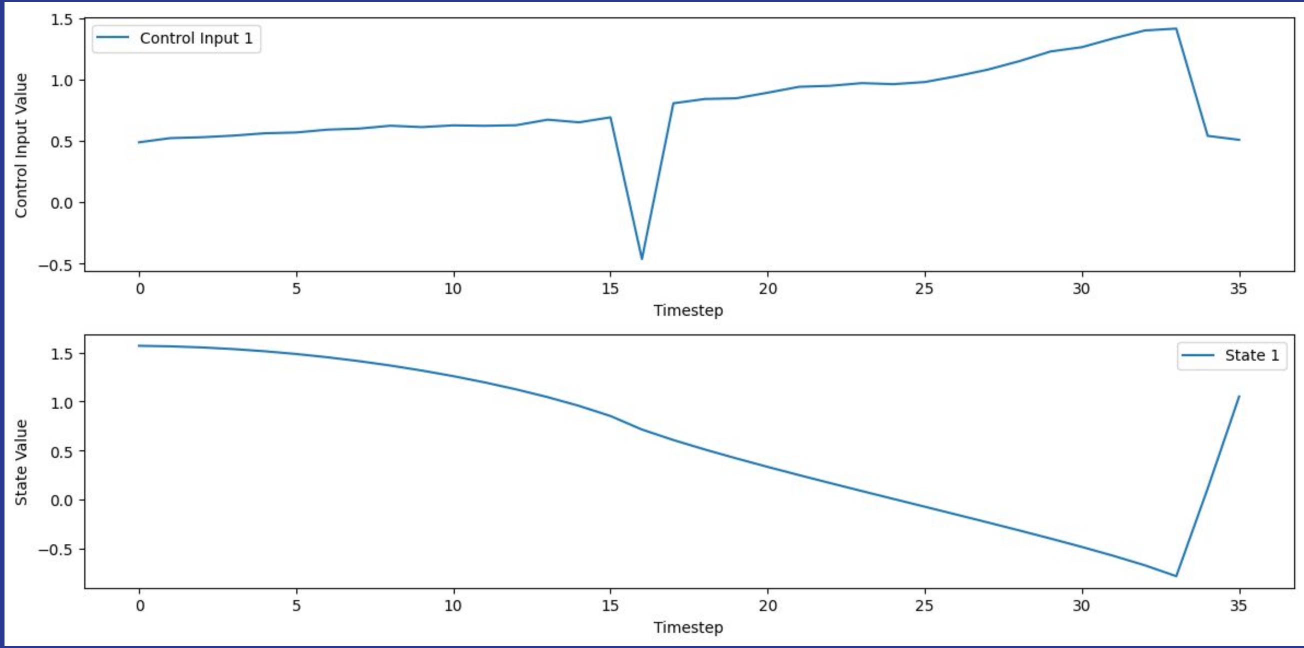


theta = pi/3



Optimal Input and state Trajectory

theta = pi/2



Observation

- We can see that from the above graphs, the state(angle of the pole with vertical) is going to zero and oscillating around that.
- Since the angles 60, 30. 90 degrees are large , maybe the oscillation range is somewhat graeter.

Deep Q-Learning (DQN)

QNetwork

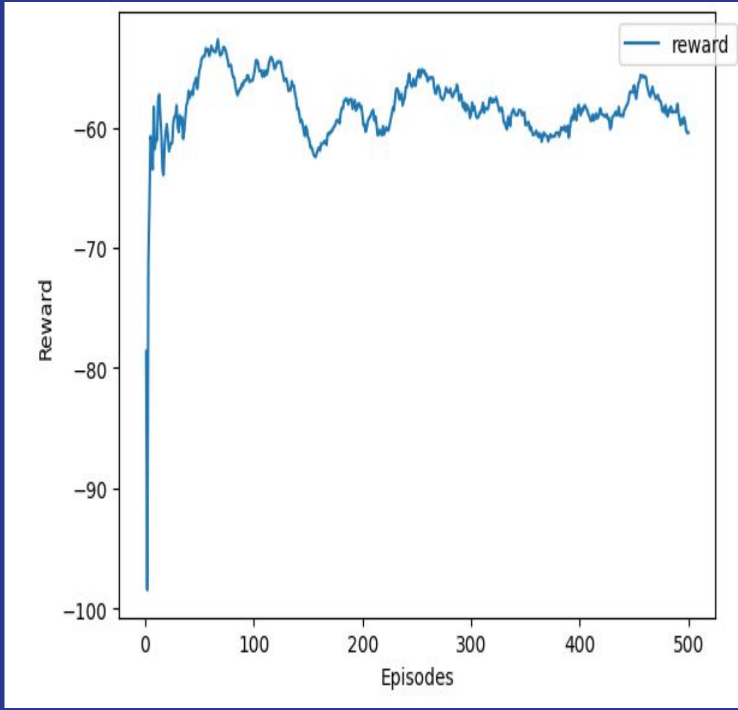
```python
class QNetwork(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim) -> None:
        """DQN Network

        Args:
            input_dim (int): `state` dimension.
                `state` is 2-D tensor of shape (n, input_dim)
            output_dim (int): Number of actions.
                Q_value is 2-D tensor of shape (n, output_dim)
            hidden_dim (int): Hidden dimension in fc layer
        """

        super(QNetwork, self).__init__()

        self.layer1 = torch.nn.Sequential(
            torch.nn.Linear(input_dim, hidden_dim),
#            torch.nn.BatchNorm1d(hidden_dim),
            torch.nn.PReLU()
        )

        self.layer2 = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.PReLU()
        )

        self.layer3 = torch.nn.Sequential(
            torch.nn.Linear(hidden_dim, hidden_dim),
            torch.nn.PReLU()
        )

        self.final = torch.nn.Linear(hidden_dim, output_dim)
```

```python
def forward(self, x: torch.Tensor) -> torch.Tensor:
    """Returns a Q_value

    Args:
        x (torch.Tensor): `State` 2-D tensor of shape (n, input_dim)

    Returns:
        torch.Tensor: Q_value, 2-D tensor of shape (n, output_dim)
    """

    ## print('type(x) of forward:', type(x))
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    x = self.final(x)

    return x
```

Hyper-Parameters

- BATCH_SIZE = 64
- TAU = 0.005
- Gamma = 0.99
- Learning_rate = 0.001
- TARGET_UPDATE = 10
- Hidden_dim = 16

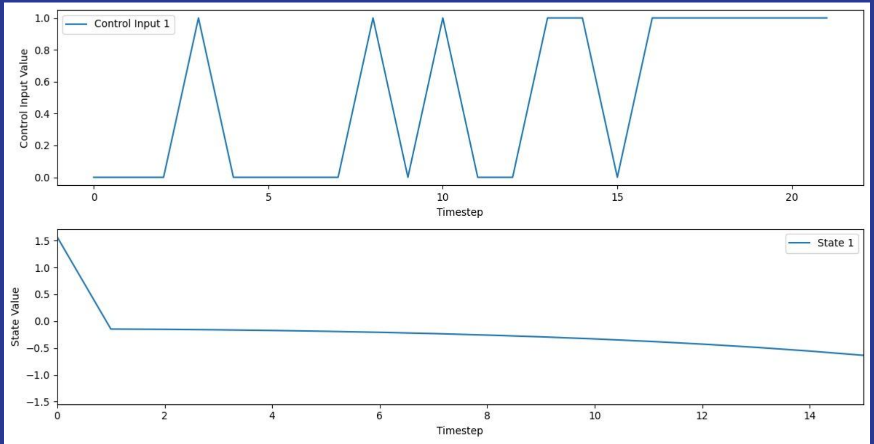
Average reward vs episode



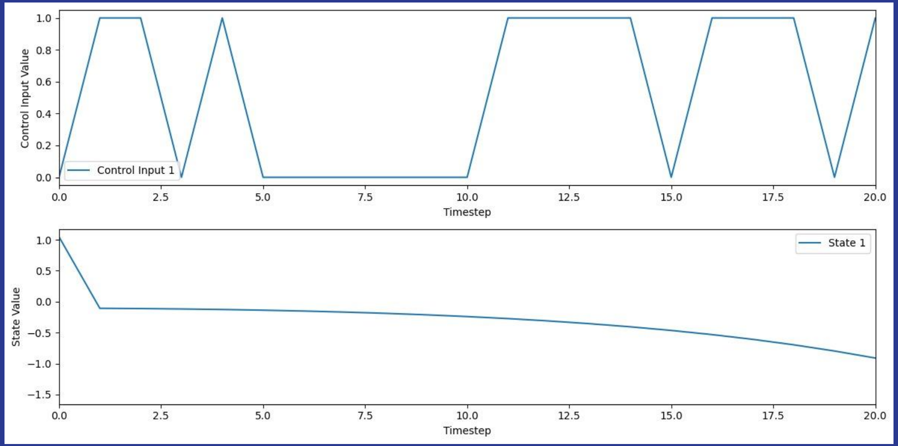
This graph denotes average reward vs number of episodes.
We can see learning is happening till 100 episodes using DQN algorithm.

Optimal Input and state Trajectory

theta = pi/2



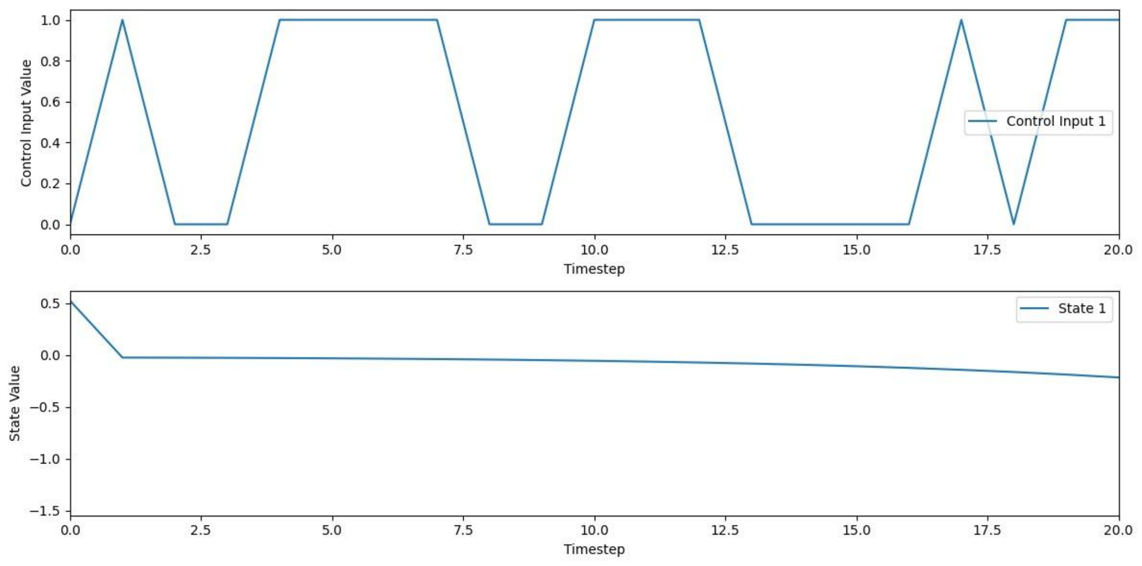
theta = pi/3



In control input vs timetep graph, 0 on the y-axis represents force with magnitude 1 to the left and 1 represents force with magnitude 1 to the right

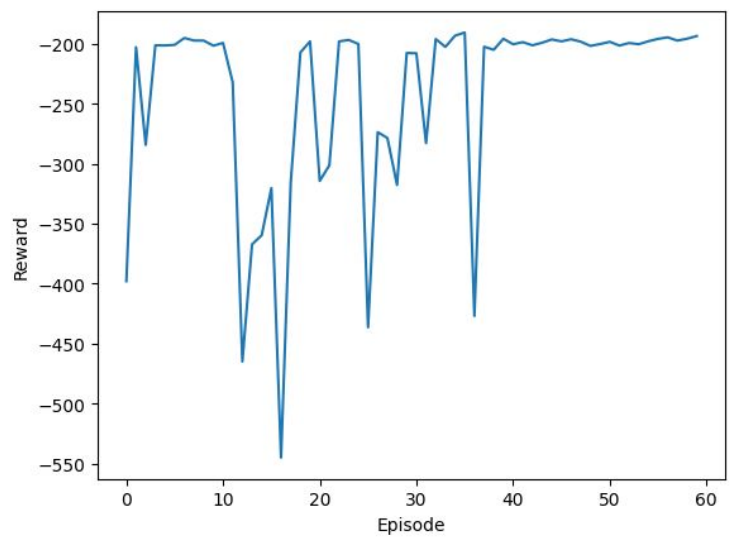
Optimal Input and state Trajectory

theta = pi/6



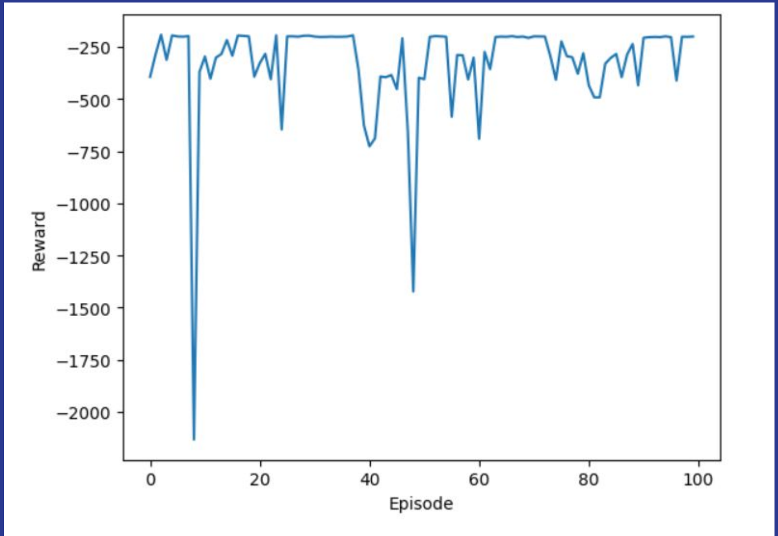
Experiments

Average reward vs episode [DQN]



- This graph shows average reward vs number of episodes when the reward funcion is taken as -|θ| where θ is the angle made by the pole with the vertical
- We can see that the learning is not happening here and this may be because of the poor choice of the reward function

Average reward vs episode [DDPG]



- This is the graph of average reward vs number of episodes for ddpg algorithm when the learning rate is taken too low.
- This is indicating that the learning of the model is not happening within the first 100 episodes

Thank you