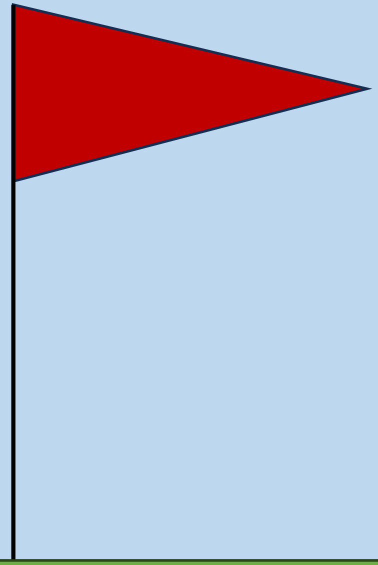
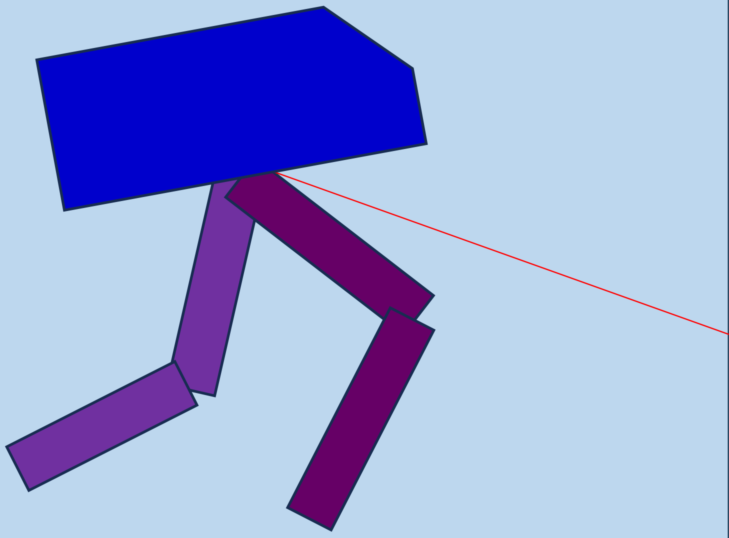


Di-Gait-Tron

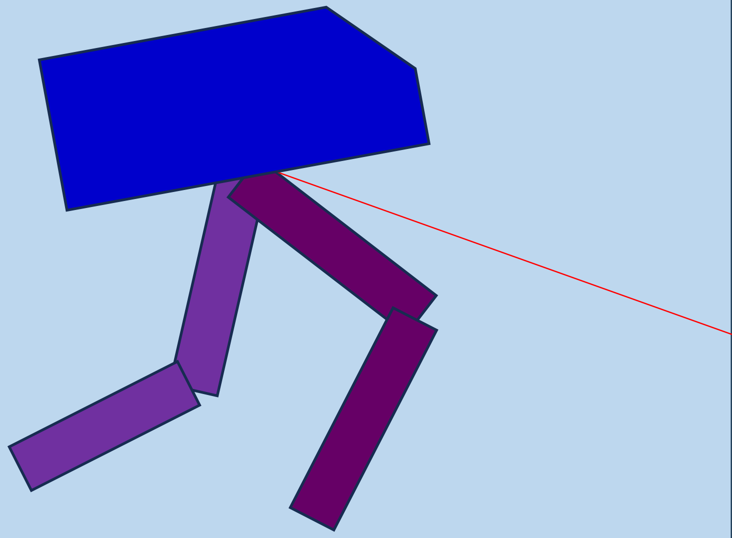
Bipedal Walker with DeepRL algorithm



Biswajit Rana
B2330026

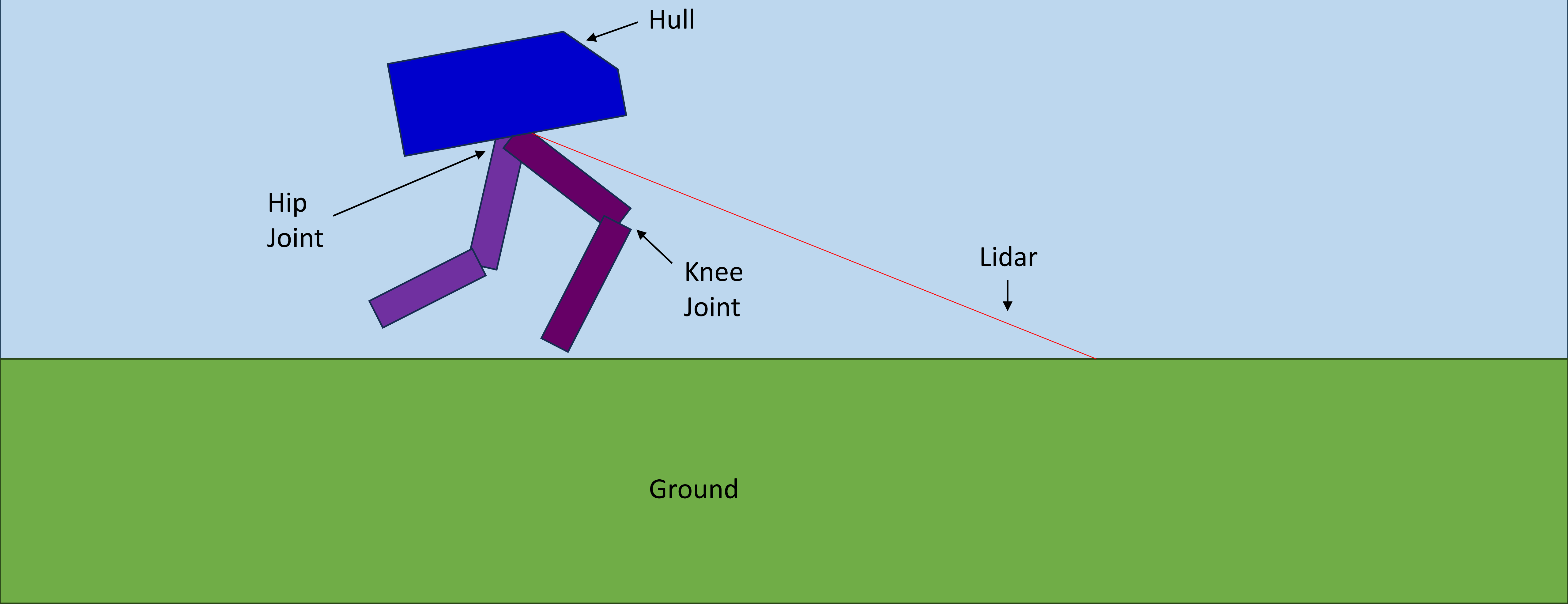
Debayan Datta
B2330027

Details of the Agent and Environment



Details of the Agent and Environment :

- **Description:** This is a simple 4-joint walker robot environment. There are two versions:
  - Normal, with slightly uneven terrain.
  - Hardcore, with ladders, stumps, pitfalls.

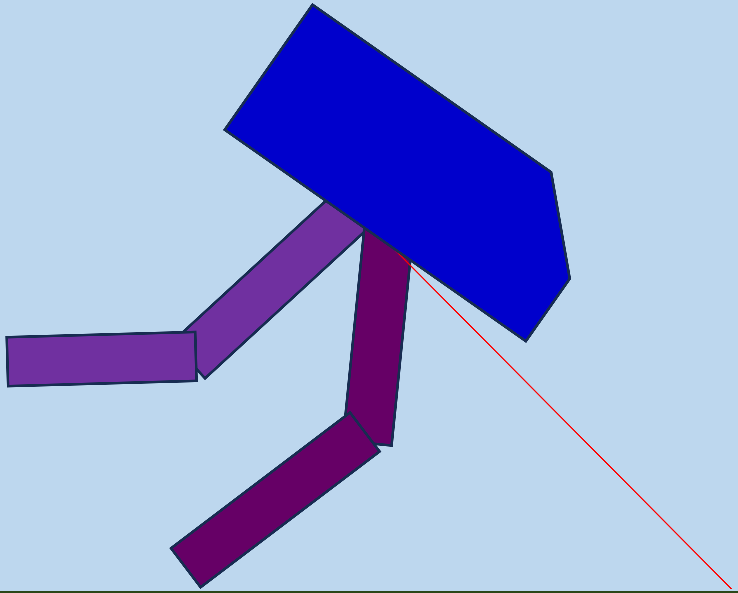


Details of the Agent and Environment :

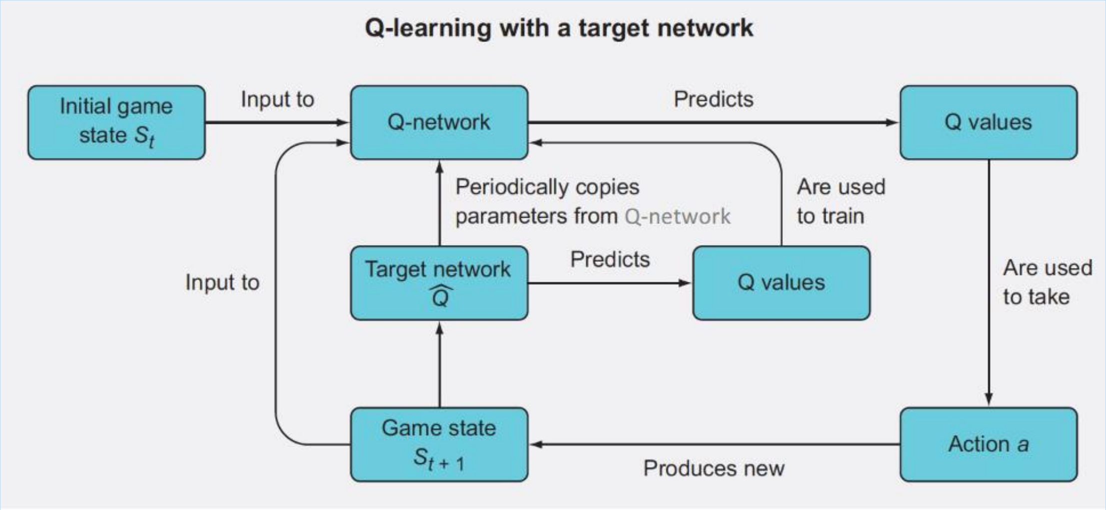
- **Action Space:** Actions are motor speed values in the [-1, 1] range for each of the 4 joints at both hips and knees.
- **Observation Space:** State consists of *hull angle speed*, *angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 lidar rangefinder measurements*. There are no coordinates in the state vector.
- **Rewards:** Reward is given for moving forward, totaling 300+ points up to the far end. If the robot falls, it gets -100. Applying motor torque costs a small amount of points. A more optimal agent will get a better score.
- **Starting State:** The walker starts standing at the left end of the terrain with the hull horizontal, and both legs in the same position with a slight knee angle.
- **Episode Termination:** The episode will terminate if the hull gets in contact with the ground or if the walker exceeds the right end of the terrain length.

$$\text{Observation space} = [\underbrace{-3.14\ -5.}_{\textit{hull angle speed, angular velocity}}\ \underbrace{-5.\ -5.}_{\textit{horizontal speed, vertical speed}}\ \underbrace{-3.14\ -5.\ -3.14\ -5.\ -0.\ -3.14\ -5.\ -3.14}_{\textit{position of joints and joints angular speed}}\ \underbrace{1.\ 0.}_{\textit{legs contact with ground}}\ \underbrace{-1.\ -1.\ -1.\ -1.\ -1.\ -1.\ -1.\ -1.}_{\textit{10 lidar rangefinder measurements}}\ ],$$

ALGORITHMS



Deep Q-Learning :

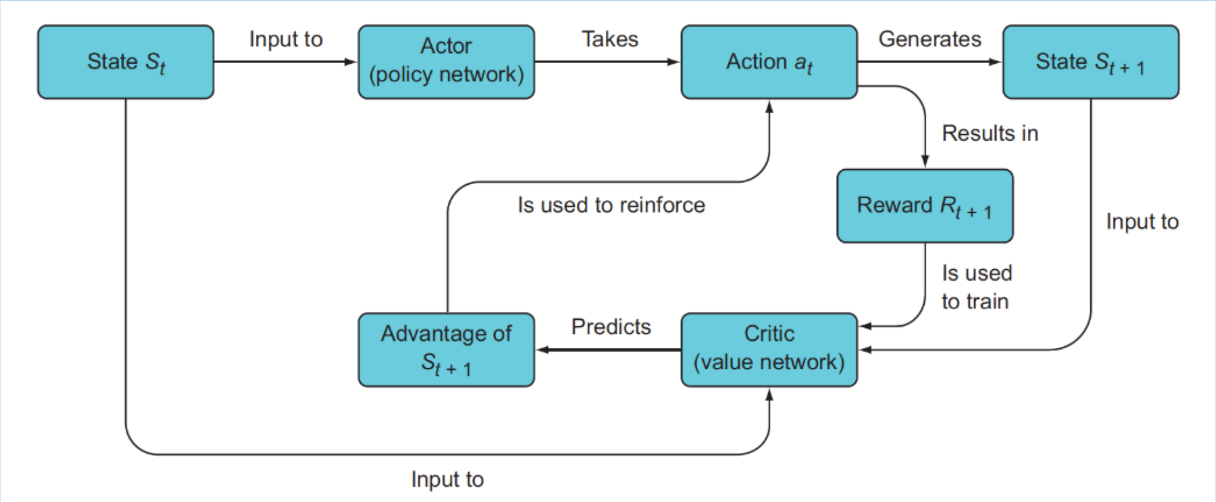


Algorithm 1: deep Q-learning with experience replay. and target network

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Paper : Human-level control through deep reinforcement learning

Deep Deterministic Policy Gradient :



Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

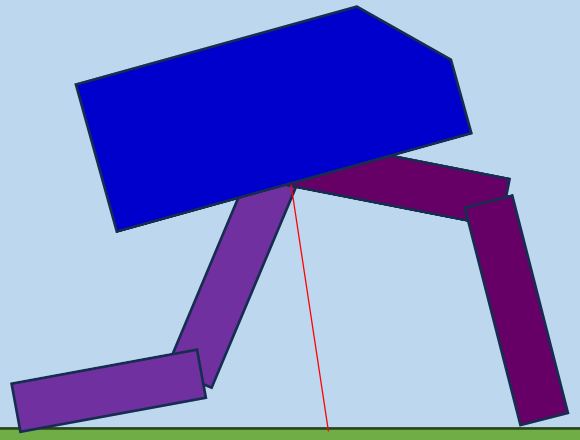
    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Paper : CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING

IMPLEMENTATION



- Using Pytorch to implement the algorithm.
- Making separate classes for actor network ,critic network ,replay buffer

```python
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(state_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, action_dim),
            nn.Tanh()
        )
        self.max_action = max_action

    def forward(self, x):
        return self.max_action * self.network(x)
```

```python
# Critic Network
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(state_dim + action_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 256),
            nn.ReLU(),
            nn.Linear(256, 1)
        )

    def forward(self, state, action):
        return self.network(torch.cat([state, action], dim=1))
```

```python
#soft updates
def update_targets(self):
    for param, target_param in zip(self.actor.parameters(), self.target_actor.parameters()):
        target_param.data.copy_(self.tau * param.data + (1 - self.tau) * target_param.data)
    for param, target_param in zip(self.critic.parameters(), self.target_critic.parameters()):
        target_param.data.copy_(self.tau * param.data + (1 - self.tau) * target_param.data)
```

```python
self.memory = deque(maxlen=100000)
self.gamma = 0.99
self.tau = 0.005
batch_size=64
noise=0.1
```

Github : https://github.com/biswajit-github-2022/bipedal-walker-with-ddpg

Thank You