CSE 676 Deep Learning

FINAL PROJECT

On

FOREST FIRE MITIGATION

Using MARL (Multi-Agent Reinforcement Learning)

University at Buffalo
Information Technology

By:
Group - 10
Tanwin Chowdary (tanwinch)
Kumar Raja Chidella (kumarraj)
Veekshith Kajeepeta (vkajeepe)

Project Description

- Objective: Develop an efficient solution for managing forest fires by utilizing a reinforcement learning based approach.
- Create a grid-based simulation environment that accurately models the dynamics of forest fires and allows for the simulation of various scenarios.
- Use Multiple agents, which can be firefighters and drones, will be deployed to move around the grid and take appropriate actions.
- By training RL algorithms, we aim to enable these agents to learn optimal firefighting strategies tailored to different situations
- Ultimately, minimize the damage caused by forest fires, safeguard lives, and enhance overall firefighting efficiency.

Background

- In today's world, there is a worldwide problem of Wild Forest Fires and how to deal with it
- Need for mitigation of problem
- Replicate similar conditions in a test environment
- Integrate Reinforcement Learning Techniques to solve environment.
- Previous works show implementation of Forest Fire detection using CNN algorithms and Image classification.

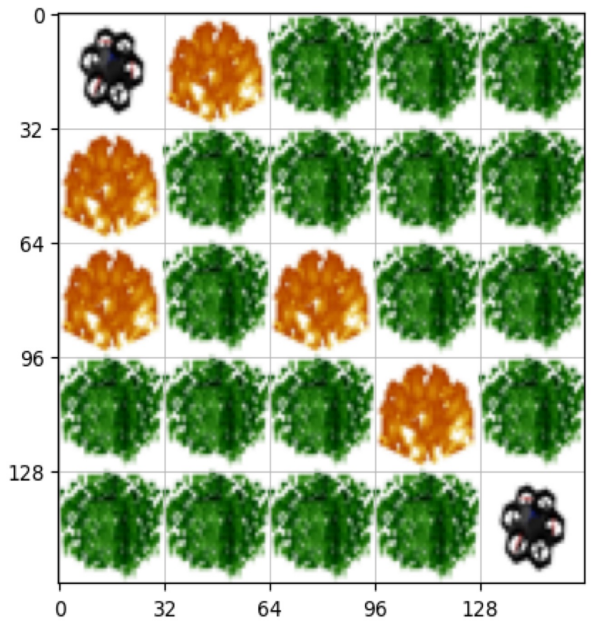
About the Environment

- Existing environments such as pendulum and CartPole from gym.
- Observation Space/Grid Dimensions: n*n
- Actions: Move Up/Down/Left/Right
- Rewards:

a)-0.1 for visiting a cell with no fire.

b)+2.0 for visiting a cell with fire and dousing it.

c)+5.0 for dousing all cells with fire.



- Has Stochastic and Deterministic variability
- Stochastic Environment

a)Random Position Fires.

b)Fires Spread with Random Probability.

- Deterministic Environment

a)Fire at Static Positions.

b)No spread of Fire.

- - Q-table with grid position where particular agent is in state.
- - Q-table with values of all grid positions in state.

Methods

- The following methods were used to enhance forest fire mitigation task.
- Proximal Policy Optimisation (PPO)
- Soft Actor Critic (SAC)
- Deep Deterministic Policy Gradient (DDPG)
- Deep Q-Networks (DQN)
- Dueling Deep Q-Networks (DDQN)

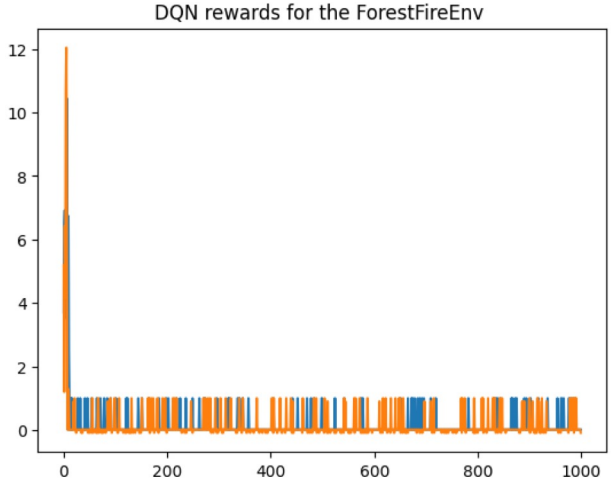
DQN

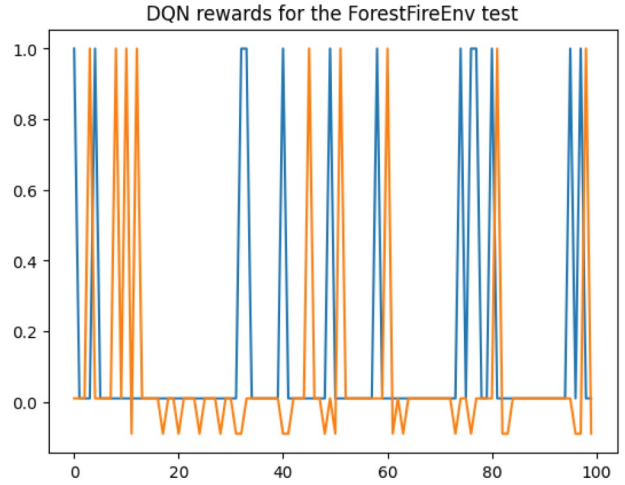
- DQNAgent is a class that represents an agent in the DQN algorithm.
- The agent has a neural network model that predicts the Q-values of each possible action for a given state.
- The agent uses an experience replay buffer to store experiences (state, action, reward, next_state, done) during training.
- The agent chooses an action using an epsilon-greedy policy, where it selects a random action with probability epsilon and selects the action with the highest Q-value with probability (1-epsilon).
- The agent trains its neural network by sampling a random batch of experiences from the replay buffer and updating the model's parameters using the Bellman equation.
- The agent updates the target network(which has the same architecture as the model but with frozen parameters, to estimate the Q-values used in the Bellman equation.) every update_frequency=100 steps.
- The code trains two DQN agents on the ForestFireEnv environment

DQN

```
[ ]
import torch
import torch.nn as nn
import torch.optim as optim
import random
from collections import deque

class DQNAgent:
    def __init__(self, state_size, action_size, device, lr=0.001, gamma=0.99, buffer_size=10000, batch_size=64):
        self.state_size = state_size
        self.action_size = action_size
        self.device = device
        self.lr = lr
        self.gamma = gamma
        self.batch_size = batch_size

        self.memory = deque(maxlen=buffer_size)
        self.model = self._build_model().to(device)
        self.target_model = self._build_model().to(device)
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.lr)
        self.update_target_model()

    def _build_model(self):
        model = nn.Sequential(
            nn.Linear(self.state_size, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, self.action_size)
        )
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def choose_action(self, state, epsilon):
        if random.random() < epsilon:
            return random.randrange(self.action_size)
        else:
            state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
            return self.model(state).argmax().item()

    def update_target_model(self):
        self.target_model.load_state_dict(self.model.state_dict())

    def train(self):
        if len(self.memory) < self.batch_size:
            return

        minibatch = random.sample(self.memory, self.batch_size)

        states, actions, rewards, next_states, dones = zip(*minibatch)
        states = torch.tensor(states, dtype=torch.float32, device=self.device)
        actions = torch.tensor(actions, dtype=torch.long, device=self.device).unsqueeze(1)
        rewards = torch.tensor(rewards, dtype=torch.float32, device=self.device)
        next_states = torch.tensor(next_states, dtype=torch.float32, device=self.device)
        dones = torch.tensor(dones, dtype=torch.float32, device=self.device)

        current_q_values = self.model(states).gather(1, actions)
        next_q_values = self.target_model(next_states).max(1)[0].detach()
        target_q_values = rewards + self.gamma * next_q_values * (1 - dones)

        loss = nn.MSELoss()(current_q_values, target_q_values.unsqueeze(1))
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

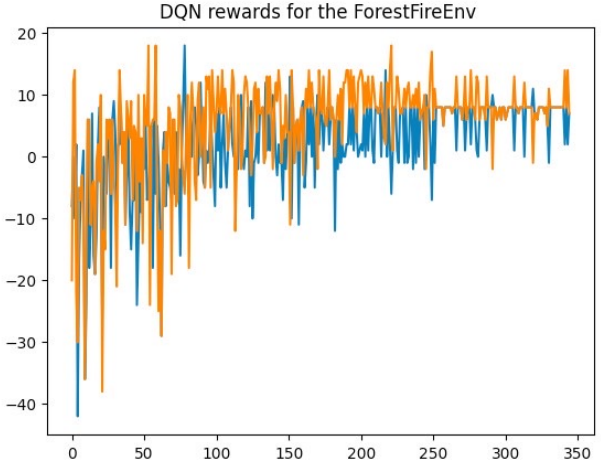
DQN stochastic v/s determinstic





Due to the volatility of the environment defined the results of the DQN implemented are no as expected with max timesteps of 1000 on a 6x6 grid env the results are [-0.1,-0.1] for the agents.

And because of the training handling the volatile env the test results are fluctuating for the stochastic env for the 100 episodes.



Due to the stability of the environment defined the results of the DQN implemented are quite impressive with max timesteps of 100 on a 4x4 grid env the results are stabilizing at [8, 8] for the agents.

And because of the training handling the stable env the test results are constant for the static env for the 10 episodes. Results are achieved in 4 timesteps.

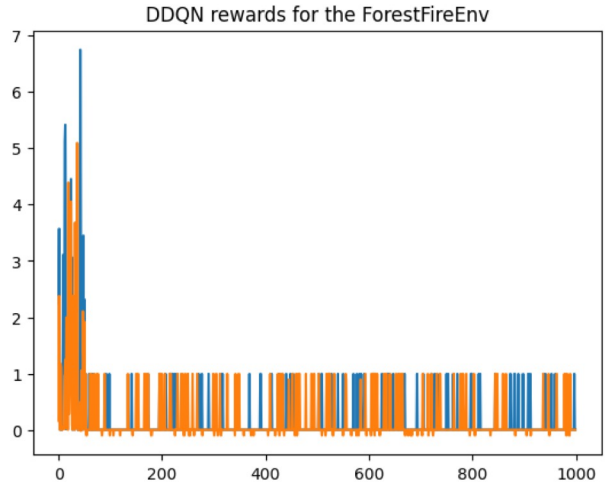
DDQN

- It is an improvement over DQN, which reduces overestimation of Q-values.
- It involves using two separate neural networks to estimate Q-values: the online model and the target model.
- The online model is used to select actions, while the target model is used to evaluate the actions.
- In DDQN, the target Q-values are computed using the target model, but the action selection is done using the online model.
- The code for DDQNAgent is similar to that of DQNAgent, but with modifications in the train() method to compute the target Q-values using the target model and the online model.
- The DDQN code presented here trains the agents using the ForestFireEnv.
- The training loop is similar to that of DQN, but with the use of two separate models to estimate Q-values.

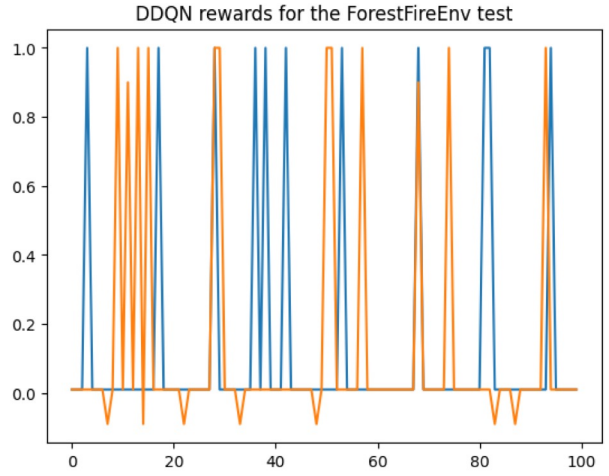
DDQN

```python
import torch
import torch.nn as nn
import torch.optim as optim
import random
from collections import deque

class DDQNAgent:
    def __init__(self, state_size, action_size, device, lr=0.001, gamma=0.99, buffer_size=10000, batch_size=64):
        self.state_size = state_size
        self.action_size = action_size
        self.device = device
        self.lr = lr
        self.gamma = gamma
        self.batch_size = batch_size

        self.memory = deque(maxlen=buffer_size)
        self.model = self._build_model().to(device)
        self.target_model = self._build_model().to(device)
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.lr)
        self.update_target_model()

    def _build_model(self):
        model = nn.Sequential(
            nn.Linear(self.state_size, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, self.action_size)
        )
        return model

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def choose_action(self, state, epsilon):
        if random.random() < epsilon:
            return random.randrange(self.action_size)
        else:
            state = torch.tensor(state, dtype=torch.float32, device=self.device).unsqueeze(0)
            return self.model(state).argmax().item()

    def update_target_model(self):
        self.target_model.load_state_dict(self.model.state_dict())


    def train(self):
        if len(self.memory) < self.batch_size:
            return

        minibatch = random.sample(self.memory, self.batch_size)

        states, actions, rewards, next_states, dones = zip(*minibatch)
        states = torch.tensor(states, dtype=torch.float32, device=self.device)
        actions = torch.tensor(actions, dtype=torch.long, device=self.device).unsqueeze(1)
        rewards = torch.tensor(rewards, dtype=torch.float32, device=self.device)
        next_states = torch.tensor(next_states, dtype=torch.float32, device=self.device)
        dones = torch.tensor(dones, dtype=torch.float32, device=self.device)

        current_q_values = self.model(states).gather(1, actions)

        # Use online model to select actions
        online_next_actions = self.model(next_states).argmax(1).unsqueeze(1)
        # Use target model to evaluate actions
        next_q_values = self.target_model(next_states).gather(1, online_next_actions).squeeze(1).detach()

        target_q_values = rewards + self.gamma * next_q_values * (1 - dones)

        loss = nn.MSELoss()(current_q_values, target_q_values.unsqueeze(1))
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
```

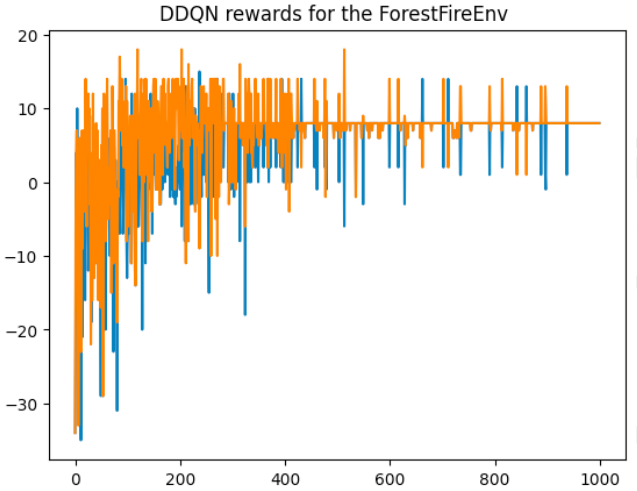
DDQN stochastic v/s deterministic



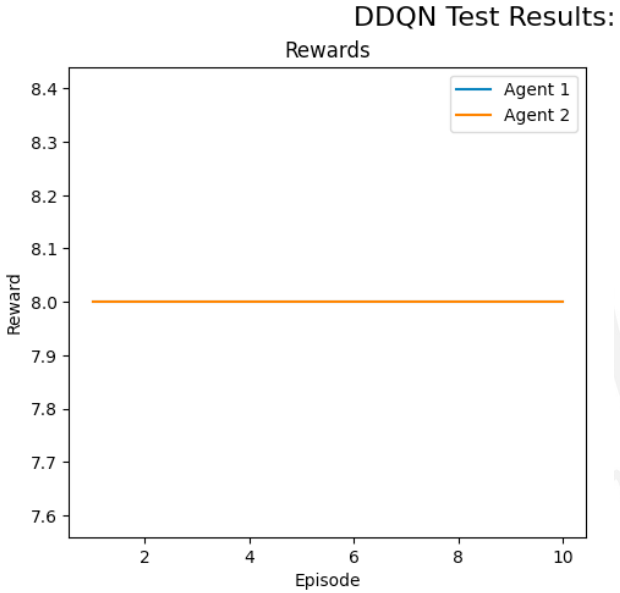
Due to the volatility of the environment defined the results of the DDQN implemented are no as expected with max timesteps of 1000 on a 6x6 grid env the results are [0.1, 1] for the agents.



And because of the training handling the volatile env the test results are fluctuating for the stochastic env for the 100 episodes.



Due to the stability of the environment defined the results of the DDQN implemented are quite impressive with max timesteps of 100 on a 4x4 grid env the results are stabilizing at [8, 8] for the agents.

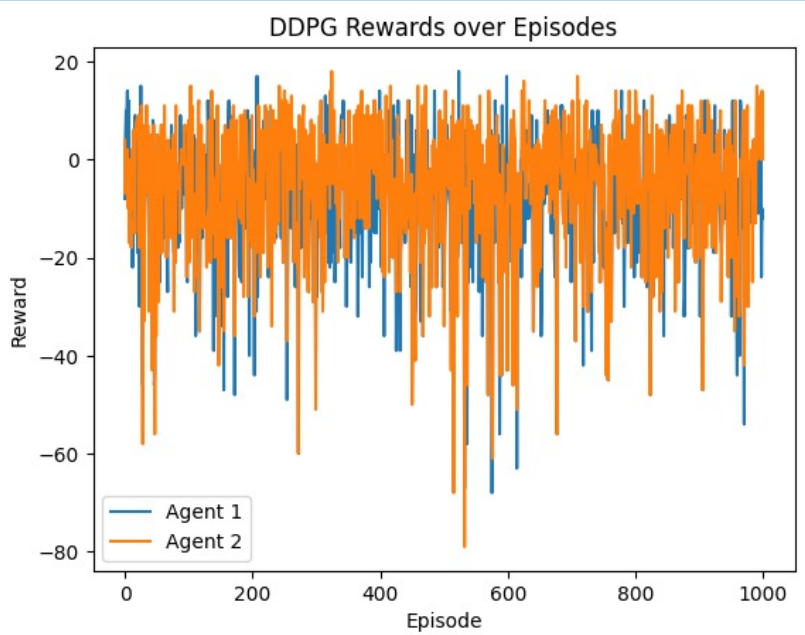


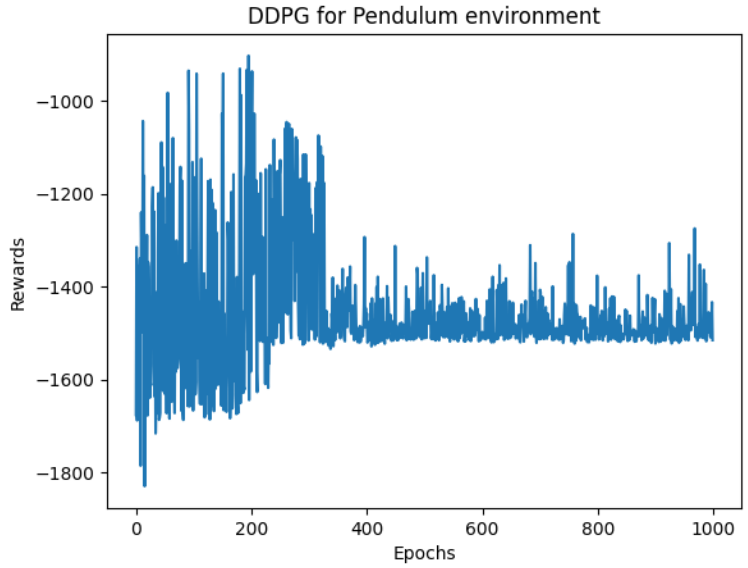
And because of the training handling the stable env the test results are constant for the static env for the 10 episodes. Results are achieved in 4 timesteps.

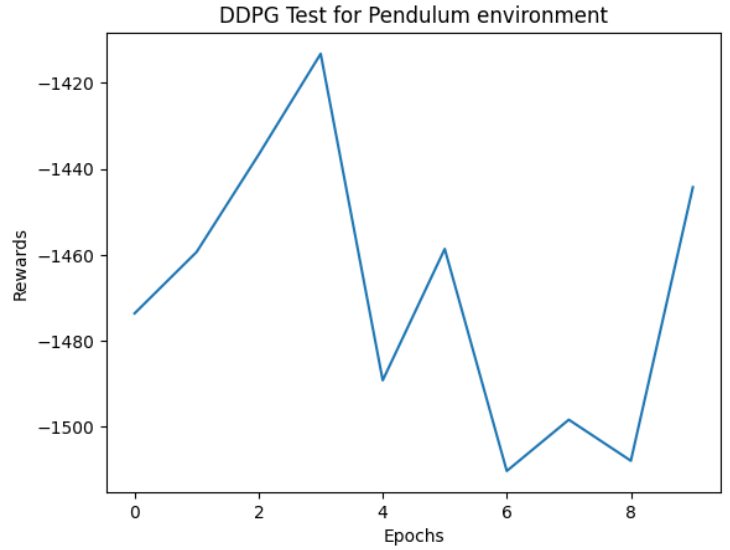
Results

DDPG:

- The above graph represents the DDPG performance on our stochastic environment, where fire starts and spreads randomly on the grid.
- The below following graphs show us the implementation of the DDPG algorithm on the pendulum gym environment. As we can see that after a certain number of epochs, the model has stabilized with some satisfactory rewards.





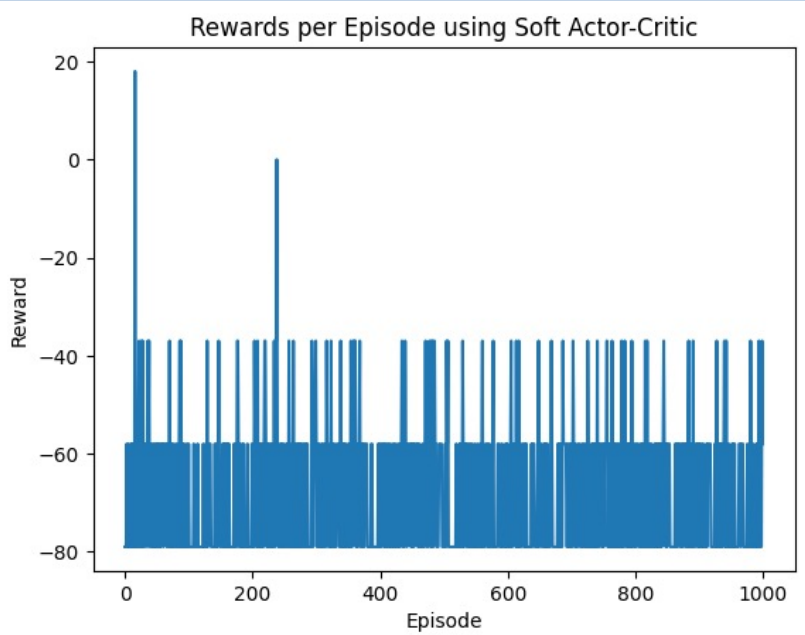


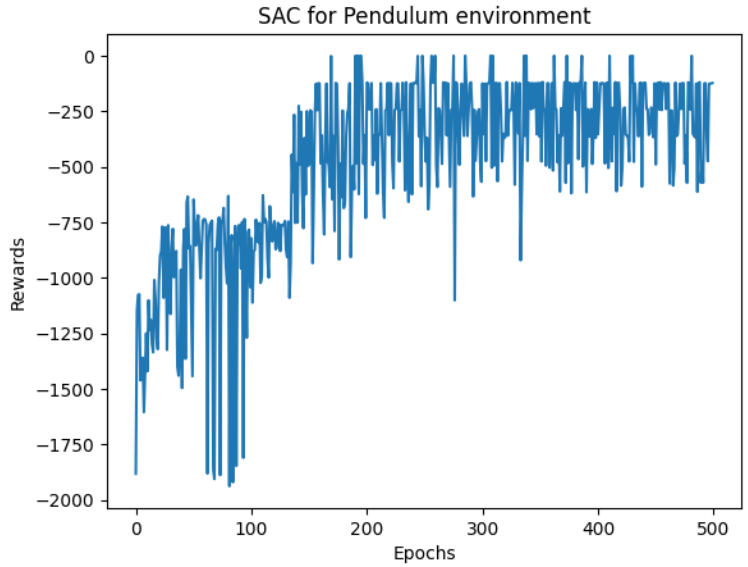
Results

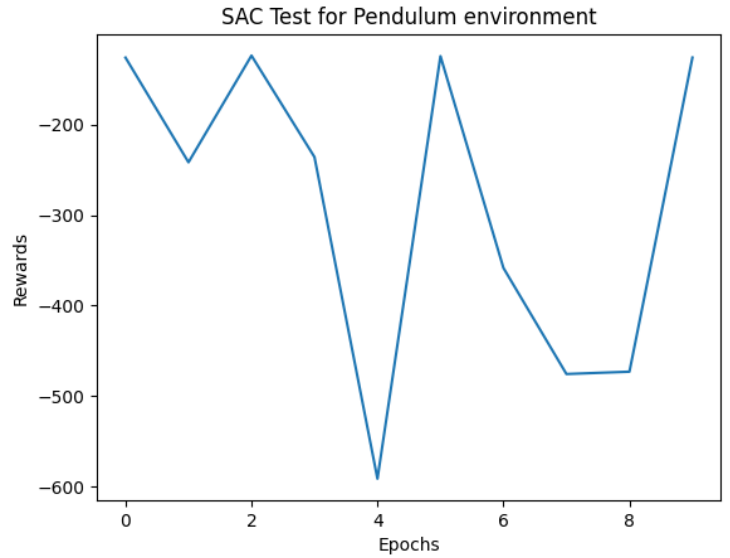
SAC:

The above graph represents the SAC algorithm on the stochastic ForestFireEnv, SAC performance has been pretty underwhelming in this environment compared to the pendulum gym environment.

The model has done pretty well on the pendulum environment, which we can notice from the graph below that with quite a less of training, after 200 epochs the model has some significant progress and has stabilized near the end of the training phase with some positive rewards as well.



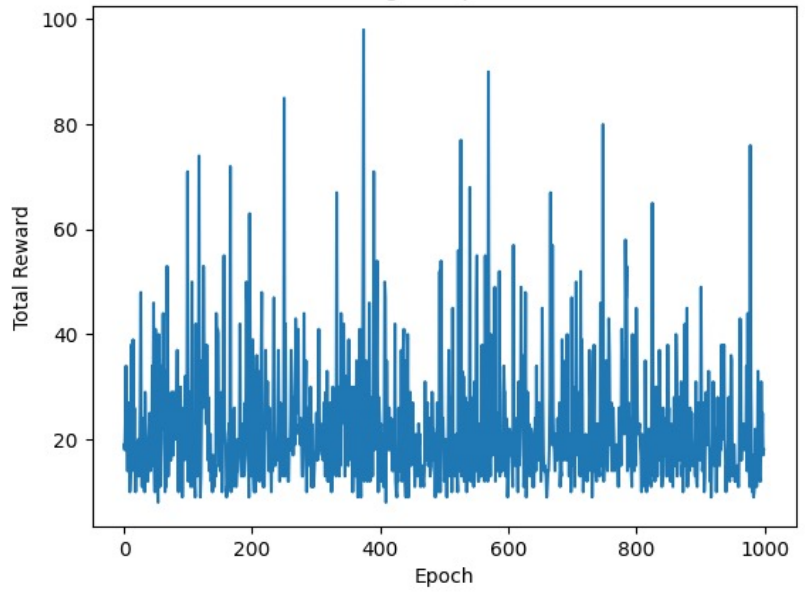




Results

PPO:

Proximal Policy Optimization algorithm has performance on our ForestFireEnv and CartPole have been suboptimal as they have not learnt the best policy with significant amount of training. This can be improved with better hyper parameter tuning, reward systems, Actor and Critic models.



Key Observations

- Challenges we faced:
  - Working in a MARL setup for different kinds of algorithms has been difficult to modify and exploring how the algorithms work has been fascinating.
  - Modifying the environment and deep RL methods.
  - Performance tuning to get the optimal results.
- Observations:
  - Need a more comprehensive reward structure
  - Most of the Deep RL methods have not been performing well on the stochastic setup as it is unpredictable.
  - All the methods we have explored work a little better on existing environments with continuous action sets and less punishing reward systems.

References

•Class notes and Piazza

•Assignment reference document

•Pytorch Documentation

•OpenAI Gym Documentation

•https://github.com/msinto93/DDPG

•https://github.com/nikhilbarhate99/PPO-PyTorch

•"Wildfire Detection using Deep Learning and Satellite Imagery" by J. He and H. Zhang. This paper proposes a deep learning approach to detecting wildfires using satellite imagery. The authors achieved an accuracy of 92% on their test set. DOI: 10.1109/JSTARS.2020.3043111

•Our CSE546 works on the Final project.

CONTRIBUTION SUMMARY

Team Member	Project Part	Contribution (%)
tanwinch	ENV,DDPG,PPO,SAC,REPORT	33.33%
kumarraj	ENV,DDPG,PPO,SAC,REPORT	33.33%
vkajeepe	DDPG,PPO,REPORT	33.33%

Though the work was assigned among teammates, each task has been done with the involvement of each teammate.

THANK YOU!

University at Buffalo
Information Technology