Installation

First, make sure you have the necessary libraries:

```
bash
Copy code
pip install numpy gym matplotlib torch torchvision
```

1. Q-Learning

```
python
Copy code
import numpy as np
import gym

# Create the environment
env = gym.make("Taxi-v3")

# Initialize parameters
num_episodes = 5000
learning_rate = 0.1
discount_factor = 0.9
num_actions = env.action_space.n
num_states = env.observation_space.n

# Q-Table initialization
Q = np.zeros((num_states, num_actions))

# Q-Learning algorithm
for episode in range(num_episodes):
    state = env.reset()
    done = False

    while not done:
        # Choose action (epsilon-greedy)
        if np.random.rand() < 0.1:  # Epsilon = 0.1
            action = env.action_space.sample()  # Explore
        else:
            action = np.argmax(Q[state])  # Exploit

        next_state, reward, done, _ = env.step(action)
        Q[state, action] += learning_rate * (reward + discount_factor *
np.max(Q[next_state]) - Q[state, action])
        state = next_state

# Testing the trained Q-Table
state = env.reset()
done = False
while not done:
    action = np.argmax(Q[state])
    next_state, reward, done, _ = env.step(action)
    env.render()
    state = next_state

env.close()
```

2. Deep Q-Networks (DQN)

```
python
Copy code
import numpy as np
```

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

# Create the environment
env = gym.make("CartPole-v1")

# Neural Network for DQN
class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 24)
        self.fc2 = nn.Linear(24, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.fc2(x)

# Parameters
num_episodes = 1000
learning_rate = 0.001
discount_factor = 0.99
epsilon = 1.0
epsilon_decay = 0.995
min_epsilon = 0.01
batch_size = 64
memory = deque(maxlen=2000)

# DQN model
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
model = DQN(input_dim, output_dim)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
criterion = nn.MSELoss()

# Training loop
for episode in range(num_episodes):
    state = env.reset()
    done = False

    while not done:
        # Epsilon-greedy action selection
        if np.random.rand() < epsilon:
            action = env.action_space.sample()  # Explore
        else:
            action = np.argmax(model(torch.FloatTensor(state)).detach().numpy())
# Exploit

        next_state, reward, done, _ = env.step(action)
        memory.append((state, action, reward, next_state, done))

        state = next_state

        # Experience replay
        if len(memory) > batch_size:
            minibatch = random.sample(memory, batch_size)
            for s, a, r, ns, d in minibatch:
                target = r + (1 - d) * discount_factor *
np.max(model(torch.FloatTensor(ns)).detach().numpy())
                target_f = model(torch.FloatTensor(s))
                target_f[a] = target
```

```
            optimizer.zero_grad()
            loss = criterion(model(torch.FloatTensor(s)), target_f)
            loss.backward()
            optimizer.step()

    epsilon = max(min_epsilon, epsilon * epsilon_decay)

# Test the trained model
state = env.reset()
done = False
while not done:
    action = np.argmax(model(torch.FloatTensor(state)).detach().numpy())
    next_state, reward, done, _ = env.step(action)
    env.render()
    state = next_state

env.close()
```

3. Policy Gradient Methods

```
python
Copy code
import numpy as np
import gym
import torch
import torch.nn as nn
import torch.optim as optim

# Create the environment
env = gym.make("CartPole-v1")

# Policy Network
class PolicyNetwork(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 24)
        self.fc2 = nn.Linear(24, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return torch.softmax(self.fc2(x), dim=-1)

# Parameters
num_episodes = 1000
learning_rate = 0.01
discount_factor = 0.99

# Policy network
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
policy_net = PolicyNetwork(input_dim, output_dim)
optimizer = optim.Adam(policy_net.parameters(), lr=learning_rate)

# Training loop
for episode in range(num_episodes):
    state = env.reset()
    rewards = []
    log_probs = []

    while True:
        state_tensor = torch.FloatTensor(state)
        probs = policy_net(state_tensor)
        action = np.random.choice(output_dim, p=probs.detach().numpy())
```

```
        log_prob = torch.log(probs[action])
        next_state, reward, done, _ = env.step(action)

        rewards.append(reward)
        log_probs.append(log_prob)

        state = next_state
        if done:
            break

    # Compute the loss
    returns = []
    R = 0
    for r in reversed(rewards):
        R = r + discount_factor * R
        returns.insert(0, R)

    returns = torch.FloatTensor(returns)
    log_probs = torch.stack(log_probs)

    # Policy gradient update
    loss = -torch.sum(log_probs * returns)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Test the trained policy
state = env.reset()
done = False
while not done:
    state_tensor = torch.FloatTensor(state)
    action_probs = policy_net(state_tensor)
    action = np.random.choice(output_dim, p=action_probs.detach().numpy())
    next_state, reward, done, _ = env.step(action)
    env.render()
    state = next_state

env.close()
```

4. Actor-Critic Methods

```
python
Copy code
import numpy as np
import gym
import torch
import torch.nn as nn
import torch.optim as optim

# Create the environment
env = gym.make("CartPole-v1")

# Actor-Critic Network
class ActorCriticNetwork(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ActorCriticNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 24)
        self.actor = nn.Linear(24, output_dim)
        self.critic = nn.Linear(24, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        return self.actor(x), self.critic(x)
```

```
# Parameters
num_episodes = 1000
learning_rate = 0.01
discount_factor = 0.99

# Actor-Critic model
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
model = ActorCriticNetwork(input_dim, output_dim)
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
for episode in range(num_episodes):
    state = env.reset()
    done = False

    while not done:
        state_tensor = torch.FloatTensor(state)
        action_probs, value = model(state_tensor)
        action = np.random.choice(output_dim, p=torch.softmax(action_probs,
dim=-1).detach().numpy())
        next_state, reward, done, _ = env.step(action)

        # Compute advantage
        next_value = model(torch.FloatTensor(next_state))[1].detach()
        advantage = reward + (1 - done) * discount_factor * next_value - value

        # Update actor
        actor_loss = -torch.log_softmax(action_probs, dim=-1)[action] *
advantage
        critic_loss = advantage.pow(2)

        optimizer.zero_grad()
        (actor_loss + critic_loss).backward()
        optimizer.step()

        state = next_state

# Test the trained model
state = env.reset()
done = False
while not done:
    state_tensor = torch.FloatTensor(state)
    action_probs, _ = model(state_tensor)
    action = np.random.choice(output_dim, p=torch.softmax(action_probs,
dim=-1).detach().numpy())
    next_state, reward, done, _ = env.step(action)
    env.render()
    state = next_state

env.close()
```

5. Monte Carlo Methods

For Monte Carlo methods, we will use a simple episodic task.

```
python
Copy code
import numpy as np
import gym

# Create the environment
```

```
env = gym.make("Blackjack-v1", natural=True)

# Monte Carlo Policy
def generate_episode(policy):
    state = env.reset()
    episode = []
    done = False

    while not done:
        action = policy[state[0], state[1], state[2]]
        next_state, reward, done, _ = env.step(action)
        episode.append((state, action, reward))
        state = next_state

    return episode

# Policy (random for example)
policy = np.random.choice(env.action_space.n, (32, 11, 2))

# Monte Carlo Control
num_episodes = 10000
returns = {}
returns_count = {}
Q = np.zeros((32, 11, 2, env.action_space.n))

for episode in range(num_episodes):
    ep = generate_episode(policy)
    G = 0
    for state, action, reward in reversed(ep):
        G = reward + 0.9 * G
        if (state, action) not in returns:
            returns[(state, action)] = 0
            returns_count[(state, action)] = 0

        returns[(state, action)] += G
        returns_count[(state, action)] += 1
        Q[state[0], state[1], state[2], action] = returns[(state, action)] /
returns_count[(state, action)]

    # Policy Improvement
    for s in range(32):
        for a in range(env.action_space.n):
            policy[s] = np.argmax(Q[s])

# Test the trained policy
state = env.reset()
done = False
while not done:
    action = policy[state[0], state[1], state[2]]
    next_state, reward, done, _ = env.step(action)
    env.render()
    state = next_state

env.close()
```