In [1]:
```
import gym
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
from collections import namedtuple
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
# Hyperparameters
BATCH_SIZE = 64
GAMMA = 0.99
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
TARGET_UPDATE = 10
```

In [3]:
```
# Define the neural network
class DQN(nn.Module):
    def __init__(self, input_size, output_size):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)
```

In [4]:
```
# Experience Replay
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 're
```

In [5]:
```
class ReplayMemory:
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

In [6]:
```
# DQN Agent
class DQNAgent:
    def __init__(self, input_size, output_size):
        self.policy_net = DQN(input_size, output_size)
        self.target_net = DQN(input_size, output_size)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()
        self.optimizer = optim.Adam(self.policy_net.parameters())
        self.memory = ReplayMemory(10000)
        self.steps_done = 0
        self.input_size = input_size
        self.output_size = output_size

    def select_action(self, state):
        eps_threshold = EPS_END + (EPS_START - EPS_END) * np.exp(-1. * self
        self.steps_done += 1
        if random.random() > eps_threshold:
            with torch.no_grad():
                return self.policy_net(state).max(1)[1].view(1, 1)
        else:
            return torch.tensor([[random.randrange(self.output_size)]], dty

    def optimize_model(self):
        if len(self.memory) < BATCH_SIZE:
            return

        transitions = self.memory.sample(BATCH_SIZE)
        batch = Transition(*zip(*transitions))

        non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, ba
        non_final_next_states = torch.cat([s for s in batch.next_state if s

        state_batch = torch.cat(batch.state)
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)

        state_action_values = self.policy_net(state_batch).gather(1, action

        next_state_values = torch.zeros(BATCH_SIZE)
        if non_final_next_states.size(0) > 0:
            next_state_values[non_final_mask] = self.target_net(non_final_n

        expected_state_action_values = (next_state_values * GAMMA) + reward

        loss = nn.functional.smooth_l1_loss(state_action_values, expected_s

        self.optimizer.zero_grad()
        loss.backward()
        for param in self.policy_net.parameters():
            param.grad.data.clamp_(-1, 1)
        self.optimizer.step()
```

In [7]:
```
# Environment Setup
env = gym.make('CartPole-v1', render_mode=None)
input_size = env.observation_space.shape[0]
output_size = env.action_space.n
agent = DQNAgent(input_size, output_size)
```

In [8]:
```
# Training Loop
num_episodes = 1000
for i_episode in range(num_episodes):
    state, _ = env.reset()
    state = torch.tensor([state], dtype=torch.float32)

    for t in range(500):
        action = agent.select_action(state)
        next_state, reward, terminated, truncated, _ = env.step(action.item
        done = terminated or truncated

        reward = torch.tensor([reward], dtype=torch.float32)
        if not done:
            next_state = torch.tensor([next_state], dtype=torch.float32)
        else:
            next_state = None

        agent.memory.push(state, action, next_state, reward)
        state = next_state

        agent.optimize_model()

        if done:
            print(f"Episode {i_episode + 1}: lasted {t + 1} timesteps")
            break

    if i_episode % TARGET_UPDATE == 0:
        agent.target_net.load_state_dict(agent.policy_net.state_dict())

print("Training finished.")
env.close()
```

```
Episode 983: lasted 500 timesteps
Episode 984: lasted 371 timesteps
Episode 985: lasted 500 timesteps
Episode 986: lasted 252 timesteps
Episode 987: lasted 497 timesteps
Episode 988: lasted 500 timesteps
Episode 989: lasted 500 timesteps
Episode 990: lasted 213 timesteps
Episode 991: lasted 500 timesteps
Episode 992: lasted 500 timesteps
Episode 993: lasted 500 timesteps
Episode 994: lasted 500 timesteps
Episode 995: lasted 500 timesteps
Episode 996: lasted 500 timesteps
Episode 997: lasted 500 timesteps
Episode 998: lasted 500 timesteps
Episode 999: lasted 500 timesteps
Episode 1000: lasted 500 timesteps
Training finished.
```