Deep Q-Networks (DQN)

Overview

Deep Q-Networks (DQN) combine Q-learning with deep neural networks to enable reinforcement learning in environments with high-dimensional state spaces. By approximating the Q-function using deep neural networks, DQNs can handle complex tasks such as playing Atari games directly from pixel inputs. (medium.com)

Why Use DQN?

DQN addresses the limitations of traditional Q-learning by:

- **Handling High-Dimensional States**: Utilizing deep neural networks allows DQN to process raw, high-dimensional inputs like images.
- **Stabilizing Training**: Techniques such as experience replay and target networks help stabilize training in complex environments.

These features make DQN suitable for tasks where traditional Q-learning is impractical due to the complexity of the state space.

Prerequisites

To implement DQN, ensure the following Python packages are installed:

- **PyTorch**: For building and training neural networks.
- **Gymnasium**: For creating and interacting with reinforcement learning environments.

Install them using pip:

```
pip install torch gymnasium
```

Files Included

- **Python Script**: Contains the code to define the DQN model, set up the environment, and train the agent.

Code Description

1. **Import Libraries**:

   ```
   import torch
   import torch.nn as nn
   import torch.optim as optim
   import gymnasium as gym
   import numpy as np
   ```

   Imports necessary libraries for building the neural network, optimization, and interacting with the environment.

2. **Define the DQN Model**:

```
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, 128)
        self.fc3 = nn.Linear(128, output_size)

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

Defines a simple feedforward neural network with two hidden layers to approximate the Q-function.

3. **Initialize Environment and Model**:

```
 env = gym.make('CartPole-v1')
model = DQN(input_size=4, output_size=2)
target_model = DQN(input_size=4, output_size=2)
target_model.load_state_dict(model.state_dict())
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Sets up the CartPole environment, initializes the DQN model, and creates a target model with the same architecture.

4. **Experience Replay Buffer**:

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = []
        self.capacity = capacity
        self.position = 0

    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return np.random.choice(self.buffer, batch_size, replace=False)

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

Implements a simple experience replay buffer to store and sample transitions for training.

5. **Training Loop**:

```
def train(env, model, target_model, optimizer, episodes=1000, batch_size=64, gamma
    replay_buffer = ReplayBuffer(10000)
    for episode in range(episodes):
        state, _ = env.reset()
        state = torch.tensor(state, dtype=torch.float32)
```

```
        done = False
        total_reward = 0
        while not done:
            action = select_action(state, model)
            next_state, reward, done, _, _ = env.step(action)
            next_state = torch.tensor(next_state, dtype=torch.float32)
            replay_buffer.push(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward
            if len(replay_buffer) > batch_size:
                experiences = replay_buffer.sample(batch_size)
                batch = list(zip(*experiences))
                states, actions, rewards, next_states, dones = [torch.tensor(x) for
                # Compute Q targets and loss
                # Update model parameters
        if episode % 10 == 0:
            target_model.load_state_dict(model.state_dict())
        print(f"Episode {episode}, Total Reward: {total_reward}")
```

Defines the training loop where the agent interacts with the environment, stores experiences, and updates the model.

6. **Action Selection**:

```
def select_action(state, model, epsilon=0.1):
    if np.random.rand() < epsilon:
        return np.random.choice([0, 1])
    with torch.no_grad():
        q_values = model(state)
        return torch.argmax(q_values).item()
```

Implements an epsilon-greedy policy for action selection during training.

7. **Run Training**:

```
train(env, model, target_model, optimizer)
```

Starts the training process.

Expected Output

The training loop will output the total reward for each episode, allowing you to monitor the agent's performance over time.

Use Cases

DQN has been successfully applied to various reinforcement learning tasks, including:

- **Atari Game Playing**: Achieving human-level performance in classic video games.
- **Robotics**: Enabling robots to learn complex manipulation tasks.
- **Autonomous Vehicles**: Training self-driving cars to navigate complex environments.

Its ability to handle high-dimensional state spaces makes it versatile for a wide range of applications.

Advantages

DQN offers several advantages:

- **Scalability**: Can handle large