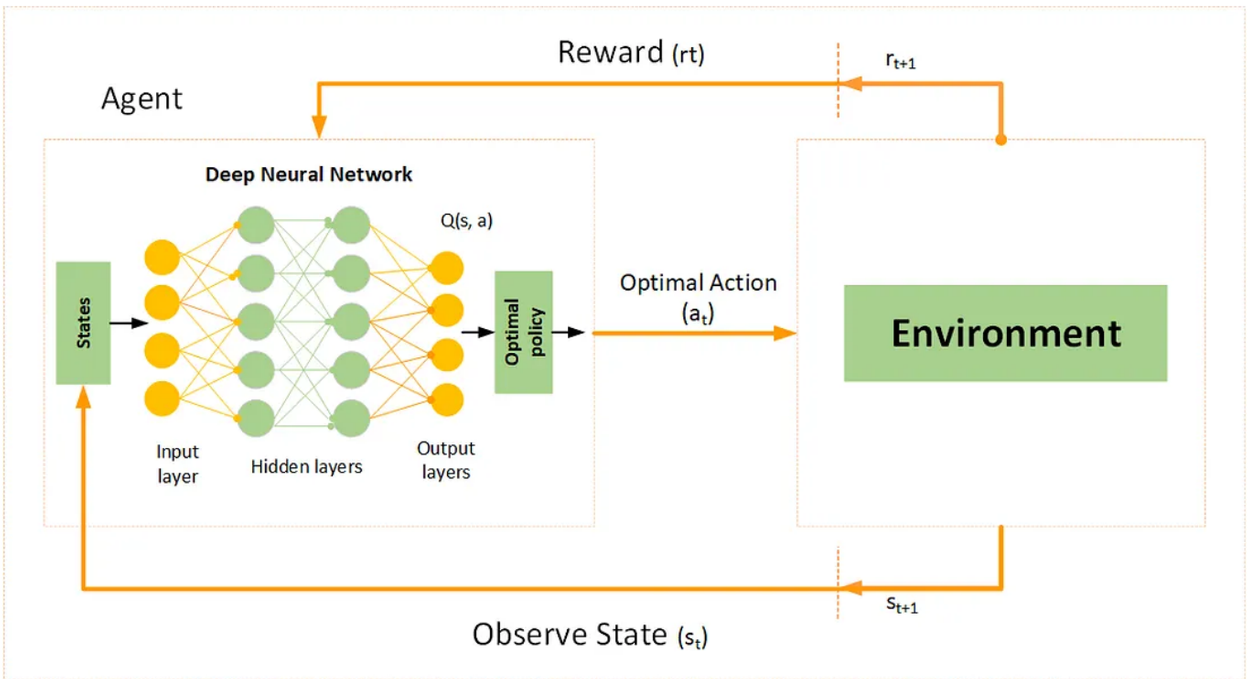
Deep Q-Learning (DQN)

Samina Amin Follow 4 min read · Sep 14, 2024

Deep Q-Learning or Deep Q Network (DQN) is an extension of the basic Q-Learning algorithm, which uses deep neural networks to approximate the Q-values. Traditional Q-Learning works well for environments with a small and finite number of states, but it struggles with large or continuous state spaces due to the size of the Q-table. Deep Q-Learning overcomes this limitation by replacing the Q-table with a neural network that can approximate the Q-values for every state-action pair.



Structure of DQN

Key Concepts of Deep Q-Learning

1. **Q-Function Approximation**: Instead of using a table to store Q-values for each state-action pair, DQN uses a neural network to approximate the Q-values. The input to the network is the state, and the output is a set of Q-values for all possible actions.

2. **Experience Replay**: To stabilize the training, DQN uses a memory buffer (replay buffer) to store experiences (state, action, reward, next state). The network is trained on random mini-batches of experiences from this buffer, breaking the correlation between consecutive experiences and improving sample efficiency.
3. **Target Network**: DQN introduces a second neural network, called the target network, which is used to calculate the target Q-values. This target network is updated less frequently than the main network to prevent rapid oscillations in learning.
4. **Bellman Equation in DQN**: The update rule for DQN is based on the Bellman equation, similar to Q-Learning:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s,a;\theta) \right)$$

Bellman Equation i

Where:

- θ are the weights of the main Q-network,
- θ− are the weights of the target Q-network,
- s is the current state,
- a is the action taken,
- ris the reward received,
- s′ is the next state,
- maxa′QQ(s′,a′) is the maximum Q-value for the next state.

Python Implementation of Deep Q-Learning (DQN)

We will implement a simple version of Deep Q-Learning using the `CartPole` environment from OpenAI Gym, where the agent must balance a pole on a moving cart.

1. Installing Dependencies

```
pip install gym torch numpy matplotlib
```

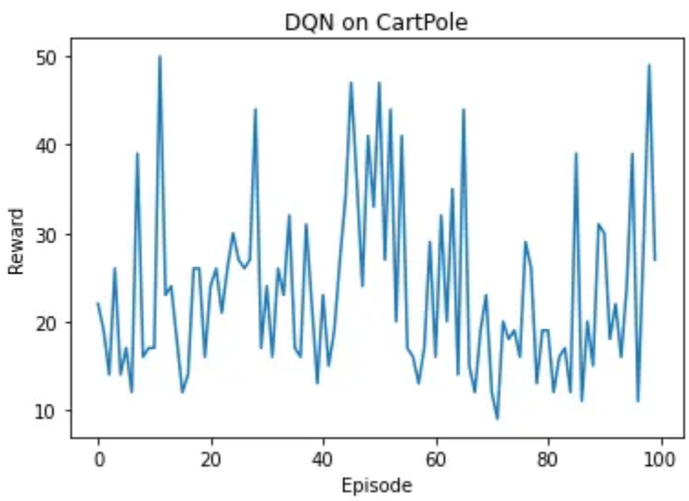
2. DQN Implementation for CartPole

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

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

# Neural network model for approximating Q-values
class DQN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, output_dim)

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

# Hyperparameters
learning_rate = 0.001
gamma = 0.99
epsilon = 1.0
epsilon_min = 0.01
epsilon_decay = 0.995
batch_size = 64
target_update_freq = 1000
memory_size = 10000
episodes = 1000

# Initialize Q-networks
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
policy_net = DQN(input_dim, output_dim)
target_net = DQN(input_dim, output_dim)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = optim.Adam(policy_net.parameters(), lr=learning_rate)
memory = deque(maxlen=memory_size)

# Function to choose action using epsilon-greedy policy
def select_action(state, epsilon):
    if random.random() < epsilon:
        return env.action_space.sample()  # Explore
    else:
```

```
        state = torch.FloatTensor(state).unsqueeze(0)
        q_values = policy_net(state)
        return torch.argmax(q_values).item()  # Exploit

# Function to optimize the model using experience replay
def optimize_model():
    if len(memory) < batch_size:
        return

    batch = random.sample(memory, batch_size)
    state_batch, action_batch, reward_batch, next_state_batch, done_batch = zip(

    state_batch = torch.FloatTensor(state_batch)
    action_batch = torch.LongTensor(action_batch).unsqueeze(1)
    reward_batch = torch.FloatTensor(reward_batch)
    next_state_batch = torch.FloatTensor(next_state_batch)
    done_batch = torch.FloatTensor(done_batch)

    # Compute Q-values for current states
    q_values = policy_net(state_batch).gather(1, action_batch).squeeze()

    # Compute target Q-values using the target network
    with torch.no_grad():
        max_next_q_values = target_net(next_state_batch).max(1)[0]
        target_q_values = reward_batch + gamma * max_next_q_values * (1 - done_b

    loss = nn.MSELoss()(q_values, target_q_values)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Main training loop
rewards_per_episode = []
steps_done = 0

for episode in range(episodes):
    state = env.reset()
    episode_reward = 0
    done = False

    while not done:
        # Select action
        action = select_action(state, epsilon)
        next_state, reward, done, _ = env.step(action)

        # Store transition in memory
        memory.append((state, action, reward, next_state, done))

        # Update state
        state = next_state
        episode_reward += reward

        # Optimize model
        optimize_model()

        # Update target network periodically
        if steps_done % target_update_freq == 0:
            target_net.load_state_dict(policy_net.state_dict())

        steps_done += 1

    # Decay epsilon
    epsilon = max(epsilon_min, epsilon_decay * epsilon)

    rewards_per_episode.append(episode_reward)
```

```
# Plotting the rewards per episode
import matplotlib.pyplot as plt
plt.plot(rewards_per_episode)
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.title('DQN on CartPole')
plt.show()
```

Output



Output

Real-World Applications of Deep Q-Learning

DQN and its variations have been applied in numerous complex tasks, including:

1. Video Game AI

- DQN gained popularity for its application in playing Atari games, where the agent learned to achieve superhuman performance by interacting with the environment.

2. Robotics

- In robotic manipulation tasks, DQN can be used to help robots learn complex behaviors, such as picking and placing objects or navigating dynamic environments.

3. Autonomous Vehicles

- Self-driving cars can use DQN to learn driving policies in simulated environments, allowing them to navigate through traffic, avoid obstacles, and optimize routes.

4. Finance

- DQN can be applied to algorithmic trading strategies, where an agent learns to maximize profit by making buy/sell/hold decisions in real-time.

5. Healthcare

- In healthcare, DQN can be used to personalize treatment plans by learning from patient data and maximizing long-term health outcomes.

Conclusion

DQN extends traditional Q-Learning by leveraging neural networks to approximate the Q-values for environments with large or continuous state spaces. Through techniques like experience replay and the use of a target network, DQN overcomes many of the limitations of basic Q-Learning. Its ability to learn complex tasks from high-dimensional input makes it a powerful tool for solving real-world problems like game playing, robotics, and autonomous driving.

Reinforcement Learning | Deep Q Learning | Deep Q Network | Artificial Intelligence | Artificial Neural Network

Written by Samina Amin

162 followers · 57 following

A girl who codes with a passion for AI and machine learning | AI Researcher | Deep Learning | Reinforcement Learning | Data Science | NLP

Follow

Responses (1)

Jagannath Rao

What are your thoughts?