FINSEARCH

USE DEEP REINFORCEMENT LEARNING (RL) TO OPTIMISE STOCK TRADING STRATEGY AND THUS MAXIMISE INVESTMENT RETURN

Team:
Janani Rajesh Kumar-24B2181
Mridhula Vishwanath-24B2262
Sanaa Tabassum-24B1078
Siddharth Dey-24B1285

DQN Algorithm:

DQN, or Deep Q-Network, is a kind of deep RL algorithm that combines Q-learning and deep neural networks. DeepMind developed it and allows an agent to function optimally in complex, high-dimensional environments (usually in video games).

There are two key concepts to understand before diving into how DQN performs: a) Q-learning and b) Deep Neural Networks

a) Q-Learning:

In Q-Learning, the agent starts with zero knowledge about the environment and basically figures out the best outcomes in the environment through trial and error.
Essentially, the agent explores by taking actions, and determines whether the action was good/bad based on outcomes, and learns from these experiences.
Each time the agent performs an action and ends up in a new state, it receives feedback in the form of a reward. Based on the value of this reward, it judges whether the action taken was good or not, and updates its internal logic accordingly.
By repeating this process and storing it, the agent gradually devises a strategy and acts optimally in each state.
At the end, the agent consistently chooses actions that move it toward its goal, without ever being explicitly told how.

The agent interacts with the environment in discrete time steps. Assuming each step to be of time ‘t’, the agent observes a state “s_t”, chooses an action “a_t”, receives a reward “r_t”, and transitions to a new state “s_{t+1}”.
The agent’s end goal is to devise a strategy, called $\pi(s)$, that maximises expected outcomes.

Q-value is basically a number that tells the agent how good it is to take a particular action in a particular state, keeping in mind the immediate rewards and also expected future rewards

Q(s,a) = Expected total reward if you take action a in state s, and then follow the best possible strategy afterwards

A higher Q-value means that the action is better — it will likely lead to more rewards overall.
The agent learns these Q-values during training and picks the best action by choosing the action with the **highest Q-value** for the current state.
The Q-values are updated using the **Bellman Equation**.

Bellman Equation:

$$\mathbf{Q(s_t,a_t) \leftarrow Q(s_t,a_t)+\alpha[r_t+\gamma \max_{a'} Q(s_{t+1},a')-Q(s_t,a_t)]}$$

where $Q(s_t,a_t)$ is the current Q-value, α is the learning rate (how fast we learn), r_t is the reward received after taking action, γ is the discount factor (importance of future rewards), and $\max_{a'} Q(s_{t+1},a')$ is the best future value

To explore the environment and avoid only exploiting current knowledge, Q-learning typically employs the Greedy Algorithm.

With probability ϵ: choose a random action
With probability $1-\epsilon$: choose $\arg\max_a Q(s,a)$

We gradually decay ϵ to favor exploitation over time

Advantages-

1. Doesn't need prerequisite knowledge of the environment
2. Very simple and intuitive to understand
3. Can learn easily and no matter what
4. Performs well in small environments

Limitations-

1. Does not work well for large environments/spaces
2. It requires discrete state and action spaces to work
3. Exploration challenges arise because of the Greedy Algorithm

b) Deep Neural Networks:

A deep neural network(DNN) is an artificial neural network that has many layers between its input layer and output layer. They are able to learn complex representations of data, making them useful for many tasks such as image recognition, NLP, etc.

Theory Behind It -

DNNs are loosely modeled after how neurons in the brain transmit information; each neuron calculates a weighted sum of its inputs along with a bias, and then passes it through a nonlinear activation function. Multiple layers of neurons are stacked such that each layer's output is received by the next.

We train DNNs using backpropagation, which is an algorithm that computes gradients of the loss function with respect to each parameter using the chain rule. It then updates them using optimization methods. Neural networks with even one hidden layer of sufficient width can approximate a continuous function. DNNs model complex hierarchical patterns more efficiently by extending this by using depth.

Core concepts -

An input layer receives raw features, and the output layer produces predictions. The hidden layers are where transformations and feature extractions occur. Activation functions introduce nonlinearity; examples are ReLU, Sigmoid, and Tanh. Loss functions help quantify the error; examples are Mean Squared Error and Cross-Entropy Loss. Optimization helps adjust the weights to minimize loss (SGD, Adam optimizer). Regularization prevents overfitting; we can use techniques like Dropout or L1/L2 weight penalties. Batch normalization speeds up and stabilizes training.

Advantages- We can capture complex patterns in data automatically due to its hierarchical feature learning. We can also approximate a wide range of functions due to its flexibility. DNNs also perform very well in tasks like vision, speech, and language. Another advantage is that we can use this on very large networks and datasets.

Limitations- Unfortunately, DNNs require a large amount of labeled data and need powerful GPUs/TPUs for training them. There is also a risk of overfitting on small datasets without regularization. Its performance also heavily depends on tuning and architecture.

DQN:

What is DQN? (theory)

The DQN agent interacts with the environment, observes states, takes actions based on the Q-function (often using an epsilon-greedy strategy), and receives rewards. These experiences are stored in a replay buffer, and the agent learns from these experiences by updating the neural network weights, effectively improving its understanding of the Q-function.

The agent assesses the environment and takes actions based on the Q function and receives rewards. The past tasks are stored in a replay buffer, and the agent learns from the experiences by updating the neural network weights.

The agent takes action based on strategies like the epsilon greedy strategy,

Core Concepts and Working:

DQN uses neural networks and Q-learning to estimate how good any given strategy is in its action state. We do this in the following steps:

1. Using Neural Networks as a Q-Function Approximator:
A deep neural network Q(s,a,θ) takes state s as input and then outputs Q-values (explained above) for all possible actions. Here, θ is the parameter/weight of the network

2. Experience Replay
The agent stores experiences (s,a,r,s') in a replay buffer instead of training on sequential data, as it's highly correlated. Random batches are then sampled from this buffer during training to break correlations, if any, and improve stability

3. Target Network
A separate network $Q_{target}(s,a,\theta^{-})$ is used to compute the target Q-values. This target network is a copy of the main network and is updated slowly once every few steps to stabilize training.

4. Loss Function
The loss is calculated using something called the Bellman error.

$$\mathbf{L(\theta) = E_{(s,a,r,s')\sim D}[(y-Q(s,a,\theta))^{2}]}$$

$$\mathbf{where\ y=r+\gamma_{a'}maxQ_{target}(s',a',\theta^{-})}$$

<u>Here's how training (through training loops) happens:</u>

First, observe the current state 's'. Then choose action 'a' using the epsilon-greedy policy, choose a random action with probability epsilon, or choose an action that maximises Q(s,a). Execute the action a and receive reward r, and we get a new state s'. Now, store transition (s,a,r,s′) in the replay buffer. Sample the minibatch of transitions obtained from the buffer. Then, compute the loss and update theta using gradient descent. Periodically update the target network.

<u>Advantages -</u> DQNs can handle high-dimensional inputs, and they use deep neural networks to process raw inputs, which makes them helpful in complex environments. They also don't require a model of the environment; they can
Add Headings (Format > Paragraph styles) and they will appear in your table of contents.
learn with interaction and experience, which makes it simpler than model-based approaches. It uses a separate target network to reduce divergence and fluctuations during its training. It also stores past transitions and randomly samples them to break the correlation between consecutive samples. This improves stability and sample efficiency. It has also proven to outperform human-level performances.

Limitations- Unfortunately, it requires a large number of environment interactions to efficiently learn, which makes it computationally expensive and impractical in real-world scenarios. Since it learns from past experiences, it can suffer from stale data, which affects the learning progress. DQN tends to overestimate Q-values, which might hurt performance. The training is also sensitive to the learning rate, replay buffer size, epsilon decay, etc. It also relies on epsilon-greedy exploration, which is suboptimal, and it may fail to explore well in deceptive environments.

CODE FOR SOLVING THE INVERTED PENDULUM PROBLEM USING DQN:

```
import gymnasium as gym
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
from collections import deque
import matplotlib.pyplot as plt


ENV_NAME = "CartPole-v1"
EPISODES = 500
GAMMA = 0.99
LR = 1e-3
BATCH_SIZE = 64
MEMORY_SIZE = 10000
TARGET_UPDATE = 10
EPS_START = 1.0
EPS_END = 0.01
EPS_DECAY = 500

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


class DQN(nn.Module):
    def _init_(self, obs_dim, act_dim):
        super(DQN, self)._init_()
        self.net = nn.Sequential(
            nn.Linear(obs_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, act_dim)
        )

    def forward(self, x):
        return self.net(x)

class ReplayBuffer:
    def _init_(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def push(self, s, a, r, s_, done):
        self.buffer.append((s, a, r, s_, done))

    def sample(self, batch_size):
        samples = random.sample(self.buffer, batch_size)
        s, a, r, s_, d = zip(*samples)
        return (
            torch.tensor(s, dtype=torch.float32).to(device),
            torch.tensor(a, dtype=torch.int64).unsqueeze(1).to(device),
            torch.tensor(r, dtype=torch.float32).unsqueeze(1).to(device),
            torch.tensor(s_, dtype=torch.float32).to(device),
            torch.tensor(d, dtype=torch.float32).unsqueeze(1).to(device)
        )

    def _len_(self):
        return len(self.buffer)


def select_action(state, steps_done):
    epsilon = EPS_END + (EPS_START - EPS_END) * np.exp(-1. * steps_done /
EPS_DECAY)
    if random.random() < epsilon:
        return random.randrange(n_actions)
    else:
        with torch.no_grad():
            state_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0).to(device)
            return policy_net(state_tensor).argmax(1).item()


env = gym.make(ENV_NAME, render_mode='human')
obs_dim = env.observation_space.shape[0]
n_actions = env.action_space.n

policy_net = DQN(obs_dim, n_actions).to(device)
target_net = DQN(obs_dim, n_actions).to(device)
target_net.load_state_dict(policy_net.state_dict())

optimizer = optim.Adam(policy_net.parameters(), lr=LR)
```

```
memory = ReplayBuffer(MEMORY_SIZE)


plt.ion()
fig, ax = plt.subplots()
rewards_plot = []
reward_line, = ax.plot([], [], label="Episode Reward")
ax.set_xlabel('Episode')
ax.set_ylabel('Total Reward')
ax.set_title('DQN Learning Progress')
ax.grid(True)
ax.legend()

steps_done = 0
for episode in range(EPISODES):
    state, _ = env.reset()
    total_reward = 0
    done = False

    while not done:
        env.render()
        action = select_action(state, steps_done)
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated

        memory.push(state, action, reward, next_state, done)
        state = next_state
        total_reward += reward
        steps_done += 1

        if len(memory) >= BATCH_SIZE:
            s, a, r, s_, d = memory.sample(BATCH_SIZE)
            q_vals = policy_net(s).gather(1, a)
            next_q = target_net(s_).max(1)[0].unsqueeze(1).detach()
            expected_q = r + GAMMA * next_q * (1 - d)

            loss = nn.functional.mse_loss(q_vals, expected_q)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    if episode % TARGET_UPDATE == 0:
        target_net.load_state_dict(policy_net.state_dict())

    rewards_plot.append(total_reward)
    reward_line.set_data(range(len(rewards_plot)), rewards_plot)
    ax.set_xlim(0, len(rewards_plot))
    ax.set_ylim(0, max(rewards_plot) + 10)
    plt.pause(0.01)

    epsilon_now = EPS_END + (EPS_START - EPS_END) * np.exp(-1. * steps_done /
EPS_DECAY)
    print(f"Episode {episode}, Reward: {total_reward:.1f}, Epsilon: {epsilon_now:.3f}")

env.close()
plt.ioff()
plt.show()
```