Finsearch '24 - Midterm Report

Using Deep Reinforcement Learning (RL) to Optimize Stock Trading Strategy

Aayush Borkar (23B0944)
Aditya Neeraje (23B0940)
Balaji Karedla (23B1029)
Nirav Bhattad (23B3307)

Mentored by Abhishek Kumar

Contents

1 Introduction

Reinforcement Learning (RL) is a type of machine learning that allows an agent to learn how to behave in an environment by performing actions and observing the rewards it receives. The agent learns to achieve a goal by maximizing the cumulative reward it receives. The agent interacts with the environment by taking actions, and the environment responds with rewards and new states. The agent learns to take actions that maximize the expected cumulative reward over time.

RL has been used to optimize stock trading strategies by learning to make decisions about buying and selling stocks. The agent learns to make decisions that maximize the expected return on investment. The agent learns to take actions that maximize the expected cumulative reward over time. The agent learns to make decisions about buying and selling stocks by learning from historical data and by interacting with the environment.

The goal of this project is to use RL to optimize stock trading strategies by learning to make decisions about buying and selling stocks. The agent learns to make decisions that maximize the expected return on investment. The agent learns to take actions that maximize the expected cumulative reward over time. The agent learns to make decisions about buying and selling stocks by learning from historical data and by interacting with the environment.

The motivation behind this project is to develop a stock trading strategy that can outperform human traders. The agent learns to make decisions about buying and selling stocks by learning from historical data and by interacting with the environment. The agent learns to take actions that maximize the expected cumulative reward over time. The agent learns to make decisions that maximize the expected return on investment.

2 DQN Algorithm

2.1 Introduction

Deep Q-Network (DQN) is a reinforcement learning algorithm that uses a deep neural network to approximate the Q-value function. The Q-value function is a function that takes in the current state and action and returns the expected future reward. The DQN algorithm uses a neural network to approximate this function and uses it to make decisions about which action to take in a given state.

2.2 How it works

The DQN algorithm works by training a neural network to predict the Q-value function. The neural network takes in the current state as input and outputs the Q-value for each possible action. The algorithm then uses the Q-values to select the best action to take in the current state.

The DQN algorithm uses a technique called experience replay to improve the stability and efficiency of training. Experience replay involves storing the agent's experiences in a replay buffer and sampling from this buffer to train the neural network. This helps to break the correlation between consecutive experiences and makes the training process more stable.

The DQN algorithm also uses a target network to improve the stability of training. The target network is a copy of the main neural network that is used to calculate the target Q-values during training. The target network is updated less frequently than the main network, which helps to stabilize the training process.

The Q-value function is defined as:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

where:

- $Q(s, a)$ is the Q-value for state s and action a
- r is the reward for taking action a in state s
- γ is the discount factor
- s' is the next state
- a' is the next action

The loss function for training the neural network is defined as:

$$L(\theta) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2]$$

where:

- θ are the parameters of the neural network
- θ^- are the parameters of the target network

The neural network is trained using gradient descent to minimize the loss function.

2.3 Advantages

The DQN algorithm has several advantages over traditional reinforcement learning algorithms:

- **Scalability**: The DQN algorithm can handle large state and action spaces, making it suitable for complex problems.
- **Generalization**: The DQN algorithm can generalize across different states and actions, making it more efficient than tabular methods.
- **Stability**: The DQN algorithm uses experience replay and target networks to stabilize the training process and improve performance.

2.4 Limitations

The DQN algorithm also has some limitations:

- **Sample Efficiency**: The DQN algorithm can be sample inefficient, requiring large amounts of data to learn an effective policy.
- **Hyperparameter Sensitivity**: The DQN algorithm is sensitive to hyperparameters and can be difficult to tune.
- **Overestimation Bias**: The DQN algorithm can overestimate Q-values, leading to suboptimal policies.

2.5 Application in Stock Trading

The DQN algorithm can be used in stock trading to optimize trading strategies. The algorithm can learn to make decisions about when to buy or sell stocks based on the current state of the market. By training the algorithm on historical stock data, it can learn to predict the best actions to take in different market conditions.

The DQN algorithm can be used to optimize trading strategies by maximizing the expected future reward. The algorithm learns to make decisions that maximize the expected future reward, which can lead to better trading performance.

3 DDPG Algorithm

3.1 Introduction

Deep Deterministic Policy Gradient (DDPG) is a reinforcement learning algorithm that combines the power of deep neural networks with the stability of deterministic policy gradients. DDPG is an off-policy algorithm that can learn policies in high-dimensional, continuous action spaces. It is particularly well-suited for problems with continuous action spaces, such as robotic control and stock trading.

3.2 Architecture

The DDPG algorithm consists of two main components: an actor network and a critic network. The actor network is responsible for learning the policy function, which maps states to actions. The critic network is responsible for learning the value function, which estimates the expected return for a given state-action pair.

The actor network takes the current state as input and outputs the action to take in that state. The critic network takes the current state and action as input and outputs the Q-value for that state-action pair. The actor and critic networks are trained using gradient descent to minimize the loss functions associated with each network.'

The Q-value function for the critic network is defined as:

$$Q(s, a) = r + \gamma Q(s', \mu'(s'))$$

where:

- $Q(s, a)$ is the Q-value for state s and action a
- r is the reward for taking action a in state s
- γ is the discount factor
- s' is the next state
- $\mu'(s')$ is the target action for the next state

The loss function for training the critic network is defined as:

$$L(\theta) = \mathbb{E}[(r + \gamma Q(s', \mu'(s'; \theta^-)) - Q(s, a; \theta))^2]$$

where:

- θ are the parameters of the critic network
- θ^- are the parameters of the target critic network

3.3 Training Process

The training process for the DDPG algorithm involves interacting with the environment, collecting experiences, and updating the actor and critic networks. The algorithm uses a technique called experience replay to store and sample experiences from a replay buffer. This helps to break the correlation between consecutive experiences and makes the training process more stable.

The actor network is trained using the deterministic policy gradient, which is the gradient of the Q-value with respect to the action. The critic network is trained using the temporal difference error, which is the difference between the predicted Q-value and the target Q-value.

The target Q-value is calculated using the target actor and target critic networks, which are copies of the main actor and critic networks. The target networks are updated less frequently than the main networks, which helps to stabilize the training process.

3.4 Advantages

The DDPG algorithm has several advantages over traditional reinforcement learning algorithms:

- **Continuous Action Spaces**: DDPG can learn policies in continuous action spaces, making it suitable for problems with complex action spaces.
- **Stability**: DDPG uses experience replay and target networks to stabilize the training process and improve performance.
- **Efficiency**: DDPG can learn policies quickly and efficiently, even in high-dimensional state spaces.

3.5 Limitations

Despite its advantages, the DDPG algorithm has some limitations:

- **Hyperparameters**: DDPG requires careful tuning of hyperparameters to achieve good performance.
- **Exploration**: DDPG can struggle with exploration in high-dimensional action spaces, leading to suboptimal policies.
- **Sample Efficiency**: DDPG can be sample inefficient, requiring large amounts of data to learn an effective policy.

4 Cartpole Environment

4.1 Introduction

The Cartpole environment is a classic reinforcement learning problem that involves balancing a pole on a cart. The goal of the agent is to keep the pole balanced by moving the cart left or right. The environment is implemented in the OpenAI Gym library and is commonly used to test and benchmark reinforcement learning algorithms.

4.2 Description

The Cartpole environment consists of a pole attached to a cart, which can move left or right along a track. The pole is initially upright, and the agent must take actions to keep it balanced. The state of the environment is defined by four variables:

- Cart position: The position of the cart along the track
- Cart velocity: The velocity of the cart
- Pole angle: The angle of the pole with respect to the vertical axis
- Pole angular velocity: The angular velocity of the pole

The agent can take two actions: move the cart left or move the cart right. The agent receives a reward of +1 for each time step that the pole remains upright. The episode ends when the pole falls below a certain angle or the cart moves outside the track boundaries.

4.3 Methodology

We employ a deep Q-network (DQN) approach with experience replay and target networks to train a policy for the CartPole environment. Below are the key components of our implementation:

- **DiscretizedActionWrapper**: A wrapper class to discretize the continuous action space of the environment.
- **ReplayBuffer**: A cyclic buffer to store and sample experiences (transitions) for training.
- **Policy (Neural Network)**: A feedforward neural network that serves as the policy, mapping state observations to action probabilities.
- **Training Loop**: Iterative process where the agent interacts with the environment, collects experiences, and updates the policy based on the loss calculated from the expected and observed rewards.
- **Target Network**: A separate network periodically updated with the parameters of the policy network to stabilize training.

4.4 Implementation Details

Listing 1: DiscretizedActionWrapper Class

```
class DiscretizedActionWrapper(gym.ActionWrapper):
    def __init__(self, env, n_actions):
        super(DiscretizedActionWrapper, self).__init__(env)
        self.n_actions = n_actions
        self.action_space = gym.spaces.Discrete(n_actions)
        self.continuous_action_space = env.action_space
        self.actions = np.linspace(self.continuous_action_space.low,
                                   self.continuous_action_space.high,
                                   self.n_actions)

    def action(self, action):
        continuous_action = self.actions[action]
        return continuous_action
```

Listing 2: ReplayBuffer Class

```
# Define the ReplayBuffer class
class ReplayBuffer(object):
    def __init__(self, capacity):
        self.buffer = deque([], maxlen=capacity)

    def push(self, *args):
        self.buffer.append(Transition(*args))

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

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

# Define the Transition namedtuple used in ReplayBuffer
Transition = namedtuple('Transition', ('state', 'action', 'next_state', '
    reward'))
```

Listing 3: Policy Neural Network Class

```
# Define the Policy neural network class
class Policy(nn.Module):
    def __init__(self, obs, act):
        super(Policy, self).__init__()
        self.fc1 = nn.Linear(obs, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, act)

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

    def act(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        probs = self.forward(state).cpu()
        m = Categorical(probs)
        action = m.sample()
        return action.item(), m.log_prob(action)
```

Listing 4: Training Loop and Utilities

```
# Define the training loop function
def train_loop():
    if len(replay_buffer) < hyperparameters["h_size"]:
        return
    transitions = replay_buffer.sample(hyperparameters["h_size"])
    batch = Transition(*zip(*transitions))

    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.
       next_state)), device=device, dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state if s is not
        None])
    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    state_action_values = policy_net(state_batch).gather(1, action_batch)

    next_state_values = torch.zeros(hyperparameters["h_size"], device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).
           max(1).values
    expected_state_action_values = (next_state_values * hyperparameters["gamma
       "]) + reward_batch

    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.
       unsqueeze(1))

    optimizer.zero_grad()
    loss.backward()

    torch.nn.utils.clip_grad_value_(policy_net.parameters(), 100)
    optimizer.step()

# Define epsilon-greedy policy function
def eps_greedy(state):
```

```
    global num_steps
    decay_term = math.exp(-1. * num_steps / hyperparameters["eps_decay"])
    eps = hyperparameters["eps_end"] + (hyperparameters["eps_start"] -
       hyperparameters["eps_end"]) * decay_term
    num_steps += 1
    if random.random() <= eps:
        return torch.tensor([[env.action_space.sample()]], device=device,
           dtype=torch.long)
    else:
        with torch.no_grad():
            return policy_net(state).max(1).indices.view(1, 1)
```

Listing 5: Main Training Loop

```
# Main training loop
for _ in tqdm(range(hyperparameters["n_training_episodes"]), desc="Training"):
    state, info = env.reset()
    state = torch.tensor(state, dtype=torch.float32, device=device).unsqueeze
       (0)
    for t in count():
        action = eps_greedy(state)
        observation, reward, terminated, truncated, _ = env.step(action.item()
           )
        reward = torch.tensor([reward], device=device)
        done = terminated or truncated

        if terminated:
            next_state = None
        else:
            next_state = torch.tensor(observation, dtype=torch.float32, device
               =device).unsqueeze(0)

        replay_buffer.push(state, action, next_state, reward)
        state = next_state
        train_loop()

        target_net_state_dict = target_net.state_dict()
        policy_net_state_dict = policy_net.state_dict()
        for key in policy_net_state_dict:
            target_net_state_dict[key] = policy_net_state_dict[key] *
               hyperparameters["tau"] + target_net_state_dict[key] * (1 -
               hyperparameters["tau"])
        target_net.load_state_dict(target_net_state_dict)

        if done:
            episode_durations.append(t + 1)
            break
```

4.5 Results

We trained the policy for 2000 episodes and evaluated its performance over 10 episodes. The training progress was tracked by episode durations (time steps before termination). The target network's soft update mechanism helped in achieving stable convergence of the policy network.

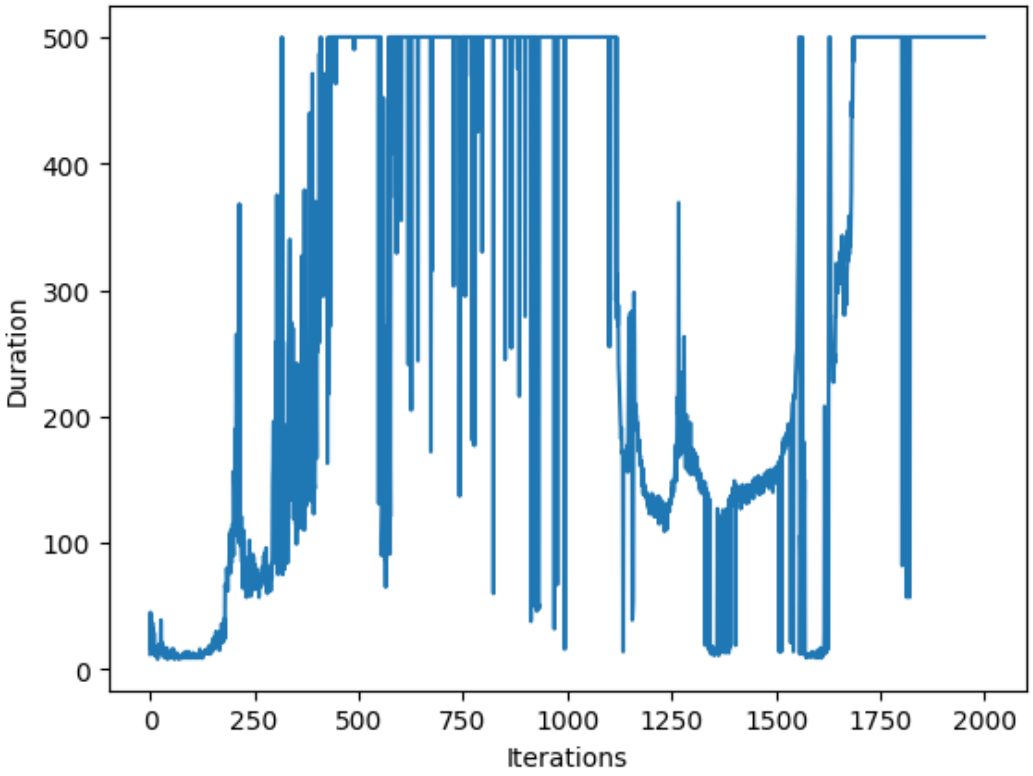

Figure 1: Training Progress of CartPole Environment

4.6 Conclusion

The CartPole environment serves as a good introductory problem for RL algorithms. Our implementation successfully trained a policy network to balance the pole for extended periods. Future work could involve experimenting with different network architectures, exploring other RL algorithms, or applying this approach to more complex environments.