Machine Learning for Financial Markets Project

Deep Reinforcement Learning for Investor Agents in Stock Trading

Oumou SOW
Lath ESSOH
Tsiba RAZAFINDRAKOTO

Table of Contents

Introduction and Motivation

Financial Market

- complex and dynamic, requiring intelligent decision-making
- Traditional trading strategies often struggle to adapt to market fluctuations.

Deep Reinforcement Learning

- provides a way to optimize investment decisions
- enables autonomous trading agents to learn from market data and adjust strategies dynamically
- Combines RL & Deep Neural Networks for real-time decision-making.

key motivations

- Develop a self-learning trading agent using DRL techniques.
- Learn profitable strategies without human supervision.
- Adapt dynamically to market trends & minimize risks.
- Outperform benchmarks like Buy & Hold, Moving Averages.
- Use real-world financial data for validation.

Related Work

Key Contributions:

1. LFinRL: A Deep Reinforcement Learning Library for Automated Stock Trading in Quantitative Finance (Xiao-Yang Liu et al., 2020)

- Open-source DRL library for automated stock trading.
- Supports DQN, PPO, A2C, TD3, and SAC models.
- Incorporates transaction costs, liquidity, and risk constraints.
- Benchmarked on NASDAQ-100, S&P 500, and other indices.

2. Deep Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy (Hongyang Yang et al., 2020)

- Combines PPO, A2C, and DDPG models.
- Dynamically selects the best-performing agent based on the Sharpe ratio.
- Applied to Dow Jones 30 stocks, outperforming traditional strategies.
- Adapts to bullish, bearish, and volatile market phases.

Data Exploration & Preprocessing

Data Source

- Financial data retrieved from **Yahoo Finance** (AAPL – Apple)
- Historical prices (OHLC), volume, and technical indicators

Data Preparation

- **Cleaning:** Handling missing values and formatting columns
- **Feature Engineering:** Adding key indicators (SMA, RSI, MACD)
- **Normalization:** Scaling data for better model stability
- **Data Splitting:** Dividing into train (80%) and test (20%)

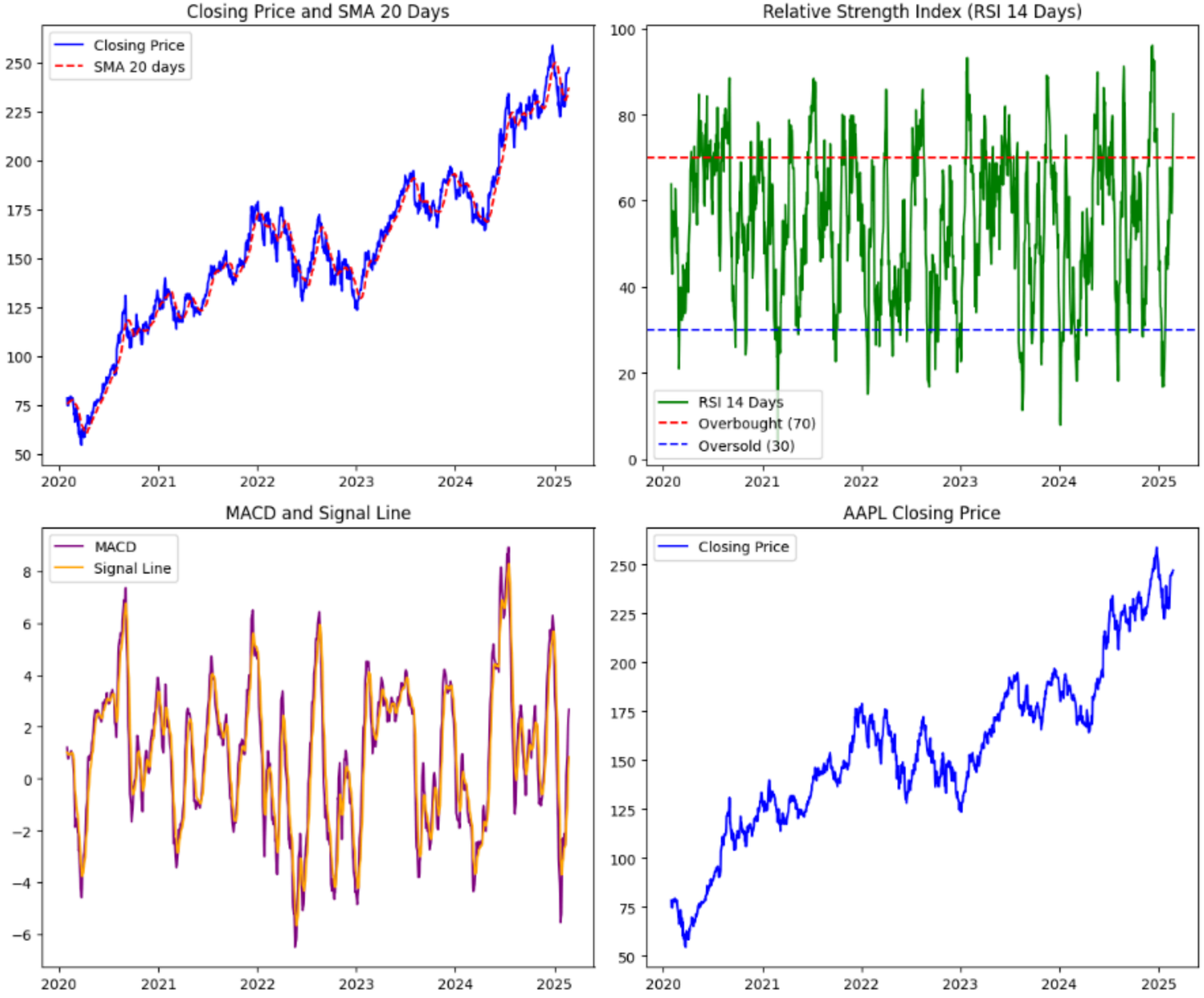
Goal

Obtain clean and optimized data for an efficient trading model.

Data Visualization

We create a figure with four subplots, each displaying different aspects of the stock's technical indicators:

- The first subplot shows the closing price along with the Simple Moving Average (SMA) to observe the overall trend.
- The second displays the RSI (Relative Strength Index) with horizontal lines at 70 (overbought) and 30 (oversold) to identify potential entry and exit points.
- The third illustrates the MACD (Moving Average Convergence Divergence) along with its signal line, helping detect momentum shifts.
- The fourth subplot presents only AAPL's closing price, allowing a clear reference for price movements.



3. Create trading environment Custom environment
3.1. Defining the Trading Environment

Initialization of environment variables

```
class StockTradingEnv(gym.Env):
    def __init__(self, data, initial_balance=10000):
        super().__init__()
        self.data = data.ffill()
        self.initial_balance = self.current_balance = initial_balance
        self.current_step = self.position = 0
        self.total_assets = self.current_balance
```

Initialization of environment variables:

- **data**: Historical market data
- **balance**: The agent's initial balance.
- **position** : The current position in stocks (number of shares held).
- **total_assets**: The total value of assets, which is the sum of the balance and the value of the held shares

Defining the action and observation spaces

```
    self.action_space = gym.spaces.Discrete(3)
    self.observation_space = gym.spaces.Box(0, np.inf, (4,), np.float32)

def _next_observation(self):
    state = self.data.iloc[self.current_step]
    return np.array([
        state['Close'],
        state['SMA_20'],
        state['RSI_14'],
        state['MACD']
    ], dtype=np.float32)
```

- **Action space**: The agent can choose from three actions: buy (0), sell (1), or hold (2)
- **Observation space**: The state data, which includes technical indicators: Close, SMA_20, RSI_14, and MACD.

3. Create trading environment
3.1. Defining the Trading Environment

reset() method and step() method

```python
def reset(self):
    self.current_balance = self.initial_balance
    self.position = self.current_step = 0
    self.total_assets = self.current_balance
    return self._next_observation()

def step(self, action):
    done = self.current_step >= len(self.data) - 1
    if done: return self._next_observation(), 0, done, {}

    current_price = self.data.iloc[self.current_step]['Close']
    reward = 0

    if self.current_step == 0:
        self.current_step += 1
        return self._next_observation(), 0, done, {}

    if action == 0 and self.current_balance >= current_price:
        amount_to_buy = int(self.current_balance // current_price)
        if amount_to_buy > 0:
            self.position += amount_to_buy
            self.current_balance -= amount_to_buy * current_price

    elif action == 1 and self.position > 0:
        sale_value = self.position * current_price
        reward = sale_value - self.initial_balance
        self.current_balance = sale_value
        self.position = 0

    elif action == 1 and self.position == 0:
        pass

    self.total_assets = self.current_balance + self.position * current_price
    reward = self.total_assets - self.initial_balance
    self.current_step += 1
    return self._next_observation(), reward, done, {}
```

`reset(self)`:

- Resets the environment at the start of a new episode.
- Resets balance, position, and step count.
- Returns the initial observation.

`step(self, action)`:

- Executes an action (buy, sell, or hold) based on the agent's input.
- Updates balance, position, and total asset value using the formula:

$$\text{total\_assets} = \text{current\_balance} + (\text{position} \times \text{current\_price})$$

- Calculates a reward based on the change in total asset value using the formula:

$$\text{reward} = \text{total\_assets} - \text{initial\_balance}$$

- Returns the next observation, the reward, and the episode's done status.

render() method

```python
def render(self):
    profit = self.total_assets - self.initial_balance
    print(f'Step: {self.current_step}, Balance: {self.current_balance}, Position: {self.position}, Profit: {profit}', flush=True)
```

`render(self)`:

- Displays the current step, balance, position, and profit (calculated separately):

$$\text{profit} = \text{total\_assets} - \text{initial\_balance}$$

3. Create trading environment Custom environment
3.2. Create an environment with training data

```
env = StockTradingEnv(train_data)
state = env.reset()
done = False
total_reward = 0

while not done:
    action = env.action_space.sample()
    next_state, reward, done, info = env.step(action)
    total_reward += reward
    env.render()
    state = next_state
display("Total reward:", total_reward)
```

- **Initialization**: The environment is created and reset
- **Testing**: Random actions are performed, the environment is updated, and the reward is accumulated
- **Display**: The state and reward are shown after each action
- **End**: The total reward is displayed at the end

4. Testing Different Methods
4.1. Method 1: Custom DQN with PyTorch

Q-Learning Model Definition

```python
import torch
import torch.nn as nn
import numpy as np
import gym

class QNetwork(nn.Module):
    def __init__(self, input_size, output_size):
        super(QNetwork, 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))
        x = self.fc3(x)
        return x
```

- **Initialization**: The class defines a simple feedforward neural network with three fully connected layers (fc1, fc2, and fc3).
- fc1 maps input to 128 units.
- fc2 has 128 units.
- fc3 outputs the size of the action space (output_size)
- **Forward Pass**: The input x passes through the layers, using ReLU activation for fc1 and fc2, and a linear transformation for the output layer (fc3). The output is returned

Agent Training and Evaluation on the Test Set

```python
env = StockTradingEnv(train_data)
model = QNetwork(4, env.action_space.n)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
epsilon, epsilon_min, epsilon_decay, gamma = 1.0, 0.01, 0.995, 0.99

for episode in range(1):
    state = env.reset()
    done, total_reward = False, 0

    while not done:
        state_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
        action = env.action_space.sample() if np.random.rand() < epsilon else torch.argmax(model(state_tensor)).item()
        next_state, reward, done, _ = env.step(action)
        total_reward += reward

        next_state_tensor = torch.tensor(next_state, dtype=torch.float32).unsqueeze(0)
        target = reward + gamma * torch.max(model(next_state_tensor)) if not done else reward
        loss = (target - model(state_tensor)[0][action]) ** 2

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

        state = next_state
        env.render()

    env.render()
    print(f"Episode {episode+1} - Total Reward: {total_reward}")
    epsilon = max(epsilon_min, epsilon * epsilon_decay)
```

```python
test_env = StockTradingEnv(test_data)
portfolio_values_dqn1 = []
total_test_reward, test_done = 0, False
state = test_env.reset()

while not test_done:
    state_tensor = torch.tensor(state, dtype=torch.float32).unsqueeze(0)
    action = torch.argmax(model(state_tensor)).item()
    next_state, reward, done, _ = test_env.step(action)

    total_test_reward += reward
    portfolio_values_dqn1.append(test_env.total_assets)
    state = next_state

    print(f"Step: {test_env.current_step}, Done: {done}, Total Reward: {total_test_reward}")
    test_done = done
print(f"Test Reward DQN: {total_test_reward}")
```

- **Initialization**: The environment, model, and agent parameters are set up

Training Loop:

- For each episode, the agent chooses an action (exploration or exploitation)
- The environment is updated, and the reward is accumulated.
- The model is updated based on the difference between the predicted and target Q-values
- Display: The environment state is rendered after each action and at the end of the episode

4.2 Method 2: Using DQN with the Stable Baselines3 Library

Preparing the DQN environment

```
from stable_baselines3 import DQN
from stable_baselines3.common.vec_env import DummyVecEnv

env = DummyVecEnv([lambda: StockTradingEnv(train_data)])

# Initialize the DQN model
model_dqn = DQN("MlpPolicy", env, verbose=1)
model_dqn.learn(total_timesteps=10000)

model_dqn.save("dqn_trading_model")
```

- **Environment Setup**: A vectorized Gym environment (DummyVecEnv) is created using the StockTradingEnv with train_data.
- **DQN Model Initialization**: A Deep Q-Network (DQN) model is initialized with the MlpPolicy for the environment
- **Training**: The model is trained for 10,000 timesteps
- Model Saving: After training, the model is saved to a file named "dqn_trading_model"

Test the DQN model and record the portfolio value

```
model_dqn = DQN.load("dqn_trading_model")

env = DummyVecEnv([lambda: StockTradingEnv(test_data)])

portfolio_values_dqn2 = []
state = env.reset()
done = False
total_reward_dqn2 = 0
step_count = 0

while not done:
    action, _states = model_dqn.predict(state)

    state, reward, done, info = env.step(action)

    total_reward_dqn2 += reward
    portfolio_values_dqn2.append(env.envs[0].total_assets)

    print(f"Step: {step_count} - Action: {action} - Reward: {reward} - Total Assets: {env.envs[0].total_assets}")

    step_count += 1

    if step_count == 254:
        print(f"Step {step_count} reached, end of episode.")
        done = True

print(f"Test Reward DQN: {total_reward_dqn2}")
```

- **Model Loading**: The pre-trained DQN model is loaded from the file "dqn_trading_model".
- **Environment Setup**: A new DummyVecEnv is created using the StockTradingEnv with test_data

Testing Loop:

- The model predicts actions, and the environment is updated based on those actions
- Rewards and portfolio values are accumulated, and the results are displayed at each step
- The loop stops after 254 steps or when the episode ends
- Results: The total test reward is printed at the end of the testing phase

5. Other Models and Strategies
5.1. PPO Model

Preparing the DQN environment

```
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv

env = DummyVecEnv([lambda: StockTradingEnv(train_data)])

# Initialize the PPO model
model_ppo = PPO("MlpPolicy", env, verbose=1)
model_ppo.learn(total_timesteps=10000)

model_ppo.save("ppo_trading_model")
```

- **Environment Setup**: A vectorized Gym environment (DummyVecEnv) is created using StockTradingEnv with train_data
- **PPO Model Initialization**: The Proximal Policy Optimization (PPO) model is initialized with the MlpPolicy for the environment.
- **Training**: The model is trained for 10,000 timesteps.
- **Model Saving**: After training, the model is saved to a file named "ppo_trading_model"

Test the PPO model and record the portfolio value

```
env = DummyVecEnv([lambda: StockTradingEnv(test_data)])
model_ppo = PPO.load("ppo_trading_model")

portfolio_values_ppo = []
state = env.reset()
done = False
total_reward_ppo = 0
step_count = 0

while not done:
    action, _states = model_ppo.predict(state)

    state, reward, done, info = env.step(action)

    total_reward_ppo += reward
    portfolio_values_ppo.append(env.envs[0].total_assets)

    print(f"Step: {step_count} - Action: {action} - Reward: {reward} - Total Assets: {env.envs[0].total_assets}")

    step_count += 1

    if step_count == 254:
        print(f"Step {step_count} reached, end of episode.")
        done = True

print(f"Test Reward PPO: {total_reward_ppo}")
```

- **Model Loading**: The pre-trained PPO model is loaded from "ppo_trading_model".
- **Environment Setup**: A new DummyVecEnv is created using StockTradingEnv with test_data

Testing Loop:

- The model predicts actions, and the environment is updated based on those actions
- Rewards and portfolio values are accumulated, and the results are displayed at each step
- The loop stops after 254 steps or when the episode ends
- Results: The total test reward is printed at the end of the testing phase

Buy and Hold

```
portfolio_values_buy_and_hold = []
initial_balance = 10000
buy_and_hold_position = 0

initial_price = test_data['Close'].iloc[0]
buy_and_hold_position = initial_balance // initial_price

for i in range(1, len(test_data)):
    current_price = test_data['Close'].iloc[i]
    portfolio_value = buy_and_hold_position * current_price + initial_balance - (buy_and_hold_position * initial_price)
    portfolio_values_buy_and_hold.append(portfolio_value)

buy_and_hold_reward = portfolio_values_buy_and_hold[-1] - initial_balance
print(f"Buy and Hold Reward: {buy_and_hold_reward}")
```

- **Initial Setup**: A "Buy and Hold" strategy is implemented with an initial balance of $10,000. The first stock purchase is made at the first observation's closing price
- **Portfolio Value Calculation**: At each step, the portfolio value is updated based on the stock's current price.
- **Strategy Performance**: The return of the "Buy and Hold" strategy is calculated by subtracting the initial balance from the final portfolio value
- **Results**: The strategy's reward (final return) is printed at the end

Momentum (Buy when the trend is up, sell when it's down)

```
portfolio_values_momentum = []
balance = 10000
position = 0

for i in range(1, len(test_data)):
    current_price = train_data['Close'].iloc[i]

    if test_data['MACD'].iloc[i] > 0 and position == 0:
        position = balance // current_price
        balance -= position * current_price
    elif test_data['MACD'].iloc[i] < 0 and position > 0:
        balance += position * current_price
        position = 0

    portfolio_value = balance + position * current_price
    portfolio_values_momentum.append(portfolio_value)

momentum_reward = portfolio_values_momentum[-1] - 10000
print(f"Momentum Reward: {momentum_reward}")
```

- **Initial Setup**: The "Momentum" strategy is applied with an initial balance of $10,000 and no initial position
- **Strategy Logic**: The strategy buys stocks when the MACD value is positive (indicating upward momentum) and sells when the MACD value is negative (indicating downward momentum)
- **Portfolio Value Calculation**: At each step, the portfolio value is updated based on the current price and the position held
- **Strategy Performance**: The return of the "Momentum" strategy is calculated by subtracting the initial balance from the final portfolio value
- **Results**: The strategy's reward (final return) is printed at the end

Contrarian (Sell when the trend is up, buy when it's down)

```
portfolio_values_contrarian = []
balance = 10000
position = 0

for i in range(1, len(test_data)):
    current_price = test_data['Close'].iloc[i]

    if test_data['MACD'].iloc[i] < 0 and position == 0:
        position = balance // current_price
        balance -= position * current_price
    elif test_data['MACD'].iloc[i] > 0 and position > 0:
        balance += position * current_price
        position = 0

    portfolio_value = balance + position * current_price
    portfolio_values_contrarian.append(portfolio_value)

contrarian_reward = portfolio_values_contrarian[-1] - 10000
print(f"Contrarian Reward: {contrarian_reward}")
```

- **Initial Setup**: The "Contrarian" strategy is applied with an initial balance of $10,000 and no initial position
- **Strategy Logic**: The strategy buys stocks when the MACD value is negative (indicating downward momentum) and sells when the MACD value is positive (indicating upward momentum), going against the trend
- **Portfolio Value Calculation**: At each step, the portfolio value is updated based on the current price and the position held
- **Strategy Performance**: The return of the "Contrarian" strategy is calculated by subtracting the initial balance from the final portfolio value.
- **Results**: The strategy's reward (final return) is printed at the end

6. Portfolio of models and strategies

Rewards

total_test_rewarddqn1	450337.516708374
total_test_rewarddqn2	382632.03
total_reward_ppo	-246131.16
buy_and_hold_reward	3610.574264526367
momentum_reward	8022.167610168457
contrarian_reward	2529.176315307617

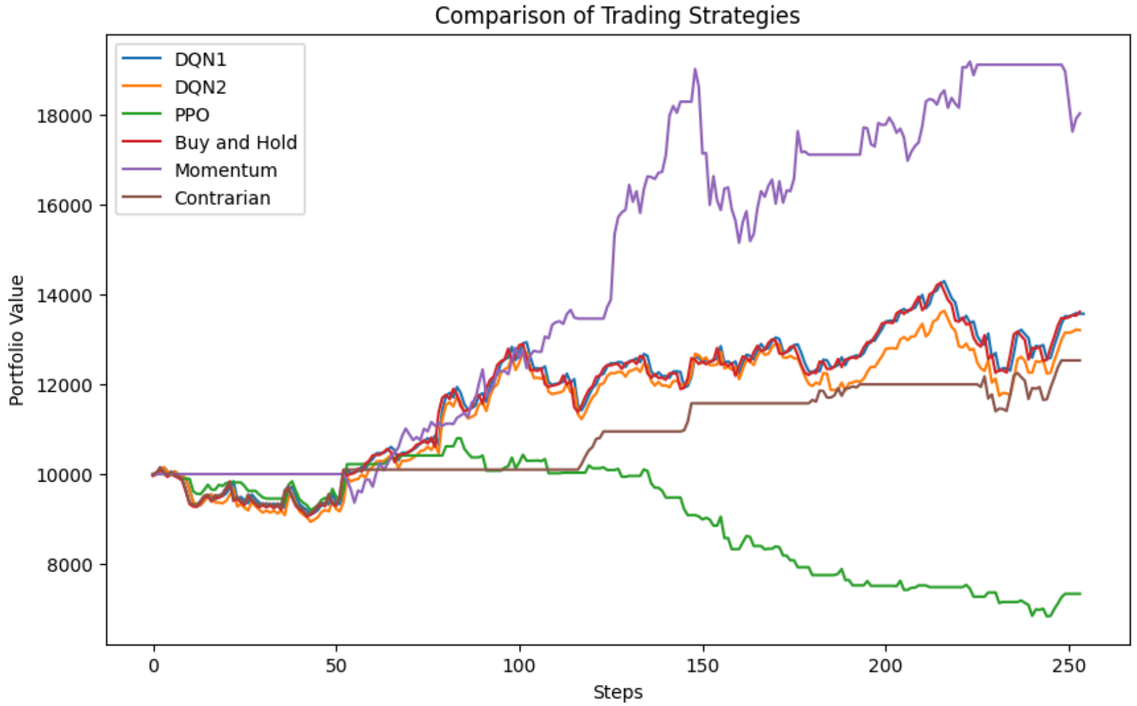
In terms of rewards:

- **DQN1** (450K) & **DQN2** (382K) → Best performers, capturing market patterns well.
- **PPO** (-246K) → Worst performer, likely due to poor adaptation.
- **Momentum** (8K) > **Buy & Hold** (3.6K) > **Contrarian** (2.5K) → Trend-following works better

Key Takeaways:

- DQN1 outperforms all strategies
- Momentum is effective but weaker than RL
- PPO fails to learn a viable strategy
- Next Steps: Improve PPO tuning and explore hybrid models

Portfolio



- **Momentum**: Best performance (~18,000+), but highly volatile
- **DQN1 & DQN2:** Stable growth (~12,500-13,000), but not optimal
- **Buy & Hold**: Robust (~12,500), effective in bullish markets
- **Contrarian**: Defensive (~11,500-12,000), limits gains
- **PPO**: Worst performance (<8,000), continuous losses
- **Improvements**:
- Optimize PPO
- Test a Momentum + DQN hybrid
- Reduce Momentum volatility

Metrics of Success

Return on Investment (ROI)

Measures the percentage gain or loss on the initial capital.

- Most intuitive measure of performance.
- Directly tells us how much profit the strategy generated.

$$ROI = \frac{Final\ Portfolio\ Value - Initial\ Investment}{Initial\ Investment}$$

Volatility

- Measures the degree of variation in returns over time.
- High volatility → More risk, Low volatility → More stability.
- A good trading strategy should balance high returns with controlled risk.
- Helps assess the stability of our DQN model.

Max Drawdown

Largest decline from a peak portfolio value to a trough before recovery.

- Important risk measure in trading.
- A model with a high drawdown is risky, even if returns are high.

$$Max\ Drawdown = \frac{Peak\ Value - Lowest\ Value}{Peak\ Value}$$

Metrics of Success

Sharpe Ratio

Measures risk-adjusted return by comparing returns to volatility.

- Rewards models with high returns & low risk.
- Used widely in finance to compare strategies.

$$Sharpe\ Ratio = \frac{Mean\ Returns - Risk\ Free\ Rate}{Standard\ Deviation\ of\ Returns}$$

Sortino Ratio

- Like the Sharpe Ratio, but only considers downside risk (negative volatility).
- Better than Sharpe Ratio for trading, since we care more about downside risk.

$$Sortino\ Ratio = \frac{Mean\ Returns - Risk\ Free\ Rate}{Downside\ Deviation}$$

Calmar Ratio

Compares annual return to max drawdown.

- Balances profitability & risk exposure.
- Preferred for long-term strategy evaluation.

$$Calmar\ Ratio = \frac{Annualized\ Return}{Max\ Drawdown}$$

Metrics of Success: the results

	ROI	Volatility	Max Drawdown	Sharpe Ratio	Sortino Ratio	Calmar Ratio
DQN1	0.356638	0.014828	0.139852	0.088421	0.130626	0.009375
DQN2	0.320141	0.014734	0.138621	0.081888	0.122421	0.008704
PPO	-0.266200	0.011459	0.366860	-0.100956	-0.109855	-0.003153
Buy and Hold	0.366668	0.014914	0.140255	0.090271	0.133672	0.009599
Momentum	0.802217	0.017990	0.203335	0.138515	0.156759	0.012255
Contrarian	0.252918	0.010287	0.104428	0.091793	0.090521	0.009042

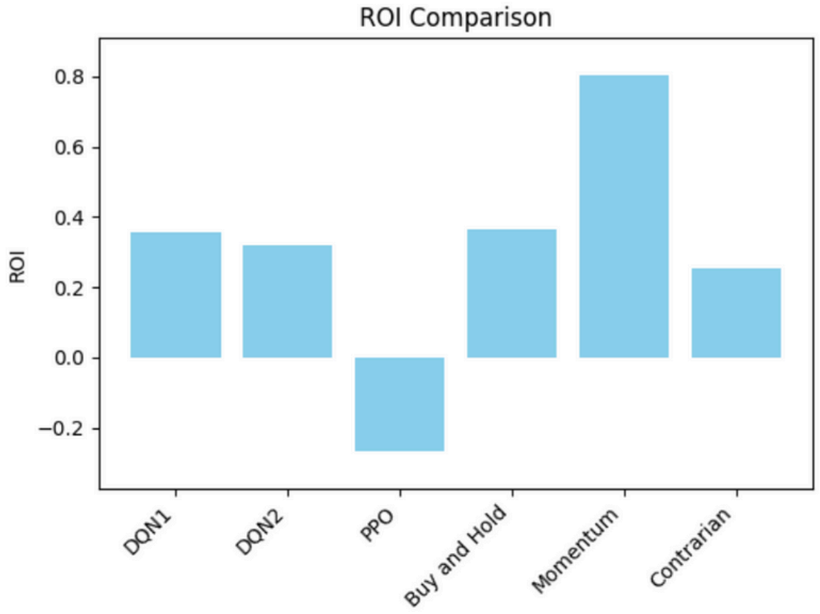
- Momentum strategy achieved the highest ROI but came with higher volatility and drawdown.
- DQN1 & DQN2 performed close to Buy & Hold, but did not significantly outperform traditional strategies.
- PPO model failed, showing negative returns and high drawdown, indicating poor learning stability.

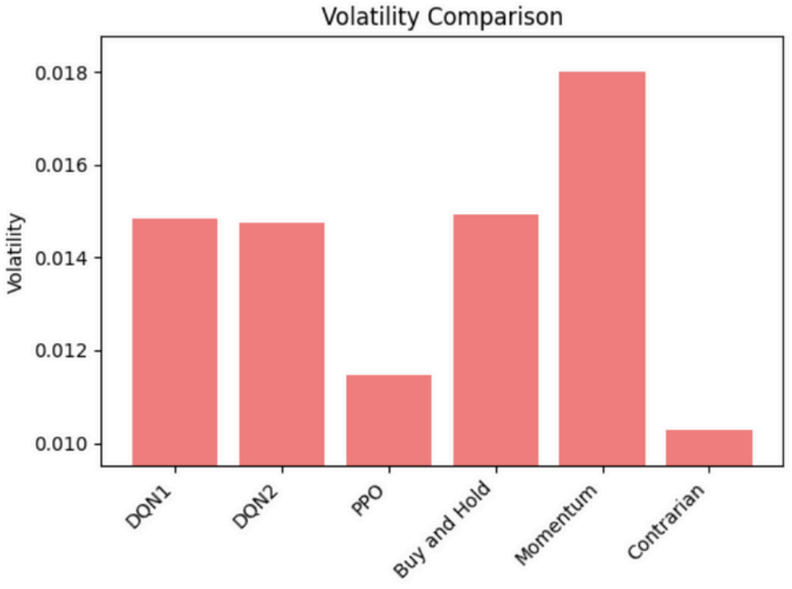
Deep RL vs. Traditional Strategies

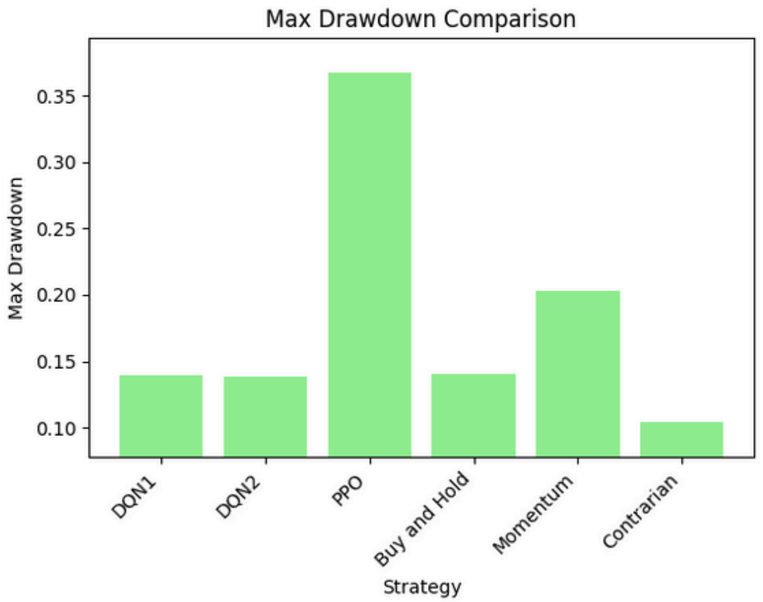
→ Deep RL models struggled to consistently outperform traditional benchmarks.

→ Risk-adjusted metrics (Sharpe & Sortino) did not show a clear advantage for DRL models.

Metrics of Success: strategy performance comparison







Conclusion

Project summary

- Implemented Deep Reinforcement Learning (DRL) agents for trading using DQN and PPO.
- Compared their performance against traditional trading strategies like Buy & Hold and Momentum.
- Used real-world financial data to test adaptability and profitability.

Takeaways of our analysis

- Momentum strategy had the highest returns but with higher risk.
- DQN models performed similarly to Buy & Hold, showing potential but not a clear advantage.
- PPO failed to learn profitable strategies, highlighting stability issues.

Oumou SOW
Lath ESSOH
Tsiba RAZAFINDRAKOTO

Thank You

for your attention