1. Introduction

This project simulates financial markets using multiple stock datasets with the goal of maximizing profit over a longer period using advanced reinforcement learning techniques.

2. Trading Environments

Selecting the right environment for training our reinforcement learning trading bot was a challenging task due to the complexity of financial markets. Key factors such as action space, customizability, and the ability to replicate realistic market conditions play a critical role in training effective agents. We've tested various environments to evaluate these aspects and ensure the bot's adaptability. Below is a list of the environments we have used:

a. The *StockTradingEnv* class used in this project is a multi-stock trading environment designed to simulate realistic financial markets, making it ideal for developing and evaluating decision-making agents in portfolio management tasks. It provides a controlled market simulation where an agent can interact by making trades based on multiple stock prices and portfolio information.
b. The GymTradingEnv class allows for trading a single asset pair. The action space is defined by a discrete list of positions that the agent can take, which must be set in advance. Parameters such as the trading fees, the borrowing fees (for leverage), the number of previous states to be included provided a complex market simulation where the agent could train.

3. Data Description

This project utilizes multiple datasets to provide a comprehensive view of market conditions and to ensure robust testing of the reinforcement learning trading bot. By combining data from different sources, we were able to simulate a variety of trading scenarios and evaluate the bot's performance in diverse environments.

- Stocks Dataset:
  - Historical market data fetched from Yahoo Finance on the 1-day timeframe.
    - Includes Open, High, Low, Close prices, and Volume for each ticker.
  - Technical indicators calculated:
    - RSI: Measures momentum and potential overbought/oversold conditions.
    - MACD: Trend-following momentum indicator using EMAs.
    - Signal: A line used in conjunction with MACD.
    - CCI: Identifies cyclical price movements.
    - ADX: Measures the strength of trends using directional movement indicators.
  - Final dataset includes: [Open, High, Low, Close, Volume, MACD Signal, RSI, CCI, ADX]
- Crypto Dataset:
  - Downloaded from Binance on the 1-minute timeframe.
  - Data processed to ensure reliability: splitted data into buckets to ensure consecutive candles.
  - Technical indicators calculated:
    - RSI: Measures momentum.
    - EMA: Exponential Moving Average for trend analysis.
    - MACD: Trend-following momentum indicator.
  - Data aggregated to the 5-minute timeframe for better trend analysis.
  - Final dataset includes [Open, High, Low, Close, Volume, MACD Signal, RSI, Number of trades]

Environment Description:

a. *StockTradingEnv*

- **State Space**: The state space consists of multiple features representing the market state and the agent's portfolio across several stocks. These features may include historical price data, technical indicators, the agent's cash balance, current step, and the number of shares held for each stock. The continuous state space offers a comprehensive representation for the RL agent to make informed decisions.

```
""" Observation space:
    - "self.n_features * len(self.tickers)" = Each stock's data
    - "2" = account balance and net worth
    - "len(self.tickers)" = number of shares held for each stock
    - "2" = maximum net worth and current step
"""
self.obs_shape = self.n_features * len(self.tickers) + 2 + len(self.tickers) + 2
self.observation_space = spaces.Box(low=-np.inf, high=np.inf, shape=(self.obs_shape,), dtype=np.float32)
```

```
    def _next_observation(self):
        frame = np.zeros(self.obs_shape)

        idx = 0
        for ticker in self.tickers:
            df = self.stock_data[ticker]
            if self.current_step < len(df):
                frame[idx:idx+self.n_features] = df.iloc[self.current_step].values
            elif len(df) > 0:
                frame[idx:idx+self.n_features] = df.iloc[-1].values
            idx += self.n_features

        frame[-4-len(self.tickers)] = self.balance
        frame[-3-len(self.tickers):-3] = [self.shares_held[ticker] for ticker in self.tickers]
        frame[-3] = self.net_worth
        frame[-2] = self.max_net_worth
        frame[-1] = self.current_step

        return frame
```

- **Action Space**: The action space is continuous and allows the agent to simultaneously buy, sell, or hold multiple stocks. Actions are represented as percentage changes in stock positions or cash allocations, making the environment suitable for policy gradient methods.

```
self.action_space = spaces.Box(low=-1, high=1, shape=(len(self.tickers),), dtype=np.float32)
```

- **Reward Structure**: The reward function is designed around the financial performance of the agent's decisions. It often reflects the portfolio value changes, profit and loss (PnL), or returns over a trading period, encouraging the agent to maximize profits while managing risks.

```
self.net_worth = self.balance + sum(self.shares_held[ticker] * current_prices[ticker] for ticker in self.tickers)
self.max_net_worth = max(self.net_worth, self.max_net_worth)

reward = self.net_worth - self.initial_balance
```

- **Episode Termination:** An episode concludes when a predefined time horizon is reached, or the agent depletes its capital. This structure allows the agent to experience various market conditions across multiple stocks within a single episode.

```
done = self.net_worth <= 0 or self.current_step >= self.max_steps
```

- **Market Dynamics**: The environment can simulate realistic multi-stock market behavior, including price fluctuations, transaction costs, and constraints like limited cash availability. This complexity ensures the agent learns to balance risk and reward effectively.

b. *GymTradingEnv*

- **Observation Space**: Features in this environment are divided into static and dynamic types. Static features are pre-computed before the simulation starts, such as price changes and volume ratios, providing a consistent input for the agent. Dynamic features are calculated at each step, like the last position taken or the real portfolio position, allowing the agent to adapt to real-time market changes. Both types enrich the observation space, enhancing the agent's decision-making process.

```
df["feature_close"] = df["close"].pct_change()
df["feature_high"] = df["high"].pct_change()
df["feature_low"] = df["low"].pct_change()
df["feature_open"] = df["open"].pct_change()

df["feature_ema15"] = talib.EMA(df["close"], timeperiod=15).pct_change()
df["feature_ema30"] = talib.EMA(df["close"], timeperiod=30).pct_change()
df["feature_ema60"] = talib.EMA(df["close"], timeperiod=60).pct_change()
df["feature_ema90"] = talib.EMA(df["close"], timeperiod=90).pct_change()

df["feature_rsi"] = talib.RSI(df["close"], timeperiod=14) / 100

df = df.dropna().drop_duplicates()
return df
```

- **Action Space**: In this environment, the action space, which must be set in advance, revolves around positions rather than traditional buy/sell actions. Each position is represented by a discrete number, indicating how the portfolio is allocated. For instance, for the BTC/USD pair:
    - 1: Full portfolio in BTC (buy all).
    - 0: Full portfolio in USD (sell all).
    - 0.5: 50% in BTC, 50% in USD.

  Complex positions, like leveraging and short selling, are also supported:

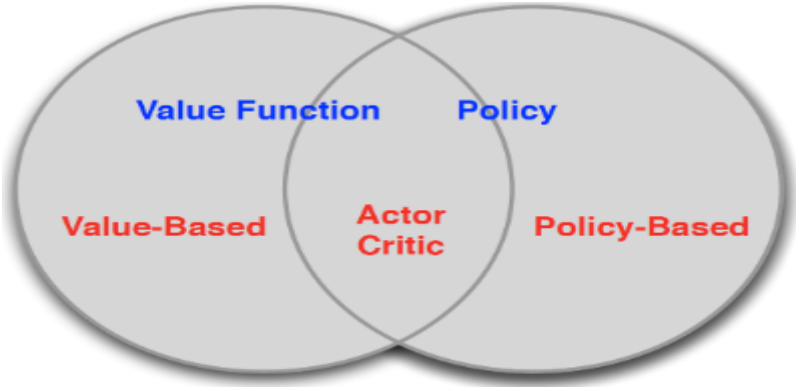
    - **-1**: Short 100% of the portfolio value in BTC.
    - **2**: Leverage 100% on BTC's rise.

```
positions = np.linspace(
    self.positions_min, self.positions_max, self.positions_count
)
if 0 not in positions:
    positions = np.sort(np.insert(positions, 0, 0)).tolist()
```

- **Reward Function:** The reward function in this environment is customizable and takes the trading history as its input. This allows users to define rewards based on specific strategies or outcomes, tailoring the agent's learning process to align with their unique trading goals.

```
def reward_function(history):
    return np.log(
        history["portfolio_valuation", -1] / history["portfolio_valuation", -2]
    )
```

4. **Methods and Algorithms**



The project uses multiple reinforcement learning algorithms such as:

- **A2C (Advantage Actor-Critic)**: An actor-critic method where the policy and value functions are updated simultaneously to improve stability and performance.
- **PPO (Proximal Policy Optimization)**: A policy gradient method with a clipping mechanism for more stable updates.
- **DDPG (Deep Deterministic Policy Gradient)**: A model-free, off-policy algorithm suited for continuous action spaces.
- **TD3 (Twin Delayed Deep Deterministic)**: An off-policy algorithm based on DDPG that addresses many of DDPG's weaknesses.

Additionally, the ***DQN (Deep Q-Network)*** *algorithm* was tested on a discrete environment to evaluate its performance compared to the continuous action-space models.

5. **A2C (Advantage Actor-Critic)**

In the *A2C (Advantage Actor-Critic)* algorithm, the goal is to improve both the actor (policy) and critic (value function) simultaneously.

A2C Algorithm Overview:

1. **Policy (Actor)**: The actor's objective is to learn a policy function that maximizes expected cumulative rewards. The policy is represented by the actor network, which outputs a distribution (e.g., Gaussian) from which actions are sampled. The action is then executed in the environment.
2. **Value Function (Critic)**: The critic estimates the value function, *V(s)*, which gives an estimate of the expected return starting from state *s*. The critic helps in computing the **advantage** (how much better or worse an action is compared to the value estimate).
3. **Advantage Calculation**: The advantage is computed using the *Temporal Difference (TD)* error:

$$A_t = \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

4. **Loss Function**:
    - **Critic Loss**: The critic is updated by minimizing the mean squared error between the predicted value and the TD target:

$$L_{\text{critic}} = \mathbb{E}\left[(V(s_t) - \delta_t)^2\right]$$

    - **Actor Loss**: The actor is updated by maximizing the expected advantage (policy gradient). The actor loss is given by:

$$L_{\text{actor}} = -\mathbb{E}\left[\log \pi(a_t|s_t) \cdot A_t\right]$$

Implementation Details:

In the ***Actor*** class, we simulate a realistic trading environment by using a **normal distribution** for action selection, allowing the agent to make continuous decisions, such as buying or selling fractional amounts of stocks. This approach gives the agent **better control** over portfolio adjustments, closely mimicking real-world trading behavior. By sampling actions based on the learned **mean** and **standard deviation**, the model captures

market uncertainty and ensures **stable exploration**, leading to more effective decision-making in dynamic financial markets.

```python
# PolicyNetwork
class Actor(nn.Module):
    def __init__(self, input_size, num_actions):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3_mean = nn.Linear(32, num_actions)
        self.fc3_std = nn.Linear(32, num_actions)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        mean = torch.tanh(self.fc3_mean(x))
        std = torch.sigmoid(self.fc3_std(x)) + 1e-6
        return mean, std
```

In the ***train*** method, the agent uses a normal distribution to select actions based on the mean and standard deviation output by the actor network. The log probability (***log_prob***) of the selected action is calculated using ***dist.log_prob(action)***, which measures how *likely the chosen action is according to the learned policy*. This is then used to compute the ***actor loss***, which is the product of the log probability and the advantage. The advantage represents how much better or worse the selected action is compared to the expected value, helping the actor network improve its policy over time.

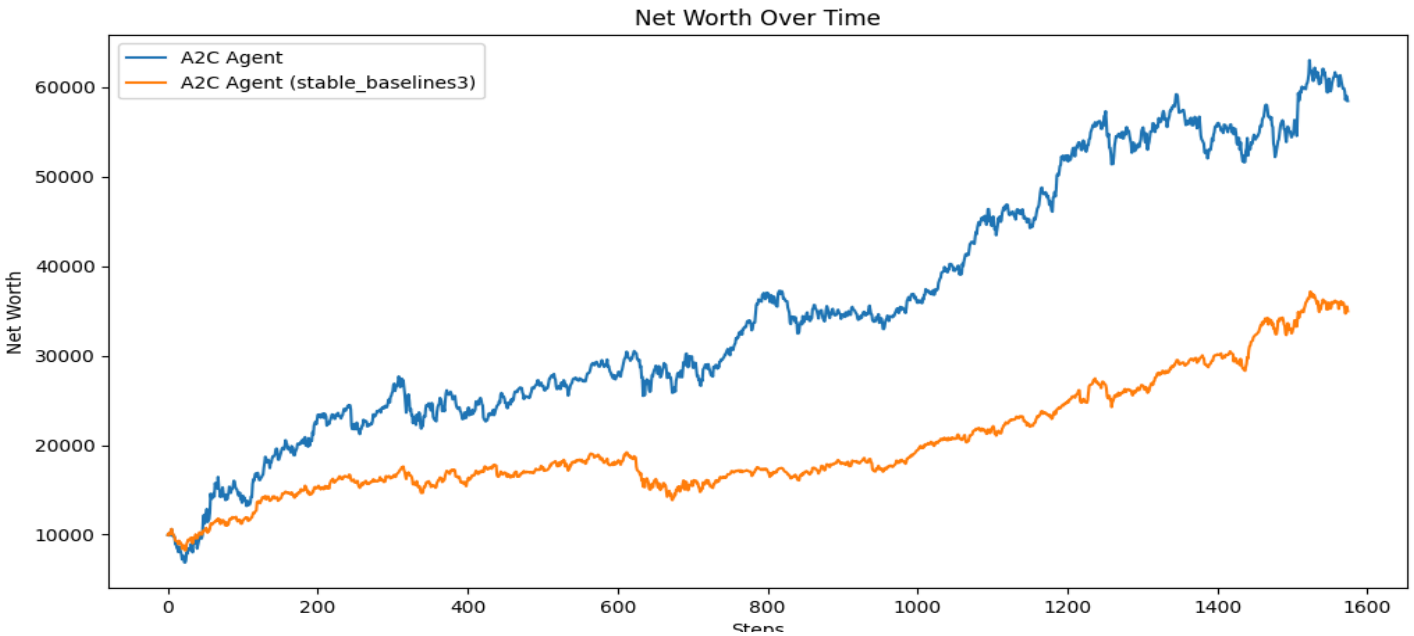
```python
# V(s), V(s')
value = self.critic(state_tensor)
next_value = self.critic(torch.FloatTensor(next_state))

# TD target & TD advantage
td_target = reward + self.gamma * next_value * (1 - done)
advantage = td_target - value

# Update critic
critic_loss = F.mse_loss(value, td_target.detach())
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()

# Update actor
log_prob = dist.log_prob(action)
actor_loss = -(log_prob * advantage.detach()).mean()
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()
```

Stable-Baselines3 comparison (50 epochs):



5. **PPO (Proximal policy optimization)**

The PPO algorithm is a widely used reinforcement learning method because it is simple, efficient, and reliable. It improves the policy by using a special objective function that prevents the model from making overly large updates, which helps maintain stable and steady training. This balance between exploration and optimization makes PPO effective in various applications, such as gaming, robotics, and trading.

PPO Algorithm Overview:

- ***Policy (Actor)***: Similar to *A2C*, the actor in PPO learns a policy to select actions that maximize expected cumulative rewards. However, unlike A2C, PPO ensures more stable updates by clipping changes to the policy during training. This prevents drastic deviations from the current policy.
- ***Value Function (Critic)***: The critic, like in A2C, estimates the value function V(s) to compute the advantage. The key difference in PPO is that the advantage is used in a clipped surrogate objective to maintain stability while optimizing the actor.

PPO uses the *Generalized Advantage Estimation (GAE)* to compute the advantage, defined as:

$$A(s,a) = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots$$

, where $$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

Loss functions:

1. ***Actor Loss***: PPO optimizes the following clipped objective:

   $$L^{CLIP}(\theta) = \mathbb{E}\left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right]$$

   ,

   where $$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$$

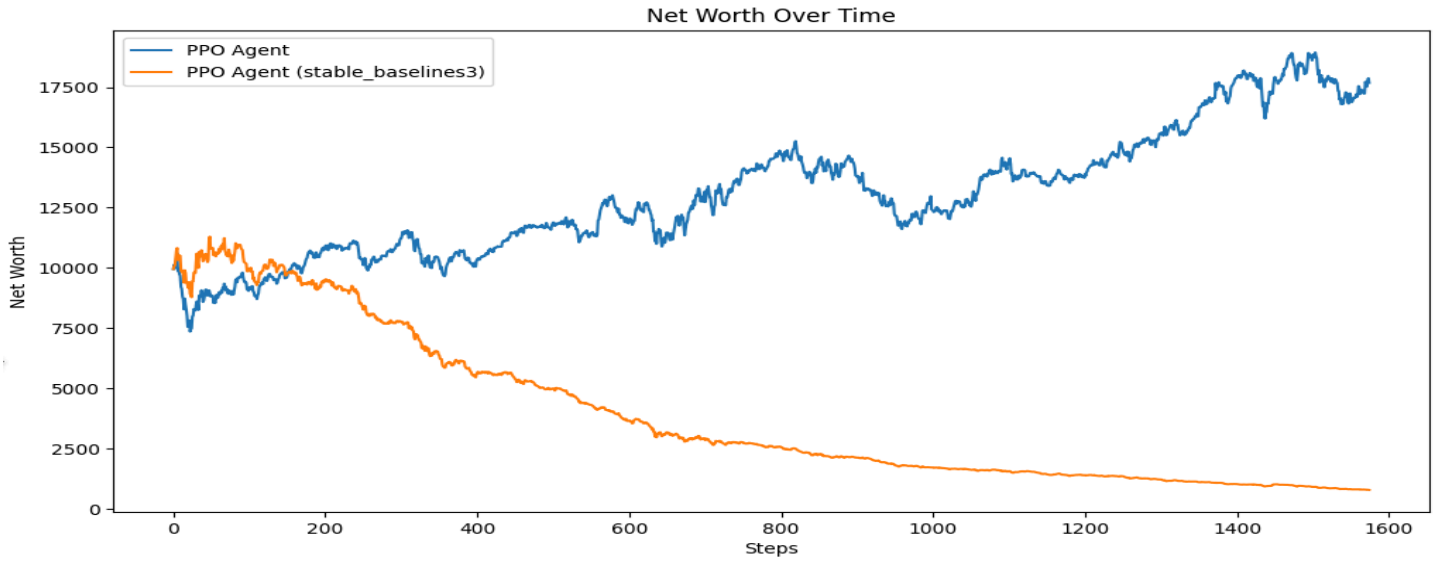
2. ***Critic Loss***: The critic minimizes the Mean Squared Error (MSE) between the predicted value and the target return

$$L^{VF} = \mathbb{E}[(V(s_t) - R_t)^2]$$

Implementation details:

- **Clipping (Policy Update):** Ensures that the updated policy does not deviate too much from the previous policy.
- **Entropy Coefficient:** Controls the trade-off between exploration and exploitation.
- **Gradient Clipping:** Ensures stability by limiting the magnitude of gradients during backpropagation.

Stable-Baselines3 comparison (50 epochs):



6. DDPG (Deep Deterministic Policy Gradient)

DDPG is an **improvement of A2C** and an **off-policy** algorithm (an algorithm that evaluates & improves another policy separate from the policy that generates the actions), meaning that here we introduce besides the existing Actor and Critic networks, **a target Actor and Critic.**

- These additional two neural networks are intended to act like **supervisors** of the initial Actor and Critic networks by **updating** themselves in a **slower and more stable** way . While the **update** of **both** the normal and target networks happens **once per step**, the Target networks are updated by transferring to them only a fraction (**Tau)** of the weights of the normal networks by **Polyak Averaging** formula, called a '**soft update**':

Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$$

This way, while the simple Actor and Critic networks have more **freedom to explore various actions** and update themselves more dramatically, the Target networks provide **stability**, and by comparing the Value Functions, we have a clearer idea of what to do further.

- Another improvement from A2C is that as the training progresses, at every step w**e add each new set of variables** (`state, action, reward, next_state, done`) in a **buffer** for replaying them later. At every step, we are then picking a **random sample of sets** from this buffer and improving the networks based on all of them.

```
next_state, reward, done, _, _ = self.env.step(action)
self.replay_buffer.add((state, action, reward, next_state, done))
```

```
action = self.actor(state_tensor).detach().numpy()
action += np.random.normal(0, 0.1, size=self.num_actions)  # add Gaussian noise for exploration
  states, actions, rewards, next_states, dones = self.replay_buffer.sample(batch_size)
```

- We also **add noise** at every step after generating an action for more exploration:

In the rest, the algorithm works as following at every step of the training:

- With the Actor NN we generate an action for the current state
- From our state with the action we obtain the next state

* From this point on, for calculations involving Target networks, we use the entire sample of sets of (`state, action, reward, next_state, done`) we picked from the Replay Buffer

- With the **Target** Actor NN we generate 'next actions' that should be made from the 'next states'
- With the **Target** Critic NN we generate 'Target Q-values' for the next states & actions
- We calculate **Temporal Difference Targets (the optimal action-value function)** from the Target Q-values using the **Bellman Formula**:

$$r + \gamma(1-d)\max_{a'} Q_\phi(s', a')$$

```
td_targets = rewards + self.gamma * target_q_values * (1 - dones)
```

- With the Critic NN we generate '**Q-values**' for the states & actions picked from the buffer

The Critic's loss is the Mean Squared Error between the Q-values and the TD Targets (*B here is the sample/batch picked from the Buffer)*:

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d)\in B} (Q_\phi(s,a) - y(r,s',d))^2$$

$$y(r,s',d) = r + \gamma(1-d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

```
critic_loss = F.mse_loss(q_values, td_targets.detach())
```

The Actor's loss is the Mean of Q-values, multiplied by -1 :

$$\nabla_\theta \frac{1}{|B|} \sum_{s\in B} Q_\phi(s, \mu_\theta(s))$$

7. **TD3 (Twin Delayed Deep Deterministic Policy Gradient)**

TD3 is an **improvement over DDPG** designed to enhance stability and performance in environments with continuous action spaces. While DDPG introduced mechanisms like Target Networks and Replay Buffers to improve stability, **TD3 addresses several limitations** of DDPG, such as **overestimation bias** and **instability** in Actor updates. It builds on the same framework of Actor-Critic and Target Networks, with additional safeguards to improve learning efficiency and policy robustness.

Key improvements

- **Clipped Double Q-Learning**: In DDPG, a single Critic network is used to estimate the Q-values, which can lead to overestimation bias. TD3 we have two Critic networks that compute two separate Q-value estimates. The minimum of these Q-values is used as the target, effectively reducing overestimation and improving stability.

```
# Calculate target Q-values
next_q1 = self.critic1_target(torch.cat([next_states, next_actions], dim=1))
next_q2 = self.critic2_target(torch.cat([next_states, next_actions], dim=1))
target_q = rewards + self.gamma * (1 - dones) * torch.min(next_q1, next_q2)
```

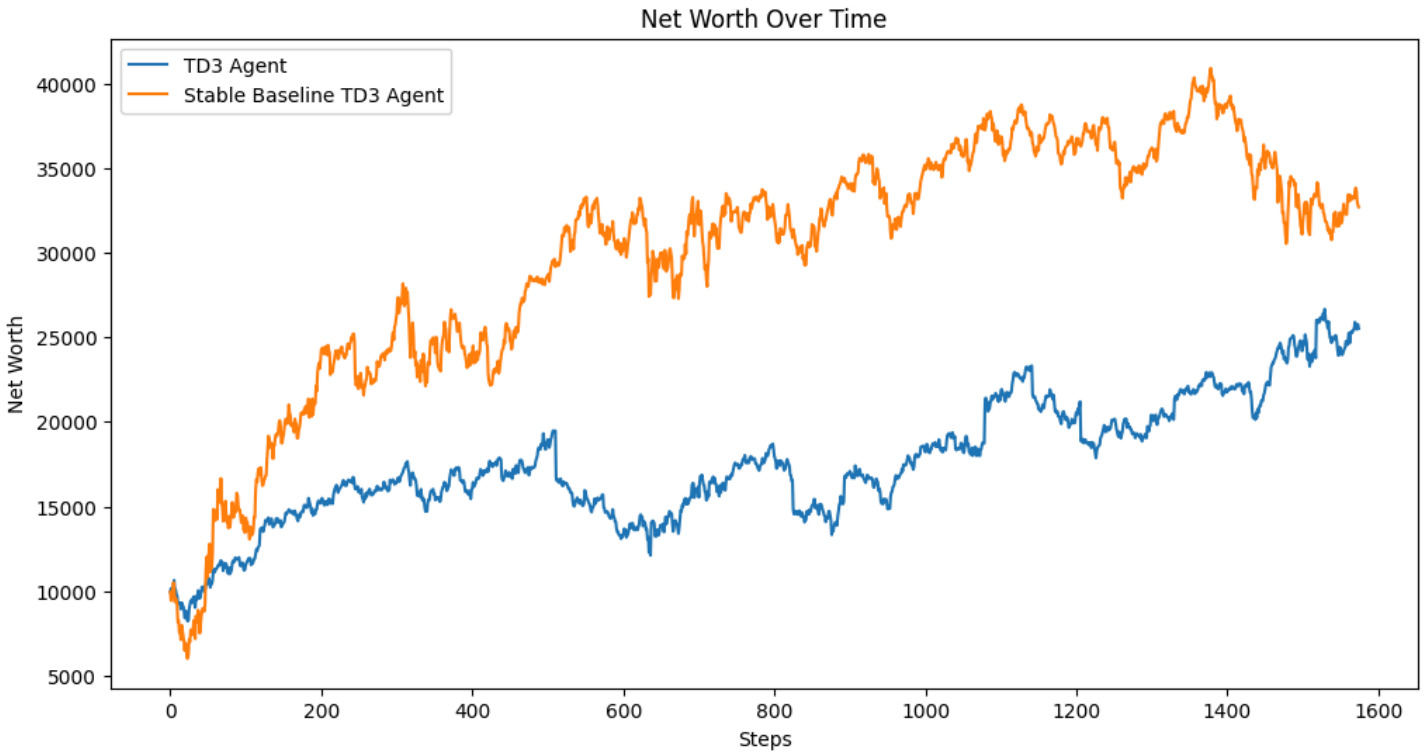
- **Delayed Policy Updates**: In DDPG, both the Actor and Critic networks are updated at the same frequency. TD3, however, we delay the Actor updates, performing one Actor update for every two Critic updates. This ensures that the policy is updated using a more stable Critic network. (Here self.policy_delay is equal to 2).

```
# Delayed Actor update
if steps % self.policy_delay == 0:
    actor_loss = -self.critic1(torch.cat([states, self.actor(states)], dim=1)).mean()
    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()
```

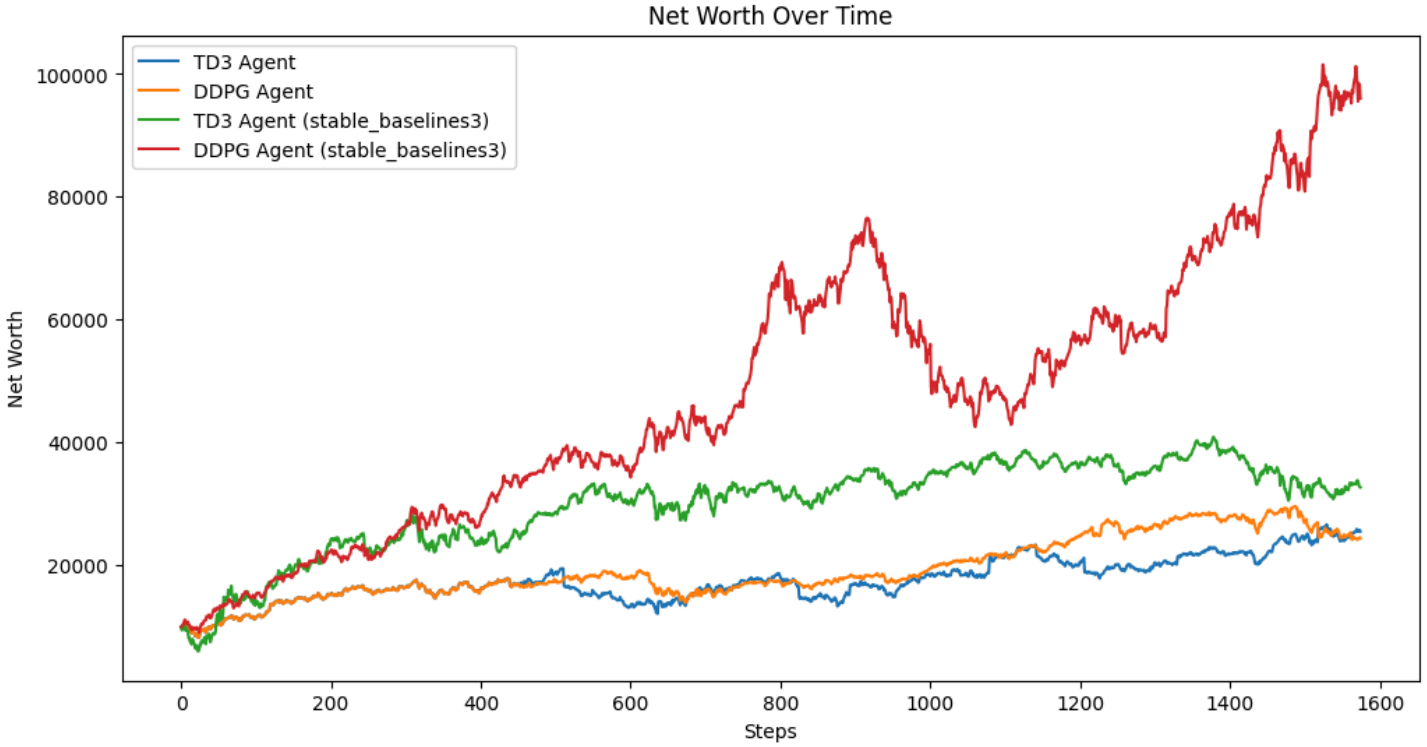
- **Target Policy Smoothing**: TD3 adds noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

```
# Add noise to target actions and clip
noise = torch.clamp(
    torch.normal(0, self.noise_std, size=actions.shape), -self.noise_clip, self.noise_clip
)
next_actions = self.actor_target(next_states) + noise
next_actions = torch.clamp(next_actions, -1, 1)
```

Stable-Baselines3 comparison (50 epochs):



Comparison between TD3, DDPG and Stable-Baselines3 (50 epochs):



8. Deep Q Networks

Deep Q-Network (DQN) is a reinforcement learning algorithm that uses neural networks to approximate the optimal action-value function $Q(s,a) = R(s,a) + \gamma \sum P(s,a,s')V(s')$. It helps agents make decisions in high-dimensional environments, such as financial markets, by learning to predict the expected returns for actions in different states. Key features of DQN include experience replay, which improves learning stability by randomizing the order of experiences, and target networks, which stabilize training by providing a more consistent learning target.

In our project, both a custom DQN implementation and the Stable-Baselines3 library were used to train a trading bot. However, the results were unreliable, with a mean profit of 0.44% on the validation set and a median profit of -0.93%, highlighting the challenges of applying DQN in financial markets.

Resources:

- https://spinningup.openai.com/en/latest/algorithms/ddpg.html
- https://medium.com/@amaresh.dm/how-ddpg-deep-deterministic-policy-gradient-algorithms-works-in-reinforcement-learning-117e6a932e68
- https://stable-baselines3.readthedocs.io/en/master/
- https://github.com/theanh97/Deep-Reinforcement-Learning-with-Stock-Trading/blob/main/notebooks/main.ipynb
- https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ
- https://gym-trading-env.readthedocs.io/en/latest/
- https://medium.com/@dixitaniket76/advantage-actor-critic-a2c-algorithm-explained-and-implemented-in-pytorch-dc3354b60b50
- https://medium.com/@danushidk507/ppo-algorithm-3b33195de14a
- https://spinningup.openai.com/en/latest/algorithms/td3.html
-