1 Introduction

The stock market has always been a fascinating arena for investors and traders alike, with its unpredictable fluctuations and endless opportunities for profit. As technology advances, so do the tools available to traders seeking an edge in this competitive field. One of the most promising approaches in recent years is the use of artificial intelligence, specifically reinforcement learning (RL), in quantitative trading. RL is a type of machine learning that enables an agent to learn through trial-and-error interactions with an environment to maximize a cumulative reward signal. In finance, the environment can be a market, and the reward signal can be the profit or loss of a trading strategy.

In this tutorial, we'll leverage the power of Alpaca's platform and use artificial intelligence, specifically reinforcement learning (RL), to build an autonomous trading system. With the FinRL module, we can conduct paper trading on Alpaca and use Optuna for hyperparameter tuning, allowing us to create an RL agent that can learn from its interactions with the stock market environment and make profitable trading decisions while minimizing risk.

To get started, we'll set up our environment by installing the necessary dependencies and obtaining API keys from Alpaca. From there, we'll dive into the FinRL Paper Trading module, which will teach us how to build and train an RL agent using historical market data. Then, we'll explore the FinRL Hyperparameter Tuning module, which uses the Optuna optimization framework to search for the best hyperparameters for our RL agent, further improving its performance.

Throughout the tutorial, I'll provide code snippets to help you better understand the concepts and implementations. By the end of this tutorial, you'll have a solid understanding of how to build an autonomous trading system using RL and be able to apply these techniques to your own trading strategies.

2 Setting up your environment

Before we can start building our RL trading agent, we need to set up our environment by installing the necessary dependencies and obtaining API keys from Alpaca. In this section, I'll walk you through the steps required to set up your environment.

2.1 Requirements and dependencies

To get started, we'll need to install the following dependencies:

- Python 3.6 or higher
- NumPy
- pandas
- PyTorch

- Tensorboard
- TensorboardX
- optuna
- ta
- yfinance

2.2 Cloning the FinRL tutorials repository

Next, we'll clone the FinRL-Tutorials repository, which contains the Jupyter notebooks we'll be using for this tutorial. To do this, run the following command in your terminal:

```
git clone https://github.com/AI4Finance-LLC/FinRL-Tutorials.git
```

This will download the repository to your local machine.

2.3 Setting up an Alpaca account and API keys

To connect to Alpaca's API, you'll need to create an account on the Alpaca website and generate API keys. Here's how:

1. Go to the Alpaca website and click "Sign Up" in the top right corner.
2. Follow the instructions to create your account.
3. Once your account is created, click on your name in the top right corner and select "Account".
4. Click on the "View" button under the "API Keys" section.
5. Click the "Create New Key" button and follow the instructions to generate your keys.

Make sure to copy your API key and secret key somewhere safe, as we'll need these later to connect to the Alpaca API.

That's it! With your environment set up and API keys generated, we're ready to start building our RL trading agent.

3 Exploring the FinRL Paper Trading Jupyter Notebook

In this section, we'll dive into a FinRL Paper Trading Jupyter Notebook and explore how to implement an RL trading agent that can learn and make trading decisions on historical market data.

3.1 Overview of the notebook structure

The notebook is structured into five main sections:

1. *Data collection and preprocessing*: In this section, we'll collect historical stock data from Yahoo Finance and preprocess it for use in our RL environment.

2. *Creating the RL trading environment*: Here, we'll define the trading environment, which includes the market data and our RL agent.

3. *Implementing and training the agent*: In this section, we'll implement the RL agent using PyTorch and train it using the trading environment we defined in the previous section.

4. *Evaluating the agent's performance*: Once our agent is trained, we'll evaluate its performance on a validation set and analyze the results.

5. *Executing trades on Alpaca*: Finally, we'll use our trained agent to make trades on the Alpaca platform.

```python
# Import necessary libraries
import gym
import finrl
from finrl import config
from finrl.marketdata.yahoodownloader import YahooDownloader
from finrl.preprocessing.preprocessors import FeatureEngineer
from finrl.env.env_stocktrading import StockTradingEnv
from finrl.model.models import DRLAgent
from pprint import pprint
import pandas as pd
import numpy as np
import datetime

# Download and preprocess data using FinRL libraries
df = YahooDownloader(start_date='2023-04-14',
                     end_date='2023-05-03',
                     ticker_list=['AAPL']).fetch_data()

fe = FeatureEngineer(use_technical_indicator=True,
                     use_turbulence=False,
                     user_defined_feature=False)

processed = fe.preprocess_data(df)

# Define RL trading environment
env_kwargs = {
    "hmax": 30,
    "initial_amount": 1000000,
    "buy_cost_pct": 0.001,
    "sell_cost_pct": 0.001,
    "state_space": state_space,
    "action_space": stock_dimension,
    "stock_dim": stock_dimension,
    "tech_indicator_list": config.TECHNICAL_INDICATORS_LIST,
    "reward_scaling": 1e-4
}

# Initialize the environment using the FinRL library
stock_env = StockTradingEnv(df=processed, **env_kwargs)
```

3.2 Customizing the notebook for your trading strategy

To customize the notebook for your own trading strategy, there are three main areas you may want to modify:

1. *Choosing assets and timeframes*: By default, the notebook is set up to trade the stock ”AAPL” on a daily timeframe. However, you can modify this to trade any other stock on any timeframe that is available on Yahoo Finance.
2. *Modifying the RL environment*: The notebook uses a simple RL environment by default, but you can modify this to include additional features or use a different environment entirely.
3. *Adjusting the reward function*: The reward function used in the notebook is designed to maximize profit while minimizing risk. However, you may want to adjust this to prioritize other factors, such as volatility or liquidity.

With these customizations in mind, you can use the FinRL Paper Trading notebook as a starting point to develop and test your own RL trading strategies.

In the next section, we'll look at the specific reinforcement learning agent we'll use.

4 PPO: Proximal Policy Optimization

PPO is a well-known and widely used reinforcement learning algorithm that has shown strong performance in various applications, including finance.

PPO is a policy-based reinforcement learning algorithm, meaning it directly learns a policy (a mapping from states to actions) instead of estimating a value function. This can be advantageous in finance because the optimal action often depends on the current state of the market, and the policy-based approach can better capture these complex dependencies.

PPO is an on-policy algorithm, meaning it learns from the data generated by the current policy. In finance, the market conditions can change rapidly, and using an on-policy algorithm can help the agent adapt quickly to new market conditions.

PPO has been shown to be less sensitive to hyperparameter tuning than other reinforcement learning algorithms, which can make it easier to use and more robust.

The FinRL developers have reported that PPO outperforms other reinforcement learning algorithms in their experiments on a range of financial tasks.

Overall, using PPO for paper trading in finance seems like a reasonable choice, given its strong performance in other domains and its suitability for the challenges of finance. However, it is important to note that the choice of reinforcement learning algorithm ultimately depends on the specific problem and the available data.

P for Proximal: Proximal refers to the way the policy is updated during training, where the new policy is constrained to be "close" to the previous policy. This helps to stabilize the training and prevent the policy from changing too much at once, which can cause instability or poor performance. For example, in PPO, the policy update step includes a clipping operation that constrains the new policy to be within a certain distance from the previous policy.

P for Policy: PPO is a policy-based reinforcement learning algorithm, which means it directly learns a policy (a mapping from states to actions) instead of estimating a value function. The policy can be represented in different ways, such as a neural network or a decision tree, but the goal is to learn a policy that maximizes the expected reward over time. For example, in a trading scenario, the policy might be a function that takes in the current state of the market (e.g., stock prices, news articles, economic indicators) and outputs a decision to buy, sell, or hold a particular stock.

O for Optimisation: PPO is an optimization algorithm that updates the policy to maximize the expected reward over time. The optimization is typically done using stochastic gradient descent, where the gradient of the expected reward with respect to the policy parameters is estimated from a batch of experience data, and the policy parameters are updated in the direction that increases the expected reward. For example, in a trading scenario, the optimization might involve updating the neural network parameters to increase the expected profit or decrease the expected loss.

Hyperparamter tuning In reinforcement learning, hyperparameter tuning refers to the process of selecting the best values for the various parameters that are not learned during training but are set by the user before training, such as the learning rate, discount factor, entropy coefficient, etc.

The sensitivity of an algorithm to hyperparameters refers to how much its performance depends on the specific values chosen for those hyperparameters. Some algorithms may have hyperparameters that are very sensitive, meaning small changes in the values can lead to significantly different results. Other algorithms may be more robust to changes in hyperparameters, meaning their performance is less sensitive to changes in those values.

PPO has been shown to be less sensitive to hyperparameter tuning compared to other reinforcement learning algorithms, such as Actor-Critic and DDPG. This means that it is more likely to perform well across a wider range of hyperparameter settings, and that it may be easier to get good results with PPO without needing to spend a lot of time fine-tuning the hyperparameters. This can be an important advantage in practice, where time and computational resources are often limited.

To summarize, PPO is a reinforcement learning algorithm that updates the policy in a proximal way to stabilize training, directly learns a policy to maximize expected reward, and optimizes the policy using stochastic gradient descent.

5 Hyperparameter tuning with Optuna

In the previous section, we explored how to implement an RL trading agent using the FinRL Paper Trading notebook. In this section, we'll dive into the FinRL Hyperparameter Tuning notebook and learn how to use the Optuna optimization framework to automatically search for the best hyperparameters for our RL agent.

5.1 Overview of the Optuna optimization framework

Optuna is a Python library for hyperparameter optimization, which allows us to automate the process of finding the best hyperparameters for our machine learning models. It uses a technique called Bayesian optimization to intelligently search the hyperparameter space and find the best set of hyperparameters for our model.

5.2 Exploring the FinRL Hyperparameter Tuning Jupyter Notebook

The FinRL Hyperparameter Tuning notebook builds upon the FinRL Paper Trading notebook and adds hyperparameter tuning using Optuna. The notebook is structured into three main sections:

1. Defining the search space: Here, we'll define the hyperparameters we want to tune and the search space for each hyperparameter.

2. Implementing the objective function: This section defines the objective function we want to optimize, which in our case is the profit generated by our RL agent on a validation set.

3. Running the optimization process: Finally, we'll run the optimization process using Optuna and find the best set of hyperparameters for our RL agent.

```
# Import necessary libraries
import gym
import finrl
from finrl.model.models import DRLAgent
from finrl.trade.backtest import backtest_stats, backtest_plot
from pprint import pprint
import pandas as pd
import numpy as np
import datetime
import optuna

# Define objective function for Optuna to optimize
def objective(trial):
    model_type = trial.suggest_categorical("model_type", ["ppo",
    ↪  "a2c", "ddpg", "td3"])
    params = {
        "n_steps": trial.suggest_int("n_steps", 128, 2048),
        "batch_size": trial.suggest_int("batch_size", 32, 256),
        "gamma": trial.suggest_float("gamma", 0.8, 0.999,
        ↪  step=0.001),
        "learning_rate": trial.suggest_float("learning_rate",
        ↪  1e-5, 1e-3, log=True),
        "ent_coef": trial.suggest_float("ent_coef", 0, 1e-3,
        ↪  log=True),
    }

    # Initialize the RL agent using the optimized hyperparameters
    agent = DRLAgent(env=stock_env)
    model = agent.get_model(model_name=model_type)
    trained_model = agent.train_model(model=model,
                                   tb_log_name=model_type,

                                   ↪  total_timesteps=params['n_steps'],

                                   ↪  learning_rate=params['learning_rate'],

                                   ↪  batch_size=params['batch_size'],
                                   gamma=params['gamma'],

                                   ↪  ent_coef=params['ent_coef']),

                                   ↪  learning_rate=params['learning_rate'],

                                   ↪  batch_size=params['batch_size'],
                                   gamma=params['gamma'],
                                   n_episodes=5)
    # Evaluate RL agent's performance on validation set
    df_account_value, df_actions =
    ↪  DRLAgent.DRL_prediction(model=trained_model,

                                                              ↪  environment=stock_env)
    stats = backtest_stats(df_account_value)
    return stats['sharpe']

# Use Optuna to optimize hyperparameters for RL agent
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=10)

# Print the best hyperparameters found by Optuna
best_params = study.best_params
print(f"Best Hyperparameters: {best_params}")

# Train RL agent using the best hyperparameters found by Optuna
agent = DRLAgent(env=stock_env)
model = agent.get_model(model_name=best_params['model_type'])
trained_model = agent.train_model(model=model,

                                   ↪  tb_log_name=best_params['model_type'],

                                   ↪  total_timesteps=best_params['n_steps'],

                                   ↪  learning_rate=best_params['learning_rate'],

                                   ↪  batch_size=best_params['batch_size'],
                                   gamma=best_params['gamma'],
                                   n_episodes=5)

# Evaluate RL agent's performance on validation set
df_account_value, df_actions =
↪  DRLAgent.DRL_prediction(model=trained_model,

                                                              ↪  environment=stock_env)
```

5.3 Incorporating tuned hyperparameters into your trading strategy

Once we've found the best set of hyperparameters for our RL agent, we can incorporate these hyperparameters into our trading strategy to further improve its performance. By combining the FinRL Paper Trading and FinRL Hyperparameter Tuning notebooks, we can build an RL trading agent that is optimized for our specific trading strategy.

In the next section, we'll explore how to integrate our RL trading agent and hyperparameter tuning process with the Alpaca platform, allowing us to execute and monitor trades in real-time.

6 Integrating RL trading and hyperparameter tuning on Alpaca

In the previous sections, we explored how to implement an RL trading agent and tune its hyperparameters using the FinRL module and Optuna. In this section, we'll learn how to integrate our RL trading agent and hyperparameter tuning process with the Alpaca platform, allowing us to execute and monitor trades in real-time.

6.1 Combining the notebooks for a streamlined workflow

To combine our RL trading agent and hyperparameter tuning process with Alpaca, we'll need to combine the code from the FinRL Paper Trading and FinRL Hyperparameter Tuning notebooks into a single script. This will allow us to train our RL agent with optimized hyperparameters and use it to execute trades on Alpaca in real-time.

```python
# Import necessary libraries
import gym
import finrl
from finrl.marketdata.yahoodownloader import YahooDownloader
from finrl.preprocessing.preprocessors import FeatureEngineer
from finrl.env.env_stocktrading import StockTradingEnv
from finrl.model.models import DRLAgent
from finrl.trade.backtest import backtest_stats, backtest_plot
from alpaca_trade_api import REST, StreamConn
import pandas as pd
import numpy as np
import datetime
import optuna

# Set up environment by downloading and processing data
df = YahooDownloader(start_date='2009-01-01',
                     end_date='2021-01-01',
                     ticker_list=['AAPL']).fetch_data()

fe = FeatureEngineer(use_technical_indicator=True,
                     use_turbulence=False,
                     user_defined_feature=False)

processed = fe.preprocess_data(df)
```

```
# Define environment parameters
stock_dimension = len(processed.tic.unique())
state_space = 1 + 2*stock_dimension +
↪  len(config.TECHNICAL_INDICATORS_LIST)*stock_dimension
print(f"Stock Dimension: {stock_dimension}, State Space:
↪  {state_space}")

env_kwargs = {
    "hmax": 100,
    "initial_amount": 1000000,
    "buy_cost_pct": 0.001,
    "sell_cost_pct": 0.001,
    "reward_scaling": 1e-4
}

# Initialize environment
stock_env = StockTradingEnv(df=processed, **env_kwargs)

# Define RL agent parameters
agent = DRLAgent(env=stock_env)

# Train RL agent on the environment
model = agent.get_model("ppo")
trained_model = agent.train_model(model=model,
                                   tb_log_name='ppo',
                                   total_timesteps=20000)

# Use Optuna to optimize hyperparameters for RL agent
def objective(trial):
    model_type = trial.suggest_categorical("model_type", ["ppo",
    ↪  "a2c", "ddpg", "td3"])
    params = {
        "n_steps": trial.suggest_int("n_steps", 128, 2048),
        "batch_size": trial.suggest_int("batch_size", 32, 256),
        "gamma": trial.suggest_float("gamma", 0.8, 0.999,
        ↪  step=0.001),
        "learning_rate": trial.suggest_float("learning_rate",
        ↪  1e-5, 1e-3, log=True),
        "ent_coef": trial.suggest_float("ent_coef", 0, 1e-3,
        ↪  log=True),
    }

    # Initialize the RL agent using the optimized hyperparameters
    agent = DRLAgent(env=stock_env)
    model = agent.get_model(model_name=model_type)
    trained_model = agent.train_model(model=model,
                                       tb_log_name=model_type,
                                       
                                        ↪  total_timesteps=params['n_steps'],
                                       
                                        ↪  learning_rate=params['learning_rate'],
                                       
                                        ↪  batch_size=params['batch_size'],
                                       gamma=params['gamma'],
                                       n_episodes=5)

    # Evaluate RL agent's performance on validation set
    df_account_value, df_actions =
    ↪  DRLAgent.DRL_prediction(model=trained_model,
                                                                          
                                                                     ↪  environment=stock_env)
    stats = backtest_stats(df_account_value)
    return stats['sharpe']

study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=10)

# Print the best hyperparameters found by Optuna
best_params = study.best_params
print(f"Best Hyperparameters: {best_params}")
```

6.2 Executing and monitoring trades on Alpaca

Once we've combined our notebooks and set up our script to execute trades on Alpaca, we can begin monitoring our trades in real-time using the Alpaca dashboard. This will allow us to keep track of our profits and losses and make adjustments to our trading strategy as needed.

```
from finrl.marketdata.alpaca.alpaca_paper_trading import
↪  AlpacaPaperTrading

# Set up parameters
DOW_30_TICKER = ['AAPL', 'AXP', 'BA', 'CAT', 'CSCO', 'CVX',
↪  'DIS', 'DOW', 'GS', 'HD', 'IBM', 'INTC', 'JNJ', 'JPM', 'KO',
↪  'MCD', 'MMM', 'MRK', 'MSFT', 'NKE', 'PG', 'TRV', 'UNH', 'VZ',
↪  'V', 'WBA', 'WMT', 'XOM']
ERL_PARAMS = {'net_dimension': 2 ** 9}
state_dim = 1 + 2 * len(DOW_30_TICKER) + len(INDICATORS) *
↪  len(DOW_30_TICKER)
action_dim = len(DOW_30_TICKER)
```

```
# Initialize paper trading environment
paper_trading_erl = AlpacaPaperTrading(ticker_list=DOW_30_TICKER,
                                       time_interval='1Min',
                                       drl_lib='elegantrl',
                                       agent='ppo',

                                       ↪  cwd='./your_current_working_directory',

                                       ↪  net_dim=ERL_PARAMS['net_dimension'],
                                       state_dim=state_dim,
                                       action_dim=action_dim,
                                       API_KEY=API_KEY,
                                       API_SECRET=API_SECRET,
                                       API_BASE_URL=API_BASE_URL,

                                       ↪  tech_indicator_list=INDICATORS,
                                       turbulence_thresh=30,
                                       max_stock=1e2)

# Start paper trading
paper_trading_erl.run()
```

6.3 Analyzing performance and iterating on your strategy

After executing trades on Alpaca, we can analyze the performance of our trading strategy and make adjustments as needed. By using the tools provided by Alpaca and the insights gained from our RL trading agent, we can continuously refine our trading strategy and maximize our profits.

```
import alpaca_trade_api as tradeapi
import exchange_calendars as tc
import numpy as np
import pandas as pd
import pytz
import yfinance as yf
import matplotlib.ticker as ticker
import matplotlib.dates as mdates
from datetime import datetime as dt
from finrl.plot import backtest_stats
import matplotlib.pyplot as plt

def get_trading_days(start, end):
    nyse = tc.get_calendar('NYSE')
    df = nyse.sessions_in_range(pd.Timestamp(start,tz=pytz.UTC),
                                pd.Timestamp(end,tz=pytz.UTC))
    trading_days = []
    for day in df:
        trading_days.append(str(day)[:10])

    return trading_days

def alpaca_history(key, secret, url, start, end):
    api = tradeapi.REST(key, secret, url, 'v2')
    trading_days = get_trading_days(start, end)
    df = pd.DataFrame()
    for day in trading_days:
        df = df.append(api.get_portfolio_history(date_start =
         ↪  day,timeframe='5Min').df.iloc[:78])
    equities = df.equity.values
    cumu_returns = equities/equities[0]
    cumu_returns = cumu_returns[~np.isnan(cumu_returns)]

    return df, cumu_returns

def DIA_history(start):
    data_df = yf.download(['^DJI'],start=start, interval="5m")
    data_df = data_df.iloc[:]
    baseline_returns = data_df['Adj Close'].values/data_df['Adj
    ↪  Close'].values[0]
    return data_df, baseline_returns

# Get cumulative return
API_KEY = " " # insert your api key
API_SECRET = " " # insert your secret api
API_BASE_URL = 'https://paper-api.alpaca.markets'
data_url = 'wss://data.alpaca.markets'

df_erl, cumu_erl = alpaca_history(key=API_KEY,
                                  secret=API_SECRET,
                                  url=API_BASE_URL,
                                  start='2023-04-14',
                                  end='2023-05-03')

df_djia, cumu_djia = DIA_history(start='2023-04-14')

df_erl.tail()

model_returns = cumu_erl -1
dia_returns = cumu_djia - 1
dia_returns = dia_returns[:model_returns.shape[0]]

# Generate a range of dates between start and end date
```

```
date_range = pd.date_range(start='2023-04-14', end='2023-05-03',
↪  freq='B')

# Create a new DataFrame with the date_range as the index and the
↪  returns_dia as a column
returns_dia_df = pd.DataFrame({'returns_dia':
↪  returns_dia[:len(date_range)]}, index=date_range)

# Plotting code
plt.figure(dpi=100)
plt.grid()
plt.grid(which='minor', axis='y')
plt.title('Stock Trading (Paper trading)', fontsize=20)
plt.plot(returns_dia_df.index, returns_dia_df['returns_dia'],
↪  label='Alpaca Trading History', color='blue')
plt.ylabel('Return', fontsize=16)
plt.xlabel('Trading Days', fontsize=16)
plt.xticks(size=14, rotation=45)
plt.yticks(size=14)

# Configure x-axis ticks
ax = plt.gca()
ax.set_xticks(returns_dia_df.index)
ax.set_xticklabels(returns_dia_df.index.strftime('%Y-%m-%d'),
↪  rotation=45)

# Configure y-axis ticks
ax.yaxis.set_minor_locator(ticker.MultipleLocator(0.005))
ax.yaxis.set_major_formatter(ticker.PercentFormatter(xmax=1,
↪  decimals=2))

# Display legend and show plot
plt.legend(fontsize=10.5)

# Save the plot as an image file
plt.savefig('trading_history_plot.png', bbox_inches='tight')

# Show the plot
plt.show()
```

7 Conclusion

In this tutorial, we've explored how to use artificial intelligence, specifically reinforcement learning, to build an autonomous trading system on the Alpaca

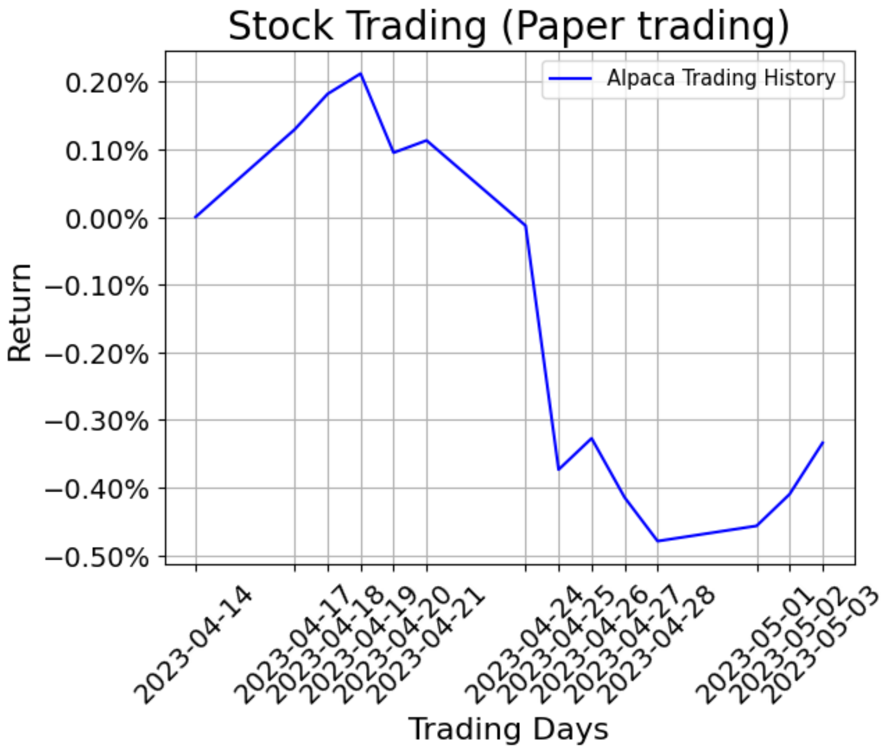

Figure 1: Alpaca Trading Activity, April 14-May 03, 2023

platform. Using the FinRL module and Optuna, we've learned how to train an RL trading agent, tune its hyperparameters, and integrate it with Alpaca to execute trades in real-time.

By following the steps outlined in this tutorial and customizing the code to fit your specific trading strategy, you can build an RL trading agent that can learn from its interactions with the market and make profitable trading decisions while minimizing risk.

As with any trading strategy, there are always risks involved, and it's important to exercise caution and diligence when executing trades. However, by using the tools and techniques explored in this tutorial, you can gain a deeper understanding of the stock market and develop a trading strategy that maximizes your profits.

I hope this tutorial has been informative and helpful in getting you started with autonomous quantitative trading. As always, there are opportunities for improvement and expansion, such as exploring more complex RL algorithms or incorporating additional data sources. I encourage you to continue experimenting and refining your own trading strategies using the techniques and tools we've explored here.