Contents

[Background and scope 1](#_dzgmq03cl4ui)

[Project body 1](#_stfpb7tn3fr3)

[Main.py 1](#_1v7ccoeakm3t)

[data\_fetcher.py 1](#_8gyf0tg8srr)

[Monitor\_performance.py 2](#_62vn23k243ys)

[utils.py 2](#_pr25satyddws)

[trading.py 3](#_c98yr1gmqqwu)

[New Approach 4](#_8ay4zliv3x96)

[Define the indicators 4](#_yq0rb82egb6s)

[Define the work 5](#_j447fimkczu2)

[Execute 5](#_3xi80cwvyae6)

[trading.py (under the new approach) 5](#_90rqq3ek9bjc)

# Background and scope

Following is the memorandum of my Python project. The aim is to document the project in python and how it is structured.

The aim is to backtest a simple trading strategy, and once is ready, to start to trade the strategy.

# Project body

This section is to define the structure of the project so far, and gives details for the functions and classes within every tab.

### Main.py

This is the main body of the project. This is the page that it will be run at the end basically. All of the rest gravitate around main.

### data\_fetcher.py

The DataFetcher class is designed to retrieve historical candlestick data from the Binance API. It allows users to fetch market data for a specified trading pair (e.g., BTCUSDT) over a defined time interval.

* Converts human-readable time intervals into Binance API-compatible constants.
* Fetches historical market data and processes it into a structured pandas DataFrame.
* Provides easy-to-use methods for data retrieval and processing.
* Create technical indicators and add them as a column

Class methods

\_\_init\_\_(client, symbol)

These are the parameters needed to initialize the data fetcher

* Client: is your API, and is made up of your api Key and your secret. This is the basic API that you can obtain from binance API manager, and allows you to retrieve historical data
* Symbol: this is the currency symbol as shown in binance (e.g. “BTCUSDT”)

Class Functions

* fetch\_data
* fetch\_VWAP
* fetch\_VWAP\_session
* VWAP
* ATR

### Monitor\_performance.py

This is a file containing functions to evaluate the performance of a given function or .py file

This will be particularly useful when dealing with trading functions

Functions

* evaluate\_performance(target, \*args, mode='function'): this function gives you the performance (second runtime) of a given function or a .py file

At the moment this functions seems to be not working when evaluating something that download data

### utils.py

Series of functions that could be usefult within the project

* plot\_VWAP
* get\_interval\_from\_string()

This function is created ad hoc to transform the interval stored in kline to human readable intervals, so that one can use strings as “1MIN” in the functions.

* plot\_intro\_strategy()

This is a function for data visualizaiton. It is created for plotting specifically the function intro\_strategy(). The plan is to scale it and implement it for other functions.

* Plot\_cumulative\_pnl()

### trading.py

#### Intro strategy

As a first step, I tried a very simple strategy just to test the implementation in python and set up the class.

The strategy enter when the price cross the VWAP and set the stop loss as a % of the ATR. I am using these two indicators as I want to gain confidence with them.

This function simply create a signal (true or False) for long or short entry:

* long entry : When the previous close was below VWAP and now cross above
* Short entry : when the previous close was above VWAP and now cross below

The functions creates also stop loss and take profit. (both long and short)

* Stop loss: current implementation shows the stop loss as a percentage of the of the ATR (ATR% \* closing price)
* Take profit: it is defined as target\_RR (target RR \* (enter price – stop loss)

#### VWAP trend following

In analysing the strategies against the VWAP, the first step has been a visual inspection, after which I collected a couple of questions that I would have addressed:

* It seems that after a certain time (12 in the morning or 15 in the afternoon) the price is more volatile with more volumes and the VWAP is better respected (if the trend is up, the price is up). Since the market is continuous, it’s important to understand where the VWAP is informative.
* The second question is when to close the session of the VWAP

As I understand from the VWAP, is that it can be useful to understand when to buy. The idea would be that it is better to buy when the price is above and trying to go below. That is what I would like to do. The first thing is 1) to understand the trend and doing trend following and 2) understand the “significance” of the VWAP in the recent days. Therefore, I would like to enter the trade when the VWAP is “respected” for couple of days on a row (for this I mean that the price touch the VWAP and shows that the market is looking at it). Therefore, for this strategy, I would like to

* Understand the trend
* Use the VWAP as a support for entering the trade and placing the stop loss
* Try if waiting 2 or 3 days on a row makes the difference to the strategy
* Try to enter if the VWAP shows inclination (meaning that the volumes are supporting the trend) and if the VWAP is respected
* I could to, when it touch enter at the next one until the distance between closing prices and VWAP is increasing and above a threshold. When they get closer, it means that the price is lowering and should enter the trade, or the other way round, if the difference gets higher exit the trade. . TBD

Note: it seems to be visually not confirmed

#### VWAP and ATR

I’m thinking about someone that want to do trend following in a crypto market. I do not aim to beat banks and institutionals investors, but small traders.

What they look?

In a crypto market, they look for volatility, cause they are risk lover driven by FOMO, and they enters trades driven by emotions. 🡪 I will trade in a low volatility setting, to avoid the randomness of these investors.

Most of them would do trend following, cause it’s easier and clear, and they would do it in a bias uptrend, cause this is what markets do.

Considering the above, I drew the following strategy:

* 1 min time frame
* ATR smoothed on 300 candles (300 minutes), because in one big screen on a minute time frame, approx. 400 candles fit
* When the volatility increases, and the price goes far from the VWAP (far from VWAP in uptrend means price not supported by volume), it means that they were buying, highering volatility, but the price is going too up, either the VWAP start growing again after a retracement, or the FOMO is over.
* So could be
  + Higher 300 ATR (to define what higher means)
  + Price far from the VWAP
  + First retracement (on the way up) could go, the second as well, but if the third one fail enter 🡪 at this point it would be better to consider a double top

Note: it seems to be visually not confirmed

#### New approach reference

A new approach has been followed, therefore the trading.py tab is explained again in the context of the new approach. Please refer to [trading.py](#_3xi80cwvyae6)

# New Approach

As a background, I understood that finding a strategy is pointless and time consuming, as looking at the chart myself always bring bias. So far, I was trying the intro strategy (buy when the price was below the VWAP and now it goes above) and tested in a bullish setting (When Trump election was supporting BTC) and it worked, but in this case whatever buying strategy would have worked.

Therefore, I thought to utilize a machine learning approach and feed the model with historical data to understand the pattern of the strategy.

## Define the indicators

#### VWAP

I selected the VWAP, because is a pretty strong indicator. Is something that the big market players (theoretically) look at. Even though the indicator is pretty

#### ATR

#### RSI

#### SMA and EMA

## Define the work

The work to be done now:

* Select the indicators that I want to use
* Try to understand how I can set up the model of supervised learning to learn the patter
* Collect the relevant data and understand in which data to train the model
* Backtesting on historical data
* Start setting up something data fetch data in real time
* Implement in demo

## Execute

### trading.py (under the new approach)

I decided to use Reinforced learning, where the model learns from a loss function, rather than supervised learning, where one has to label the data with buy/sell/hold signals and train a classifier.

For consistency, the class will be created here. The first step is to create a Trading Environment where the model can learn. The RL settings create an environment in which we have a

* Observation space: where the mode can learn. Is made up of
* Action space: where the model can take actions. Is made up of a discrete set of numbers (0,1,2) that are associate to, respectively, hold, buy and sell

As a consequence, the part in this paragraph should enhance the part already described in [trading.py](#_c98yr1gmqqwu).

#### class TradingENV

This class creates a virtual environment where the model can learn and test the patterns.

This trading environment functioning can be summarized in the following steps:

* At the beginning, an initial state is defined, with a balance, a position, an empty vector of pending orders and so on. At this point, also an observation space is created, with 3 different dimension, one for long, one for short and one for hold positions, which are identified by a code, respectively 1, 2 and 0. These are called “actions”
* With the function \_next\_observation(),the first item is taken from the dataset, therefore being in current\_step = 1 and with the previous position (rendered by the code as Position (t-1) being equal to None, as there are no observations to draw conclusion on the action)
* After that, the step() function is triggered. Firstly, this function retrieve necessary information from the dataset, as high, low, atr as percentage, previous high and previous low (that will be used to place the order), and then set the if else for action: if action is buy (action == 1), then change the position (self.position) to long, set the stop price at the previous low and place a buy order with the function \_palce\_order().
* As a second point, the step() function execute the pending orders with the function \_execute\_pending\_orders() (orders that were previously placed with \_palce\_order() ) and then add a +1 to the current step, in order for the environment to go on in the analysis of the dataset.
* The function \_execute\_pending\_orders() previously introduced go through the vector of pending orders, and check if the take profit or the stop loss level are triggered (by comparing them with the maximum and minimum at the current step). If so, the order is executed and appended to the vector of executed orders.
* At the very end, the function render() simply print the current state of the environment, showing the current step, the balance, the position and the pending orders.

The trading environment works with the following main assumptions:

* Take profit: is set as a Risk Reward target (to be finished once is done)
* Stop loss: the stop loss is currently defined as the Entry price - ATR%\*Entry price. Therefore, at the current state the stop loss is highly dependent on the ATR, that is computed on a 14-period rolling window.

For an overview of how to use the class, refer below to the subsection “Usage”.

At the current state it includes the following methods and functions.

Class methods

\_\_init\_\_(df)

These are the parameters needed to initialize the data fetcher

* df: this datafarme should be created with Datafetcher (please ref to [data\_fetcher.py](#_8gyf0tg8srr)), as for consistency and to work well, we need to have the specific column names that we have defined within the datafetcher

Class Functions

* reset(): simple function to reset the state of the environment to the basic state (in which we are at step 0 in the dataframe, the capital is the same […] and in which at the end we have the next observation (through the function \_next\_observation() )
* \_next\_observation(): this function retrieves the current market state. It takes the values of the dataframe at the current step (return self.df.iloc[self.current\_step].values)
* Step(action): this function execute an action (buy, sell or old) and give back the current environment.
* \_place\_order(): this function place the limit order to be subsequently traded
* *execute*\_pending\_orders(self, high, low): this function execute the pending orders
* render(): this function gives back the current status of the market

#### Reward

As per Open AI gym Environment (class Env) the reward function should be created inside the step() function (where the agent perform a given step). In my case, the step function creates a limit order, which is executed (if applicable) by another function, named \_execute\_pending\_order(). Therefore, I updated the reward function there and checked that it is updated in real time.

At the beginning, my reward function was computed as the percentage profit (or loss) that the agent obtained with a trade, and being 0 in case of a hold position. By keeping the value of 0 in case of a static position, the agent was not able to learn enough, leading to tremendous results on a dataset of 71’422 observations with 4 trading indicators (VWAP, ATR, RSI and SMA).

The results obtained from this approach are showed in Figure 1 below.

A graph of a diagram

AI-generated content may be incorrect.

Figure : Results deriving from the first reward function

Similar results where shown with a simple 1 and -1 reward function.

Therefore, I created a new reward function that works on a cumulative basis. Therefore, the function do not change if the agent decide an action to hold the position.

The function showed a slight improvement (as shown in below)

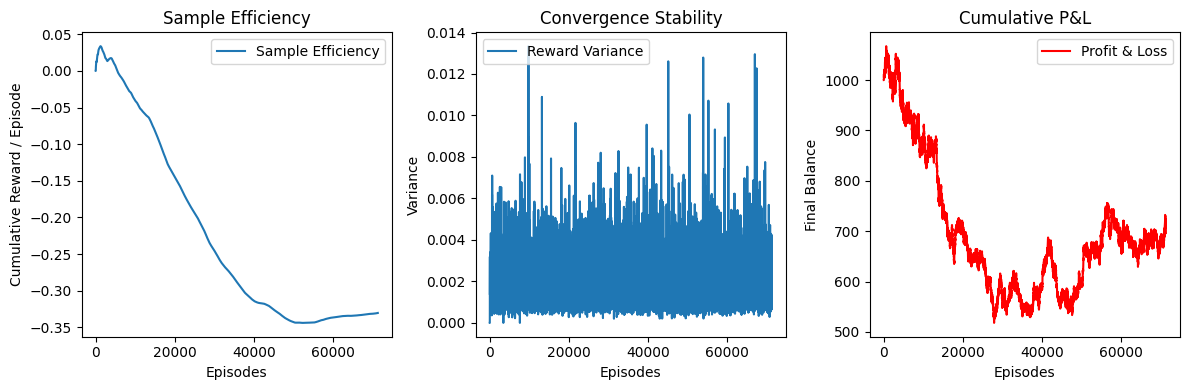


Figure : Updated reward function (cumulative percentage P&L)

Still, the poor performance convinced me to try to multiply the percentage increase for the balance, to understand how the agent would react to this.

Also this approach seems to not work.

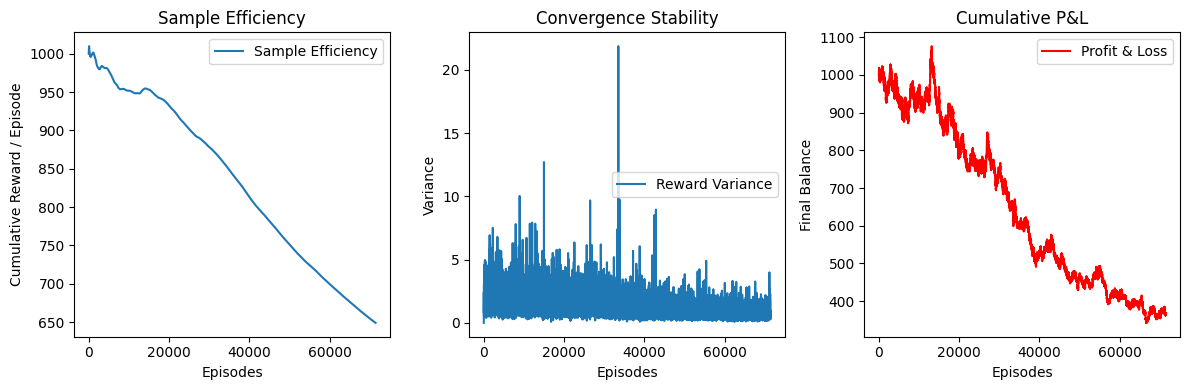


Figure : Performance of the agent with the reward function which is equal to the balance.

Usage

#Create the environment N1

env\_v1 = TradinEnv\_v1(df\_train)

test\_env\_v1 = TradinEnv\_v1(df\_test)

# Train the model

model\_v1 = PPO("MlpPolicy", env\_v1, verbose = 1) # PPO model to train the AI

model\_v1.save("trading\_ppo\_model")

model\_v1 = PPO.load("trading\_ppo\_model")

# Testing on the second (TradinEnv\_v1)

obs\_v1 = test\_env\_v1.reset()

done = False

while not done:

action, \_ = model\_v1.predict(obs\_v1)

test\_env\_v1.render()

obs\_v1, reward, done, \_ = test\_env\_v1.step(action)

#### class PerformanceEvaluation

##### Introduction

This document provides a detailed description of the PerformanceEvaluation class, which evaluates the performance of a Reinforcement Learning (RL) agent in a trading environment. The evaluation focuses on three key metrics:

1. **Sample Efficiency** – Measures how quickly the agent learns to obtain positive rewards.
2. **Convergence Stability** – Analyzes the consistency of the agent’s performance over time.
3. **Cumulative Profit & Loss (P&L)** – Tracks the financial outcome of the agent’s trading decisions.

The class includes both numerical outputs and graphical representations to help visualize performance trends over multiple episodes.

##### Class Overview: PerformanceEvaluation

**Class Definition**

class PerformanceEvaluation:

The PerformanceEvaluation class is responsible for evaluating an RL agent's performance. It tracks the agent's rewards and financial balance across multiple episodes to compute the key metrics.

**Constructor (\_\_init\_\_ Method)**

def \_\_init\_\_(self, env, model, num\_episodes=100):

This method initializes the class with the following parameters:

* **env (object)**: The trading environment in which the agent operates.
* **model (object)**: The trained RL model.
* **num\_episodes (int, default=100)**: The number of episodes over which the agent's performance will be evaluated.

The method also initializes two lists:

* reward\_history: Stores the total rewards obtained in each episode.
* balance\_history: Stores the final balance after each episode.

##### Methods in PerformanceEvaluation

**1. Running the Evaluation (run\_evaluation Method)**

def run\_evaluation(self, show\_plot=False, show\_numerics=True):

**Description:**

This method runs multiple episodes to evaluate the RL agent’s performance based on reward accumulation and financial results.

**Parameters:**

* **show\_plot (bool, default=False)**: If True, it generates visual plots for performance metrics.
* **show\_numerics (bool, default=True)**: If True, it displays numerical values for performance metrics.
* **first\_run (bool, default = True):** If true, it means that the model has not been runned before, so there is no storage of teh varaibles called obs and reward and so on. The model will be run a second time if the parameters is False, meaning that theere is the need to run the model before evaluating the performance

**Methodology:**

* Resets the environment before each episode.
* Runs the agent using its policy (model.predict), collecting rewards and updating the environment state.
* Stores the total reward and final balance for each episode.
* Calls either display\_metrics() (to print numeric values) or plot\_metrics() (to generate performance plots), depending on the user's input.

**2. Displaying Numerical Metrics (display\_metrics Method)**

def display\_metrics(self):

**Description:**

This method prints out the numerical values for the three performance metrics:

* **Sample Efficiency**: Computed as the **mean of total rewards** over all episodes.
* **Convergence Stability**: Computed as the **standard deviation of rewards** over all episodes.
* **Cumulative P&L**: The final balance after all episodes.

**Example Output:**

Performance Metrics:

Sample Efficiency: 15.67

Convergence Stability: 4.23

Cumulative P&L: 1205.50

**3. Generating Performance Plots (plot\_metrics Method)**

def plot\_metrics(self):

**Description:**

This method generates three separate plots to visualize the performance trends of the RL agent over time.

**Plots Generated:**

1. **Sample Efficiency (Reward Accumulation over Episodes)**
   * Measures whether the agent is improving over time by plotting cumulative reward per episode.
2. **Convergence Stability (Reward Variance over Episodes)**
   * Uses a rolling window to compute and plot the standard deviation of rewards.
3. **Cumulative Profit & Loss**
   * Tracks the agent’s balance at the end of each episode.

**Plot Customizations:**

* Each subplot has a title and labeled axes.
* A moving average is used for smoothing fluctuations in reward variance.
* plt.tight\_layout() ensures proper spacing between plots.

##### Conclusion

The PerformanceEvaluation class provides a **comprehensive assessment** of an RL trading agent’s performance. It allows users to:

✔️ Evaluate **sample efficiency** (learning speed).  
✔️ Assess **convergence stability** (how consistent rewards are over time).  
✔️ Track **cumulative profit and loss** (financial performance).

With **both numerical and visual outputs**, the class provides deep insights into whether an RL agent is effectively learning and making profitable decisions.

**Future improvements could include:**

* Adding a **learning rate calculation** to quantify improvement over time.
* Measuring **risk-adjusted returns** for more robust financial evaluation.

### Reinforcement learning

Now, the code create a model with the following line of code

model = PPO("MlpPolicy", env, verbose = 1)

in which PPO is a reinforcement learning algorithm developed by openAI, that allows to make decisions in environment like video games and financial modelling. PPO directly optimize the policy rather than estimating a value function. It is an Action-Critic exercise, and decide which action to take and evaluates how good is the result with a value function.

“MlpPolicy” uses a multi Layer Perceptron (MLP) as a neural network policy. It is a type of neural network with multiple layers of neurons. Each neuron is connected to every neuron in the next layer (fully connected) and it is used to process numerical data.

After that, the model is saved and then loaded

model.save("trading\_ppo\_model")

model = PPO.load("trading\_ppo\_model")

obs = env.reset() # reset function put the state back to the initial state and return the first step of the dataset

done = False

and the initial state is settled.

while not done:  # Keep running until done is True

    action, \_ = model.predict(obs) # Prediction using the trained model. Action is buy (1) or sell (2)

    obs, reward, done, \_ = env.step(action) # Apply the action in the environment

    env.render() # Print the state for debugging

with

action, \_ = model.predict(obs)

The observation is passed though the MLP of the PPOm that process the input and outputs in action. The action in this case is a descrete action that can be 0 = Hold, 1 = buy or 2 = sell.

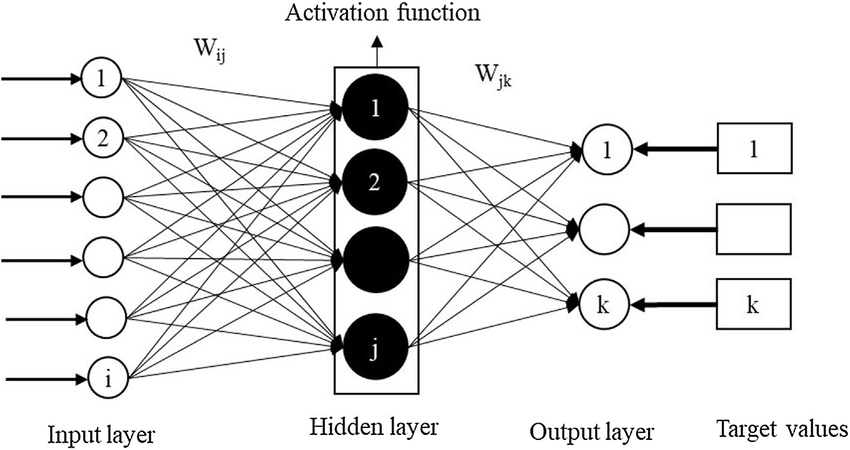
In our case, obs is a NumPy array with numerical values containing market indicators, opening, closing price, RSI and so on. The values (of which is not important the lable) are passed in the Neural Network.

At this point the PPO model use the “MlpPolicy” NN that contains:

* An input layer (that takes the obs)
* One or more **hidden layers** with neurons using activation functions (e.g., ReLU)
* An **output layer** predicting the best action.

The values of obs are first normalized (scaled between -1 and 1), and the feature are extracted from the provided data. Therefore, at the beginning the actions are random, but they then improve with the training.

PPO uses a stoacastich policy, meaning it samples actions based on probabilities, and then compare the action taken with the reward assigned. The reward and penalties are defined in the function step. The setting of the Reinforcement learning therefore is pretty simple, consisting of us giving some values to the model and asking him to learn based on the rules we gave to the system (the reward is decided by us in the function step() ). To adapt to the randomness of the financial markest, the model still keep some randomness.



*Figure 2*

# Appendix I - Working in sync in Git

### **1. Set Up Git on Both Computers**

Make sure Git is installed on both machines. If not, install it from [git-scm.com](https://git-scm.com/).

Then, set up your Git username and email:

git config --global user.name "Your Name"

git config --global user.email "your.email@example.com"

### **2. Clone the Repository on Both Computers**

On each computer, clone the repository from GitHub:

git clone https://github.com/your-username/your-repo.git

Replace your-username and your-repo with your actual GitHub details.

### **3. Workflow for Working on Both Computers**

#### **On Computer A**

1. **Pull the latest changes from GitHub** (in case there are any updates):  
   git pull origin main
2. **Make changes in VS Code** (edit files, create new ones, etc.).
3. **Add, Commit, and Push changes**:  
   git add .

git commit -m "Your commit message"

git push origin main

#### **On Computer B**

**Before starting work, always pull the latest changes**

git pull origin main

1. **Make changes, then commit and push** (same as above).

## Code Summary

### Push

Every time I change something:

git add .

git commit -m "Your commit message"

git push origin main

### Pull

Every time I want to work on the updated version

git pull origin main