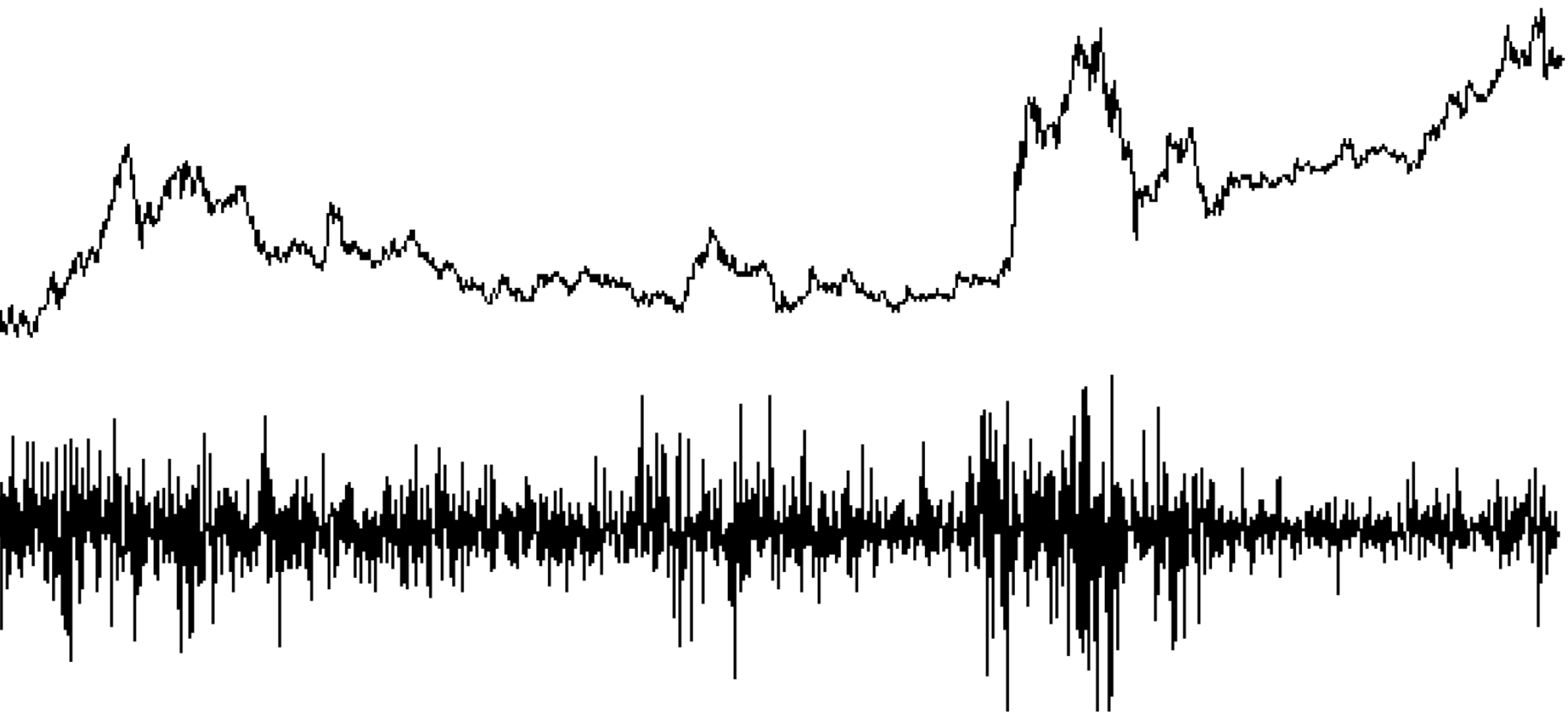
MASTER'S DOUBLE-DEGREE IN COMPUTATIONAL SCIENCE

SK6093 Hands-on Series
Independent Research in Computational Science 3
Institut Teknologi Bandung

Deep Reinforcement Learning-Based Stock Portfolio Optimization with Deep Q-Network Algorithm in Python:
A Comprehensive Tutorial



By:

Muhammad Abraar Abhirama
20923003

THIRD SEMESTER — ACADEMIC YEAR 2023/2024

TABLE OF CONTENTS

Chapter 1
How to Contribute to My GitHub Repository

In this Hands-on short module, we will deep dive into the following GitHub repository of mine: https://github.com/abraar4100/sk6093.git. This repository contains two main things, which are *DRL_Environment* and *Main Program*. As we an guess based on what we have learned in the preceding chapter, the *DRL_Environment* folder contains the required reinforcement learning environment (.py), and the *Main Program* folder contains the main notebook (.ipynb) where we deal with the analysis and data visualization.

1.1. Contributing to My Repository
1.1.1. First Step: Fork my Repository
Please fork the following repository:

https://github.com/abraar4100/sk6093.git

Assuming that the reader is not familiar GitHub, I will explain the main three terms when collaborating on a project in GitHub, namely: **forking, branching, and cloning.**

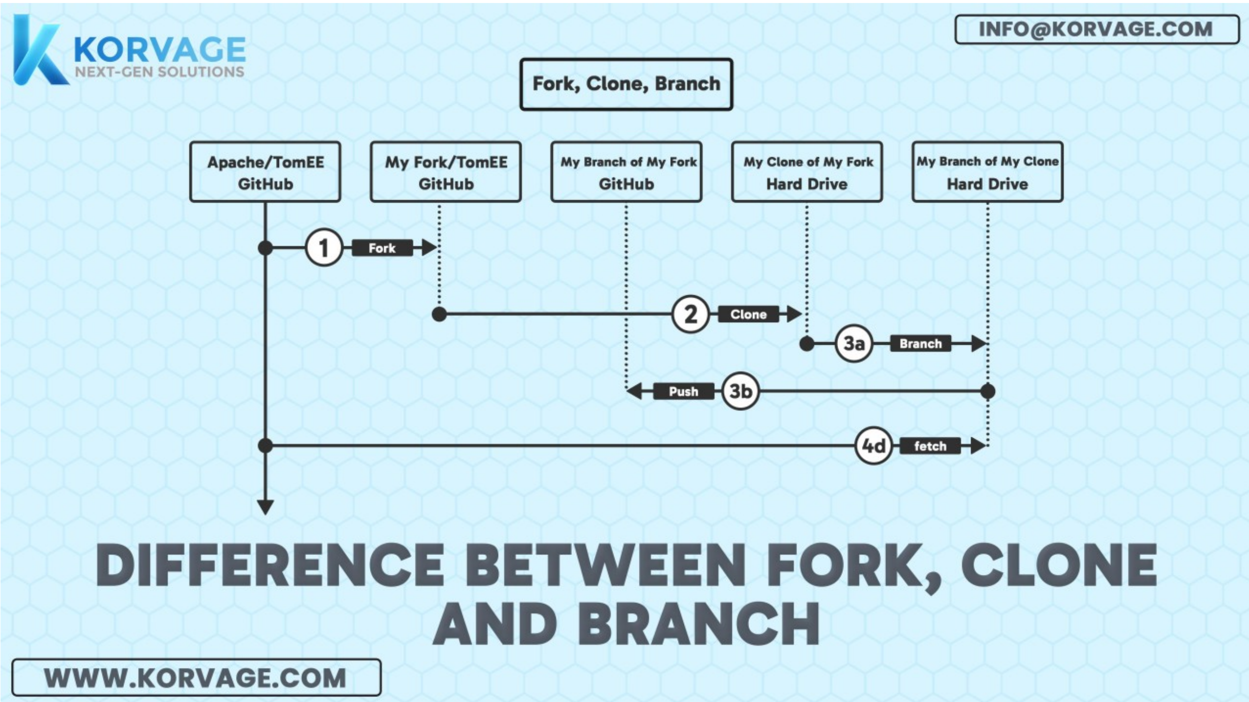


Figure 1. Difference between fork, clone, and branch

Forking: A fork is essentially a duplicate of a repository. When you fork a repository, you generate an independent copy of the entire project, including all its files, commit history, and branches. This separate copy allows you to make changes without impacting the original repository. Forking is commonly used in open-source development, enabling contributors to create their own versions of a project to experiment with modifications or suggest enhancements. A forking is often done by someone who is interested to experiment one's repository but not directly involved in the making of that repositories.

Branching: A branch is a parallel version of a repository's codebase. Creating a branch initiates a new line of development that splits from the main codebase. Branches enable multiple developers to work on distinct features or fixes concurrently without disrupting each other's progress. After the changes on a branch are finalized and tested, they can be merged back into the main codebase.

Cloning: A clone involves copying a repository onto your local machine. Unlike forking, which generates a separate copy on a remote server, cloning downloads the entire repository to your computer. This enables you to work on the code locally, make changes, and contribute to the project without needing a constant internet connection. Cloning is often used when you want to collaborate on a project or work on it offline (you don't work on GitHub).

Therefore, you can do forking to contribute to my GitHub repository since the project has been completed by me. Now it is your turn to make some improvements!

1.1.2. Second Step: Make changes, commit them, and push the commits

After you have made some improvements, then commit (finalize, but not really finalizing) it. After commiting it, you have to push it to the forked repository. Note that commiting and pushing are different step. Why? Because there are other team members (unless the only member is only you). Pushing changes after committing is necessary because **committing only saves your changes locally on your machine**, within your local repository. To share your changes with others and integrate them into the remote repository (forked repository), you need to push your commits.

1.1.3. Third Step: Create a New Issue (Optional)

Let's imagine if you find any bug after pushing your commits and you want other team members be notified on it. How do you do it? Simply by creating a new issue. In this GitHub feature, you can specify the problems you find. You can also create a new issue after doing some pull request (explained on the next step) as well.

1.1.4. Fourth Step: Pull Request

Please let me know after you have done forking the repository and making some improvement. How do I notice the changes you have made? Simply by using *Pull Request*. A GitHub pull request is a feature that allows developers to notify team members **(in this case you are my team member)** that they have completed a piece of work and request that it be reviewed and potentially merged into the **main repopsitory (not forked repository).** When you open a pull request, you are proposing your changes and requesting that someone else reviews and pulls in your contribution. This process involves comparing changes across branches, discussing modifications, and refining the code collaboratively. Pull requests are commonly used in collaborative development environments to ensure code quality and manage changes systematically.

1.2. Exploring the Repository

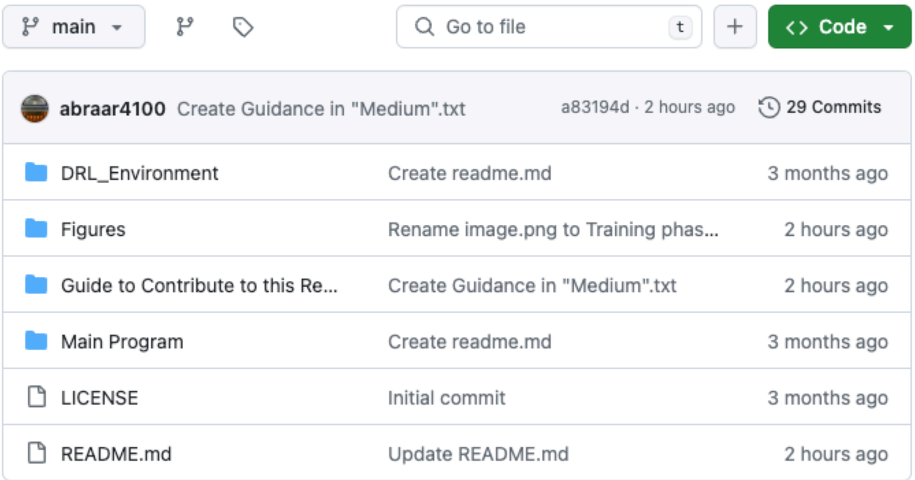


Figure 2. My repository

Figure 4.2 shows my main repository which mainly contains 4 (four) folders: DRL_Environment, Figures, Guide to Contribute to this Repository, and Main Program. The explanation of this repository is explained on the README.md (see the bottom side of the main repository). Basically, you can read about what every folder contains by looking at the README.md file on the folder you are in. Each folder is explained as follows:

1.2.1 DRL_Environment Folder

This folder contains the environment of the proposed Deep Reinforcement Learning method using Deep Q-Network Algorithm:

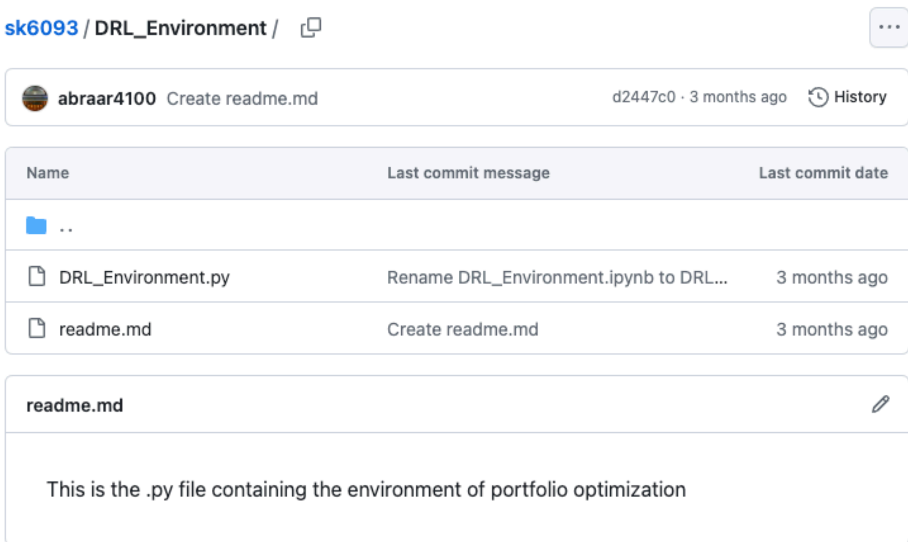


Figure 2. My repository

The DRL_Environment.py file is as follows:

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import gym

class PortfolioEnv(gym.Env):
    """
    Menggunakan environment gym untuk action space yang diskrit
    """
    def __init__(
        self,
        df,
        return_cols,
        feature_cols=[],
        window_size = 20,
        order_size = 0.1,
        starting_balance = 1,
        episode_length = 180,
        drawdown_penalty_weight = 1,
        allocations_in_obs = False
    ):
        """
        Parameters:
            - `df`: Pandas dataframe with datetime index
            - `return_cols`: list nama kolom yang berisi asset returns (first
entrynya adalah risk free returns)
            - `feature_cols`: List of column names to be used as features
            - `episode_length`: Length of each episode (-1 makes it go from
start to end)
            - `window_size`: Size of lookback window
            - `order_size`: Size of step in allocations
            - `starting_balance`: Amount of cash to start with
            - `episode_length`: Length of each episode
            - `drawdown_penalty_size`: Weight of drawdown on reward
            - `allocations_in_obs`: Whether or not to include current
allocations in the observation
        """

        # Data related constants
        self.RETURN_COLS = return_cols
        self.FEATURE_COLS = feature_cols
        self.NUM_ASSETS = len(return_cols)-1
        self.NUM_FEATURES = len(feature_cols)
        self.RETURNS = df[self.RETURN_COLS].to_numpy()
        self.FEATURES = df[self.FEATURE_COLS].to_numpy()
        self.INDEX = df.index
```

```
        # Environment constants
        self.WINDOW_SIZE = window_size
        self.ORDER_SIZE = order_size
        self.ALLOCATIONS_PRECISION = len(str(self.ORDER_SIZE).split('.')[-1]) #
number of decimal places of order_size
        self.STARTING_BALANCE = starting_balance
        self.EPISODE_LENGTH = episode_length
        self.DRAWDOWN_PENALTY_WEIGHT = drawdown_penalty_weight
        self.ALLOCATION_IN_OBS = allocations_in_obs

        # Initialize action/observation space
        self.action_space = gym.spaces.Discrete(self.NUM_ASSETS*2 + 1) #
buy/sell for each stock or do nothing
        if self.ALLOCATION_IN_OBS:
            self.observation_space = gym.spaces.Box(
                low = np.concatenate([self.FEATURES.min(axis=0) for _ in
range(self.WINDOW_SIZE)] + [np.zeros(self.NUM_ASSETS+1)]),
                high = np.concatenate([self.FEATURES.max(axis=0) for _ in
range(self.WINDOW_SIZE)] + [np.ones(self.NUM_ASSETS+1)]),
                shape = (self.WINDOW_SIZE*self.NUM_FEATURES +
self.NUM_ASSETS+1,),
                dtype = np.float64
            )
        else:
            self.observation_space = gym.spaces.Box(
                low = np.concatenate([self.FEATURES.min(axis=0) for _ in
range(self.WINDOW_SIZE)]),
                high = np.concatenate([self.FEATURES.max(axis=0) for _ in
range(self.WINDOW_SIZE)]),
                shape = (self.WINDOW_SIZE*self.NUM_FEATURES,),
                dtype = np.float64
            )

       # mereset environment
        self.reset()

    def reset(self):
        """
Me-reset environment pada index yang random dipilih
        """
        if self.EPISODE_LENGTH == -1:
            self.start_index = self.WINDOW_SIZE
        else:
            self.start_index = np.random.randint(self.WINDOW_SIZE,
len(self.RETURNS)-self.EPISODE_LENGTH) # Random start index
        self.current_index = self.start_index
```

```
        # The allocations always adds up to 1 with starting allocations as [1,
0, 0, ..., 0] (index 0 is for cash).
        self.current_allocations = np.insert(np.zeros(self.NUM_ASSETS), 0, 1.0)
        self.current_value = self.STARTING_BALANCE
        self.weighted_cumulative_return = 0

        self.return_history = [0]
        self.value_history = [self.current_value]
        self.allocations_history = [self.current_allocations.copy()]

        return self.get_observation()

    def get_observation(self):
        """
Mereturn history of return dan fitur lainnya sejumlah WINDOW_SIZE (hari).
Tidak termasuk returns dan features pada indeks saat ini.
        """
        obs = self.FEATURES[self.current_index-self.WINDOW_SIZE :
self.current_index].flatten()
        if self.ALLOCATION_IN_OBS:
            obs = np.concatenate((obs, self.current_allocations))
        return obs

    def update_current_allocations(self, action):
        """
mengupdate current_allocations sesuai dengan action yang dilakukan.
action bisa berupa hold, buy, at sell saham
Sebuah action dapat mengubah hingga satu alokasi dengan order_size.
Jika sebuah action tidak valid maka dianggap sama dengan melakukan hold.
        """
        action -= self.NUM_ASSETS # Convert the action to a number between -
len(ASSETS) and +len(ASSETS)
        action_asset, action_sign = abs(action), np.sign(action)

        # If we want to do nothing
        if action_sign==0:
            return # exit the function

        # If we want to buy and have cash (e.g action +3 means we want to buy
the asset at position 3).
        elif (action_sign>0) and (self.current_allocations[0]>0):
            self.current_allocations[action_asset] += self.ORDER_SIZE
            self.current_allocations[0] -= self.ORDER_SIZE

        # If we want to sell and have the asset (e.g -1 means we want to sell
asset at position 1).
        elif (action_sign<0) and (self.current_allocations[action_asset]>0):
            self.current_allocations[action_asset] -= self.ORDER_SIZE
```

```
            self.current_allocations[0] += self.ORDER_SIZE

        # Round to avoid floating point error
        self.current_allocations =
self.current_allocations.round(decimals=self.ALLOCATIONS_PRECISION)


    def update_current_value(self):
        """
Memperbarui `current_value` sesuai dengan `current_allocations` dan pengembalian
yang masuk pada indeks saat ini.
mereturn nilai sebelumnya untuk perhitungan pengembalian.
        """
        previous_value = self.current_value
        self.current_value *=
((1+self.RETURNS[self.current_index])*self.current_allocations).sum()
        return previous_value

    def step(self, action):
        self.current_index += 1

        if self.EPISODE_LENGTH == -1:
            done = bool(self.current_index >= len(self.RETURNS)-1)
        else:
            done = bool(self.current_index - self.start_index >=
self.EPISODE_LENGTH)

        self.update_current_allocations(action)
        previous_value = self.update_current_value()
        ret = (self.current_value - previous_value) / previous_value

        if ret > 0:
            self.weighted_cumulative_return = (1 +
self.weighted_cumulative_return) * (1 + ret) - 1
        else:
            self.weighted_cumulative_return = (1 +
self.weighted_cumulative_return) * (1 + self.DRAWDOWN_PENALTY_WEIGHT * ret) - 1

        reward = self.weighted_cumulative_return * (self.current_index -
self.start_index)/self.EPISODE_LENGTH
        observation = self.get_observation()

        self.return_history.append(ret)
        self.value_history.append(self.current_value)
        self.allocations_history.append(self.current_allocations)

        return observation, reward, done, {}
    def render(self, ax=None, title='', legend=False):
        """
```

```
Menampilkan perubahan nilai portofolio seiring waktu dalam bentuk stackplot.
        """
        value_history_array = np.array(self.value_history).reshape(-1, 1)
        allocations_history_array = np.array(self.allocations_history)
        value_breakdown = (value_history_array *
allocations_history_array).transpose()

        if ax==None:
            plt.figure(figsize=(8,6))
            ax = plt.axes()

        ax.set_title(title)
        ax.stackplot(
            self.INDEX[self.start_index : self.current_index+1],
            value_breakdown,
            labels = self.RETURN_COLS,
        );

        plt.gcf().autofmt_xdate();

    def get_portfolio_returns(self):
        """
Menghasilkan representasi nilai portofolio yang berubah seiring waktu dalam
bentuk stackplot.
        """
        return pd.Series(
            self.return_history,
            index=self.INDEX[self.start_index : self.current_index+1])
   """
    def plot_allocations(self, ax=None, title='Portfolio Allocations',
legend=True):
        if ax is None:
            plt.figure(figsize=(8, 6))
            ax = plt.axes()

        ax.set_title(title)
        for i, ticker in enumerate(self.TICKERS):
            ax.plot(
                self.INDEX[self.start_index : self.current_index + 1],
                self.allocations_history_array[:, i],  # Assuming the first
column represents cash
                label=ticker
            )
        if legend:
            ax.legend()
        plt.gcf().autofmt_xdate()
         """
```

1.2.2. Main Program Folder

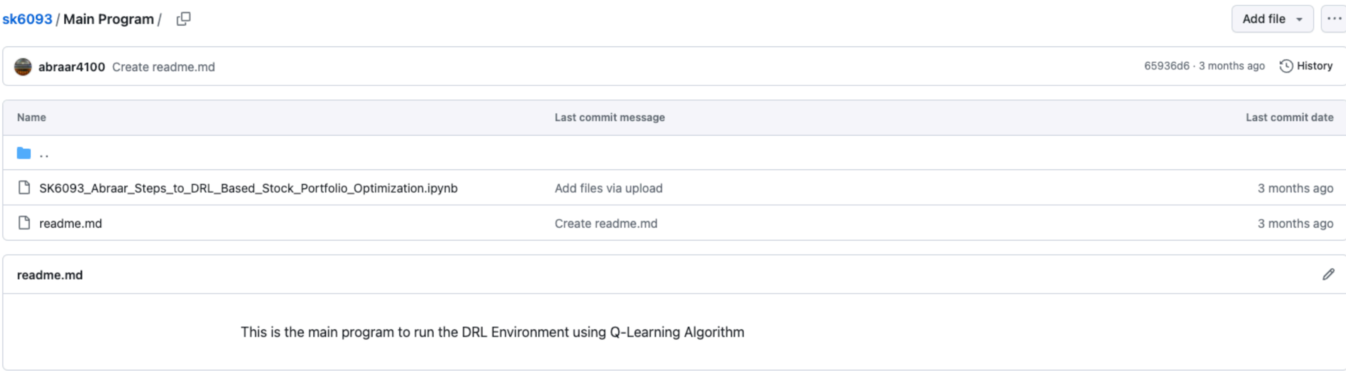
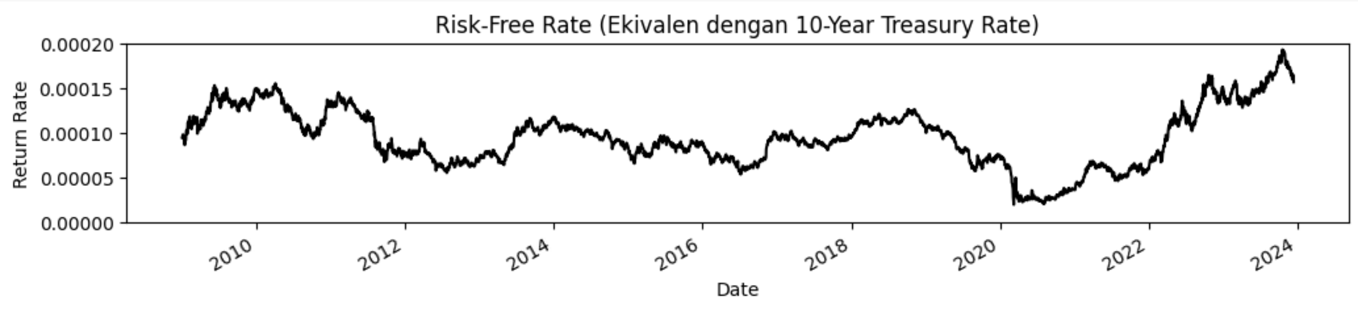


Figure 3. Main Program Folder

The main program file is as follows:

```
#REQUIREMENTS
pip install pyportfolioopt
pip install stable-baselines3
import os
from tqdm.notebook import tqdm
import yfinance as yf

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from pypfopt.expected_returns import mean_historical_return
from pypfopt.risk_models import risk_matrix
from pypfopt.efficient_frontier import EfficientFrontier

from portfolio_environment import PortfolioEnv
from stable_baselines3 import DQN

#DATA PREPROCESSING
## DATE RANGE AND STOCK TICKERS
# 10 arbitrarily selected stocks from the Dow Jones
#TICKERS = ['AXP', 'AAPL', 'BA', 'GS', 'INTC', 'JNJ', 'KO', 'NKE', 'PG', 'DIS']
TICKERS = ['MSFT', 'AAPL', 'V', 'UNH', 'JPM', 'JNJ', 'WMT', 'CVX', 'PG', 'HD']

# 9 years of pre COVID-19 data
TRAIN_START = '2009-01-01'
TRAIN_END = '2019-12-31'

# 1 year of heavily COVID-19 affected data and 1 year of post COVID-19 growth
VAL_START = '2020-01-01'
VAL_END = '2021-12-31'
```

```
# 1 year of recession period
TEST_START = '2022-01-01'
TEST_END = '2023-12-14'

## DATE RANGE AND STOCK TICKERS
yearly_risk_free_rate_percentage = yf.download('^TNX', start = TRAIN_START, end
= TEST_END, interval = '1d')['Close']
risk_free_rate = (1 + yearly_risk_free_rate_percentage / 100) ** (1 / 252) - 1 #
daily risk free rate
risk_free_rate.plot(title='Risk-Free Rate (Ekivalen dengan 10-Year Treasury
Rate)', figsize=(12,2), legend=False, ylim=(0,2e-4), color = "black");
plt.xlabel('Date')
plt.ylabel('Return Rate')
plt.show()
```




```
## STOCK RETURNS
data = {}
for ticker in tqdm(TICKERS):
    data[ticker] = yf.download(
        ticker,
        start = TRAIN_START,
        end = TEST_END,
        interval = '1d',
        progress = False
    )

data = {}
for ticker in tqdm(TICKERS):
  data[ticker]=yf.download(ticker, start = TRAIN_START, end = TEST_END, interval
= '1d', progress = False)
```

```
100%  10/10 [00:01<00:00, 6.80it/s]
100%  10/10 [00:02<00:00, 4.10it/s]
```

```
df = pd.DataFrame(index=pd.date_range(start=TRAIN_START, end=TEST_END,
freq='d')) # create a dataframe with a full index
```

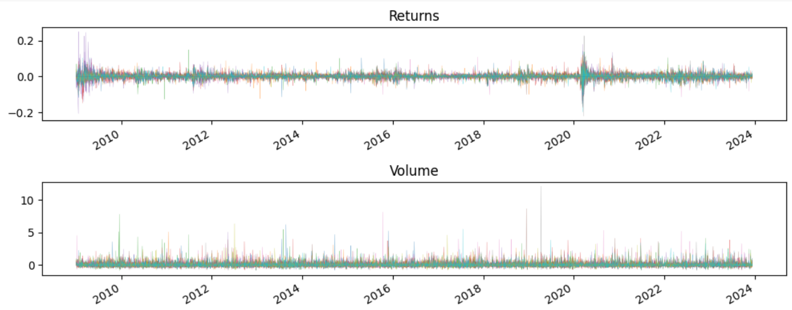
```
df['RISK_FREE'] = risk_free_rate
for ticker in TICKERS:
    df[ticker] = data[ticker]['Adj Close'].pct_change(1) # fill in each return
column
    df[f'{ticker}_VOLUME'] = data[ticker]['Volume'].pct_change(1)

print(f'Number of all NaN rows dropped: {df.isna().all(axis=1).sum()}')
df.dropna(axis=0, how='all', inplace=True) # drop rows with all NaN e.g first
row, weekends, public holidays
```

Number of all NaN rows dropped: 1698

```
# Fill remaining `NaN values`
print(df.isna().sum())
df.fillna(value=0, inplace=True) # replace any remaining NaN values with 0
return (no change in stock price)
```

```
RISK_FREE       3
MSFT            1
MSFT_VOLUME     1
AAPL            1
AAPL_VOLUME     1
V               1
V_VOLUME        1
UNH             1
UNH_VOLUME      1
JPM             1
JPM_VOLUME      1
JNJ             1
JNJ_VOLUME      1
WMT             1
WMT_VOLUME      1
CVX             1
CVX_VOLUME      1
PG              1
PG_VOLUME       1
HD              1
HD_VOLUME       1
dtype: int64
```

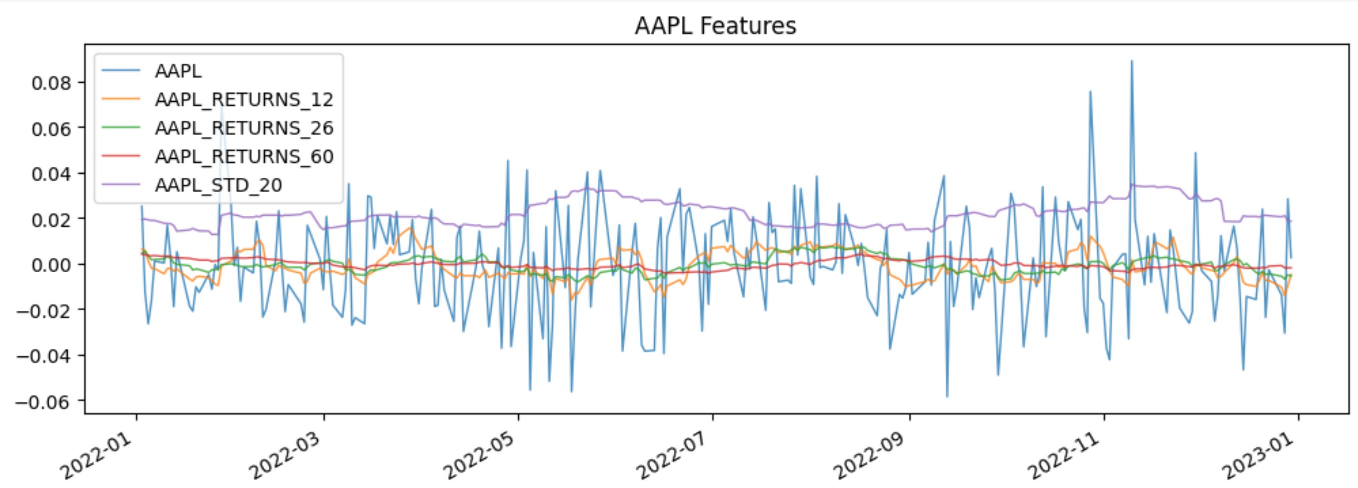
```
## VISUALIZATION OF RETURNS AND VOLUME
fig, axes = plt.subplots(2)
df[[ticker for ticker in TICKERS] + ['RISK_FREE']].plot(title='Returns',
figsize=(10, 4), legend=False, lw=0.2, alpha=0.8, ax=axes[0]);
df[[f'{ticker}_VOLUME' for ticker in TICKERS]].plot(title='Volume',
legend=False, lw=0.2, alpha=0.8, ax=axes[1]);
plt.tight_layout()
```



```
## GENERATE STATIONARY FEATURES

def rolling_returns(returns, window=10):
    return (returns+1).rolling(window=window).agg(lambda x : x.prod()) **
(1/window) - 1

def rolling_std(returns, window=10):
    return returns.rolling(window=window).std()

for ticker in tqdm(TICKERS):
    df[f'{ticker}_RETURNS_12'] = rolling_returns(df[ticker], 10)
    df[f'{ticker}_RETURNS_26'] = rolling_returns(df[ticker], 26)
    df[f'{ticker}_RETURNS_60'] = rolling_returns(df[ticker], 60)
    df[f'{ticker}_STD_20'] = rolling_std(df[ticker], 20)
```

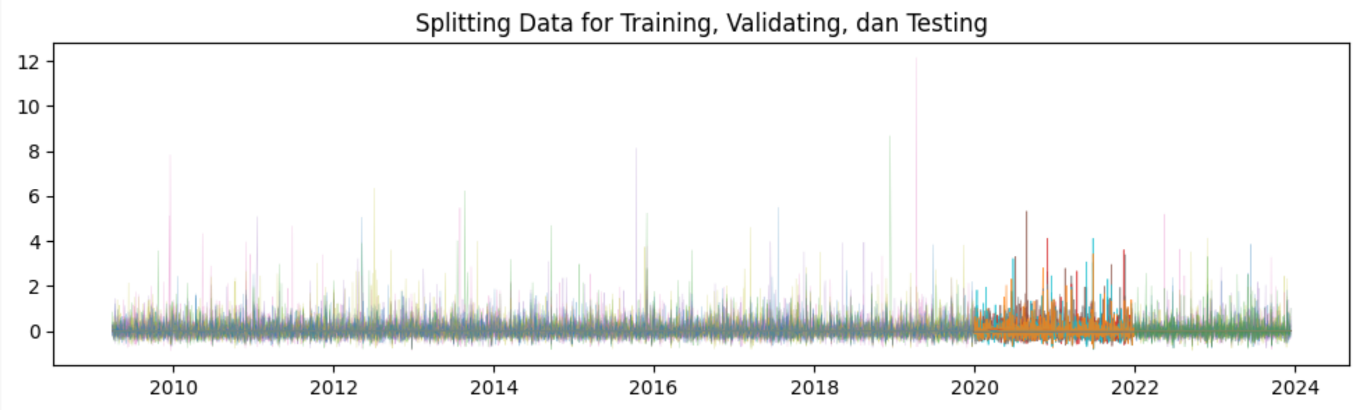
```
100%|██████████| 10/10 [00:22<00:00, 1.72s/it]
```

```
## VISUALIZATION OF STATIONARY FEATURES

ticker = 'AAPL'
features = [ticker, f'{ticker}_RETURNS_12', f'{ticker}_RETURNS_26',
f'{ticker}_RETURNS_60', f'{ticker}_STD_20']
df.loc['2022'][features].plot(title=f'{ticker} Features', legend=True, lw=1,
alpha=0.8, figsize=(12, 4));
```



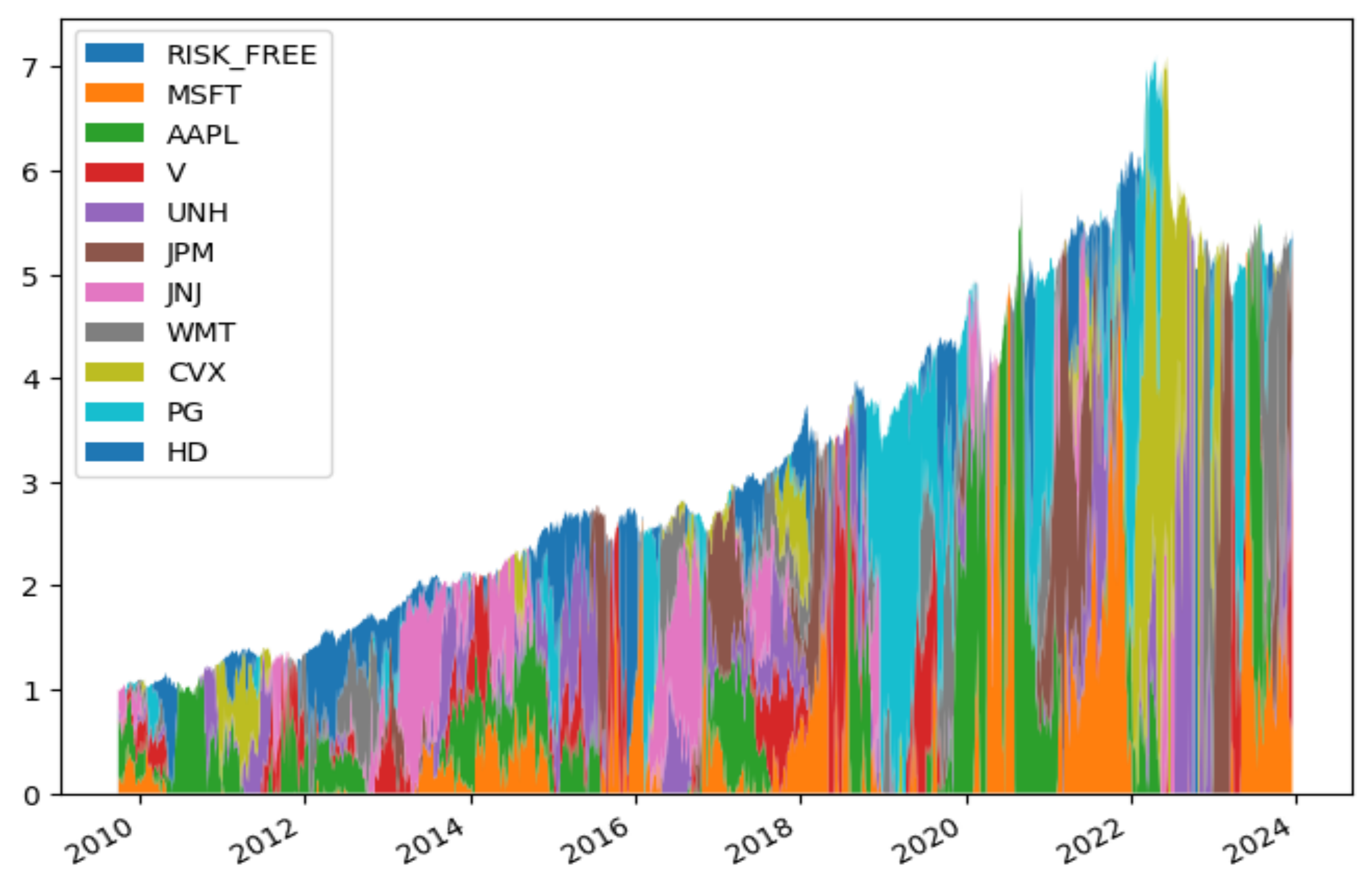

```
## TRAIN-VAL-TEST SPLIT

train_df = df[TRAIN_START : TRAIN_END]
val_df = df[VAL_START : VAL_END]
test_df = df[TEST_START : TEST_END]
plt.figure(figsize=(12,3));

plt.title('Splitting Data for Training, Validating, dan Testing')
```

```
plt.plot(train_df, alpha=0.6, lw=0.2);
plt.plot(val_df, alpha=0.8, lw=0.5);
plt.plot(test_df, alpha=0.4, lw=0.3);
```

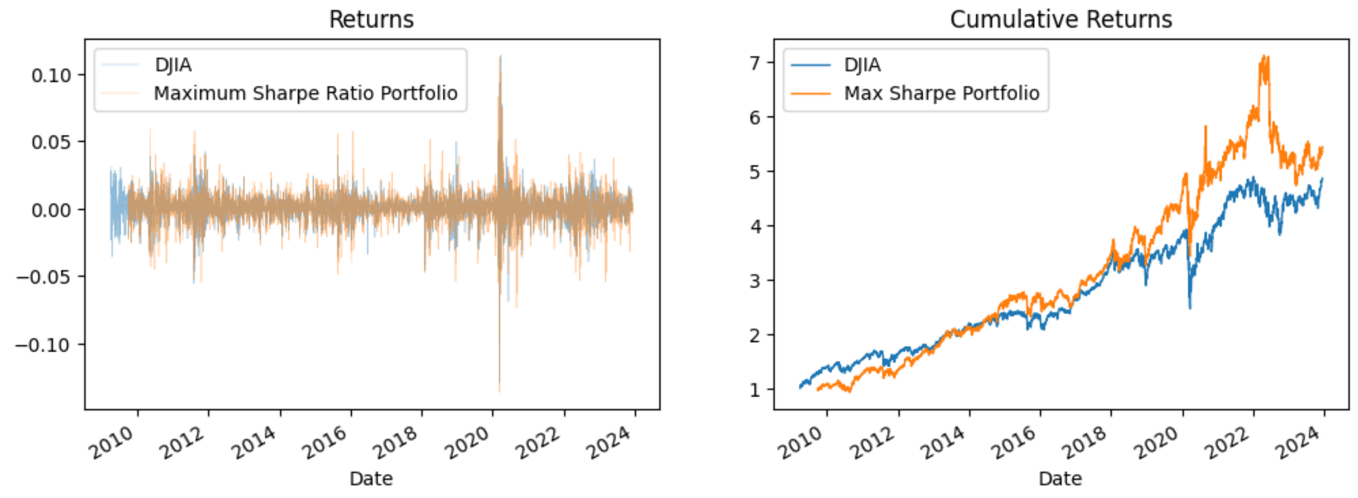



```
data_dir = 'data'

if not os.path.exists(data_dir):
    os.makedirs(data_dir)

df.to_csv(f'{data_dir}/all_data.csv')
train_df.to_csv(f'{data_dir}/train_data.csv')
val_df.to_csv(f'{data_dir}/val_data.csv')
test_df.to_csv(f'{data_dir}/test_data.csv')

# BASELINES
## DJIA BASELINE
djia_returns = yf.download('^DJI', start = df.index[0], end = df.index[-1],
interval = '1d')['Adj Close'].pct_change(1)

## MAXIMUM SHARPE RATIO BASELINE
TICKERS = ['MSFT', 'AAPL', 'V', 'UNH', 'JPM', 'JNJ', 'WMT', 'CVX', 'PG', 'HD']
RETURN_COLS = ['RISK_FREE'] + TICKERS
FEATURE_COLS = TICKERS

WINDOW_SIZE = 126 # half a trading year

env = PortfolioEnv(df, RETURN_COLS, FEATURE_COLS, episode_length=-1,
window_size=WINDOW_SIZE)

obs, done = env.reset(), False

while not done:

    observation_df = pd.DataFrame(obs.reshape(-1, env.NUM_ASSETS),
columns=FEATURE_COLS)
    annualized_mean_return = mean_historical_return(observation_df,
returns_data=True, compounding=False)
```

```
    annualized_covariance = risk_matrix(observation_df, returns_data=True,
method='sample_cov')
    ef = EfficientFrontier(annualized_mean_return, annualized_covariance)

    try:
        weights =
ef.max_sharpe(risk_free_rate=(1+env.RETURNS[env.current_index,0])**252-1)
        cleaned_weights = ef.clean_weights()
        env.current_allocations = np.insert(np.array([w for w in
cleaned_weights.values()]), 0, 0)

    except Exception as e:
        print(f"Solver error: {e}")
        env.current_allocations = np.insert(np.zeros(len(FEATURE_COLS)), 0, 1) #
invest everything into the risk-free rate

    obs, reward, done, info = env.step(env.NUM_ASSETS) # do nothing

env.render()  # title='Maximum Sharpe Ratio Portfolio Allocations'
plt.legend(loc='upper left');

max_sharpe_returns = env.get_portfolio_returns()

env.close()
```



```
## VISUALIZING BASELINES
fig, axes = plt.subplots(1, 2, figsize=(12,4))

djia_returns.plot(ax = axes[0], lw=0.3, alpha=0.5, title='Returns', label='DJIA', legend=True);
max_sharpe_returns.plot(ax = axes[0], lw=0.3, alpha=0.5, title='Returns', label='Maximum Sharpe Ratio Portfolio', legend=True);

(djia_returns+1).cumprod().plot(ax = axes[1], lw=1, alpha=1, title='Cumulative Returns', label='DJIA', legend=True);
(max_sharpe_returns+1).cumprod().plot(ax = axes[1], lw=1, alpha=1, title='Cumulative Returns', label='Max Sharpe Portfolio', legend=True);
```




```
## TRAIN DQN MODEL

models_dir, log_dir = 'models', 'logs'

if not os.path.exists(models_dir):
    os.makedirs(models_dir)

if not os.path.exists(log_dir):
    os.makedirs(log_dir)

TICKERS = ['MSFT', 'AAPL', 'V', 'UNH', 'JPM', 'JNJ', 'WMT', 'CVX', 'PG', 'HD']
FEATURES = ['RETURNS_12', 'RETURNS_26', 'RETURNS_60', 'STD_20', 'VOLUME']

RETURN_COLS = ['RISK_FREE'] + [ticker for ticker in TICKERS]
FEATURE_COLS = RETURN_COLS + [f'{ticker}_{feature}' for ticker in TICKERS for feature in FEATURES]

train_env = PortfolioEnv(
    train_df,
    RETURN_COLS,
    FEATURE_COLS,
    window_size=10,
```

```
    episode_length=180,
    allocations_in_obs=True,
)

#Below codelines will be commented out if the model has already been trained
model = DQN(
     policy='MlpPolicy',
     env=train_env,
     verbose=1,
     tensorboard_log=log_dir,
     learning_rate=3e-1, #3e-4
     batch_size=64,
     buffer_size=100_000,
     exploration_fraction=1.05,
     seed=5,
 )

#Will be commented out if the model has already been trained
TIMESTEPS = 100 # number of timesteps between saves #10_000
for i in tqdm(range(1, 300)):
     model.learn(total_timesteps=TIMESTEPS, reset_num_timesteps=False,
tb_log_name='DQN-Model')
     model.save(f'{models_dir}/{TIMESTEPS*i}')
```

```
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
/usr/local/lib/python3.10/dist-packages/gym/core.py:256: DeprecationWarning: WARN:
Function `env.seed(seed)` is marked as deprecated and will be removed in the future.
Please use `env.reset(seed=seed)` instead.
  deprecation(
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
----------------------------------
| rollout/            |          |
|    ep_len_mean      | 180      |
|    ep_rew_mean      | 6.64     |
|    exploration_rate | 0.186    |
| time/               |          |
|    episodes         | 4        |
|    fps              | 769      |
|    time_elapsed     | 0        |
|    total_timesteps  | 720      |
----------------------------------
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
.
.
.
```

```
.
.
.
.
----------------------------------
| rollout/              |          |
|    ep_len_mean        | 180      |
|    ep_rew_mean        | 6.45     |
|    exploration_rate   | 0.0977   |
| time/                 |          |
|    episodes           | 164      |
|    fps                | 1556     |
|    time_elapsed       | 0        |
|    total_timesteps    | 29520    |
----------------------------------
Logging to logs/DQN-Model_0
Logging to logs/DQN-Model_0
```

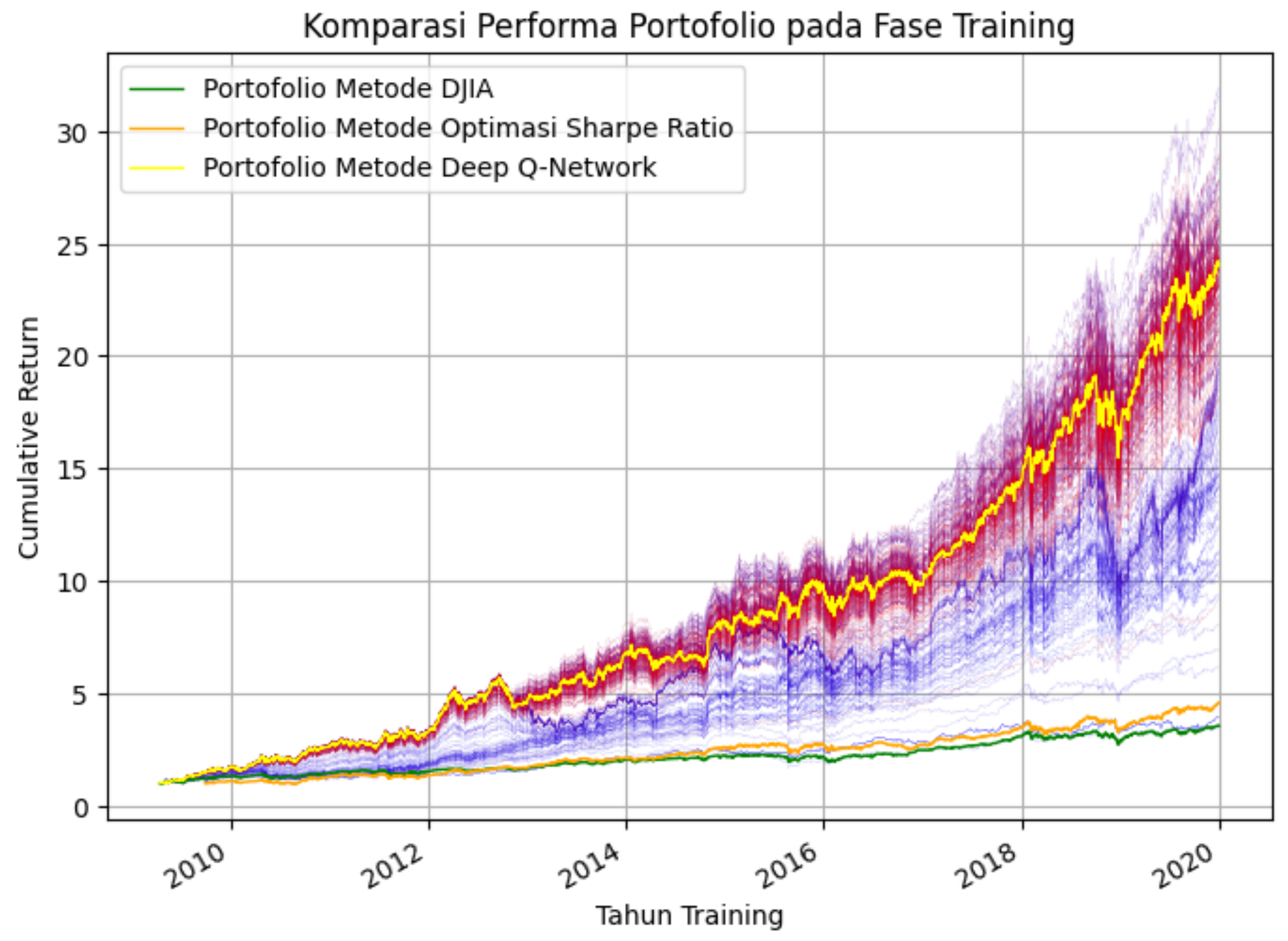
```python
## FUNCTIONS TO VISUALIZE RESULTS

def returns_to_stats(returns):
    """
    Returns the annualized mean rate of return, annualized risk, and Sharpe
ratio given an array of daily returns.
    """
    annualized_mean_rate_of_return = (1 + returns).prod() ** (252 /
len(returns)) - 1
    annualized_risk = (returns.var() * 252) ** 0.5
    sharpe_ratio =  annualized_mean_rate_of_return / annualized_risk
    return {
        'rate of return' : annualized_mean_rate_of_return,
        'risk' : annualized_risk,
        'sharpe' : sharpe_ratio
    }

def linear_color_map(x, start_color=[1,0,0], end_color=[0,0,1]):
    """
    Maps a number x (between 0 and 1) to a color between `start_color` and
`end_color`.
    """
    return [x*c1 + (1-x)*c2 for c1, c2 in zip(start_color, end_color)]

def get_returns_from_models(df, start_timestep=10_000, end_timestep=3_000_000,
step=10_000, models_dir=models_dir):
    """
    Create a dictionary of returns for each timestep in the training process.
    """

    returns_dict = {}

    env = PortfolioEnv(
        df,
```

```
        RETURN_COLS,
        FEATURE_COLS,
        window_size=10,
        episode_length=-1,
        allocations_in_obs=True,
    )

    for model_number in tqdm(range(start_timestep, end_timestep, step)):

        model = DQN.load(f'{models_dir}/{model_number}'

        obs, done = env.reset(), False
        while not done:
            action, _states = model.predict(obs, deterministic=True)
            obs, _reward, done, _info = env.step(action)

        returns_dict[model_number] = env.get_portfolio_returns().copy()

        del model

    return returns_dict

## TRAINING THE DATA

train_returns_dict = get_returns_from_models(train_df, start_timestep=10000,
end_timestep=3000000, step=10000)
```

```
100% 299/299 [05:33<00:00, 1.10s/it]
```

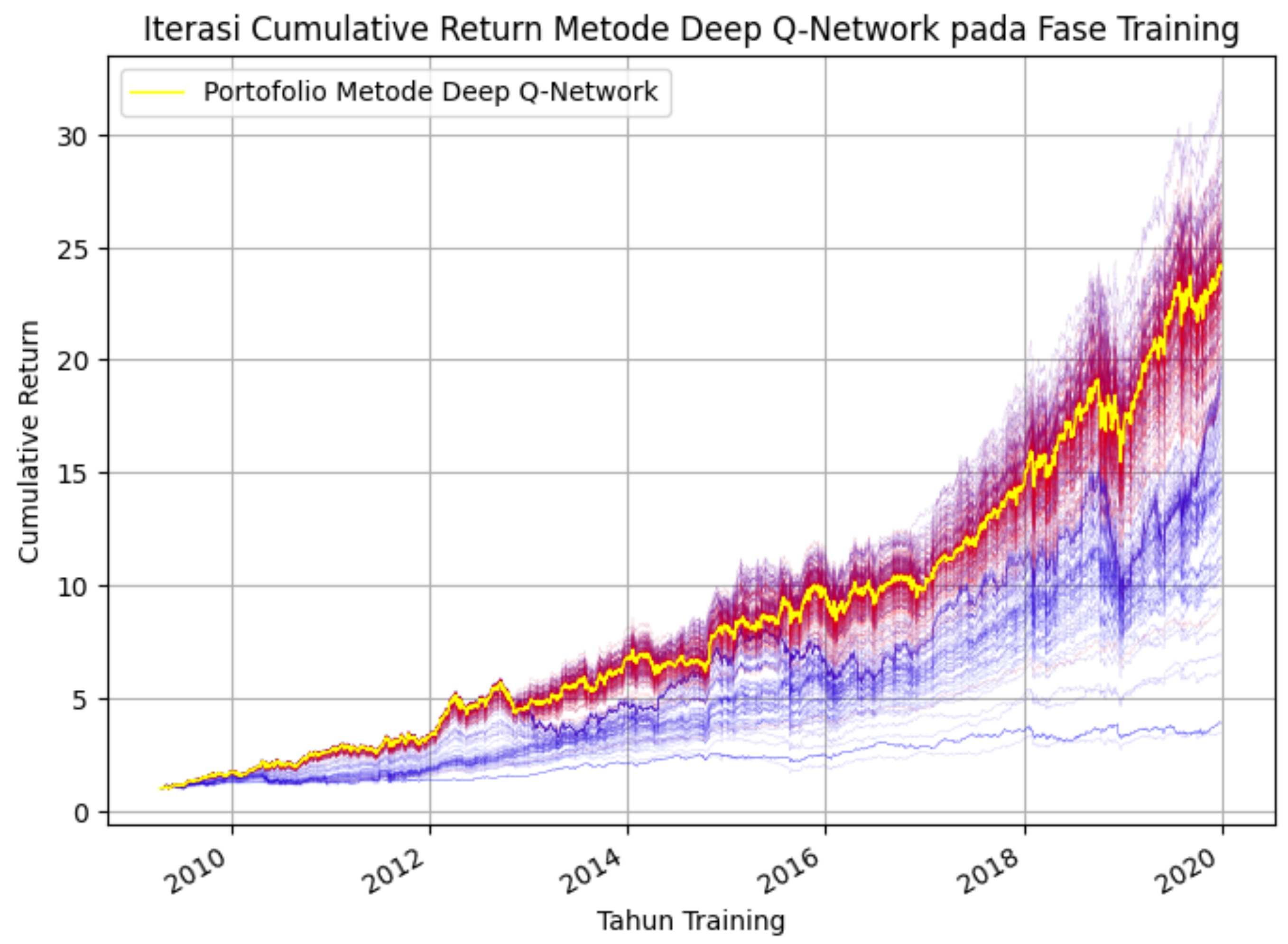
```
env = PortfolioEnv(train_df, RETURN_COLS, FEATURE_COLS, window_size=10,
episode_length=-1, allocations_in_obs=True)
model = DQN.load(f'{models_dir}/{2_900_000}')
obs, done = env.reset(), False
while not done:
    action, _states = model.predict(obs, deterministic=True)
    obs, _reward, done, _info = env.step(action)
model_train_returns = env.get_portfolio_returns().copy()
del model

plt.figure(figsize=(8,6))
for model_number, returns in train_returns_dict.items():
    (1+returns).cumprod().plot(alpha=0.5, lw=0.1,
color=linear_color_map(model_number/max(train_returns_dict.keys())))
(1+djia_returns.loc[djia_returns.index.intersection(model_train_returns.index)])
.cumprod().plot(color='green', lw=1, label='Portofolio Metode DJIA');
```

```
(1+max_sharpe_returns.loc[max_sharpe_returns.index.intersection(model_train_retu
rns.index)]).cumprod().plot(color='orange', lw=1, label='Portofolio Metode
Optimasi Sharpe Ratio');
(1+model_train_returns).cumprod().plot(color='yellow', lw=1, label='Portofolio
Metode Deep Q-Network');
plt.legend();
plt.title("Komparasi Performa Portofolio pada Fase Training")
plt.xlabel('Tahun Training')
plt.ylabel('Cumulative Return')
plt.grid();
```



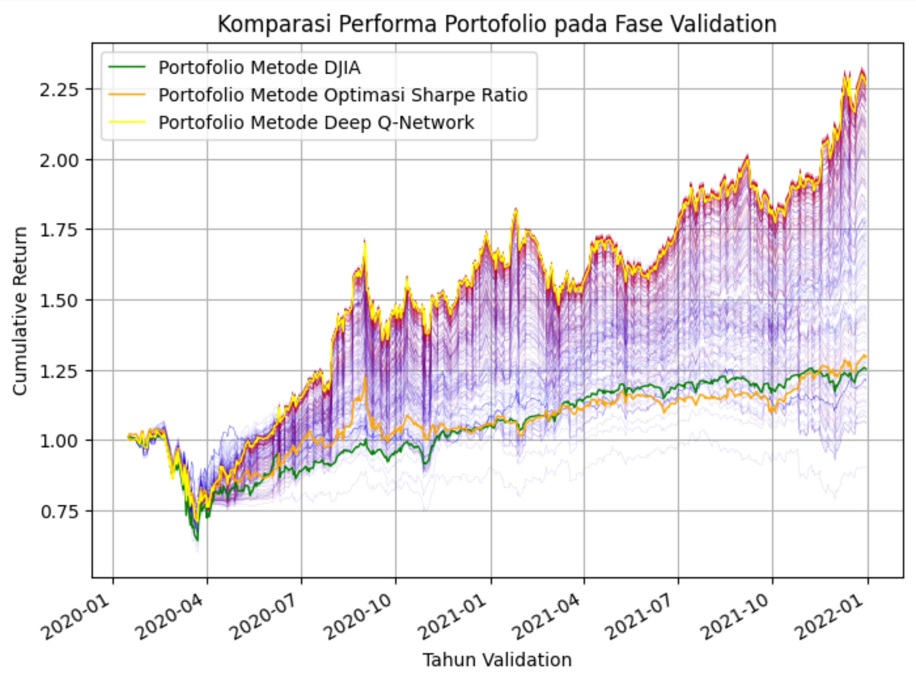

```
env = PortfolioEnv(train_df, RETURN_COLS, FEATURE_COLS, window_size=10,
episode_length=-1, allocations_in_obs=True)
model = DQN.load(f'{models_dir}/{2_900_000}')
obs, done = env.reset(), False
while not done:
    action, _states = model.predict(obs, deterministic=True)
    obs, _reward, done, _info = env.step(action)
model_train_returns = env.get_portfolio_returns().copy()
del model

plt.figure(figsize=(8,6))
```

```
for model_number, returns in train_returns_dict.items():
    (1+returns).cumprod().plot(alpha=0.5, lw=0.1,
color=linear_color_map(model_number/max(train_returns_dict.keys())))
(1+model_train_returns).cumprod().plot(color='yellow', lw=1, label='Portofolio
Metode Deep Q-Network');
plt.legend();
plt.title("Iterasi Cumulative Return Metode Deep Q-Network pada Fase Training")
plt.xlabel('Tahun Training')
plt.ylabel('Cumulative Return')
plt.grid();
```



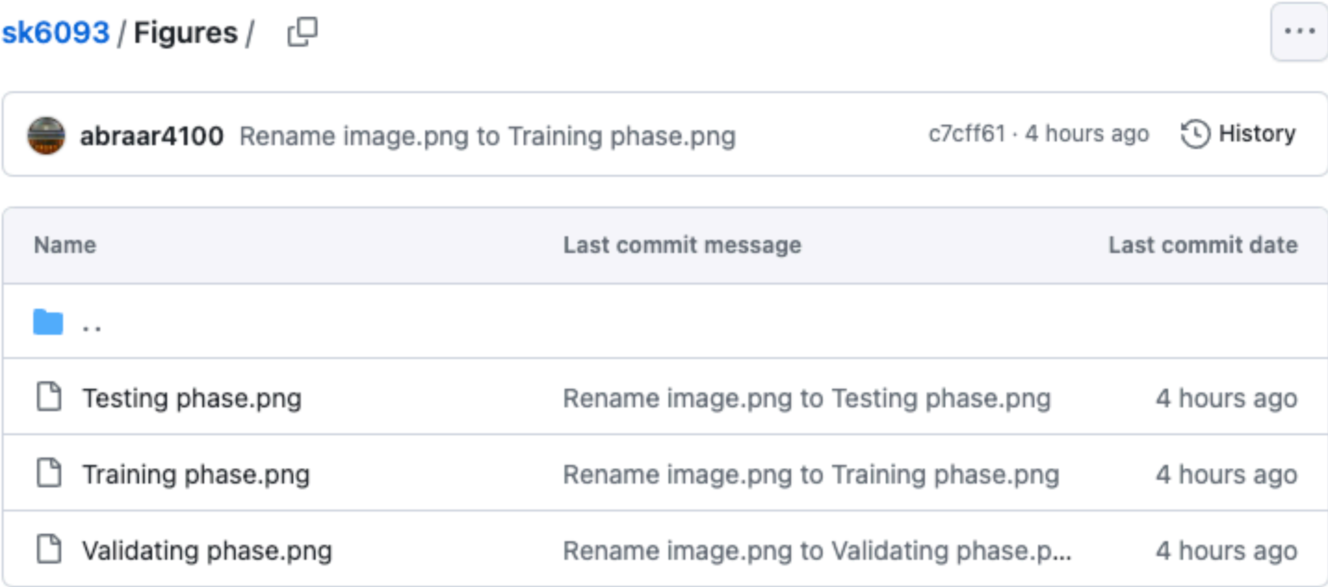

```
print('DJIA:',
returns_to_stats(djia_returns.loc[djia_returns.index.intersection(model_train_re
turns.index)]))
print('Max Sharpe:',
returns_to_stats(max_sharpe_returns.loc[max_sharpe_returns.index.intersection(mo
del_train_returns.index)]))
print('Deep Q-Network:', returns_to_stats(model_train_returns))
DJIA: {'rate of return': 0.12532739225728795, 'risk': 0.14322190901783946,
'sharpe': 0.8750574064871416}
Max Sharpe: {'rate of return': 0.15985396591550338, 'risk': 0.16741084779462778,
'sharpe': 0.954860261574}
Deep Q-Network: {'rate of return': 0.3456887158873767, 'risk':
0.2277909691202505, 'sharpe': 1.51756988972152}
```

```
## VALIDATION PHASE

val_returns_dict = get_returns_from_models(val_df, start_timestep=10_000,
end_timestep=3_000_000, step=10_000)

env = PortfolioEnv(val_df, RETURN_COLS, FEATURE_COLS, window_size=10,
episode_length=-1, allocations_in_obs=True)
model = DQN.load(f'{models_dir}/{2_900_000}')
obs, done = env.reset(), False
while not done:
    action, _states = model.predict(obs, deterministic=True)
    obs, _reward, done, _info = env.step(action)
model_val_returns = env.get_portfolio_returns().copy()
del model

plt.figure(figsize=(8,6))
for model_number, returns in val_returns_dict.items():
    (1+returns).cumprod().plot(alpha=0.5, lw=0.1,
color=linear_color_map(model_number/max(train_returns_dict.keys())))
(1+djia_returns.loc[djia_returns.index.intersection(returns.index)]).cumprod().p
lot(color='green', lw=1, label='Portofolio Metode DJIA');
(1+max_sharpe_returns.loc[max_sharpe_returns.index.intersection(returns.index)])
.cumprod().plot(color='orange', lw=1, label='Portofolio Metode Optimasi Sharpe
Ratio');
(1+model_val_returns).cumprod().plot(color='yellow', lw=1, label='Portofolio
Metode Deep Q-Network');
plt.legend();
plt.title("Komparasi Performa Portofolio pada Fase Validation")
plt.xlabel('Tahun Validation')
plt.ylabel('Cumulative Return')
plt.grid();
```



1.2.3. Figures Folder

This folder contains the corresponding figures obtained from the .ipynb main program.



sk6093 / Figures /

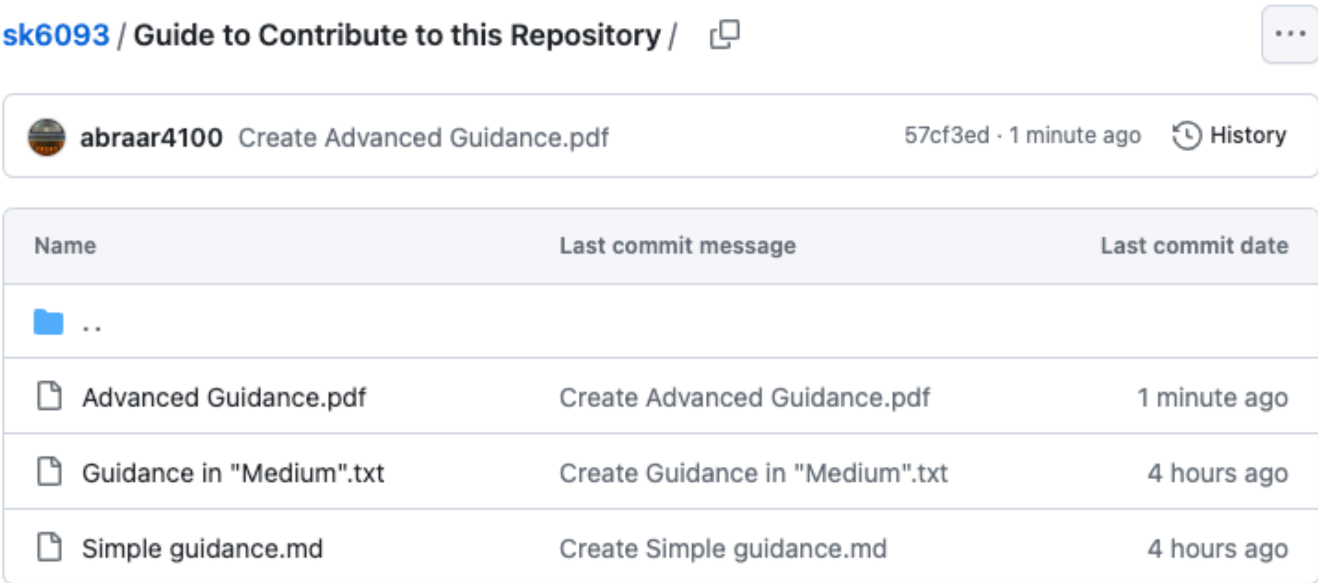
abraar4100 Rename image.png to Training phase.png c7cff61 · 4 hours ago History

Name	Last commit message	Last commit date
..		
Testing phase.png	Rename image.png to Testing phase.png	4 hours ago
Training phase.png	Rename image.png to Training phase.png	4 hours ago
Validating phase.png	Rename image.png to Validating phase.p...	4 hours ago

Figure 4. Figures Folder

1.2.4. Guide to Contribute to this Repository Folder

This folder contains the guide to contribute the repository: .pdf files (you are reading now) and a simpler guide in .txt file for GitHub reader.



sk6093 / Guide to Contribute to this Repository /

abraar4100 Create Advanced Guidance.pdf 57cf3ed · 1 minute ago History

Name	Last commit message	Last commit date
..		
Advanced Guidance.pdf	Create Advanced Guidance.pdf	1 minute ago
Guidance in "Medium".txt	Create Guidance in "Medium".txt	4 hours ago
Simple guidance.md	Create Simple guidance.md	4 hours ago

Figure 5. Guide to Contribute to this Repository Folder

— THE END OF CHAPTER 1 —

Chapter 2

Getting Started with the Data

The first thing to do is always related to the data. We obtain the data from finance.yahoo.com:

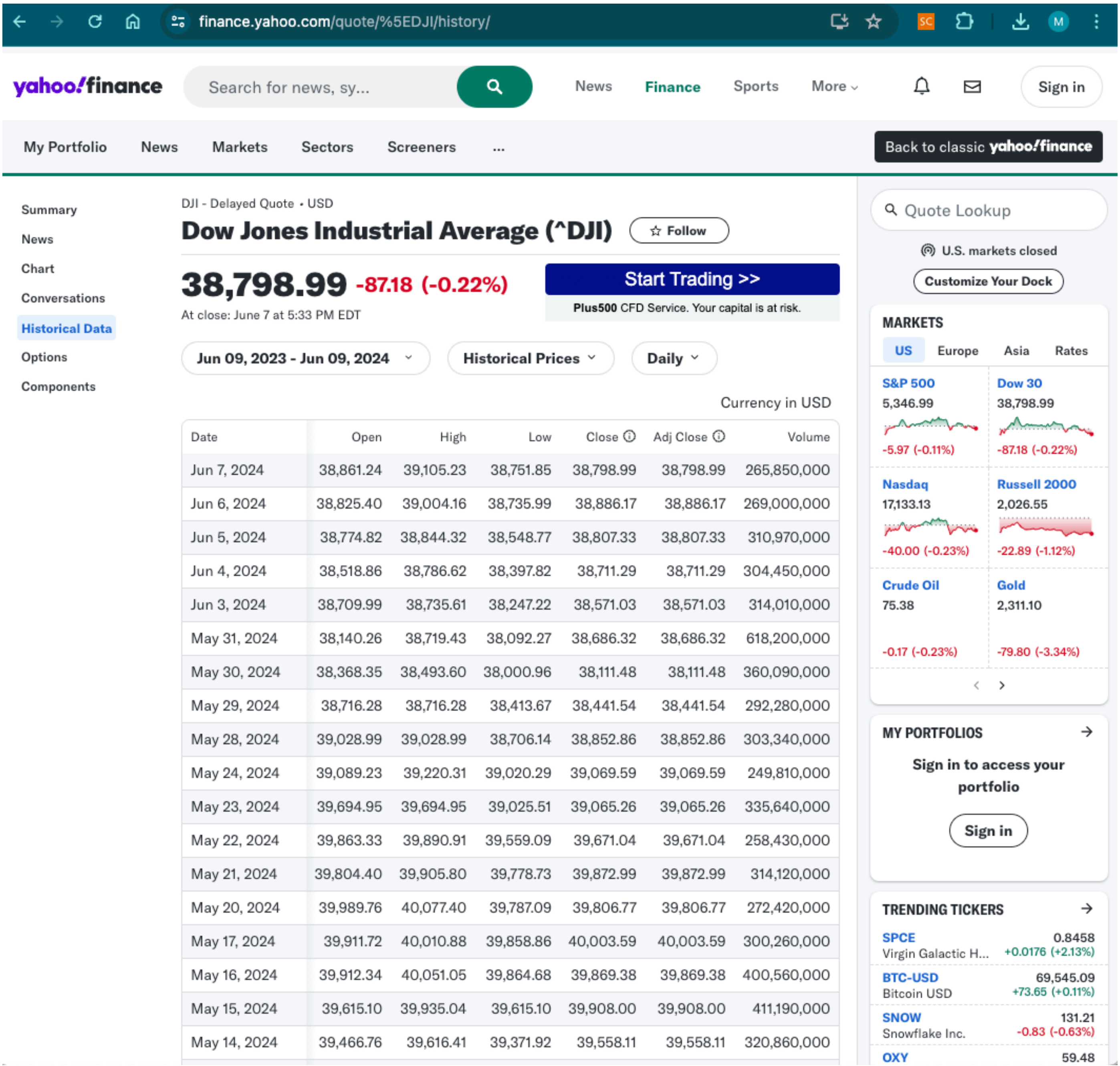


Figure 6. Data from yahoo finance

The following step-by-step tutorial refers from the attached code snippets from **Chapter 1**. Thus, please kindly refer to **Chapter 1** when following the following tutorial.

2.1. Downloading the Data

Even though we can download the data manually from the website, we will use the *yfinance* library to directly download the data from our notebook/compiler:

```
# 10 arbitrarily selected stocks from the Dow Jones
#TICKERS = ['AXP', 'AAPL', 'BA', 'GS', 'INTC', 'JNJ', 'KO', 'NKE', 'PG', 'DIS']
TICKERS = ['MSFT', 'AAPL', 'V', 'UNH', 'JPM', 'JNJ', 'WMT', 'CVX', 'PG', 'HD']

# 9 years of pre COVID-19 data
TRAIN_START = '2009-01-01'
TRAIN_END = '2019-12-31'

# 1 year of heavily COVID-19 affected data and 1 year of post COVID-19 growth
VAL_START = '2020-01-01'
VAL_END = '2021-12-31'

# 1 year of recession period
TEST_START = '2022-01-01'
TEST_END = '2023-12-14'
```

2.2. Preprocessing the Data

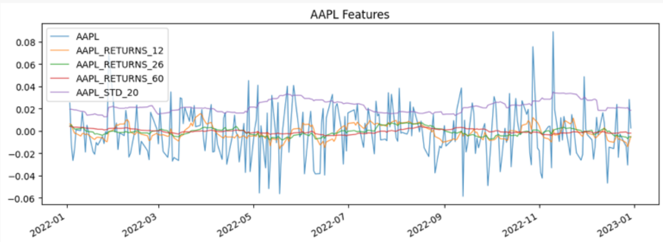
There occurs some missing values, therefore we have to remove them by doing:

```
df = pd.DataFrame(index=pd.date_range(start=TRAIN_START, end=TEST_END,
freq='d')) # create a dataframe with a full index

df['RISK_FREE'] = risk_free_rate
for ticker in TICKERS:
    df[ticker] = data[ticker]['Adj Close'].pct_change(1) # fill in each return
column
    df[f'{ticker}_VOLUME'] = data[ticker]['Volume'].pct_change(1)

print(f'Number of all NaN rows dropped: {df.isna().all(axis=1).sum()}')
df.dropna(axis=0, how='all', inplace=True) # drop rows with all NaN e.g first
row, weekends, public holidays
```

2.3. Visualizing the Stock Features

This step is important to analyze the behaviour of the data:

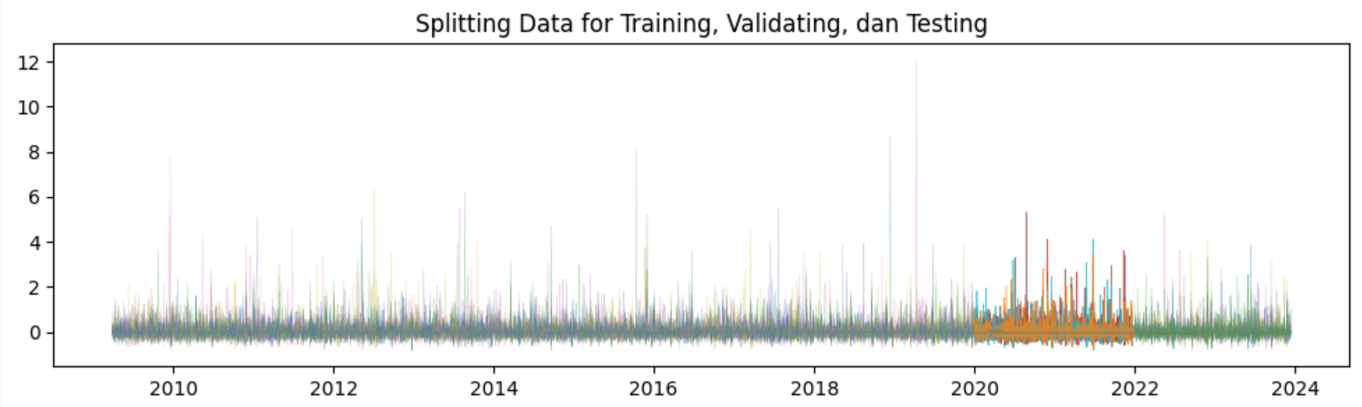
```
ticker = 'AAPL'
features = [ticker, f'{ticker}_RETURNS_12', f'{ticker}_RETURNS_26',
f'{ticker}_RETURNS_60', f'{ticker}_STD_20']
df.loc['2022'][features].plot(title=f'{ticker} Features', legend=True, lw=1,
alpha=0.8, figsize=(12, 4));
```



2.4. Splitting the data

The data are split to three steps: training, validating, and testing. The data for this is explained on the first step, and the visualization goes like this:

```
train_df = df[TRAIN_START : TRAIN_END]
val_df = df[VAL_START : VAL_END]
test_df = df[TEST_START : TEST_END]
plt.figure(figsize=(12,3));
plt.title('Splitting Data for Training, Validating, dan Testing')
plt.plot(train_df, alpha=0.6, lw=0.2);
plt.plot(val_df, alpha=0.8, lw=0.5);
plt.plot(test_df, alpha=0.4, lw=0.3);
```



2.5. Setting up the Baselines

We use 2 (two) baselines: DJIA and Sharpe Ratio. We obtain the DJIA Baseline directly from yahoo! Finance as:

```
djia_returns = yf.download('^DJI', start = df.index[0], end = df.index[-1],
interval = '1d')['Adj Close'].pct_change(1)
```

The sharpe ratio baseline is defined as follows:

```
TICKERS = ['MSFT', 'AAPL', 'V', 'UNH', 'JPM', 'JNJ', 'WMT', 'CVX', 'PG', 'HD']
RETURN_COLS = ['RISK_FREE'] + TICKERS
FEATURE_COLS = TICKERS

WINDOW_SIZE = 126 # half a trading year

env = PortfolioEnv(df, RETURN_COLS, FEATURE_COLS, episode_length=-1,
window_size=WINDOW_SIZE)

obs, done = env.reset(), False

while not done:

    observation_df = pd.DataFrame(obs.reshape(-1, env.NUM_ASSETS),
columns=FEATURE_COLS)
```

```
    annualized_mean_return = mean_historical_return(observation_df,
returns_data=True, compounding=False)
    annualized_covariance = risk_matrix(observation_df, returns_data=True,
method='sample_cov')
    ef = EfficientFrontier(annualized_mean_return, annualized_covariance)

    try:
        weights =
ef.max_sharpe(risk_free_rate=(1+env.RETURNS[env.current_index,0])**252-1)
        cleaned_weights = ef.clean_weights()
        env.current_allocations = np.insert(np.array([w for w in
cleaned_weights.values()]), 0, 0)

    except Exception as e:
        print(f"Solver error: {e}")
        env.current_allocations = np.insert(np.zeros(len(FEATURE_COLS)), 0, 1) #
invest everything into the risk-free rate

    obs, reward, done, info = env.step(env.NUM_ASSETS) # do nothing

env.render()  # title='Maximum Sharpe Ratio Portfolio Allocations'
plt.legend(loc='upper left');

max_sharpe_returns = env.get_portfolio_returns()

env.close()
```

— THE END OF CHAPTER 2 —

Chapter 3

Setting up the Main Property of DRL: Environment

The most crucial part of a Reinforcement Learning algorithm lies on its environment — it is the baseline for those agents to interact. In this chapter, I breakdown my environment thoroughly. The following environment allows an agent **to make buy, sell, or hold decisions** based on historical data and receive feedback in the form of rewards based on portfolio performance. The environment supports both discrete action spaces and customizable observations, making it flexible for various reinforcement learning scenarios.

Note that the following step-by-step tutorial refers from the attached code snippets from **Chapter 1**. Thus, please kindly refer to **Chapter 1** when following the following tutorial.

3.1. _init_

The **_init_** method sets up the environment with the given parameters and prepares it for use:

- **df**: A DataFrame containing the historical financial data.
- **return_cols**: List of column names that contain asset returns, with the first entry being risk-free returns.
- **feature_cols**: List of column names to be used as features.
- **window_size**: Size of the lookback window for observations.
- **order_size**: Size of the step in allocations.
- **starting_balance**: Initial cash balance.
- **episode_length**: Length of each episode.
- **drawdown_penalty_weight**: Penalty weight for drawdowns.
- **allocations_in_obs**: Whether to include current allocations in the observation.

```
class PortfolioEnv(gym.Env):
    def __init__(
        self,
        df,
        return_cols,
        feature_cols=[],
        window_size = 20,
        order_size = 0.1,
        starting_balance = 1,
        episode_length = 180,
        drawdown_penalty_weight = 1,
        allocations_in_obs = False
    ):
        # Data related constants
```

```
        self.RETURN_COLS = return_cols
        self.FEATURE_COLS = feature_cols
        self.NUM_ASSETS = len(return_cols)-1
        self.NUM_FEATURES = len(feature_cols)
        self.RETURNS = df[self.RETURN_COLS].to_numpy()
        self.FEATURES = df[self.FEATURE_COLS].to_numpy()
        self.INDEX = df.index

        # Environment constants
        self.WINDOW_SIZE = window_size
        self.ORDER_SIZE = order_size
        self.ALLOCATIONS_PRECISION = len(str(self.ORDER_SIZE).split('.')[-1]) #
number of decimal places of order_size
        self.STARTING_BALANCE = starting_balance
        self.EPISODE_LENGTH = episode_length
        self.DRAWDOWN_PENALTY_WEIGHT = drawdown_penalty_weight
        self.ALLOCATION_IN_OBS = allocations_in_obs

        # Initialize action/observation space
        self.action_space = gym.spaces.Discrete(self.NUM_ASSETS*2 + 1) #
buy/sell for each stock or do nothing
        if self.ALLOCATION_IN_OBS:
            self.observation_space = gym.spaces.Box(
                low = np.concatenate([self.FEATURES.min(axis=0) for _ in
range(self.WINDOW_SIZE)] + [np.zeros(self.NUM_ASSETS+1)]),
                high = np.concatenate([self.FEATURES.max(axis=0) for _ in
range(self.WINDOW_SIZE)] + [np.ones(self.NUM_ASSETS+1)]),
                shape = (self.WINDOW_SIZE*self.NUM_FEATURES +
self.NUM_ASSETS+1,),
                dtype = np.float64
            )
        else:
            self.observation_space = gym.spaces.Box(
                low = np.concatenate([self.FEATURES.min(axis=0) for _ in
range(self.WINDOW_SIZE)]),
                high = np.concatenate([self.FEATURES.max(axis=0) for _ in
range(self.WINDOW_SIZE)]),
                shape = (self.WINDOW_SIZE*self.NUM_FEATURES,),
                dtype = np.float64
            )
        # mereset environment
        self.reset()
```

3.2. reset

The **_reset** method initializes the environment to a random starting point within the data, setting initial values for allocations, current value, and history tracking.

```
    def reset(self):
        if self.EPISODE_LENGTH == -1:
```

```
            self.start_index = self.WINDOW_SIZE
        else:
            self.start_index = np.random.randint(self.WINDOW_SIZE,
len(self.RETURNS)-self.EPISODE_LENGTH) # Random start index
        self.current_index = self.start_index

        # The allocations always adds up to 1 with starting allocations as [1,
0, 0, ..., 0] (index 0 is for cash).
        self.current_allocations = np.insert(np.zeros(self.NUM_ASSETS), 0, 1.0)
        self.current_value = self.STARTING_BALANCE
        self.weighted_cumulative_return = 0

        self.return_history = [0]
        self.value_history = [self.current_value]
        self.allocations_history = [self.current_allocations.copy()]

        return self.get_observation()
```

3.3. get_observation

The **get_observation** method returns a flattened array of historical returns and features, along with current allocations if specified.

```
def get_observation(self):
            obs = self.FEATURES[self.current_index-self.WINDOW_SIZE :
self.current_index].flatten()
        if self.ALLOCATION_IN_OBS:
            obs = np.concatenate((obs, self.current_allocations))
        return obs
```

3.4. update_current_allocations

This method adjusts the current asset allocations based on the action taken by the agent.

```
def update_current_allocations(self, action):
        action -= self.NUM_ASSETS # Convert the action to a number between -
len(ASSETS) and +len(ASSETS)
        action_asset, action_sign = abs(action), np.sign(action)

        # If we want to do nothing
        if action_sign==0:
            return # exit the function

        # If we want to buy and have cash (e.g action +3 means we want to buy
the asset at position 3).
        elif (action_sign>0) and (self.current_allocations[0]>0):
            self.current_allocations[action_asset] += self.ORDER_SIZE
            self.current_allocations[0] -= self.ORDER_SIZE

        # If we want to sell and have the asset (e.g -1 means we want to sell
asset at position 1).
        elif (action_sign<0) and (self.current_allocations[action_asset]>0):
```

```
            self.current_allocations[action_asset] -= self.ORDER_SIZE
            self.current_allocations[0] += self.ORDER_SIZE

        # Round to avoid floating point error
        self.current_allocations =
self.current_allocations.round(decimals=self.ALLOCATIONS_PRECISION)
```

3.5. update_current_value

This method updates the current value of the portfolio based on the current allocations and returns at the current index.

```
def update_current_value(self):
        previous_value = self.current_value
        self.current_value *=
((1+self.RETURNS[self.current_index])*self.current_allocations).sum()
        return previous_value
```

3.6. step

This method progresses the environment by one time step, updates the allocations and value, calculates the reward, and returns the new state, reward, and done flag.

```
def step(self, action):
        self.current_index += 1

        if self.EPISODE_LENGTH == -1:
            done = bool(self.current_index >= len(self.RETURNS)-1)
        else:
            done = bool(self.current_index - self.start_index >=
self.EPISODE_LENGTH)

        self.update_current_allocations(action)
        previous_value = self.update_current_value()
        ret = (self.current_value - previous_value) / previous_value

        if ret > 0:
            self.weighted_cumulative_return = (1 +
self.weighted_cumulative_return) * (1 + ret) - 1
        else:
            self.weighted_cumulative_return = (1 +
self.weighted_cumulative_return) * (1 + self.DRAWDOWN_PENALTY_WEIGHT * ret) - 1

        reward = self.weighted_cumulative_return * (self.current_index -
self.start_index)/self.EPISODE_LENGTH
        observation = self.get_observation()

        self.return_history.append(ret)
        self.value_history.append(self.current_value)
        self.allocations_history.append(self.current_allocations)
```

```
        return observation, reward, done, {}
```

3.7. render

This method visualizes the portfolio value changes over time using a stack plot.

```
def render(self, ax=None, title='', legend=False):
        value_history_array = np.array(self.value_history).reshape(-1, 1)
        allocations_history_array = np.array(self.allocations_history)
        value_breakdown = (value_history_array *
allocations_history_array).transpose()

        if ax==None:
            plt.figure(figsize=(8,6))
            ax = plt.axes()

        ax.set_title(title)
        ax.stackplot(
            self.INDEX[self.start_index : self.current_index+1],
            value_breakdown,
            labels = self.RETURN_COLS,
        );

        plt.gcf().autofmt_xdate();
```

3.8. get_portfolio_returns

This method returns the portfolio returns over time as a pandas Series.

```
    def get_portfolio_returns(self):
        return pd.Series(
            self.return_history,
            index=self.INDEX[self.start_index : self.current_index+1])
```

— THE END OF CHAPTER 3 —

Chapter 4

Executing the Main Program

Now we are ready to execute the main program. Note that the following step-by-step tutorial refers from the attached code snippets of **main_program** from **Chapter 1**. Thus, please kindly refer to **Chapter 1** when following the following tutorial.

In the following explanation, the **main_program.ipynb** will only be explained from the **training phase** since I have explained the rest of the **main_program.ipynb** on the preceding chapters.

4.1. Training Phase

The following code defines the training phase:

```
train_returns_dict = get_returns_from_models(train_df, start_timestep=10000,
end_timestep=3000000, step=10000)

env = PortfolioEnv(train_df, RETURN_COLS, FEATURE_COLS, window_size=10,
episode_length=-1, allocations_in_obs=True)
model = DQN.load(f'{models_dir}/{2_900_000}')
obs, done = env.reset(), False
while not done:
    action, _states = model.predict(obs, deterministic=True)
    obs, _reward, done, _info = env.step(action)
model_train_returns = env.get_portfolio_returns().copy()
del model

plt.figure(figsize=(8,6))
```

Pay attention to the following code! The below code plots all the training process occurred in the training phase as attached in **Figure 7.**

```
for model_number, returns in train_returns_dict.items():
    (1+returns).cumprod().plot(alpha=0.5, lw=0.1,
color=linear_color_map(model_number/max(train_returns_dict.keys())))
(1+djia_returns.loc[djia_returns.index.intersection(model_train_returns.index)])
.cumprod().plot(color='green', lw=1, label='Portofolio Metode DJIA');
(1+max_sharpe_returns.loc[max_sharpe_returns.index.intersection(model_train_retu
rns.index)]).cumprod().plot(color='orange', lw=1, label='Portofolio Metode
Optimasi Sharpe Ratio');
(1+model_train_returns).cumprod().plot(color='yellow', lw=1, label='Portofolio
Metode Deep Q-Network');
plt.legend();
plt.title("Komparasi Performa Portofolio pada Fase Training")
plt.xlabel('Tahun Training')
plt.ylabel('Cumulative Return')
plt.grid();
```

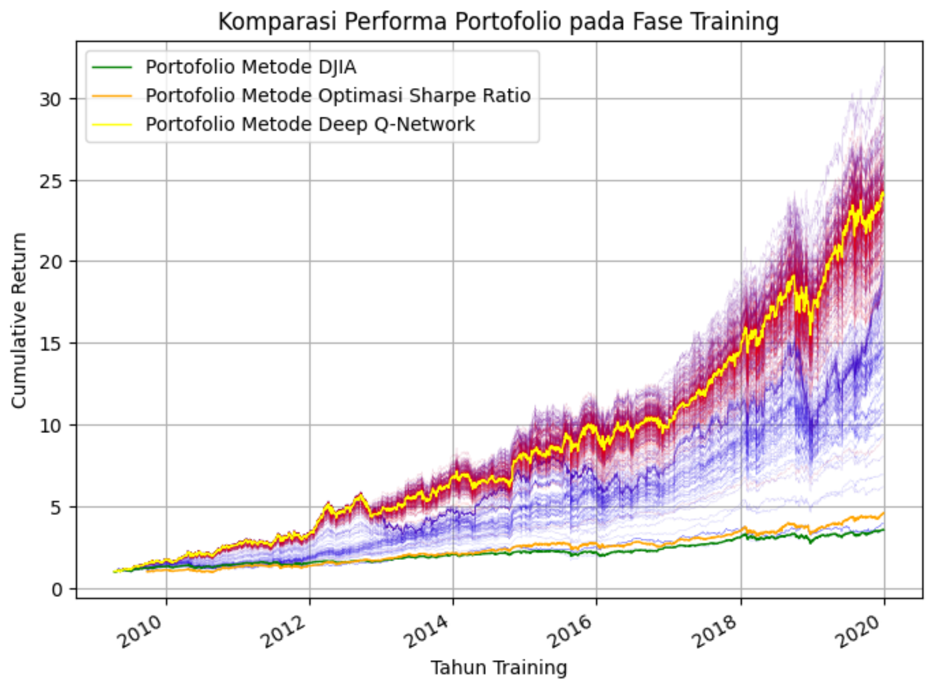


Figure 7. Training phase result

The below code prints out the obtained portfolio return compared to other baselines

```
print('DJIA:',
returns_to_stats(djia_returns.loc[djia_returns.index.intersection(model_train_re
turns.index)]))
print('Max Sharpe:',
returns_to_stats(max_sharpe_returns.loc[max_sharpe_returns.index.intersection(mo
del_train_returns.index)]))
print('Deep Q-Network:', returns_to_stats(model_train_returns))
```

```
DJIA: {'rate of return': 0.12532739225728795, 'risk': 0.14322190901783946,
'sharpe': 0.8750574064871416}
Max Sharpe: {'rate of return': 0.15985396591550338, 'risk': 0.16741084779462778,
'sharpe': 0.954860261574}
Deep Q-Network: {'rate of return': 0.3456887158873767, 'risk': 0.2277909691202505,
'sharpe': 1.51756988972152}
```

4.2. Validating Phase

The following code defines the validating phase:

```
val_returns_dict = get_returns_from_models(val_df, start_timestep=10_000,
end_timestep=3_000_000, step=10_000)
```

Pay attention to the following code! The below code plots all the training process occurred in the validating phase as attached in **Figure 8.**

```
env = PortfolioEnv(val_df, RETURN_COLS, FEATURE_COLS, window_size=10,
episode_length=-1, allocations_in_obs=True)
model = DQN.load(f'{models_dir}/{2_900_000}')
obs, done = env.reset(), False
while not done:
    action, _states = model.predict(obs, deterministic=True)
    obs, _reward, done, _info = env.step(action)
```

```
model_val_returns = env.get_portfolio_returns().copy()
del model

plt.figure(figsize=(8,6))
for model_number, returns in val_returns_dict.items():
    (1+returns).cumprod().plot(alpha=0.5, lw=0.1,
color=linear_color_map(model_number/max(train_returns_dict.keys())))
(1+djia_returns.loc[djia_returns.index.intersection(returns.index)]).cumprod().p
lot(color='green', lw=1, label='Portofolio Metode DJIA');
(1+max_sharpe_returns.loc[max_sharpe_returns.index.intersection(returns.index)])
.cumprod().plot(color='orange', lw=1, label='Portofolio Metode Optimasi Sharpe
Ratio');
(1+model_val_returns).cumprod().plot(color='yellow', lw=1, label='Portofolio
Metode Deep Q-Network');
plt.legend();
plt.title("Komparasi Performa Portofolio pada Fase Validation")
plt.xlabel('Tahun Validation')
plt.ylabel('Cumulative Return')
plt.grid();
```

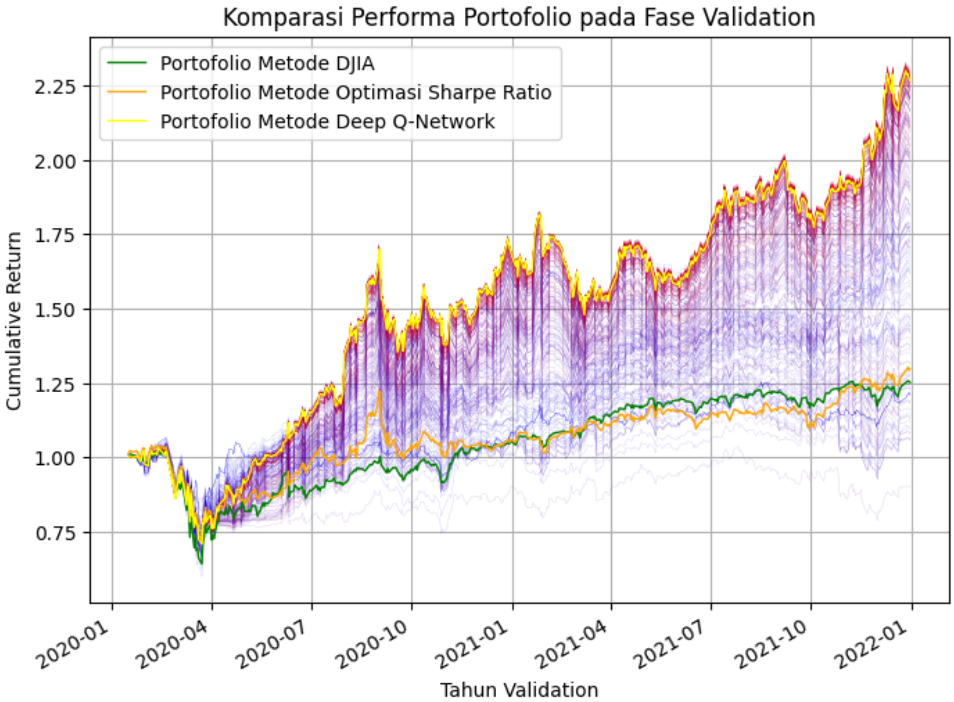


Figure 8. Validating phase result

The below code prints out the obtained portfolio return compared to other baselines

```
print('DJIA:',
returns_to_stats(djia_returns.loc[djia_returns.index.intersection(model_val_retu
rns.index)]))
print('Max Sharpe:',
returns_to_stats(max_sharpe_returns.loc[max_sharpe_returns.index.intersection(mo
del_val_returns.index)]))
print('Deep Q-Network:', returns_to_stats(model_val_returns))
```

```
DJIA: {'rate of return': 0.12109814074104142, 'risk': 0.27678284711303075,
'sharpe': 0.43752039551637445}
Max Sharpe: {'rate of return': 0.14180056643023642, 'risk': 0.3053466428838095,
'sharpe': 0.4643920925117043}
Deep Q-Network: {'rate of return': 0.5166226170592525, 'risk': 0.3761682233798835,
'sharpe': 1.373381867339516}
```

4.3. Testing Phase

The following code defines the testing phase:

```
test_returns_dict = get_returns_from_models(test_df, start_timestep=10_000,
end_timestep=3_000_000, step=10_000)
```

Pay attention to the following code! The below code plots all the training process occurred in the testing phase as attached in **Figure 9.**

```
env = PortfolioEnv(test_df, RETURN_COLS, FEATURE_COLS, window_size=10,
episode_length=-1, allocations_in_obs=True)
model = DQN.load(f'{models_dir}/{2_900_000}')
obs, done = env.reset(), False
while not done:
    action, _states = model.predict(obs, deterministic=True)
    obs, _reward, done, _info = env.step(action)
model_test_returns = env.get_portfolio_returns().copy()
del model

plt.figure(figsize=(8,6))
for model_number, returns in test_returns_dict.items():
    (1+returns).cumprod().plot(alpha=0.5, lw=0.1,
color=linear_color_map(model_number/max(train_returns_dict.keys())))
(1+djia_returns.loc[djia_returns.index.intersection(returns.index)]).cumprod().p
lot(color='green', lw=1, label='Portofolio Metode DJIA');
(1+max_sharpe_returns.loc[max_sharpe_returns.index.intersection(returns.index)])
.cumprod().plot(color='orange', lw=1, label='Portofolio Metode Optimasi Sharpe
Ratio');
(1+model_test_returns).cumprod().plot(color='yellow', lw=1, label='Portofolio
Metode Deep Q-Network');
plt.legend();
plt.title("Komparasi Performa Portofolio pada Fase Testing")
plt.xlabel('Tahun Testing')
plt.ylabel('Cumulative Return')
plt.grid();
```

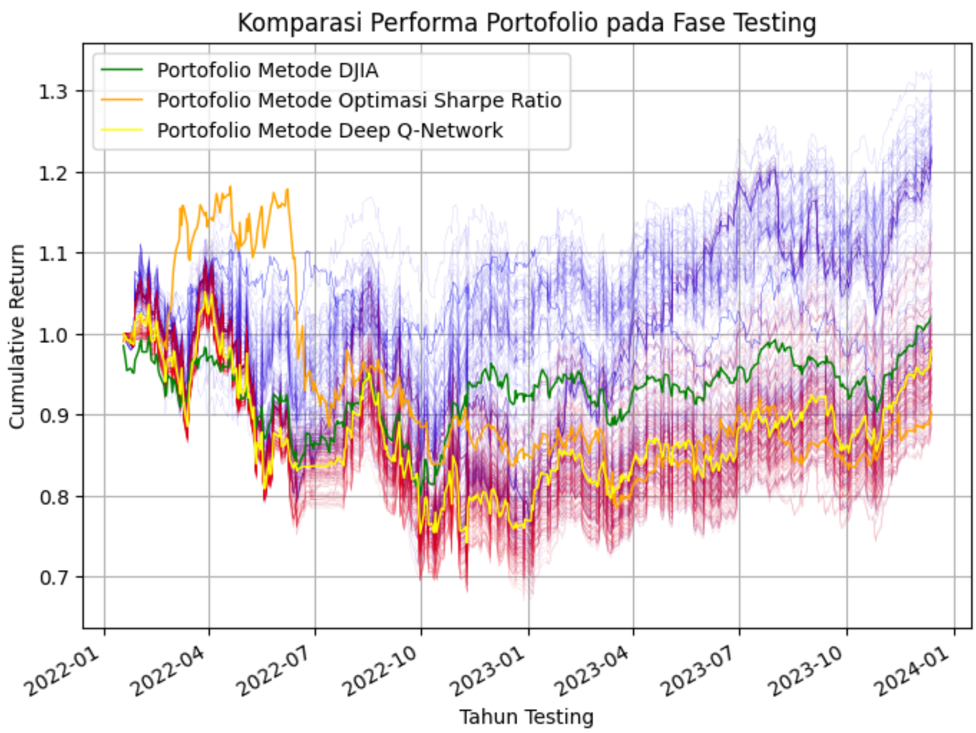


Figure 8. Validating phase result

The below code prints out the obtained portfolio return compared to other baselines

```
print('DJIA:',
returns_to_stats(djia_returns.loc[djia_returns.index.intersection(model_test_ret
urns.index)]))
print('Max Sharpe:',
returns_to_stats(max_sharpe_returns.loc[max_sharpe_returns.index.intersection(mo
del_test_returns.index)]))
print('Deep Q-Network:', returns_to_stats(model_test_returns))
```

```
DJIA: {'rate of return': 0.009716118640073601, 'risk': 0.16410429050686656,
'sharpe': 0.05920697508921653}
Max Sharpe: {'rate of return': -0.05215221916187596, 'risk': 0.202363578089592,
'sharpe': -0.25771544293799115}
Deep Q-Network: {'rate of return': -0.010812475299892599, 'risk':
0.24526302445762627, 'sharpe': -0.044085223705461786}
```

— THE END OF CHAPTER 4 —