YANDEX Stock
Configuration Manual

1. Introduction

This manual documents all the details about hardware and software specifications that have been used in this research process. All the following sections illustrate the steps that should be followed to setup and run the environment for this study. It also contains different applications that should be configured and utilized.

2. System Requirements

The system specification contains all the resources used in order to complete the research study. Figure 1 shows the specifications of the machine used to run this project while Figure 2 shows the windows specifications for the machine utilized for research. All hardware accelerators have been utilized for research and the training was done on CPU due to GPU being AMD only.

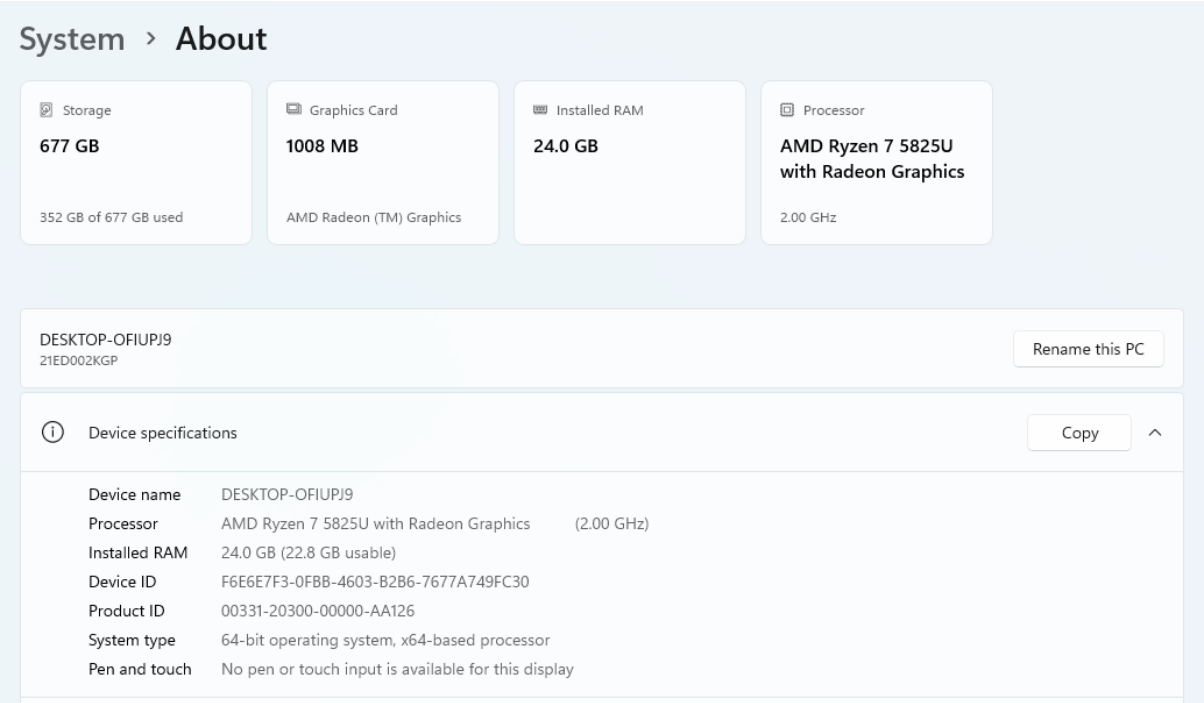

Figure 1. System specifications

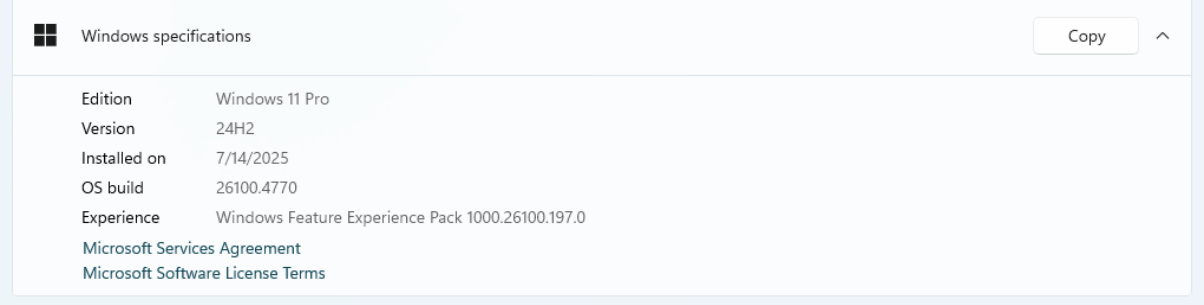

Figure 2. Windows specifications

2.1 Tools and Softwares

- Python 3: All the code is written in python
- Jupyter notebook: All the modeling is done using jupyter notebooks (it allows to run notebooks in browser)

3. Dataset specification

The dataset used for this research is ***YANDEX Stock***. This is a timeseries dataset with 24 collection points (1 for each hour of the day). Dataset has date and time that specify its temporal nature and the rest of the columns are from quantitative trading. Figure 3 shows a small snippet of the dataset.

For model training and validation this 80% of the total is used as training data and 20% is used as validation data

	<DATE>	<TIME>	<OPEN>	<HIGH>	<LOW>	<CLOSE>	<VOL>
1	20240411	10000	0.227293	0.231184	0.226998	0.230291	3055686.9127095
2	20240411	20000	0.230193	0.230393	0.229093	0.229891	1850548.0432534
3	20240411	30000	0.229791	0.230691	0.228879	0.229779	2020891.9336963
4	20240411	40000	0.229779	0.2308	0.2294	0.229598	1697452.4731928
5	20240411	50000	0.2296	0.231891	0.229389	0.230184	2944267.0933747
6	20240411	60000	0.230086	0.231198	0.229482	0.230298	2871484.0457681
7	20240411	70000	0.2305	0.2305	0.229595	0.229714	2134099.1236177
8	20240411	80000	0.229716	0.229716	0.2297	0.229705	425396.35745346
9	20240411	90000	0.230405	0.232402	0.230102	0.232105	1896095.7846268
10	20240411	100000	0.232102	0.233198	0.231505	0.233177	1096058.804049
11	20240411	110000	0.233179	0.234486	0.232688	0.234486	1867768.930118
12	20240411	120000	0.234384	0.234586	0.232063	0.232163	3041664.2920294
13	20240411	130000	0.232063	0.232356	0.230556	0.231054	2176072.0635766
14	20240411	140000	0.231054	0.232056	0.228584	0.228584	2495174.6973695
15	20240411	150000	0.228784	0.232593	0.227982	0.231077	2145914.9368667
16	20240411	160000	0.230977	0.232581	0.2287	0.231193	1737117.9258622
17	20240411	170000	0.231393	0.231393	0.227989	0.228064	1022878.826684
18	20240411	180000	0.228064	0.230647	0.22802	0.230547	1501690.5112223
19	20240411	190000	0.230447	0.231051	0.22665	0.228543	2496093.7758439
20	20240411	200000	0.228641	0.230863	0.228441	0.230075	1842751.2317969
21	20240411	210000	0.230077	0.230693	0.228595	0.230098	5104504.1056609
22	20240411	220000	0.230198	0.230395	0.228395	0.2293	2345418.0690999
23	20240411	230000	0.2294	0.229905	0.228698	0.229805	1810083.0296966
24	20240411	235900	0.229805	0.229905	0.226814	0.227016	2179753.1130169

Figure 3. Chunk of data extracted from dataset used in this research

The training and testing data is placed in their respective comma-separated-value (CSV) files. These are loaded and used as it is while model training / validation.

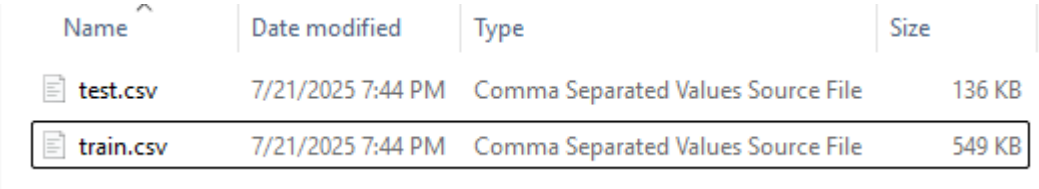

Figure 4. Training and testing csv files

4. Project Development

After cloning the datasets for training and test and storing it on disk, notebooks can be launched. First open the directory containing all the notebooks, open a command line from the directory and type "jupyter notebook" to run jupyter in the browser. Figure 5 (a) illustrates the command line upon launching jupyter notebook. As all the code files are stored in .ipynb file it can be opened easily and will contain all previous run results. There is also the option available to run all the notebooks simultaneously or you can also choose to run them one by one.

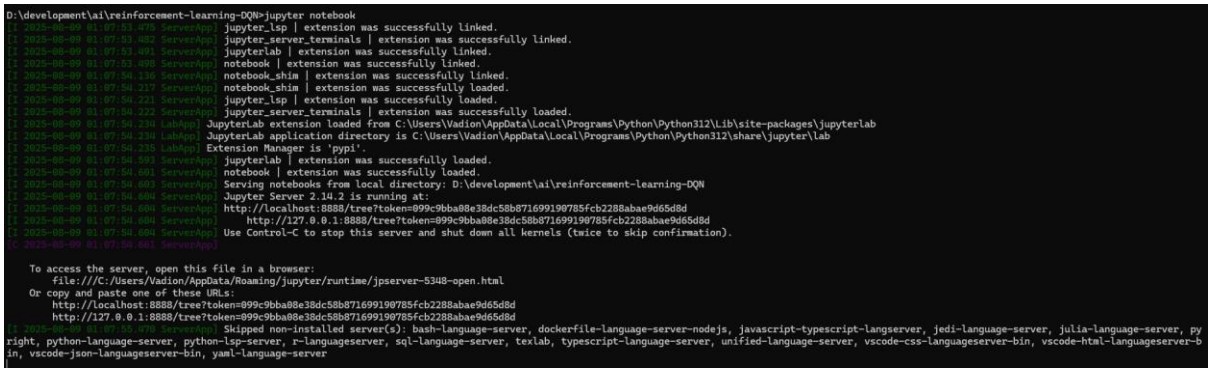

Figure 5 (a). Command line after opening jupter

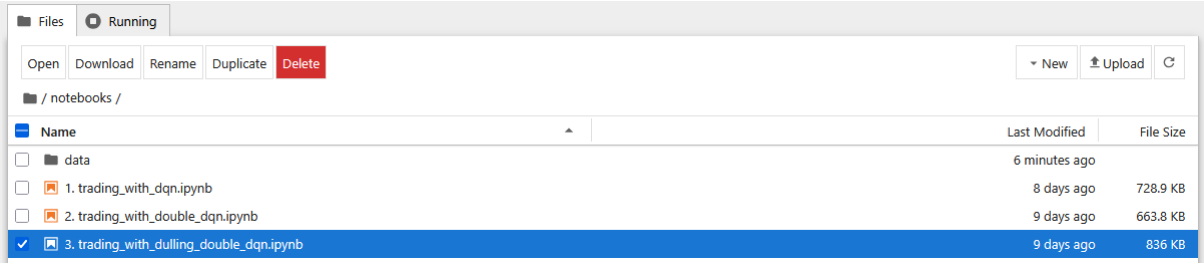

Figure 5 (b). Jupyter notebook in browser

All the three notebooks contain the similar code except the architecture of the models. All the important code cells are discussed in detail below:

4.1 Installing required libraries

All the required python libraries and packages are displayed in Figure 6.

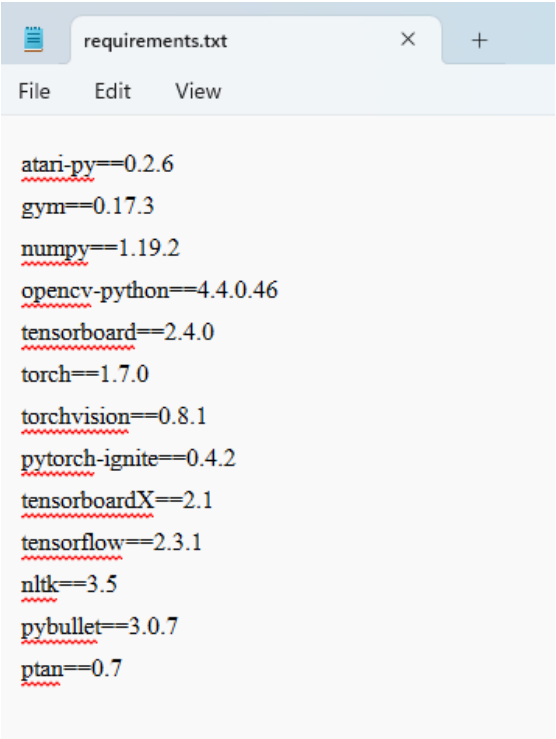

Figure 6. Required packages with their versions

Install the complete requirements file using pip. These libraries are freely available and can be easily installed from the official python site.

4.2 Importing libraries

After installing all the packages, import all the libraries used in this research.

```
[1]: import math
     import numpy as np
     import pandas as pd

[2]: import torch
     import torch.nn as nn
     import torch.nn.functional as F
     import torch.optim as optim

     C:\Users\Vadion\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.7_qbz5n2kfra8p0\LocalCache\local-packages\Python37\site-packages\tqdm\auto.py:21:
     TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
       from .autonotebook import tqdm as notebook_tqdm

[3]: import os

     import pathlib
     import argparse
     import numpy as np

[4]: import sys
     import os

[5]: project_root = os.getcwd()

[6]: sys.path.append(project_root)

[7]: import csv
     import glob

[8]: import collections

[9]: import gym
     import gym.spaces
     from gym.utils import seeding
     from gym.envs.registration import EnvSpec
```

Figure 7(a). First part of packages imported

```
[11]: import warnings
      from typing import Iterable
      from datetime import datetime, timedelta

[12]: import tensorboard

[13]: import os
      import matplotlib.pyplot as plt
      from tensorboard.backend.event_processing import event_accumulator
      from itertools import accumulate

[14]: from ignite.engine import Engine
      from ignite.contrib.handlers import tensorboard_logger as tb_logger
      from tensorboard.backend.event_processing import event_accumulator
      from ignite.engine import Engine, Events
      from ignite.metrics import RunningAverage

[15]: import ptan
      import ptan.ignite as ptan_ignite

[16]: import gym.wrappers

[17]: import matplotlib.pyplot as plt
      from tensorboard.backend.event_processing import event_accumulator
      from itertools import accumulate
```

Figure 7(b). Second part of packages imported

4.3 Defining model architectures

Before model training, model definition is required. A separate class is defined for each model.

```
[19]: class SimpleFFDQN(nn.Module):
          def __init__(self, obs_len, actions_n):
              super(SimpleFFDQN, self).__init__()
              self.net = nn.Sequential(
                  nn.Linear(obs_len, 512),
                  nn.ReLU(),
                  nn.Linear(512, 512),
                  nn.ReLU(),
                  nn.Linear(512, actions_n)
              )

          def forward(self, x):
              return self.net(x)
```

```
[20]: class DQNConv1D(nn.Module):
          def __init__(self, shape, actions_n):
              super(DQNConv1D, self).__init__()

              self.conv = nn.Sequential(
                  nn.Conv1d(shape[0], 128, 5),
                  nn.ReLU(),
                  nn.Conv1d(128, 128, 5),
                  nn.ReLU(),
              )

              out_size = self._get_conv_out(shape)

              self.fc = nn.Sequential(
                  nn.Linear(out_size, 512),
                  nn.ReLU(),
                  nn.Linear(512, actions_n)
              )

          def _get_conv_out(self, shape):
              o = self.conv(torch.zeros(1, *shape))
              return int(np.prod(o.size()))

          def forward(self, x):
              conv_out = self.conv(x).view(x.size()[0], -1)
              return self.fc(conv_out)
```

Figure 8. Defining reinforcement learning model architectures

4.4 Loading the dataset

After the model architecture is defined the next steps are to load the data from csvs and define an environment using Ptan.

[22]:
```
Prices_p = collections.namedtuple('Prices', field_names=['open', 'high', 'low', 'close', 'volume'])
```

[23]:
```
def read_csv(file_name, sep=',', filter_data=True, fix_open_price=False):
    print("Reading", file_name)
    with open(file_name, 'rt', encoding='utf-8') as fd:
        reader = csv.reader(fd, delimiter=sep)
        h = next(reader)
        if '<OPEN>' not in h and sep == ',':
            return read_csv(file_name, ';')
        indices = [h.index(s) for s in ('<OPEN>', '<HIGH>', '<LOW>', '<CLOSE>', '<VOL>')]
        o, h, l, c, v = [], [], [], [], []
        count_out = 0
        count_filter = 0
        count_fixed = 0
        prev_vals = None
        for row in reader:
            vals = list(map(float, [row[idx] for idx in indices]))
            if filter_data and all(map(lambda v: abs(v-vals[0]) < 1e-8, vals[:-1])):
                count_filter += 1
                continue

            po, ph, pl, pc, pv = vals

            # fix open price for current bar to match close price for the previous bar
            if fix_open_price and prev_vals is not None:
                ppo, pph, ppl, ppc, ppv = prev_vals
                if abs(po - ppc) > 1e-8:
                    count_fixed += 1
                    po = ppc
                    pl = min(pl, po)
                    ph = max(ph, po)
            count_out += 1
            o.append(po)
            c.append(pc)
            h.append(ph)
            l.append(pl)
            v.append(pv)
            prev_vals = vals
    print("Read done, got %d rows, %d filtered, %d open prices adjusted" % (
        count_filter + count_out, count_filter, count_fixed))
    return Prices_p(open=np.array(o, dtype=np.float32),
                    high=np.array(h, dtype=np.float32),
                    low=np.array(l, dtype=np.float32),
                    close=np.array(c, dtype=np.float32),
                    volume=np.array(v, dtype=np.float32))
```

Figure 9. Loading the dataset

4.5 Define hyperparameters and start the training

Once all the models are configured and the dataset is loaded in memory. We define the hyperparameters for the model. Figure 10 (a) shows the definition of all hyperparameters.

```
[44]: BATCH_SIZE = 1
      BARS_COUNT = 10

      EPS_START = 1.0
      EPS_FINAL = 0.05
      EPS_STEPS = 50_000

      GAMMA = 0.95

      REPLAY_SIZE = 50_000
      # REPLAY_INITIAL = 1_000
      REPLAY_INITIAL = 20_000
      REWARD_STEPS = 3
      LEARNING_RATE = 0.005
      STATES_TO_EVALUATE = 1000

      MAX_EPISODES = 10_000

      MAX_EPISODE_STEPS = 1_000
```

Figure 10 (a). Configuring all hyperparameters

Once all the hyperparameters are configured then model training is started. Training is done for the configured number of steps. Also it can be stopped by generating a system interrupt anytime in the middle of training.

```
[202]: engine.run(batch_generator(buffer, REPLAY_INITIAL, BATCH_SIZE))

Episode 4500: reward=18, steps=6, speed=324.0 f/s, elapsed=0:00:25
Episode 4600: reward=12, steps=3, speed=324.9 f/s, elapsed=0:00:27
Episode 4700: reward=12, steps=4, speed=325.9 f/s, elapsed=0:00:29
Episode 4800: reward=34, steps=13, speed=326.8 f/s, elapsed=0:00:31
validation_v also execution
Shape of validation DF :  (100, 2)
   total_reward  episode_steps
0    881.976281            299
0    454.163797            158
0    362.142192            126
0    967.317559            324
0   1367.754720            461
10000: tst: {'episode_reward': 939.7470873536797, 'episode_steps': 313.33, 'order_profits': -0.5241115225991232, 'order_steps': 160.54}
Shape of validation DF :  (100, 2)
   total_reward  episode_steps
0    353.589948            118
0    699.441013            226
0    540.454091            181
0    781.750024            265
0    284.592653             93
10000: val: {'episode_reward': 792.9122898927927, 'episode_steps': 264.32, 'order_profits': -0.1358178228685002, 'order_steps': 138.33684210526314}
Episode 4900: reward=20, steps=7, speed=321.4 f/s, elapsed=0:00:44
Episode 5000: reward=11, steps=4, speed=322.4 f/s, elapsed=0:00:46
Episode 5100: reward=29, steps=9, speed=323.3 f/s, elapsed=0:00:49
Episode 5200: reward=9, steps=3, speed=324.3 f/s, elapsed=0:00:51
Episode 5300: reward=9, steps=3, speed=325.3 f/s, elapsed=0:00:53
```

Figure 10 (b). Running model training

4.6 Model evaluation

Once the training is complete the models, multiple scalers are plotted from the tensorboard saved while training. Figure 11 shows a graph of cumulative actions across all steps of training.

```python
for run_name in os.listdir(base_log_dir):
    run_path = os.path.join(base_log_dir, run_name)

    # Only process directories
    if not os.path.isdir(run_path):
        continue

    try:
        ea = event_accumulator.EventAccumulator(run_path)
        ea.Reload()

        scalar_tags = ea.Tags().get('scalars', [])
        if reward_tag not in scalar_tags:
            print(f"[SKIP] Tag '{reward_tag}' not found in: {run_name}")
            continue

        # Load scalar data and compute cumulative reward
        events = ea.Scalars(reward_tag)
        steps = [event.step for event in events]
        values = list(accumulate(event.value for event in events))  # Cumulative sum

        # Plot this run
        plt.plot(steps, values, label=run_name)

    except Exception as e:
        print(f"[ERROR] Problem loading {run_path}: {e}")

# Plot formatting
plt.xlabel("Episodes")
plt.ylabel("Numbers of actions")
plt.title(f"Cumulative actions Across All Runs")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

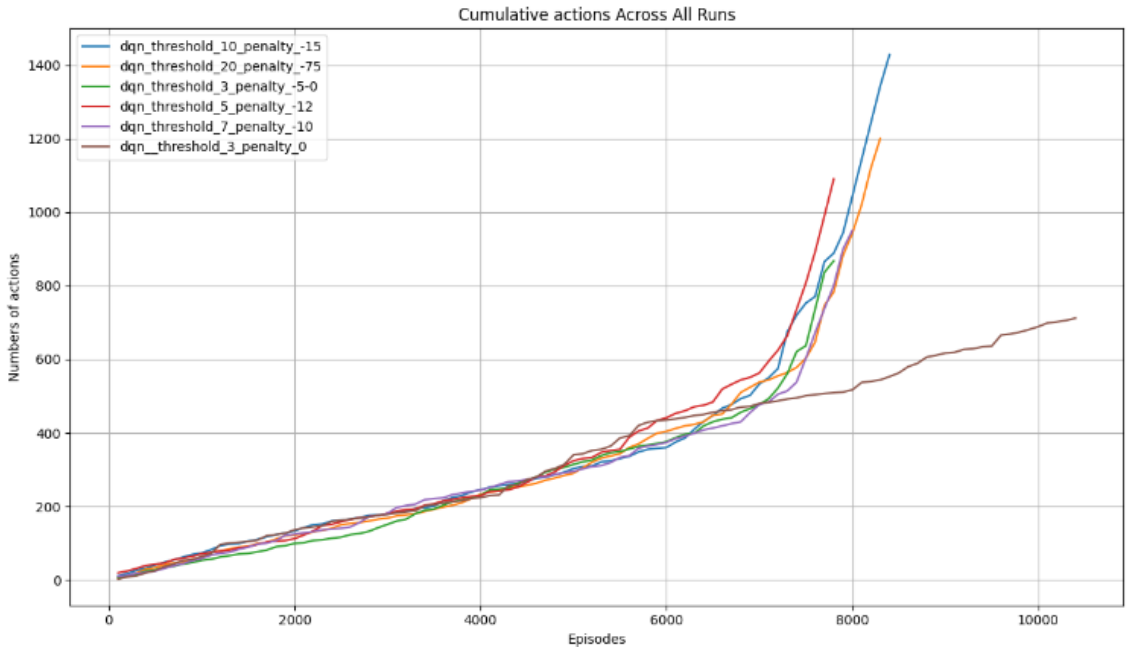

Figure 11. Plotting one of the model's evaluation metrics i.e. action frequency

Note: All the notebooks follow this exact same pattern and can be run similarly by following the steps mentioned above.

References

[1] J. E. Moody and M. Saffell, "Reinforcement Learning for Trading".

[2] S. Sarkar, “Quantitative Trading using Deep Q Learning,” Int. J. Res. Appl. Sci. Eng. Technol., vol. 11, no. 4, pp. 731–738, Apr. 2023, doi: 10.22214/ijraset.2023.50170.

[3] B. Jin, “A Mean-VaR Based Deep Reinforcement Learning Framework for Practical Algorithmic Trading,” IEEE Access, vol. 11, pp. 28920–28933, 2023, doi: 10.1109/access.2023.3259108.

[4] Y. Huang, C. Zhou, L. Zhang, and X. Lu, “A Self-Rewarding Mechanism in Deep Reinforcement Learning for Trading Strategy Optimization,” Mathematics, vol. 12, no. 24, p. 4020, Dec. 2024, doi: 10.3390/math12244020.