

Set up environment

```
In [48]: import gymnasium as gym
import numpy as np
import pandas as pd
import torch
```

数据准备

```
In [49]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

```
Out[49]: device(type='cuda')
```

```
In [50]: tech_daily = pd.read_csv(r"data\科技股票.csv")
tech_daily.set_index('date', inplace=True)
tech_daily.columns=['AAPL', 'GOOG', 'MSFT']
tech_daily
```

```
Out[50]:
```

	AAPL	GOOG	MSFT
--	------	------	------

date			
2022-01-03	178.270326	144.088458	324.504611
2022-01-04	176.007789	143.434930	318.940290
2022-01-05	171.326001	136.717897	306.696845
2022-01-06	168.465997	136.616093	304.273361
2022-01-07	168.632504	136.073308	304.428464
...
2025-11-03	268.789439	284.120000	517.030000
2025-11-04	269.778480	278.060000	514.330000
2025-11-05	269.878383	284.750000	507.160000
2025-11-06	269.508741	285.340000	497.100000
2025-11-07	268.210000	279.700000	496.820000

967 rows × 3 columns

```
In [51]: debt=pd.read_csv(r"data\无风险.csv",encoding='gbk')
debt.set_index('date', inplace=True)
debt.columns=['US_debt']
debt
```

Out[51]:

US_debt	
date	
2022-01-03	1.63
2022-01-04	1.66
2022-01-05	1.71
2022-01-06	1.73
2022-01-07	1.76
...	...
2025-11-03	4.13
2025-11-04	4.10
2025-11-05	4.17
2025-11-06	4.11
2025-11-07	4.11

963 rows × 1 columns

```
In [52]: tmp=pd.read_csv(r"data\指数和贵金属.csv", encoding='gbk')
tmp.columns=['date', 'SP500', 'Gold']
tmp.set_index('date', inplace=True)
tmp
```

Out[52]:

SP500 Gold		
date		
2022-01-03	4796.56	1801.3
2022-01-04	4793.54	1814.9
2022-01-05	4700.58	1810.6
2022-01-06	4696.05	1790.9
2022-01-07	4677.03	1796.5
...
2025-11-03	6851.97	4013.7
2025-11-04	6771.55	3941.3
2025-11-05	6796.29	3990.4
2025-11-06	6720.32	3984.8
2025-11-07	6728.80	4007.8

1001 rows × 2 columns

```
In [53]: df=pd.merge(tech_daily,debt,how='left',on='date')
df=pd.merge(df,tmp,how='left',on='date')
df['date']=pd.to_datetime(df.index)
```

```
df.set_index('date', inplace=True)
df[df.isnull().values == True]
```

Out[53]:

	AAPL	GOOG	MSFT	US_debt	SP500	Gold
date						
2022-10-10	138.103983	98.039090	223.685532	NaN	3612.39	1675.7
2022-11-11	147.475157	96.072547	241.112025	NaN	3992.93	1774.2
2023-10-09	177.082149	138.551849	324.923430	NaN	4335.66	1875.0
2024-10-14	230.005541	165.625113	416.016745	NaN	5859.85	2665.8
2024-11-11	223.220427	181.177047	414.895165	NaN	6001.35	2626.1
2025-10-13	247.420154	244.640000	514.050000	NaN	6654.72	4130.0

In [54]: `df.interpolate(method='time', inplace=True)`
df

Out[54]:

	AAPL	GOOG	MSFT	US_debt	SP500	Gold
date						
2022-01-03	178.270326	144.088458	324.504611	1.63	4796.56	1801.3
2022-01-04	176.007789	143.434930	318.940290	1.66	4793.54	1814.9
2022-01-05	171.326001	136.717897	306.696845	1.71	4700.58	1810.6
2022-01-06	168.465997	136.616093	304.273361	1.73	4696.05	1790.9
2022-01-07	168.632504	136.073308	304.428464	1.76	4677.03	1796.5
...
2025-11-03	268.789439	284.120000	517.030000	4.13	6851.97	4013.7
2025-11-04	269.778480	278.060000	514.330000	4.10	6771.55	3941.3
2025-11-05	269.878383	284.750000	507.160000	4.17	6796.29	3990.4
2025-11-06	269.508741	285.340000	497.100000	4.11	6720.32	3984.8
2025-11-07	268.210000	279.700000	496.820000	4.11	6728.80	4007.8

967 rows × 6 columns

```
In [ ]: class PortfolioOptimizationEnv(gym.Env):
    def __init__(self, tickers, window_size, start_date, end_date,
                 initial_balance, seed=None):
        super().__init__()

        # Initialize the environment's properties
        self.tickers = tickers
        self.window_size = window_size
        self.initial_balance = initial_balance

        # Download historical stock data
        self.data = self.get_data(tickers, start_date, end_date)
```

```

# Define the action and observation spaces
self.action_space = gym.spaces.Box(low=0, high=1, shape=(len(tickers),))
self.observation_space = gym.spaces.Box(low=0, high=np.inf,
                                         shape=(window_size, len(tickers)))

# Set the random seed for reproducibility
if seed is not None:
    np.random.seed(seed)
    self.action_space.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed_all(seed)

def get_data(self, tickers, start_date, end_date):
    # Download historical stock data using yfinance
    # data = yf.download(tickers, start=start_date, end=end_date)['Close']
    data=df.copy()
    data= data.loc[start_date:end_date, tickers]
    return data.dropna()

def reset(self, seed=None):
    # Reset the environment to its initial state
    self.balance = self.initial_balance
    self.current_step = self.window_size

    # Return the initial observation (historical stock prices)
    obs = self.data.iloc[self.current_step - self.window_size:self.current_s
    info={"balance": self.balance}
    return obs.reshape(self.observation_space.shape), info

def step(self, action):
    # Normalize the action values (portfolio weights) so they sum to one
    action = action / (np.sum(action) + 1e-8) # where 1e-8 is a small const

    # Calculate the previous balance and asset returns
    prev_balance = self.balance
    asset_prices = self.data.iloc[self.current_step].values
    asset_returns = asset_prices / self.data.iloc[self.current_step - 1].val

    # Update the portfolio balance based on the weighted asset returns
    self.balance = self.balance * (1 + np.sum(asset_returns * action))

    # Calculate the reward as the logarithmic return of the portfolio
    reward = np.log(self.balance / prev_balance)

    # Update the current step and check if the episode is done
    self.current_step += 1
    done = self.current_step == len(self.data) - 1

    # Generate the next observation (historical stock prices)
    obs = self.data.iloc[self.current_step - self.window_size:self.current_s

    terminated = bool(done)
    truncated = False

    # Include the current balance in the info dictionary
    info = {'balance': self.balance}

    return obs.reshape(self.observation_space.shape), reward, terminated, tr

```

```
In [56]: tickers = df.columns.tolist()
window_size = 30
start_date = '2022-01-01'
end_date = '2025-09-01'
initial_balance = 10000
seed = 8

# Initialize the environment
env = PortfolioOptimizationEnv(
    tickers,
    window_size,
    start_date,
    end_date,
    initial_balance,
    seed)

# Get the initial state
state = env.reset(seed=seed)
# Sample and execute a random action
action = env.action_space.sample()
next_state, reward, terminated, truncated, info = env.step(action)
done = bool(terminated or truncated)
# print(f"State: {state}")
print(f"Action: {action}")
# print(f"Next state: {next_state}")
print(f"Reward: {reward}")
print(f"Balance: {info['balance']}")
print(f"Done: {done}")
```

Action: [0.32697228 0.98727685 0.31871083 0.78854895 0.86989653 0.39108482]
 Reward: 0.016035833305975484
 Balance: 10161.65097308565
 Done: False

Training DRL agent

```
In [57]: # ===== Train PPO + plot training profile + plot cumulative wealth + print metrics
# Prereqs (run once in your env if needed):
# pip install "stable-baselines3>=2.3.0" "shimmy>=2.0" matplotlib pandas

from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.monitor import Monitor
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os

# ----- 1) Build vectorized env (wrap with Monitor to log training reward
log_dir = "./sb3_logs"
os.makedirs(log_dir, exist_ok=True)
monitor_path = os.path.join(log_dir, "monitor.csv")

def make_env():
    env_ = PortfolioOptimizationEnv(
        tickers=tickers,
        window_size=window_size,
        start_date=start_date,
```

```

        end_date=end_date,
        initial_balance=initial_balance,
        seed=seed,
    )
    # Log episode reward/length; include "balance" from info if you want (option
    env_ = Monitor(env_, filename=monitor_path) # single-env -> single monitor
    return env_

vec_env = DummyVecEnv([make_env])

# ----- 2) Train PPO -----
model = PPO(
    "MlpPolicy", vec_env, verbose=1, device=device
) # PPO (近端策略优化): 适合连续动作空间, 训练稳定, 是投资组合优化的常用算法
model.learn(total_timesteps=20000) # increase as needed
model.save("ppo_portfolio_optimization")

# ----- 3) Load training profile (episode rewards) and plot -----
# SB3 Monitor CSV starts with commented metadata lines beginning with '#'
train_df = pd.read_csv(monitor_path, comment="#")
# Columns typically: r (ep reward), l (ep length), t (time)
# Make a simple moving average of episode rewards for a smooth training curve
if len(train_df) > 0:
    train_df["ep"] = np.arange(1, len(train_df) + 1)
    train_df["reward_smooth"] = train_df["r"].rolling(window=max(5, len(train_df)

    plt.figure(figsize=(8, 4))
    plt.plot(train_df["ep"], train_df["r"], alpha=0.3, label="Episode reward")
    plt.plot(train_df["ep"], train_df["reward_smooth"], label="Smoothed")
    plt.xlabel("Episode")
    plt.ylabel("Episode reward")
    plt.title("Training profile (episode rewards)")
    plt.legend()
    plt.tight_layout()
    plt.show()
else:
    print("No episodes logged in monitor file (did training terminate too early?)

# ----- 4) Roll out a full episode deterministically and record balances
obs = vec_env.reset()
dones = [False]
balances = []
rewards = []
weights_seq = []

while not dones[0]:
    action, _ = model.predict(obs, deterministic=True)
    obs, reward, dones, infos = vec_env.step(action)
    rewards.append(float(reward[0]))
    balances.append(float(infos[0].get("balance", np.nan)))
    weights_seq.append(infos[0].get("weights", None))

# Try to build a date index for nicer plotting/annualization
try:
    # Access the underlying single env inside DummyVecEnv -> Monitor -> Portfolio
    base_env = vec_env.envs[0].env
    # For safety if nested wrappers differ:
    while hasattr(base_env, "env"):
        base_env = base_env.env
    price_index = base_env.data.index # pandas.DatetimeIndex from your env

```

```

# The first balance corresponds to the first step after initial window
# Align dates accordingly:
start_idx = window_size
end_idx = start_idx + len(balances)
dates = price_index[start_idx:end_idx]
except Exception:
    dates = pd.RangeIndex(start=1, stop=len(balances) + 1) # fallback to simple

# ----- 5) Plot cumulative wealth curve -----
plt.figure(figsize=(8, 4))
plt.plot(dates, balances)
plt.xlabel("Date" if isinstance(dates, pd.DatetimeIndex) else "Step")
plt.ylabel("Wealth")
plt.title("Cumulative wealth (deterministic policy rollout)")
plt.tight_layout()
plt.show()

# ----- 6) Compute portfolio performance metrics -----
# Step simple returns from balance path
balances_arr = np.asarray(balances, dtype=float)
rets = balances_arr[1:] / balances_arr[:-1] - 1.0 if len(balances_arr) > 1 else

# Infer periods per year from dates if possible; else default 252
def infer_ppy(idx):
    if isinstance(idx, pd.DatetimeIndex) and len(idx) >= 2:
        # Use median days between points
        deltas = np.diff(idx.view("int64")) / 1e9 / 86400.0 # ns -> days
        median_days = np.median(deltas) if len(deltas) else 1.0
        if median_days <= 0:
            return 252.0
        return 365.25 / median_days
    return 252.0

PPY = infer_ppy(dates)

def max_drawdown(equity):
    peak = np.maximum.accumulate(equity)
    dd = (equity - peak) / peak
    return float(dd.min()) if len(dd) else 0.0

if len(rets) > 0:
    total_growth = balances_arr[-1] / balances_arr[0]
    N = len(rets)
    CAGR = total_growth ** (PPY / N) - 1.0 # PPY: 每年交易天数; CAGR: 年复合增长率
    mean_r = float(np.mean(rets))
    std_r = float(np.std(rets, ddof=1)) if N > 1 else 0.0
    ann_return = (1.0 + mean_r) ** PPY - 1.0 if mean_r > -1 else np.nan # geometric
    ann_vol = std_r * np.sqrt(PPY) if std_r > 0 else 0.0
    # Sharpe (rf=0 for simplicity; replace with your risk-free rate if desired)
    rf = 0.0
    sharpe = (mean_r - rf/PPY) / std_r * np.sqrt(PPY) if std_r > 0 else np.nan
    mdd = max_drawdown(balances_arr)
    calmar = (CAGR / abs(mdd)) if mdd != 0 else np.nan

# Alternative geometric annualized return from total growth (more stable):
ann_return_geom = total_growth ** (PPY / N) - 1.0

print("\n=== Portfolio Performance (deterministic rollout) ===")
print(f"Periods per year (inferred): {PPY:.2f}")
print(f"Start wealth: {balances_arr[0]:.2f}")

```

```

print(f"End wealth:      {balances_arr[-1]:.2f}")
print(f"Total growth:   {total_growth:.6f}")
print(f"CAGR:           {CAGR:.6%}")
print(f"Ann. Return (geom from equity): {ann_return_geom:.6%}")
print(f"Ann. Vol:         {ann_vol:.6%}")
print(f"Sharpe (rf=0):     {sharpe:.4f}")
print(f"Max Drawdown:     {mdd:.2%}")
print(f"Calmar:           {calmar:.4f}")
else:
    print("Not enough steps to compute metrics (need at least 2 balances).")

# ----- 7) (Optional) Inspect Last action weights -----
if any(w is not None for w in weights_seq):
    last_w = [w for w in weights_seq if w is not None][-1]
    print("\nLast portfolio weights:", np.round(last_w, 4))

```

Using cuda device

```

d:\Develop\miniconda\envs\py310\lib\site-packages\stable_baselines3\common\on_policy_algorithm.py:150: UserWarning: You are trying to run PPO on the GPU, but it is primarily intended to run on the CPU when not using a CNN policy (you are using ActorCriticPolicy which should be a MlpPolicy). See https://github.com/DLR-RM/stable-baselines3/issues/1245 for more info. You can pass `device='cpu'` or `export CUDA_VISIBLE_DEVICES=` to force using the CPU.Note: The model will train, but the GPU utilization will be poor and the training might take longer than on CPU.
  warnings.warn(

```


rollout/	
ep_len_mean	887
ep_rew_mean	0.14
time/	
fps	360
iterations	1
time_elapsed	5
total_timesteps	2048

rollout/	
ep_len_mean	887
ep_rew_mean	0.216
time/	
fps	317
iterations	2
time_elapsed	12
total_timesteps	4096
train/	
approx_kl	0.007700341
clip_fraction	0.0781
clip_range	0.2
entropy_loss	-8.56
explained_variance	-0.642
learning_rate	0.0003
loss	0.025
n_updates	10
policy_gradient_loss	-0.00625
std	1.01
value_loss	0.00264

rollout/	
ep_len_mean	887
ep_rew_mean	0.393
time/	
fps	297
iterations	3
time_elapsed	20
total_timesteps	6144
train/	
approx_kl	0.0060810614
clip_fraction	0.0708
clip_range	0.2
entropy_loss	-8.58
explained_variance	0
learning_rate	0.0003
loss	-0.00308
n_updates	20
policy_gradient_loss	-0.00519
std	1.01
value_loss	0.00127

rollout/	
ep_len_mean	887
ep_rew_mean	0.422
time/	
fps	290

iterations	4
time_elapsed	28
total_timesteps	8192
train/	
approx_kl	0.0047936584
clip_fraction	0.0515
clip_range	0.2
entropy_loss	-8.62
explained_variance	0
learning_rate	0.0003
loss	0.00206
n_updates	30
policy_gradient_loss	-0.00387
std	1.02
value_loss	0.00116

rollout/	
ep_len_mean	887
ep_rew_mean	0.467
time/	
fps	280
iterations	5
time_elapsed	36
total_timesteps	10240
train/	
approx_kl	0.0063556572
clip_fraction	0.0571
clip_range	0.2
entropy_loss	-8.61
explained_variance	5.96e-08
learning_rate	0.0003
loss	-0.0188
n_updates	40
policy_gradient_loss	-0.00446
std	1.01
value_loss	0.00149

rollout/	
ep_len_mean	887
ep_rew_mean	0.504
time/	
fps	271
iterations	6
time_elapsed	45
total_timesteps	12288
train/	
approx_kl	0.0065818965
clip_fraction	0.0518
clip_range	0.2
entropy_loss	-8.57
explained_variance	0
learning_rate	0.0003
loss	0.0029
n_updates	50
policy_gradient_loss	-0.00436
std	1.01
value_loss	0.00103

rollout/		
ep_len_mean	887	
ep_rew_mean	0.495	
time/		
fps	266	
iterations	7	
time_elapsed	53	
total_timesteps	14336	
train/		
approx_kl	0.007141088	
clip_fraction	0.0632	
clip_range	0.2	
entropy_loss	-8.55	
explained_variance	0	
learning_rate	0.0003	
loss	-0.0122	
n_updates	60	
policy_gradient_loss	-0.00503	
std	1.01	
value_loss	0.00106	

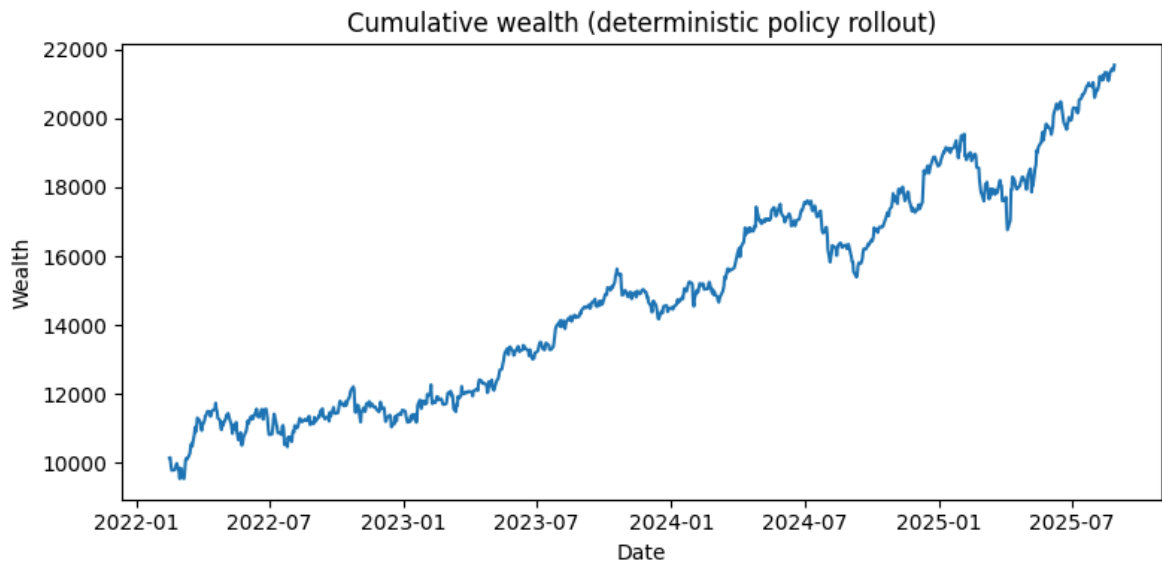
rollout/		
ep_len_mean	887	
ep_rew_mean	0.505	
time/		
fps	262	
iterations	8	
time_elapsed	62	
total_timesteps	16384	
train/		
approx_kl	0.004275806	
clip_fraction	0.0264	
clip_range	0.2	
entropy_loss	-8.58	
explained_variance	1.19e-07	
learning_rate	0.0003	
loss	-0.0117	
n_updates	70	
policy_gradient_loss	-0.00204	
std	1.02	
value_loss	0.0012	

rollout/		
ep_len_mean	887	
ep_rew_mean	0.519	
time/		
fps	260	
iterations	9	
time_elapsed	70	
total_timesteps	18432	
train/		
approx_kl	0.005100432	
clip_fraction	0.0465	
clip_range	0.2	
entropy_loss	-8.61	
explained_variance	0	
learning_rate	0.0003	

loss	-0.0168
n_updates	80
policy_gradient_loss	-0.00221
std	1.01
value_loss	0.00107

rollout/	
ep_len_mean	887
ep_rew_mean	0.504
time/	
fps	259
iterations	10
time_elapsed	78
total_timesteps	20480
train/	
approx_kl	0.006563508
clip_fraction	0.0539
clip_range	0.2
entropy_loss	-8.59
explained_variance	0
learning_rate	0.0003
loss	0.0262
n_updates	90
policy_gradient_loss	-0.00296
std	1.01
value_loss	0.00115





```
=== Portfolio Performance (deterministic rollout) ===  
Periods per year (inferred): 365.25  
Start wealth: 10145.55  
End wealth: 21543.04  
Total growth: 2.123399  
CAGR: 36.400972%  
Ann. Return (geom from equity): 36.400972%  
Ann. Vol: 19.821596%  
Sharpe (rf=0): 1.6658  
Max Drawdown: -14.23%  
Calmar: 2.5587
```

修改部分

添加了科技股票、国债、标普500和黄金等数据