Data Science Collective

Member-only story

Building Smarter Portfolios with Dynamic Programming and Reinforcement Learning

Shenggang Li · Subscribed

Published in Data Science Collective · 24 min read · Mar 30, 2025

234 1

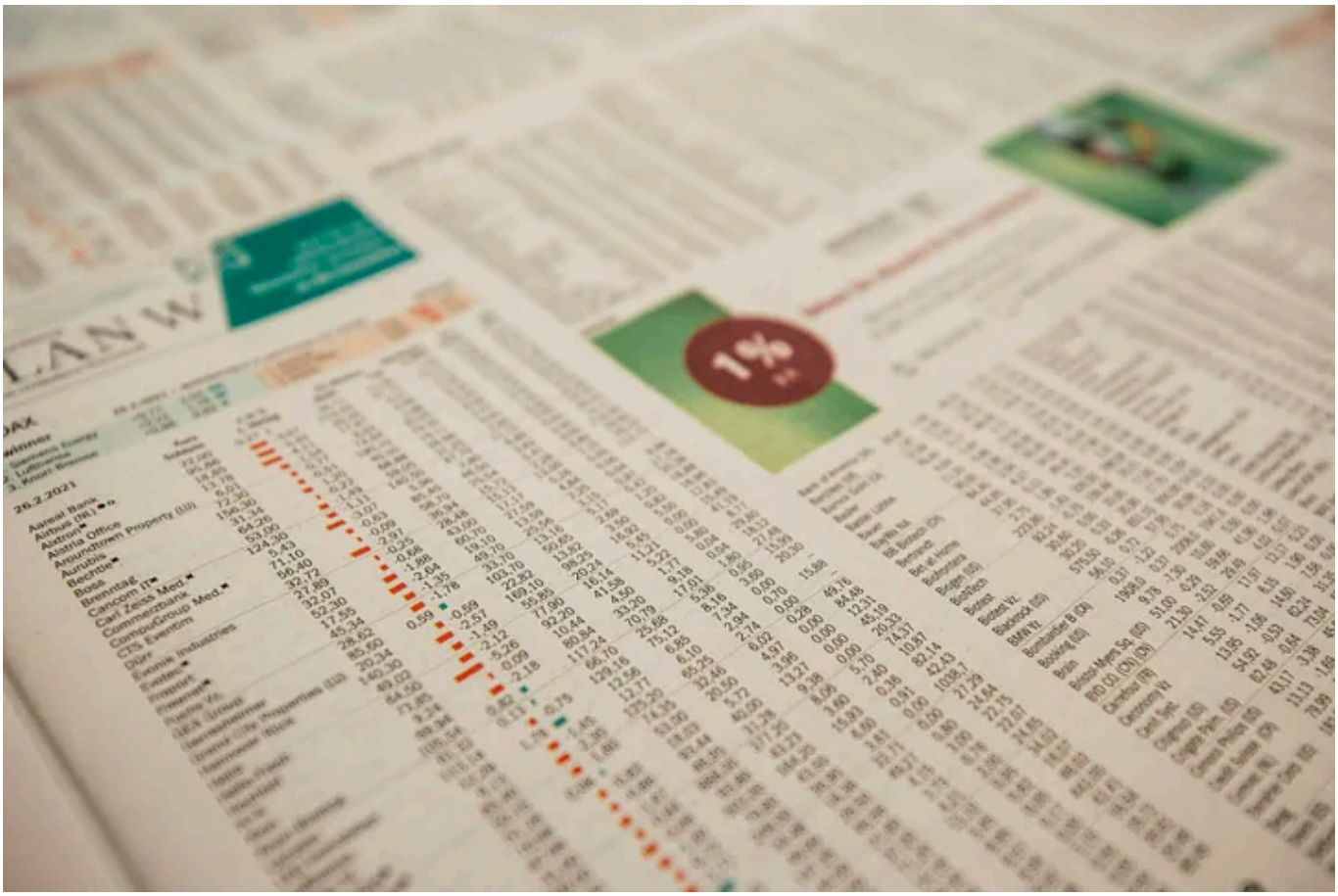


Photo by Markus Spiske on Unsplash

Introduction

I present a novel approach that integrates reinforcement learning and dynamic programming to build a state-driven portfolio using real market data. This framework uses price ratios and moving averages — such as comparing a stock's price to the combined value of *SPY* and *QQQ* — to generate trading signals. For instance, if Stock A's short-term moving average falls below its long-term average by a preset threshold, it triggers an investment of a fixed *$1000*, after which the position is held.

Theoretically, an *RL* policy network is designed to learn optimal decision-making via policy gradients, effectively guiding portfolio selection. Practically, the strategy is tested over a substantial historical period,

simulating real trading scenarios, which makes it relevant for both academic research and real-world portfolio management.

The process is broken down into clear, step-by-step stages — from data preparation and signal generation to policy training — making it straightforward to understand and implement.

Finally, this method provides a practical, efficient framework for navigating complex market dynamics with state-based decisions — offering valuable insights for researchers and actionable strategies for investors.

Rationale for the Proposed RL-Based Dynamic Portfolio Algorithm

The proposed reinforcement learning (*RL*) algorithm integrates mathematical optimization with practical investment strategy, significantly enhancing traditional portfolio methods by dynamically managing both stock selection and investment timing.

Consider a portfolio consisting of N carefully selected "good" stocks indexed by $i=1,2,\ldots, N$, over a discrete investment horizon segmented into intervals $t=1,2,\ldots,T$.

Define $r_{i, t}$ as the return of stock i at time t, and $s_{i, t}$ as the state variable representing stock i's market indicators (e.g., price ratios, moving averages). The *RL*-driven portfolio aims to dynamically optimize expected returns adjusted by inverse risk (variance) through the following objective function:

$$\max \quad E\left[\sum_{t=1}^{T}\sum_{i=1}^{N} w_{i,t} r_{i,t}\right] - \lambda \mathrm{Var}\left(\sum_{t=1}^{T}\sum_{i=1}^{N} w_{i,t} r_{i,t}\right)$$

Subject to constraints:

Budget Constraint:

$$\sum_{i=1}^{N} w_{i,t} \leq B_t, \quad \forall t$$

where *w_{i, t}* is the investment in stock *i* at time *t*, and *B_t* is the total budget available.

Non-negativity Constraint:

$$w_{i,t} \geq 0, \quad \forall i, t$$

State-based RL Investment Constraint:

$$w_{i,t} = f(s_{i,t}; \theta)$$

where the function $f(\cdot)$, parameterized by θ, is learned via *RL* to determine investment allocations based on market states.

Under key assumptions — that each stock exhibits positive long-run returns $E[r_{i,t}]>0$ and that state indicators $s_{i,t}$ effectively identify optimal investment timing (temporary price dips) — this approach strategically diversifies across both assets and time. Unlike traditional static portfolios or ETFs, this dynamic method minimizes the temporal variance of returns by capitalizing on state-specific optimal purchase points:

$$\mathrm{Var}\left(\frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{N} w_{i,t} r_{i,t}\right) \leq \mathrm{Var}\left(\frac{1}{N}\sum_{i=1}^{N} w_i r_i\right)$$

Here, the left-hand side represents the dynamically-managed *RL* portfolio variance, while the right-hand side denotes the variance from conventional static diversification. Intuitively, this mathematical formulation and RL algorithm leverage market inefficiencies by systematically investing during temporary weaknesses in high-quality stocks, significantly reducing overall variance and enhancing risk-adjusted returns.

In short, the *RL* model implicitly solves a sequential optimization problem, selecting each stock at its relatively favorable state. Probabilistically, this process maximizes the likelihood of purchasing at optimal points (state-conditioned buy decisions):

$$P\left(\text{OptimalBuyTime}\right) = \prod_{t=1}^{T} P(a_t|S_t, \theta^*)$$

where $\theta^{\wedge} *$ is the optimal learned parameter set from the *RL* policy. Ultimately, this method achieves superior diversification and risk-adjusted performance by systematically exploiting temporal variability, a capability traditional one-point portfolio optimization fundamentally lacks.

An Integrated Approach for RL-Driven Portfolio Construction

I will introduce a unified framework for building investment portfolios using reinforcement learning (*RL*). While I'll use Policy Gradient training as the main example, this approach is very flexible. We can apply it to many other *RL* methods too — like actor-critic algorithms, *PPO, GRPO*, or even simpler ones like naïve policy inference (sometimes called inspired *RL*), which can work surprisingly well in some cases.

Overview of the Integrated Approach

The key idea behind integrating reinforcement learning (*RL*) with dynamic programming (*DP*) is to use market states — defined by features such as price ratios and moving averages — to determine which stock to purchase. Mathematically, each stock *j* at time *t* is represented by a ratio:

$$\text{ratio}_j(t) = \frac{P_j(t)}{P_{\text{SPY}}(t) + P_{\text{QQQ}}(t)}$$

where *P_j(t)* is the price of stock *j*, and *P_SPY(t)* and *P_QQQ(t)* are the prices of the benchmark indices.

This ratio is further smoothed using a short-term moving average *MA_k* and a long-term moving average *MA_n*, computed as:

$$MA_k(\text{ratio}_j)(t) = \frac{1}{k} \sum_{i=t-k+1}^{t} \text{ratio}_j(i) \quad \text{and} \quad MA_n(\text{ratio}_j)(t) = \frac{1}{n} \sum_{i=t-n+1}^{t} \text{ratio}_j(i)$$

A binary signal *I_j(t)* is then generated by comparing the short-term and long-term moving averages using a threshold δ:

$$I_j(t) = \begin{cases} 1, & \text{if } MA_k(\text{ratio}_j)(t) < MA_n(\text{ratio}_j)(t) - \delta \\ 0, & \text{otherwise.} \end{cases}$$

This initial stage filters stocks based on their relative performance against the benchmarks, setting the foundation for our decision-making process.

State Construction and Feature Engineering

In this framework, the state at time *t* is a comprehensive vector that encapsulates key market features for each stock. For every stock *j*, the state

vector includes the current ratio *ratio_j(t)*, the short-term moving average *MA_k(ratio_j)(t)*, and the long-term moving average *MA_n(ratio_j)(t)*.

This will produce a state representation of dimension *60×360* when considering *60* stocks. The complete state is then defined as:

$$s_t = [\text{ratio}_1(t), MA_k(\text{ratio}_1)(t), MA_n(\text{ratio}_1)(t), \ldots, \text{ratio}_{60}(t), MA_k(\text{ratio}_{60})(t), MA_n(\text{ratio}_{60})(t)]$$

The high-dimensional representation captures both short-term and long-term market dynamics and is used as input to the policy network.

Policy Network and Action Selection

The policy network, implemented as a multilayer perceptron, is responsible for mapping the state *s_t* to a probability distribution over possible actions.

The network is parameterized by θ and outputs:

$$\pi(a_t \mid s_t; \theta)$$

where the action *a_t* corresponds to selecting one of the *60* stocks or choosing “no action”. The network consists of several fully connected layers with *ReLU* activations, culminating in a softmax layer that ensures the outputs form a valid probability distribution. This softmax function is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

where *z_i* are the outputs of the final linear layer. The action space thus comprises *61* actions (*60* stocks and one no-action option). When the policy network receives the state *s_t*, it computes the probabilities for each action, from which an action is then sampled. Importantly, before sampling, the probabilities for stock actions are masked by the binary signals *I_j(t)* so that only stocks with a valid buy signal (where *I_j(t) = 1*) are considered. This ensures that the agent only chooses stocks that are deemed promising by the signal generation process.

Dynamic Programming for Return Optimization

Once an action is taken, the mechanism uses dynamic programming to evaluate the decision based on long-term returns. When a stock *j* is selected at time *t*, an investment of a fixed amount (e.g., *$1000*) is made. The reward for this action is computed as the return from the purchase day to the terminal day *T* of the episode:

$$r_t = \frac{P_j(T)}{P_j(t)} - 1$$

In dynamic programming terms, we define the value function *V(s_t)* as the maximum expected return from state *s_t* onward, and it follows the Bellman equation:

$$V(s_t) = \max_{a_t}\{r_t(a_t) + \gamma V(s_{t+1})\}$$

Since the agent holds the stock without further trading, the *DP* framework mainly evaluates the timing of the initial buy decision. The discount factor γ (often set to 1 in our case) can be adjusted if necessary to account for the time value of money.

By integrating dynamic programming, I will make sure that the strategy does not merely optimize for immediate gains but also considers the future payoff of each decision, balancing short-term signals with long-term performance.

Episode Simulation and Policy Gradient Training

The training framework operates by simulating multiple episodes within a predefined training window. Each episode spans a backtesting period (e.g., *600* days) and involves daily state evaluations by the *RL* agent. For each day *t* in the episode, the state *s_t* is generated, and the policy network produces a probability distribution over the actions. After masking with the signal *I_j(t)*, an action is sampled. If a valid stock is selected, the episode terminates with the purchase being made. The corresponding reward is then computed based on the return from the day of purchase to the terminal day of the episode. The policy gradient update is performed using the loss function defined as:

$$L(\theta) = -\sum_t \log \pi(a_t \mid s_t; \theta) \cdot r_t$$

where the sum is taken over the decision steps in the episode. This update rule reinforces actions that yield higher returns by increasing their log probabilities. By iterating over numerous episodes, the network parameters θ are progressively updated to favor strategies that consistently produce positive rewards, thereby improving the overall decision-making capability of the *RL* agent.

Example Simulation:

Consider an episode where the agent is evaluating states from day *t = 100* to *t = 700* within a training window. On each day, the state vector is constructed by concatenating the ratio, *MA_k*, and *MA_n* for every stock. The policy network processes this state and outputs a probability distribution over 61 actions. A candidate mask, derived from the binary signals *I_j(t)*, filters out stocks that do not meet the buy criteria. Suppose that on day *150*, the policy network assigns a high probability to *Stock_7*, which has a valid buy signal. The agent then selects Stock_7 and invests *$1000* at its price. Assume that *Stock_7* is trading at *$50* on day *150* and its price rises to *$60* by day *700*. The reward for this action is computed as

$$r_{150} = \frac{60}{50} - 1 = 0.2 \quad (20\% \text{ return})$$

This reward, in combination with the log probability of the chosen action, is used to compute the policy gradient for updating the network parameters. The training loop iterates over many such episodes, each time refining the policy network to favor actions that historically resulted in higher returns. This iterative process effectively balances immediate signals with long-term

return optimization, ensuring that the agent learns a robust trading strategy even in non-stationary market environments.

Parameterization and Signal Robustness

A significant strength of this method lies in its flexible parameterization. The moving average windows k and n are crucial parameters that determine the sensitivity of the signal generation process. A smaller k may lead to more reactive, albeit noisier, short-term indicators, while a larger n provides a smoother, long-term trend analysis. The threshold δ plays a critical role by ensuring that the signal only triggers when the short-term moving average is sufficiently lower than the long-term moving average. This filter helps reduce false positives in volatile market conditions. By adjusting k, n, and δ, the model can be tuned to optimize performance across different market regimes. The flexibility to experiment with these parameters also allows the incorporation of additional features or indicators, further enhancing the model's robustness. Testing various configurations through grid search or Bayesian optimization methods can lead to an optimal setup that maximizes the expected portfolio return while controlling for risk.

Summary and Practical Considerations

The mechanism starts by generating state representations based on market ratios and moving averages, which are then used by a policy network to select actions probabilistically. The decision to invest is reinforced through a dynamic programming framework that evaluates the long-term return on investment. The backbone of the approach — featuring formulas for moving averages, the *Bellman* equation, and policy gradient loss — ensures that both immediate and future returns are considered in the decision-making process. In addition, the flexibility provided by the parameterized signal generation enables the method to adapt to various market conditions and risk profiles.

This methodology is designed to be implemented with readily available financial data, such as historical prices for individual stocks and benchmarks like *SPY* and *QQQ*. The use of a fixed investment amount per trade (e.g., *$1000*) simplifies the trading logic while focusing on optimizing the timing and selection of trades. The episodic simulation framework allows for extensive backtesting. For practitioners, the actionable outputs — such as the list of buy actions and the aggregated return over the evaluation period — offer direct insights into the strategy's performance.

Code Experiment: Actor-Critic Network RL-Based Dynamic Portfolio Algorithm

In this experiment, I'll introduce an actor-critic reinforcement learning (*RL*) approach to dynamically picking stocks, aiming to boost returns and manage risks effectively. I first gather and process market data to create useful signals. Then, an actor-critic neural network decides which stocks to trade (the "actor") and evaluates their potential performance (the "critic").

To fine-tune our model, I use grid search to systematically test various hyperparameter combinations, such as moving-average windows and trade intervals. The algorithm learns through policy gradients, continually improving its trading decisions by evaluating past outcomes, allowing it to adapt to changing market conditions.

```
import itertools
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
```

```
import yfinance as yf
import random

########################################
# Data Download & Organization
########################################
def getdatabyday(tiklst, st, ed):
    download_data = pd.DataFrame([])
    for tk in tiklst:
        try:
            stock = yf.Ticker(tk)
            stock_data = stock.history(start=st, end=ed, interval='1d').reset_in
            if len(stock_data) > 0:
                stock_data['ticker'] = tk
                download_data = pd.concat([download_data, stock_data])
                print('Downloaded data for:', tk)
            else:
                print(f"No data returned for: {tk}")
        except Exception as e:
            print(f"Skipping {tk} due to error: {e}")
            continue
    # Sort
    download_data = download_data.sort_values(['ticker', 'Date'], ascending=[Tru
    download_data['Date'] = download_data['Date'].astype(str)
    cols_keep = ['ticker', 'Date', 'Open', 'High', 'Low', 'Close', 'Volume']
    download_data = download_data[cols_keep]
    tm_frame = pd.DataFrame(list(set(download_data['Date'])), columns=['Date'])
    tm_frame = tm_frame.sort_values(['Date'], ascending=False)
    tm_frame['dayseq'] = range(1, len(tm_frame) + 1)
    download_data = pd.merge(download_data, tm_frame, on=['Date'], how='inner')
    download_data = download_data.sort_values(['ticker', 'Date'], ascending=[Fal
    download_data.ffill(inplace=True)
    download_data.bfill(inplace=True)
    return download_data

def organize_data_for_rl(stock_data):
    stock_data['Date'] = pd.to_datetime(stock_data['Date'], utc=True).dt.tz_loca
    stock_data.sort_values(by='Date', inplace=True)
    wide_df = stock_data.pivot(index='Date', columns='ticker', values='Close')
    wide_df.columns = [col.upper().replace('-', '_') for col in wide_df.columns]
    wide_df.ffill(inplace=True)
    wide_df.bfill(inplace=True)
    return wide_df

########################################
# Actor-Critic Network
########################################
class ActorCriticNetwork(nn.Module):
    """
    Minimal Actor-Critic network:
```

```
        - One shared backbone
        - Actor head outputting a probability distribution over (stocks + no-actio
        - Critic head producing a single scalar state-value
    """
    def __init__(self, input_dim, output_dim):
        super(ActorCriticNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 64)
        # Actor outputs distribution over 'output_dim' discrete actions
        self.actor = nn.Linear(64, output_dim)
        # Critic outputs a single scalar value
        self.critic = nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        policy = torch.softmax(self.actor(x), dim=-1)
        value  = self.critic(x)
        return policy, value

########################################
# Feature Engineering & State
########################################
def compute_signals(DF, k, n, delta):
    df_ratios = DF.copy()
    stocks = [col for col in DF.columns if col not in ['SPY','QQQ']]
    for stock in stocks:
        df_ratios[stock] = DF[stock] / (DF['SPY'] + DF['QQQ'])
    signals = {}
    for stock in stocks:
        ma_k = df_ratios[stock].rolling(window=k).mean()
        ma_n = df_ratios[stock].rolling(window=n).mean()
        signals[stock] = (ma_k < (ma_n - delta)).astype(int)
    signals_df = pd.DataFrame(signals, index=DF.index)
    return signals_df, df_ratios

def get_state(DF, df_ratios, t, k, n):
    """
    Build a state vector from ratio, short-term MA, long-term MA for each stock
    (excluding SPY, QQQ).
    """
    stocks = [col for col in DF.columns if col not in ['SPY','QQQ']]
    state_feats = []
    for stock in stocks:
        ratio_t = df_ratios[stock].iloc[t]
        ma_k_t = df_ratios[stock].rolling(window=k).mean().iloc[t]
        ma_n_t = df_ratios[stock].rolling(window=n).mean().iloc[t]
        state_feats.extend([ratio_t, ma_k_t, ma_n_t])
    return np.array(state_feats, dtype=np.float32)
```

```
########################################
# Multi-Trade Simulation + A2C Updates
########################################
def simulate_trading(
    DF,
    signals_df,
    df_ratios,
    actor_critic_net,
    optimizer,
    start_idx,
    end_idx,
    k, n,
    train_mode=False,    # If True, do advantage-based updates
    gamma=0.99,          # discount factor
    trade_interval_m=3,
    max_weight_w=0.34,
    cooldown_R=10,
    required_distinct=4
):
    stocks = [col for col in DF.columns if col not in ['SPY','QQQ']]
    portfolio = {}
    transactions = []
    last_trade_day = -trade_interval_m
    distinct_stocks = set()

    # We'll store experiences if train_mode is True
    all_states  = []
    all_actions = []
    all_values  = []
    all_rewards = []

    for t in range(start_idx, end_idx):
        if t - last_trade_day < trade_interval_m:
            # no trade day
            continue

        # Build state
        state = get_state(DF, df_ratios, t, k, n)
        st_tensor = torch.from_numpy(state).unsqueeze(0).float()
        # forward pass
        policy, value = actor_critic_net(st_tensor)
        probs = policy.squeeze(0).detach().numpy()
        v_s   = value.item()

        # Candidate mask
        candidate_mask = np.array([signals_df[stock].iloc[t] for stock in stocks
        candidate_mask = np.concatenate([candidate_mask, np.array([1])])  # no-a
        masked_probs   = probs[:len(stocks)] * candidate_mask[:len(stocks)]
        no_action_prob = probs[len(stocks)]
        final_probs    = np.concatenate([masked_probs, np.array([no_action_prob]
```

```
            if final_probs.sum() > 0:
                final_probs = final_probs / final_probs.sum()
            else:
                final_probs = np.ones_like(final_probs) / len(final_probs)

            action = np.random.choice(len(final_probs), p=final_probs)
            if action == len(stocks):
                # no action
                if train_mode:
                    # store experience with 0 reward
                    all_states.append(state)
                    all_actions.append(action)
                    all_values.append(v_s)
                    all_rewards.append(0.0)
                continue

            chosen_stock = stocks[action]
            # cooldown
            if chosen_stock in portfolio and (t - portfolio[chosen_stock]['last_trad
                if train_mode:
                    # invalid => treat as no-action, store 0 reward
                    all_states.append(state)
                    all_actions.append(len(stocks))
                    all_values.append(v_s)
                    all_rewards.append(0.0)
                continue

            # portfolio weighting
            current_value = sum(portfolio[s]['shares'] * DF[s].iloc[t] for s in port
            new_trade_amount = 1000.0
            est_total = current_value + new_trade_amount
            if len(portfolio) >= 2:
                existing_cost = portfolio.get(chosen_stock, {}).get('cost', 0.0)
                new_cost = existing_cost + new_trade_amount
                if (new_cost / est_total) > max_weight_w:
                    if train_mode:
                        # store no action
                        all_states.append(state)
                        all_actions.append(len(stocks))
                        all_values.append(v_s)
                        all_rewards.append(0.0)
                    continue

            # valid trade => execute
            trade_price = DF[chosen_stock].iloc[t]
            shares = new_trade_amount / trade_price
            if chosen_stock in portfolio:
                portfolio[chosen_stock]['shares'] += shares
                portfolio[chosen_stock]['cost']   += new_trade_amount
                portfolio[chosen_stock]['last_trade_day'] = t
```

```
            else:
                portfolio[chosen_stock] = {'shares': shares, 'cost': new_trade_amoun
                distinct_stocks.add(chosen_stock)

            transactions.append({
                'Date': DF.index[t],
                'Stock': chosen_stock,
                'Trade_Price': trade_price,
                'Shares_Bought': shares,
                'Trade_Amount': new_trade_amount,
                'Cumulative_Cost': portfolio[chosen_stock]['cost'],
                'Avg_Cost': portfolio[chosen_stock]['cost'] / portfolio[chosen_stock
                'Day_Index': t
            })
            last_trade_day = t

            # if train_mode => define immediate reward for the buy action
            # we do a single-step reward approach by looking at next day's price (if
            reward = 0.0
            next_idx = t+1 if (t+1 < end_idx) else t
            future_price = DF[chosen_stock].iloc[next_idx]
            buy_price    = trade_price
            reward       = (future_price / buy_price) - 1.0  # 1-step lookahead

            if train_mode:
                all_states.append(state)
                all_actions.append(action)
                all_values.append(v_s)
                all_rewards.append(reward)

        # If train_mode => do an advantage-based update (A2C style)
        if train_mode and len(all_states) > 0:
            # We'll do single-step returns: G_i = r_i + gamma * V(s_{i+1})
            # If i+1 is out of range or no next state, we set V(s_{i+1})=0
            returns = []
            for i in range(len(all_states)):
                r_i = all_rewards[i]
                if i < len(all_states) - 1:
                    next_val = all_values[i+1]   # approximate for next state
                else:
                    next_val = 0.0
                G = r_i + gamma * next_val
                returns.append(G)

            returns_t = torch.tensor(returns, dtype=torch.float32)
            values_t  = torch.tensor(all_values, dtype=torch.float32)
            advantages= returns_t - values_t

            # Now do a typical actor-critic update
            total_loss = 0.0
```

```
            for i in range(len(all_states)):
                s_i = torch.from_numpy(all_states[i]).unsqueeze(0).float()
                pol, val = actor_critic_net(s_i)
                action_i = all_actions[i]
                advantage_i = advantages[i]

                # Actor loss = -log(pi(a_i|s_i)) * advantage_i
                log_prob = torch.log(pol[0, action_i] + 1e-8)
                actor_loss  = -log_prob * advantage_i
                # Critic loss
                critic_loss = (returns_t[i] - val) ** 2
                total_loss += (actor_loss + critic_loss)

            optimizer.zero_grad()
            total_loss.backward()
            optimizer.step()

        transactions_df = pd.DataFrame(transactions)
        final_num_distinct = len(distinct_stocks)
        return transactions_df, portfolio, final_num_distinct

    ########################################
    # Pipeline w/ A2C in Training
    ########################################
    def full_pipeline(
        DF,
        actor_critic_net,
        optimizer,
        train_end_idx,
        k=5, n=20, delta=0.005,
        trade_interval_m=3,
        max_weight_w=0.34,
        cooldown_R=10,
        required_distinct=4
    ):
        signals_df, df_ratios = compute_signals(DF, k, n, delta)

        # Training simulation WITH advantage updates
        train_tx, train_port, train_distinct = simulate_trading(
            DF,
            signals_df,
            df_ratios,
            actor_critic_net,
            optimizer,
            start_idx=0,
            end_idx=train_end_idx,
            k=k,
            n=n,
            train_mode=True,  # This triggers the advantage updates
            trade_interval_m=trade_interval_m,
```

```
    max_weight_w=max_weight_w,
    cooldown_R=cooldown_R,
    required_distinct=required_distinct
)

# Calculate training portfolio return
train_value = 0.0
for stk in train_port:
    train_value += train_port[stk]['shares'] * DF[stk].iloc[train_end_idx -
train_invested = sum(train_port[s]['cost'] for s in train_port)
train_ret = (train_value / train_invested - 1.0) if train_invested > 0 else

train_spy_ret = None
train_qqq_ret = None
if 'SPY' in DF.columns:
    spy_start = DF['SPY'].iloc[0]
    spy_end   = DF['SPY'].iloc[train_end_idx - 1]
    train_spy_ret = (spy_end / spy_start) - 1.0
if 'QQQ' in DF.columns:
    qqq_start = DF['QQQ'].iloc[0]
    qqq_end   = DF['QQQ'].iloc[train_end_idx - 1]
    train_qqq_ret = (qqq_end / qqq_start) - 1.0

# Validation (inference mode, no updates)
val_tx, val_port, val_distinct = simulate_trading(
    DF,
    signals_df,
    df_ratios,
    actor_critic_net,
    optimizer,
    start_idx=train_end_idx,
    end_idx=len(DF),
    k=k, n=n,
    train_mode=False,  # No learning updates
    trade_interval_m=trade_interval_m,
    max_weight_w=max_weight_w,
    cooldown_R=cooldown_R,
    required_distinct=required_distinct
)

val_value = 0.0
for stk in val_port:
    val_value += val_port[stk]['shares'] * DF[stk].iloc[-1]
val_invested = sum(val_port[s]['cost'] for s in val_port)
val_ret = (val_value / val_invested - 1.0) if val_invested > 0 else 0.0

spy_val_ret = None
qqq_val_ret = None
if 'SPY' in DF.columns:
    spy_val_ret = (DF['SPY'].iloc[-1] / DF['SPY'].iloc[train_end_idx]) - 1.0
```

```
        if 'QQQ' in DF.columns:
            qqq_val_ret = (DF['QQQ'].iloc[-1] / DF['QQQ'].iloc[train_end_idx]) - 1.0

        return {
            'train_tx': train_tx,
            'train_ret': train_ret,
            'train_spy_ret': train_spy_ret,
            'train_qqq_ret': train_qqq_ret,
            'val_tx': val_tx,
            'val_ret': val_ret,
            'spy_val_ret': spy_val_ret,
            'qqq_val_ret': qqq_val_ret
        }

########################################
# Random Seeding
########################################
seed_val = 99
random.seed(seed_val)
np.random.seed(seed_val)
torch.manual_seed(seed_val)
torch.cuda.manual_seed_all(seed_val)

########################################
# Parameter Search with Actor-Critic
########################################
if __name__ == "__main__":
    # Tickers, date range
    tickers = [
        "SPY", "QQQ",
        "AAPL", "MSFT", "META", "NVDA", "TSLA",
        "AMZN", "PANW", "CRWD", "IYW", "ISRG"
    ]
    start_date = "2021-01-01"
    end_date   = "2024-12-31"
    cutoff_date= "2024-02-01"

    print("Downloading data...")
    stock_data = getdatabyday(tickers, start_date, end_date)
    DF = organize_data_for_rl(stock_data)

    DF.index = pd.to_datetime(DF.index, utc=True).tz_localize(None)
    DF.sort_index(inplace=True)

    all_dates = DF.index
    train_end_idx = np.searchsorted(all_dates, pd.Timestamp(cutoff_date))
    print(f"train_end_idx = {train_end_idx}, date = {all_dates[train_end_idx]}")

    num_cols = len(DF.columns)
    input_dim = 3*(num_cols - 2)
```

```
output_dim= (num_cols - 2)+1

# We define parameter grids
k_values       = [5, 8, 10]
n_values       = [20, 25, 30]
delta_values   = [0.005, 0.01, 0.02]
trade_intervals= [3, 4, 5]
cooldown_vals  = [10, 15]
weight_limits  = [0.25, 0.34]
required_min   = [4, 5]

results = []
for k_v, n_v, d_v, ti_v, cd_v, w_v, req_v in itertools.product(
    k_values, n_values, delta_values, trade_intervals, cooldown_vals, weight
):
    # Create a fresh ActorCritic network & optimizer for each parameter set
    actor_critic_net = ActorCriticNetwork(input_dim, output_dim)
    optimizer = optim.Adam(actor_critic_net.parameters(), lr=0.0006)

    # Run the pipeline with advantage-based updates in training
    outcome = full_pipeline(
        DF,
        actor_critic_net,
        optimizer,
        train_end_idx,
        k=k_v,
        n=n_v,
        delta=d_v,
        trade_interval_m=ti_v,
        max_weight_w=w_v,
        cooldown_R=cd_v,
        required_distinct=req_v
    )
    train_ret  = outcome['train_ret']
    train_spy  = outcome['train_spy_ret']
    train_qqq  = outcome['train_qqq_ret']
    val_ret    = outcome['val_ret']
    spy_val_ret= outcome['spy_val_ret']
    qqq_val_ret= outcome['qqq_val_ret']

    results.append({
        'k': k_v,
        'n': n_v,
        'delta': d_v,
        'trade_interval': ti_v,
        'cooldown_R': cd_v,
        'max_weight': w_v,
        'required_distinct': req_v,
        'train_return': train_ret,
        'train_spy': train_spy,
```

```
            'train_qqq': train_qqq,
            'val_return': val_ret,
            'val_spy': spy_val_ret,
            'val_qqq': qqq_val_ret
        })

    df_results = pd.DataFrame(results)
    df_results['excess_return_qqq_val'] = df_results['val_return'] - df_results[
    df_results.sort_values('excess_return_qqq_val', ascending=False, inplace=Tru

    print("\n=== TOP 10 PARAM SETS (ValReturn vs QQQ) ===")
    print(df_results.head(10).to_string(index=False))
```

Here are the results:

```
=== TOP 10 PARAM SETS (ValReturn vs QQQ) ===
 k  n  delta  trade_interval  cooldown_R  max_weight  required_distinct  train_r
 5 20  0.005              4          15        0.25                  5      0.0
10 20  0.010              4          15        0.25                  5      0.2
 8 20  0.005              5          10        0.25                  4      0.1
 5 25  0.010              3          15        0.25                  5      0.2
 8 25  0.020              3          10        0.25                  4      0.4
10 20  0.010              3          10        0.25                  5      0.4
10 25  0.020              5          10        0.25                  4      0.7
 8 30  0.005              4          10        0.25                  4      0.4
 5 25  0.010              4          15        0.25                  5      0.1
 8 30  0.010              4          15        0.25                  4      0.3
```

Summary of Results

The top-performing parameter sets achieved an excess validation return of approximately *19.55%* over *QQQ*. Among these, two configurations stood out clearly:

First configuration:

k=5, n=20, delta=0.005, trade_interval=4, cooldown_R=15, max_weight=0.25,

required_distinct=5

- Train Return: ~5.41%
- Val Return: ~42.49%

Second configuration *(recommended)*:
k=10, n=20, delta=0.01, trade_interval=4, cooldown_R=15, max_weight=0.25, required_distinct=5

- **Train Return**: ~23.22% (much higher consistency)
- **Val Return**: ~42.49%

Even though both configurations offer identical validation returns, the second one demonstrates significantly higher performance during training, indicating greater robustness.

Other parameter sets showed slightly lower outperformance (~17–19%), confirming the strength of these top two options. For optimal consistency and highest overall performance, the second configuration is recommended.

Code Experiment: Naive Policy Dynamic Portfolio Algorithm

This method may resemble reinforcement learning (*RL*), but it performs no actual learning. It uses a simple neural network (*PolicyNetwork*) that takes market signals — like short- vs. long-term moving averages — and outputs a probability distribution over possible stock picks and a “no-action” option.

Actions are then selected randomly based on these probabilities using `np.random.choice`.

The process is straightforward:

1. Input market state features.
2. Output softmax probabilities for all stocks and a no-action choice.
3. Sample one action and simulate trades under constraints like cooldowns or max position weights.

For example, if given *AAPL, MSFT*, and *TSLA*, the network might output *[0.4, 0.3, 0.2, 0.1]*, where the last value represents doing nothing. A random action is picked based on this. However, there's no feedback — no matter the outcome, the policy never updates.

Despite lacking a learning loop, this "fake-RL" approach is simple, quick to test, and allows experimentation under realistic trading rules. With smart initialization or light supervised tuning, it can perform reasonably well — without the overhead and instability of full RL methods like actor-critic or *PPO*.

```
import itertools
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import yfinance as yf

########################################
# Existing Code (shortened to focus on pipeline)
########################################
```

```
def getdatabyday(tiklst, st, ed):
    download_data = pd.DataFrame([])
    for tk in tiklst:
        try:
            stock = yf.Ticker(tk)
            stock_data = stock.history(start=st, end=ed, interval='1d').reset_in
            if len(stock_data) > 0:
                stock_data['ticker'] = tk
                download_data = pd.concat([download_data, stock_data])
                print('Downloaded data for:', tk)
            else:
                print(f"No data returned for: {tk}")
        except Exception as e:
            print(f"Skipping {tk} due to error: {e}")
            continue
    # Sort
    download_data = download_data.sort_values(['ticker', 'Date'], ascending=[Tru
    download_data['Date'] = download_data['Date'].astype(str)
    cols_keep = ['ticker', 'Date', 'Open', 'High', 'Low', 'Close', 'Volume']
    download_data = download_data[cols_keep]
    tm_frame = pd.DataFrame(list(set(download_data['Date'])), columns=['Date'])
    tm_frame = tm_frame.sort_values(['Date'], ascending=False)
    tm_frame['dayseq'] = range(1, len(tm_frame) + 1)
    download_data = pd.merge(download_data, tm_frame, on=['Date'], how='inner')
    download_data = download_data.sort_values(['ticker', 'Date'], ascending=[Fal
    download_data.ffill(inplace=True)
    download_data.bfill(inplace=True)
    return download_data

def organize_data_for_rl(stock_data):
    stock_data['Date'] = pd.to_datetime(stock_data['Date'], utc=True).dt.tz_loca
    stock_data.sort_values(by='Date', inplace=True)
    wide_df = stock_data.pivot(index='Date', columns='ticker', values='Close')
    wide_df.columns = [col.upper().replace('-', '_') for col in wide_df.columns]
    wide_df.ffill(inplace=True)
    wide_df.bfill(inplace=True)
    return wide_df

class PolicyNetwork(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return torch.softmax(self.fc3(x), dim=-1)
```

```
def compute_signals(DF, k, n, delta):
    df_ratios = DF.copy()
    stocks = [col for col in DF.columns if col not in ['SPY','QQQ']]
    for stock in stocks:
        df_ratios[stock] = DF[stock] / (DF['SPY'] + DF['QQQ'])

    signals = {}
    for stock in stocks:
        ma_k = df_ratios[stock].rolling(window=k).mean()
        ma_n = df_ratios[stock].rolling(window=n).mean()
        signals[stock] = (ma_k < (ma_n - delta)).astype(int)

    signals_df = pd.DataFrame(signals, index=DF.index)
    return signals_df, df_ratios

def get_state(DF, df_ratios, t, k, n):
    stocks = [col for col in DF.columns if col not in ['SPY','QQQ']]
    state_feats = []
    for stock in stocks:
        ratio_t = df_ratios[stock].iloc[t]
        ma_k_t = df_ratios[stock].rolling(window=k).mean().iloc[t]
        ma_n_t = df_ratios[stock].rolling(window=n).mean().iloc[t]
        state_feats.extend([ratio_t, ma_k_t, ma_n_t])
    return np.array(state_feats, dtype=np.float32)

def simulate_trading(
    DF,
    signals_df,
    df_ratios,
    policy_net,
    start_idx,
    end_idx,
    k, n,
    trade_interval_m=3,
    max_weight_w=0.34,
    cooldown_R=10,
    required_distinct=4
):
    stocks = [col for col in DF.columns if col not in ['SPY', 'QQQ']]
    portfolio = {}
    transactions = []
    last_trade_day = -trade_interval_m
    distinct_stocks = set()

    for t in range(start_idx, end_idx):
        if t - last_trade_day < trade_interval_m:
            continue
        state = get_state(DF, df_ratios, t, k, n)
        state_tensor = torch.from_numpy(state).unsqueeze(0)
        with torch.no_grad():
```

```
        probs = policy_net(state_tensor).squeeze(0).numpy()

    candidate_mask = np.array([signals_df[stock].iloc[t] for stock in stocks
    candidate_mask = np.concatenate([candidate_mask, np.array([1])])  # no-a
    masked_probs = probs[:len(stocks)] * candidate_mask[:len(stocks)]
    no_action_prob = probs[len(stocks)]
    final_probs = np.concatenate([masked_probs, np.array([no_action_prob])])
    if final_probs.sum() > 0:
        final_probs = final_probs / final_probs.sum()
    else:
        final_probs = np.ones_like(final_probs) / len(final_probs)

    action = np.random.choice(len(final_probs), p=final_probs)
    if action == len(stocks):
        continue

    chosen_stock = stocks[action]
    if chosen_stock in portfolio and (t - portfolio[chosen_stock]['last_trad
        continue

    current_value = sum(portfolio[s]['shares'] * DF[s].iloc[t] for s in port
    new_trade_amount = 1000.0
    est_total = current_value + new_trade_amount
    if len(portfolio) >= 2:
        existing_cost = portfolio.get(chosen_stock, {}).get('cost', 0.0)
        new_cost = existing_cost + new_trade_amount
        if (new_cost / est_total) > max_weight_w:
            continue

    trade_price = DF[chosen_stock].iloc[t]
    shares = new_trade_amount / trade_price
    if chosen_stock in portfolio:
        portfolio[chosen_stock]['shares'] += shares
        portfolio[chosen_stock]['cost']   += new_trade_amount
        portfolio[chosen_stock]['last_trade_day'] = t
    else:
        portfolio[chosen_stock] = {'shares': shares, 'cost': new_trade_amoun
        distinct_stocks.add(chosen_stock)

    transactions.append({
        'Date': DF.index[t],
        'Stock': chosen_stock,
        'Trade_Price': trade_price,
        'Shares_Bought': shares,
        'Trade_Amount': new_trade_amount,
        'Cumulative_Cost': portfolio[chosen_stock]['cost'],
        'Avg_Cost': portfolio[chosen_stock]['cost'] / portfolio[chosen_stock
        'Day_Index': t
    })
    last_trade_day = t
```

```
    transactions_df = pd.DataFrame(transactions)
    final_num_distinct = len(distinct_stocks)
    return transactions_df, portfolio, final_num_distinct

def full_pipeline(
    DF,
    policy_net,
    train_end_idx,
    k=5, n=20, delta=0.005,
    trade_interval_m=3,
    max_weight_w=0.34,
    cooldown_R=10,
    required_distinct=4
):
    signals_df, df_ratios = compute_signals(DF, k, n, delta)

    # Train
    train_tx, train_port, train_distinct = simulate_trading(
        DF, signals_df, df_ratios,
        policy_net,
        start_idx=0,
        end_idx=train_end_idx,
        k=k, n=n,
        trade_interval_m=trade_interval_m,
        max_weight_w=max_weight_w,
        cooldown_R=cooldown_R,
        required_distinct=required_distinct
    )

    # Training Return
    train_value = 0.0
    for stk in train_port:
        train_value += train_port[stk]['shares'] * DF[stk].iloc[train_end_idx -
    train_invested = sum(train_port[s]['cost'] for s in train_port)
    train_ret = (train_value / train_invested - 1.0) if train_invested > 0 else

    train_spy_ret = train_qqq_ret = None
    if 'SPY' in DF.columns:
        spy_start = DF['SPY'].iloc[0]
        spy_end   = DF['SPY'].iloc[train_end_idx - 1]
        train_spy_ret = (spy_end / spy_start) - 1.0
    if 'QQQ' in DF.columns:
        qqq_start = DF['QQQ'].iloc[0]
        qqq_end   = DF['QQQ'].iloc[train_end_idx - 1]
        train_qqq_ret = (qqq_end / qqq_start) - 1.0

    # Validation
    val_tx, val_port, val_distinct = simulate_trading(
        DF, signals_df, df_ratios,
```

```
            policy_net,
            start_idx=train_end_idx,
            end_idx=len(DF),
            k=k, n=n,
            trade_interval_m=trade_interval_m,
            max_weight_w=max_weight_w,
            cooldown_R=cooldown_R,
            required_distinct=required_distinct
        )
        val_value = 0.0
        for stk in val_port:
            val_value += val_port[stk]['shares'] * DF[stk].iloc[-1]
        val_invested = sum(val_port[s]['cost'] for s in val_port)
        val_ret = (val_value / val_invested - 1.0) if val_invested > 0 else 0.0

        spy_val_ret = qqq_val_ret = None
        if 'SPY' in DF.columns:
            spy_val_ret = (DF['SPY'].iloc[-1] / DF['SPY'].iloc[train_end_idx]) - 1.0
        if 'QQQ' in DF.columns:
            qqq_val_ret = (DF['QQQ'].iloc[-1] / DF['QQQ'].iloc[train_end_idx]) - 1.0

        return {
            'train_tx': train_tx,
            'train_ret': train_ret,
            'train_spy_ret': train_spy_ret,
            'train_qqq_ret': train_qqq_ret,
            'val_tx': val_tx,
            'val_ret': val_ret,
            'spy_val_ret': spy_val_ret,
            'qqq_val_ret': qqq_val_ret
        }

    import random
    seed_val = 99
    random.seed(seed_val)
    np.random.seed(seed_val)
    torch.manual_seed(seed_val)
    # If using CUDA:
    torch.cuda.manual_seed_all(seed_val)

    ########################################
    # Parameter Search
    ########################################
    if __name__ == "__main__":
        # Tickers, date range
        tickers = [
            "SPY", "QQQ",
            "AAPL", "MSFT", "META", "NVDA", "TSLA",
            "AMZN", "PANW", "CRWD", "IYW", "ISRG"
        ]
```

```
start_date = "2021-01-01"
end_date   = "2024-12-31"
cutoff_date= "2024-02-01"
# Download / prep data
print("Downloading data...")
stock_data = getdatabyday(tickers, start_date, end_date)
DF = organize_data_for_rl(stock_data)

DF.index = pd.to_datetime(DF.index, utc=True).tz_localize(None)
DF.sort_index(inplace=True)

all_dates = DF.index
train_end_idx = np.searchsorted(all_dates, pd.Timestamp(cutoff_date))
print(f"train_end_idx = {train_end_idx}, date = {all_dates[train_end_idx]}")

num_cols = len(DF.columns)
input_dim = 3*(num_cols - 2)
output_dim= (num_cols - 2)+1

# Let's define a small grid
k_values       = [5, 8, 10]
n_values       = [20, 25, 30]
delta_values   = [0.005, 0.01, 0.02]
trade_intervals= [3, 4, 5]
cooldown_vals  = [10, 15]
weight_limits  = [0.25, 0.34]
required_min   = [4, 5]

results = []
# Each combination
for k_v, n_v, d_v, ti_v, cd_v, w_v, req_v in itertools.product(
    k_values, n_values, delta_values, trade_intervals, cooldown_vals, weight
):
    # create fresh policy net & optimizer each time
    policy_net = PolicyNetwork(input_dim, output_dim)
    optimizer = optim.Adam(policy_net.parameters(), lr=0.0006)

    # run pipeline
    outcome = full_pipeline(
        DF,
        policy_net,
        train_end_idx,
        k=k_v,
        n=n_v,
        delta=d_v,
        trade_interval_m=ti_v,
        max_weight_w=w_v,
        cooldown_R=cd_v,
        required_distinct=req_v
    )
```

```
            train_ret  = outcome['train_ret']
            train_spy  = outcome['train_spy_ret']
            train_qqq  = outcome['train_qqq_ret']
            val_ret    = outcome['val_ret']
            spy_val_ret= outcome['spy_val_ret']
            qqq_val_ret= outcome['qqq_val_ret']

            # store
            results.append({
                'k': k_v,
                'n': n_v,
                'delta': d_v,
                'trade_interval': ti_v,
                'cooldown_R': cd_v,
                'max_weight': w_v,
                'required_distinct': req_v,
                'train_return': train_ret,
                'train_spy': train_spy,
                'train_qqq': train_qqq,
                'val_return': val_ret,
                'val_spy': spy_val_ret,
                'val_qqq': qqq_val_ret
            })

    # Convert results to DataFrame
    df_results = pd.DataFrame(results)

    # For example, let's find combos that do better than QQQ in validation
    # We'll define "excess_return_qqq_val" as val_return - val_qqq
    df_results['excess_return_qqq_val'] = df_results['val_return'] - df_results[
    # Sort by best excess over QQQ in validation
    df_results.sort_values('excess_return_qqq_val', ascending=False, inplace=Tru

    print("\n=== TOP 10 PARAM SETS (ValReturn vs QQQ) ===")
    print(df_results.head(10).to_string(index=False))
```

Here is a summary of the results:

```
Downloading data...
Downloaded data for: SPY
Downloaded data for: QQQ
Downloaded data for: AAPL
Downloaded data for: MSFT
```

```
Downloaded data for: META
Downloaded data for: NVDA
Downloaded data for: TSLA
Downloaded data for: AMZN
Downloaded data for: PANW
Downloaded data for: CRWD
Downloaded data for: IYW
Downloaded data for: ISRG
train_end_idx = 774, date = 2024-02-01 05:00:00

=== TOP 10 PARAM SETS (ValReturn vs QQQ) ===
 k  n  delta  trade_interval  cooldown_R  max_weight  required_distinct  train_r
 8 20   0.02              4          10        0.25                  5      0.6
10 30   0.02              3          15        0.25                  4      0.4
 5 20   0.02              3          15        0.25                  5     -0.1
 5 20   0.02              5          15        0.25                  4      0.0
10 30   0.02              4          10        0.25                  5      0.2
10 30   0.02              4          15        0.25                  5      0.1
10 30   0.02              3          15        0.25                  5      0.6
10 20   0.02              3          15        0.25                  5      0.4
10 25   0.02              5          10        0.25                  4      0.7
 5 25   0.02              5          15        0.25                  5      0.4
```

Summary of Results

The top-performing configuration in the Naive Policy Dynamic Portfolio Algorithm strongly outperformed the *QQQ* benchmark, delivering a validation return of around 84.18%, surpassing *QQQ* (*~22.94%*) by roughly 61.24 percentage points. This result was achieved with the parameters:

k=8, n=20, delta=0.02, trade_interval=4, cooldown_R=10, max_weight=0.25, required_distinct=5

This strategy also showed consistent performance, yielding a strong training return (*~61.17%*) and significantly outperforming *QQQ* during the training period. The use of a higher threshold (delta=*0.02*) effectively filtered out minor price fluctuations, generating fewer but more reliable trading signals. Additionally, a moderate trade interval (trade_interval=4) combined with a

brief cooldown (*cooldown_R =10*) struck a good balance between trading frequency and risk management.

However, although this Naive Policy Dynamic Portfolio Algorithm outperformed the previously discussed actor-critic RL method, it doesn't necessarily indicate that the naive policy approach is universally superior. Performance can vary significantly depending on market conditions, asset volatility, transaction costs, and regime shifts in the market. Therefore, careful consideration and testing under different scenarios and market environments are advised before concluding the absolute superiority of one method over the other.

Conclusion

I explored a practical and innovative way to build a trading portfolio using reinforcement learning (*RL*) combined with dynamic programming. By using simple yet effective state-driven decisions — particularly price ratios and moving averages — our RL model learned optimal trading signals from historical data. The method isn't just theoretical; it's easy to implement and highly practical for real-world portfolio management.

However, picking the right group of stocks is important. Not every stock works equally well with this method. Future research should carefully look at how different stock selections — considering market cap, volatility, industry, or liquidity — can significantly change overall performance. Identifying and narrowing down the most responsive stock universe will greatly enhance results.

Timing is another factor we can't ignore. This model may behave differently across market cycles like bullish trends, bearish downturns, or sideways markets. Understanding when and under what market conditions the model performs best will add huge practical value, allowing traders to effectively leverage the method at the right moments.

Lastly, the current method mainly uses price ratios for state inputs. But I strongly believe there's plenty of room to improve by integrating more informative states. Combining multiple factors — such as economic indicators, technical signals, or sentiment analysis — into joint states with flexible parameters could boost accuracy significantly. Exploring these multi-factor states is a promising direction for future studies.

This initial work has set the stage for further advancements. Stock selection, market timing, and multi-dimensional states all present valuable areas for improvement and testing, potentially transforming this method into an even stronger tool for real investment decisions.

About me

With over 20 years of experience in software and database management and 25 years teaching IT, math, and statistics, I am a Data Scientist with extensive expertise across multiple industries.

You can connect with me at:

Email: datalev@gmail.com | LinkedIn | X/Twitter

Reinforcement Learning | Stock Market | Dynamic Programming | AI | Data Science

Published in Data Science Collective

835K Followers · Last published 14 hours ago

Follow

Advice, insights, and ideas from the Medium data science community

Written by Shenggang Li

2.3K Followers · 77 Following

Subscribed

Responses (1)

Alex Mylnikov

What are your thoughts?

R. Thompson (PhD) he/him

4 days ago

portfolio.optimize() // $1000 invested, profit activated 🚀

Reply

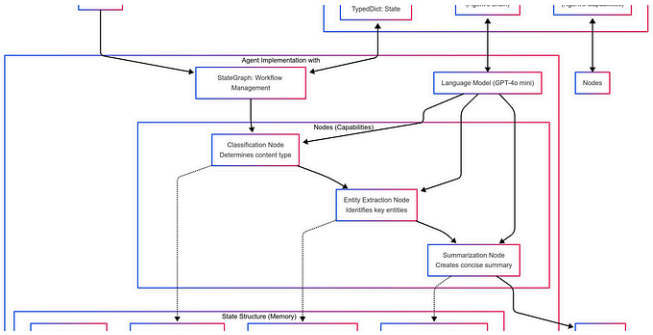
More from Shenggang Li and Data Science Collective

In Towards AI by Shenggang Li

Reinforcement Learning for Business Optimization: A Genetic...

Applying PPO and Genetic Algorithms to Dynamic Pricing in Competitive Markets

Mar 17 187 3



In Data Science Collective by Paolo Perrone

The Complete Guide to Building Your First AI Agent with...

Three months into building my first commercial AI agent, everything collapsed...

Mar 11 2.8K 56

In Data Science Collective by Buse Şenol

Model Context Protocol (MCP): An End-To-End Tutorial With Hands-...

What is MCP? How to create an MPC Server that brings news from a web site with Claude...

Mar 18 1.1K 12

In Towards AI by Shenggang Li

Reinforcement Learning-Enhanced Gradient Boosting Machines

A Novel Approach to Integrating Reinforcement Learning within Gradient...

Apr 1 276 3

See all from Shenggang Li

See all from Data Science Collective

Recommended from Medium

In Towards AI by Shenggang Li

Reinforcement Learning-Enhanced Gradient Boosting Machines

A Novel Approach to Integrating Reinforcement Learning within Gradient...

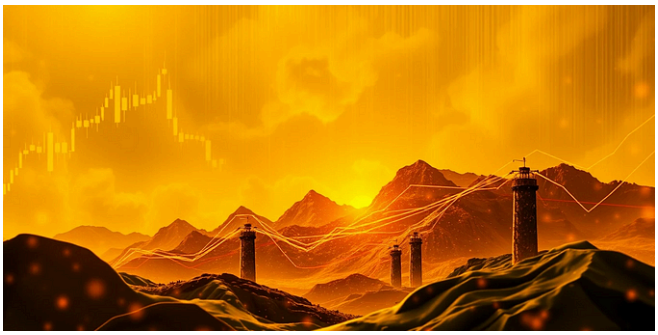
Apr 1 276 3

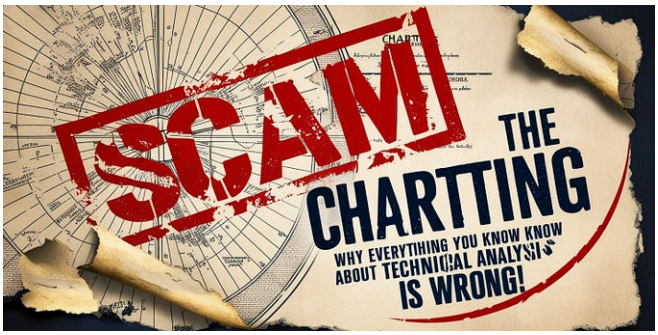
In Data Science Collecti... by Bradley Stephen Sh...

Teach Your GBM to Extrapolate with Model Stacking

Background

5d ago 170 4





Unicorn Day

Decoding the Market with Lorentzian Distance: A Python...

What if you could spot the market's wild swings before they happen... and trade...

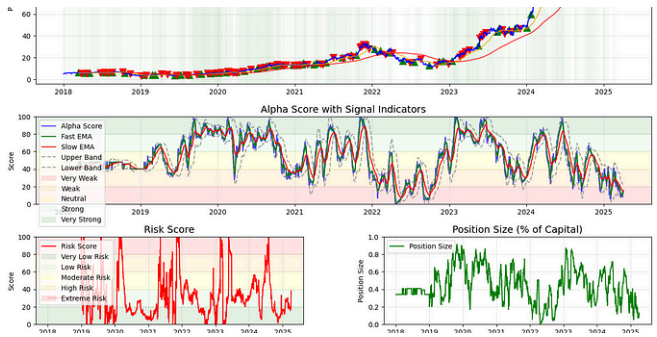
Apr 2 135 3

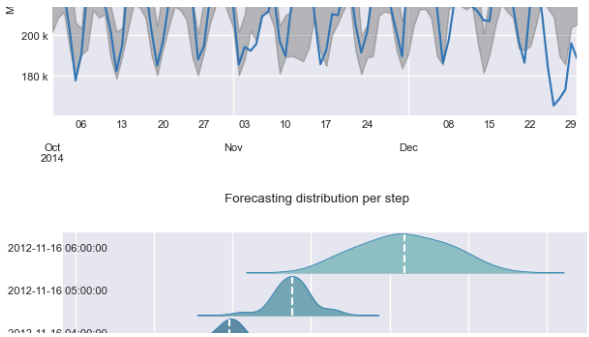
In InsiderFinance Wire by Nayab Bhutta

The Charting SCAM: Why Everything You Know About...

For decades, traders have relied on technical analysis (TA) to predict market movements....

6d ago 351 7





Nicolae Filip

NVDA Momentum-Volatility-Quality Trading Strategy: A...

Introduction

In Coding Nexus by Code Pulse

Probabilistic Forecasting with skforecast: A Python Adventure

Okay, so I've been messing with time series stuff in Python for ages—ten years if I'm...

4d ago 12 2 Apr 2 5

See more recommendations